otpod Documentation

Release

Antoine Dumas

Jul 07, 2017

otpod is a module for OpenTURNS.

CHAPTER

ONE

CONTENTS:

1.1 Documentation of the API

This is the user manual for the Python bindings to the otpod library.

1.1.1 Data analysis

UnivariateLinearModelAnalysis	Linear regression analysis with residuals hypothesis tests.

UnivariateLinearModelAnalysis

class **UnivariateLinearModelAnalysis** (**args*)

Linear regression analysis with residuals hypothesis tests.

Available constructors:

UnivariateLinearModelAnalysis(*inputSample, outputSample*)

UnivariateLinearModelAnalysis(*inputSample, outputSample, noiseThres, saturationThres, resDistFact, boxCox*)

Parameters **inputSample** : 2-d sequence of float

Vector of the defect sizes, of dimension 1.

outputSample : 2-d sequence of float

Vector of the signals, of dimension 1.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None.

resDistFact : `openturns.DistributionFactory`

Distribution hypothesis followed by the residuals. Default is `openturns.NormalFactory`.

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

Notes

This method automatically :

- computes the Box Cox parameter if *boxCox* is True,
- computes the transformed signals if *boxCox* is True or a float,
- builds the univariate linear regression model on the data,
- computes the linear regression parameters for censored data if needed,
- computes the residuals,
- runs all hypothesis tests.

Examples

Generate data :

```
>>> import openturns as ot
>>> import otpod
>>> N = 100
>>> ot.RandomGenerator.SetSeed(0)
>>> defectDist = ot.Uniform(0.1, 0.6)
>>> epsilon = ot.Normal(0, 1.9)
>>> defects = defectDist.getSample(N)
>>> signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
>>> invBoxCox = ot.InverseBoxCoxTransform(0.3)
>>> signals = invBoxCox(signalsInvBoxCox)
```

Run analysis with gaussian hypothesis on the residuals :

```
>>> analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, boxCox=True)
>>> print analysis.getIntercept() # get intercept value
[Intercept for uncensored case : 2.51037]
>>> print analysis.getKolmogorovPValue()
[Kolmogorov p-value for uncensored case : 0.835529]
```

Run analysis with noise and saturation threshold :

```
>>> analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, 60., 1700.,
↪boxCox=True)
>>> print analysis.getIntercept() # get intercept value for uncensored and
↪censored case
[Intercept for uncensored case : 4.28758, Intercept for censored case : 3.11243]
>>> print analysis.getKolmogorovPValue()
[Kolmogorov p-value for uncensored case : 0.346827, Kolmogorov p-value for
↪censored case : 0.885006]
```

Run analysis with a Weibull distribution hypothesis on the residuals

```
>>> analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, 60., 1700.,
↪ot.WeibullFactory(), boxCox=True)
>>> print analysis.getIntercept() # get intercept value for uncensored and
↪censored case
[Intercept for uncensored case : 4.28758, Intercept for censored case : 3.11243]
>>> print analysis.getKolmogorovPValue()
[Kolmogorov p-value for uncensored case : 0.476036, Kolmogorov p-value for
↪censored case : 0.71764]
```

Methods

drawBoxCoxLikelihood([name])	Draw the loglikelihood versus the Box Cox parameter.
drawLinearModel([model, name])	Draw the linear regression prediction versus the true data.
drawResiduals([model, name])	Draw the residuals versus the defect values.
drawResidualsDistribution([model, name])	Draw the residuals histogram with the fitted distribution.
drawResidualsQQplot([model, name])	Draw the residuals QQ plot with the fitted distribution.
getAndersonDarlingPValue()	Accessor to the Anderson Darling test p-value.
getBoxCoxParameter()	Accessor to the Box Cox parameter.
getBreuschPaganPValue()	Accessor to the Breusch Pagan test p-value.
getCramerVonMisesPValue()	Accessor to the Cramer Von Mises test p-value.
getDurbinWatsonPValue()	Accessor to the Durbin Watson test p-value.
getHarrisonMcCabePValue()	Accessor to the Harrison McCabe test p-value.
getInputSample()	Accessor to the input sample.
getIntercept()	Accessor to the intercept of the linear regression model.
getKolmogorovPValue()	Accessor to the Kolmogorov test p-value.
getNoiseThreshold()	Accessor to the noise threshold.
getOutputSample()	Accessor to the output sample.
getR2()	Accessor to the R2 value.
getResiduals()	Accessor to the residuals.
getResidualsDistribution()	Accessor to the residuals distribution.
getResults()	Print results of the linear analysis.
getSaturationThreshold()	Accessor to the saturation threshold.
getSlope()	Accessor to the slope of the linear regression model.
getStandardError()	Accessor to the standard error of the estimate.
getZeroMeanPValue()	Accessor to the Zero Mean test p-value.
saveResults(name)	Save all analysis test results in a file.

drawBoxCoxLikelihood(*name=None*)

Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

drawLinearModel(*model='uncensored'*, *name=None*)
Draw the linear regression prediction versus the true data.

Parameters **model** : string

The linear regression model to be used, either *uncensored* or *censored* if censored threshold were given. Default is *uncensored*.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawResiduals(*model='uncensored'*, *name=None*)
Draw the residuals versus the defect values.

Parameters **model** : string

The residuals to be used, either *uncensored* or *censored* if censored threshold were given. Default is *uncensored*.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawResidualsDistribution(*model='uncensored'*, *name=None*)
Draw the residuals histogram with the fitted distribution.

Parameters **model** : string

The residuals to be used, either *uncensored* or *censored* if censored threshold were given. Default is *uncensored*.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawResidualsQQplot(*model='uncensored'*, *name=None*)
Draw the residuals QQ plot with the fitted distribution.

Parameters **model** : string

The residuals to be used, either *uncensored* or *censored* if censored threshold were given. Default is *uncensored*.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getAndersonDarlingPValue()
Accessor to the Anderson Darling test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

getBoxCoxParameter()
Accessor to the Box Cox parameter.

Returns **lambdaBoxCox** : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getBreuschPaganPValue()
Accessor to the Breusch Pagan test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

getCramerVonMisesPValue()
Accessor to the Cramer Von Mises test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

getDurbinWatsonPValue()
Accessor to the Durbin Watson test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

getHarrisonMcCabePValue()
Accessor to the Harrison McCabe test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

getInputSample()
Accessor to the input sample.

Returns **defects** : `openturns.NumericalSample`

The input sample which is the defect values.

getIntercept()

Accessor to the intercept of the linear regression model.

Returns **intercept** : `openturns.NumericalPoint`

The intercept parameter for the uncensored and censored (if so) linear regression model.

getKolmogorovPValue()

Accessor to the Kolmogorov test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

getNoiseThreshold()

Accessor to the noise threshold.

Returns **noiseThres** : float

The noise threhold if it exists, if not it returns *None*.

getOutputSample()

Accessor to the output sample.

Returns **signals** : `openturns.NumericalSample`

The input sample which is the signal values.

getR2()

Accessor to the R2 value.

Returns **R2** : `openturns.NumericalPoint`

Either the R2 for the uncensored case or for both cases.

getResiduals()

Accessor to the residuals.

Returns **residuals** : `openturns.NumericalSample`

The residuals computed from the uncensored and censored linear regression model. The first column corresponds with the uncensored case.

getResidualsDistribution()

Accessor to the residuals distribution.

Returns **distribution** : list of `openturns.Distribution`

The fitted distribution on the residuals, computed in the uncensored and censored (if so) case.

getResults()

Print results of the linear analysis.

getSaturationThreshold()

Accessor to the saturation threshold.

Returns **saturationThres** : float

The saturation threhold if it exists, if not it returns *None*.

getSlope()

Accessor to the slope of the linear regression model.

Returns **slope** : `openturns.NumericalPoint`

The slope parameter for the uncensored and censored (if so) linear regression model.

getStandardError()

Accessor to the standard error of the estimate.

Returns **stderr** : `openturns.NumericalPoint`

The standard error of the estimate for the uncensored and censored (if so) linear regression model.

getZeroMeanPValue()

Accessor to the Zero Mean test p-value.

Returns **pValue** : `openturns.NumericalPoint`

Either the p-value for the uncensored case or for both cases.

saveResults(*name*)

Save all analysis test results in a file.

Parameters **name** : string

Name of the file or full path name.

Notes

The file can be saved as a csv file. Separations are made with tabulations.

If *name* is the file name, then it is saved in the current working directory.

1.1.2 POD computation methods

UnivariateLinearModelPOD	Linear regression based POD.
QuantileRegressionPOD	Quantile regression based POD.
PolynomialChaosPOD	Polynomial chaos based POD.
KrigingPOD	Kriging based POD.

UnivariateLinearModelPOD

class **UnivariateLinearModelPOD**(**args*)

Linear regression based POD.

Available constructors:

UnivariateLinearModelPOD(*analysis=analysis, detection=detection*)

UnivariateLinearModelPOD(*inputSample, outputSample, detection, noiseThres, saturationThres, resDistFact, boxCox*)

Parameters **analysis** : *UnivariateLinearModelAnalysis*

Linear analysis object.

inputSample : 2-d sequence of float

Vector of the defect sizes, of dimension 1.

outputSample : 2-d sequence of float

Vector of the signals, of dimension 1.

detection : float

Detection value of the signal.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

resDistFact : `openturns.DistributionFactory`

Distribution hypothesis followed by the residuals. Default is None.

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

Notes

This class aims at building the POD based on a linear regression model. If a linear analysis has been launched, it can be used as prescribed in the first constructor. It can be noticed that, in this case, with the default parameters of the linear analysis, the POD will corresponds with the linear regression model associated to a Gaussian hypothesis on the residuals.

Otherwise, all parameters can be given as in the second constructor.

Following the given distribution in *resDistFact*, the POD model is built different hypothesis:

- if *resDistFact* = *None*, it corresponds with Berens-Binomial. This is the default case.
- if *resDistFact* = `openturns.NormalFactory`, it corresponds with Berens-Gauss.
- if *resDistFact* = {`openturns.KernelSmoothing`, `openturns.WeibullFactory`, ...}, the confidence interval is built by bootstrap.

If bootstrap is used, a progress bar is shown if the verbosity is enabled. It can be disabled using the method *setVerbose*.

Methods

`computeDetectionSize`(*args, **kwargs)	Compute the detection size for a given probability level.
`drawBoxCoxLikelihood`([name])	Draw the loglikelihood versus the Box Cox parameter.
`drawPOD`(*args, **kwargs)	Draw the POD curve.
`getBoxCoxParameter`()	Accessor to the Box Cox parameter.
`getPODCLModel`([confidenceLevel])	Accessor to the POD model at a given confidence level.
`getPODModel`()	Accessor to the POD model.
`getR2`()	Accessor to the R2 value.
`getSimulationSize`()	Accessor to the simulation size.
`getVerbose`()	Accessor to the verbosity.
`run`()	Build the POD models.
`setSimulationSize`(size)	Accessor to the simulation size.
`setVerbose`(verbose)	Accessor to the verbosity.

computeDetectionSize(**args*, ***kwargs*)
Compute the detection size for a given probability level.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

Returns **result** : collection of `openturns.NumericalPointWithDescription`

A NumericalPointWithDescription containing the detection size computed at the given probability level and confidence level if provided.

drawBoxCoxLikelihood(*name=None*)
Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

drawPOD(**args*, ***kwargs*)
Draw the POD curve.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

defectMin, defectMax : float

Define the interval where the curve is plotted. Default : min and max values of the input sample.

nbPt : int

The number of points to draw the curves. Default is 100.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getBoxCoxParameter()
Accessor to the Box Cox parameter.

Returns **lambdaBoxCox** : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getPODCLModel(*confidenceLevel=0.95*)
Accessor to the POD model at a given confidence level.

Parameters **confidenceLevel** : float

The confidence level the POD must be computed. Default is 0.95

Returns **PODModelCl** : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value at the confidence level given as parameter.

getPODModel()
Accessor to the POD model.

Returns **PODModel** : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value.

getR2()
Accessor to the R2 value.

Returns **R2** : float

The R2 value.

getSimulationSize()
Accessor to the simulation size.

Returns **size** : int

The size of the simulation used to compute the confidence interval.

getVerbose()
Accessor to the verbosity.

Returns **verbose** : bool

Enable or disable the verbosity. Default is True.

run()
Build the POD models.

Notes

This method build the linear model for the uncensored or censored case depending of the input parameters. Then it builds the POD model following the given residuals distribution factory.

setSimulationSize(*size*)
Accessor to the simulation size.

Parameters **size** : int

The size of the simulation used to compute the confidence interval.

setVerbose(*verbose*)
Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

QuantileRegressionPOD

class QuantileRegressionPOD(**args*)
Quantile regression based POD.

Available constructor:

QuantileRegressionPOD(*inputSample, outputSample, detection, noiseThres, saturationThres, boxCox*)

Parameters **inputSample** : 2-d sequence of float

Vector of the defect sizes, of dimension 1.

outputSample : 2-d sequence of float

Vector of the signals, of dimension 1.

detection : float

Detection value of the signal.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

Notes

This class aims at building the POD based on a quantile regression model. The return POD model corresponds with an interpolate function built with the defect values computed for the given quantile as parameters. The default is 21 quantile values from 0.05 to 0.98. They can be user-defined using the method *setQuantile*.

The confidence level is computed by bootstrap. The POD model at the given confidence level is also an interpolate function based on the defect quantile value computed at the given confidence level.

The computeDetectionSize method calls the real quantile regression at the given probability level.

A progress bar is shown if the verbosity is enabled. It can be disabled using the method *setVerbose*.

Methods

computeDetectionSize(*args, **kwargs)	Compute the detection size for a given probability level.
drawBoxCoxLikelihood([name])	Draw the loglikelihood versus the Box Cox parameter.
drawLinearModel(probabilityLevel[, name])	Draw the quantile regression prediction versus the true data.
drawPOD(*args, **kwargs)	Draw the POD curve.
getBoxCoxParameter()	Accessor to the Box Cox parameter.
getPODCLModel([confidenceLevel])	Accessor to the POD model at a given confidence level.
getPODModel()	Accessor to the POD model.
getQuantile()	Accessor to the quantile list for the regression.
getR2(quantile)	Accessor to the pseudo R2 value.
getSimulationSize()	Accessor to the simulation size.
getVerbose()	Accessor to the verbosity.
run()	Build the POD models.
setQuantile(quantile)	Accessor to the quantile list for the regression.
setSimulationSize(size)	Accessor to the simulation size.
setVerbose(verbose)	Accessor to the verbosity.

computeDetectionSize(**args*, ***kwargs*)

Compute the detection size for a given probability level.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

Returns **result** : collection of `openturns.NumericalPointWithDescription`

A NumericalPointWithDescription containing the detection size computed at the given probability level and confidence level if provided.

drawBoxCoxLikelihood(*name=None*)

Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

drawLinearModel(*probabilityLevel*, *name=None*)

Draw the quantile regression prediction versus the true data.

Parameters **probabilityLevel** : float

The probability level for which the quantile regression is performed

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawPOD(**args*, ***kwargs*)
Draw the POD curve.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

defectMin, defectMax : float

Define the interval where the curve is plotted. Default : min and max values of the input sample.

nbPt : int

The number of points to draw the curves. Default is 100.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getBoxCoxParameter()
Accessor to the Box Cox parameter.

Returns **lambdaBoxCox** : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getPODCLModel(*confidenceLevel=0.95*)
Accessor to the POD model at a given confidence level.

Parameters **confidenceLevel** : float

The confidence level the POD must be computed. Default is 0.95

Returns PODModelCl : openturns.NumericalMathFunction

The function which computes the probability of detection for a given defect value at the confidence level given as parameter.

getPODModel()

Accessor to the POD model.

Returns PODModel : openturns.NumericalMathFunction

The function which computes the probability of detection for a given defect value.

getQuantile()

Accessor to the quantile list for the regression.

getR2(*quantile*)

Accessor to the pseudo R2 value.

Parameters quantile : float

The quantile value for which the regression is performed.

Returns R2 : float

The pseudo R2 value.

getSimulationSize()

Accessor to the simulation size.

Returns size : int

The size of the simulation used to compute the confidence interval.

getVerbose()

Accessor to the verbosity.

Returns verbose : bool

Enable or disable the verbosity. Default is True.

run()

Build the POD models.

Notes

This method build the quantile regression model. First the censored data are filtered if needed. The Box Cox transformation is performed if it is enabled. Then it builds the POD model for given data and computes using bootstrap all the defects quantile needed to build the POD model at the confidence level.

setQuantile(*quantile*)

Accessor to the quantile list for the regression.

Parameters quantile : sequence of float

The quantile value for which the regression is performed and the corresponding defect size is computed.

setSimulationSize(*size*)

Accessor to the simulation size.

Parameters size : int

The size of the simulation used to compute the confidence interval.

setVerbose (*verbose*)
Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

PolynomialChaosPOD

class **PolynomialChaosPOD** (**args*)
Polynomial chaos based POD.

Available constructor:

PolynomialChaosPOD(*inputSample, outputSample, detection, noiseThres, saturationThres, boxCox*)

Parameters **inputSample** : 2-d sequence of float

Vector of the input values. The first column must correspond with the defect sizes.

outputSample : 2-d sequence of float

Vector of the signals, of dimension 1.

detection : float

Detection value of the signal.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

Warning: The first column of the input sample must corresponds with the defects sample.

Notes

This class aims at building the POD based on a polynomial chaos model. This method must be used under the assumption that the residuals follows a Normal distribution.

The return POD model corresponds with an interpolate function built with the POD values computed for the given defect sizes. The default values are 20 defect sizes between the minimum and maximum value of the defect sample. The defect sizes can be changed using the method *setDefectSizes*.

The default polynomial chaos model is built with uniform distributions for each parameters. Coefficients are computed using the LAR algorithm combined with the KFold. The AdaptiveStrategy is chosen fixed with a linear enumerate function of maximum degree 3.

For advanced use, all parameters can be defined thanks to dedicated set methods. Moreover, if the user has already built a polynomial chaos result, it can be given as parameter using the method *setPolynomialChaosResult*, then the POD is computed based on this polynomial chaos result.

A progress bar is shown if the verbosity is enabled. It can be disabled using the method *setVerbose*.

Methods

computeDetectionSize(*args, **kwargs)	Compute the detection size for a given probability level.
drawBoxCoxLikelihood([name])	Draw the loglikelihood versus the Box Cox parameter.
drawPOD(*args, **kwargs)	Draw the POD curve.
drawPolynomialChaosModel([name])	Draw the polynomial chaos prediction versus the true data.
drawValidationGraph(*args, **kwargs)	Draw the validation graph of the metamodel.
getAdaptiveStrategy()	Accessor to the adaptive strategy.
getBoxCoxParameter()	Accessor to the Box Cox parameter.
getCoefficientDistribution()	Accessor to the distribution of the polynomial chaos coefficients.
getDefectSizes()	Accessor to the defect size where POD is computed.
getDegree()	Accessor to the polynomial chaos degree.
getDistribution()	Accessor to the parameters distribution.
getPODCLModel([confidenceLevel])	Accessor to the POD model at a given confidence level.
getPODModel()	Accessor to the POD model.
getPolynomialChaosResult()	Accessor to the polynomial chaos result.
getProjectionStrategy()	Accessor to the projection strategy.
getQ2()	Accessor to the Q2 value.
getR2()	Accessor to the R2 value.
getSamplingSize()	Accessor to the Monte Carlo sampling size.
getSimulationSize()	Accessor to the simulation size.
getVerbose()	Accessor to the verbosity.
run()	Build the POD models.
setAdaptiveStrategy(strategy)	Accessor to the adaptive strategy.
setDefectSizes(size)	Accessor to the defect size where POD is computed.
setDegree(degree)	Accessor to the polynomial chaos degree.
setDistribution(distribution)	Accessor to the parameters distribution.
setPolynomialChaosResult(chaosResult)	Accessor to the polynomial chaos result.
setProjectionStrategy(strategy)	Accessor to the projection strategy.
setSamplingSize(size)	Accessor to the Monte Carlo sampling size.
setSimulationSize(size)	Accessor to the simulation size.
setVerbose(verbose)	Accessor to the verbosity.

computeDetectionSize(**args*, ***kwargs*)

Compute the detection size for a given probability level.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

Returns **result** : collection of `openturns.NumericalPointWithDescription`

A NumericalPointWithDescription containing the detection size computed at the given probability level and confidence level if provided.

drawBoxCoxLikelihood(*name=None*)

Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

drawPOD(**args*, ***kwargs*)

Draw the POD curve.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

defectMin, defectMax : float

Define the interval where the curve is plotted. Default : min and max values of the input sample.

nbPt : int

The number of points to draw the curves. Default is 100.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawPolynomialChaosModel(*name=None*)

Draw the polynomial chaos prediction versus the true data.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method only works if the dimension of the input sample is 1.

drawValidationGraph(**args*, ***kwargs*)

Draw the validation graph of the metamodel.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getAdaptiveStrategy()

Accessor to the adaptive strategy.

Returns **strategy** : `openturns.AdaptiveStrategy`

The adaptive strategy for the polynomial chaos.

getBoxCoxParameter()

Accessor to the Box Cox parameter.

Returns **lambdaBoxCox** : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getCoefficientDistribution()

Accessor to the distribution of the polynomial chaos coefficients.

Returns **dist** : `openturns.Distribution`

The distribution of the coefficients.

getDefectSizes()

Accessor to the defect size where POD is computed.

Returns **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDegree()

Accessor to the polynomial chaos degree.

Returns **degree** : int

The degree of the polynomial chaos.

getDistribution()

Accessor to the parameters distribution.

Returns **distribution** : `openturns.ComposedDistribution`

The input parameters distribution, default is a Uniform distribution for all parameters.

getPODCLModel(*confidenceLevel=0.95*)
Accessor to the POD model at a given confidence level.

Parameters confidenceLevel : float

The confidence level the POD must be computed. Default is 0.95

Returns PODModelCl : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value at the confidence level given as parameter.

getPODModel()
Accessor to the POD model.

Returns PODModel : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value.

getPolynomialChaosResult()
Accessor to the polynomial chaos result.

Returns result : `openturns.FunctionalChaosResult`

The polynomial chaos result.

getProjectionStrategy()
Accessor to the projection strategy.

Returns strategy : `openturns.ProjectionStrategy`

The projection strategy for the polynomial chaos.

getQ2()
Accessor to the Q2 value.

Returns Q2 : float

The Q2 value computed analytically.

getR2()
Accessor to the R2 value.

Returns R2 : float

The R2 value.

getSamplingSize()
Accessor to the Monte Carlo sampling size.

Returns size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

getSimulationSize()
Accessor to the simulation size.

Returns size : int

The size of the simulation used to compute the confidence interval.

getVerbose()
Accessor to the verbosity.

Returns verbose : bool

Enable or disable the verbosity. Default is True.

run()
Build the POD models.

Notes

This method build the polynomial chaos model. First the censored data are filtered if needed. The Box Cox transformation is performed if it is enabled. Then it builds the POD models, the Monte Carlo simulation is performed for each given defect sizes. The confidence interval is computed by simulating new coefficients of the polynomial chaos, then Monte Carlo simulations are performed.

setAdaptiveStrategy(*strategy*)
Accessor to the adaptive strategy.

Parameters **strategy** : `openturns.AdaptiveStrategy`

The adaptive strategy for the polynomial chaos.

setDefectSizes(*size*)
Accessor to the defect size where POD is computed.

Parameters **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDegree(*degree*)
Accessor to the polynomial chaos degree.

Parameters **degree** : int

The degree of the polynomial chaos.

setDistribution(*distribution*)
Accessor to the parameters distribution.

Parameters **distribution** : `openturns.ComposedDistribution`

The input parameters distribution.

setPolynomialChaosResult(*chaosResult*)
Accessor to the polynomial chaos result.

Parameters **chaosResult** : `openturns.FunctionalChaosResult`

The polynomial chaos result.

setProjectionStrategy(*strategy*)
Accessor to the projection strategy.

Parameters **strategy** : `openturns.ProjectionStrategy`

The projection strategy for the polynomial chaos.

setSamplingSize(*size*)
Accessor to the Monte Carlo sampling size.

Parameters **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

setSimulationSize(*size*)
Accessor to the simulation size.

Parameters **size** : int

The size of the simulation used to compute the confidence interval.

setVerbose (*verbose*)
Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

KrigingPOD

class **KrigingPOD** (**args*)
Kriging based POD.

Available constructor:

KrigingPOD(*inputSample, outputSample, detection, noiseThres, saturationThres, boxCox*)

Parameters **inputSample** : 2-d sequence of float

Vector of the input values. The first column must correspond with the defect sizes.

outputSample : 2-d sequence of float

Vector of the signals, of dimension 1.

detection : float

Detection value of the signal.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

Warning: The first column of the input sample must corresponds with the defects sample.

Notes

This class aims at building the POD based on a kriging model. No assumptions are required for the residuals with this method. The POD are computed by simulating conditional prediction. For each, a Monte Carlo simulation is performed. The accuracy of the Monte Carlo simulation is taken into account using the TCL.

The return POD model corresponds with an interpolate function built with the POD values computed for the given defect sizes. The default values are 20 defect sizes between the minimum and maximum value of the defect sample. The defect sizes can be changed using the method *setDefectSizes*.

The default kriging model is built with a linear basis only for the defect size and constant otherwise. The covariance model is an anisotropic squared exponential model. Parameters are estimated using the TNC algorithm, the initial starting point of the TNC is found thanks to a quasi random search of the best loglikelihood value among 1000 computations.

For advanced use, all parameters can be defined thanks to dedicated set methods. Moreover, if the user has already built a kriging result, it can be given as parameter using the method *setKrigingResult*, then the POD is computed based on this kriging result.

A progress bar is shown if the verbosity is enabled. It can be disabled using the method *setVerbose*.

Methods

`computeDetectionSize`(*args, **kwargs)	Compute the detection size for a given probability level.
`drawBoxCoxLikelihood`([name])	Draw the loglikelihood versus the Box Cox parameter.
`drawPOD`(*args, **kwargs)	Draw the POD curve.
`drawValidationGraph`(*args, **kwargs)	Draw the validation graph of the metamodel.
`getBasis`()	Accessor to the kriging basis.
`getBoxCoxParameter`()	Accessor to the Box Cox parameter.
`getCovarianceModel`()	Accessor to the kriging covariance model.
`getDefectSizes`()	Accessor to the defect size where POD is computed.
`getDistribution`()	Accessor to the parameters distribution.
`getInitialStartSize`()	Accessor to the initial random search size.
`getKrigingResult`()	Accessor to the kriging result.
`getPODCLModel`([confidenceLevel])	Accessor to the POD model at a given confidence level.
`getPODModel`()	Accessor to the POD model.
`getQ2`()	Accessor to the Q2 value.
`getSamplingSize`()	Accessor to the Monte Carlo sampling size.
`getSimulationSize`()	Accessor to the simulation size.
`getVerbose`()	Accessor to the verbosity.
`run`()	Build the POD models.
`setBasis`(basis)	Accessor to the kriging basis.
`setCovarianceModel`(covarianceModel)	Accessor to the kriging covariance model.
`setDefectSizes`(size)	Accessor to the defect size where POD is computed.
`setDistribution`(distribution)	Accessor to the parameters distribution.
`setInitialStartSize`(size)	Accessor to the initial random search size.
`setKrigingResult`(result)	Accessor to the kriging result.
`setSamplingSize`(size)	Accessor to the Monte Carlo sampling size.
`setSimulationSize`(size)	Accessor to the simulation size.
`setVerbose`(verbose)	Accessor to the verbosity.

computeDetectionSize(**args*, ***kwargs*)

Compute the detection size for a given probability level.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

Returns **result** : collection of `openturns.NumericalPointWithDescription`

A NumericalPointWithDescription containing the detection size computed at the given probability level and confidence level if provided.

drawBoxCoxLikelihood(*name=None*)

Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

drawPOD(**args*, ***kwargs*)

Draw the POD curve.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

defectMin, defectMax : float

Define the interval where the curve is plotted. Default : min and max values of the input sample.

nbPt : int

The number of points to draw the curves. Default is 100.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawValidationGraph(**args*, ***kwargs*)

Draw the validation graph of the metamodel.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getBasis()

Accessor to the kriging basis.

Returns **basis** : `openturns.Basis`

The basis used as trend in the kriging model. Default is a linear basis for the defect and constant for the other parameters.

getBoxCoxParameter()

Accessor to the Box Cox parameter.

Returns **lambdaBoxCox** : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getCovarianceModel()

Accessor to the kriging covariance model.

Returns **covarianceModel** : `openturns.CovarianceModel`

The covariance model in the kriging model. Default is an anisotropic squared exponential covariance model.

getDefectSizes()

Accessor to the defect size where POD is computed.

Returns **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDistribution()

Accessor to the parameters distribution.

Returns **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation. Default is a Uniform distribution for all parameters.

getInitialStartSize()

Accessor to the initial random search size.

Returns **size** : int

The size of the initial random search to find the best loglikelihood value to start the TNC algorithm to optimize the covariance model parameters. Default is 1000.

getKrigingResult()

Accessor to the kriging result.

Returns **result** : `openturns.KrigingResult`

The kriging result.

getPODCLModel(*confidenceLevel=0.95*)

Accessor to the POD model at a given confidence level.

Parameters **confidenceLevel** : float

The confidence level the POD must be computed. Default is 0.95

Returns **PODModelCl** : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value at the confidence level given as parameter.

getPODModel()

Accessor to the POD model.

Returns **PODModel** : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value.

getQ2()

Accessor to the Q2 value.

Returns **Q2** : float

The Q2 value computed analytically using Dubrule (1983) technique.

getSamplingSize()

Accessor to the Monte Carlo sampling size.

Returns **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

getSimulationSize()

Accessor to the simulation size.

Returns **size** : int

The size of the simulation used to compute the confidence interval.

getVerbose()

Accessor to the verbosity.

Returns **verbose** : bool

Enable or disable the verbosity. Default is True.

run()

Build the POD models.

Notes

This method build the kriging model. First the censored data are filtered if needed. The Box Cox transformation is performed if it is enabled. Then it builds the POD models : conditional samples are simulated for each defect size, then the distributions of the probability estimator (for MC simulation) are built. Eventually, a sample of this distribution is used to compute the mean POD and the POD at the confidence level.

setBasis(*basis*)

Accessor to the kriging basis.

Parameters **basis** : `openturns.Basis`

The basis used as trend in the kriging model.

setCovarianceModel(*covarianceModel*)

Accessor to the kriging covariance model.

Parameters **covarianceModel** : `openturns.CovarianceModel`

The covariance model in the kriging model.

setDefectSizes (*size*)
Accessor to the defect size where POD is computed.

Parameters **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDistribution (*distribution*)
Accessor to the parameters distribution.

Parameters **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation.

setInitialStartSize (*size*)
Accessor to the initial random search size.

Parameters **size** : int

The size of the initial random search to find the best loglikelihood value to start the TNC algorithm to optimize the covariance model parameters.

setKrigingResult (*result*)
Accessor to the kriging result.

Parameters **result** : `openturns.KrigingResult`

The kriging result.

setSamplingSize (*size*)
Accessor to the Monte Carlo sampling size.

Parameters **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

setSimulationSize (*size*)
Accessor to the simulation size.

Parameters **size** : int

The size of the simulation used to compute the confidence interval.

setVerbose (*verbose*)
Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

1.1.3 Adaptive algorithms

AdaptiveSignalPOD	Adaptive algorithm for signal data type.
AdaptiveHitMissPOD	Adaptive algorithm for hit miss data type.

AdaptiveSignalPOD

class AdaptiveSignalPOD (**args*)
Adaptive algorithm for signal data type.

Available constructor:

AdaptiveSignalPOD(*inputDOE, outputDOE, physicalModel, nMorePoints, detection, noiseThres, satura-*

tionThres, boxCox)

Parameters **inputDOE** : 2-d sequence of float

Vector of the input values. The first column must correspond with the defect sizes.

outputDOE : 2-d sequence of float

Vector of the signals, of dimension 1.

physicalModel : `NumericalMathFunction`

True model used to compute the real signal value to be added to the DOE.

nMorePoints : positive int

The number of points to add to the DOE, computed by the *physicalModel*.

detection : float

Detection value of the signal.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

Warning: The first column of the input sample must corresponds with the defect sizes.

Notes

This class aims at building the POD based on a kriging model where the design of experiments is iteratively enriched. The initial design of experiments is given as input parameters. The enrichment criterion is based on the integrated mean squared of the POD. The criterion is computed on several candidate points and the one that minimizes the criterion is added to the current design of experiments. The sample of candidate points is created using a low discrepancy sequence (Sobol') if the input distribution has an independant copula, otherwise a Monte Carlo experiment is used. This is a time consuming technique because it requires to compute the mean and variance of the POD for all candidate points. The stopping criterion is only based on the number of points that must be added to the design of experiments.

No assumptions are required for the residuals with this method. The POD are computed by simulating conditional predictions. For each, a Monte Carlo simulation is performed. The accuracy of the Monte Carlo simulation is taken into account using the TCL.

The return POD model corresponds with an interpolate function built with the POD values computed for the given defect sizes. The default values are 20 defect sizes between the minimum and maximum value of the defect sample. The defect sizes can be changed using the method *setDefectSizes*. It is adviced to run a preliminary POD study in order to know the interesting range of defect sizes. This enables reducing the computing time.

The default kriging model is built with a linear basis only for the defect size and constant otherwise. The covariance model is an anisotropic squared exponential model. Parameters are estimated using the TNC algorithm, the initial starting point of the TNC is found thanks to a quasi random search of the best loglikelihood value among 1000 computations.

In the algorithm, when a point is added to the design of experiments, the kriging model is not always optimized. The covariance model scale coefficients are optimized only if the Q2 value is lower than 0.95.

For advanced use, all parameters can be defined thanks to dedicated set methods.

A progress bar is shown if the verbosity is enabled. It can be disabled using the method *setVerbose*.

Methods

`computeDetectionSize`(*args, **kwargs)	Compute the detection size for a given probability level.
`drawBoxCoxLikelihood`([name])	Draw the loglikelihood versus the Box Cox parameter.
`drawPOD`(*args, **kwargs)	Draw the POD curve.
`drawValidationGraph`(*args, **kwargs)	Draw the validation graph of the metamodel.
`getBasis`()	Accessor to the kriging basis.
`getBoxCoxParameter`()	Accessor to the Box Cox parameter.
`getCandidateSize`()	Accessor to the number of candidate points.
`getCovarianceModel`()	Accessor to the kriging covariance model.
`getDefectSizes`()	Accessor to the defect size where POD is computed.
`getDistribution`()	Accessor to the parameters distribution.
`getGraphActive`()	Accessor to the graph verbosity.
`getInitialStartSize`()	Accessor to the initial random search size.
`getInputDOE`()	Accessor to the final input values of the DOE.
`getKrigingResult`()	Accessor to the kriging result.
`getOutputDOE`()	Accessor to the final output values of the DOE.
`getPODCLModel`([confidenceLevel])	Accessor to the POD model at a given confidence level.
`getPODModel`()	Accessor to the POD model.
`getQ2`()	Accessor to the Q2 value.
`getSamplingSize`()	Accessor to the Monte Carlo sampling size.
`getSimulationSize`()	Accessor to the simulation size.
`getVerbose`()	Accessor to the verbosity.
`run`()	Launch the algorithm and build the POD models.
`setBasis`(basis)	Accessor to the kriging basis.
`setCandidateSize`(size)	Accessor to the number of candidate points.
`setCovarianceModel`(covarianceModel)	Accessor to the kriging covariance model.
`setDefectSizes`(size)	Accessor to the defect size where POD is computed.
`setDistribution`(distribution)	Accessor to the parameters distribution.
`setGraphActive`(graphVerbose[, ...])	Accessor to the graph verbosity.
`setInitialStartSize`(size)	Accessor to the initial random search size.
`setSamplingSize`(size)	Accessor to the Monte Carlo sampling size.
`setSimulationSize`(size)	Accessor to the simulation size.
`setVerbose`(verbose)	Accessor to the verbosity.

computeDetectionSize(**args*, ***kwargs*)

Compute the detection size for a given probability level.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

Returns **result** : collection of `openturns.NumericalPointWithDescription`

A NumericalPointWithDescription containing the detection size computed at the given probability level and confidence level if provided.

`drawBoxCoxLikelihood`(*name=None*)

Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

`drawPOD`(**args*, ***kwargs*)

Draw the POD curve.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

defectMin, defectMax : float

Define the interval where the curve is plotted. Default : min and max values of the input sample.

nbPt : int

The number of points to draw the curves. Default is 100.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

`drawValidationGraph`(**args*, ***kwargs*)

Draw the validation graph of the metamodel.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getBasis()

Accessor to the kriging basis.

Returns **basis** : `openturns.Basis`

The basis used as trend in the kriging model. Default is a linear basis for the defect and constant for the other parameters.

getBoxCoxParameter()

Accessor to the Box Cox parameter.

Returns **lambdaBoxCox** : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getCandidateSize()

Accessor to the number of candidate points.

Returns **size** : int

The number of candidate points on which the criterion is computed.

getCovarianceModel()

Accessor to the kriging covariance model.

Returns **covarianceModel** : `openturns.CovarianceModel`

The covariance model in the kriging model. Default is an anisotropic squared exponential covariance model.

getDefectSizes()

Accessor to the defect size where POD is computed.

Returns **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDistribution()

Accessor to the parameters distribution.

Returns **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation. Default is a Uniform distribution for all parameters.

getGraphActive()

Accessor to the graph verbosity.

Returns **graphVerbose** : bool

Enable or disable the display of the POD graph at each iteration. Default is False.

getInitialStartSize()

Accessor to the initial random search size.

Returns size : int

The size of the initial random search to find the best loglikelihood value to start the TNC algorithm to optimize the covariance model parameters. Default is 1000.

getInputDOE()

Accessor to the final input values of the DOE.

getKrigingResult()

Accessor to the kriging result.

Returns result : `openturns.KrigingResult`

The kriging result.

getOutputDOE()

Accessor to the final output values of the DOE.

getPODCLModel(*confidenceLevel=0.95*)

Accessor to the POD model at a given confidence level.

Parameters confidenceLevel : float

The confidence level the POD must be computed. Default is 0.95

Returns PODModelCl : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value at the confidence level given as parameter.

getPODModel()

Accessor to the POD model.

Returns PODModel : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value.

getQ2()

Accessor to the Q2 value.

Returns Q2 : float

The Q2 value computed analytically using Dubrule (1983) technique.

getSamplingSize()

Accessor to the Monte Carlo sampling size.

Returns size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

getSimulationSize()

Accessor to the simulation size.

Returns size : int

The size of the simulation used to compute the confidence interval.

getVerbose()

Accessor to the verbosity.

Returns verbose : bool

Enable or disable the verbosity. Default is True.

run()

Launch the algorithm and build the POD models.

Notes

This method launches the iterative algorithm. First the censored data are filtered if needed. The Box Cox transformation is performed if it is enabled. Then the enrichment of the design of experiments is performed. Once the algorithm stops, it builds the POD models : conditional samples are simulated for each defect size, then the distributions of the probability estimator (for MC simulation) are built. Eventually, a sample of this distribution is used to compute the mean POD and the POD at the confidence level.

setBasis(*basis*)
Accessor to the kriging basis.

Parameters **basis** : `openturns.Basis`

The basis used as trend in the kriging model.

setCandidateSize(*size*)
Accessor to the number of candidate points.

Parameters **size** : int

The number of candidate points on which the criterion is computed

setCovarianceModel(*covarianceModel*)
Accessor to the kriging covariance model.

Parameters **covarianceModel** : `openturns.CovarianceModel`

The covariance model in the kriging model.

setDefectSizes(*size*)
Accessor to the defect size where POD is computed.

Parameters **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDistribution(*distribution*)
Accessor to the parameters distribution.

Parameters **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation.

setGraphActive(*graphVerbose*, *probabilityLevel=None*, *confidenceLevel=None*, *directory=None*)
Accessor to the graph verbosity.

Parameters **graphVerbose** : bool

Enable or disable the display of the POD graph at each iteration.

probabilityLevel : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

directory : string

Directory where to save the graphs as png files.

setInitialStartSize(*size*)
Accessor to the initial random search size.

Parameters **size** : int

The size of the initial random search to find the best loglikelihood value to start the TNC algorithm to optimize the covariance model parameters.

setSamplingSize(*size*)

Accessor to the Monte Carlo sampling size.

Parameters **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

setSimulationSize(*size*)

Accessor to the simulation size.

Parameters **size** : int

The size of the simulation used to compute the confidence interval.

setVerbose(*verbose*)

Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

AdaptiveHitMissPOD

class **AdaptiveHitMissPOD**(**args*)

Adaptive algorithm for hit miss data type.

Available constructor:

AdaptiveHitMissPOD(*inputDOE, outputDOE, physicalModel, nMorePoints, detection, noiseThres, saturationThres*)

Parameters **inputDOE** : 2-d sequence of float

Vector of the input values. The first column must correspond with the defect sizes.

outputDOE : 2-d sequence of float

Vector of the signals, of dimension 1.

physicalModel : `NumericalMathFunction`

True model used to compute the real hit miss value of the signal value to be added to the DOE.

nMorePoints : positive int

The number of points to add to the DOE, computed by the *physicalModel*.

detection : float

Detection value of the signal if the physical model does not return a hit miss value.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

Warning: The first column of the input sample must corresponds with the defect sizes.

Notes

This class aims at building the POD based on a classifier model where the design of experiments is iteratively enriched. The initial design of experiments is given as input parameters. The enrichment criterion is based on the misclassification empirical risk. The criterion is computed on several candidate points. The sample of candidate points is created using a low discrepancy sequence (Sobol') if the input distribution has an independant copula, otherwise a Monte Carlo experiment is used. The stopping criterion is only based on the number of points that must be added to the design of experiments.

The classifier algorithms availables are the SVC and the random forests. The choice of the algorithm can be defined using *setClassifierType*. The default algorithm is the random forests.

The physical model can return either the hit miss value (0 or 1) or the signal value. In this case, the detection value must be given and the physical model is transformed in order to provide a hit miss value.

The POD are computed by a Monte Carlo simulation for several defect values. The accuracy of the Monte Carlo simulation is taken into account using the TCL. The return POD model corresponds with an interpolate function built with the POD values computed for the given defect sizes. The default values are 20 defect sizes between the minimum and maximum value of the defect sample. The defect sizes can be changed using the method *setDefectSizes*.

A progress bar is shown if the verbosity is enabled. It can be disabled using the method *setVerbose*.

Methods

`computeDetectionSize`(*args, **kwargs)	Compute the detection size for a given probability level.
`drawBoxCoxLikelihood`([name])	Draw the loglikelihood versus the Box Cox parameter.
`drawPOD`(*args, **kwargs)	Draw the POD curve.
`getBoxCoxParameter`()	Accessor to the Box Cox parameter.
`getCandidateSize`()	Accessor to the number of candidate points.
`getClassifier`()	Accessor to the classifier model.
`getClassifierParameters`()	Accessor to the classifier parameters.
`getClassifierType`()	Accessor to the classifier type.
`getConfusionMatrix`()	Accessor to the confusion matrix.
`getDefectSizes`()	Accessor to the defect size where POD is computed.
`getDistribution`()	Accessor to the parameters distribution.
`getGraphActive`()	Accessor to the graph verbosity.
`getInputDOE`()	Accessor to the final input values of the DOE.
`getOutputDOE`()	Accessor to the final output values of the DOE.
`getPMax`()	Accessor to the upper probability bound for the point selections.
`getPMin`()	Accessor to the lower probability bound for the point selections.
`getPODCLModel`([confidenceLevel])	Accessor to the POD model at a given confidence level.
`getPODModel`()	Accessor to the POD model.
`getSamplingSize`()	Accessor to the Monte Carlo sampling size.
`getSimulationSize`()	Accessor to the simulation size.
`getVerbose`()	Accessor to the verbosity.

Continued on next page

Table 1.10 – continued from previous page

run()	Launch the algorithm and build the POD models.
setCandidateSize(size)	Accessor to the number of candidate points.
setClassifierParameters(parameters)	Accessor to the classifier parameters.
setClassifierType(classifier)	Accessor to the classifier type.
setDefectSizes(size)	Accessor to the defect size where POD is computed.
setDistribution(distribution)	Accessor to the parameters distribution.
setGraphActive(graphVerbose[, ...])	Accessor to the graph verbosity.
setPMax(pmax)	Accessor to the upper probability bound for the point selections.
setPMin(pmin)	Accessor to the lower probability bound for the point selections.
setSamplingSize(size)	Accessor to the Monte Carlo sampling size.
setSimulationSize(size)	Accessor to the simulation size.
setVerbose(verbose)	Accessor to the verbosity.

computeDetectionSize(**args*, ***kwargs*)
Compute the detection size for a given probability level.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

Returns **result** : collection of `openturns.NumericalPointWithDescription`

A NumericalPointWithDescription containing the detection size computed at the given probability level and confidence level if provided.

drawBoxCoxLikelihood(*name=None*)
Draw the loglikelihood versus the Box Cox parameter.

Parameters **name** : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

Notes

This method is available only when the parameter *boxCox* is set to True.

drawPOD(**args*, ***kwargs*)
Draw the POD curve.

Parameters **probabilityLevel** : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

defectMin, defectMax : float

Define the interval where the curve is plotted. Default : min and max values of the input sample.

nbPt : int

The number of points to draw the curves. Default is 100.

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns fig : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getBoxCoxParameter()

Accessor to the Box Cox parameter.

Returns lambdaBoxCox : float

The Box Cox parameter used to transform the data. If the transformation is not enabled None is returned.

getCandidateSize()

Accessor to the number of candidate points.

Returns size : int

The number of candidate points on which the criterion is computed.

getClassifier()

Accessor to the classifier model.

Returns result : classifier

The classifier model, either random forest or svm.

getClassifierParameters()

Accessor to the classifier parameters.

getClassifierType()

Accessor to the classifier type.

getConfusionMatrix()

Accessor to the confusion matrix.

getDefectSizes()

Accessor to the defect size where POD is computed.

Returns defectSize : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDistribution()
Accessor to the parameters distribution.

Returns distribution : `openturns.ComposedDistribution`

The input parameters distribution, default is a Uniform distribution for all parameters.

getGraphActive()
Accessor to the graph verbosity.

Returns graphVerbose : bool

Enable or disable the display of the POD graph at each iteration. Default is False.

getInputDOE()
Accessor to the final input values of the DOE.

getOutputDOE()
Accessor to the final output values of the DOE.

getPMax()
Accessor to the upper probability bound for the point selections.

getPMin()
Accessor to the lower probability bound for the point selections.

getPODCLModel(*confidenceLevel=0.95*)
Accessor to the POD model at a given confidence level.

Parameters confidenceLevel : float

The confidence level the POD must be computed. Default is 0.95

Returns PODModelCl : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value at the confidence level given as parameter.

getPODModel()
Accessor to the POD model.

Returns PODModel : `openturns.NumericalMathFunction`

The function which computes the probability of detection for a given defect value.

getSamplingSize()
Accessor to the Monte Carlo sampling size.

Returns size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

getSimulationSize()
Accessor to the simulation size.

Returns size : int

The size of the simulation used to compute the confidence interval.

getVerbose()
Accessor to the verbosity.

Returns verbose : bool

Enable or disable the verbosity. Default is True.

run()
Launch the algorithm and build the POD models.

Notes

This method launches the iterative algorithm. Once the algorithm stops, it builds the POD models : Monte Carlo simulation are performed for each defect sizes with the final classifier model. Eventually, the sample is used to compute the mean POD and the POD at the confidence level.

setCandidateSize(*size*)
Accessor to the number of candidate points.

Parameters **size** : int

The number of candidate points on which the criterion is computed

setClassifierParameters(*parameters*)
Accessor to the classifier parameters.

setClassifierType(*classifier*)
Accessor to the classifier type.

setDefectSizes(*size*)
Accessor to the defect size where POD is computed.

Parameters **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDistribution(*distribution*)
Accessor to the parameters distribution.

Parameters **distribution** : `openturns.ComposedDistribution`

The input parameters distribution.

setGraphActive(*graphVerbose*, *probabilityLevel=None*, *confidenceLevel=None*, *directory=None*)
Accessor to the graph verbosity.

Parameters **graphVerbose** : bool

Enable or disable the display of the POD graph at each iteration.

probabilityLevel : float

The probability level for which the defect size is computed. Default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

directory : string

Directory where to save the graphs as png files.

setPMax(*pmax*)
Accessor to the upper probability bound for the point selections.

setPMin(*pmin*)
Accessor to the lower probability bound for the point selections.

setSamplingSize(*size*)
Accessor to the Monte Carlo sampling size.

Parameters **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

setSimulationSize(*size*)

Accessor to the simulation size.

Parameters **size** : int

The size of the simulation used to compute the confidence interval.

setVerbose(*verbose*)

Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

1.1.4 Sensitivity analysis

SobolIndices	Sensitivity analysis based on Sobol' indices.
PLI	PLI base class.
PLIMean	PLI based on a mean perturbation.
PLIVariance	PLI based on a mean perturbation.

SobolIndices

class **SobolIndices**(**args*)

Sensitivity analysis based on Sobol' indices.

Available constructor:

SobolIndices(*POD, N*)

Parameters **POD** : *KrigingPOD*, *AdaptiveSignalPOD* or *PolynomialChaosPOD*

The POD object where the run method has been performed.

N : int

Size of samples to generate

Returns **sa** : openturns.SobolIndicesAlgorithm

The openturns object that perform the sensitivity algorithm.

Notes

This class uses the openturns.SobolIndicesAlgorithm class of OpenTURNS. The sensitivity analysis can be performed only with a POD built with a Kriging metamodel or a polynomial chaos where the input dimension is greater than 3 (counting the defect).

When using Kriging, the POD at a given point is computed using the kriging mean and variance. For polynomial chaos, random coefficients are generated, the signal is computed for all coefficients and the POD is eventually estimated. The default simulation size is set to 1000. This value can be changed using *setSimulationSize()*.

The sensitivity analysis allows to computed aggregated Sobol indices for the given range of defect sizes. The default defect sizes correspond with those defined in the *POD* object. It can be changed using `setDefectSizes()`.

The four methods developed in OpenTURNS are availables and can be chosen thanks to `setSensitivityMethod()`. The default method is "Saltelli".

The result of the sensitivity analysis is available using `getSensitivityResult()`. It returns the openturns sensitivity object from which the sensitivity values are given using proper methods.

Methods

`drawAggregatedIndices`([label, name])	Plot the aggregated Sobol indices.
`drawFirstOrderIndices`([label, name])	Plot the first Sobol indices for all defect values.
`drawTotalOrderIndices`([label, name])	Plot the total Sobol indices for all defect values.
`getDefectSizes`()	Accessor to the defect size where the POD is computed.
`getDistribution`()	Accessor to the parameters distribution.
`getSensitivityMethod`()	Accessor to the sensitivity method.
`getSensitivityResult`()	Accessor to the OpenTURNS sensitivity object.
`getSimulationSize`()	Accessor to the simulation size when using polynomial chaos.
`run`()	Compute the Sobol indices with the chosen algorithm.
`setDefectSizes`(size)	Accessor to the defect size where the POD is computed.
`setDistribution`(distribution)	Accessor to the parameters distribution.
`setSensitivityMethod`(method)	Accessor to the sensitivity method.
`setSimulationSize`(size)	Accessor to the simulation size when using polynomial chaos.

drawAggregatedIndices(*label=None*, *name=None*)

Plot the aggregated Sobol indices.

Parameters **label** : sequence of float

The name of the input parameters

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawFirstOrderIndices(*label=None*, *name=None*)

Plot the first Sobol indices for all defect values.

Parameters **label** : sequence of float

The name of the input parameters

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawTotalOrderIndices(*label=None*, *name=None*)

Plot the total Sobol indices for all defect values.

Parameters **label** : sequence of float

The name of the input parameters

name : string

name of the figure to be saved with *transparent* option sets to True and *bbox_inches='tight'*. It can be only the file name or the full path name. Default is None.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getDefectSizes()

Accessor to the defect size where the POD is computed.

Returns **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDistribution()

Accessor to the parameters distribution.

Returns **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation. Default is a Uniform distribution for all parameters.

getSensitivityMethod()

Accessor to the sensitivity method.

Returns **method** : str

The sensitivity method.

getSensitivityResult()

Accessor to the OpenTURNS sensitivity object.

Returns **sa** : `SobolIndicesAlgorithm`

getSimulationSize()

Accessor to the simulation size when using polynomial chaos.

Returns **size** : int

The size of the simulation used to compute POD at a given point.

run()
Compute the Sobol indices with the chosen algorithm.

setDefectSizes(*size*)
Accessor to the defect size where the POD is computed.

Parameters defectSize : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDistribution(*distribution*)
Accessor to the parameters distribution.

Parameters distribution : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation.

setSensitivityMethod(*method*)
Accessor to the sensitivity method.

Parameters method : str

The sensitivity method: either "Saltelli", "Martinez", "Jansen" or "MauntzKucherenko". Default is "Saltelli".

setSimulationSize(*size*)
Accessor to the simulation size when using polynomial chaos.

Parameters size : int

The size of the simulation used to compute at a given point. Default is 1000.

PLI

class PLI(**args*)
PLI base class.

See also:

PLIMean, *PLIVariance*

Notes

The Perturbation Law Indices are based upon the modification of the probability density function (pdf) of the random inputs, when the quantity of interest is a failure probability. An input is considered influential if the input pdf modification leads to a broad change in the failure probability. These sensitivity indices can be computed using the sole set of simulations that has already been used to estimate the failure probability, thus limiting the number of calls to the numerical model. In this implementation, the sample must come from a Monte Carlo simulation.

The input perturbation is defined to obtain the perturbed density function as the closest to the original one, in the sense of the Kullback-Leibler divergence. The implemented perturbation includes a mean shift and a variance shift, accessible through the derived class. The current implementation only allows to modifiy Normal and Uniform density functions.

In order to compare equivalently the indices when the input distributions are not the same, it is possible to plot the indices with respect to the Hellinger distance.

These indices have been developed by Paul Lemaitre:

- Paul Lemaître, Ekatarina Sergienko, Aurélie Arnaud, Nicolas Bousquet, Fabrice Gamboa, et al.. Density modification based reliability sensitivity analysis. 2012.
- Paul Lemaitre. Analyse de sensibilité en fiabilité des structures. Mécanique des structures [physics.class-ph]. Université de Bordeaux, 2014. Français.

Methods

`computeConfidenceInterval`([confidenceLevel])	Accessor to the confidence interval of the indices.
`drawIndices`([confidenceLevel, label, ...])	Draw all indices
`drawMarginal1DPDF`(marginal, idelta[, ...])	Draw the probability density function of a margin.
`getDeltaSample`()	Accessor to applied delta values.
`getGaussKronrod`()	Accessor to the Gauss Kronrod algorithm used to compute integrals
`getIndices`()	Accessor to the Pertubation Law Indices.
`getOriginalDelta`(marginal)	Accessor to the original delta value
`getPerturbedProbabilityEstimate`()	Accessor to the perturbed probability of failure
`run`()	Run the analysis:
`setGaussKronrod`(algo)	Accessor to the Gauss Kronrod algorithm used to compute integrals

computeConfidenceInterval(*confidenceLevel=0.95*)

Accessor to the confidence interval of the indices.

Parameters **confidenceLevel** : 0 < float < 1

The wanted confidence level to compute the interval.

Returns **ci** : list of 2d sequence of float

A list of arrays for each marginal containing the lower and upper bound of the confidence interval for each delta values.

drawIndices(*confidenceLevel=0.95*, *label=None*, *hellinger=False*, *name=None*)

Draw all indices

Parameters **confidenceLevel** : 0 < float < 1 or None

The wanted confidence level to compute the interval. If set to 'None' only the indices are plotted.

label : list of string

The labels of each parameters.

hellinger : bool

If True, the indices are plotted with respect to the hellinger distance between the original PDF and the perturbed PDF.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawMarginal1DPDF(*marginal*, *idelta*, *showOriginal=True*, *label=None*, *xMin=None*, *xMax=None*, *pointNumber=None*, *name=None*)

Draw the probability density function of a margin.

Parameters **marginal** : int

The index of the margin of interest.

idelta : int

The index in the delta array.

showOriginal : bool

Display on the same figure the original pdf or not.

x_min : float

The starting value that is used for meshing the x-axis. Defaults uses the quantile associated to the probability level 0.05.

x_max : float, $x_{\max} > x_{\min}$

The ending value that is used for meshing the x-axis. Defaults uses the quantile associated to the probability level 0.95.

n_points : int

The number of points that is used for meshing the x-axis. Defaults uses *DistributionImplementation-DefaultPointNumber* from the `ResourceMap`.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getDeltaSample()

Accessor to applied delta values.

getGaussKronrod()

Accessor to the Gauss Kronrod algorithm used to compute integrals

getIndices()

Accessor to the Pertubation Law Indices.

getOriginalDelta(*marginal*)

Accessor to the original delta value

Parameters **marginal** : int

The indice of the perturbed marginal.

getPerturbedProbabilityEstimate()

Accessor to the perturbed probability of failure

Returns **pfdelta** : float

The probability of failure computed with the perturbed density function.

run()

Run the analysis: - get the failure sample - evaluate the probabilities with the perturbed distributions - define the estimator distributions

setGaussKronrod(*algo*)

Accessor to the Gauss Kronrod algorithm used to compute integrals

Parameters **algo** : `GaussKronrod`

The algorithm

PLIMean

class PLIMean (**args*)

PLI based on a mean perturbation.

Parameters **POD** : *KrigingPOD*, *AdaptiveSignalPOD* or *PolynomialChaosPOD*

The POD object where the run method has been performed.

delta : 1d or 2d sequence of float

The new values of the mean or sigma coefficient. Either 1d if delta values are the same for all marginals, or 2d if delta values are defined independently for each marginal.

sigmaScaled : bool

Change the type of the applied mean shiftingfor all the variables. If False (default case), the given delta values are the new marginal means. If True, newMean = mean + sigma x delta, where sigma is the standard deviation of each marginals.

Methods

drawContourIndices(marginal[, label, name])	Draw a contour plot of the indices for a specific marginal
drawIndices(idefect[, confidenceLevel, ...])	Draw the indices of all margins for a specific defect
getDefectSizes()	Accessor to the defect size where the indices are computed.
getDistribution()	Accessor to the parameters distribution.
getGaussKronrod()	Accessor to the Gauss Kronrod algorithm used to compute integrals
getIndices([idelta, marginal, idefect])	Accessor to the indices
getPLIObject(idefect)	Accessor to the PLI object for a specific defect.
getSamplingSize()	Accessor to the Monte Carlo sampling size.
run()	Compute the indices
setDefectSizes(size)	Accessor to the defect size where the indices are computed.
setDistribution(distribution)	Accessor to the parameters distribution.
setGaussKronrod(algo)	Accessor to the Gauss Kronrod algorithm used to compute integrals
setSamplingSize(size)	Accessor to the Monte Carlo sampling size.

drawContourIndices (*marginal*, *label=None*, *name=None*)

Draw a contour plot of the indices for a specific marginal

Parameters **marginal** : int

The indice of the perturbed marginal.

label : list of string

The labels of each parameters.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawIndices(*idefect*, *confidenceLevel=0.95*, *label=None*, *hellinger=True*, *name=None*)

Draw the indices of all margins for a specific defect

Parameters **idefect** : int

The indice of the defect in the given delta list.

confidenceLevel : 0 < float < 1 or None

The wanted confidence level to compute the interval. If set to 'None' only the indices are plotted.

label : list of string

The labels of each parameters.

hellinger : bool

If True, the indices are plotted with respect to the hellinger distance between the original PDF and the perturbed PDF. Default is True.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getDefectSizes()

Accessor to the defect size where the indices are computed.

Returns **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDistribution()

Accessor to the parameters distribution.

Returns **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation. Default is a Uniform distribution for all parameters.

getGaussKronrod()

Accessor to the Gauss Kronrod algorithm used to compute integrals

getIndices(*idelta=None*, *marginal=None*, *idefect=None*)

Accessor to the indices

Parameters **idelta** : int

The indice of the delta in the given delta list. Default is None = all.

marginal : int

The indice of the perturbed marginal. Default is None = all.

idefect : int

The indice of the defect in the given delta list. Default is None = all.

Returns **indices** : float, 1d, 2d or 3d array.

The parameter order of the full matrix is delta, marginal and defect. The returned array depends on the given parameter values.

getPLIObject(*idefect*)
Accessor to the PLI object for a specific defect.

Parameters **idefect** : int

The indice of the defect in the given delta list.

Returns **pli** : *PLI*

The PLI base object from which more results can be obtained.

getSamplingSize()
Accessor to the Monte Carlo sampling size.

Returns **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

run()
Compute the indices

Notes

Run the analysis:

- run a Monte Carlo simulation
- compute the indices for each defect size

If, for a defect size, the probability estimate is less than 1e-3 or greater than 0.999, then the indices are not computed.

setDefectSizes(*size*)
Accessor to the defect size where the indices are computed.

Parameters **defectSize** : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDistribution(*distribution*)
Accessor to the parameters distribution.

Parameters **distribution** : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation.

setGaussKronrod(*algo*)
Accessor to the Gauss Kronrod algorithm used to compute integrals

Parameters **algo** : `GaussKronrod`

The algorithm

setSamplingSize(*size*)
Accessor to the Monte Carlo sampling size.

Parameters **size** : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

PLIVariance

class PLIVariance (**args*)

PLI based on a mean perturbation.

Parameters **POD** : *KrigingPOD*, *AdaptiveSignalPOD* or *PolynomialChaosPOD*

The POD object where the run method has been performed.

delta : 1d or 2d sequence of float

The new values of the mean. Either 1d if delta values are the same for all marginals, or 2d if delta values are defined independently for each marginal.

covScaled : bool

Change the type of the applied variance shifting for all the variables. If False (default case), the given delta values are the new marginal variances. If True, newVariance = variance x delta.

Methods

drawContourIndices(marginal[, label, name])	Draw a contour plot of the indices for a specific marginal
drawIndices(idefect[, confidenceLevel, ...])	Draw the indices of all margins for a specific defect
getDefectSizes()	Accessor to the defect size where the indices are computed.
getDistribution()	Accessor to the parameters distribution.
getGaussKronrod()	Accessor to the Gauss Kronrod algorithm used to compute integrals
getIndices([idelta, marginal, idefect])	Accessor to the indices
getPLIObject(idefect)	Accessor to the PLI object for a specific defect.
getSamplingSize()	Accessor to the Monte Carlo sampling size.
run()	Compute the indices
setDefectSizes(size)	Accessor to the defect size where the indices are computed.
setDistribution(distribution)	Accessor to the parameters distribution.
setGaussKronrod(algo)	Accessor to the Gauss Kronrod algorithm used to compute integrals
setSamplingSize(size)	Accessor to the Monte Carlo sampling size.

drawContourIndices (*marginal*, *label=None*, *name=None*)

Draw a contour plot of the indices for a specific marginal

Parameters **marginal** : int

The indice of the perturbed marginal.

label : list of string

The labels of each parameters.

Returns **fig** : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

drawIndices(*idefect*, *confidenceLevel=0.95*, *label=None*, *hellinger=True*, *name=None*)
Draw the indices of all margins for a specific defect

Parameters idefect : int

The indice of the defect in the given delta list.

confidenceLevel : 0 < float < 1 or None

The wanted confidence level to compute the interval. If set to 'None' only the indices are plotted.

label : list of string

The labels of each parameters.

hellinger : bool

If True, the indices are plotted with respect to the hellinger distance between the original PDF and the perturbed PDF. Default is True.

Returns fig : matplotlib.figure

Matplotlib figure object.

ax : matplotlib.axes

Matplotlib axes object.

getDefectSizes()
Accessor to the defect size where the indices are computed.

Returns defectSize : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

getDistribution()
Accessor to the parameters distribution.

Returns distribution : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation. Default is a Uniform distribution for all parameters.

getGaussKronrod()
Accessor to the Gauss Kronrod algorithm used to compute integrals

getIndices(*idelta=None*, *marginal=None*, *idefect=None*)
Accessor to the indices

Parameters idelta : int

The indice of the delta in the given delta list. Default is None = all.

marginal : int

The indice of the perturbed marginal. Default is None = all.

idefect : int

The indice of the defect in the given delta list. Default is None = all.

Returns indices : float, 1d, 2d or 3d array.

The parameter order of the full matrix is delta, marginal and defect. The returned array depends on the given parameter values.

getPLIObject(*idefect*)
Accessor to the PLI object for a specific defect.

Parameters idefect : int

The indice of the defect in the given delta list.

Returns pli : *PLI*

The PLI base object from which more results can be obtained.

getSamplingSize()
Accessor to the Monte Carlo sampling size.

Returns size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

run()
Compute the indices

Notes

Run the analysis:

- run a Monte Carlo simulation
- compute the indices for each defect size

If, for a defect size, the probability estimate is less than 1e-3 or greater than 0.999, then the indices are not computed.

setDefectSizes(*size*)
Accessor to the defect size where the indices are computed.

Parameters defectSize : sequence of float

The defect sizes where the Monte Carlo simulation is performed to compute the POD.

setDistribution(*distribution*)
Accessor to the parameters distribution.

Parameters distribution : `openturns.ComposedDistribution`

The input parameters distribution used for the Monte Carlo simulation.

setGaussKronrod(*algo*)
Accessor to the Gauss Kronrod algorithm used to compute integrals

Parameters algo : `GaussKronrod`

The algorithm

setSamplingSize(*size*)
Accessor to the Monte Carlo sampling size.

Parameters size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size.

1.1.5 Tools

`PODSummary`	Run the analysis and compute POD with several methods.
`DataHandling`	Static methods for data handling.

PODSummary

class **`PODSummary`**(**args*)

Run the analysis and compute POD with several methods.

Available constructor:

PODSummary(*inputSample, outputSample, detection, noiseThres, saturationThres, boxCox*)

Parameters **inputSample** : 2-d sequence of float

Vector of the input values. The first column must correspond with the defect sizes.

outputSample : 2-d sequence of float

Vector of the signals, of dimension 1.

detection : float

Detection value of the signal.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

boxCox : bool or float

Enable or not the Box Cox transformation. If boxCox is a float, the Box Cox transformation is enabled with the given value. Default is False.

> **Warning:** The first column of the input sample must corresponds with the defects sample.

Notes

This class aims at running the linear analysis and computing the POD with different models:

- Linear regression model with Gaussian residuals hypothesis,
- Linear regression model with no hypothesis on the residuals (binomial),
- Linear regression model with with kernel smoothing on the residuals,
- Quantile regression,
- Polynomial chaos,
- kriging if the dimension of the input sample is greater than 1.

Each method can be deactivated using the method *setMethodActive* and using the key corresponding to the method.

All results can be displayed and saved thanks to the methods *printResults*, *saveResults* and *saveGraphs*. For each method, the probability level and confidence level can be specified in order to compute the defect size to the wanted probability level.

The verbosity is enabled by default but it can be disabled using the method *setVerbose*.

Methods

drawGraphs([directory, extension, ...])	draw and save all possible graphs
getKrigingPOD()	Accessor to the kriging POD object.
getLinearBinomialPOD()	Accessor to the linear model POD object with no hypothesis on the residuals.
getLinearGaussPOD()	Accessor to the linear model POD object with Gaussian hypothesis.
getLinearKernelSmoothingPOD()	Accessor to the linear model POD object with kernel smoothing on the residuals.
getMethodActive()	Accessor to the dictionnary of active methods.
getPolynomialChaosPOD()	Accessor to the polynomial chaos POD object.
getQuantileRegressionPOD()	Accessor to the quantile regression POD object.
getResults([probabilityLevel, confidenceLevel])	Print all results in the terminal.
getSamplingSize()	Accessor to the Monte Carlo sampling size.
getSimulationSize()	Accessor to the simulation size.
getVerbose()	Accessor to the verbosity.
run()	Run all active methods.
saveResults(name[, probabilityLevel, ...])	Save all analysis test results in a file.
setMethodActive(method, activation)	Accessor to the dictionnary of active methods.
setSamplingSize(size)	Accessor to the Monte Carlo sampling size.
setSimulationSize(size)	Accessor to the simulation size.
setVerbose(verbose)	Accessor to the verbosity.

drawGraphs(*directory=None*, *extension='png'*, *probabilityLevel=None*, *confidenceLevel=None*)

draw and save all possible graphs

Parameters **directory** : string

Directory where to save the graphs. Default is the working directory.

extension : string

File extension of the graphs. Default is 'png'.

probabilityLevel : float

The probability level for which the defect size is computed. default is None.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is None.

getKrigingPOD()

Accessor to the kriging POD object.

Returns **algorithm** : KrigingPOD

The KrigingPOD object that is used to compute the POD.

getLinearBinomialPOD()

Accessor to the linear model POD object with no hypothesis on the residuals.

Returns **algorithm** : UnivariateLinearModelPOD

The UnivariateLinearModelPOD object that is used to compute the POD.

getLinearGaussPOD()
Accessor to the linear model POD object with Gaussian hypothesis.

Returns algorithm : UnivariateLinearModelPOD

The UnivariateLinearModelPOD object that is used to compute the POD.

getLinearKernelSmoothingPOD()
Accessor to the linear model POD object with kernel smoothing on the residuals.

Returns algorithm : UnivariateLinearModelPOD

The UnivariateLinearModelPOD object that is used to compute the POD.

getMethodActive()
Accessor to the dictionnary of active methods.

Returns activeDict : dict

The dictionnary containing the bool telling if the methods is activated or not.

getPolynomialChaosPOD()
Accessor to the polynomial chaos POD object.

Returns algorithm : PolynomialChaosPOD

The PolynomialChaosPOD object that is used to compute the POD.

getQuantileRegressionPOD()
Accessor to the quantile regression POD object.

Returns algorithm : QuantileRegressionPOD

The QuantileRegressionPOD object that is used to compute the POD.

getResults(*probabilityLevel=0.9*, *confidenceLevel=0.95*)
Print all results in the terminal.

Parameters probabilityLevel : float

The probability level for which the defect size is computed. default is 0.9.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is 0.95.

Notes

The probability level and confidence level can be specified in order to display the defect size for different probability level.

getSamplingSize()
Accessor to the Monte Carlo sampling size.

Returns size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size for polynomial chaos and kriging.

getSimulationSize()
Accessor to the simulation size.

Returns size : int

The size of the simulation used to compute the confidence interval.

getVerbose()

Accessor to the verbosity.

Returns verbose : bool

Enable or disable the verbosity. Default is True.

run()

Run all active methods.

saveResults(*name*, *probabilityLevel=0.9*, *confidenceLevel=0.95*)

Save all analysis test results in a file.

Parameters name : string

Name of the file or full path name.

probabilityLevel : float

The probability level for which the defect size is computed. default is 0.9.

confidenceLevel : float

The confidence level associated to the given probability level the defect size is computed. Default is 0.95.

Notes

The probability level and confidence level can be specified in order to display the defect size for different probability level.

The file can be saved as a csv file. Separations are made with tabulations.

If *name* is the file name, then it is saved in the current working directory.

setMethodActive(*method*, *activation*)

Accessor to the dictionnary of active methods.

Parameters method : string

The key of the method to activate or deactivate.

activation : bool

Set to True to activate and False to deactivate.

setSamplingSize(*size*)

Accessor to the Monte Carlo sampling size.

Parameters size : int

The size of the Monte Carlo simulation used to compute the POD for each defect size for polynomial chaos and kriging.

setSimulationSize(*size*)

Accessor to the simulation size.

Parameters size : int

The size of the simulation used to compute the confidence interval.

setVerbose (*verbose*)
Accessor to the verbosity.

Parameters **verbose** : bool

Enable or disable the verbosity.

DataHandling

class **DataHandling**
Static methods for data handling.

Methods

filterCensoredData(inputSample, signals, ...)	Sort inputSample and signals with respect to the censore thresholds.

static **filterCensoredData** (*inputSample*, *signals*, *noiseThres*, *saturationThres*)
Sort inputSample and signals with respect to the censore thresholds.

Parameters **inputSample** : 2-d sequence of float

Vector of the input sample.

signals : 2-d sequence of float

Vector of the signals, of dimension 1.

noiseThres : float

Value for low censored data. Default is None.

saturationThres : float

Value for high censored data. Default is None

Returns **inputSampleUnc** : 2-d sequence of float

Vector of the input sample in the uncensored area.

inputSampleNoise : 2-d sequence of float

Vector of the input sample in the noisy area.

inputSampleSat : 2-d sequence of float

Vector of the input sample in the saturation area.

signalsUnc : 2-d sequence of float

Vector of the signals in the uncensored area.

Notes

The data are sorted in three different vectors whether they belong to the noisy area, the uncensored area or the saturation area.

1.2 Examples of the API

ipynb source code

1.2.1 Linear model analysis

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
N = 100
ot.RandomGenerator.SetSeed(123456)
defectDist = ot.Uniform(0.1, 0.6)
# normal epsilon distribution
epsilon = ot.Normal(0, 1.9)
defects = defectDist.getSample(N)
signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
# Inverse Box Cox transformation
invBoxCox = ot.InverseBoxCoxTransform(0.3)
signals = invBoxCox(signalsInvBoxCox)
```

Run analysis without Box Cox

```
analysis = otpod.UnivariateLinearModelAnalysis(defects, signals)
```

Get some particular results

```
print(analysis.getIntercept())
print(analysis.getR2())
print(analysis.getKolmogorovPValue())
```

```
[Intercept for uncensored case : -604.758]
[R2 for uncensored case : 0.780469]
[Kolmogorov p-value for uncensored case : 0.803087]
```

Print all results of the linear regression and all tests on the residuals

A warning is printed because some residuals tests failed : the p-value is less than 0.5.

```
print(analysis.getResults())
```

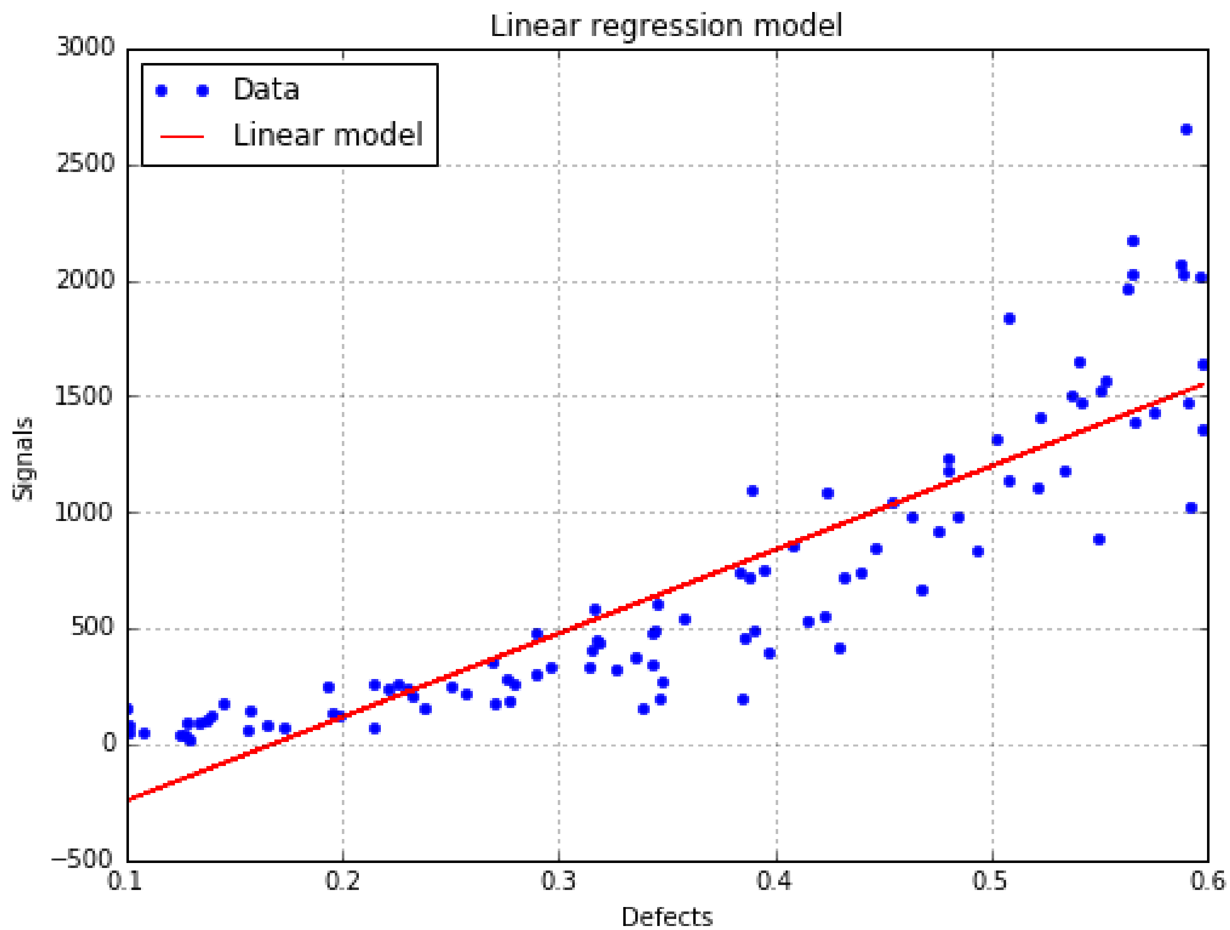
```
--------------------------------------------------------------------------------
         Linear model analysis results
--------------------------------------------------------------------------------
```

```
Box Cox parameter :                                 Not enabled

                                                     Uncensored

Intercept coefficient :                                 -604.76
Slope coefficient :                                     3606.04
Standard error of the estimate :                         291.47

Confidence interval on coefficients
Intercept coefficient :                        [-755.60, -453.91]
Slope coefficient :                            [3222.66, 3989.43]
Level :                                                    0.95

Quality of regression
R2 (> 0.8):                                                0.78
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
         Residuals analysis results
--------------------------------------------------------------------------------
Fitted distribution (uncensored) :             Normal(mu = 6.01403e-13, sigma = 289.
↪998)

                                                     Uncensored
Distribution fitting test
Kolmogorov p-value (> 0.05):                                0.8

Normality test
Anderson Darling p-value (> 0.05):                         0.07
Cramer Von Mises p-value (> 0.05):                         0.09

Zero residual mean test
p-value (> 0.05):                                           1.0

Homoskedasticity test (constant variance)
Breush Pagan p-value (> 0.05):                              0.0
Harrison McCabe p-value (> 0.05):                           0.2

Non autocorrelation test
Durbin Watson p-value (> 0.05):                            0.99
--------------------------------------------------------------------------------

Warning : Some hypothesis tests failed : you may consider to use the Box Cox␣
↪transformation.
```

Show graphs

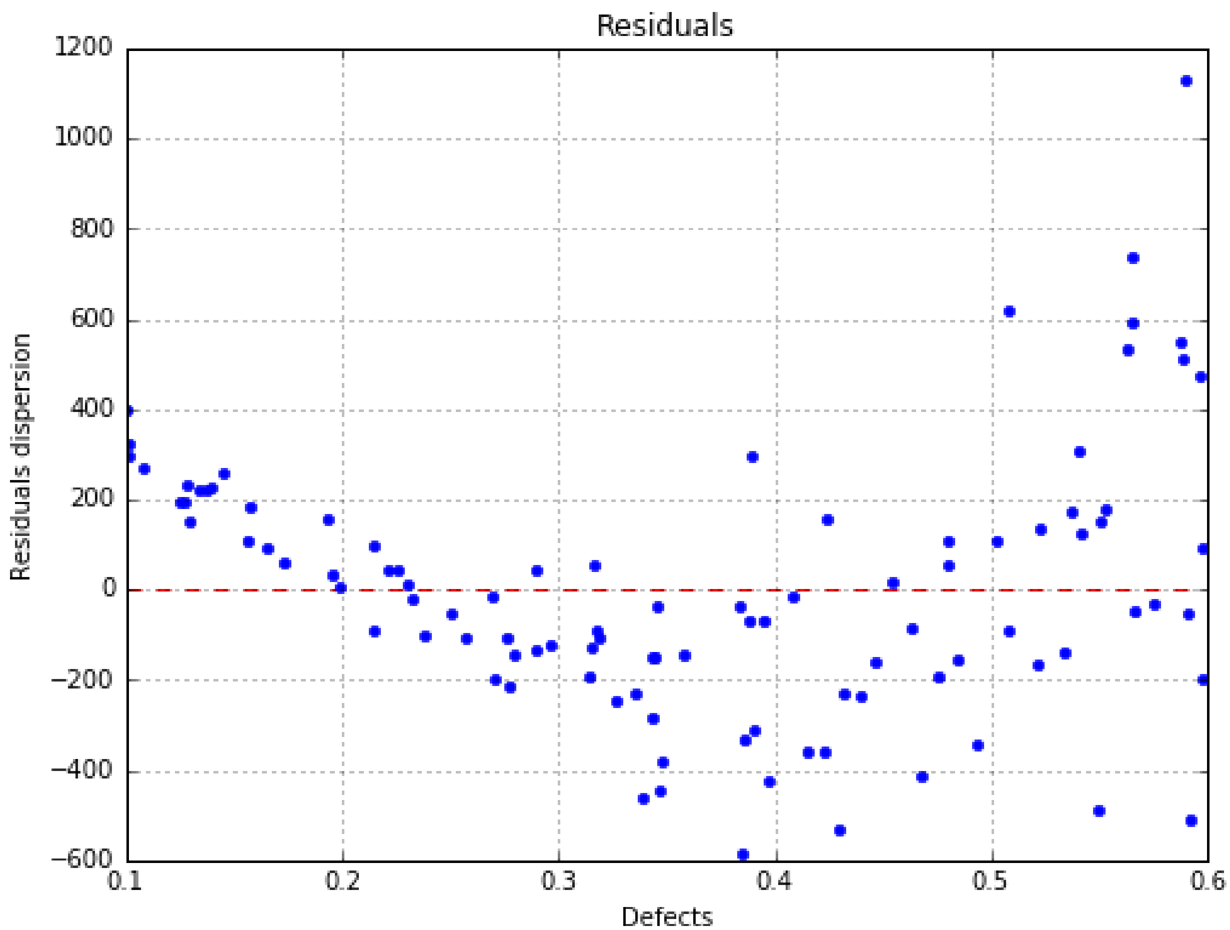
The linear model is not correct

```
fig, ax = analysis.drawLinearModel()
fig.show()
```



The residuals are not homoskedastic

```
fig, ax = analysis.drawResiduals()
fig.show()
```



Run analysis with Box Cox

```
analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, boxCox=True)
```

Print results of the linear regression and all tests on the residuals

```
print(analysis.getResults())
```

```
--------------------------------------------------------------------------------
         Linear model analysis results
--------------------------------------------------------------------------------
Box Cox parameter :                                        0.22

                                                     Uncensored

Intercept coefficient :                                    4.02
Slope coefficient :                                       25.55
Standard error of the estimate :                           1.34

Confidence interval on coefficients
Intercept coefficient :                            [3.33, 4.72]
```

```
Slope coefficient :                                [23.80, 27.31]
Level :                                                     0.95

Quality of regression
R2 (> 0.8):                                                 0.89
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
         Residuals analysis results
--------------------------------------------------------------------------------
Fitted distribution (uncensored) :              Normal(mu = 4.15668e-15, sigma = 1.
↪32901)

                                                   Uncensored
Distribution fitting test
Kolmogorov p-value (> 0.05):                                0.34

Normality test
Anderson Darling p-value (> 0.05):                          0.06
Cramer Von Mises p-value (> 0.05):                          0.07

Zero residual mean test
p-value (> 0.05):                                            1.0

Homoskedasticity test (constant variance)
Breush Pagan p-value (> 0.05):                              0.65
Harrison McCabe p-value (> 0.05):                           0.51

Non autocorrelation test
Durbin Watson p-value (> 0.05):                             0.97
--------------------------------------------------------------------------------
```

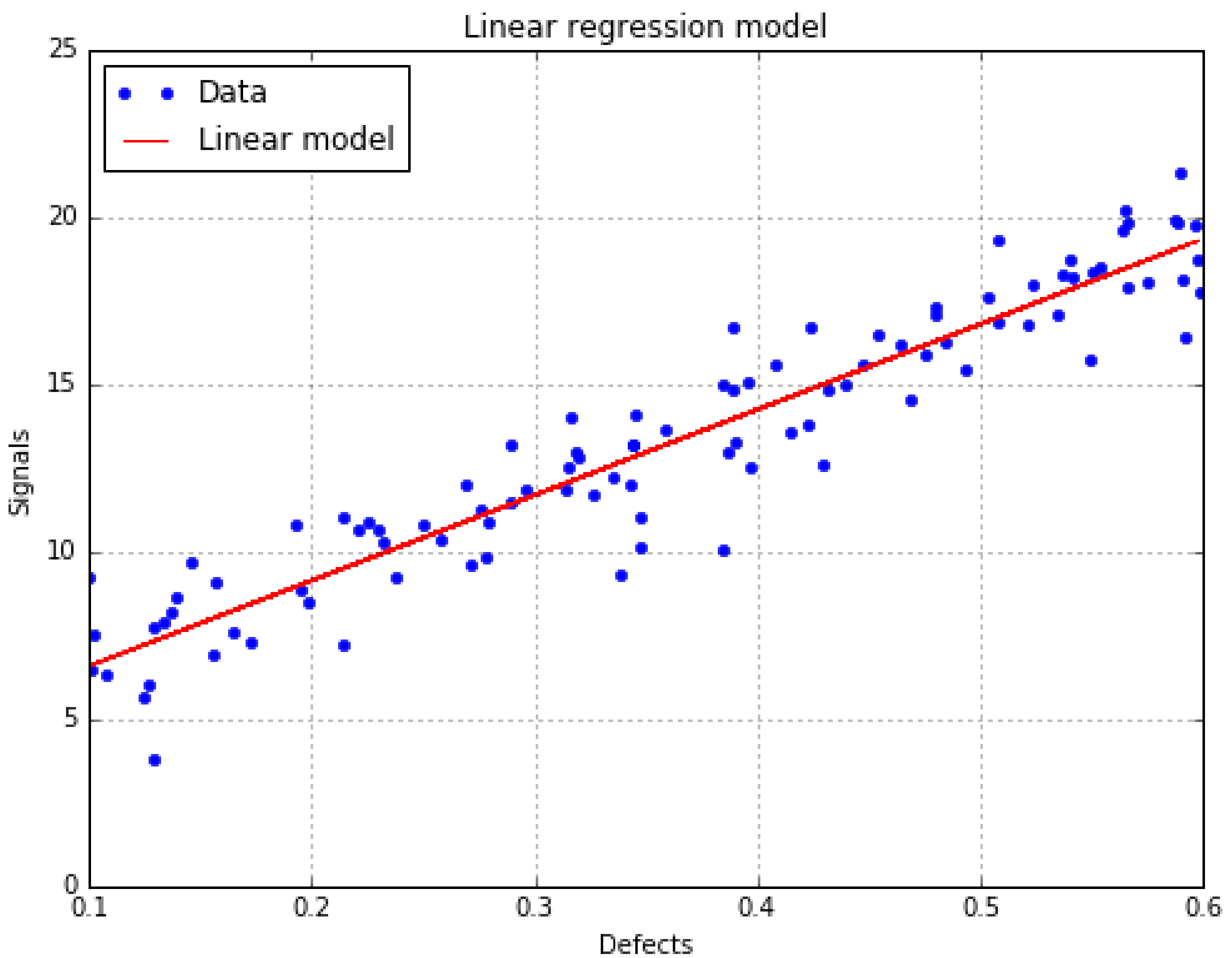
Save all results in a csv file

```
analysis.saveResults('results.csv')
```

Show graphs

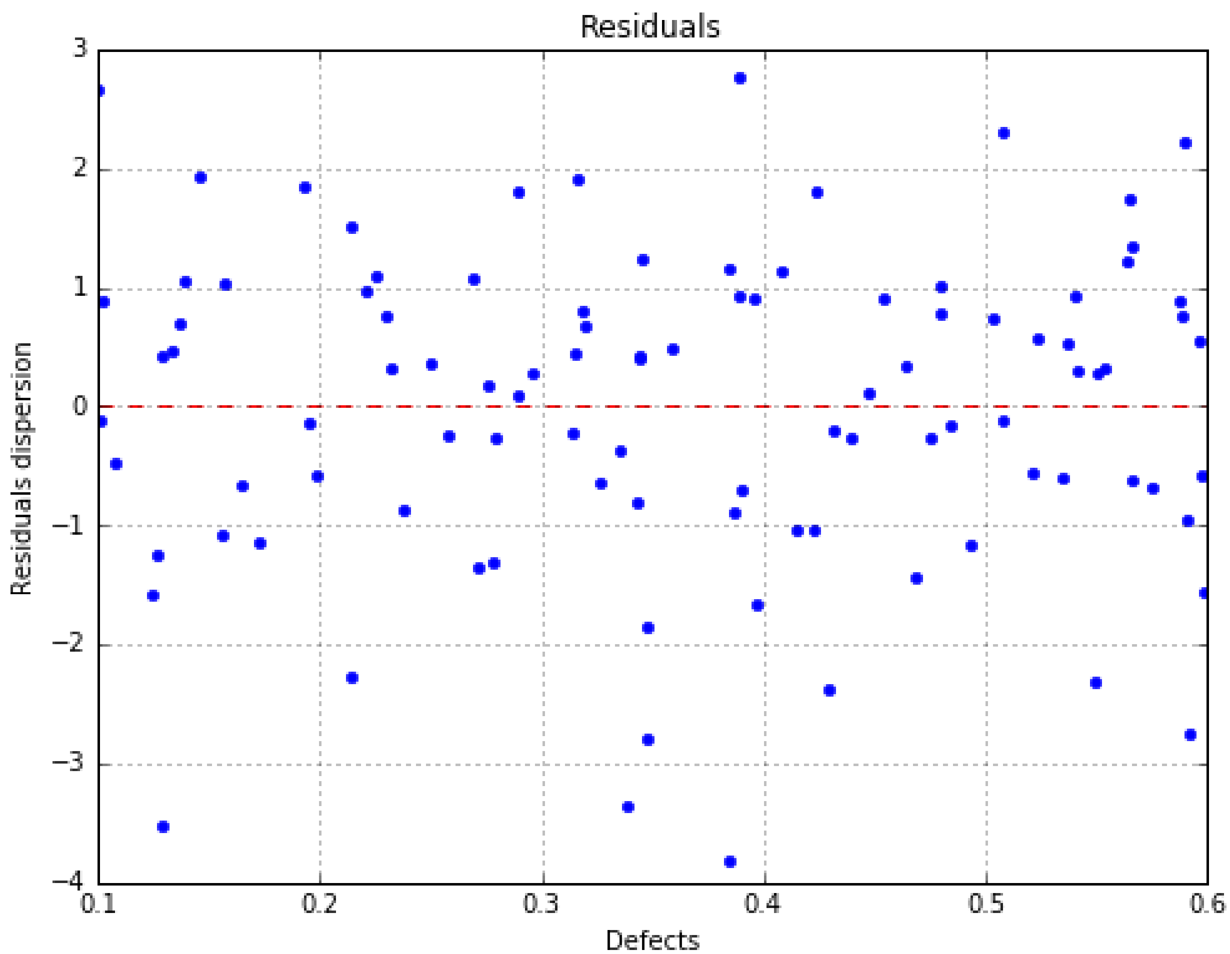
The linear regression model with data

```
fig, ax = analysis.drawLinearModel(name='figure/linearModel.png')
# The figure is saved as png file
fig.show()
```



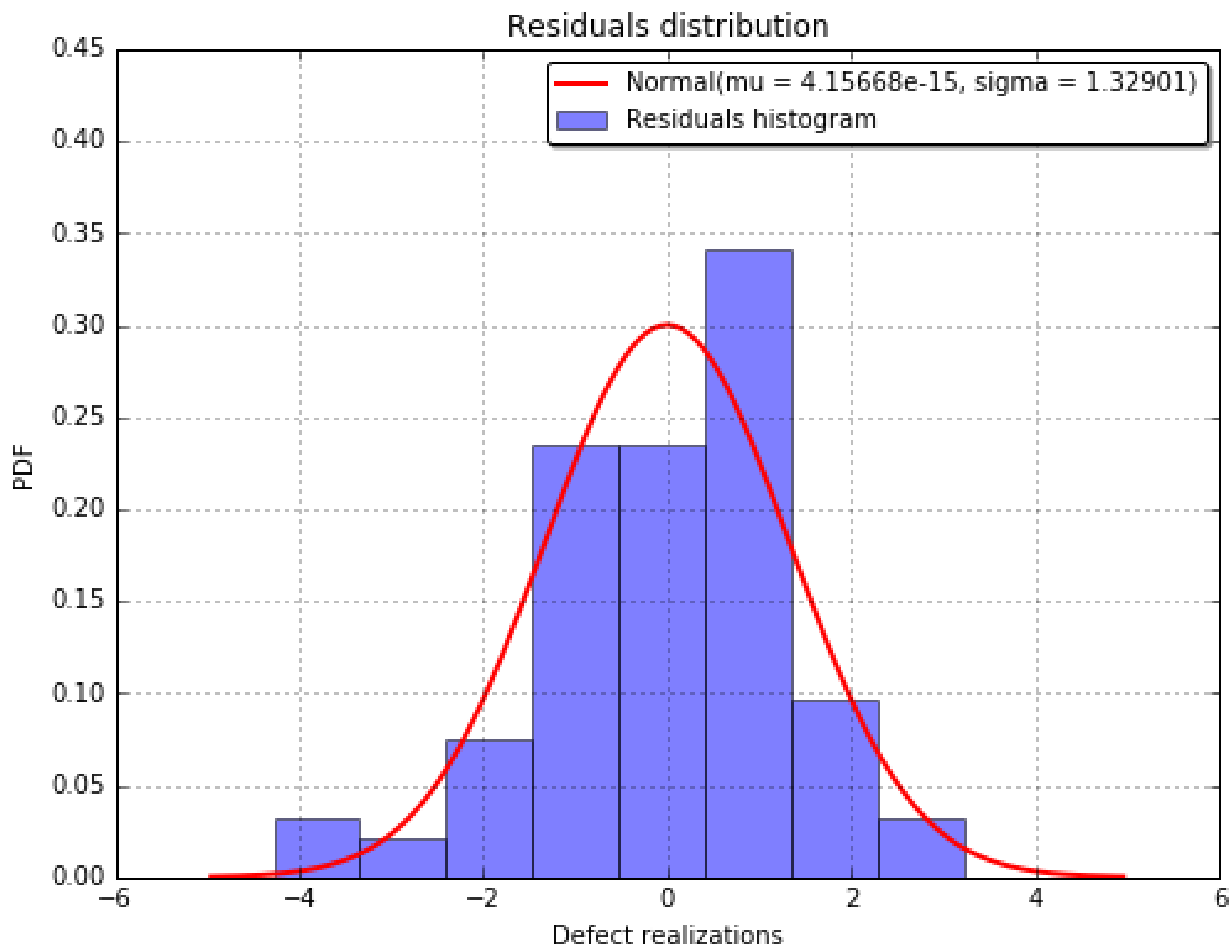
The residuals with respect to the defects

```
fig, ax = analysis.drawResiduals(name='figure/residuals.eps')
# The figure is saved as eps file
fig.show()
```



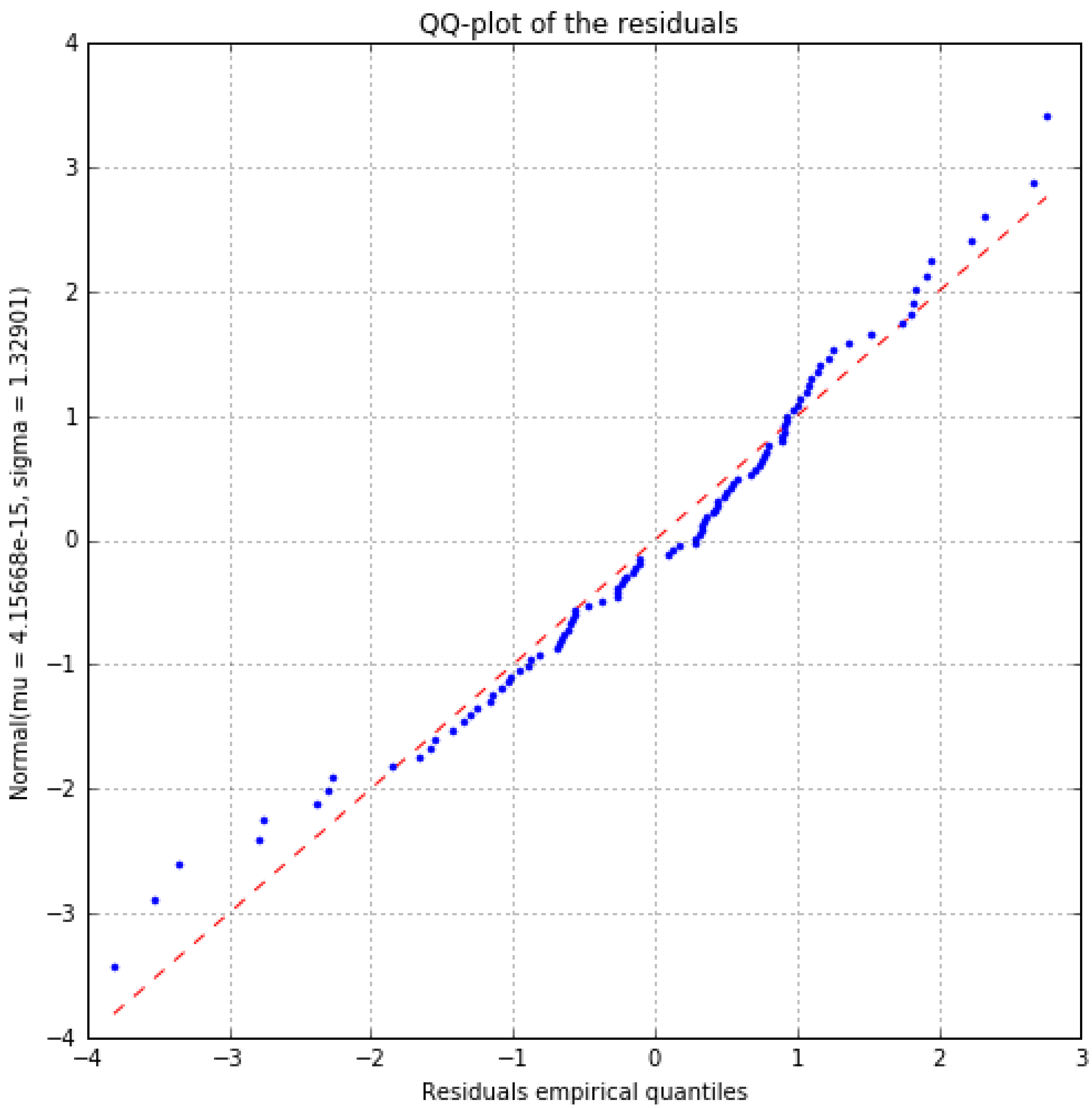
The fitted residuals distribution with the histogram

```
fig, ax = analysis.drawResidualsDistribution()
ax.set_ylim(ymax=0.45)
fig.show()
# The figure is saved after the changes
fig.savefig('figure/residualsDistribution.png', bbox_inches='tight')
```



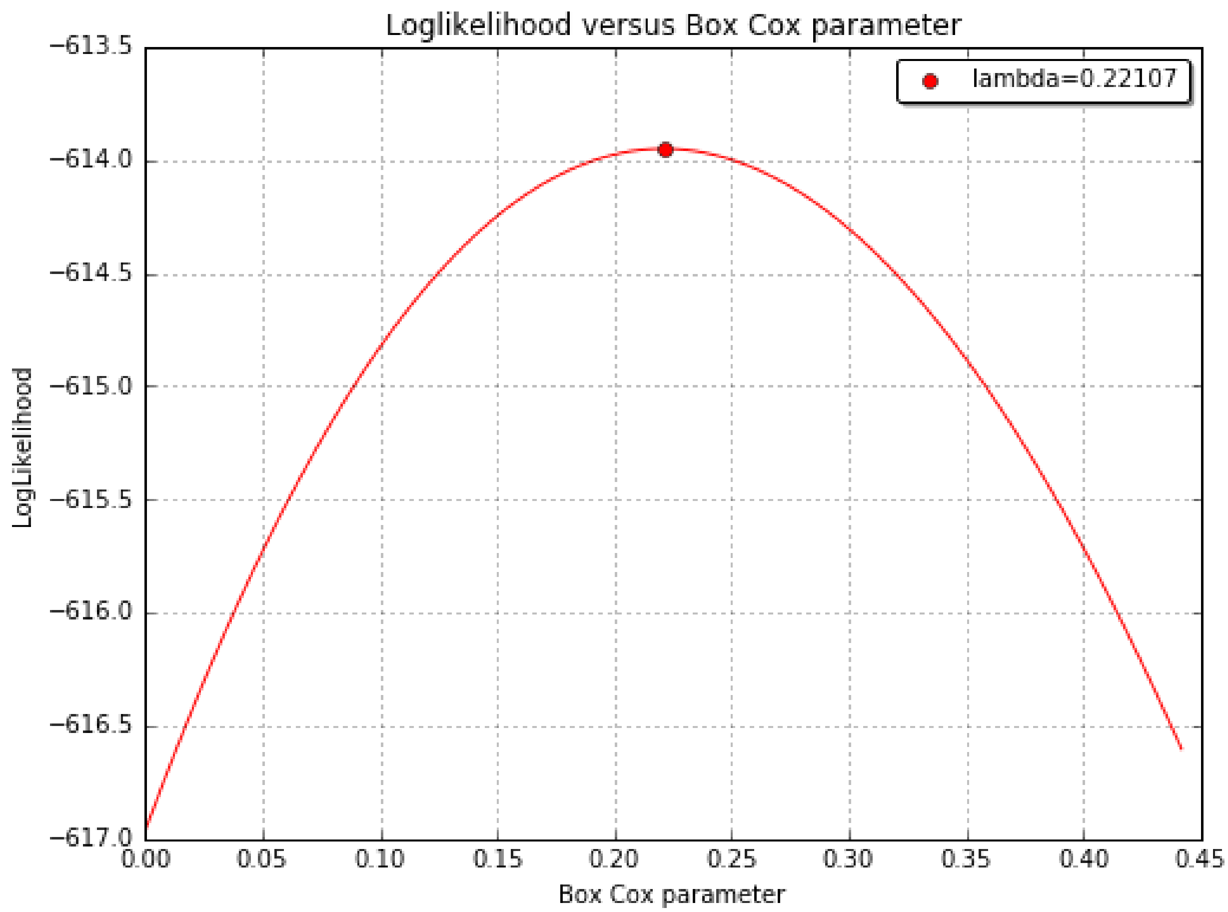
The residuals QQ plot

```
fig, ax = analysis.drawResidualsQQplot()
fig.show()
```



The Box Cox likelihood with respect to the defect

```
fig, ax = analysis.drawBoxCoxLikelihood(name='figure/BoxCoxlikelihood.png')
fig.show()
```



ipynb source code

1.2.2 Linear model analysis with censored data

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
N = 100
ot.RandomGenerator.SetSeed(123456)
defectDist = ot.Uniform(0.1, 0.6)
# normal epsilon distribution
epsilon = ot.Normal(0, 1.9)
defects = defectDist.getSample(N)
signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
# Inverse Box Cox transformation
invBoxCox = ot.InverseBoxCoxTransform(0.3)
signals = invBoxCox(signalsInvBoxCox)
```

Run analysis with Box Cox

```
noiseThres = 60.
saturationThres = 1700.
analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, noiseThres,
                                               saturationThres, boxCox=True)
```

Get some particular results

Result values are given for both analysis performed on filtered data (uncensored case) and on censored data.

```
print(analysis.getIntercept())
print(analysis.getR2())
print(analysis.getKolmogorovPValue())
```

```
[Intercept for uncensored case : 4.777, Intercept for censored case : 4.1614]
[R2 for uncensored case : 0.869115, R2 for censored case : 0.860722]
[Kolmogorov p-value for uncensored case : 0.477505, Kolmogorov p-value for censored␣
↪case : 0.505919]
```

Print all results of the linear regression and all tests on the residuals

```
# Results are displayed for both case
print(analysis.getResults())
```

```
--------------------------------------------------------------------------------
         Linear model analysis results
--------------------------------------------------------------------------------
Box Cox parameter :                                         0.18

                                                      Uncensored       Censored

Intercept coefficient :                                     4.78          4.16
Slope coefficient :                                        18.15         19.94
Standard error of the estimate :                            0.97          1.03

Confidence interval on coefficients
Intercept coefficient :                              [4.19, 5.36]
Slope coefficient :                                 [16.63, 19.67]
Level :                                                     0.95

Quality of regression
R2 (> 0.8):                                                 0.87          0.86
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
         Residuals analysis results
--------------------------------------------------------------------------------
Fitted distribution (uncensored) :                 Normal(mu = -2.19492e-15, sigma = 0.
↪968046)
Fitted distribution (censored) :                   Normal(mu = -0.0237409, sigma = 0.
↪998599)

                                                      Uncensored       Censored
```

```
Distribution fitting test
Kolmogorov p-value (> 0.05):                          0.48          0.51

Normality test
Anderson Darling p-value (> 0.05):                    0.06          0.08
Cramer Von Mises p-value (> 0.05):                    0.07          0.09

Zero residual mean test
p-value (> 0.05):                                      1.0          0.83

Homoskedasticity test (constant variance)
Breush Pagan p-value (> 0.05):                        0.69          0.71
Harrison McCabe p-value (> 0.05):                      0.6          0.51

Non autocorrelation test
Durbin Watson p-value (> 0.05):                       0.43          0.48
--------------------------------------------------------------------------------
```

Save all results in a csv file

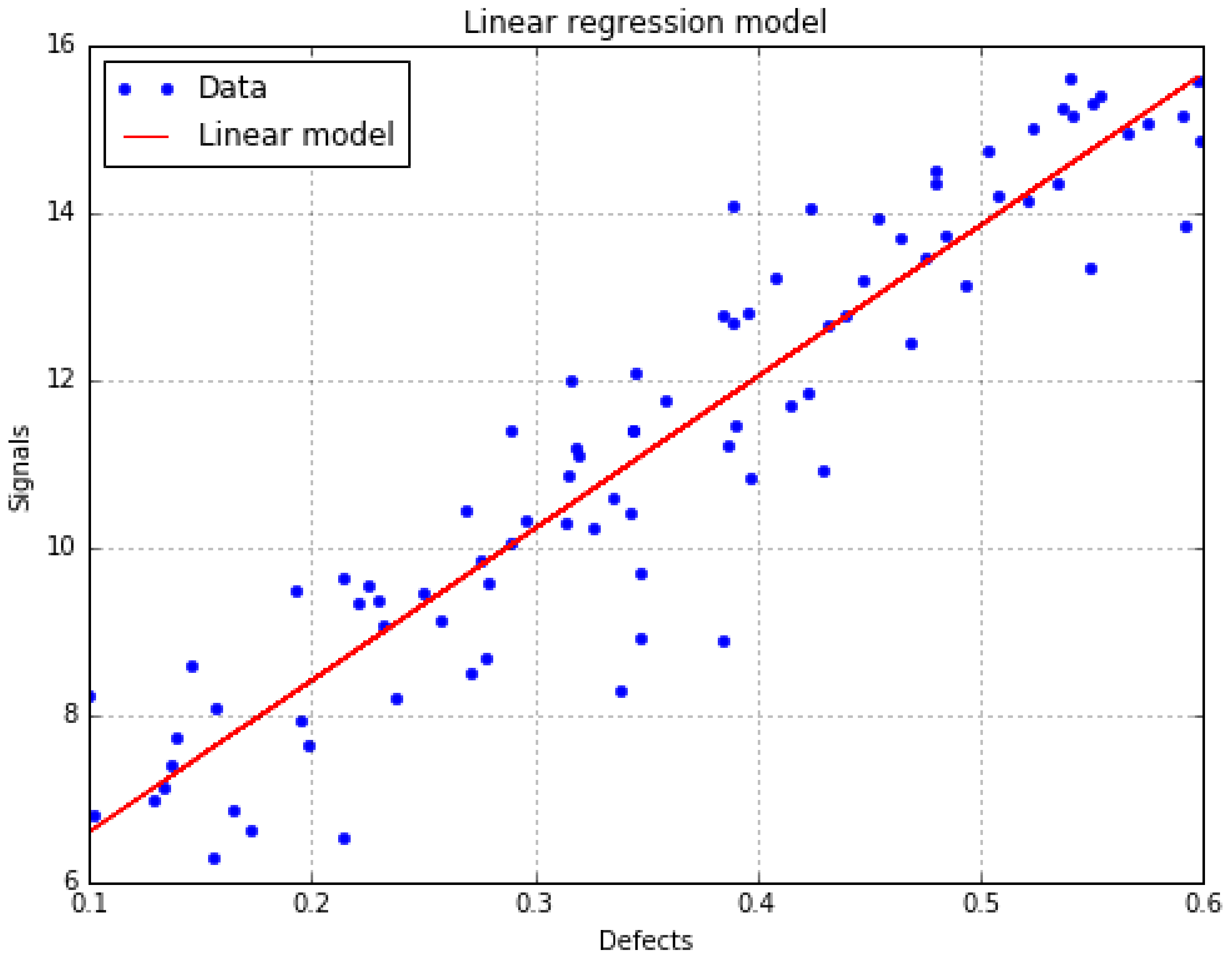
```
analysis.saveResults('results.csv')
```

Show graphs

The linear regression model with data for the uncensored case (default case)

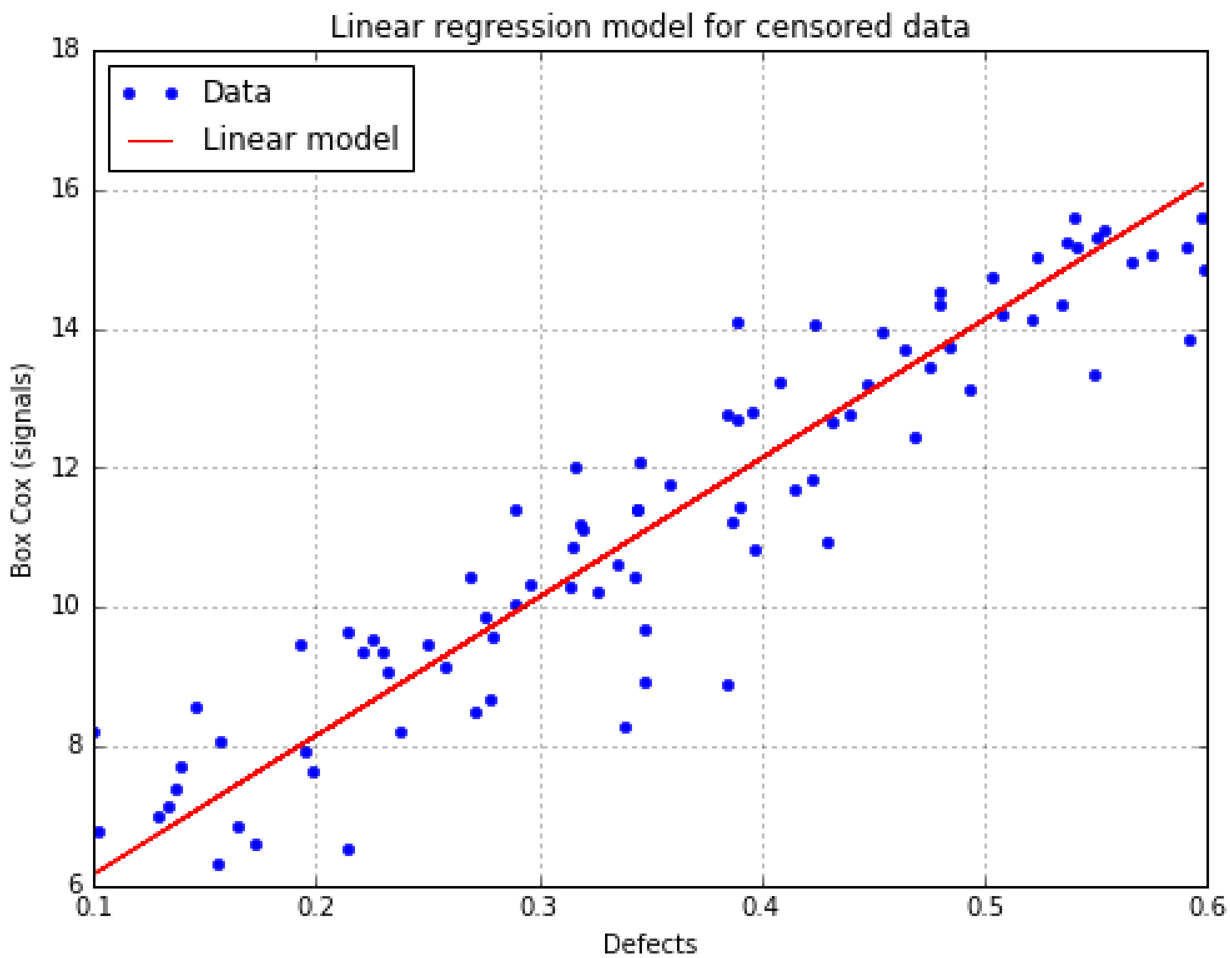
```
# draw the figure for the uncensored case and save it as png file
fig, ax = analysis.drawLinearModel(name='figure/linearModelUncensored.png')
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



The linear regression model with data for the censored case

```
# draw the figure for the censored case and save it as png file
fig, ax = analysis.drawLinearModel(model='censored', name='figure/linearModelCensored.
↪png')
fig.show()
```



ipynb source code

1.2.3 Linear model POD

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
N = 100
ot.RandomGenerator.SetSeed(123456)
defectDist = ot.Uniform(0.1, 0.6)
# normal epsilon distribution
epsilon = ot.Normal(0, 1.9)
defects = defectDist.getSample(N)
signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
# Inverse Box Cox transformation
```

```
invBoxCox = ot.InverseBoxCoxTransform(0.3)
signals = invBoxCox(signalsInvBoxCox)
```

Build POD using previous linear analysis

```
# run the analysis with Gaussian hypothesis of the residuals (default case)
analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, boxCox=True)
```

```
# signal detection threshold
detection = 200.
# Use the analysis to build the POD with Gaussian hypothesis
# keyword arguments must be given
PODGauss = otpod.UnivariateLinearModelPOD(analysis=analysis, detection=detection)
PODGauss.run()
```

Build POD with Gaussian hypothesis

```
# The previous POD is equivalent to the following POD
PODGauss = otpod.UnivariateLinearModelPOD(defects, signals, detection,
                                          resDistFact=ot.NormalFactory(),
                                          boxCox=True)
PODGauss.run()
```

Get the R2 value of the regression

```
print('R2 : {:0.3f}'.format(PODGauss.getR2()))
```

```
R2 : 0.895
```

Compute detection size

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(PODGauss.computeDetectionSize(0.9, 0.95))

# probability level 0.95 with confidence level 0.99
print(PODGauss.computeDetectionSize(0.95, 0.99))
```

```
[a90 : 0.303982, a90/95 : 0.317157]
[a95 : 0.323048, a95/99 : 0.343536]
```

get POD NumericalMathFunction

```
# get the POD model
PODmodel = PODGauss.getPODModel()
# get the POD model at the given confidence level
PODmodelCl95 = PODGauss.getPODCLModel(0.95)
```

```
# compute the probability of detection for a given defect value
print('POD : {:0.3f}'.format(PODmodel([0.3])[0]))
print('POD at level 0.95 : {:0.3f}'.format(PODmodelCl95([0.3])[0]))
```

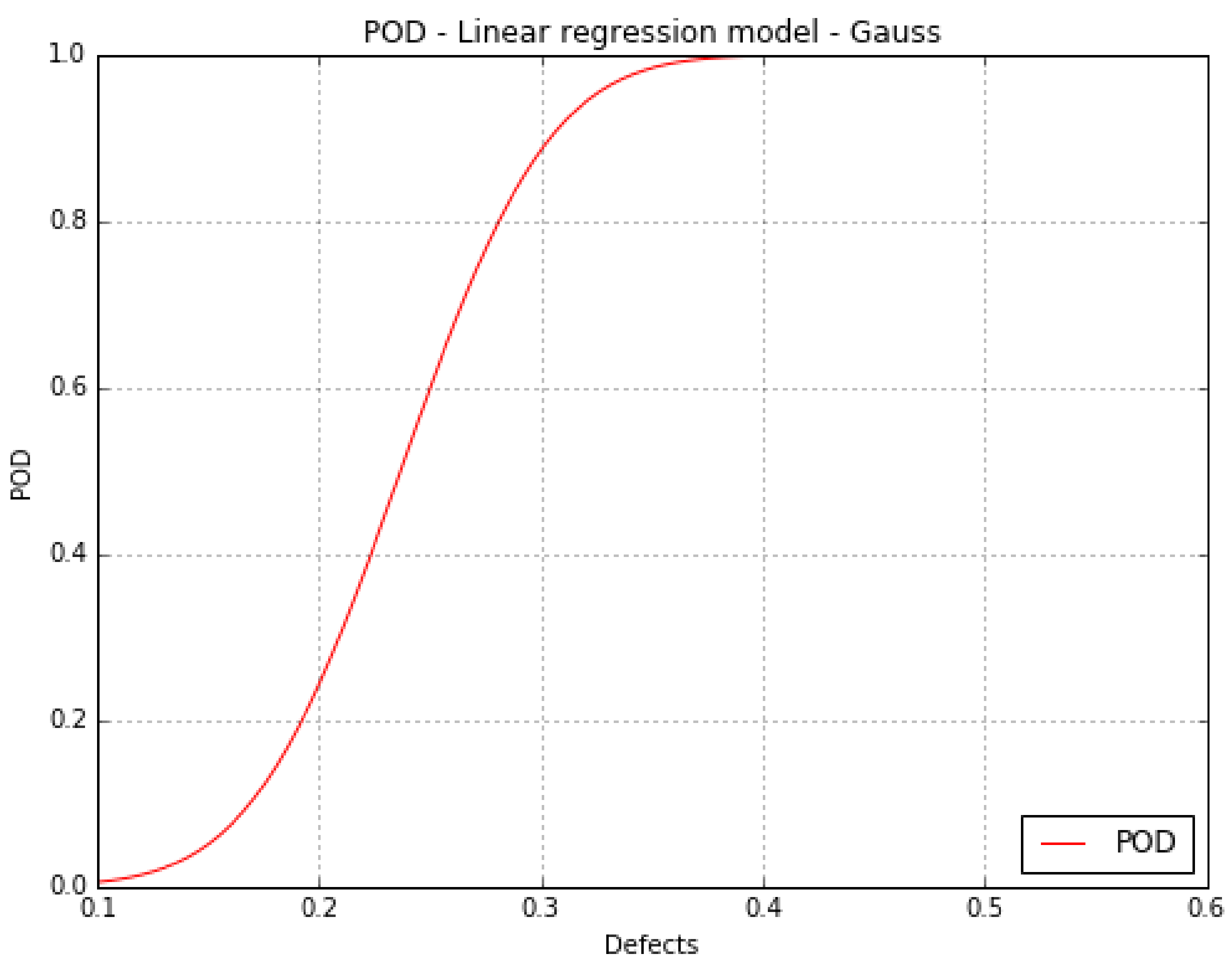
```
POD : 0.886
POD at level 0.95 : 0.834
```

Show POD graphs

Only the mean POD

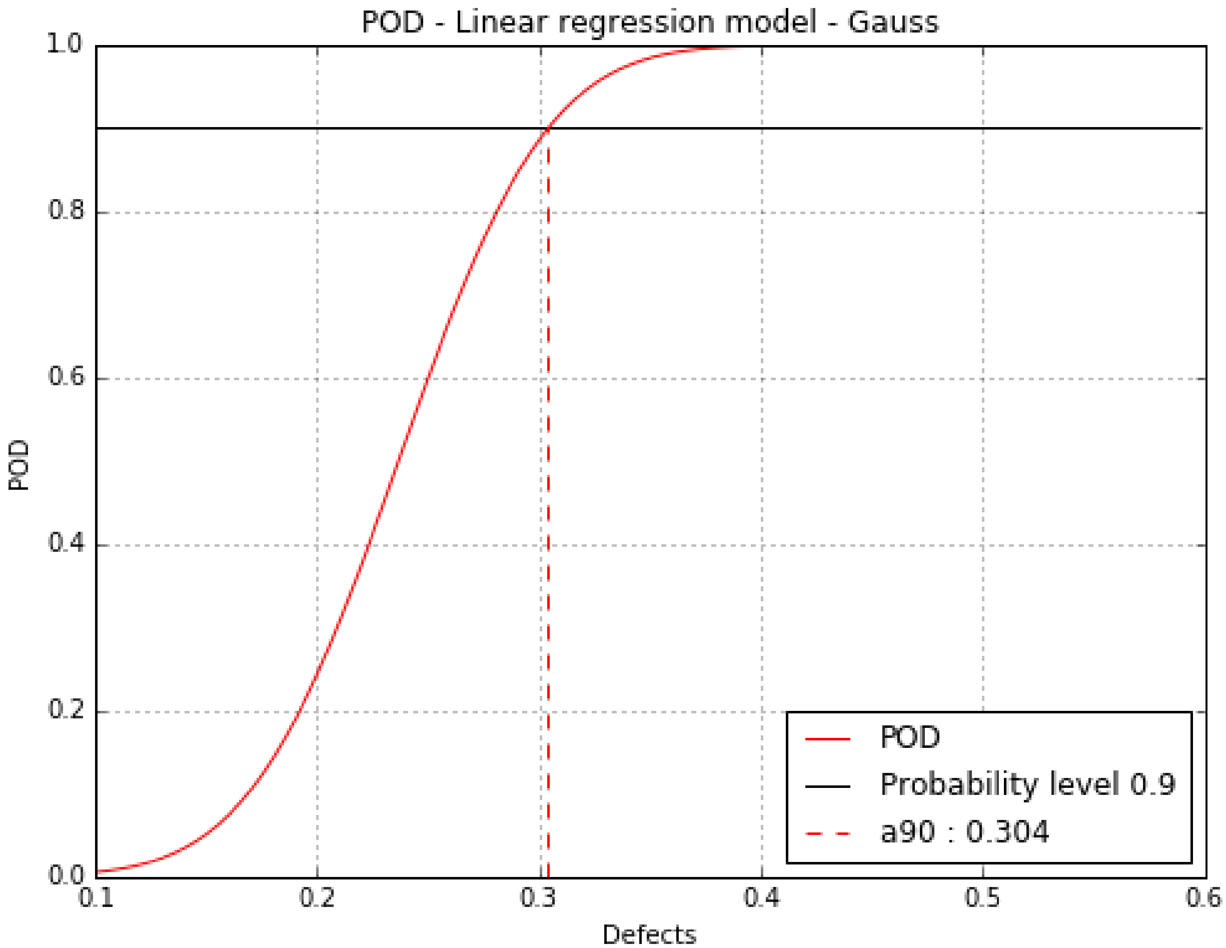
```
fig, ax = PODGauss.drawPOD()
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



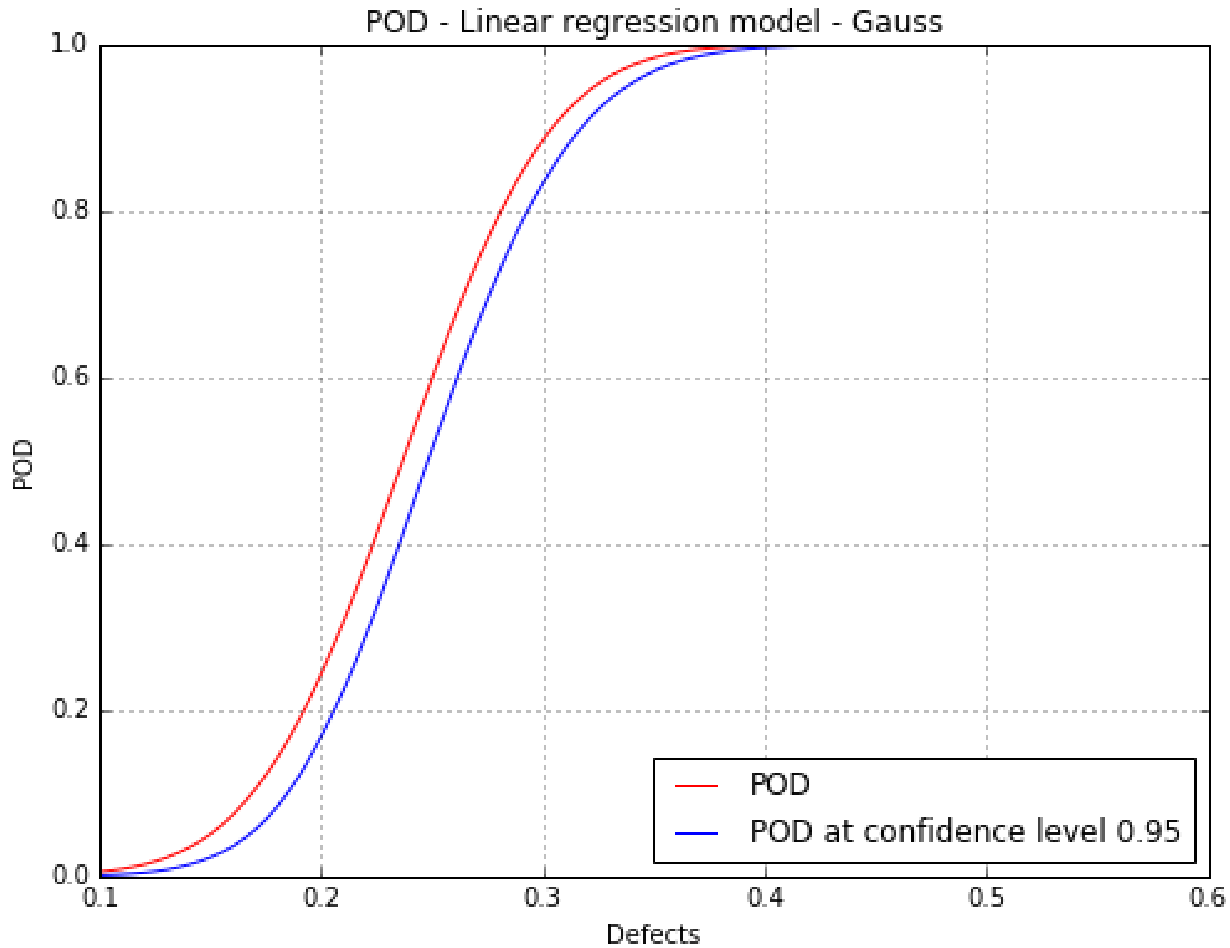
Mean POD with the detection size for a given probability level

```
fig, ax = PODGauss.drawPOD(probabilityLevel=0.9)
fig.show()
```



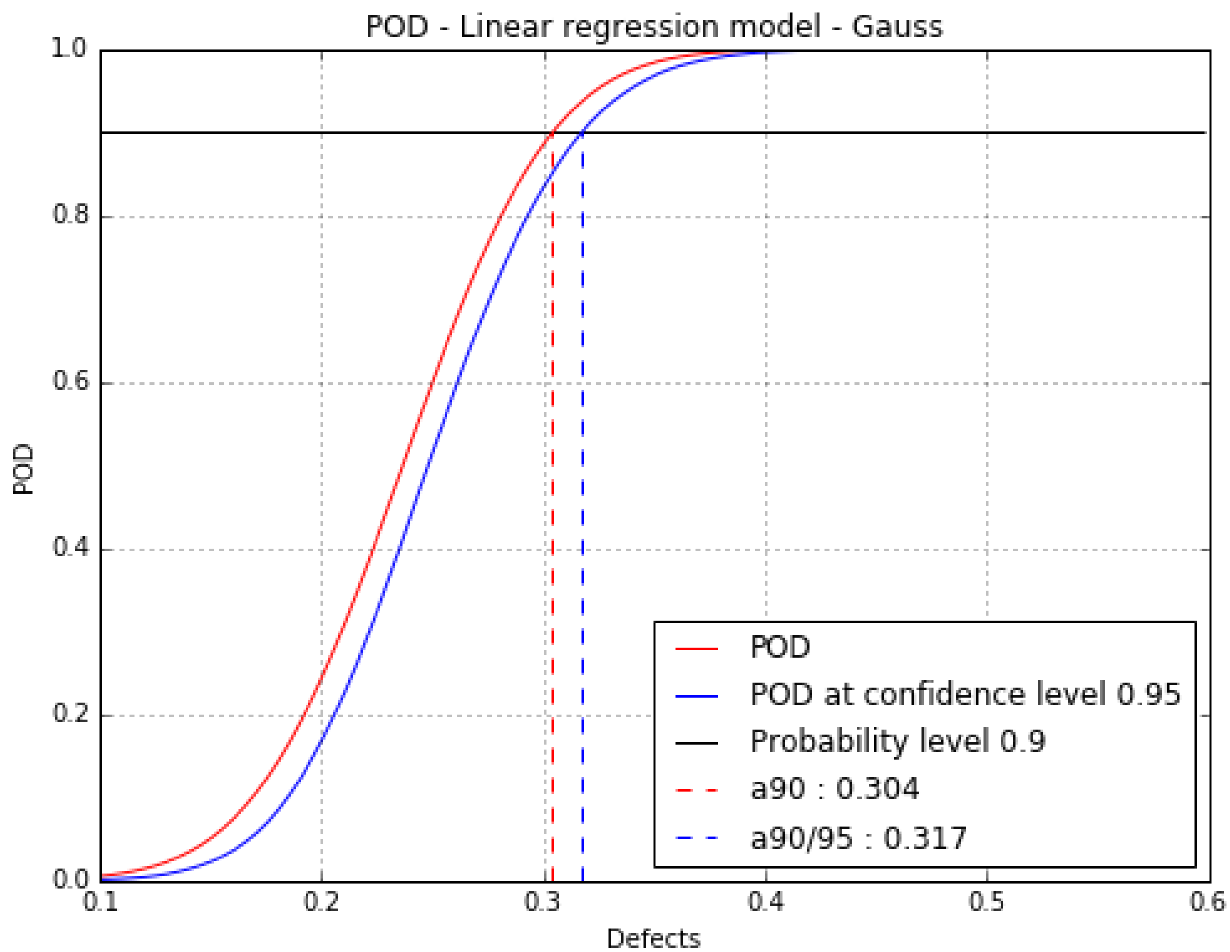
Mean POD with POD at confidence level

```
fig, ax = PODGauss.drawPOD(confidenceLevel=0.95)
fig.show()
```



Mean POD and POD at confidence level with the detection size for a given probability level

```
fig, ax = PODGauss.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODGauss.png')
# The figure is saved in PODGauss.png
fig.show()
```



Build POD with no hypothesis on the residuals

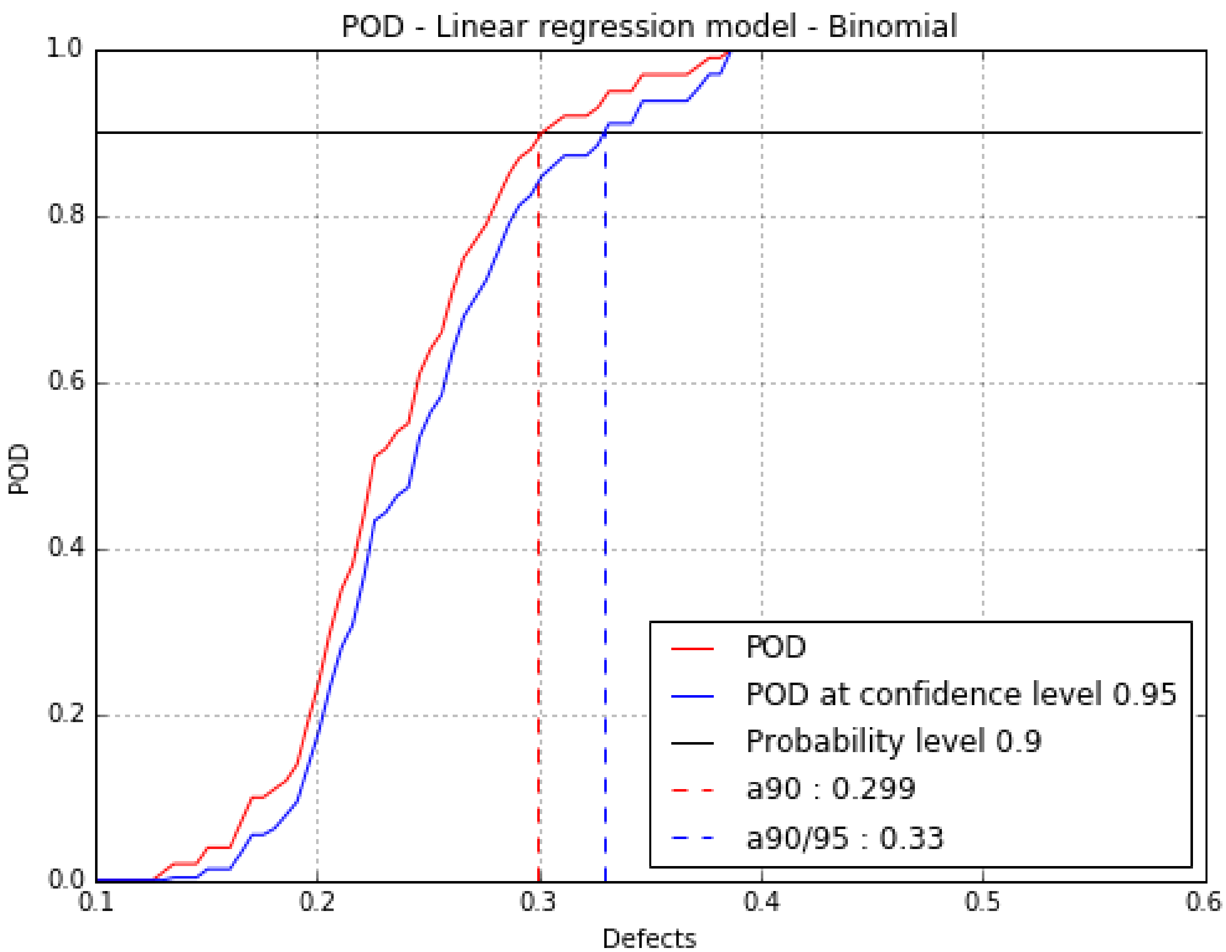
This corresponds with the Berens Binomial method.

```
PODBinomial = otpod.UnivariateLinearModelPOD(defects, signals, detection, boxCox=True)
PODBinomial.run()
```

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(PODBinomial.computeDetectionSize(0.9, 0.95))
```

```
[a90 : 0.298739, a90/95 : 0.329606]
```

```
fig, ax = PODBinomial.drawPOD(0.9, 0.95)
fig.show()
```



Build POD with kernel smoothing on the residuals

The POD at the given confidence level is built using bootstrap. It may take few seconds. A progress bar if displayed is in this case. It can be remove using setVerbose(False)

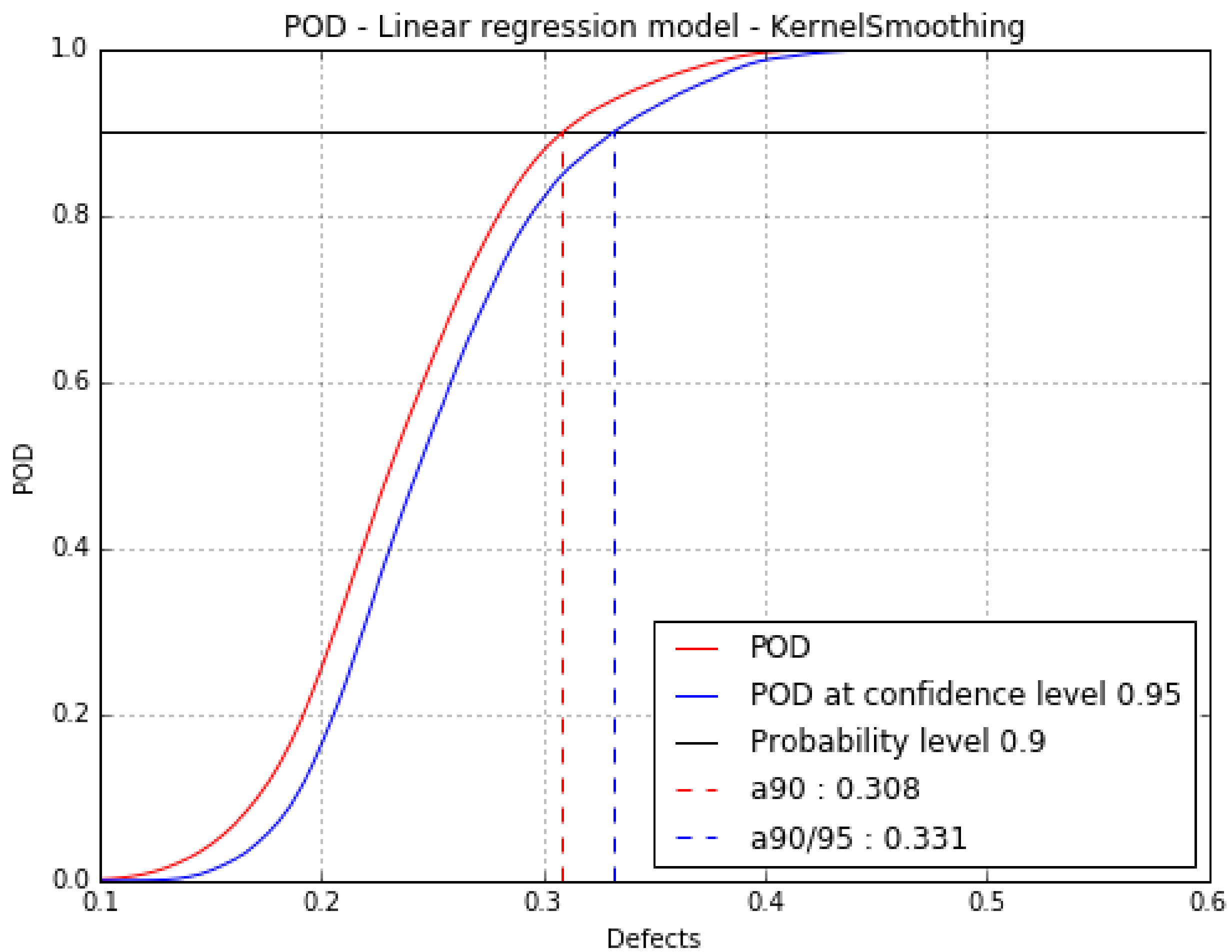
```
PODks = otpod.UnivariateLinearModelPOD(defects, signals, detection,
                                        resDistFact=ot.KernelSmoothing(),
                                        boxCox=True)
PODks.run()
```

```
Computing POD (bootstrap): [==================================================] 100.00
↪% Done
```

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(PODks.computeDetectionSize(0.9, 0.95))
```

```
[a90 : 0.308381, a90/95 : 0.331118]
```

```
fig, ax = PODks.drawPOD(0.9, 0.95)
fig.show()
```



ipynb source code

1.2.4 Linear model POD with censored data

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
N = 100
ot.RandomGenerator.SetSeed(123456)
defectDist = ot.Uniform(0.1, 0.6)
# normal epsilon distribution
epsilon = ot.Normal(0, 1.9)
defects = defectDist.getSample(N)
signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
# Inverse Box Cox transformation
```

```
invBoxCox = ot.InverseBoxCoxTransform(0.3)
signals = invBoxCox(signalsInvBoxCox)
```

Build POD using previous linear analysis

```
noiseThres = 60.
saturationThres = 1700.

# run the analysis with Gaussian hypothesis of the residuals (default case)
analysis = otpod.UnivariateLinearModelAnalysis(defects, signals, noiseThres,
                                               saturationThres, boxCox=True)
```

```
# signal detection threshold
detection = 200.
# Use the analysis to build the POD with Gaussian hypothesis
# keyword arguments must be given
PODGauss = otpod.UnivariateLinearModelPOD(analysis=analysis, detection=detection)
PODGauss.run()
```

Build POD with Gaussian hypothesis

```
# The previous POD is equivalent to the following POD
PODGauss = otpod.UnivariateLinearModelPOD(defects, signals, detection,
                                          noiseThres, saturationThres,
                                          resDistFact=ot.NormalFactory(),
                                          boxCox=True)
PODGauss.run()
```

Get the R2 value of the regression

```
print('R2 : {:0.3f}'.format(PODGauss.getR2()))
```

```
R2 : 0.861
```

Compute detection size

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(PODGauss.computeDetectionSize(0.9, 0.95))
```

```
[a90 : 0.30373, a90/95 : 0.317848]
```

get POD NumericalMathFunction

```
# get the POD model
PODmodel = PODGauss.getPODModel()
# get the POD model at the given confidence level
```

```
PODmodelCl95 = PODGauss.getPODCLModel(0.95)

# compute the probability of detection for a given defect value
print('POD : {:0.3f}'.format(PODmodel([0.3])[0]))
print('POD at level 0.95 : {:0.3f}'.format(PODmodelCl95([0.3])[0]))
```

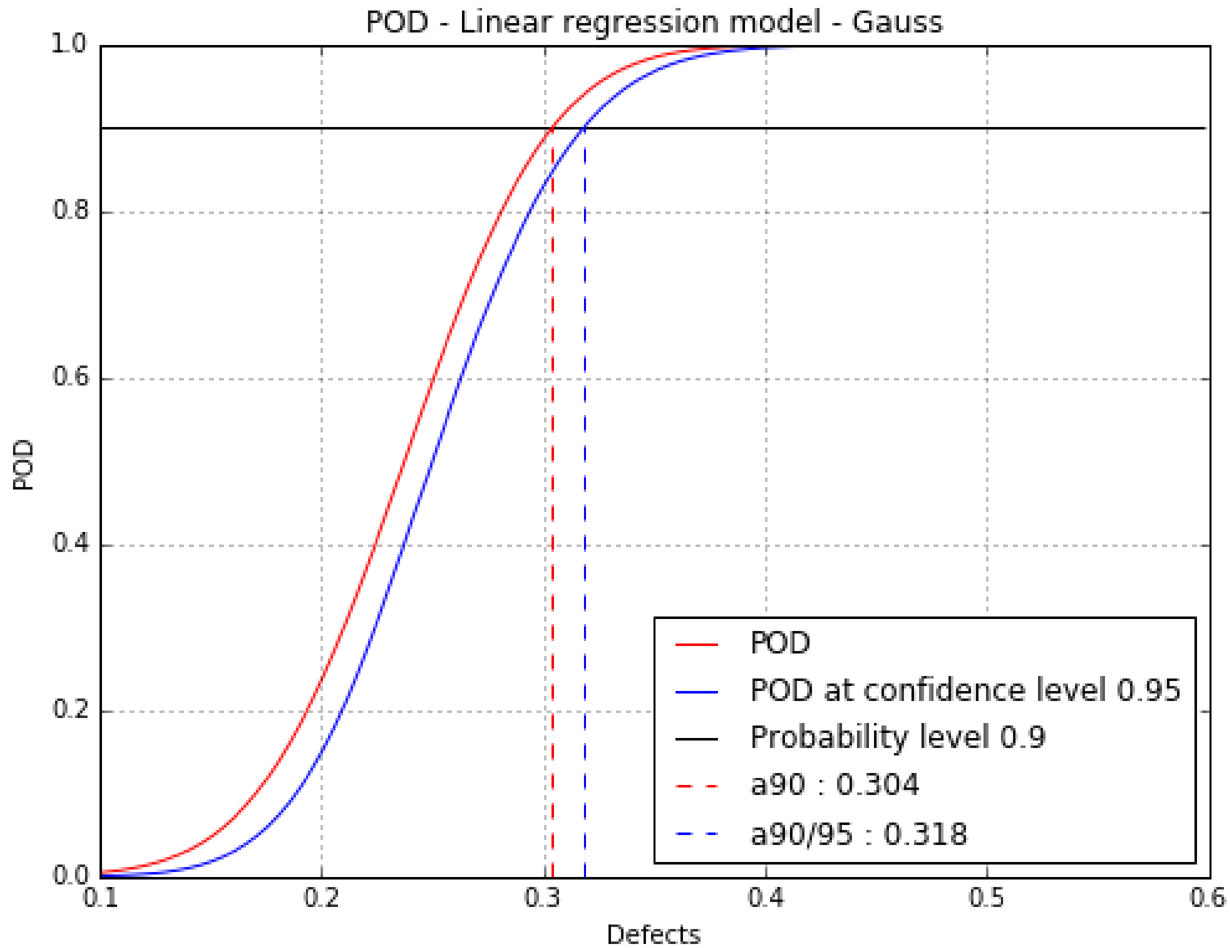
```
POD : 0.887
POD at level 0.95 : 0.830
```

Show POD graph

Mean POD and POD at confidence level with the detection size for a given probability level

```
fig, ax = PODGauss.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODGaussCensored.png')
# The figure is saved in PODGauss.png
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



Build POD only with the filtered data

A static method is used to get the defects and signals only in the uncensored area.

```
print(otpod.DataHandling.filterCensoredData.__doc__)
```

```
Sort inputSample and signals with respect to the censore thresholds.

Parameters
----------
inputSample : 2-d sequence of float
    Vector of the input sample.
signals : 2-d sequence of float
    Vector of the signals, of dimension 1.
noiseThres : float
    Value for low censored data. Default is None.
saturationThres : float
    Value for high censored data. Default is None

Returns
-------
inputSampleUnc : 2-d sequence of float
    Vector of the input sample in the uncensored area.
```

```
inputSampleNoise : 2-d sequence of float
    Vector of the input sample in the noisy area.
inputSampleSat : 2-d sequence of float
    Vector of the input sample in the saturation area.
signalsUnc : 2-d sequence of float
    Vector of the signals in the uncensored area.

Notes
-----
The data are sorted in three different vectors whether they belong to
the noisy area, the uncensored area or the saturation area.
```

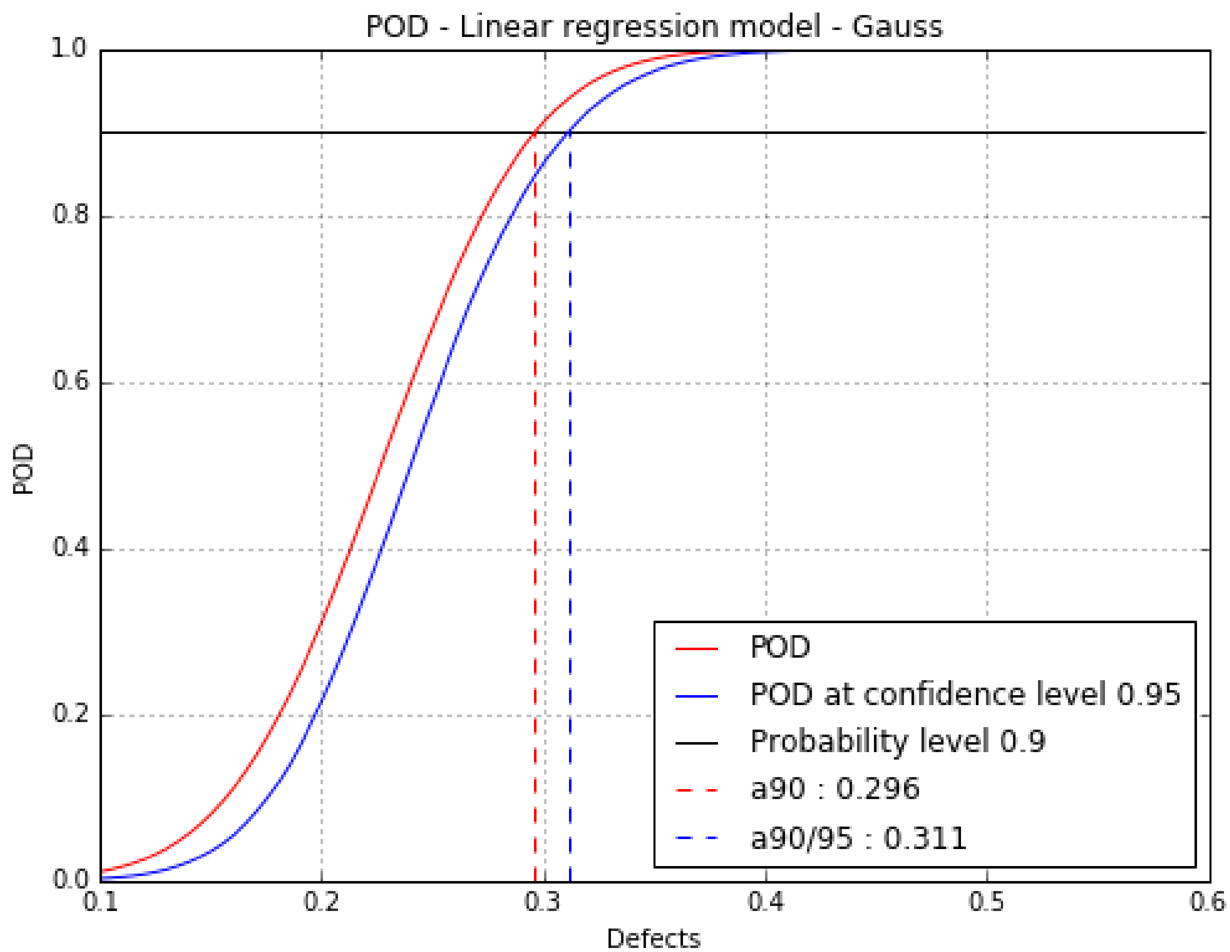
```
result = otpod.DataHandling.filterCensoredData(defects, signals,
                                               noiseThres, saturationThres)
defectsFiltered = result[0]
signalsFiltered = result[3]
```

```
PODfilteredData = otpod.UnivariateLinearModelPOD(defectsFiltered, signalsFiltered,
                                                 detection,
                                                 resDistFact=ot.NormalFactory(),
                                                 boxCox=True)
PODfilteredData.run()
```

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(PODfilteredData.computeDetectionSize(0.9, 0.95))
```

```
[a90 : 0.295976, a90/95 : 0.310948]
```

```
fig, ax = PODfilteredData.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODGaussFiltered.png')
# The figure is saved in PODGauss.png
fig.show()
```



ipynb source code

1.2.5 Quantile Regression POD

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
from time import time
```

Generate data

```
N = 100
ot.RandomGenerator.SetSeed(123456)
defectDist = ot.Uniform(0.1, 0.6)
# normal epsilon distribution
epsilon = ot.Normal(0, 1.9)
defects = defectDist.getSample(N)
signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
# Inverse Box Cox transformation
```

```
invBoxCox = ot.InverseBoxCoxTransform(0.3)
signals = invBoxCox(signalsInvBoxCox)
```

Build POD with quantile regression technique

```
# signal detection threshold
detection = 200.
# The POD with censored data actually builds a POD only on filtered data.
# A warning is diplayed in this case.
POD = otpod.QuantileRegressionPOD(defects, signals, detection,
                                  noiseThres=60., saturationThres=1700.,
                                  boxCox=True)
```

Quantile user-defined

```
# Default quantile values
print('Default quantile : ')
print(POD.getQuantile())
# Defining user quantile, they must range between 0 and 1.
POD.setQuantile([0.1, 0.3, 0.5, 0.7, 0.8, 0.85, 0.9, 0.95])
print('User-defined quantile : ')
print(POD.getQuantile())
```

```
Default quantile :
[ 0.05     0.0965   0.143    0.1895   0.236    0.2825   0.329    0.3755   0.422
  0.4685   0.515    0.5615   0.608    0.6545   0.701    0.7475   0.794    0.8405
  0.887    0.9335   0.98   ]
User-defined quantile :
[ 0.1    0.3    0.5    0.7    0.8    0.85   0.9    0.95]
```

Running quantile regression POD

```
# Due to the bootstrap technique used to compute the confidence
# interval, the run takes few minutes.
# A progress bar is displayed (can be removed using setVerbose(False))
t0 = time()
POD = otpod.QuantileRegressionPOD(defects, signals, detection,
                                  boxCox=True)
POD.run()
print('Computing time : {:0.2f} s'.format(time()-t0))
```

```
Computing defect quantile: [==================================================] 100.00
↪% Done
Computing time : 324.25 s
```

The computing time can be reduced by setting the simulation size attribute to another value. However the confidence interval is less accurate.

The number of quantile values can also be reduced to save time.

```
t0 = time()
PODsimulSize100 = otpod.QuantileRegressionPOD(defects, signals, detection,
                                    boxCox=True)
PODsimulSize100.setSimulationSize(100) # default is 1000
PODsimulSize100.run()
print('Computing time : {:0.2f} s'.format(time()-t0))
```

```
Computing defect quantile: [==================================================] 100.00
↪% Done
Computing time : 33.81 s
```

Compute detection size

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(POD.computeDetectionSize(0.9, 0.95))

# probability level 0.95 with confidence level 0.99
print(POD.computeDetectionSize(0.95, 0.99))
```

```
[a90 : 0.298115, a90/95 : 0.328585]
[a95 : 0.331931, a95/99 : 0.372112]
```

get POD NumericalMathFunction

```
# get the POD model
PODmodel = POD.getPODModel()
# get the POD model at the given confidence level
PODmodelCl95 = POD.getPODCLModel(0.95)

# compute the probability of detection for a given defect value
print('POD : {:0.3f}'.format(PODmodel([0.3])[0]))
print('POD at level 0.95 : {:0.3f}'.format(PODmodelCl95([0.3])[0]))
```

```
POD : 0.899
POD at level 0.95 : 0.832
```

Compute the pseudo R2 for a given quantile

```
print('Pseudo R2 for quantile 0.9 : {:0.3f}'.format(POD.getR2(0.9)))
print('Pseudo R2 for quantile 0.95 : {:0.3f}'.format(POD.getR2(0.95)))
```

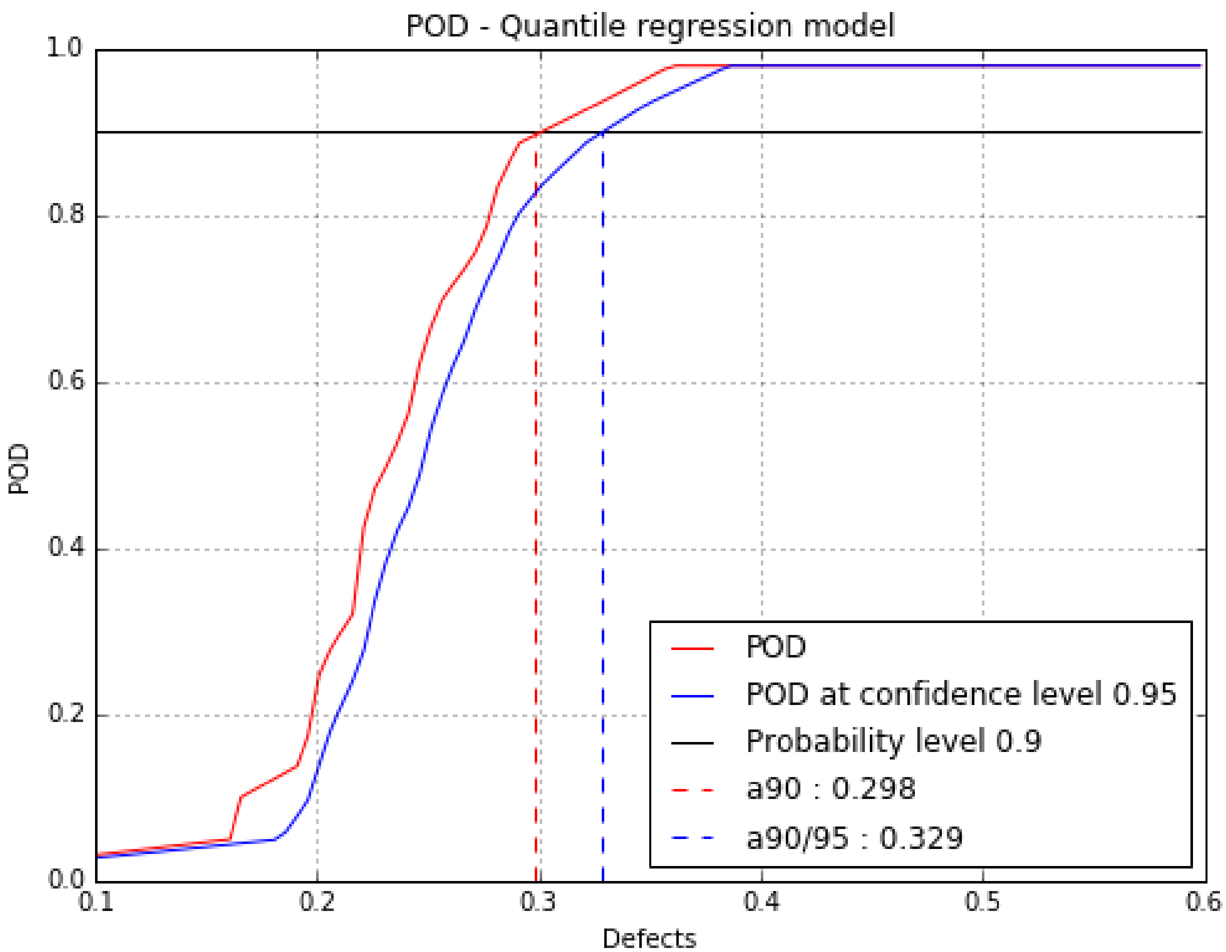
```
Pseudo R2 for quantile 0.9 : 0.675
Pseudo R2 for quantile 0.95 : 0.656
```

Show POD graphs

Mean POD and POD at confidence level with the detection size for a given probability level

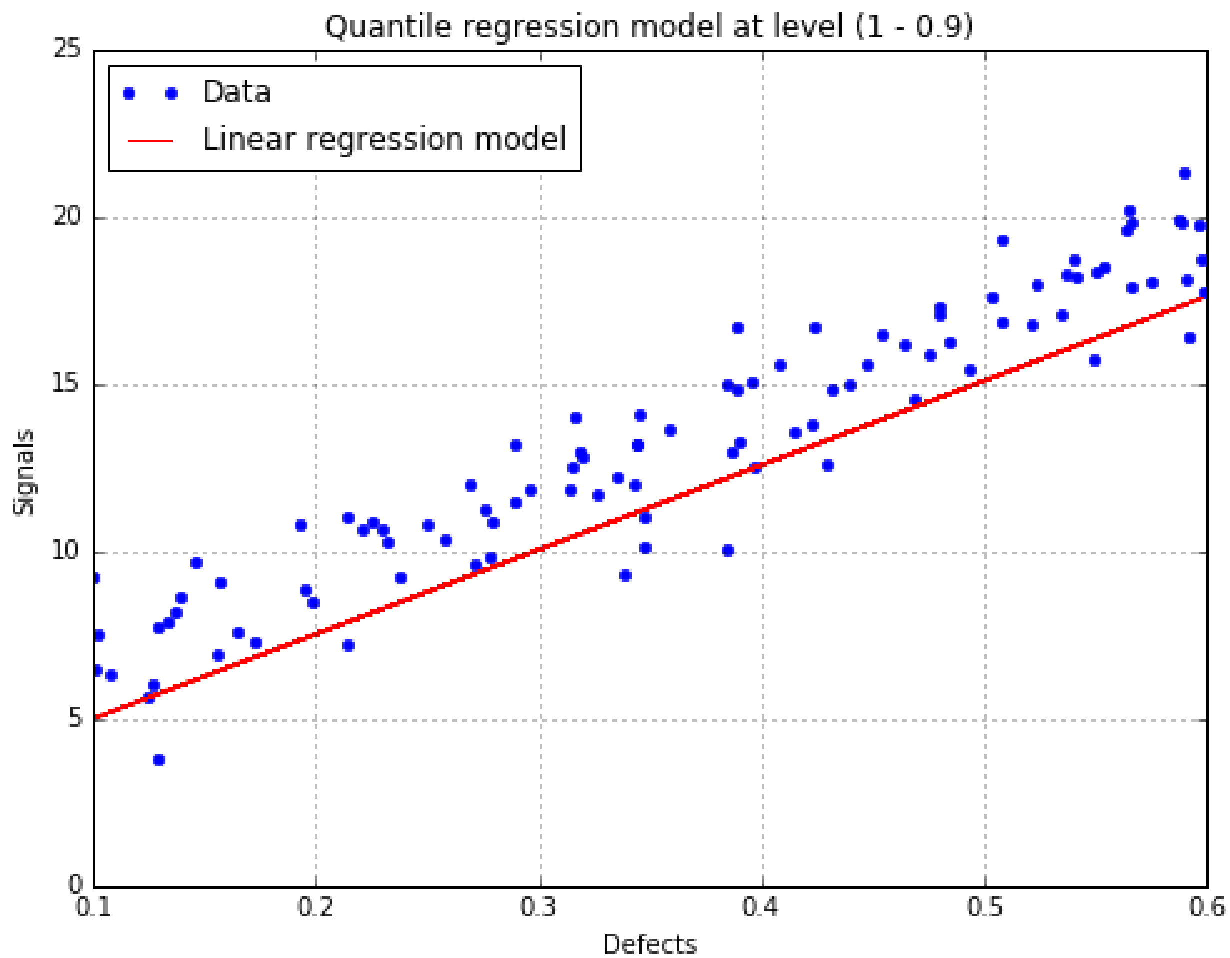
```
fig, ax = POD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODQuantReg.png')
# The figure is saved in PODQuantReg.png
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



Show the linear regression model at the given quantile

```
fig, ax = POD.drawLinearModel(0.9)
fig.show()
```



ipynb source code

1.2.6 Polynomial chaos POD

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate 1D data

```
N = 100
ot.RandomGenerator.SetSeed(123456)
defectDist = ot.Uniform(0.1, 0.6)
# normal epsilon distribution
epsilon = ot.Normal(0, 1.9)
defects = defectDist.getSample(N)
signalsInvBoxCox = defects * 43. + epsilon.getSample(N) + 2.5
# Inverse Box Cox transformation
```

```
invBoxCox = ot.InverseBoxCoxTransform(0.3)
signals = invBoxCox(signalsInvBoxCox)
```

Build POD with polynomial chaos model

```
# signal detection threshold
detection = 200.
# The POD with censored data actually builds a POD only on filtered data.
# A warning is diplayed in this case.
POD = otpod.PolynomialChaosPOD(defects, signals, detection,
                               noiseThres=200., saturationThres=1700.,
                               boxCox=True)
```

User-defined defect sizes

The user-defined defect sizes must range between the minimum and maximum of the defect values after filtering. An error is raised if it is not the case. The available range is then returned to the user.

```
# Default defect sizes
print('Default defect sizes : ')
print(POD.getDefectSizes())

# Wrong range
try:
    POD.setDefectSizes([0.12, 0.3, 0.5, 0.57])
except ValueError as e:
    print('')
    print('Range of the defect sizes is too large, it returns a value error : ')
    print(e)
```

```
Default defect sizes :
[ 0.19288542  0.21420345  0.23552149  0.25683952  0.27815756  0.29947559
  0.32079363  0.34211166  0.3634297   0.38474773  0.40606577  0.4273838
  0.44870184  0.47001987  0.49133791  0.51265594  0.53397398  0.55529201
  0.57661005  0.59792808]

Range of the defect sizes is too large, it returns a value error :
Defect sizes must range between 0.1929 and 0.5979.
```

```
# Good range
POD.setDefectSizes([0.1929, 0.3, 0.4, 0.5, 0.5979])
print('User-defined defect size : ')
print(POD.getDefectSizes())
```

```
User-defined defect size :
[ 0.1929  0.3     0.4     0.5     0.5979]
```

Running the polynomial chaos based POD

The computing time can be reduced by setting the simulation size attribute to another value. However the confidence interval is less accurate.

The sampling size is the number of the samples used to compute the POD with the Monte Carlo simulation for each defect sizes.

A progress is displayed, which can be disabled with the method *setVerbose*.

```
# Computing the confidence interval in the run takes few minutes.
POD = otpod.PolynomialChaosPOD(defects, signals, detection,
                                    boxCox=True)
# we can change the sample size of the Monte Carlo simulation
POD.setSamplingSize(2000) # default is 5000
# we can also change the size of the simulation to compute the confidence interval
POD.setSimulationSize(500) # default is 1000
# we can change the degree of the polynomial chaos, default is 3.
POD.setDegree(3)
%time POD.run()
```

```
Start build polynomial chaos model...
Polynomial chaos model completed
Polynomial chaos validation R2 (>0.8) : 0.8947
Polynomial chaos validation Q2 (>0.8) : 0.8914
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 2min 1s, sys: 6.55 s, total: 2min 8s
Wall time: 2min 19s
```

Compute detection size

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(POD.computeDetectionSize(0.9, 0.95))

# probability level 0.95 with confidence level 0.99
print(POD.computeDetectionSize(0.95, 0.99))
```

```
[a90 : 0.307344, a90/95 : 0.314406]
[a95 : 0.328888, a95/99 : 0.335715]
```

get POD NumericalMathFunction

```
# get the POD model
PODmodel = POD.getPODModel()
# get the POD model at the given confidence level
PODmodelCl95 = POD.getPODCLModel(0.95)

# compute the probability of detection for a given defect value
print('POD : {:0.3f}'.format(PODmodel([0.3])[0]))
print('POD at level 0.95 : {:0.3f}'.format(PODmodelCl95([0.3])[0]))
```

```
POD : 0.871
POD at level 0.95 : 0.841
```

Compute the R2 and the Q2

Enable to check the quality of the model.

```
print('R2 : {:0.4f}'.format(POD.getR2()))
print('Q2 : {:0.4f}'.format(POD.getQ2()))
```

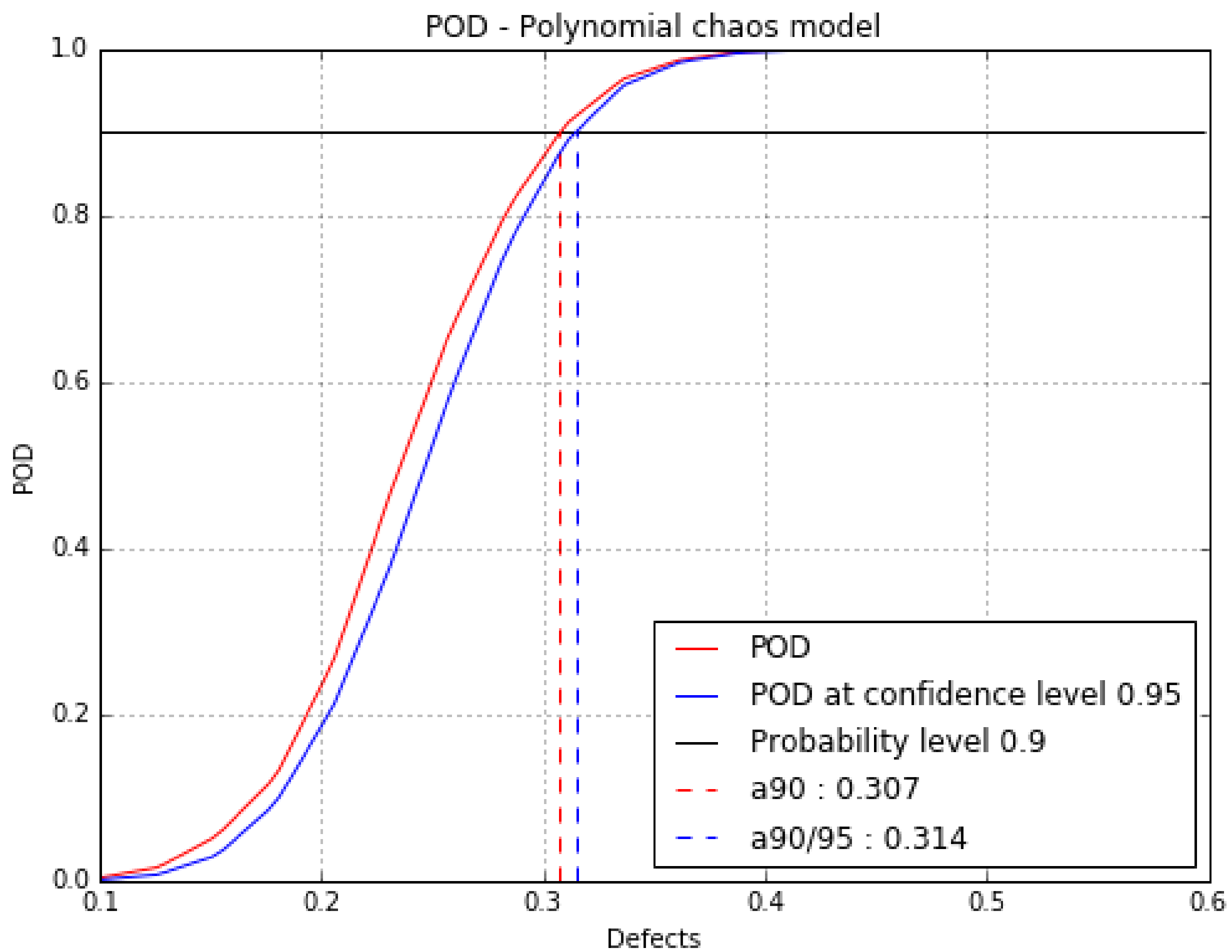
```
R2 : 0.8947
Q2 : 0.8914
```

Show POD graphs

Mean POD and POD at confidence level with the detection size for a given probability level

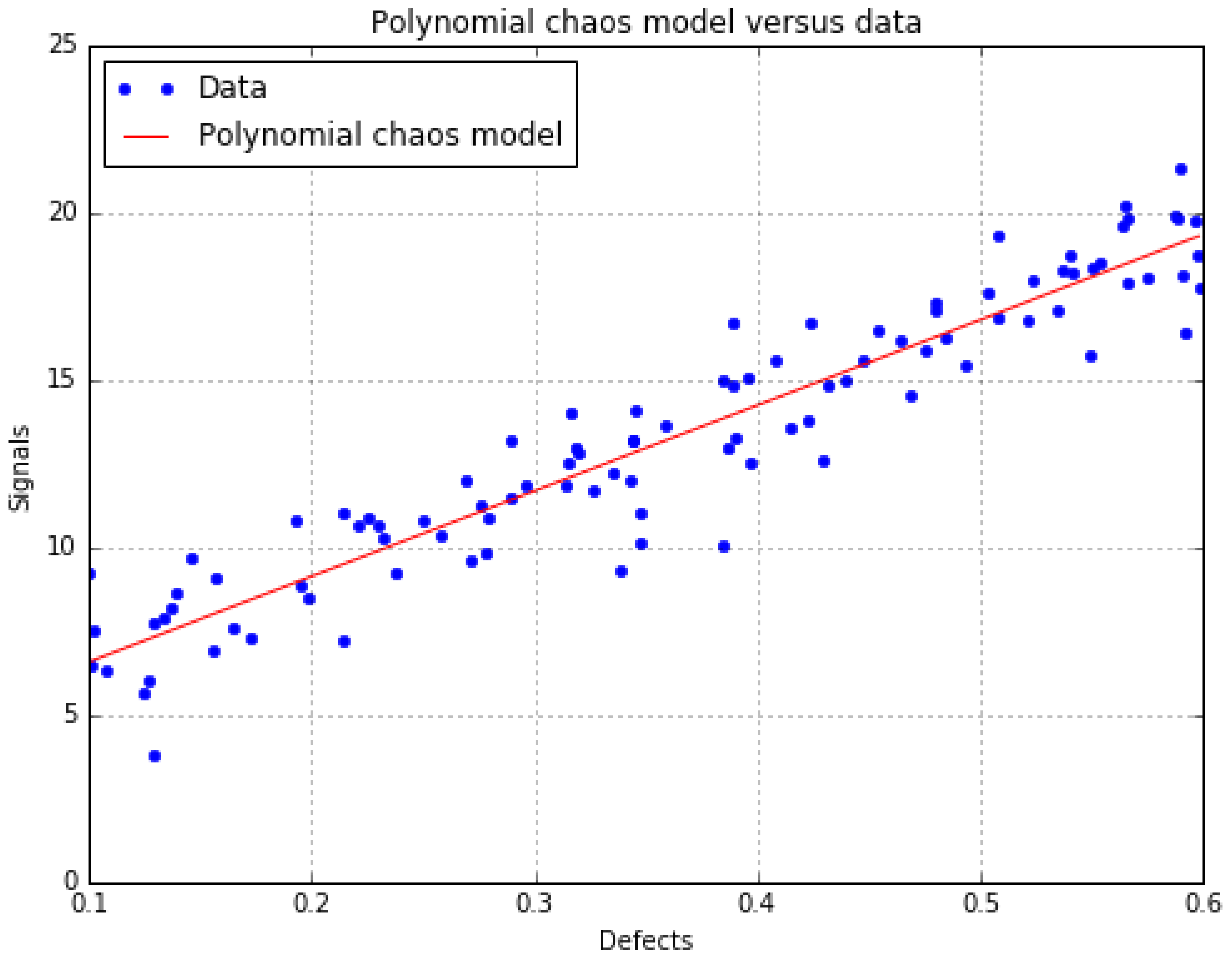
```
fig, ax = POD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODPolyChaos.png')
# The figure is saved in PODPolyChaos.png
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



Show the polynomial chaos model (only available if the input dimension is 1)

```
fig, ax = POD.drawPolynomialChaosModel()
fig.show()
```



Advanced user mode

The user can defined one or all parameters of the polynomial chaos algorithm : - the distribution of the input parameters - the adaptive strategy - the projection strategy

```
# new POD study
PODnew = otpod.PolynomialChaosPOD(defects, signals, detection,
                                  boxCox=True)
```

```
# define the input parameter distribution
distribution = ot.ComposedDistribution([ot.Normal(0.3, 0.1)])
PODnew.setDistribution(distribution)
```

```
# define the adaptive strategy
polyCol = [ot.HermiteFactory()]
enumerateFunction = ot.EnumerateFunction(1)
multivariateBasis = ot.OrthogonalProductPolynomialFactory(polyCol, enumerateFunction)
# degree 1
p = 1
indexMax = enumerateFunction.getStrataCumulatedCardinal(p)
adaptiveStrategy = ot.FixedStrategy(multivariateBasis, indexMax)
```

```
PODnew.setAdaptiveStrategy(adaptiveStrategy)
```

```
# define the projection strategy
projectionStrategy = ot.LeastSquaresStrategy()
PODnew.setProjectionStrategy(projectionStrategy)
```

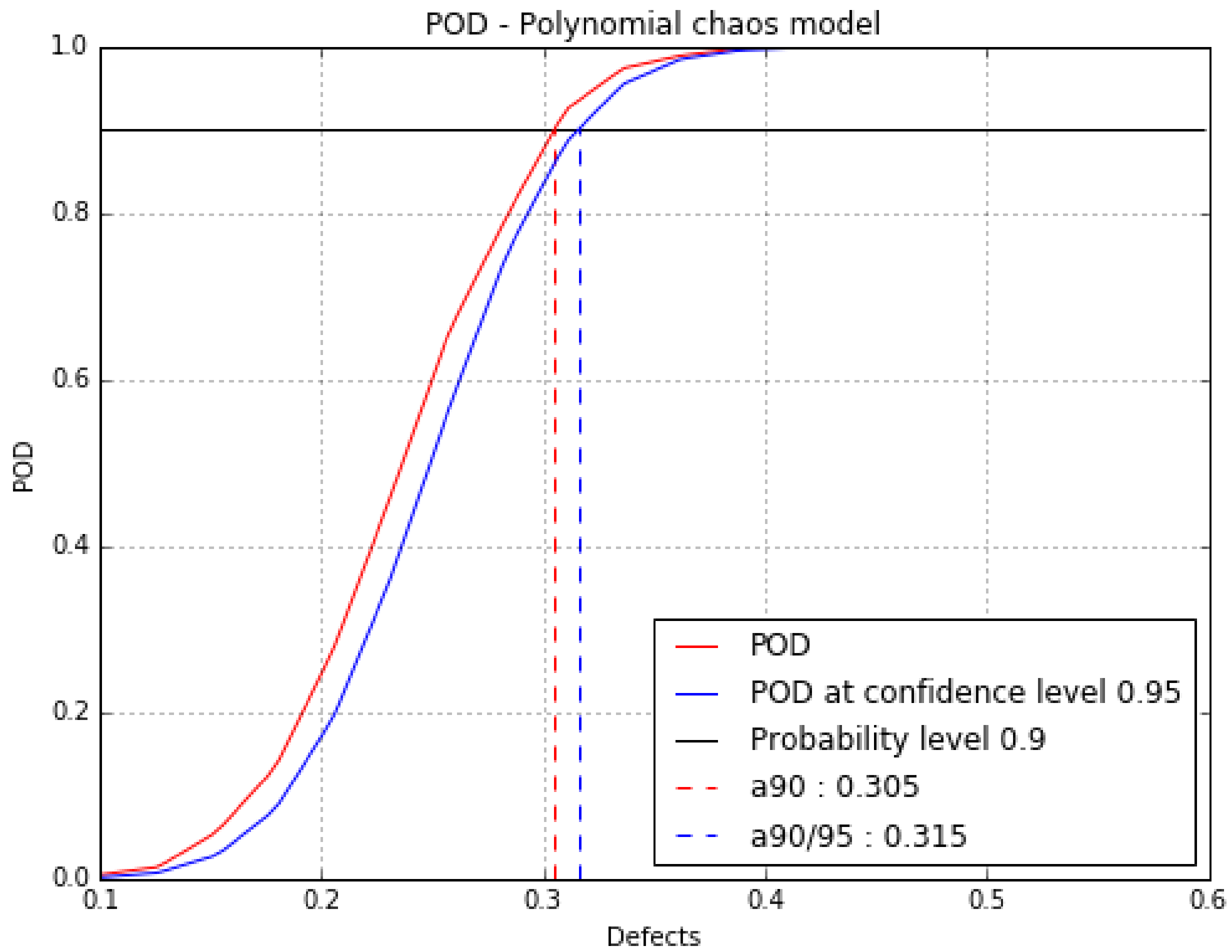
```
POD.setSamplingSize(2000)
POD.setSimulationSize(500)
PODnew.run()
```

```
Start build polynomial chaos model...
Polynomial chaos model completed
Polynomial chaos validation R2 (>0.8) : 0.8947
Polynomial chaos validation Q2 (>0.8) : 0.8914
Computing POD per defect: [==================================================] 100.00
↪% Done
```

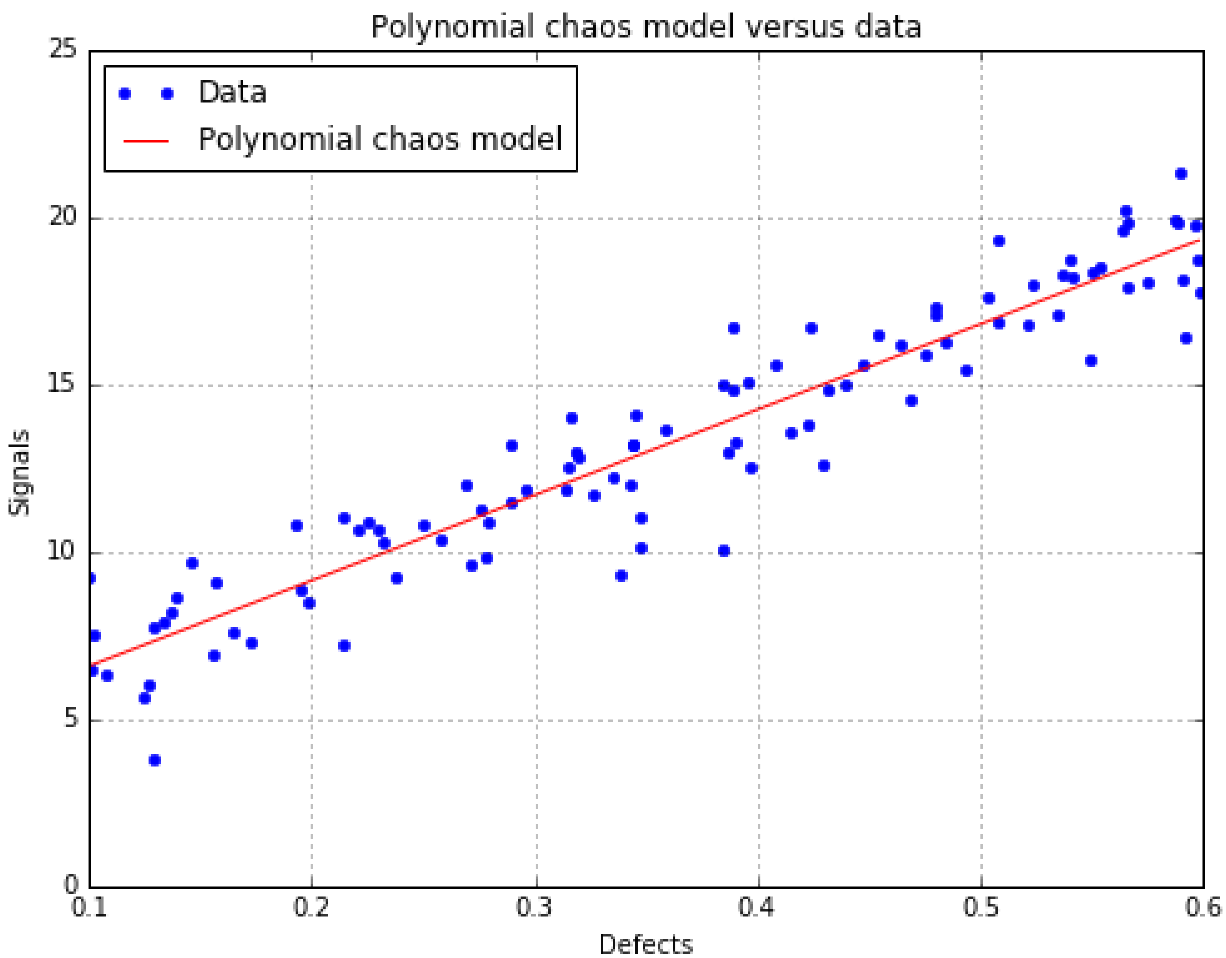
```
print(PODnew.computeDetectionSize(0.9, 0.95))
print('R2 : {:0.4f}'.format(POD.getR2()))
print('Q2 : {:0.4f}'.format(POD.getQ2()))
```

```
[a90 : 0.304772, a90/95 : 0.315494]
R2 : 0.8947
Q2 : 0.8914
```

```
fig, ax = PODnew.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95)
fig.show()
```




```
fig, ax = PODnew.drawPolynomialChaosModel()
fig.show()
```



ipynb source code

1.2.7 Kriging POD

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
inputSample = ot.NumericalSample(
    [[4.59626812e+00, 7.46143339e-02, 1.02231538e+00, 8.60042277e+01],
    [4.14315790e+00, 4.20801346e-02, 1.05874908e+00, 2.65757364e+01],
    [4.76735111e+00, 3.72414824e-02, 1.05730385e+00, 5.76058433e+01],
    [4.82811977e+00, 2.49997658e-02, 1.06954641e+00, 2.54461380e+01],
    [4.48961094e+00, 3.74562922e-02, 1.04943946e+00, 6.19483646e+00],
    [5.05605334e+00, 4.87599783e-02, 1.06520409e+00, 3.39024904e+00],
    [5.69679328e+00, 7.74915877e-02, 1.04099514e+00, 6.50990466e+01],
    [5.10193991e+00, 4.35520544e-02, 1.02502536e+00, 5.51492592e+01],
```

```
    [4.04791970e+00, 2.38565932e-02, 1.01906882e+00, 2.07875350e+01],
    [4.66238956e+00, 5.49901237e-02, 1.02427200e+00, 1.45661275e+01],
    [4.86634219e+00, 6.04693570e-02, 1.08199374e+00, 1.05104730e+00],
    [4.13519347e+00, 4.45225831e-02, 1.01900124e+00, 5.10117047e+01],
    [4.92541940e+00, 7.87692335e-02, 9.91868726e-01, 8.32302238e+01],
    [4.70722074e+00, 6.51799251e-02, 1.10608515e+00, 3.30181002e+01],
    [4.29040932e+00, 1.75426222e-02, 9.75678838e-01, 2.28186756e+01],
    [4.89291400e+00, 2.34997929e-02, 1.07669835e+00, 5.38926138e+01],
    [4.44653744e+00, 7.63175936e-02, 1.06979154e+00, 5.19109415e+01],
    [3.99977452e+00, 5.80430585e-02, 1.01850716e+00, 7.61988190e+01],
    [3.95491570e+00, 1.09302814e-02, 1.03687664e+00, 6.09981789e+01],
    [5.16424368e+00, 2.69026464e-02, 1.06673711e+00, 2.88708887e+01],
    [5.30491620e+00, 4.53802273e-02, 1.06254792e+00, 3.03856837e+01],
    [4.92809155e+00, 1.20616369e-02, 1.00700410e+00, 7.02512744e+00],
    [4.68373805e+00, 6.26028935e-02, 1.05152117e+00, 4.81271603e+01],
    [5.32381954e+00, 4.33013582e-02, 9.90522007e-01, 6.56015973e+01],
    [4.35455857e+00, 1.23814619e-02, 1.01810539e+00, 1.10769534e+01]])

signals = ot.NumericalSample(
    [[ 37.305445], [ 35.466919], [ 43.187991], [ 45.305165], [ 40.121222], [ 44.
↪609524],
     [ 45.14552 ], [ 44.80595 ], [ 35.414039], [ 39.851778], [ 42.046049], [ 34.73469␣
↪],
     [ 39.339349], [ 40.384559], [ 38.718623], [ 46.189709], [ 36.155737], [ 31.
↪768369],
     [ 35.384313], [ 47.914584], [ 46.758537], [ 46.564428], [ 39.698493], [ 45.
↪636588],
     [ 40.643948]])
```

Build POD with Kriging model

```
# signal detection threshold
detection = 38.
# The POD with censored data actually builds a POD only on filtered data.
# A warning is diplayed in this case.
POD = otpod.KrigingPOD(inputSample, signals, detection,
                       noiseThres=35., saturationThres=45.)
```

User-defined defect sizes

The user-defined defect sizes must range between the minimum and maximum of the defect values after filtering. An error is raised if it is not the case. The available range is then returned to the user.

```
# Default defect sizes
print('Default defect sizes : ')
print(POD.getDefectSizes())

# Wrong range
try:
    POD.setDefectSizes([3.2, 3.6, 4.5, 5.5])
except ValueError as e:
    print('Range of the defect sizes is too large, it returns a value error : ')
    print(e)
```

```
Default defect sizes :
[ 3.9549157   4.0152854   4.07565509  4.13602479  4.19639448  4.25676418
  4.31713387  4.37750357  4.43787326  4.49824296  4.55861265  4.61898235
  4.67935204  4.73972174  4.80009143  4.86046113  4.92083082  4.98120052
  5.04157021  5.10193991]
Range of the defect sizes is too large, it returns a value error :
Defect sizes must range between 3.9550 and 5.1019.
```

```
# Good range
POD.setDefectSizes([4., 4.3, 4.6, 4.9, 5.1])
print('User-defined defect size : ')
print(POD.getDefectSizes())
```

```
User-defined defect size :
[ 4.   4.3  4.6  4.9  5.1]
```

Running the Kriging based POD

The computing time can be reduced by setting the simulation size attribute to another value. However the confidence interval is less accurate.

The sampling size is the number of the samples used to compute the POD with the Monte Carlo simulation for each defect sizes.

A progress is displayed, which can be disabled with the method *setVerbose*.

```
POD = otpod.KrigingPOD(inputSample, signals, detection)
# we can change the number of initial random search for the best starting point
# of the TNC algorithm which optimizes the covariance model parameters
POD.setInitialStartSize(500) # default is 1000
# we can change the sample size of the Monte Carlo simulation
POD.setSamplingSize(2000) # default is 5000
# we can also change the size of the simulation to compute the confidence interval
POD.setSimulationSize(500) # default is 1000
%time POD.run()
```

```
Start optimizing covariance model parameters...
Kriging optimizer completed
kriging validation Q2 (>0.9): 1.0000
Computing POD per defect: [==================================================] 100.00
 ↪% Done
CPU times: user 37.7 s, sys: 13.2 s, total: 50.9 s
Wall time: 52.3 s
```

Compute detection size

```
# Detection size at probability level 0.9
# and confidence level 0.95
print(POD.computeDetectionSize(0.9, 0.95))

# probability level 0.95 with confidence level 0.99
print(POD.computeDetectionSize(0.95, 0.99))
```

```
[a90 : 4.62318, a90/95 : 4.63983]
[a95 : 4.66733, a95/99 : 4.6837]
```

get POD NumericalMathFunction

```
# get the POD model
PODmodel = POD.getPODModel()
# get the POD model at the given confidence level
PODmodelCl95 = POD.getPODCLModel(0.95)

# compute the probability of detection for a given defect value
print('POD : {:0.3f}'.format(PODmodel([4.2])[0]))
print('POD at level 0.95 : {:0.3f}'.format(PODmodelCl95([4.2])[0]))
```

```
POD : 0.148
POD at level 0.95 : 0.126
```

Compute the Q2

Enable to check the quality of the model.

```
print('Q2 : {:0.4f}'.format(POD.getQ2()))
```

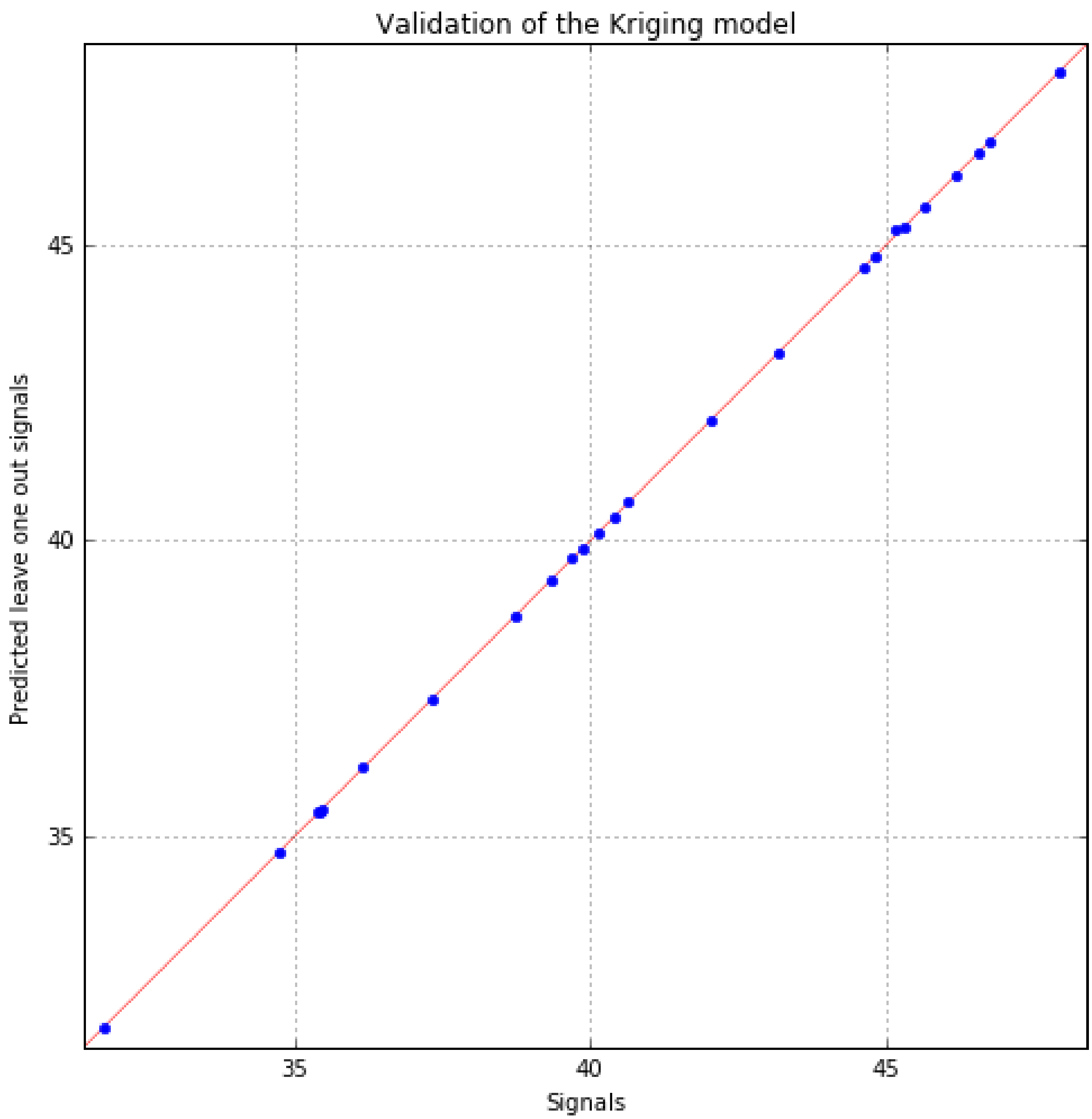
```
Q2 : 1.0000
```

Draw the validation graph

The predictions are the one computed by leave one out.

```
fig, ax = POD.drawValidationGraph()
fig.show()
```

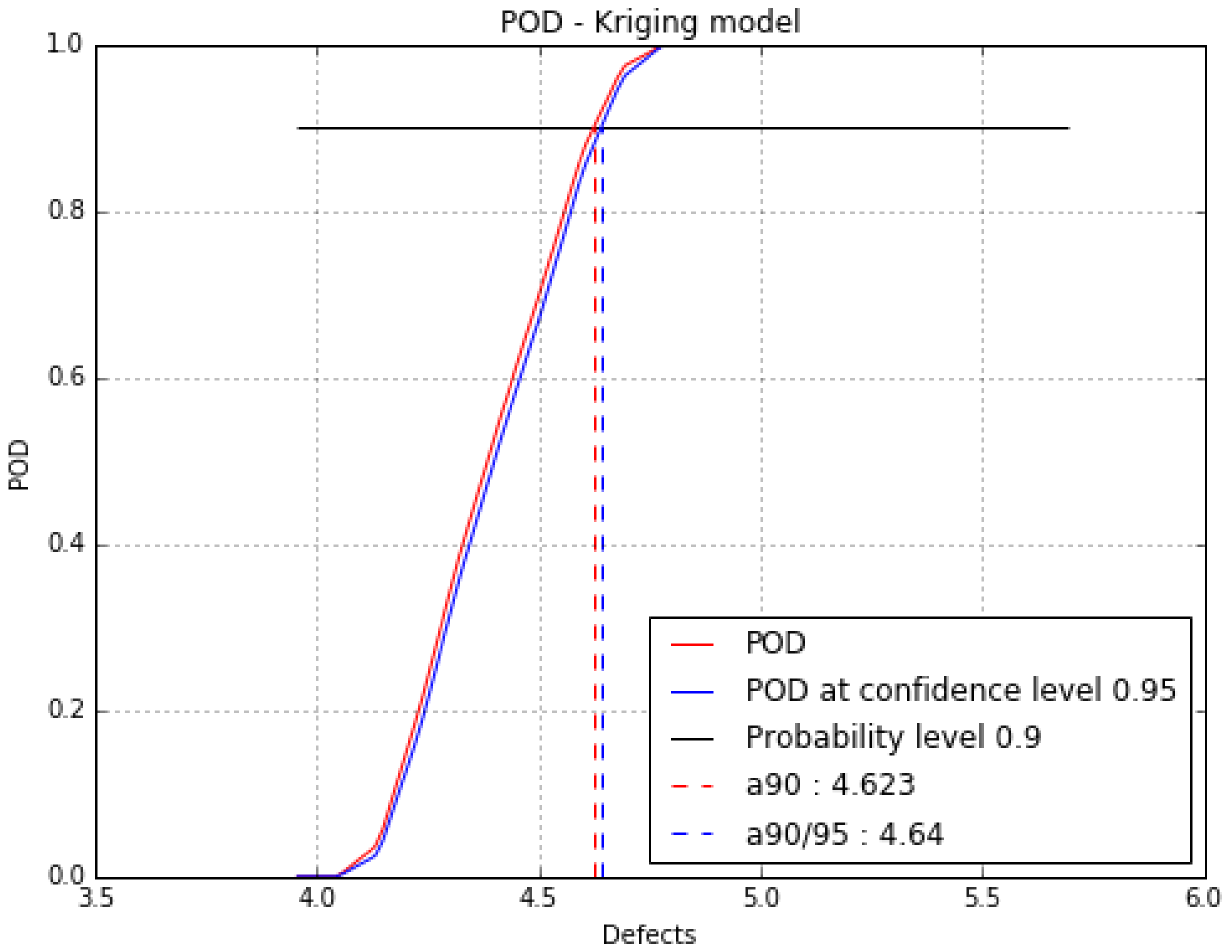
```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



Show POD graphs

Mean POD and POD at confidence level with the detection size for a given probability level

```
fig, ax = POD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODKriging.png')
# The figure is saved in PODPolyChaos.png
fig.show()
```



Advanced user mode

The user can defined one or both parameters of the kriging algorithm : - the basis - the covariance model

The user can also defined the input parameter distribution it is known.

The user can set the KrigingResult object if it built from other data.

```
# new POD study
PODnew = otpod.KrigingPOD(inputSample, signals, detection)
```

```
# set the basis constant
basis = ot.ConstantBasisFactory(4).build()
PODnew.setBasis(basis)
```

```
# set the covariance Model as an absolute exponential model
covColl = ot.CovarianceModelCollection(4)
for i in xrange(4):
    covColl[i]  = ot.AbsoluteExponential([1], [1.])
covarianceModel = ot.ProductCovarianceModel(covColl)

PODnew.setCovarianceModel(covarianceModel)
```

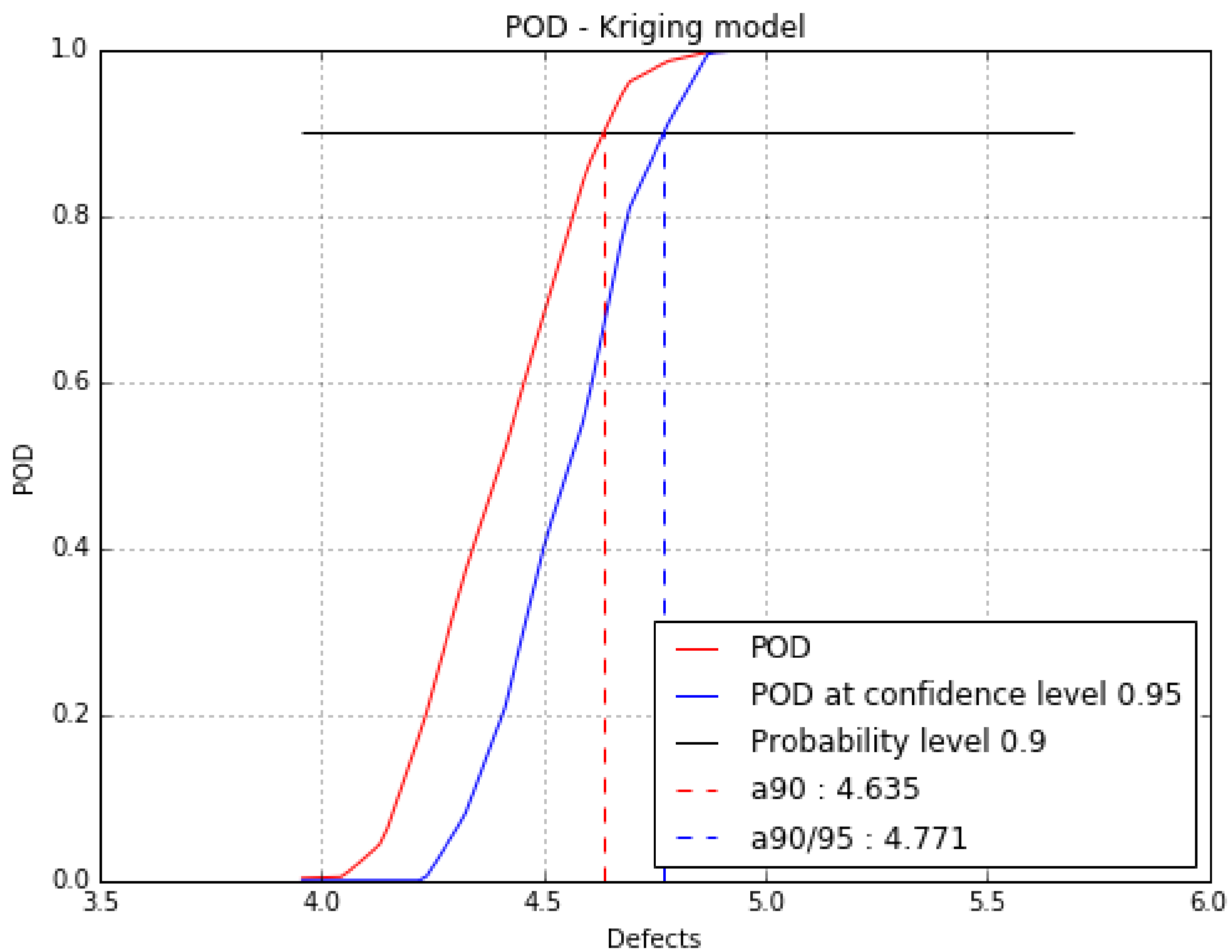
```
PODnew.run()
```

```
Start optimizing covariance model parameters...
Kriging optimizer completed
kriging validation Q2 (>0.9): 0.9660
Computing POD per defect: [==================================================] 100.00
↪% Done
```

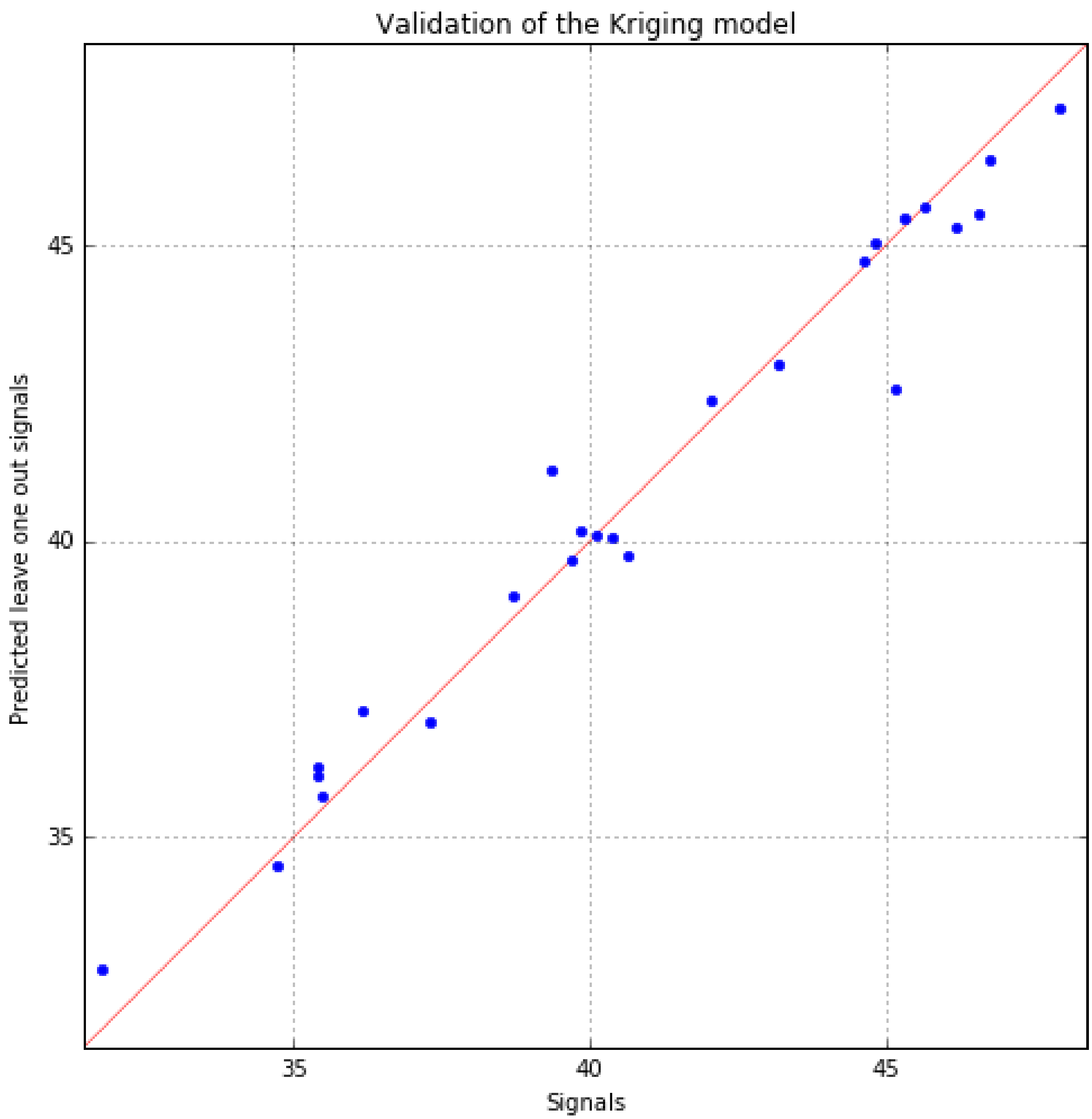
```
print(PODnew.computeDetectionSize(0.9, 0.95))
print('Q2 : {:0.4f}'.format(POD.getQ2()))
```

```
[a90 : 4.63513, a90/95 : 4.77085]
Q2 : 1.0000
```

```
fig, ax = PODnew.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95)
fig.show()
```




```
fig, ax = PODnew.drawValidationGraph()
fig.show()
```



ipynb source code

1.2.8 POD Summary

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
inputSample = ot.NumericalSample(
    [[4.59626812e+00, 7.46143339e-02, 1.02231538e+00, 8.60042277e+01],
    [4.14315790e+00, 4.20801346e-02, 1.05874908e+00, 2.65757364e+01],
    [4.76735111e+00, 3.72414824e-02, 1.05730385e+00, 5.76058433e+01],
    [4.82811977e+00, 2.49997658e-02, 1.06954641e+00, 2.54461380e+01],
    [4.48961094e+00, 3.74562922e-02, 1.04943946e+00, 6.19483646e+00],
    [5.05605334e+00, 4.87599783e-02, 1.06520409e+00, 3.39024904e+00],
    [5.69679328e+00, 7.74915877e-02, 1.04099514e+00, 6.50990466e+01],
    [5.10193991e+00, 4.35520544e-02, 1.02502536e+00, 5.51492592e+01],
    [4.04791970e+00, 2.38565932e-02, 1.01906882e+00, 2.07875350e+01],
    [4.66238956e+00, 5.49901237e-02, 1.02427200e+00, 1.45661275e+01],
    [4.86634219e+00, 6.04693570e-02, 1.08199374e+00, 1.05104730e+00],
    [4.13519347e+00, 4.45225831e-02, 1.01900124e+00, 5.10117047e+01],
    [4.92541940e+00, 7.87692335e-02, 9.91868726e-01, 8.32302238e+01],
    [4.70722074e+00, 6.51799251e-02, 1.10608515e+00, 3.30181002e+01],
    [4.29040932e+00, 1.75426222e-02, 9.75678838e-01, 2.28186756e+01],
    [4.89291400e+00, 2.34997929e-02, 1.07669835e+00, 5.38926138e+01],
    [4.44653744e+00, 7.63175936e-02, 1.06979154e+00, 5.19109415e+01],
    [3.99977452e+00, 5.80430585e-02, 1.01850716e+00, 7.61988190e+01],
    [3.95491570e+00, 1.09302814e-02, 1.03687664e+00, 6.09981789e+01],
    [5.16424368e+00, 2.69026464e-02, 1.06673711e+00, 2.88708887e+01],
    [5.30491620e+00, 4.53802273e-02, 1.06254792e+00, 3.03856837e+01],
    [4.92809155e+00, 1.20616369e-02, 1.00700410e+00, 7.02512744e+00],
    [4.68373805e+00, 6.26028935e-02, 1.05152117e+00, 4.81271603e+01],
    [5.32381954e+00, 4.33013582e-02, 9.90522007e-01, 6.56015973e+01],
    [4.35455857e+00, 1.23814619e-02, 1.01810539e+00, 1.10769534e+01]])

signals = ot.NumericalSample(
    [[ 37.305445], [ 35.466919], [ 43.187991], [ 45.305165], [ 40.121222], [ 44.
↪609524],
     [ 45.14552 ], [ 44.80595 ], [ 35.414039], [ 39.851778], [ 42.046049], [ 34.73469␣
↪],
     [ 39.339349], [ 40.384559], [ 38.718623], [ 46.189709], [ 36.155737], [ 31.
↪768369],
     [ 35.384313], [ 47.914584], [ 46.758537], [ 46.564428], [ 39.698493], [ 45.
↪636588],
     [ 40.643948]])
```

Compute POD with several methods

The object POD summary enables the user to compute the POD with all available techniques. techniques can be activated or not thanks to the method *setMethodActive*. Then results can be printed or saved in a file to be compared. Moreover all graphs from the studies can be saved in a given directory.

The techniques are all activated by default : - Univariate linear model with Gaussian residuals, - Univariate linear model with no hypothesis on the residuals (Binomial), - Univariate linear model with kernel smoothing on the residuals, - Quantile regression, - Polynomial chaos, - Kriging (if input dimension > 1)

```
# signal detection threshold
detection = 38.
# The POD summary take
POD = otpod.PODSummary(inputSample, signals, detection, 25)
# The main parameters can modified :
# The number of simulation to compute the confidence level
POD.setSimulationSize(50)
```

```
# The number of Monte Carlo simulation to compute the POD for polynomial chaos and␣
↪kriging
POD.setSamplingSize(200)
# Deactivate the quantile regression technique
POD.setMethodActive('QuantileRegression', False)
# Finally run
POD.run()
```

```
Start univariate linear model analysis...

Start univariate linear model POD with Gaussian residuals...

Start univariate linear model POD with no hypothesis on the residuals...

Start univariate linear model POD with kernel smoothing on the residuals...
Computing POD (bootstrap): [==================================================] 100.00
↪% Done

Start polynomial chaos POD...
Start build polynomial chaos model...
Polynomial chaos model completed
Polynomial chaos validation R2 (>0.8) : 0.9999
Polynomial chaos validation Q2 (>0.8) : 0.9987
Computing POD per defect: [==================================================] 100.00
↪% Done

Start kriging POD...
Start optimizing covariance model parameters...
Kriging optimizer completed
kriging validation Q2 (>0.9): 1.0000
Computing POD per defect: [==================================================] 100.00
↪% Done
```

Access to the dictionnary of the active methods

```
POD.getMethodActive()
```

```
{'Kriging': True,
 'LinearBinomial': True,
 'LinearGauss': True,
 'LinearKernelSmoothing': True,
 'PolynomialChaos': True,
 'QuantileRegression': False}
```

Show results

It is shown the linear analysis results as well as the validation results of each model with the detection size computed for a given probability level and confidence level. These both values can be changed as parameters of the *printResults* method. The default values are probability level = 0.9 and confidence level = 0.95.

A warning is printed when the detection size with a technique returns an error. In this case, the return value is -1.

```
print(POD.getResults())
```

```
--------------------------------------------------------------------------------
         Linear model analysis results
--------------------------------------------------------------------------------
Box Cox parameter :                                 Not enabled

                                                     Uncensored      Censored

Intercept coefficient :                                    0.02          0.02
Slope coefficient :                                        8.71          8.71
Standard error of the estimate :                           2.29           2.2

Confidence interval on coefficients
Intercept coefficient :                              [-10.03, 10.07]
Slope coefficient :                                  [6.58, 10.85]
Level :                                                    0.95

Quality of regression
R2 (> 0.8):                                                0.76          0.76
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
         Residuals analysis results
--------------------------------------------------------------------------------
Fitted distribution (uncensored) :                 Normal(mu = -1.62004e-14, sigma = 2.
↪2441)
Fitted distribution (censored) :                   Normal(mu = 3.42417e-05, sigma = 2.
↪2441)

                                                     Uncensored      Censored
Distribution fitting test
Kolmogorov p-value (> 0.05):                               0.99          0.99

Normality test
Anderson Darling p-value (> 0.05):                         0.76          0.76
Cramer Von Mises p-value (> 0.05):                         0.83          0.83

Zero residual mean test
p-value (> 0.05):                                           1.0           1.0

Homoskedasticity test (constant variance)
Breush Pagan p-value (> 0.05):                             0.09          0.09
Harrison McCabe p-value (> 0.05):                          0.21          0.23

Non autocorrelation test
Durbin Watson p-value (> 0.05):                            0.34          0.34
--------------------------------------------------------------------------------



--------------------------------------------------------------------------------
         Model validation results
--------------------------------------------------------------------------------
                                                        Uncensored   Censored

                                                     R2         Q2          R2

Linear Regression (> 0.8):                         0.76                   0.76
Polynomial Chaos (> 0.8):                           1.0        1.0
```

```
Kriging (> 0.8):                                              1.0
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
         POD results
--------------------------------------------------------------------------------
                                                            a90          a90/95
Linear Regression
Gaussian residuals :                                       4.69           4.88
No residuals hypothesis :                                  4.71           4.88
Kernel smoothing on residuals :                            4.75           4.81

Polynomial chaos :                                          4.6           4.66
Kriging :                                                  4.64           4.66
--------------------------------------------------------------------------------

Warning : For polynomial chaos, kriging, results are given for filtered data.
```

Results can be displayed for another probability and confidence level.

```
print(POD.getResults(0.8, 0.9))
```

```
--------------------------------------------------------------------------------
         Linear model analysis results
--------------------------------------------------------------------------------
Box Cox parameter :                                 Not enabled

                                                     Uncensored       Censored

Intercept coefficient :                                    0.02           0.02
Slope coefficient :                                        8.71           8.71
Standard error of the estimate :                           2.29            2.2

Confidence interval on coefficients
Intercept coefficient :                           [-10.03, 10.07]
Slope coefficient :                               [6.58, 10.85]
Level :                                                    0.95

Quality of regression
R2 (> 0.8):                                                0.76           0.76
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
         Residuals analysis results
--------------------------------------------------------------------------------
Fitted distribution (uncensored) :                Normal(mu = -1.62004e-14, sigma = 2.
↪2441)
Fitted distribution (censored) :                  Normal(mu = 3.42417e-05, sigma = 2.
↪2441)

                                                     Uncensored       Censored
Distribution fitting test
Kolmogorov p-value (> 0.05):                               0.99           0.99

Normality test
Anderson Darling p-value (> 0.05):                         0.76           0.76
Cramer Von Mises p-value (> 0.05):                         0.83           0.83
```

```
Zero residual mean test
p-value (> 0.05):                                      1.0           1.0

Homoskedasticity test (constant variance)
Breush Pagan p-value (> 0.05):                        0.09          0.09
Harrison McCabe p-value (> 0.05):                     0.21          0.23

Non autocorrelation test
Durbin Watson p-value (> 0.05):                       0.34          0.34
--------------------------------------------------------------------------------




--------------------------------------------------------------------------------
         Model validation results
--------------------------------------------------------------------------------
                                                     Uncensored    Censored

                                                  R2          Q2          R2

Linear Regression (> 0.8):                      0.76                    0.76
Polynomial Chaos (> 0.8):                        1.0         1.0
Kriging (> 0.8):                                             1.0
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
         POD results
--------------------------------------------------------------------------------
                                                       a80         a80/90
Linear Regression
Gaussian residuals :                                  4.58          4.67
No residuals hypothesis :                             4.64          4.68
Kernel smoothing on residuals :                       4.61          4.69

Polynomial chaos :                                    4.52          4.58
Kriging :                                             4.57          4.59
--------------------------------------------------------------------------------

Warning : For polynomial chaos, kriging, results are given for filtered data.
```

Save results

The results can be saved in a text or csv file. As for the print method, the probability level and confidence level can be specified as parameters.

```
POD.saveResults('results.csv', probabilityLevel=0.9, confidenceLevel=0.95)
```

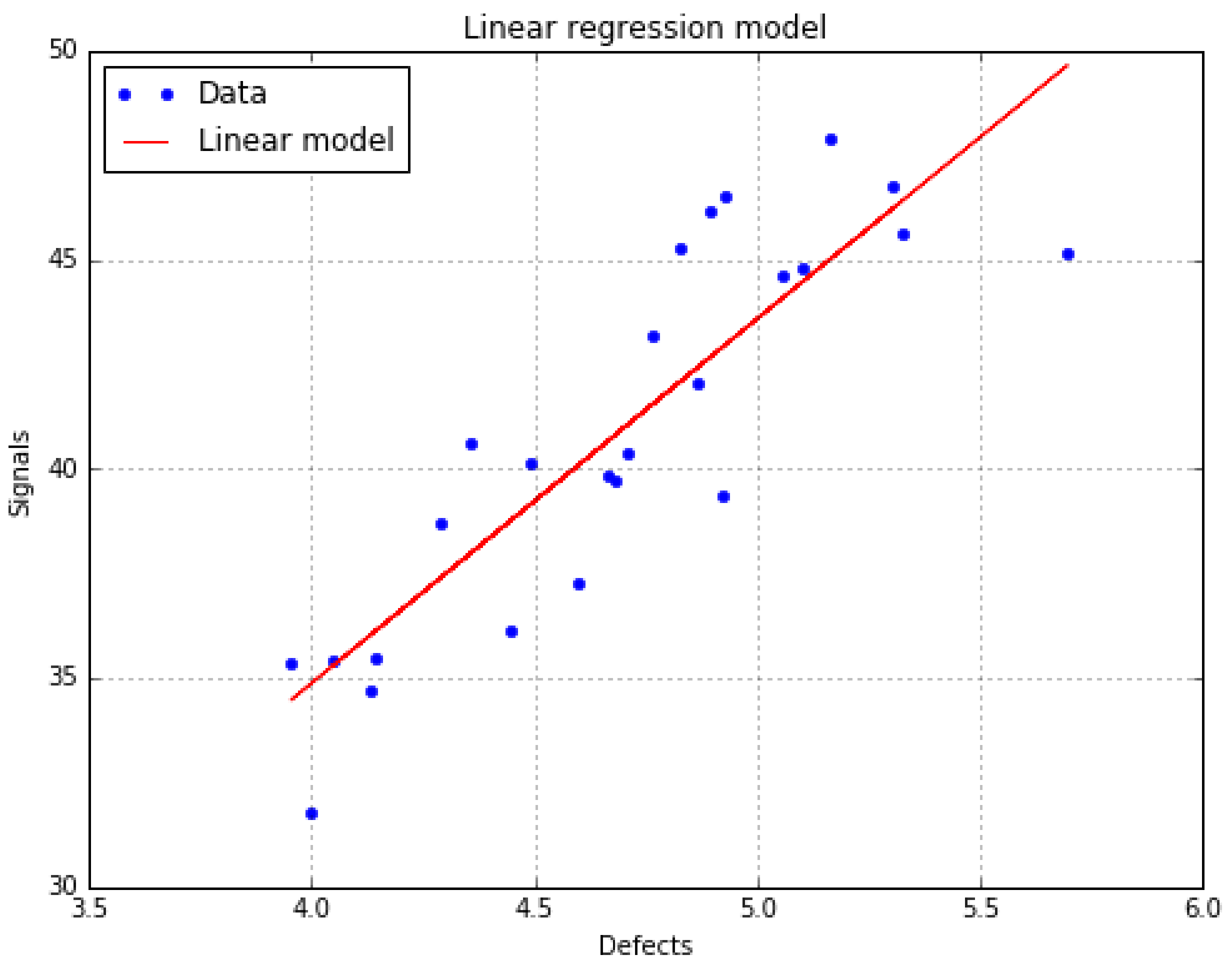
Draw and save graphs

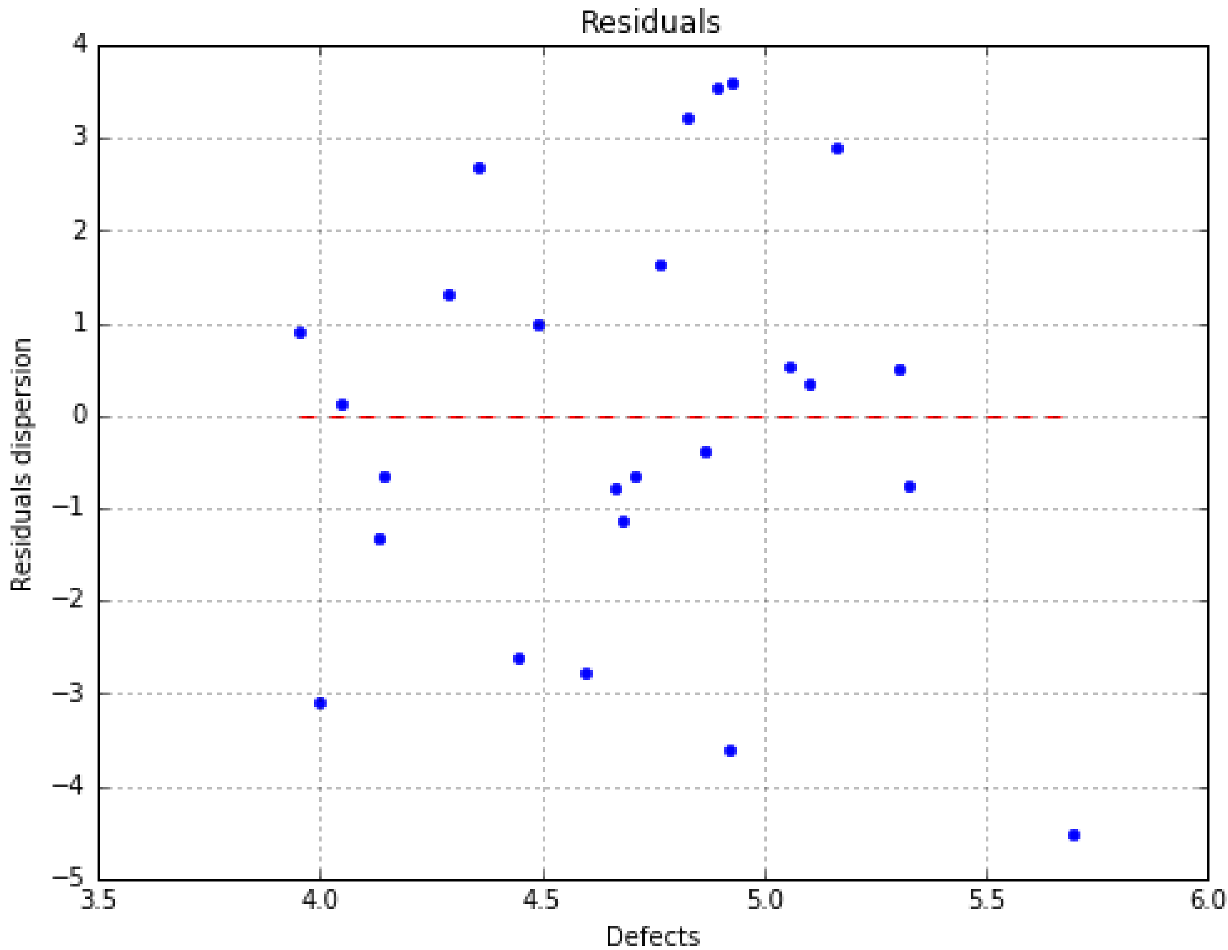
All available graphs can be saved using the method *saveGraphs*. A specific directory and the extension of the files can be given as parameters. As before the probability level and confidence level can also be chosen by the user.

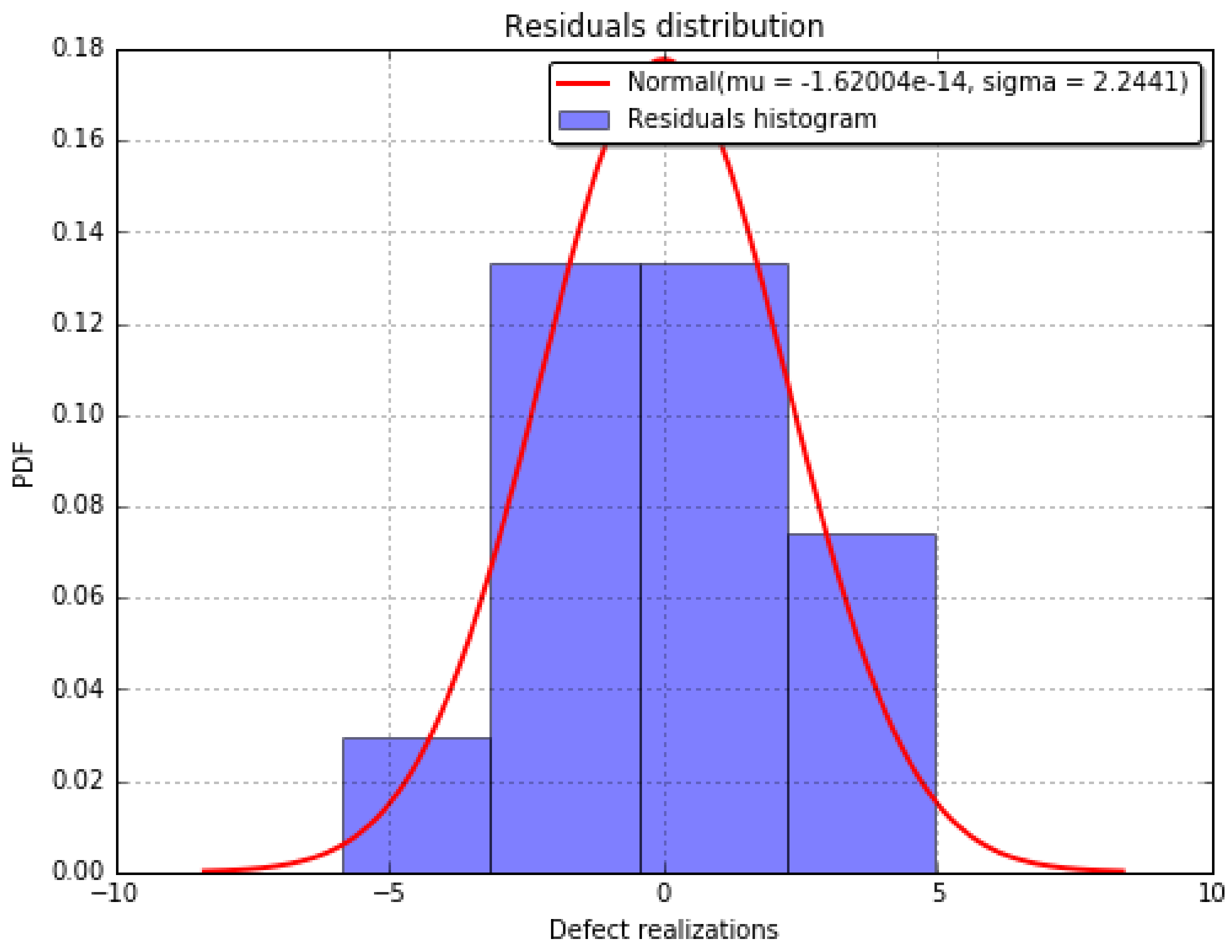
The warning is also printed here for the polynomial chaos because the detection size at the given probability level cannot be computed. A solution is to set *probabilityLevel* = *None*.

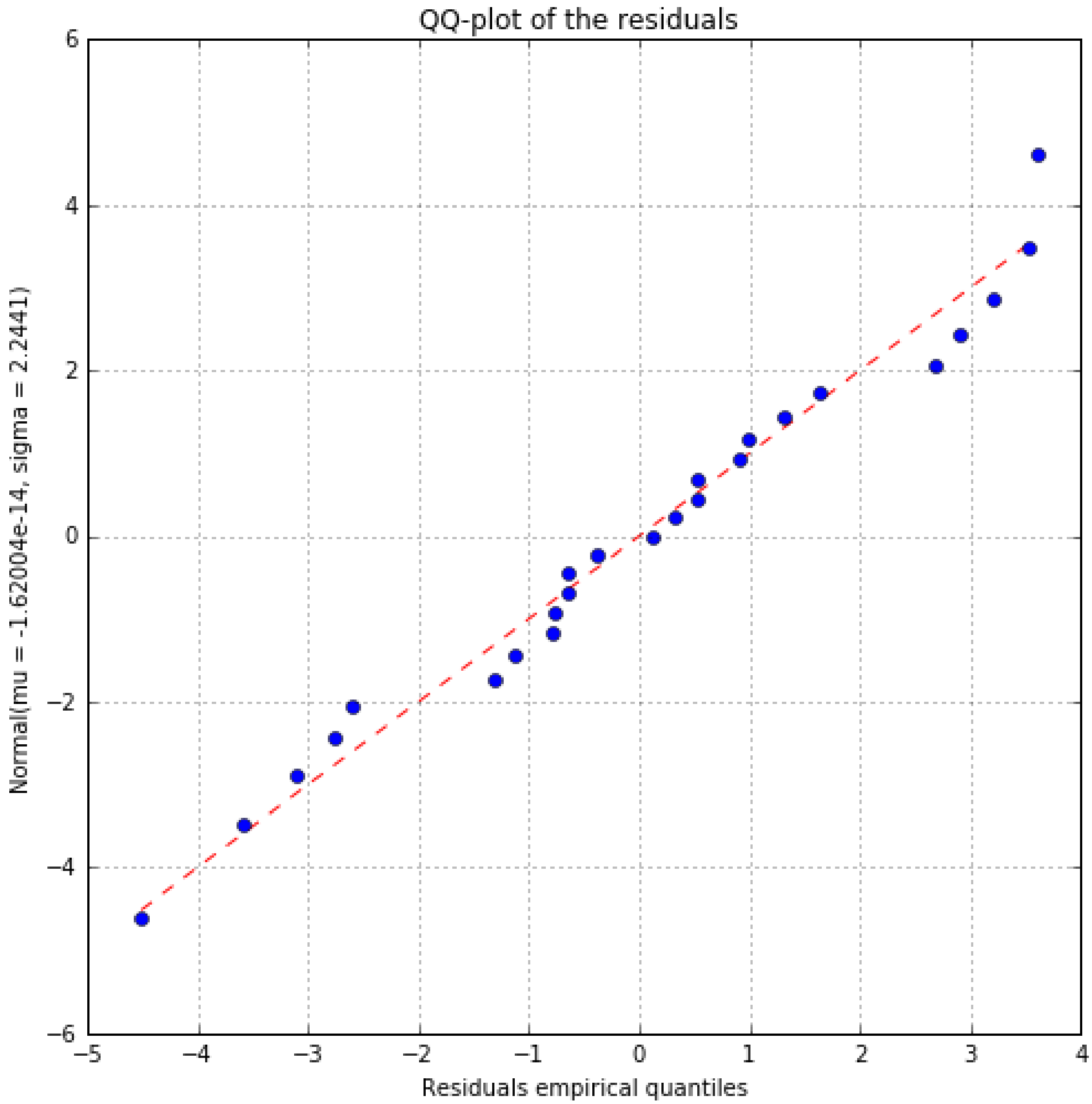
```
# return a list a figure
fig = POD.drawGraphs('./figure/', 'png', probabilityLevel=0.9, confidenceLevel=0.95)

for i in range(len(fig)):
    fig[i].show()
```

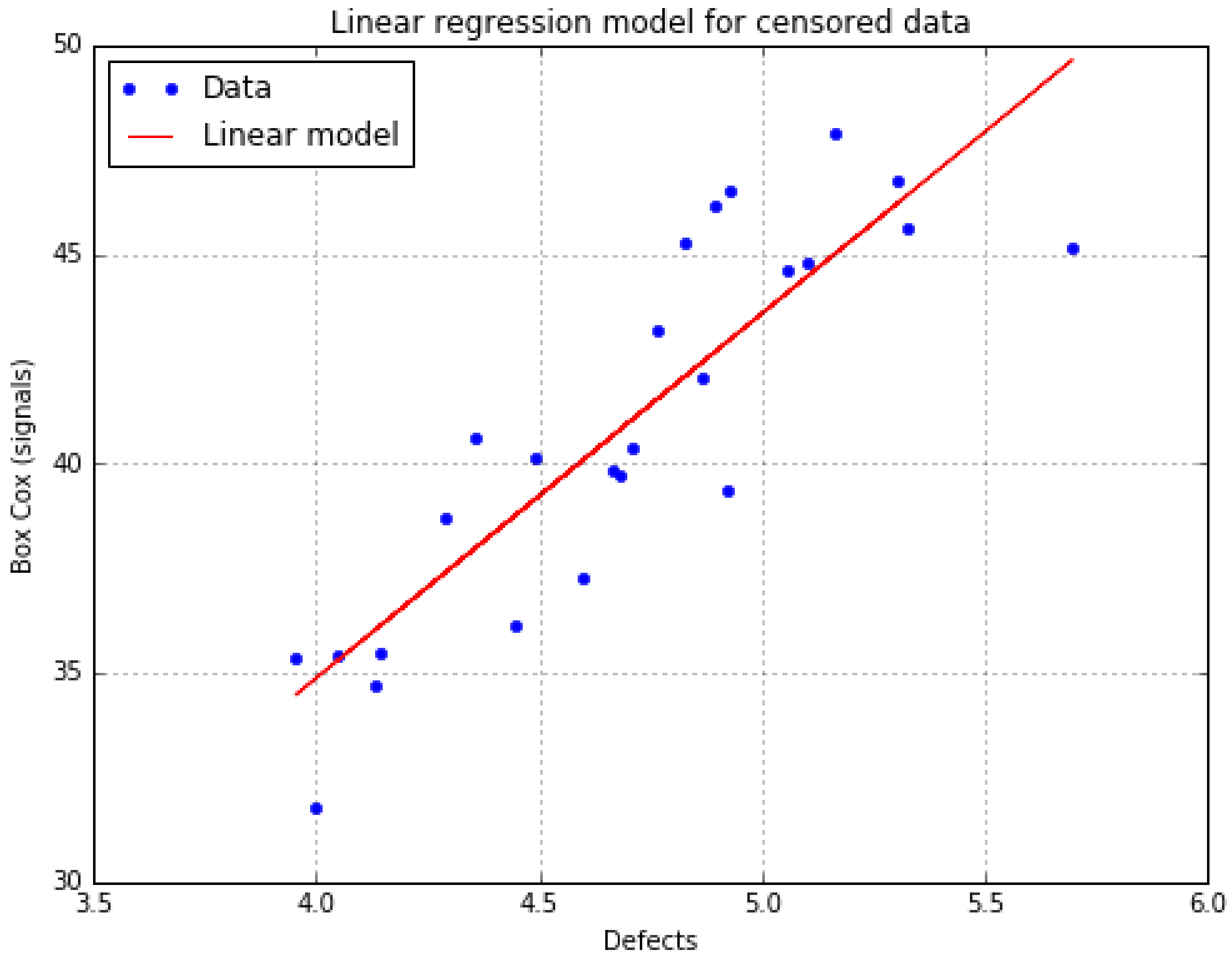
```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```

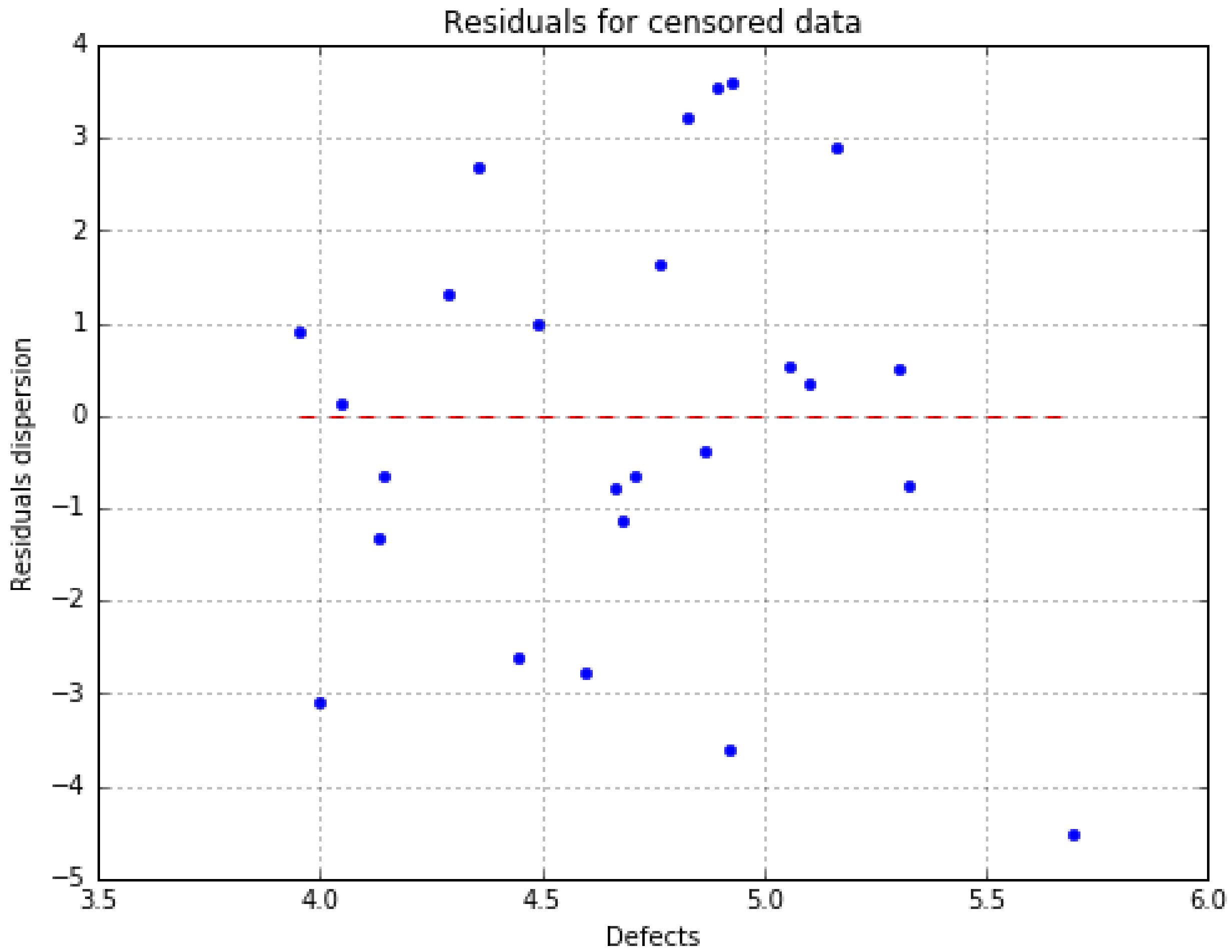



Residuals
Residuals dispersion
Defects
4
3
2
1
0
−1
−2
−3
−4
−5
3.5
4.0
4.5
5.0
5.5
6.0

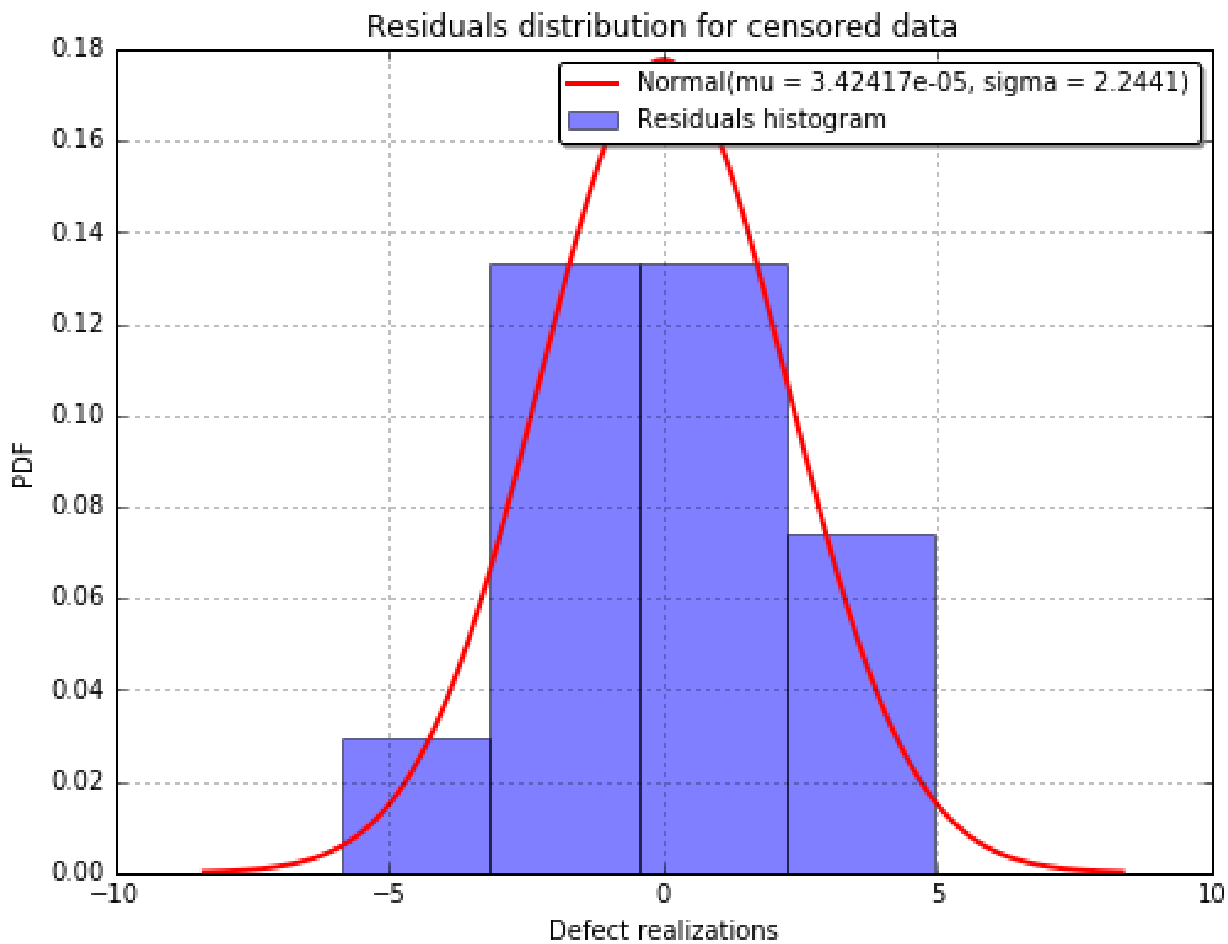

Residuals distribution
Normal(mu = -1.62004e-14, sigma = 2.2441)
Residuals histogram
PDF
Defect realizations
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
−10
−5
0
5
10

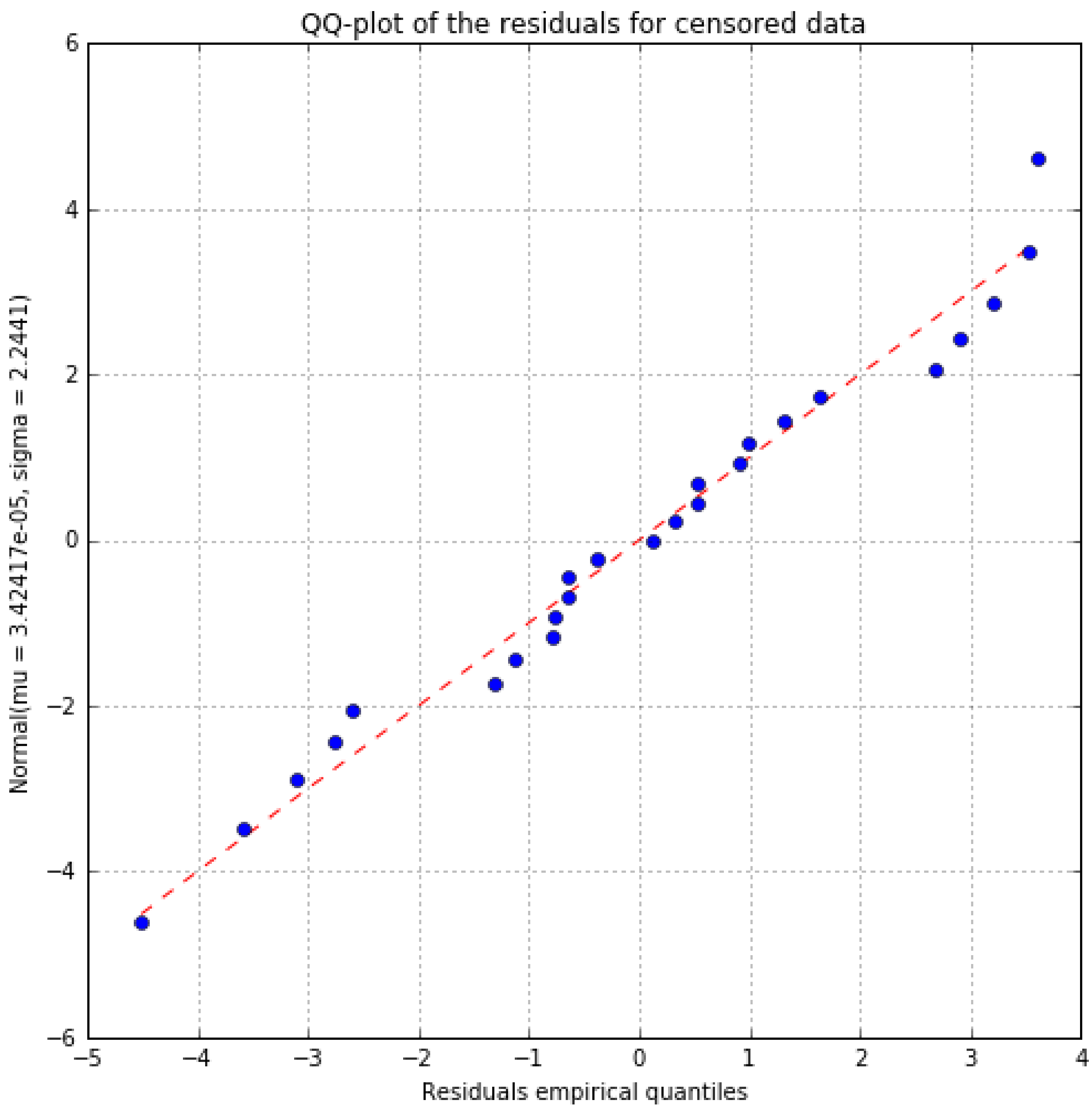

QQ-plot of the residuals
Normal(mu = -1.62004e-14, sigma = 2.2441)
Residuals empirical quantiles
6
4
2
0
−2
−4
−6
−5
−4
−3
−2
−1
0
1
2
3
4

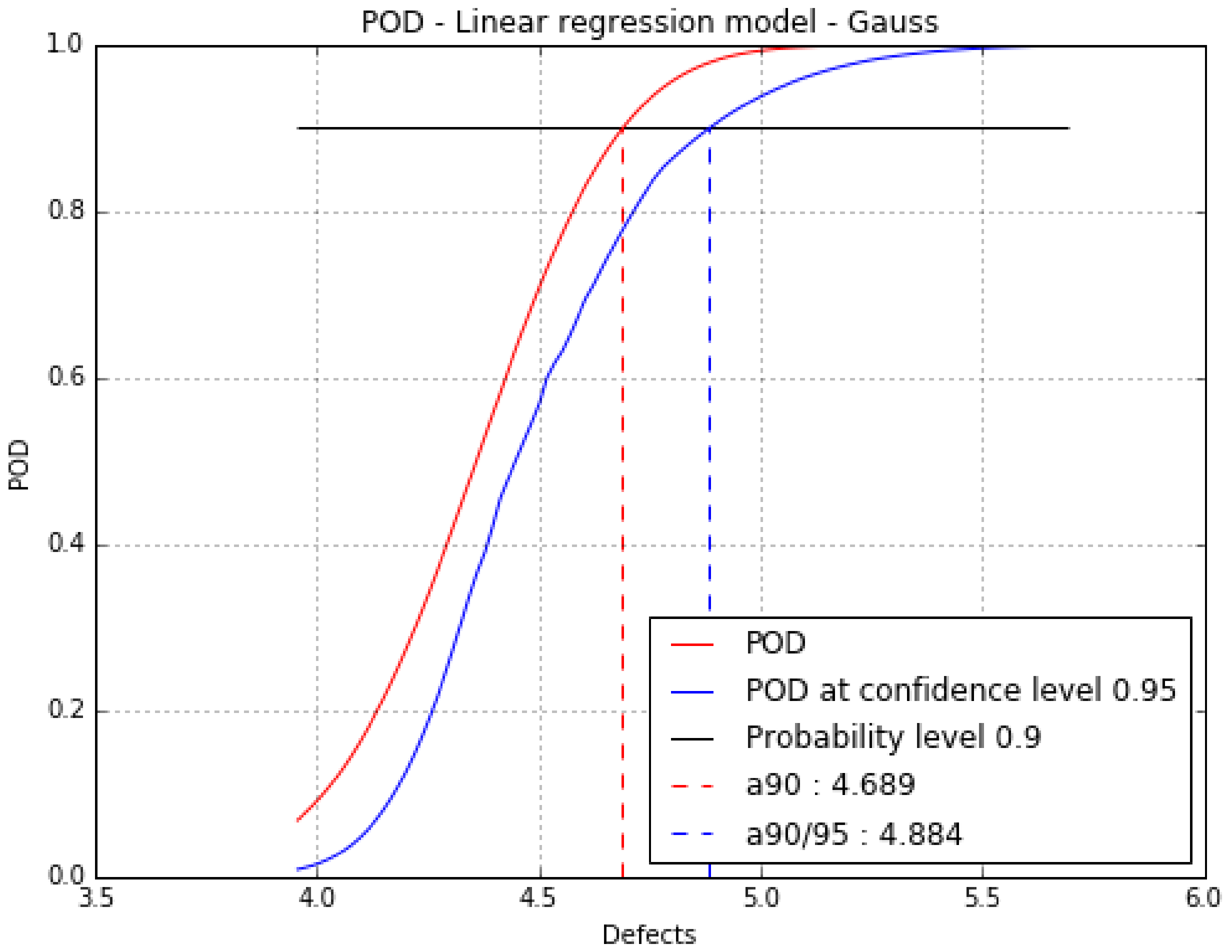

Linear regression model for censored data
Data
Linear model
Box Cox (signals)
Defects
50
45
40
35
30
3.5
4.0
4.5
5.0
5.5
6.0

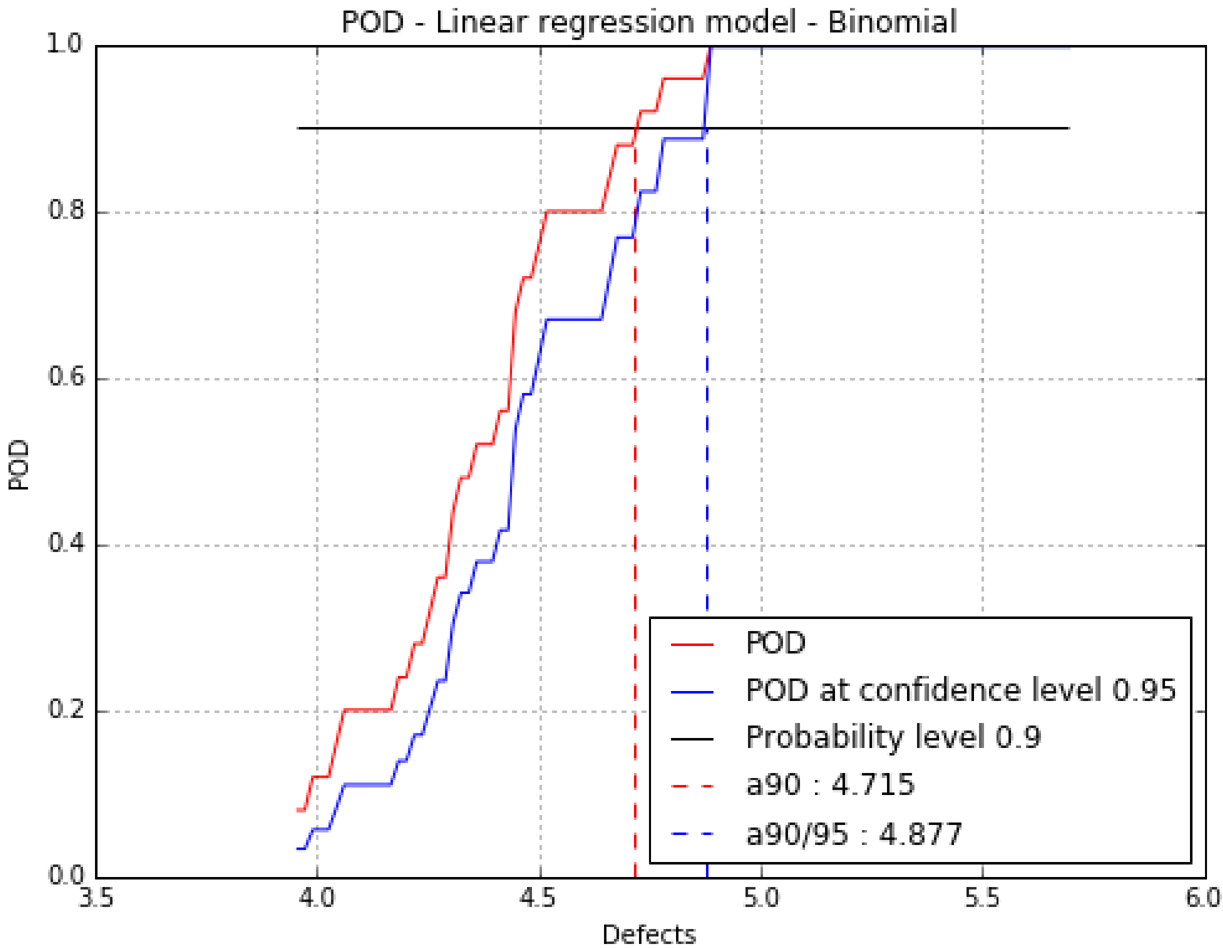


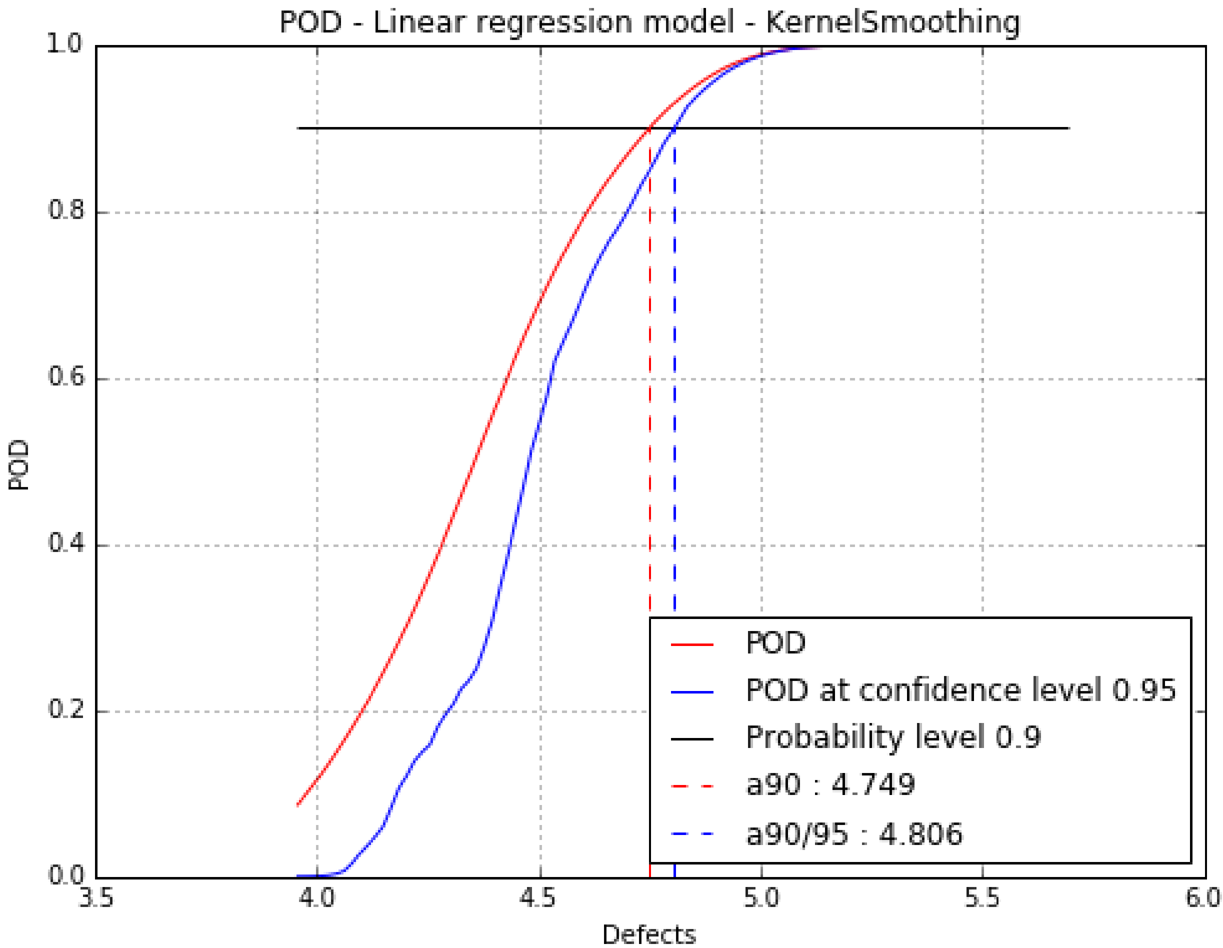
Residuals for censored data
Residuals dispersion
Defects
4
3
2
1
0
−1
−2
−3
−4
−5
3.5
4.0
4.5
5.0
5.5
6.0

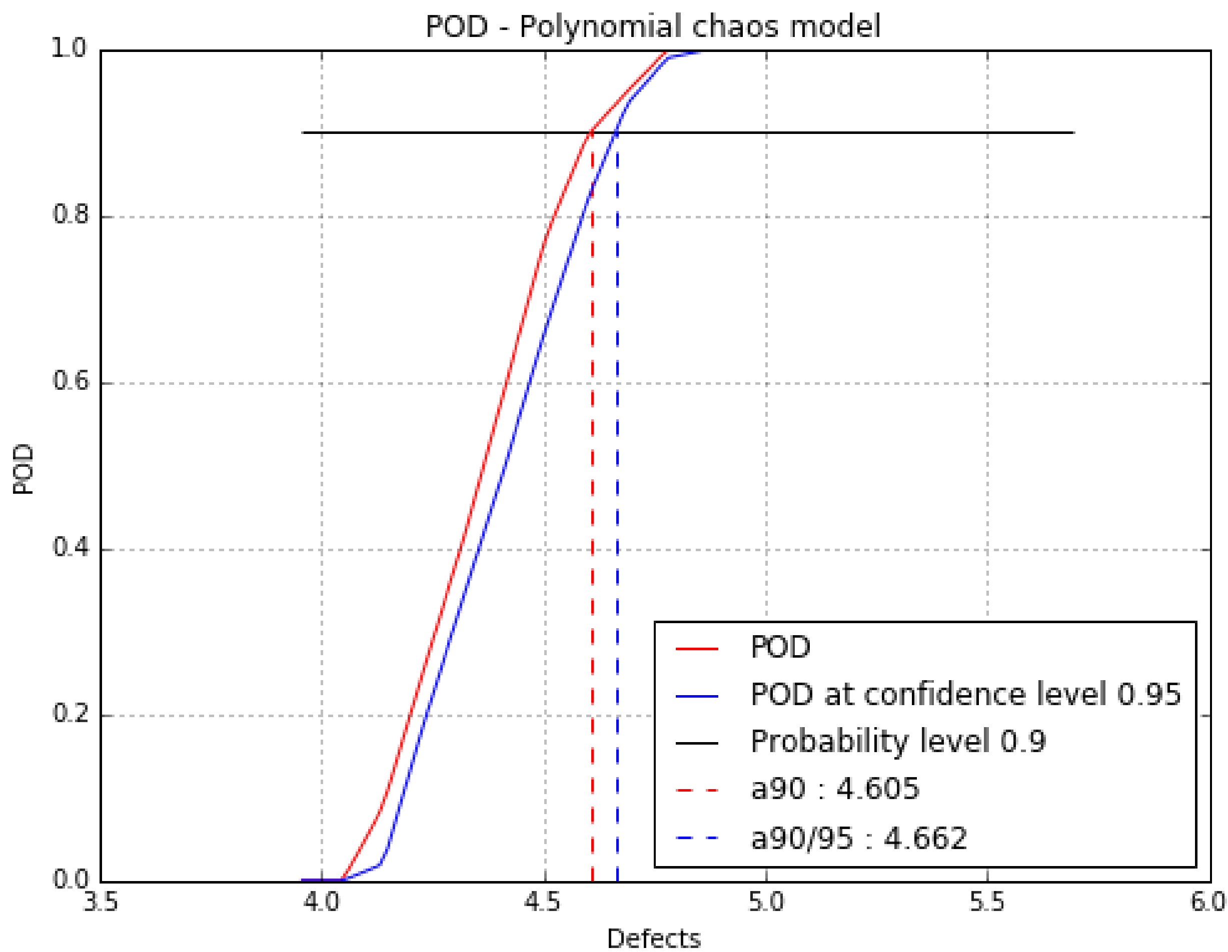

Residuals distribution for censored data
Normal(mu = 3.42417e-05, sigma = 2.2441)
Residuals histogram
PDF
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
−10
−5
0
5
10
Defect realizations

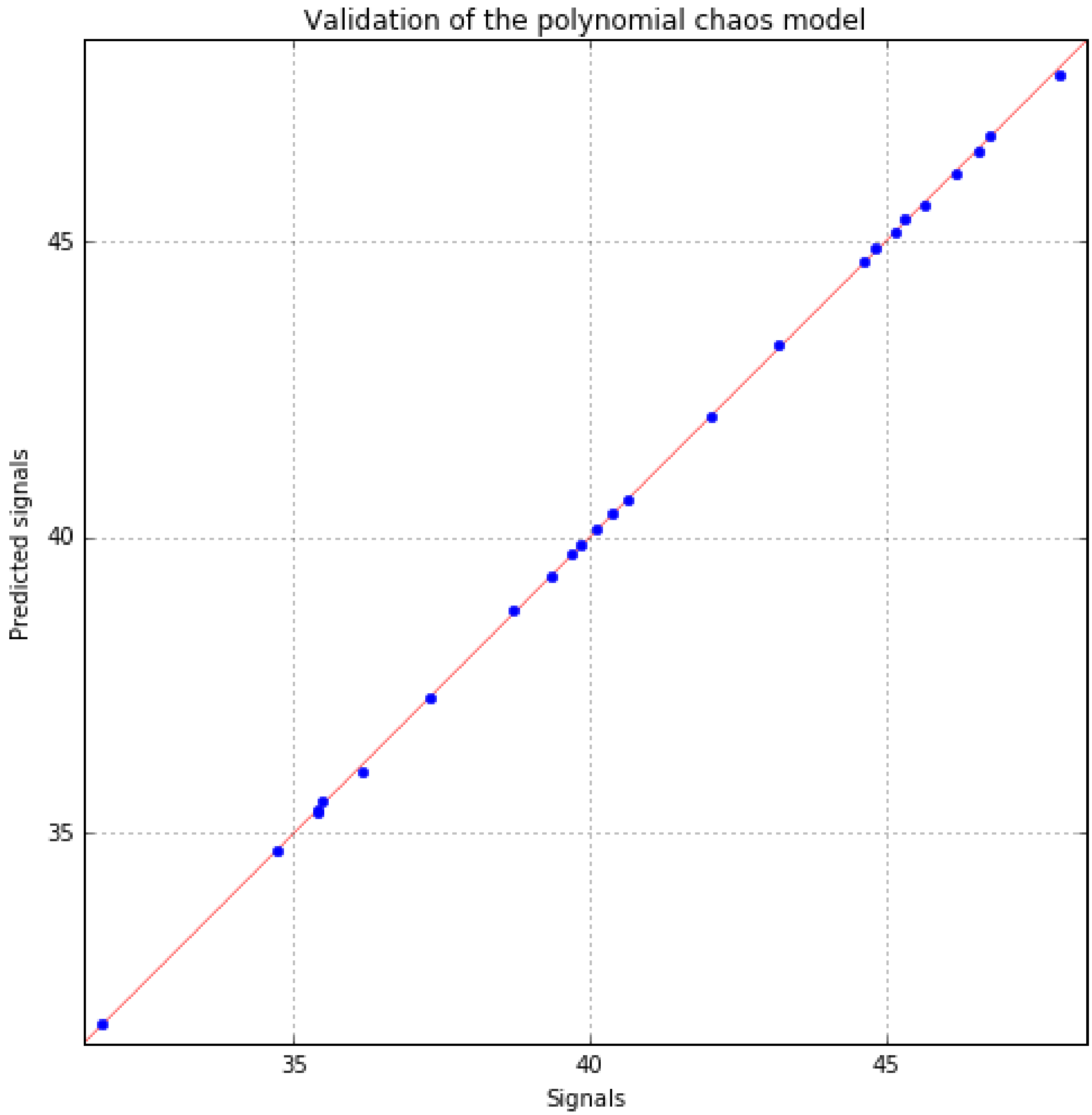

QQ-plot of the residuals for censored data
Normal(mu = 3.42417e-05, sigma = 2.2441)
Residuals empirical quantiles
6
4
2
0
−2
−4
−6
−5
−4
−3
−2
−1
0
1
2
3
4

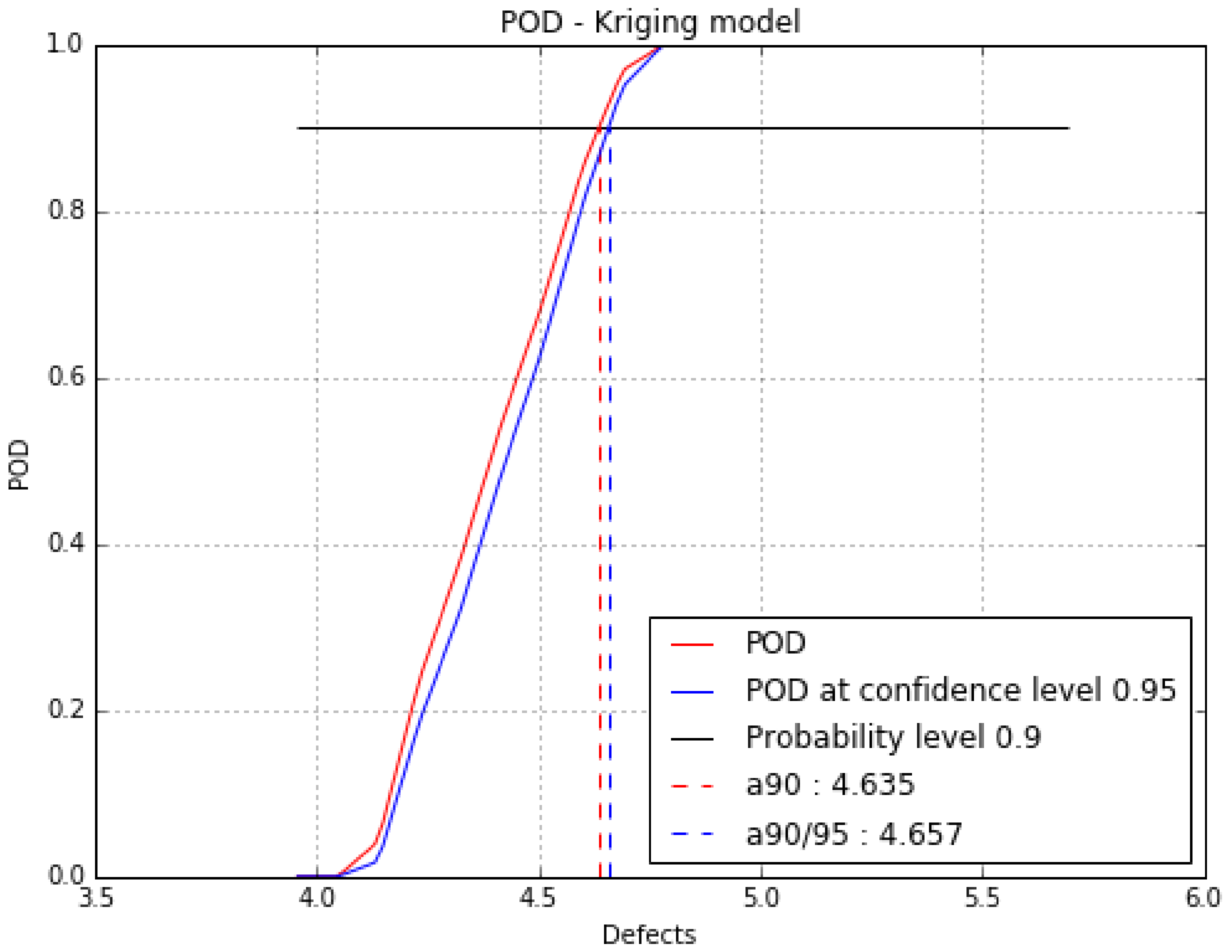

POD - Linear regression model - Gauss
1.0
0.8
0.6
0.4
0.2
0.0
POD
3.5
4.0
4.5
5.0
5.5
6.0
Defects
POD
POD at confidence level 0.95
Probability level 0.9
a90 : 4.689
a90/95 : 4.884


POD - Linear regression model - Binomial
1.0
0.8
0.6
0.4
0.2
0.0
POD
3.5
4.0
4.5
5.0
5.5
6.0
Defects
POD
POD at confidence level 0.95
Probability level 0.9
a90 : 4.715
a90/95 : 4.877

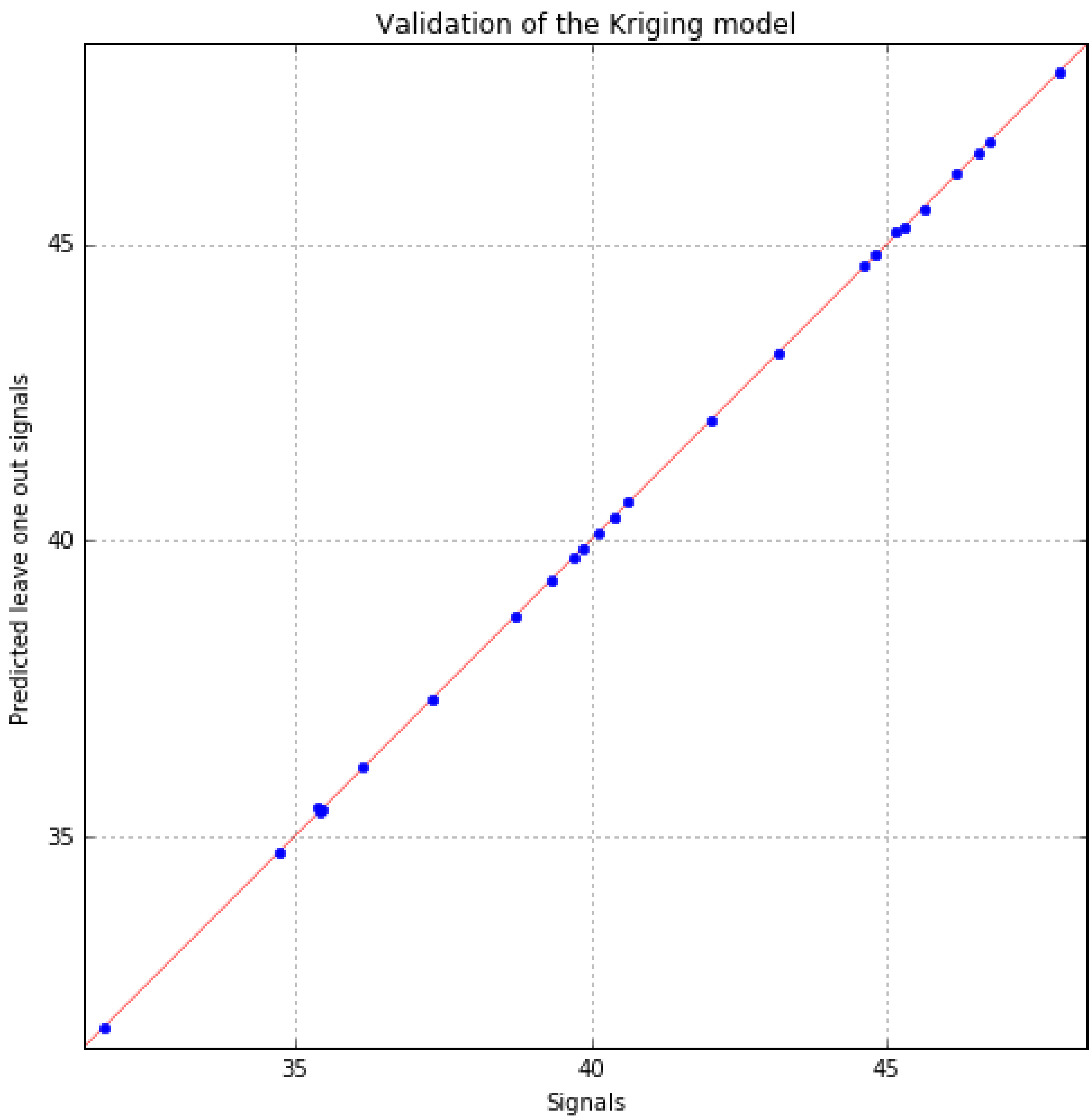

POD - Linear regression model - KernelSmoothing
POD
Defects
1.0
0.8
0.6
0.4
0.2
0.0
3.5
4.0
4.5
5.0
5.5
6.0
POD
POD at confidence level 0.95
Probability level 0.9
a90 : 4.749
a90/95 : 4.806


POD - Polynomial chaos model
1.0
0.8
0.6
0.4
0.2
0.0
POD
3.5
4.0
4.5
5.0
5.5
6.0
Defects
POD
POD at confidence level 0.95
Probability level 0.9
a90 : 4.605
a90/95 : 4.662


Validation of the polynomial chaos model
Predicted signals
Signals
35
40
45
35
40
45



POD - Kriging model
POD
Defects
1.0
0.8
0.6
0.4
0.2
0.0
3.5
4.0
4.5
5.0
5.5
6.0
POD
POD at confidence level 0.95
Probability level 0.9
a90 : 4.635
a90/95 : 4.657



ipynb source code

1.2.9 Adaptive Signal POD using Kriging

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
import numpy as np
```

Generate data

```
inputSample = ot.NumericalSample(
    [[4.59626812e+00, 7.46143339e-02, 1.02231538e+00, 8.60042277e+01],
    [4.14315790e+00, 4.20801346e-02, 1.05874908e+00, 2.65757364e+01],
    [4.76735111e+00, 3.72414824e-02, 1.05730385e+00, 5.76058433e+01],
    [4.82811977e+00, 2.49997658e-02, 1.06954641e+00, 2.54461380e+01],
    [4.48961094e+00, 3.74562922e-02, 1.04943946e+00, 6.19483646e+00],
    [5.05605334e+00, 4.87599783e-02, 1.06520409e+00, 3.39024904e+00],
    [5.69679328e+00, 7.74915877e-02, 1.04099514e+00, 6.50990466e+01],
    [5.10193991e+00, 4.35520544e-02, 1.02502536e+00, 5.51492592e+01],
    [4.04791970e+00, 2.38565932e-02, 1.01906882e+00, 2.07875350e+01],
    [4.66238956e+00, 5.49901237e-02, 1.02427200e+00, 1.45661275e+01],
    [4.86634219e+00, 6.04693570e-02, 1.08199374e+00, 1.05104730e+00],
    [4.13519347e+00, 4.45225831e-02, 1.01900124e+00, 5.10117047e+01],
    [4.92541940e+00, 7.87692335e-02, 9.91868726e-01, 8.32302238e+01],
    [4.70722074e+00, 6.51799251e-02, 1.10608515e+00, 3.30181002e+01],
    [4.29040932e+00, 1.75426222e-02, 9.75678838e-01, 2.28186756e+01],
    [4.89291400e+00, 2.34997929e-02, 1.07669835e+00, 5.38926138e+01],
    [4.44653744e+00, 7.63175936e-02, 1.06979154e+00, 5.19109415e+01],
    [3.99977452e+00, 5.80430585e-02, 1.01850716e+00, 7.61988190e+01],
    [3.95491570e+00, 1.09302814e-02, 1.03687664e+00, 6.09981789e+01],
    [5.16424368e+00, 2.69026464e-02, 1.06673711e+00, 2.88708887e+01],
    [5.30491620e+00, 4.53802273e-02, 1.06254792e+00, 3.03856837e+01],
    [4.92809155e+00, 1.20616369e-02, 1.00700410e+00, 7.02512744e+00],
    [4.68373805e+00, 6.26028935e-02, 1.05152117e+00, 4.81271603e+01],
    [5.32381954e+00, 4.33013582e-02, 9.90522007e-01, 6.56015973e+01],
    [4.35455857e+00, 1.23814619e-02, 1.01810539e+00, 1.10769534e+01]])

signals = ot.NumericalSample(
    [[ 37.305445], [ 35.466919], [ 43.187991], [ 45.305165], [ 40.121222], [ 44.
↪609524],
     [ 45.14552 ], [ 44.80595 ], [ 35.414039], [ 39.851778], [ 42.046049], [ 34.73469␣
↪],
     [ 39.339349], [ 40.384559], [ 38.718623], [ 46.189709], [ 36.155737], [ 31.
↪768369],
     [ 35.384313], [ 47.914584], [ 46.758537], [ 46.564428], [ 39.698493], [ 45.
↪636588],
     [ 40.643948]])

# detection threshold
detection = 38

# Select point as initial DOE
inputDOE = inputSample[:7]
outputDOE = signals[:7]

# simulate the true physical model
basis = ot.ConstantBasisFactory(4).build()
covModel = ot.SquaredExponential([5.03148,13.9442,20,20], [15.1697])
krigingModel = ot.KrigingAlgorithm(inputSample, signals, basis, covModel)
krigingModel.run()
physicalModel = krigingModel.getResult().getMetaModel()
```

Create the Adaptive Signal POD with Kriging model

This method aims at improving the quality of the Kriging model where the accuracy of the computed POD is the lowest.

As this method is time consuming, it is more efficient to reduce the area of the defect size only in the most interesting part. To do that, an initial POD study can be run.

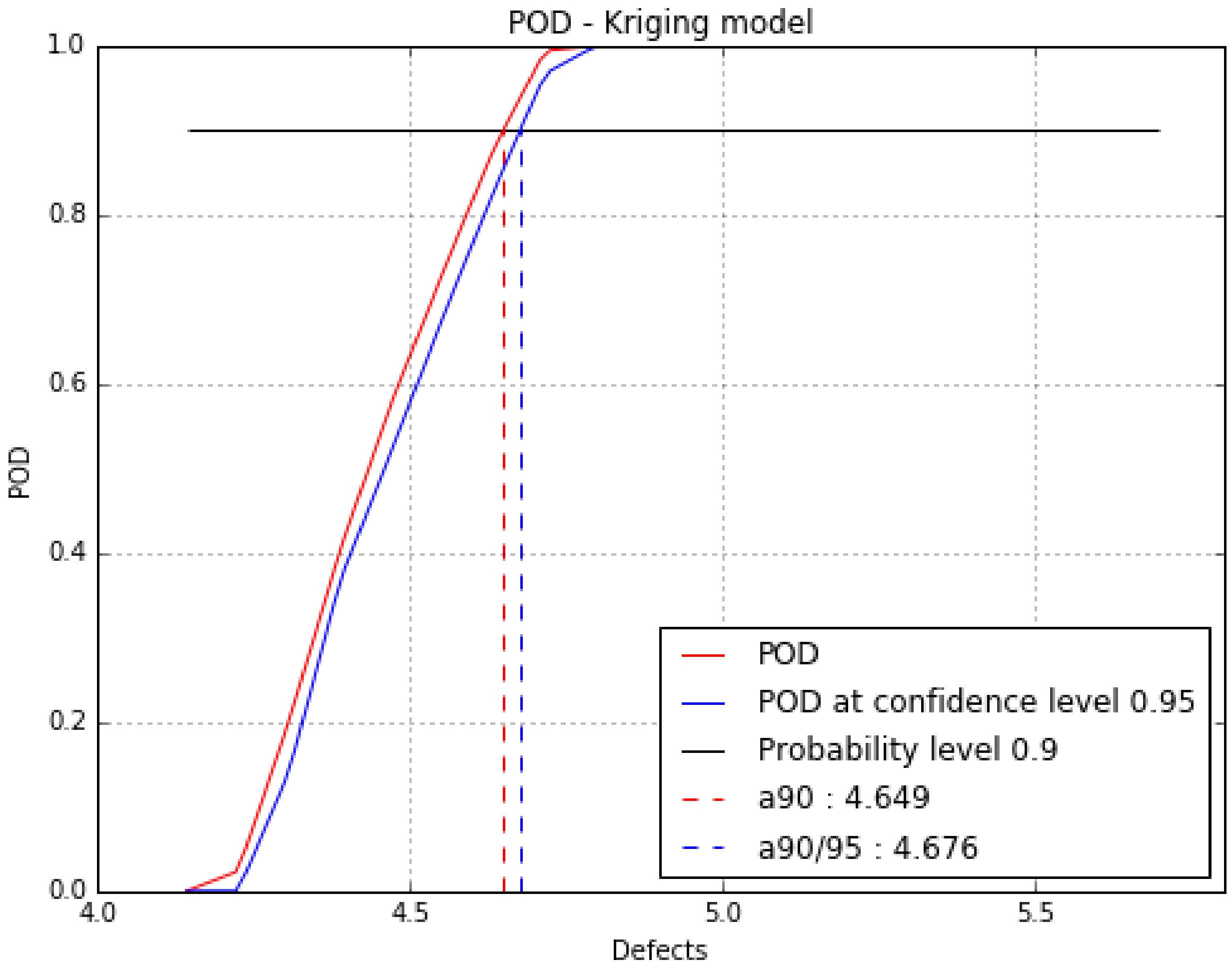
Run an initial POD study with the kriging technique

```
initialPOD = otpod.KrigingPOD(inputDOE, outputDOE, detection)
%time initialPOD.run()
```

```
Start optimizing covariance model parameters...
Kriging optimizer completed
kriging validation Q2 (>0.9): 0.8851
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 1min 19s, sys: 8.58 s, total: 1min 28s
Wall time: 1min 26s
```

```
fig, ax = initialPOD.drawPOD(0.9, 0.95)
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



Based on this study, the interesting part for the defects ranges from 4.2 to 4.8. The adaptive signal algorithm will be then reduced to this area.

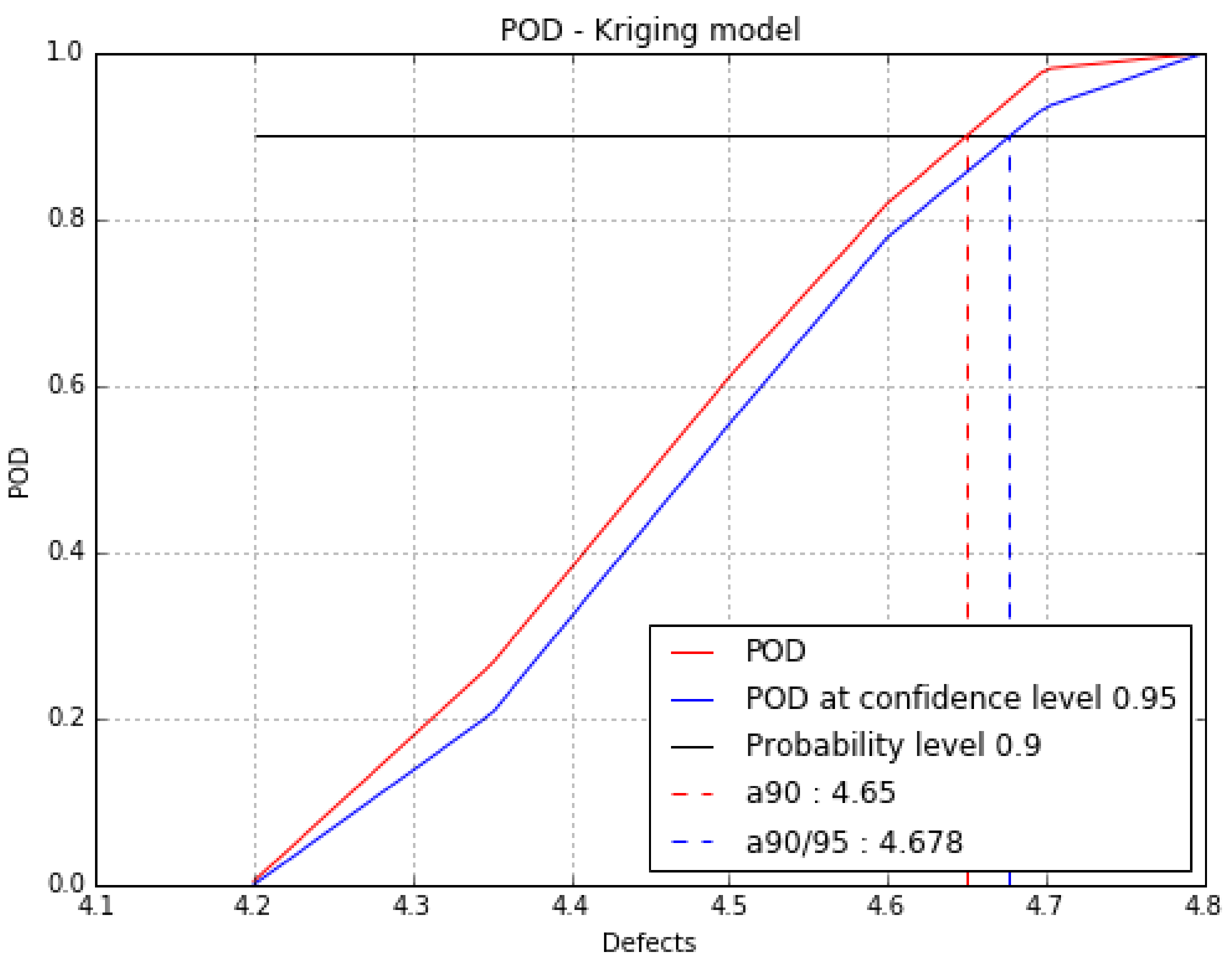
Run the adaptive algorithm

Computing the criterion is costly so the sampling and simulation size are reduced.

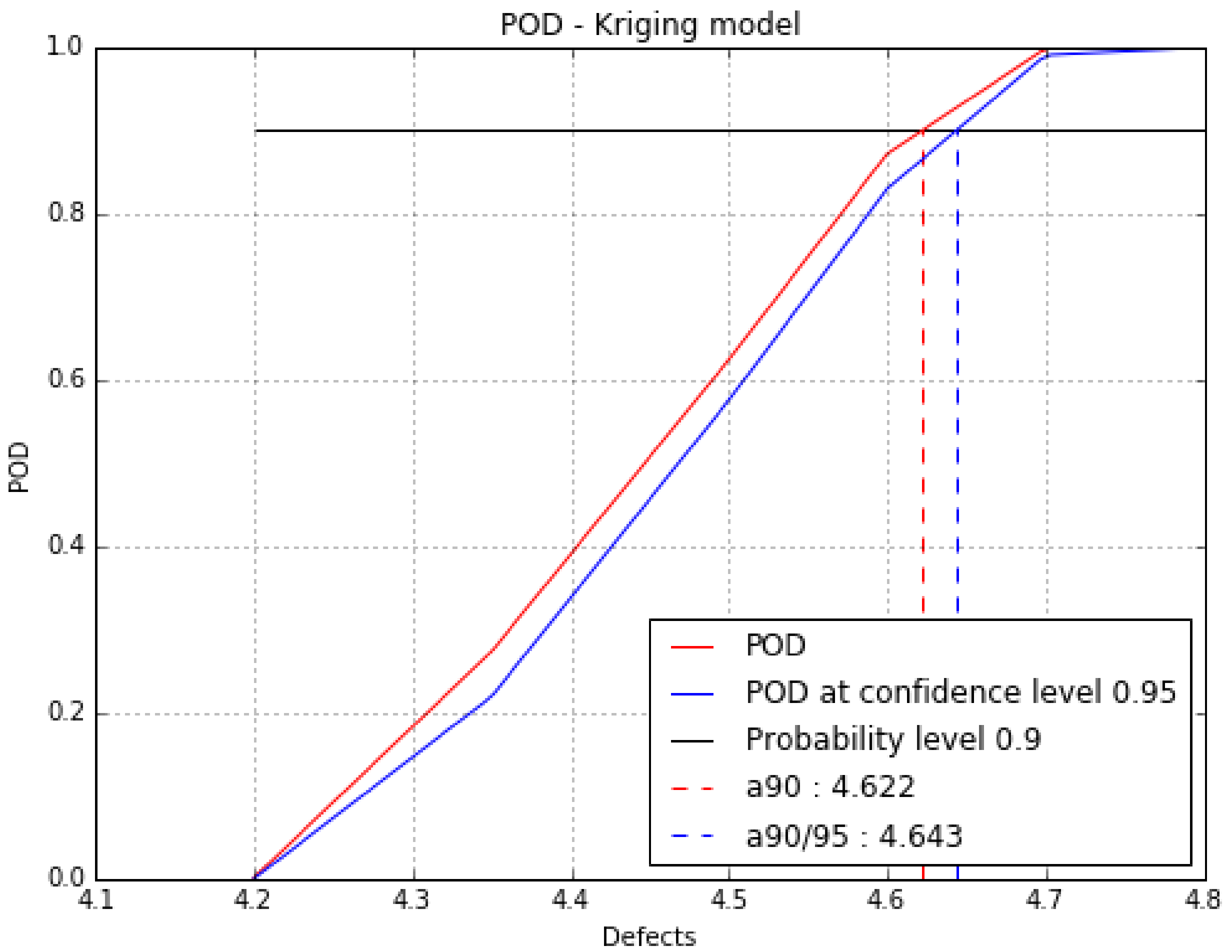
```
# set the number of iterations
nIteration = 5

# Creating the adaptivePOD object
adaptivePOD = otpod.AdaptiveSignalPOD(inputDOE, outputDOE, physicalModel, nIteration,␣
↪detection)

# Change the range for the defect sizes between 4.2 and 4.8
adaptivePOD.setDefectSizes([4.2, 4.35, 4.5, 4.6, 4.7, 4.8])

# We can change also the number of candidate points for which the critertion is␣
↪computed
adaptivePOD.setCandidateSize(100)
# we can change the sample size of the Monte Carlo simulation
adaptivePOD.setSamplingSize(500) # default is 5000
# we can also change the size of the simulation to compute the confidence interval
```

```
adaptivePOD.setSimulationSize(100) # default is 1000

# The current iteration POD graph can be displayed with multiple options :
## with or without the confidence level curve
## and with or without the intersection value at the given probability level
## if a directory is given, all graphs are saved as AdaptiveSignalPOD_i.png
adaptivePOD.setGraphActive(graphVerbose=True, probabilityLevel=0.9, confidenceLevel=0.
↪95,
                           directory='figure/')

%time adaptivePOD.run()
```

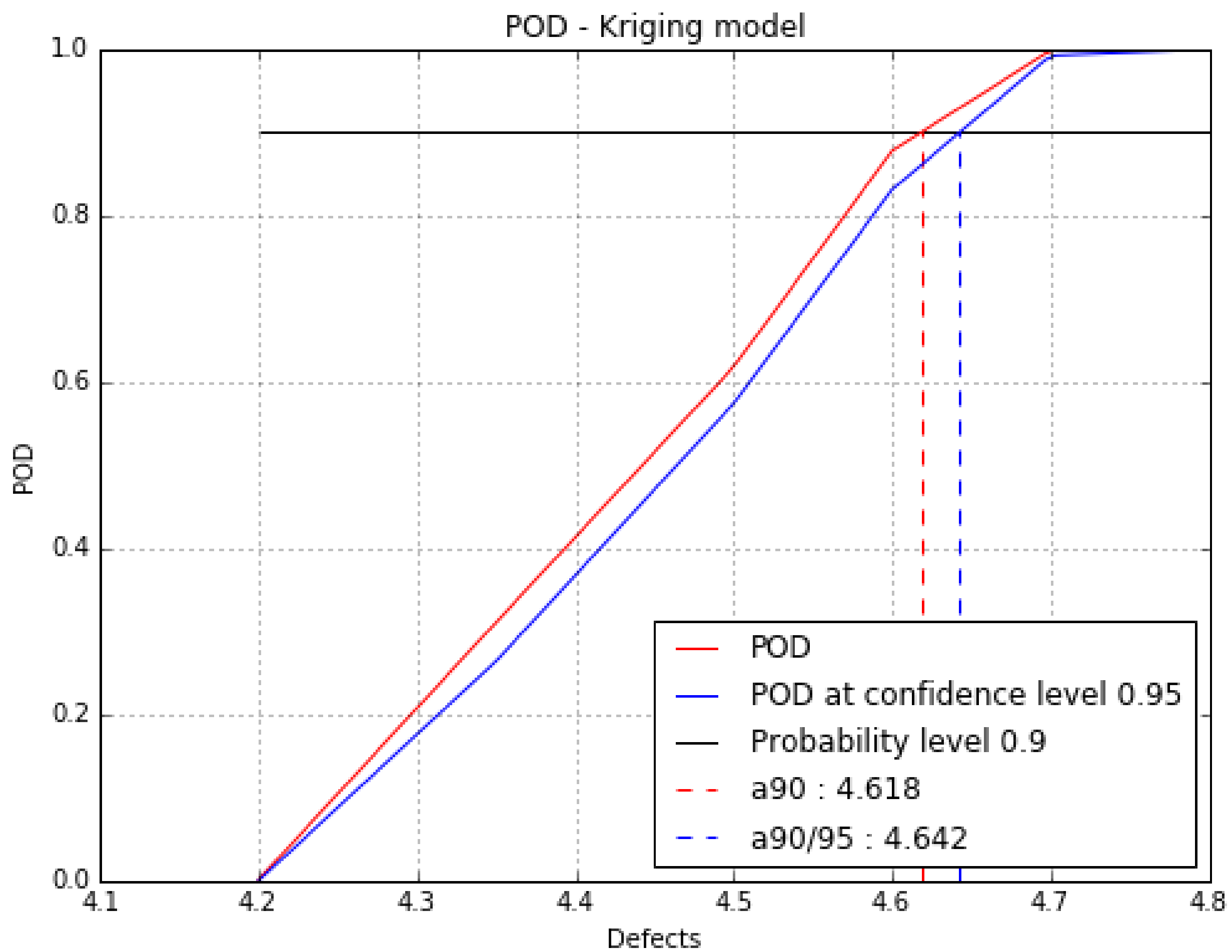
```
Building the kriging model
Optimization of the covariance model parameters...
Kriging validation Q2 (>0.9): 0.8851

Iteration : 1/5
Computing criterion: [==================================================] 100.00% Done
Criterion value : 0.0284
Added point : [4.51406,0.0721604,1.06696,28.5617]
Update the kriging model
Kriging validation Q2 (>0.9): 0.9626
```



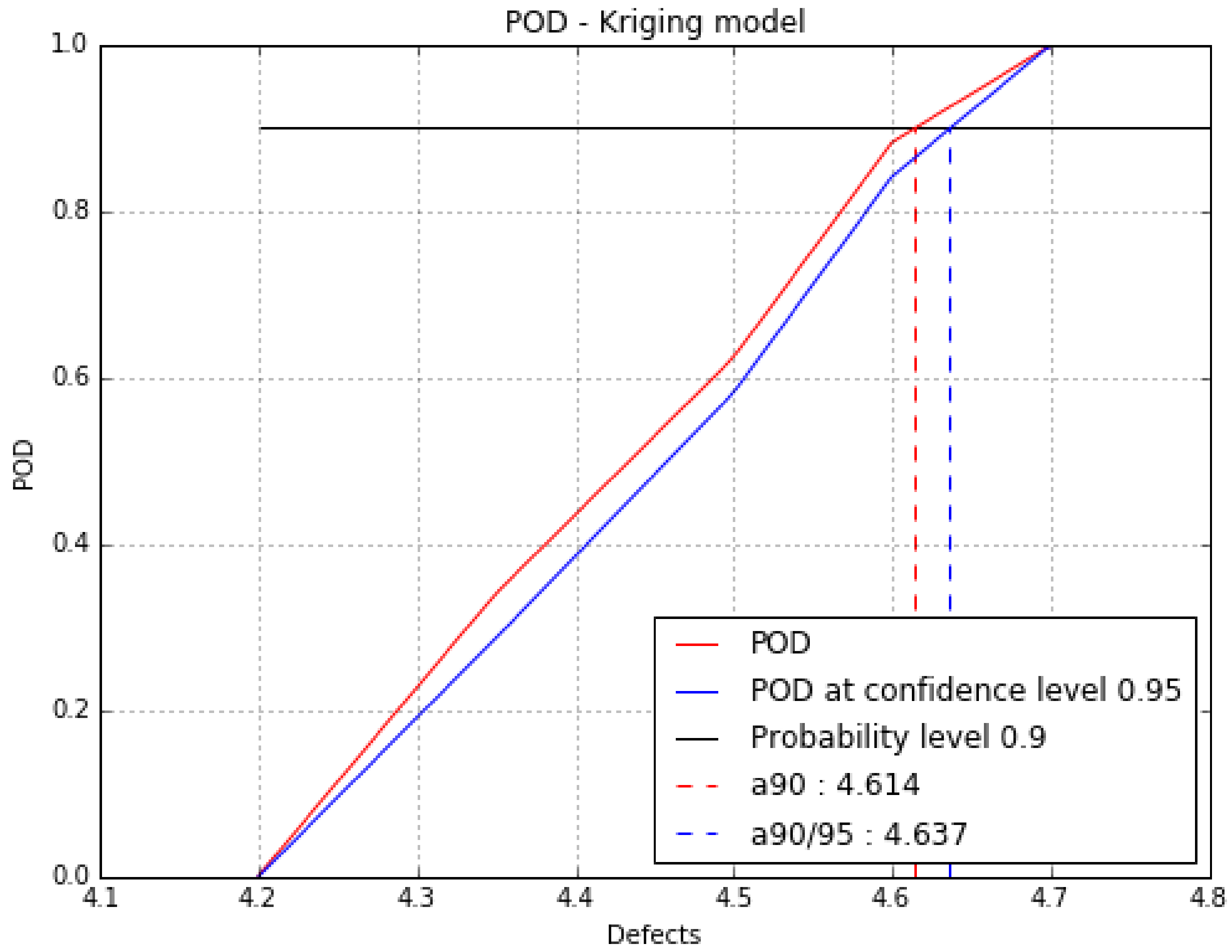
```
Iteration : 2/5
Computing criterion: [==================================================] 100.00% Done
Criterion value : 0.0229
Added point : [4.26562,0.0619081,1.03043,66.6416]
Update the kriging model
Kriging validation Q2 (>0.9): 0.9747
```



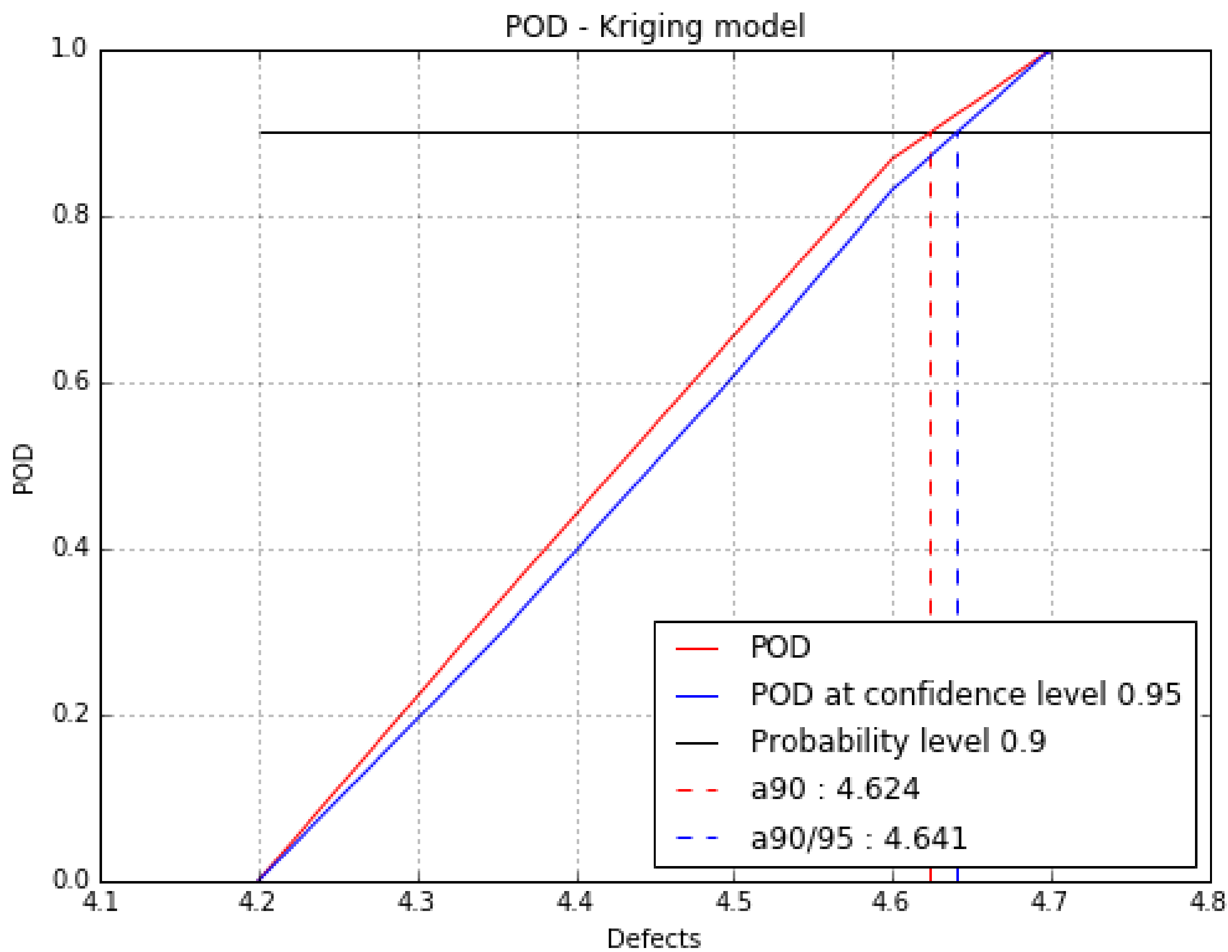

```
Iteration : 3/5
Computing criterion: [==================================================] 100.00% Done
Criterion value : 0.0216
Added point : [4.575,0.0315612,1.04003,34.3705]
Update the kriging model
Kriging validation Q2 (>0.9): 0.9789
```




```
Iteration : 4/5
Computing criterion: [==================================================] 100.00% Done
Criterion value : 0.0205
Added point : [4.77187,0.0701099,1.04077,33.0796]
Update the kriging model
Kriging validation Q2 (>0.9): 0.9698
```



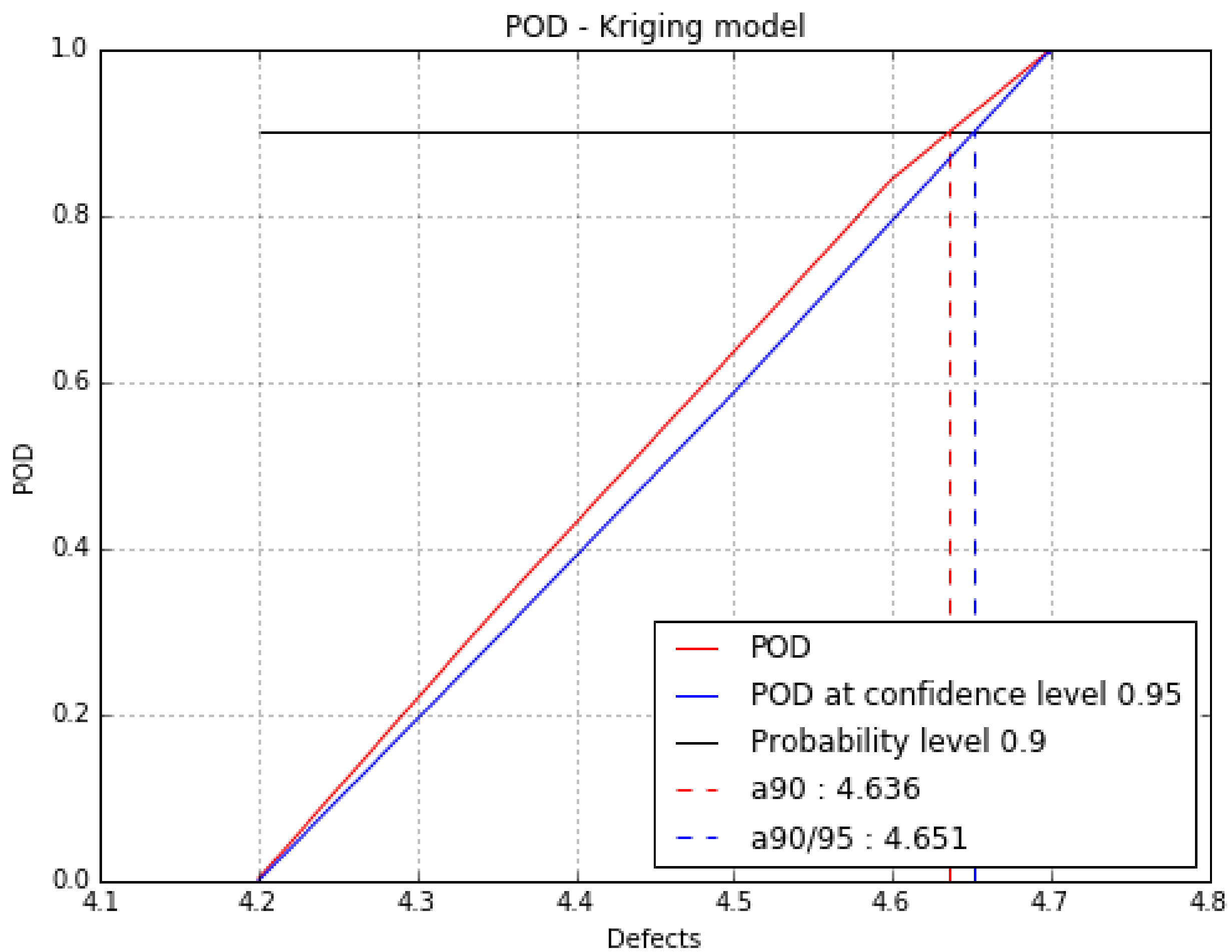

```
Iteration : 5/5
Computing criterion: [==================================================] 100.00% Done
Criterion value : 0.0190
Added point : [4.22813,0.0389429,1.0511,51.1515]
Update the kriging model
Kriging validation Q2 (>0.9): 0.9644
```




```
Start computing the POD with the last updated kriging model
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 8min 57s, sys: 1min 32s, total: 10min 29s
Wall time: 6min 14s
```

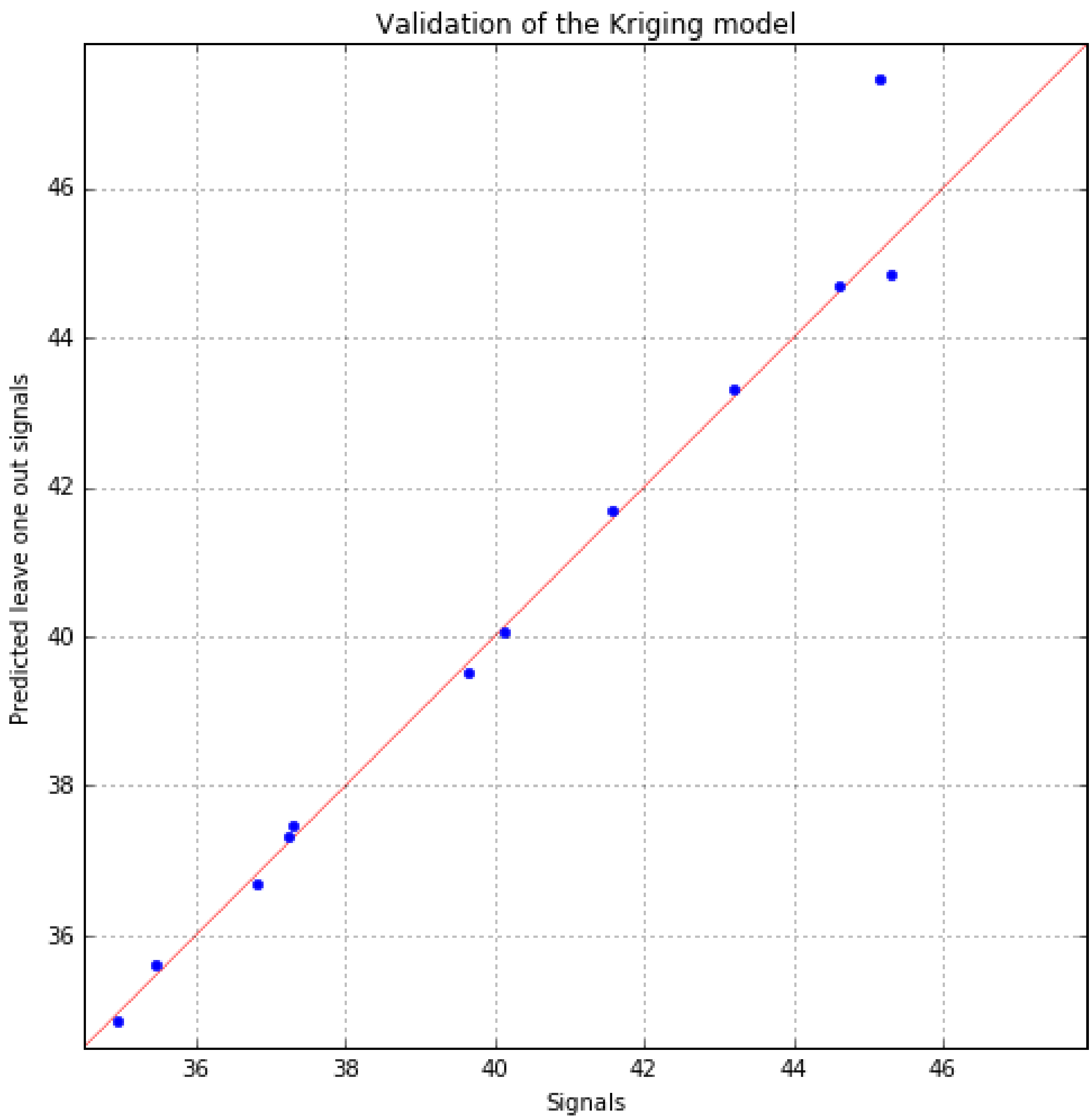
Display the POD result based on the adative kriging model

```
fig, ax = adaptivePOD.drawPOD(0.9, 0.95)
fig.show()
```



Diplay the validation graph

```
fid, ax = adaptivePOD.drawValidationGraph()
fig.show()
```



Quality improvement of the POD computation

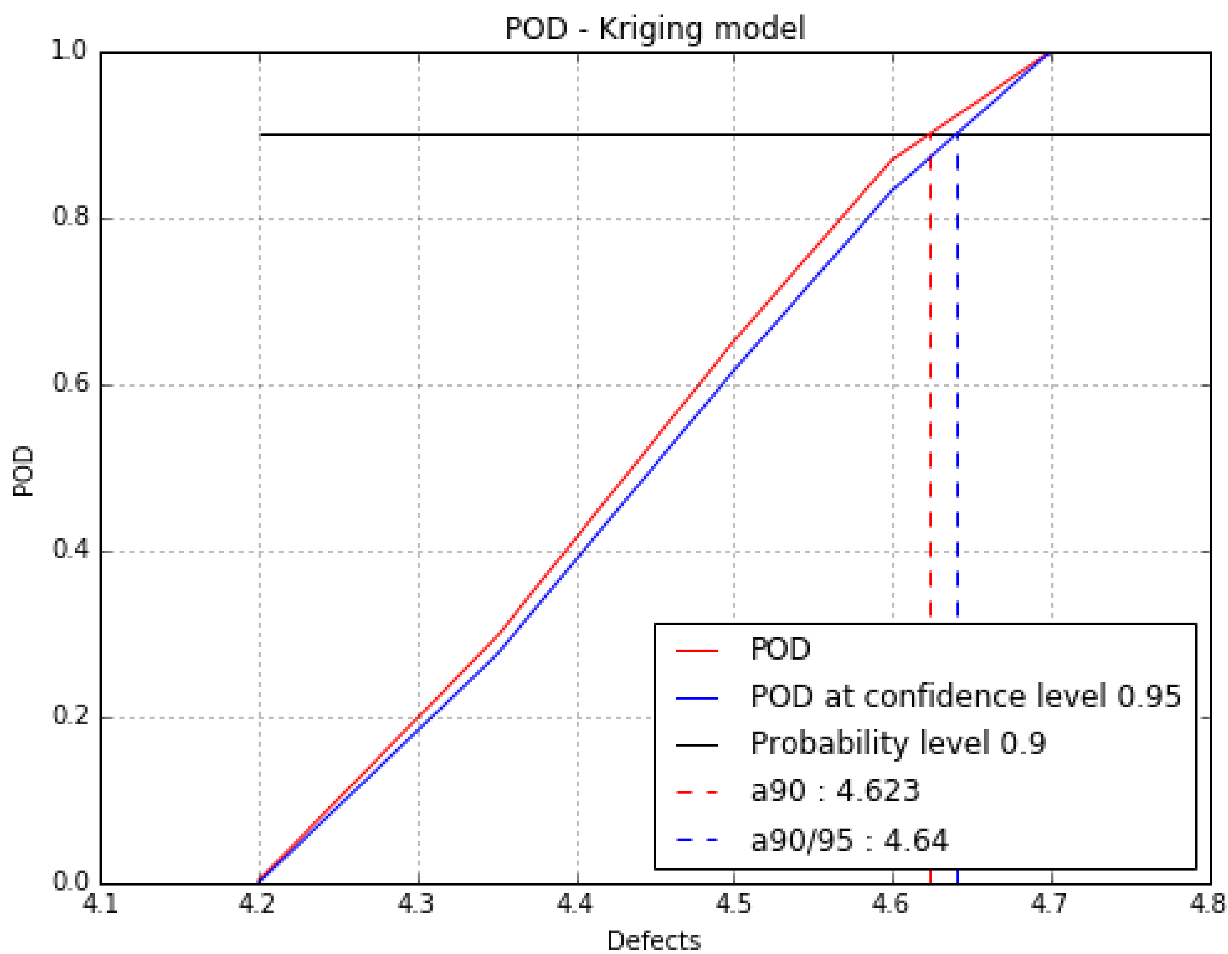
From the adaptive algorithm, the kriging result and the final DOE are available. As the number of simulations were reduced, we can compute again the POD with more accuracy than before if needed.

```
# get the kriging result and the final DOE from the adaptive algorithm
krigingRes = adaptivePOD.getKrigingResult()
inputfinal = adaptivePOD.getInputDOE()
outputfinal = adaptivePOD.getOutputDOE()
defectSizes = adaptivePOD.getDefectSizes()

# A new POD study is launch with the DOE values
```

```
finalPOD = otpod.KrigingPOD(inputfinal, outputfinal, detection)
finalPOD.setDefectSizes(defectSizes)
# The kriging model is already known so it is given to this study
finalPOD.setKrigingResult(krigingRes)
finalPOD.run()
```

```
kriging validation Q2 (>0.9): 0.9644
Computing POD per defect: [==================================================] 100.00
↪% Done
```

```
fig, ax = finalPOD.drawPOD(0.9, 0.95)
fig.show()
```



ipynb source code

1.2.10 Adaptive Hit Miss POD

```
%matplotlib inline
import numpy as np
import openturns as ot
import otpod
```

```
import warnings
warnings.filterwarnings("ignore")
```

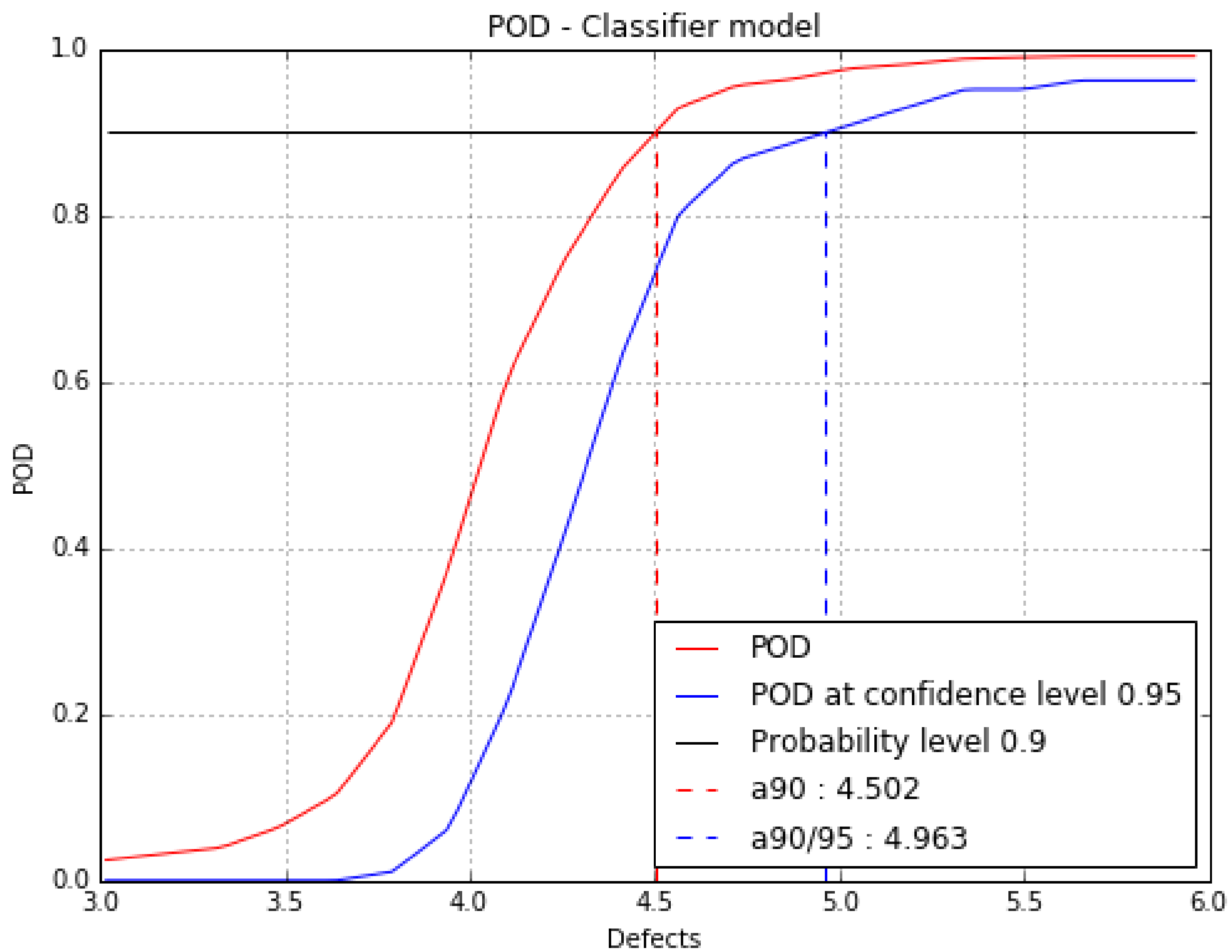
```
# The Hit/Miss function is build by executing "Make_HM.py"
# The function is called "MyHM"
%run Make_HM.py
```

```
-----------------------------------
The function 'MyHM' has been loaded
MyHM inputs dimension : 4
MyHM output dimension :
1 if signal > 33
0 if signal < 33
```

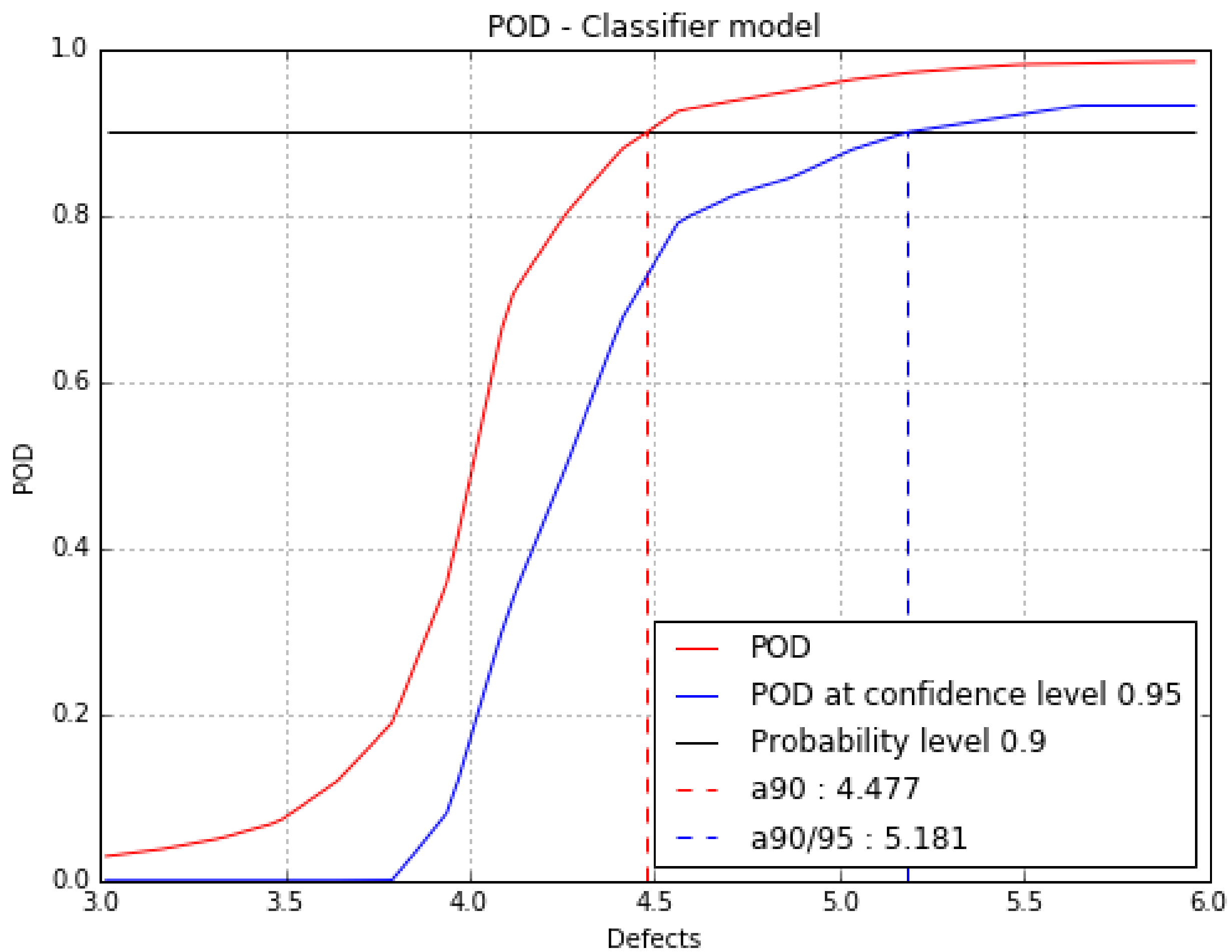
```
n_ini = 100
inputDOE = np.array([np.random.uniform(x_min[i],
             x_max[i],n_ini)
             for i in range(d)]).T
outputDOE = MyHM(inputDOE)
```

```
n_more = 30
# Add n_more points with the adaptive algorithm
# 5 points are added at each iteration
hitmiss_algo = res_algo = otpod.AdaptiveHitMissPOD(inputDOE, outputDOE, MyHM, n_more)
hitmiss_algo.setClassifierType("rf")
# Computation of the POD at each iteration activated and display the POD graph
hitmiss_algo.setGraphActive(True, 0.9, 0.95, 'figure/')
hitmiss_algo.run()
```

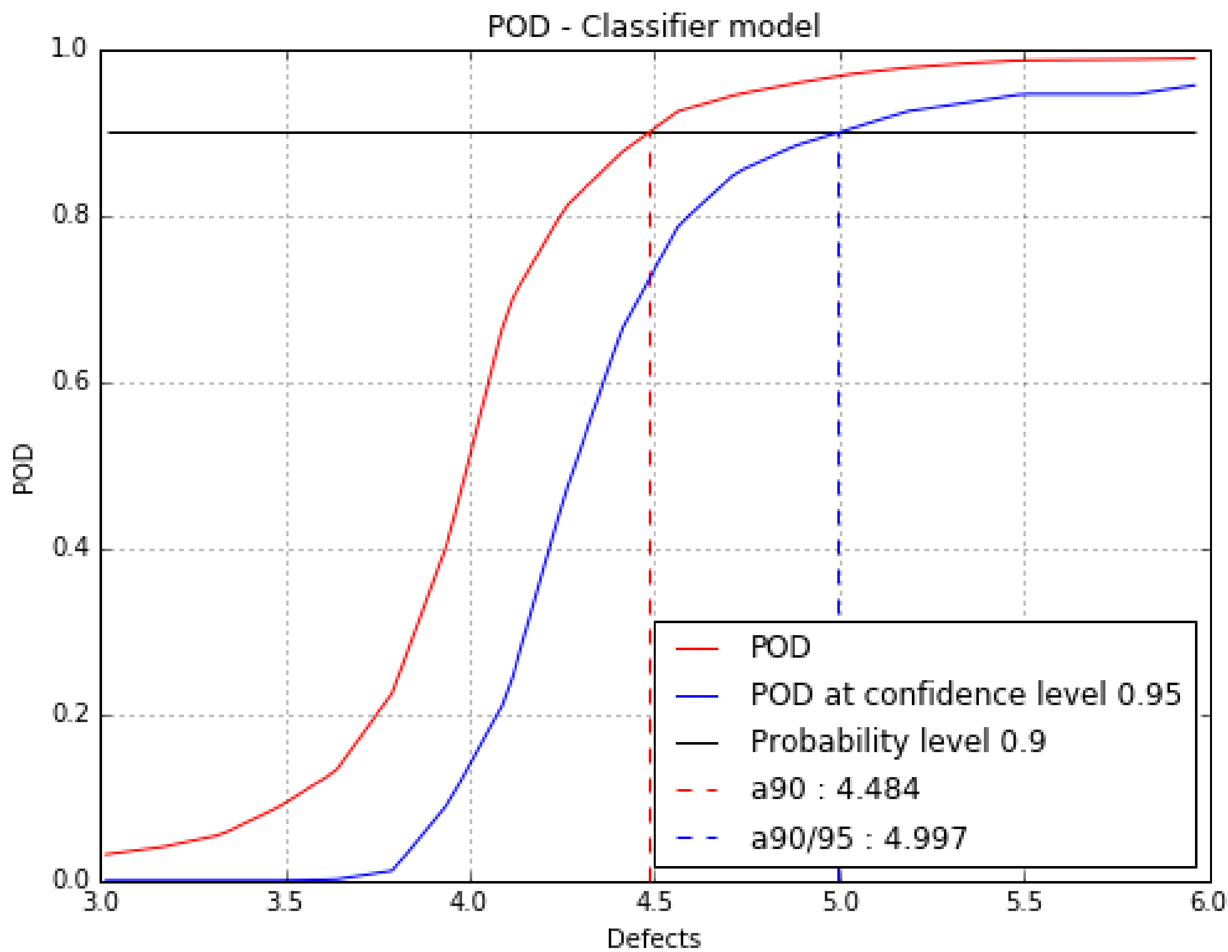
```
Building the classifier
Start the improvement loop
Adding points: [========-----------------------------------------] 16.67%
```



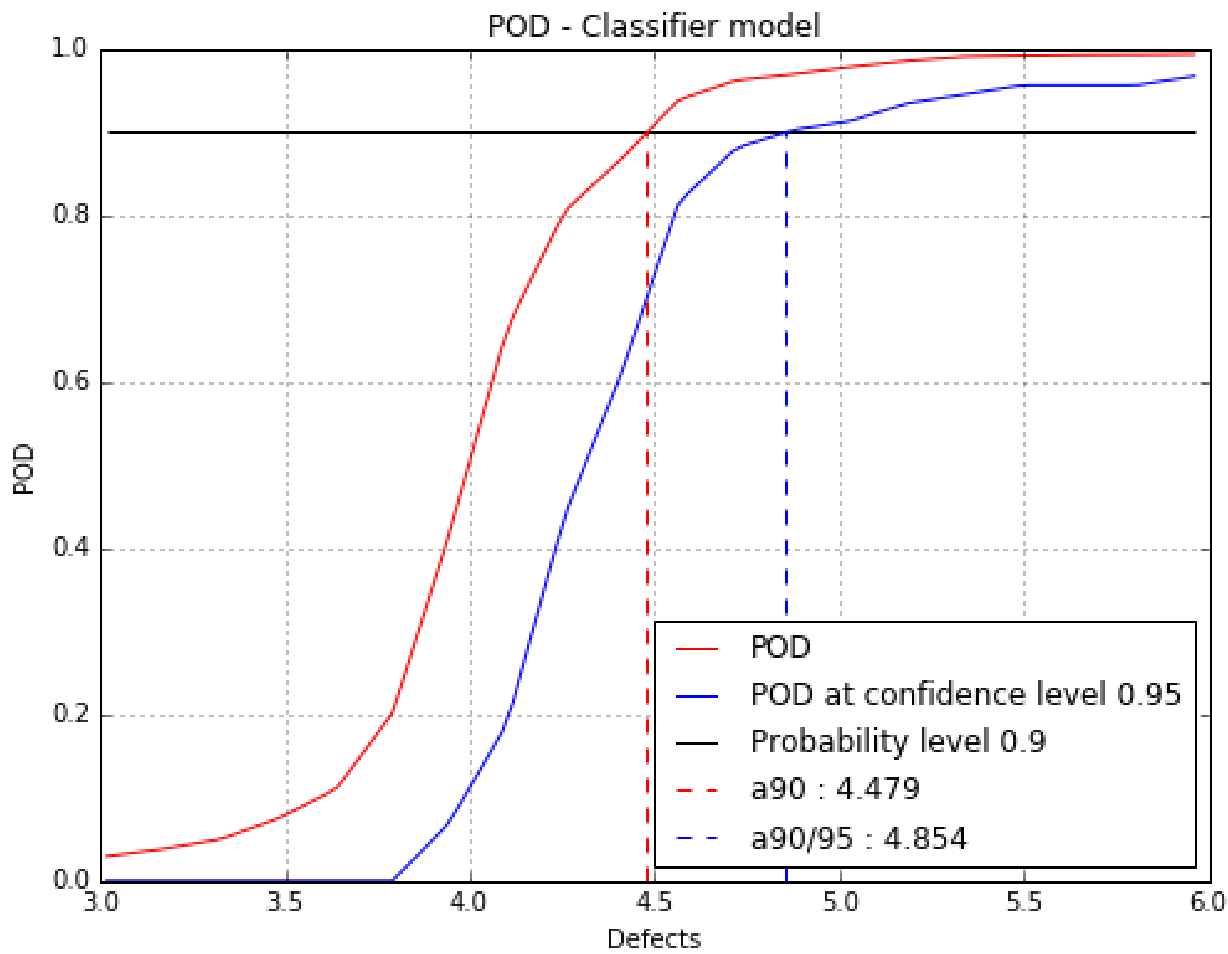

```
Adding points: [================-------------------------------] 33.33%
```



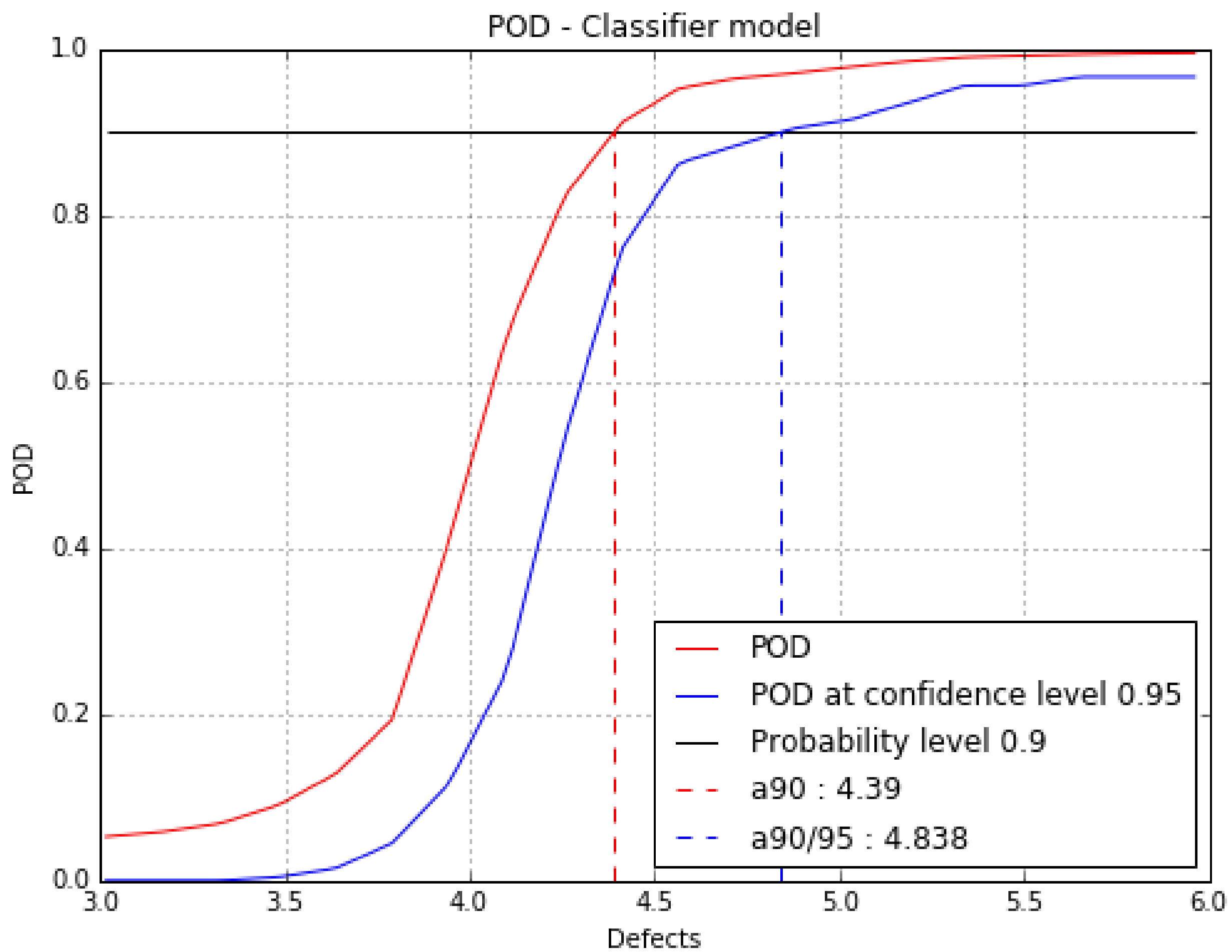

```
Adding points: [========================------------------------] 50.00%
```



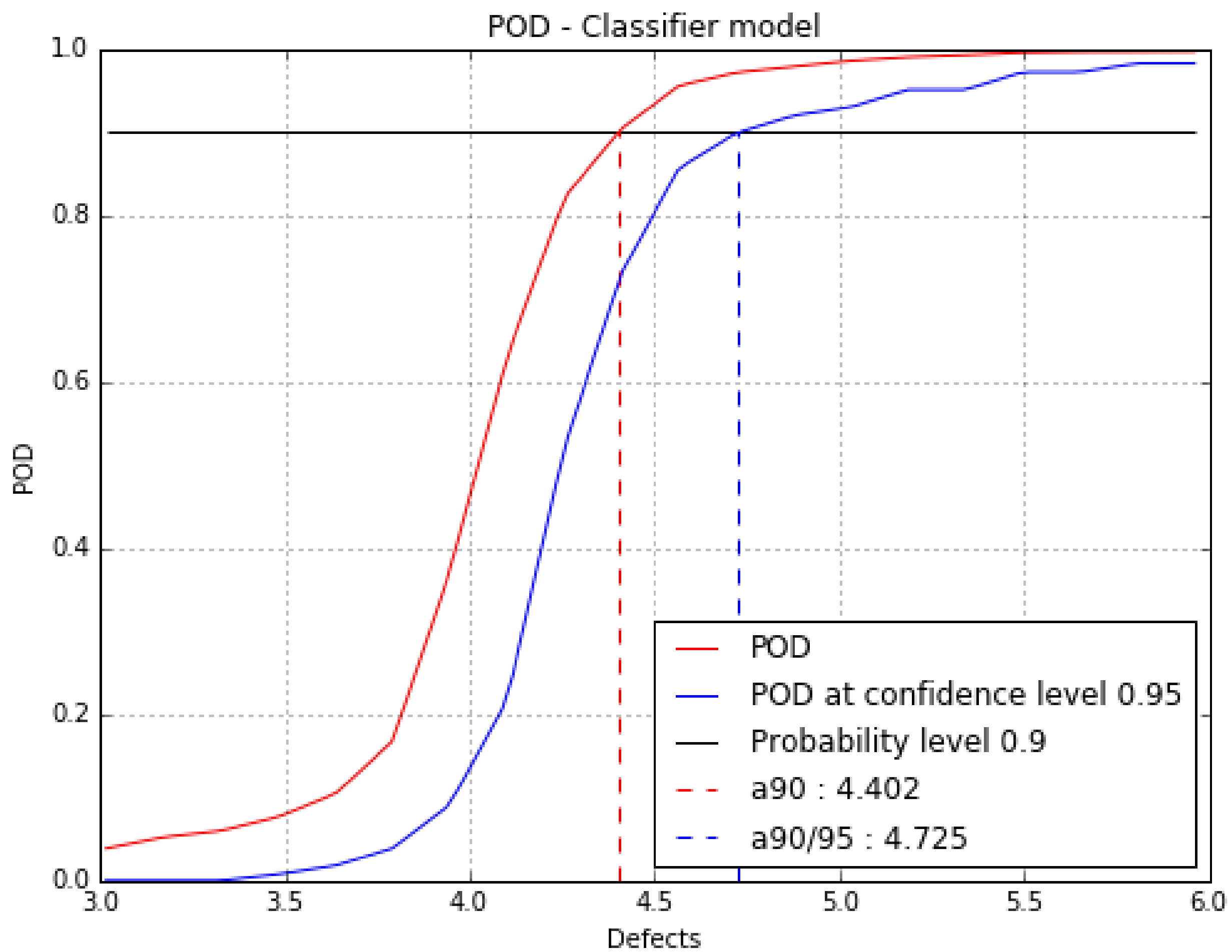

```
Adding points: [==============================------------------] 66.67%
```



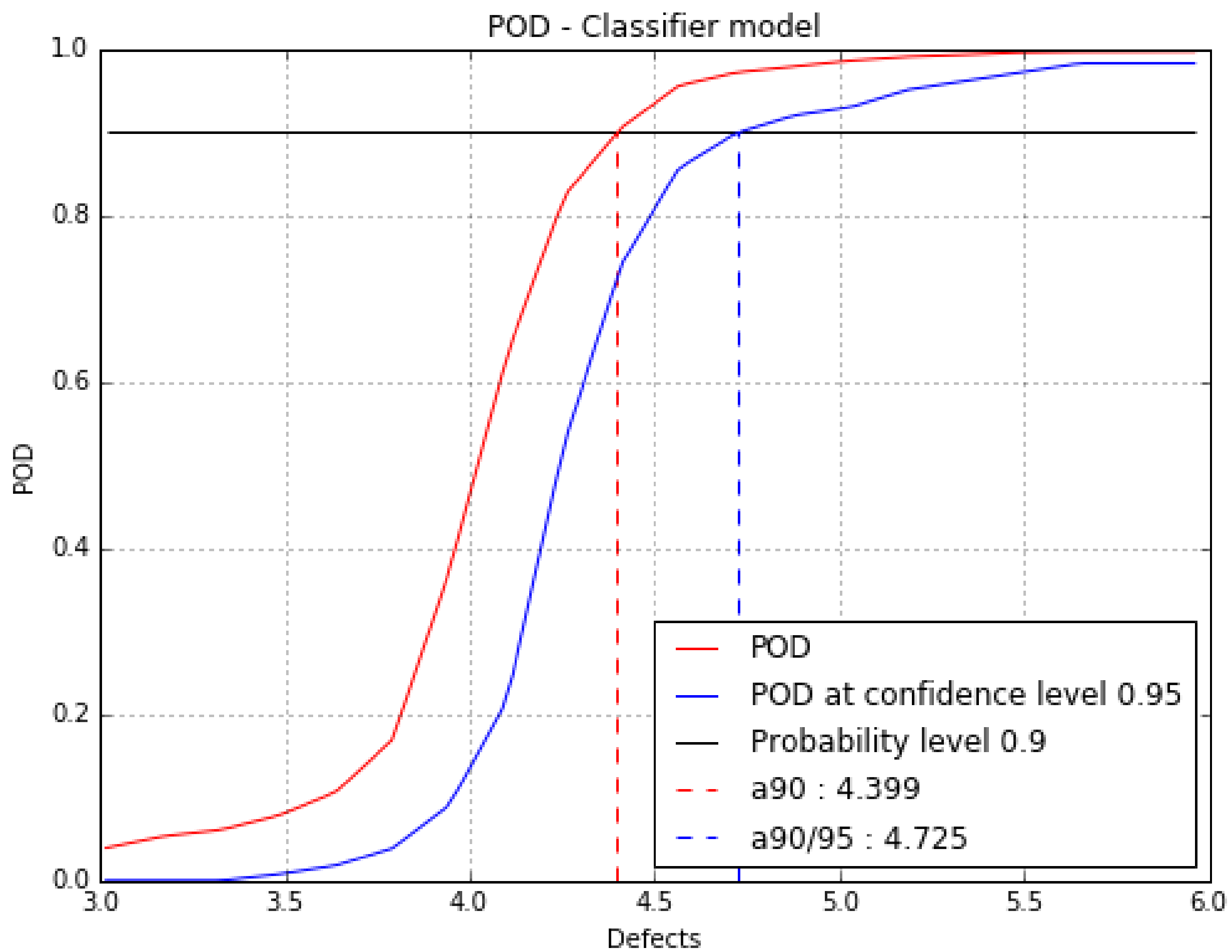

```
Adding points: [================================================--------] 83.33%
```




```
Adding points: [==================================================] 100.00% Done
```




```
fig, ax = hitmiss_algo.drawPOD(0.9, confidenceLevel=0.95)
fig.show()
```



Signal case

Case where the physical model is a function providing the a signal value. In this case, the detection threshold must be given. The hit miss function is built inside the AdaptiveHitMissPOD class and is then used in the algorithm.

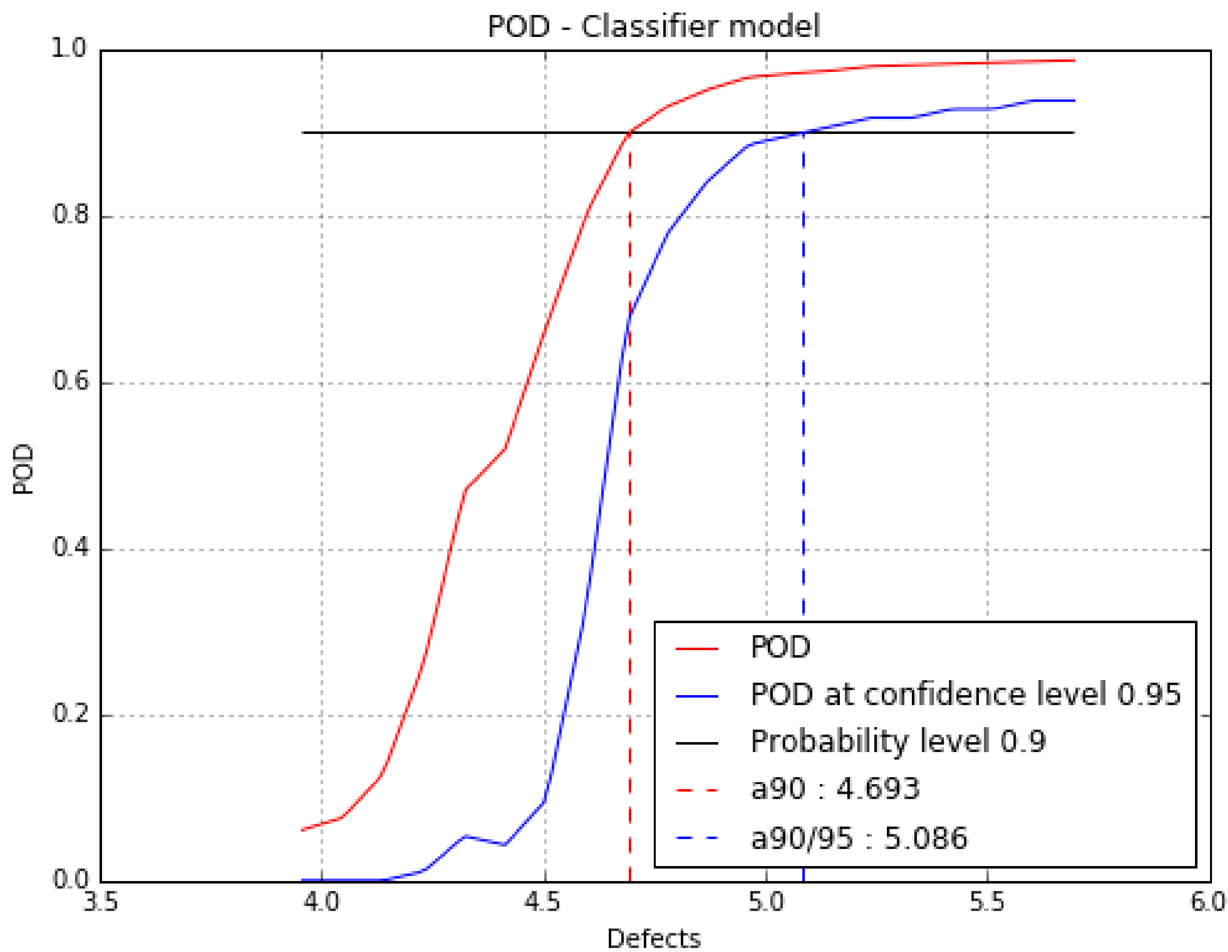
```
inputSample = ot.NumericalSample(
    [[4.59626812e+00, 7.46143339e-02, 1.02231538e+00, 8.60042277e+01],
    [4.14315790e+00, 4.20801346e-02, 1.05874908e+00, 2.65757364e+01],
    [4.76735111e+00, 3.72414824e-02, 1.05730385e+00, 5.76058433e+01],
    [4.82811977e+00, 2.49997658e-02, 1.06954641e+00, 2.54461380e+01],
    [4.48961094e+00, 3.74562922e-02, 1.04943946e+00, 6.19483646e+00],
    [5.05605334e+00, 4.87599783e-02, 1.06520409e+00, 3.39024904e+00],
    [5.69679328e+00, 7.74915877e-02, 1.04099514e+00, 6.50990466e+01],
    [5.10193991e+00, 4.35520544e-02, 1.02502536e+00, 5.51492592e+01],
    [4.04791970e+00, 2.38565932e-02, 1.01906882e+00, 2.07875350e+01],
    [4.66238956e+00, 5.49901237e-02, 1.02427200e+00, 1.45661275e+01],
    [4.86634219e+00, 6.04693570e-02, 1.08199374e+00, 1.05104730e+00],
    [4.13519347e+00, 4.45225831e-02, 1.01900124e+00, 5.10117047e+01],
    [4.92541940e+00, 7.87692335e-02, 9.91868726e-01, 8.32302238e+01],
    [4.70722074e+00, 6.51799251e-02, 1.10608515e+00, 3.30181002e+01],
    [4.29040932e+00, 1.75426222e-02, 9.75678838e-01, 2.28186756e+01],
    [4.89291400e+00, 2.34997929e-02, 1.07669835e+00, 5.38926138e+01],
    [4.44653744e+00, 7.63175936e-02, 1.06979154e+00, 5.19109415e+01],
```

```
    [3.99977452e+00, 5.80430585e-02, 1.01850716e+00, 7.61988190e+01],
    [3.95491570e+00, 1.09302814e-02, 1.03687664e+00, 6.09981789e+01],
    [5.16424368e+00, 2.69026464e-02, 1.06673711e+00, 2.88708887e+01],
    [5.30491620e+00, 4.53802273e-02, 1.06254792e+00, 3.03856837e+01],
    [4.92809155e+00, 1.20616369e-02, 1.00700410e+00, 7.02512744e+00],
    [4.68373805e+00, 6.26028935e-02, 1.05152117e+00, 4.81271603e+01],
    [5.32381954e+00, 4.33013582e-02, 9.90522007e-01, 6.56015973e+01],
    [4.35455857e+00, 1.23814619e-02, 1.01810539e+00, 1.10769534e+01]])

signals = ot.NumericalSample(
    [[ 37.305445], [ 35.466919], [ 43.187991], [ 45.305165], [ 40.121222], [ 44.
↪609524],
     [ 45.14552 ], [ 44.80595 ], [ 35.414039], [ 39.851778], [ 42.046049], [ 34.73469␣
↪],
     [ 39.339349], [ 40.384559], [ 38.718623], [ 46.189709], [ 36.155737], [ 31.
↪768369],
     [ 35.384313], [ 47.914584], [ 46.758537], [ 46.564428], [ 39.698493], [ 45.
↪636588],
     [ 40.643948]])

# detection threshold
detection = 38

# Select point as initial DOE
inputDOE = inputSample[:]
outputDOE = signals[:]

# simulate the true physical model
basis = ot.ConstantBasisFactory(4).build()
covModel = ot.SquaredExponential([5.03148,13.9442,20,20], [15.1697])
krigingModel = ot.KrigingAlgorithm(inputSample, signals, basis, covModel)
krigingModel.run()
physicalModel = krigingModel.getResult().getMetaModel()
```

```
adaptivePOD = otpod.AdaptiveHitMissPOD(inputDOE, outputDOE, physicalModel, 100,␣
↪detection)
adaptivePOD.run()
```

```
Building the classifier
Start the improvement loop
Adding points: [==================================================] 100.00% Done
```

```
fig, ax = adaptivePOD.drawPOD(0.9, confidenceLevel=0.95)
fig.show()
```



ipynb source code

1.2.11 Sobol Indices

It is required to first build a POD object based on the Kriging metamodel or on the polynomial chaos in order to compute the Sobol indices. It also can be used only if the input parameters dimension is greater than 2 (without counting the defect).

```
# import relevant module
import openturns as ot
import otpod
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
inputSample = ot.NumericalSample(
    [[4.59626812e+00, 7.46143339e-02, 1.02231538e+00, 8.60042277e+01],
    [4.14315790e+00, 4.20801346e-02, 1.05874908e+00, 2.65757364e+01],
    [4.76735111e+00, 3.72414824e-02, 1.05730385e+00, 5.76058433e+01],
    [4.82811977e+00, 2.49997658e-02, 1.06954641e+00, 2.54461380e+01],
```

```
    [4.48961094e+00, 3.74562922e-02, 1.04943946e+00, 6.19483646e+00],
    [5.05605334e+00, 4.87599783e-02, 1.06520409e+00, 3.39024904e+00],
    [5.69679328e+00, 7.74915877e-02, 1.04099514e+00, 6.50990466e+01],
    [5.10193991e+00, 4.35520544e-02, 1.02502536e+00, 5.51492592e+01],
    [4.04791970e+00, 2.38565932e-02, 1.01906882e+00, 2.07875350e+01],
    [4.66238956e+00, 5.49901237e-02, 1.02427200e+00, 1.45661275e+01],
    [4.86634219e+00, 6.04693570e-02, 1.08199374e+00, 1.05104730e+00],
    [4.13519347e+00, 4.45225831e-02, 1.01900124e+00, 5.10117047e+01],
    [4.92541940e+00, 7.87692335e-02, 9.91868726e-01, 8.32302238e+01],
    [4.70722074e+00, 6.51799251e-02, 1.10608515e+00, 3.30181002e+01],
    [4.29040932e+00, 1.75426222e-02, 9.75678838e-01, 2.28186756e+01],
    [4.89291400e+00, 2.34997929e-02, 1.07669835e+00, 5.38926138e+01],
    [4.44653744e+00, 7.63175936e-02, 1.06979154e+00, 5.19109415e+01],
    [3.99977452e+00, 5.80430585e-02, 1.01850716e+00, 7.61988190e+01],
    [3.95491570e+00, 1.09302814e-02, 1.03687664e+00, 6.09981789e+01],
    [5.16424368e+00, 2.69026464e-02, 1.06673711e+00, 2.88708887e+01],
    [5.30491620e+00, 4.53802273e-02, 1.06254792e+00, 3.03856837e+01],
    [4.92809155e+00, 1.20616369e-02, 1.00700410e+00, 7.02512744e+00],
    [4.68373805e+00, 6.26028935e-02, 1.05152117e+00, 4.81271603e+01],
    [5.32381954e+00, 4.33013582e-02, 9.90522007e-01, 6.56015973e+01],
    [4.35455857e+00, 1.23814619e-02, 1.01810539e+00, 1.10769534e+01]])

signals = ot.NumericalSample(
    [[ 37.305445], [ 35.466919], [ 43.187991], [ 45.305165], [ 40.121222], [ 44.
↪609524],
     [ 45.14552 ], [ 44.80595 ], [ 35.414039], [ 39.851778], [ 42.046049], [ 34.73469␣
↪],
     [ 39.339349], [ 40.384559], [ 38.718623], [ 46.189709], [ 36.155737], [ 31.
↪768369],
     [ 35.384313], [ 47.914584], [ 46.758537], [ 46.564428], [ 39.698493], [ 45.
↪636588],
     [ 40.643948]])
```

```
# signal detection threshold
detection = 38.
```

Build POD with Kriging model

Running the Kriging based POD

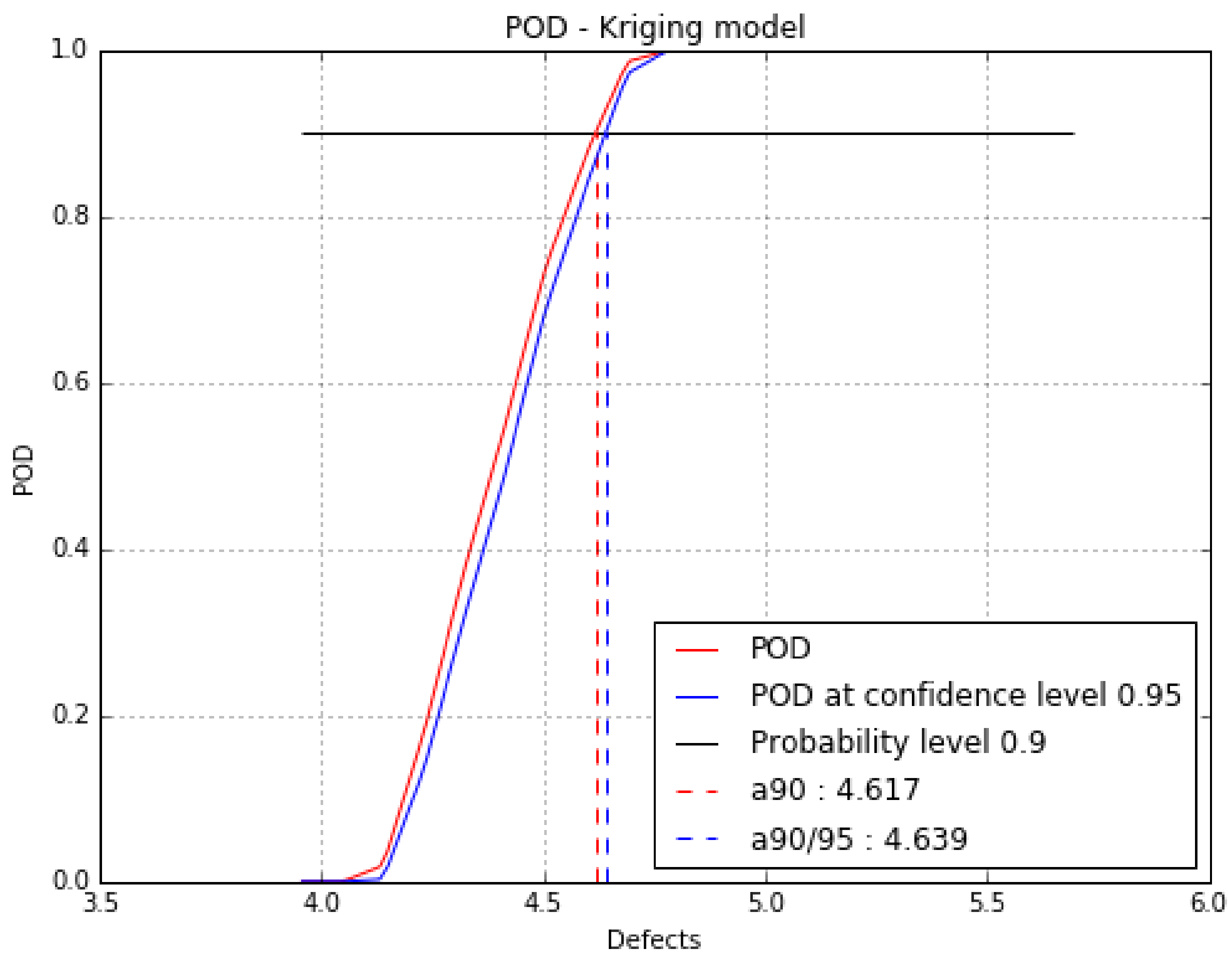
```
krigingPOD = otpod.KrigingPOD(inputSample, signals, detection)

# we can change all simulation size parameters as we are not interested in having an␣
↪accurate POD curve
krigingPOD.setSamplingSize(200)
krigingPOD.setSimulationSize(50)
%time krigingPOD.run()
```

```
Start optimizing covariance model parameters...
Kriging optimizer completed
kriging validation Q2 (>0.9): 0.9999
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 6.17 s, sys: 4.68 s, total: 10.9 s
Wall time: 4.04 s
```

Show POD graphs

```
fig, ax = krigingPOD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODKriging.png')
# The figure is saved in PODPolyChaos.png
fig.show()
```

```
/home/dumas/anaconda2/lib/python2.7/site-packages/matplotlib/figure.py:397:␣
↪UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the␣
↪figure
  "matplotlib is currently using a non-GUI backend, "
```



Build POD with polynomial chaos model

Running the chaos based POD

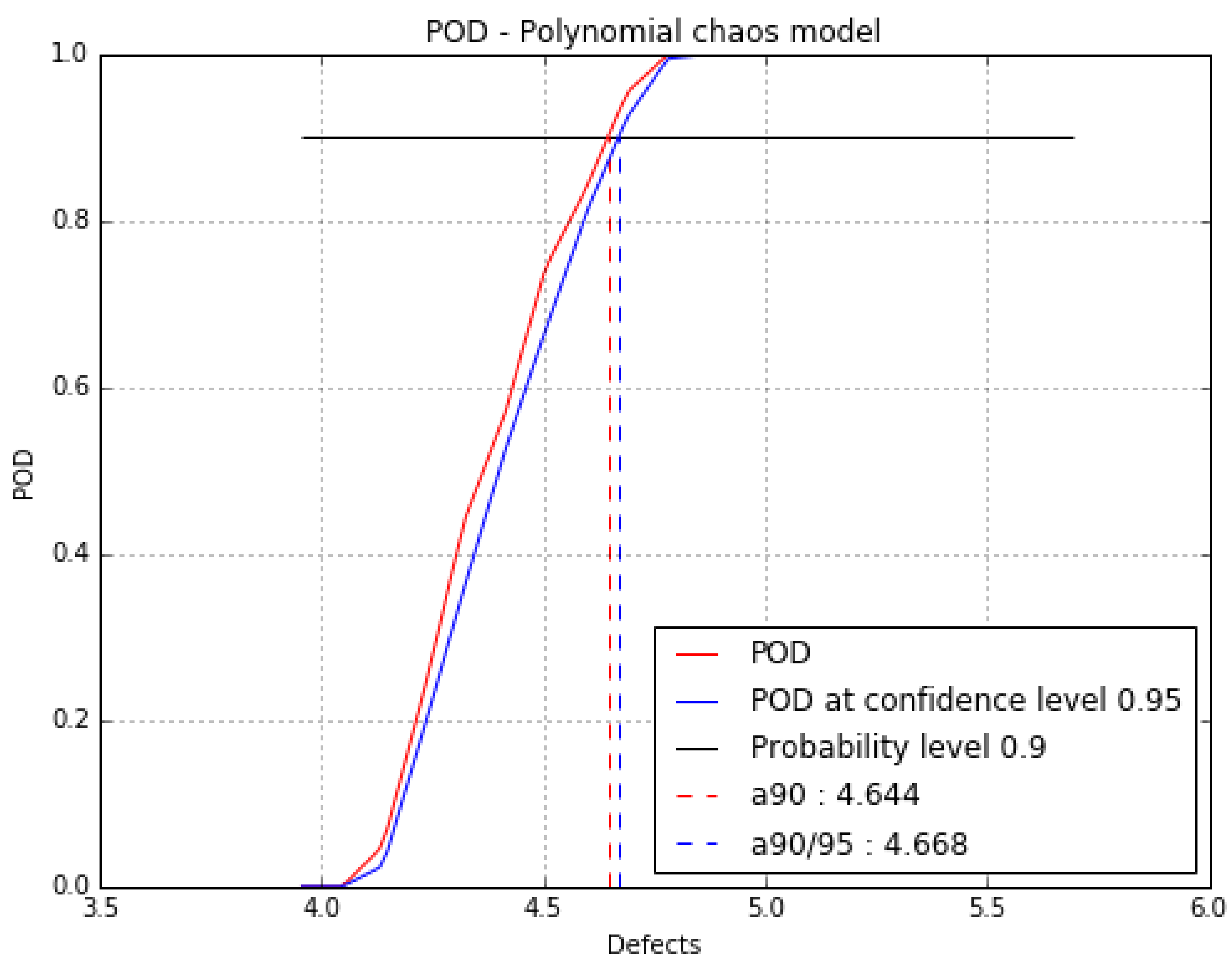
```
chaosPOD = otpod.PolynomialChaosPOD(inputSample, signals, detection)

# we can change all simulation size parameters as we are not interested in having an␣
↪accurate POD curve
chaosPOD.setSamplingSize(200)
```

```
chaosPOD.setSimulationSize(50)
%time chaosPOD.run()
```

```
Start build polynomial chaos model...
Polynomial chaos model completed
Polynomial chaos validation R2 (>0.8) : 0.9999
Polynomial chaos validation Q2 (>0.8) : 0.9987
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 3.46 s, sys: 344 ms, total: 3.8 s
Wall time: 2.19 s
```

Show POD graphs

```
fig, ax = chaosPOD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODChaos.png')
# The figure is saved in PODPolyChaos.png
fig.show()
```



Run the sensitivity analysis

The sensitivity analysis can only be performed with POD computed with a kriging metamodel or a polynomial chaos.

The Sobol indices are aggregated indices computed for the defect sizes defined in the POD study.

Using the kriging model

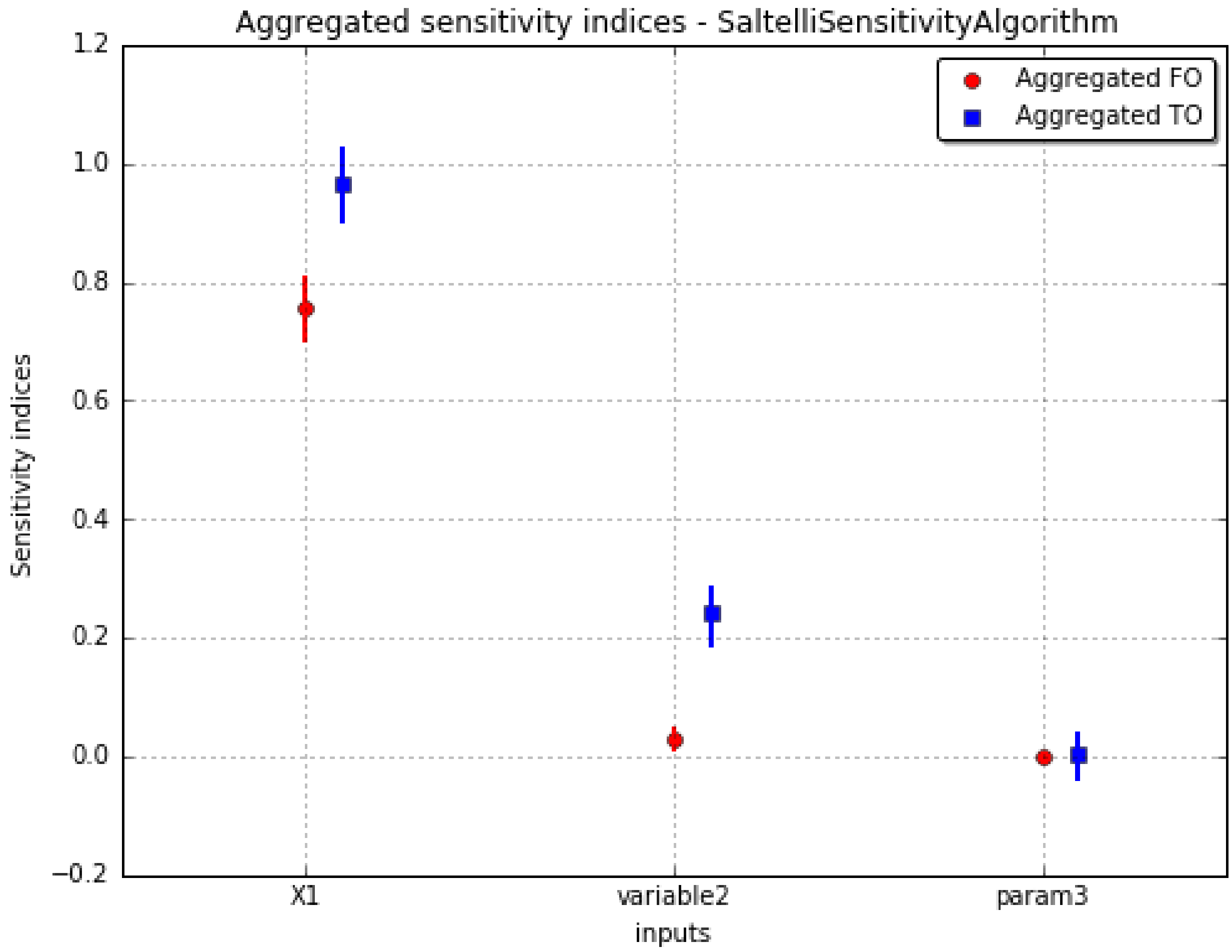
```
# number of simulations
N = 1000
sobol = otpod.SobolIndices(krigingPOD, N)
sobol.run()
```

Draw the figure with given labels

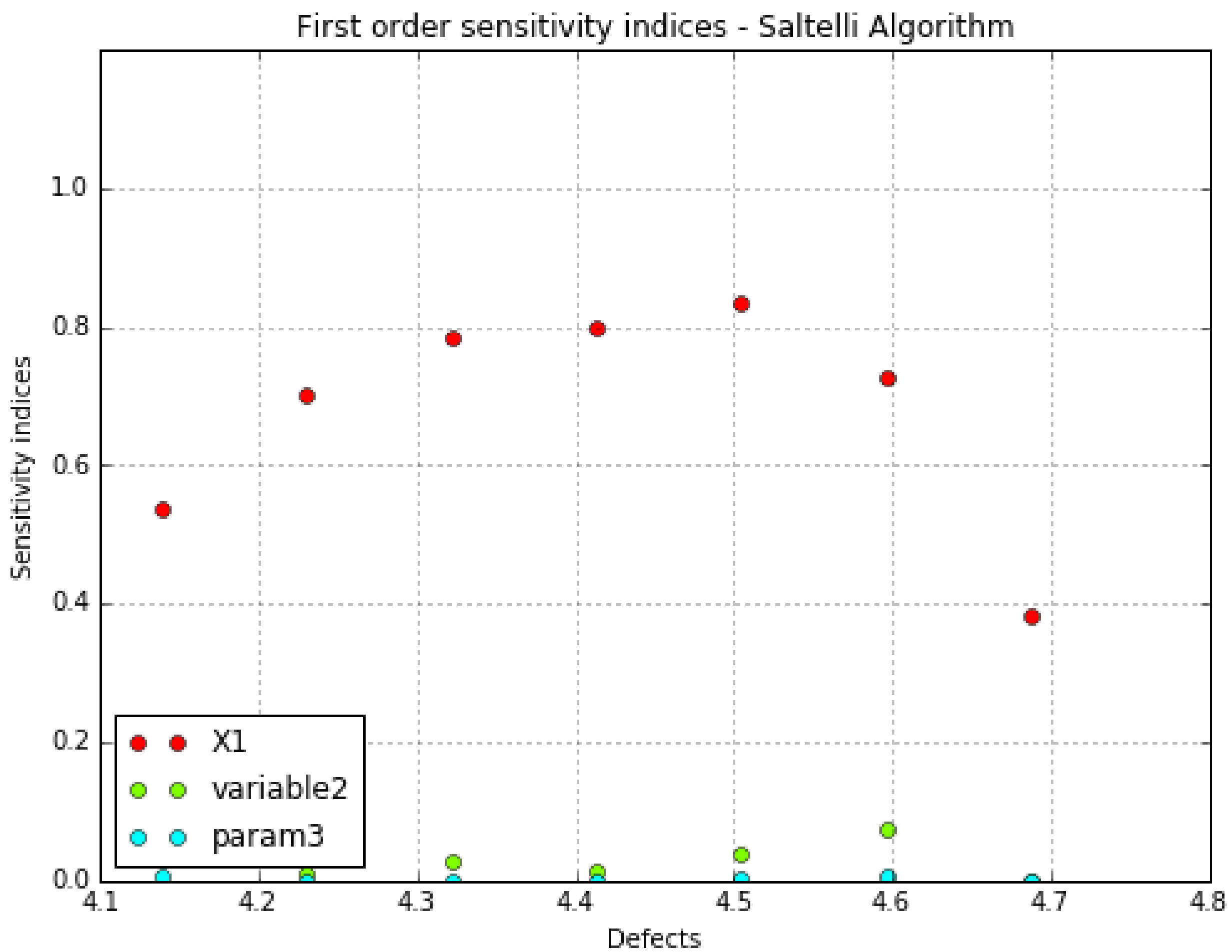
The default labels are Xi but the user can specify its own input labels. Besides, the figure can be saved specifying the attribute *name*.

```
label = ['X1', 'variable2', 'param3']
```

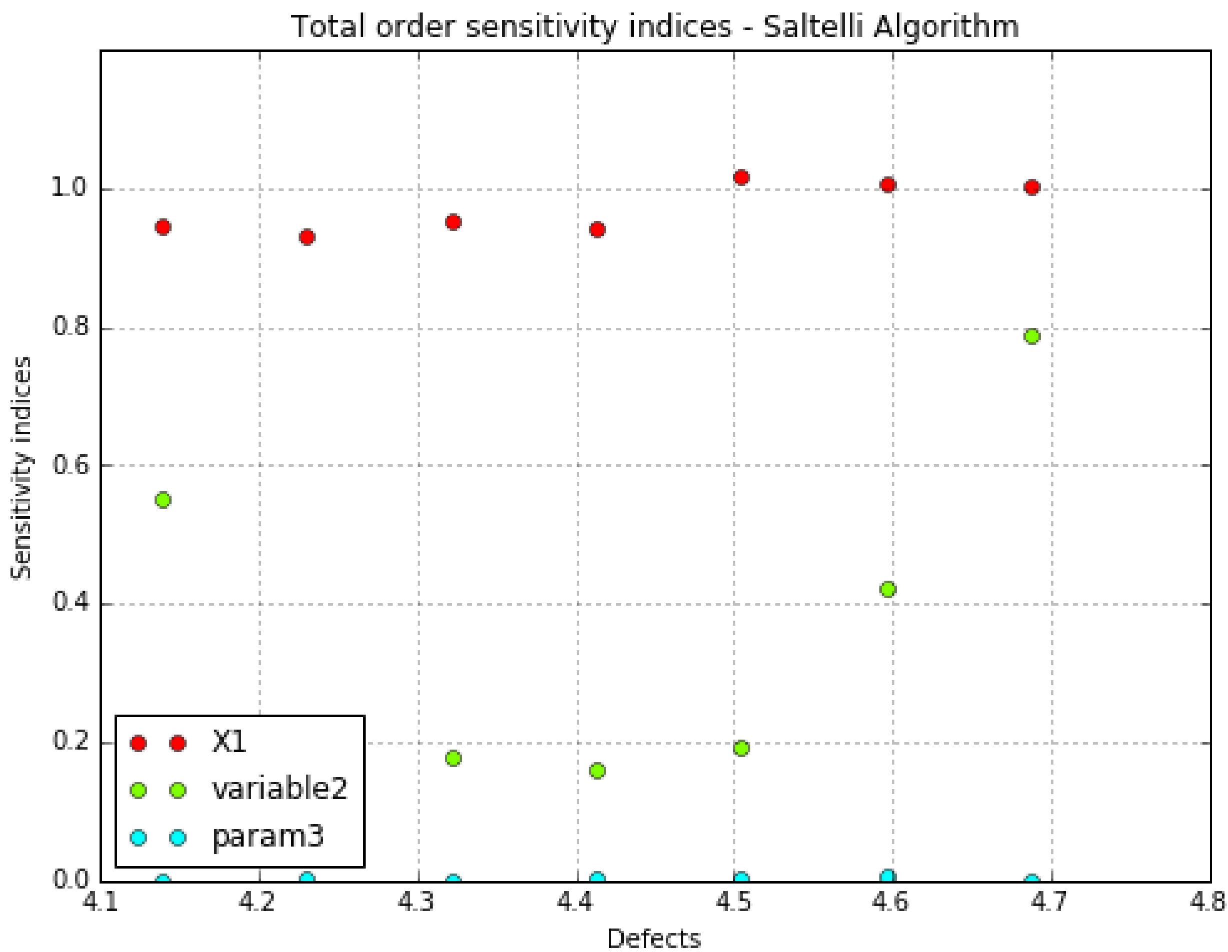
```
fig, ax = sobol.drawAggregatedIndices(label, name='figure/Sobol.png')
fig.show()
```




```
fig, ax = sobol.drawFirstOrderIndices(label, name='figure/FirstOrderSobol.png')
fig.show()
```




```
fig, ax = sobol.drawTotalOrderIndices(label, name='figure/TotalOrderSobol.png')
fig.show()
```



Get the numerical results

The Sobol indices are given in the OpenTURNS object SobolIndicesAlgorithm. The method *getSensitivityResult* allows the get this object and then get back all wanted results.

```
# get the OpenTURNS object
sobol_result = sobol.getSensitivityResult()

# get aggregated indices
print("Aggregated first order: {}".format(sobol_result.
→getAggregatedFirstOrderIndices()))
print("Aggregated total order: {}".format(sobol_result.
→getAggregatedTotalOrderIndices()))

# get the confidence interval
print('\nFirst order confidence interval:')
print(sobol_result.getFirstOrderIndicesInterval())
print('\nTotal order confidence interval:')
print(sobol_result.getTotalOrderIndicesInterval())
```

```
Aggregated first order: [0.757012,0.0304424,0.000851522]
Aggregated total order: [0.96718,0.241168,0.00201511]
```

```
First order confidence interval:
[0.702605, 0.807411]
[0.0141565, 0.0439384]
[-0.00122054, 0.0026483]

Total order confidence interval:
[0.903212, 1.02569]
[0.188041, 0.285366]
[-0.0388075, 0.0380072]
```

It is also possible to retreive the Sobol indices for one defect size among the list.

As example, we want the indices for the 4th defect size in the list. It may **return an error** if the indices cannot be computed because no variability exists. It is the case when the POD is equal to 0 or 1.

```
print("Defect sizes: {}".format(sobol.getDefectSizes()))
# get indices for a specific defect sizes
i = 3 # correspond with the 4th value.
print("\nIndices for defect size {:.3f}: {}".format(sobol.getDefectSizes()[i], sobol_
↪result.getFirstOrderIndices(i)))
```

```
Defect sizes: [ 3.9549157   4.04659347  4.13827123  4.229949    4.32162677  4.41330454
  4.5049823   4.59666007  4.68833784  4.78001561  4.87169337  4.96337114
  5.05504891  5.14672668  5.23840444  5.33008221  5.42175998  5.51343775
  5.60511551  5.69679328]

Indices for defect size 4.230: [0.703006,0.00972298,0.00081138]
```

Change the defect sizes list

It is possible to modify the list of the defect sizes either to reduce the range or to compute the indices for a specific defect value. If only one defect size is provided, then the aggregated indices correspond to the indices.

```
sobol.setDefectSizes([4.5])
sobol.run()
```

```
# get the OpenTURNS object
sobol_result = sobol.getSensitivityResult()
# get aggregated indices
print("Aggregated first order: {}".format(sobol_result.
↪getAggregatedFirstOrderIndices()))
print("Aggregated total order: {}".format(sobol_result.
↪getAggregatedTotalOrderIndices()))
# get indices
print("First order: {}".format(sobol_result.getFirstOrderIndices()))
print("Total order: {}".format(sobol_result.getTotalOrderIndices()))
```

```
Aggregated first order: [0.836394,0.0512459,4.05304e-05]
Aggregated total order: [1.0281,0.170935,0.000238741]
First order: [0.836394,0.0512459,4.05304e-05]
Total order: [1.0281,0.170935,0.000238741]
```

Change the method to compute the indices

OpenTURNS implements 4 methods : Saltelli, Martinez, Jansen and Mauntz-Kucherenko. These methods can be chosen using the method *setSensitivityMethod*.

```
sobol.setSensitivityMethod("Martinez")
sobol.run()
```

```
# get the OpenTURNS object
sobol_result = sobol.getSensitivityResult()
# get aggregated indices
print("Aggregated first order: {}".format(sobol_result.
↪getAggregatedFirstOrderIndices()))
print("Aggregated total order: {}".format(sobol_result.
↪getAggregatedTotalOrderIndices()))
```

```
Aggregated first order: [0.844923,0.0285396,-0.00563925]
Aggregated total order: [0.98182,0.164579,0.00129423]
```

Case with polynomial chaos

With polynomial chaos, the POD is computed simulating several polynomial chaos coefficients. Then it requires more times than with Kriging. The number of simulations is initially set to 1000 but it can be changed using the method *setSimulationSize*.

```
# number of simulations
N = 1000
sobol2 = otpod.SobolIndices(chaosPOD, N)
#sobol2.setSimulationSize(500)
%time sobol2.run()
```

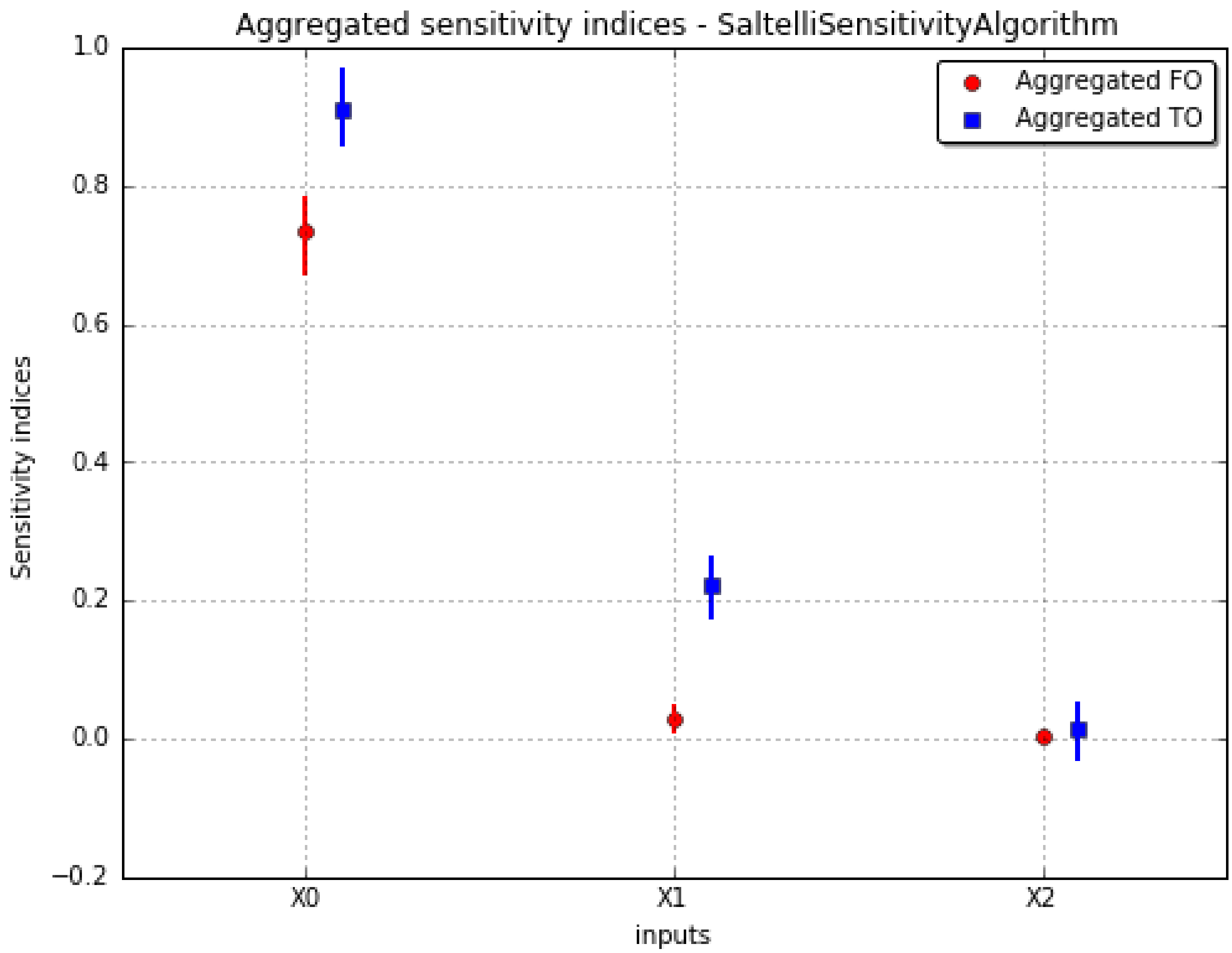
```
CPU times: user 25min 56s, sys: 27.1 s, total: 26min 23s
Wall time: 14min 3s
```

```
# get the OpenTURNS object
sobol_result2 = sobol2.getSensitivityResult()

# get aggregated indices
print("Aggregated first order: {}".format(sobol_result2.
↪getAggregatedFirstOrderIndices()))
print("Aggregated total order: {}".format(sobol_result2.
↪getAggregatedTotalOrderIndices()))

# get the confidence interval
print('\nFirst order confidence interval:')
print(sobol_result2.getFirstOrderIndicesInterval())
print('\nTotal order confidence interval:')
print(sobol_result2.getTotalOrderIndicesInterval())
```

```
Aggregated first order: [0.733816,0.0303936,0.00202081]
Aggregated total order: [0.912059,0.221395,0.0131351]

First order confidence interval:
[0.674632, 0.779876]
```

```
[0.011269, 0.0475021]
[-0.00224056, 0.00590774]

Total order confidence interval:
[0.860978, 0.966444]
[0.176179, 0.263411]
[-0.0289697, 0.0499621]
```

```
fig, ax = sobol2.drawAggregatedIndices()
fig.show()
```



ipynb source code

1.2.12 Perturbation Law Indices

It is required to first build a POD object based on the Kriging metamodel or on the polynomial chaos in order to compute the PLI. It also can be used only if the input parameters dimension is greater than 2 (without counting the defect).

```
# import relevant module
import openturns as ot
import otpod
```

```
import numpy as np
# enable display figure in notebook
%matplotlib inline
```

Generate data

```
inputSample = ot.NumericalSample(
    [[4.59626812e+00, 7.46143339e-02, 1.02231538e+00, 8.60042277e+01],
    [4.14315790e+00, 4.20801346e-02, 1.05874908e+00, 2.65757364e+01],
    [4.76735111e+00, 3.72414824e-02, 1.05730385e+00, 5.76058433e+01],
    [4.82811977e+00, 2.49997658e-02, 1.06954641e+00, 2.54461380e+01],
    [4.48961094e+00, 3.74562922e-02, 1.04943946e+00, 6.19483646e+00],
    [5.05605334e+00, 4.87599783e-02, 1.06520409e+00, 3.39024904e+00],
    [5.69679328e+00, 7.74915877e-02, 1.04099514e+00, 6.50990466e+01],
    [5.10193991e+00, 4.35520544e-02, 1.02502536e+00, 5.51492592e+01],
    [4.04791970e+00, 2.38565932e-02, 1.01906882e+00, 2.07875350e+01],
    [4.66238956e+00, 5.49901237e-02, 1.02427200e+00, 1.45661275e+01],
    [4.86634219e+00, 6.04693570e-02, 1.08199374e+00, 1.05104730e+00],
    [4.13519347e+00, 4.45225831e-02, 1.01900124e+00, 5.10117047e+01],
    [4.92541940e+00, 7.87692335e-02, 9.91868726e-01, 8.32302238e+01],
    [4.70722074e+00, 6.51799251e-02, 1.10608515e+00, 3.30181002e+01],
    [4.29040932e+00, 1.75426222e-02, 9.75678838e-01, 2.28186756e+01],
    [4.89291400e+00, 2.34997929e-02, 1.07669835e+00, 5.38926138e+01],
    [4.44653744e+00, 7.63175936e-02, 1.06979154e+00, 5.19109415e+01],
    [3.99977452e+00, 5.80430585e-02, 1.01850716e+00, 7.61988190e+01],
    [3.95491570e+00, 1.09302814e-02, 1.03687664e+00, 6.09981789e+01],
    [5.16424368e+00, 2.69026464e-02, 1.06673711e+00, 2.88708887e+01],
    [5.30491620e+00, 4.53802273e-02, 1.06254792e+00, 3.03856837e+01],
    [4.92809155e+00, 1.20616369e-02, 1.00700410e+00, 7.02512744e+00],
    [4.68373805e+00, 6.26028935e-02, 1.05152117e+00, 4.81271603e+01],
    [5.32381954e+00, 4.33013582e-02, 9.90522007e-01, 6.56015973e+01],
    [4.35455857e+00, 1.23814619e-02, 1.01810539e+00, 1.10769534e+01]])

signals = ot.NumericalSample(
    [[ 37.305445], [ 35.466919], [ 43.187991], [ 45.305165], [ 40.121222], [ 44.
↪609524],
     [ 45.14552 ], [ 44.80595 ], [ 35.414039], [ 39.851778], [ 42.046049], [ 34.73469␣
↪],
     [ 39.339349], [ 40.384559], [ 38.718623], [ 46.189709], [ 36.155737], [ 31.
↪768369],
     [ 35.384313], [ 47.914584], [ 46.758537], [ 46.564428], [ 39.698493], [ 45.
↪636588],
     [ 40.643948]])
```

```
# signal detection threshold
detection = 38.
```

Build POD with Kriging model

Running the Kriging based POD

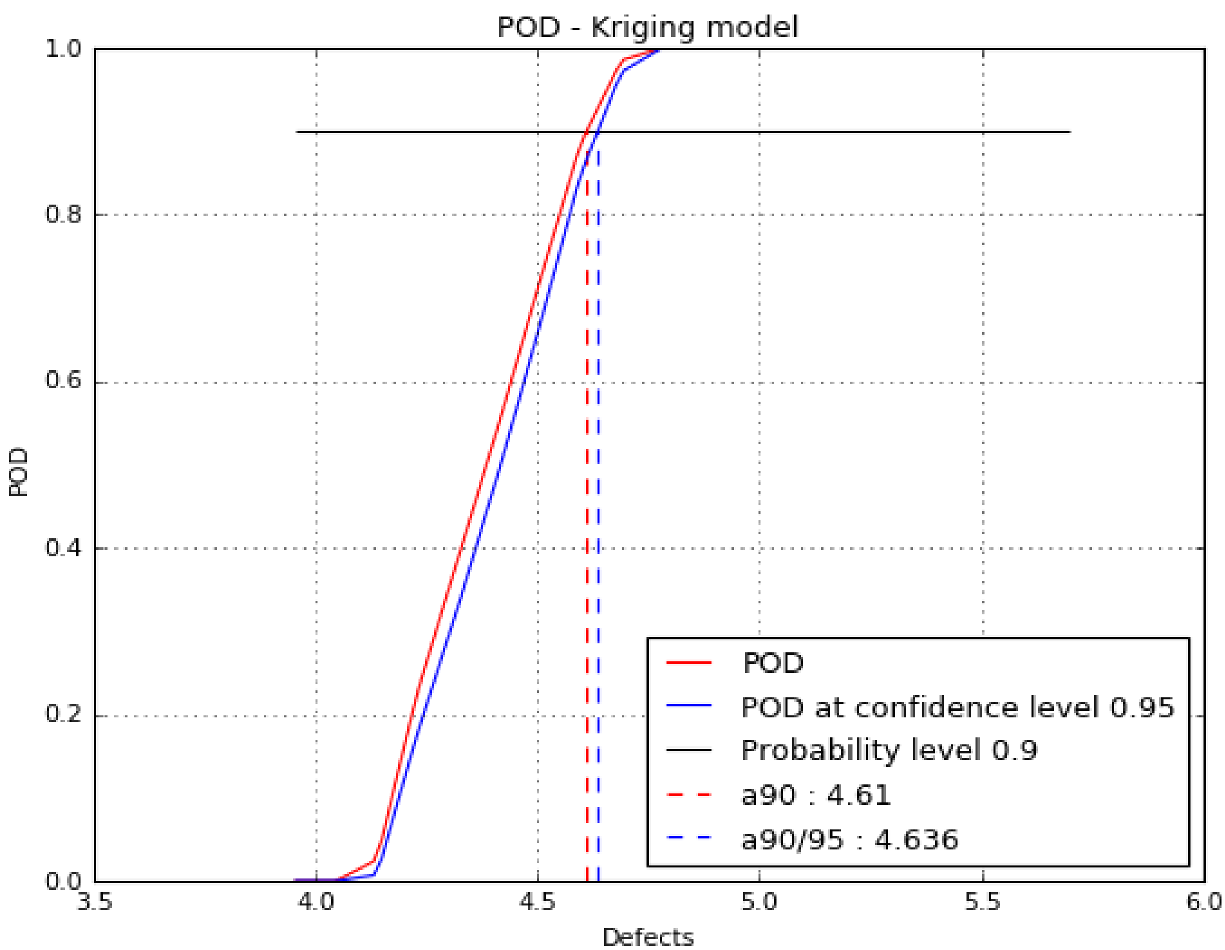
```
krigingPOD = otpod.KrigingPOD(inputSample, signals, detection)
```

```
# we can change all simulation size parameters as we are not interested in having an␣
↪accurate POD curve
krigingPOD.setSamplingSize(200)
krigingPOD.setSimulationSize(50)
%time krigingPOD.run()
```

```
Start optimizing covariance model parameters...
Kriging optimizer completed
kriging validation Q2 (>0.9): 1.0000
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 6.14 s, sys: 4.28 s, total: 10.4 s
Wall time: 3 s
```

Show POD graphs

```
fig, ax = krigingPOD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                        name='figure/PODKriging.png')
# The figure is saved in PODPolyChaos.png
fig.show()
```



Build POD with polynomial chaos model

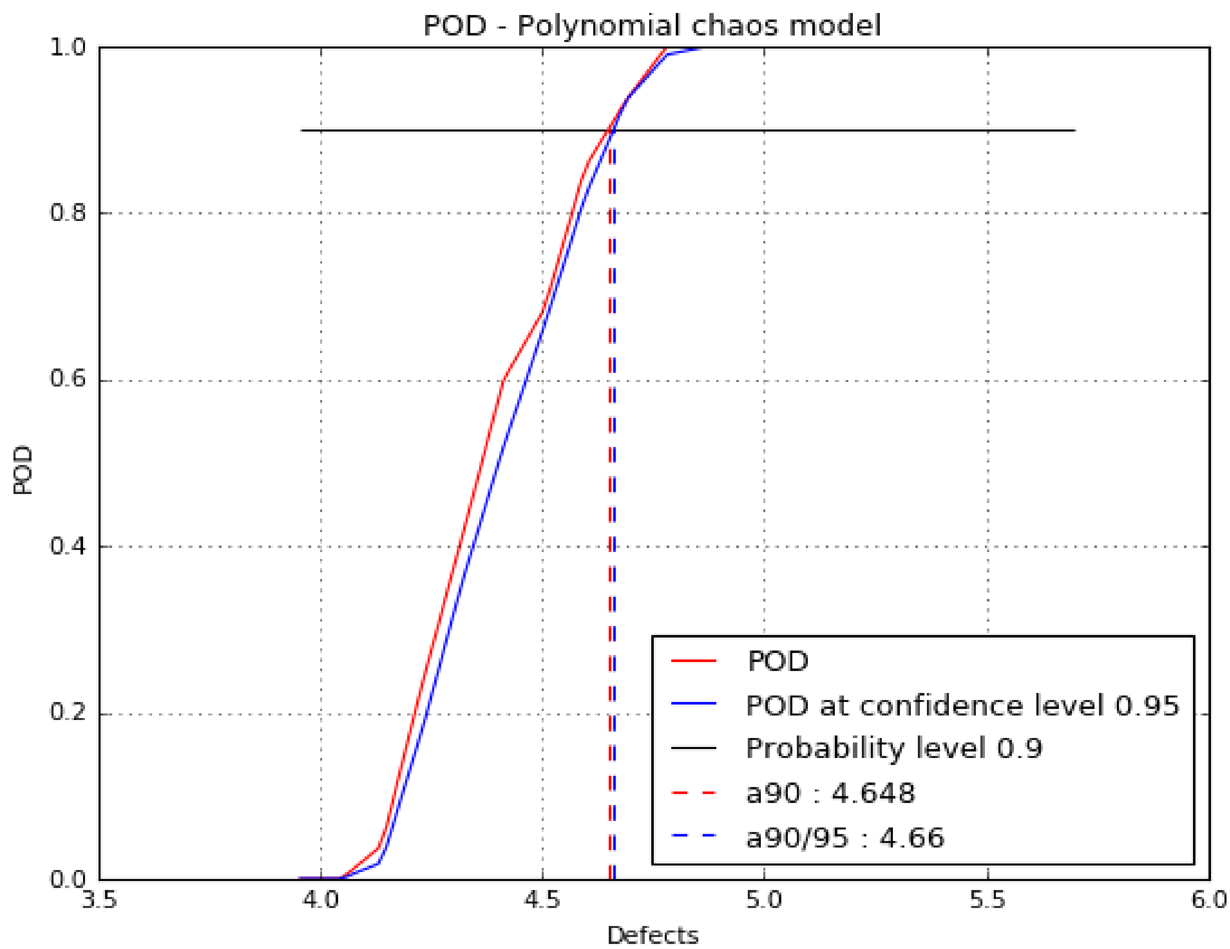
Running the chaos based POD

```
chaosPOD = otpod.PolynomialChaosPOD(inputSample, signals, detection)

# we can change all simulation size parameters as we are not interested in having an␣
↪accurate POD curve
chaosPOD.setSamplingSize(200)
chaosPOD.setSimulationSize(50)
%time chaosPOD.run()
```

```
Start build polynomial chaos model...
Polynomial chaos model completed
Polynomial chaos validation R2 (>0.8) : 0.9999
Polynomial chaos validation Q2 (>0.8) : 0.9987
Computing POD per defect: [==================================================] 100.00
↪% Done
CPU times: user 3.53 s, sys: 596 ms, total: 4.12 s
Wall time: 1.93 s
```

Show POD graphs

```
fig, ax = chaosPOD.drawPOD(probabilityLevel=0.9, confidenceLevel=0.95,
                      name='figure/PODChaos.png')
# The figure is saved in PODPolyChaos.png
fig.show()
```



Run the sensitivity analysis

The sensitivity analysis can only be performed with POD computed with a kriging metamodel or a polynomial chaos.

The PLI are computed for the defect sizes defined in the POD study. However, if the probability estimate for a defect size is less than 1e-3 or greater than 0.999, then the indices are not computed.

The PLI can be computed either with perturbed mean or a perturbed variance. Two dedicated classes exists for each case.

Mean perturbation

For the mean perturbation, it is possible to change the type of the mean shifting. If *sigmaScaled* = *False*, the given delta values are the new marginal means. If *sigmaScaled* = *True*, then $newMean = mean + sigma \times delta$, where sigma is the standard deviation of each marginals.

It is adviced to set the *sigmaScaled* parameter to True when the input distribution are not equal.

with a Kriging POD

```
# the delta values
deltas = np.linspace(-1, 1, 6)
# sigma scaled is activated because the input distributions are not the same
pliMean = otpod.PLIMean(krigingPOD, deltas, sigmaScaled=True)
pliMean.run()
```

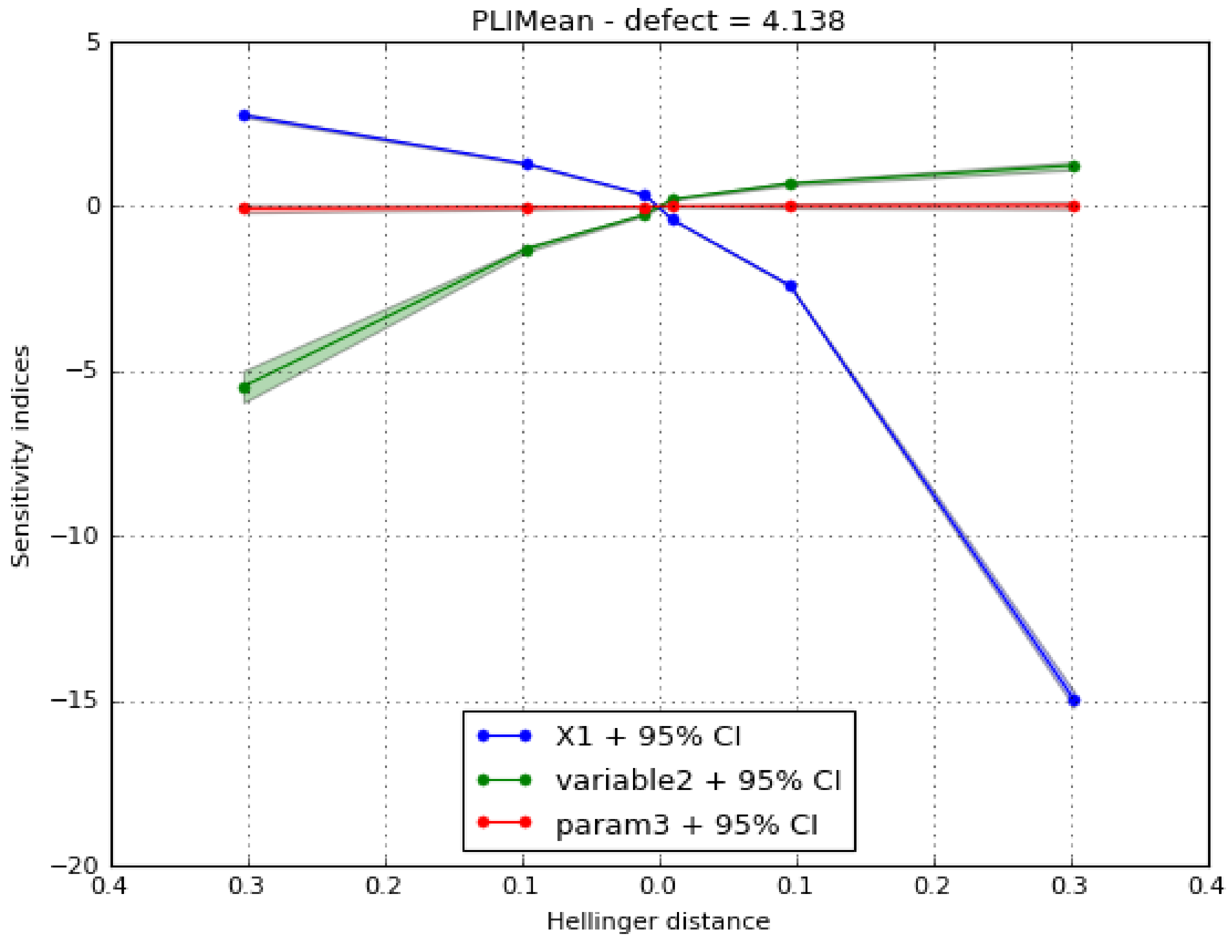
Draw the figure with given labels

The indices can be displayed either for a specific defect size. In this case, indices for all margins are plotted in the same figure. You can choose to display the indices with respect to the Hellinger distance (default case) in order to compare in the same scale the indices.

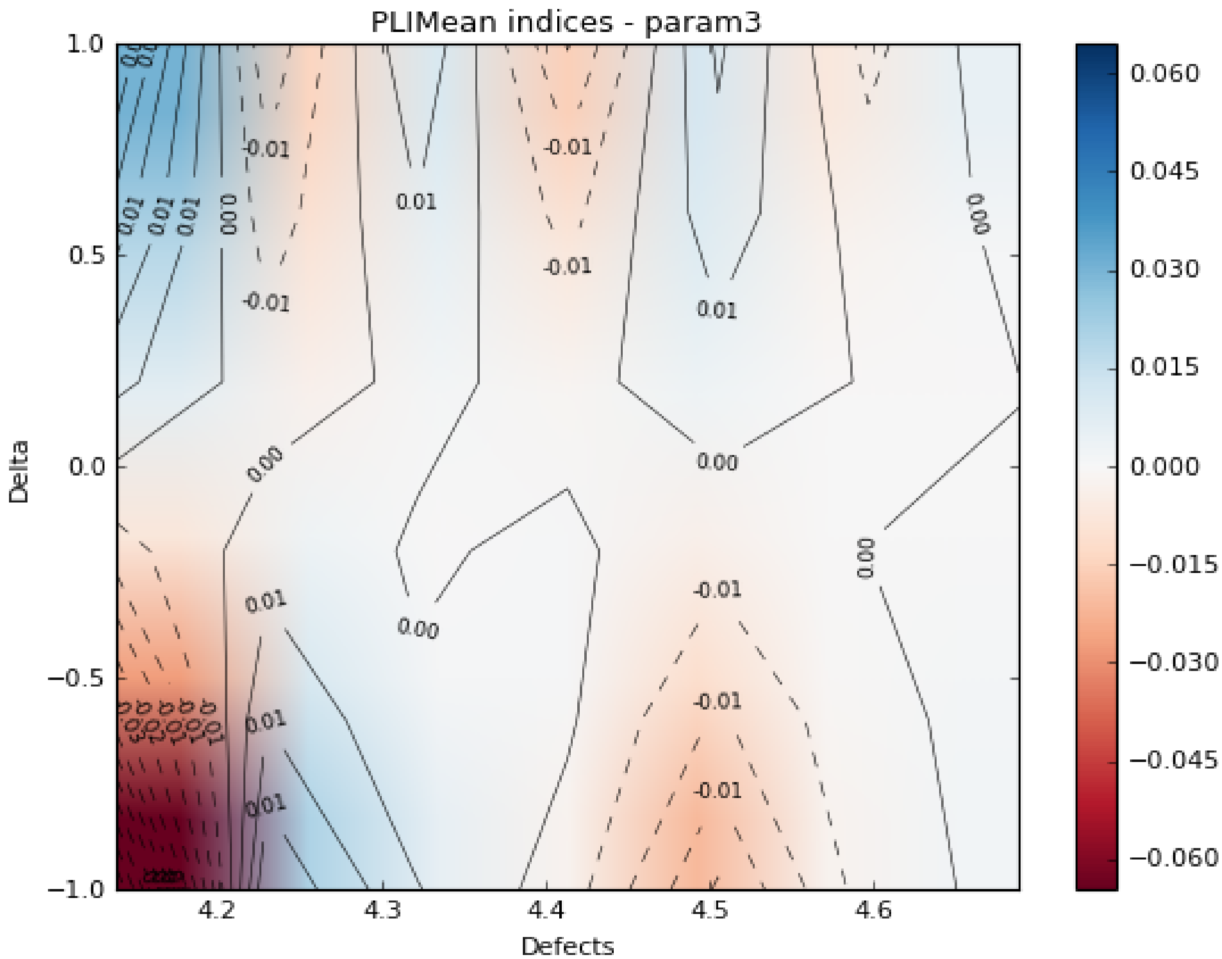
The second graph is a 2d contour plot where the indices for a given margin are plotted with respect to the defect size and the delta values. It enables you to compare the indices depending of the defect size.

```
label = ['X1', 'variable2', 'param3']
```

```
idefect = 2
fig, ax = pliMean.drawIndices(idefect, confidenceLevel=0.95,
                              label=label, hellinger=True)
ax.legend(loc='lower center')
fig.show()
```




```
marginal = 2
fig, ax = pliMean.drawContourIndices(marginal, label[marginal])
fig.show()
```



Get the numerical results

The PLI values can be obtained thanks to the method *getIndices*. You can to get: - all indices values - indices for a specific marginal - indices for a specific delta value - indices for a specific defect size - a combination of above values

The nan values corresponds to the defect sizes for which the indices cannot be computed because the probability estimate is too small or too large.

```
print('Indices for marginal 1: ')
print(pliMean.getIndices(marginal=1))
```

```
Indices for marginal 1:
[[         nan          nan -5.47183238 -0.43770215 -0.22644964 -0.14562948
  -0.09611319 -0.09935447 -0.05746253          nan          nan          nan
          nan          nan          nan          nan          nan          nan
          nan          nan]
 [         nan          nan -1.28662173 -0.22180929 -0.12398571 -0.08555665
  -0.05262299 -0.05717166 -0.03074273          nan          nan          nan
          nan          nan          nan          nan          nan          nan
          nan          nan]
 [         nan          nan -0.25378939 -0.06402069 -0.03767213 -0.02784247
  -0.01640227 -0.01827718 -0.00922909          nan          nan          nan
```

```
          nan            nan            nan            nan            nan            nan
          nan            nan]
 [        nan            nan     0.21614647     0.05992336     0.03567235     0.02796824
   0.01577573     0.01789474     0.0083912             nan            nan            nan
          nan            nan            nan            nan            nan            nan
          nan            nan]
 [        nan            nan     0.6913383      0.17951681     0.10446151     0.08710083
   0.04654376     0.05366883     0.02283794            nan            nan            nan
          nan            nan            nan            nan            nan            nan
          nan            nan]
 [        nan            nan     1.23892676     0.30078311     0.16751247     0.1528117
   0.07658196     0.09002552     0.0336249             nan            nan            nan
          nan            nan            nan            nan            nan            nan
          nan            nan]]
```

```
print('Indices for defect 4: ')
print(pliMean.getIndices(idefect=4))
```

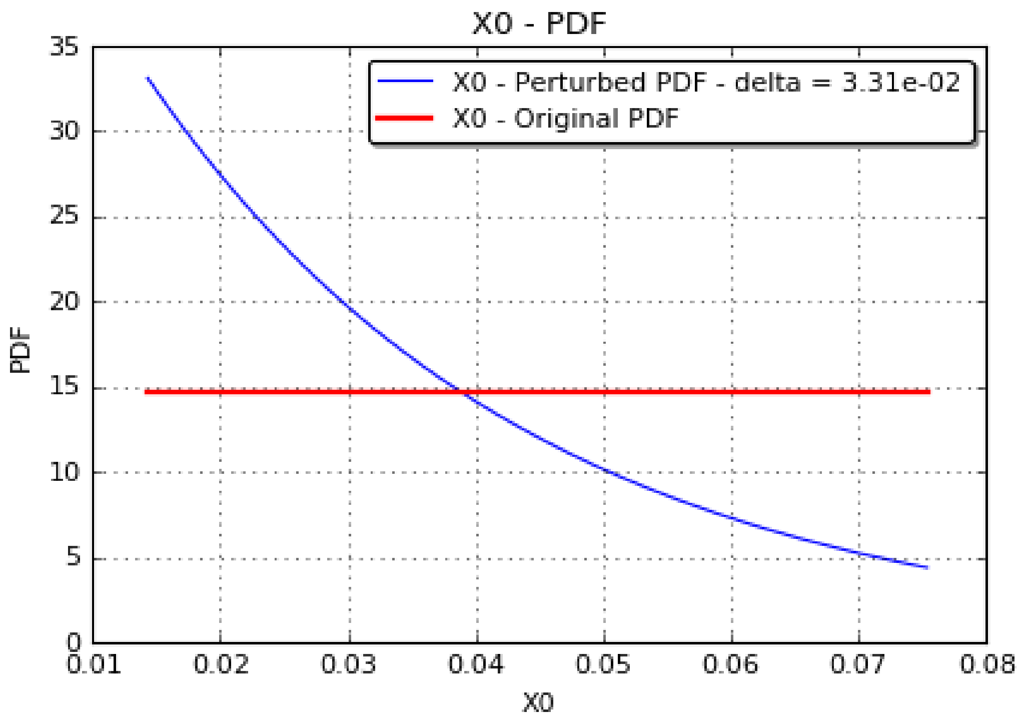
```
Indices for defect 4:
[[  1.09347293e+00  -2.26449643e-01   5.25736353e-03]
 [  6.48300677e-01  -1.23985710e-01   5.46332364e-04]
 [  2.09390155e-01  -3.76721289e-02  -4.81674899e-04]
 [ -2.50243943e-01   3.56723519e-02   1.04956196e-03]
 [ -1.29198218e+00   1.04461511e-01   4.67552207e-03]
 [ -5.74859136e+00   1.67512468e-01   9.83388899e-03]]
```

PLI object for a specific defect size

From the base PLI object computed for each defect size, you can have access to more results : - the confidence interval - the perturbed probability estimate - draw the perturbed marginal density

```
# get PLI object for the 3rd defect
pliMeanDefect3 = pliMean.getPLIObject(3)

print("Perturbed probability estimate : ")
print pliMeanDefect3.getPerturbedProbabilityEstimate()

fig, ax = pliMeanDefect3.drawMarginal1DPDF(marginal=0, idelta=1)
fig.show()
```

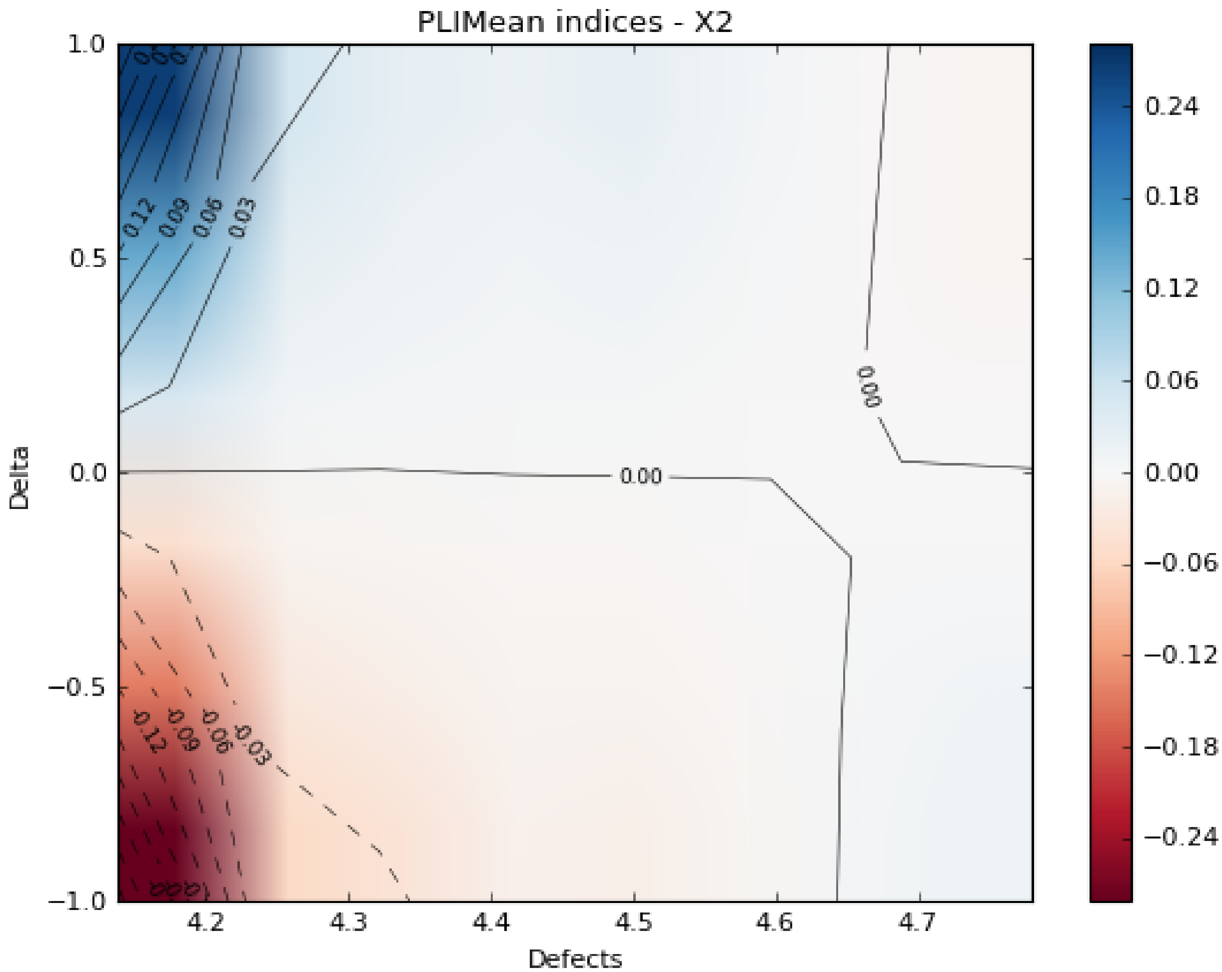
```
Perturbed probability estimate :
[[ 0.58921468  0.14307553  0.2097866 ]
 [ 0.40523398  0.16835688  0.20773887]
 [ 0.26424247  0.19332331  0.20629464]
 [ 0.15418711  0.21802623  0.20515709]
 [ 0.07163711  0.24262661  0.20412897]
 [ 0.01853949  0.26757109  0.20288179]]
```



with polynomial chaos

```
# the delta values
deltas = np.linspace(-1, 1, 6)
# sigma scaled is activated because the input distributions are not the same
pliMean = otpod.PLIMean(chaosPOD, deltas, sigmaScaled=True)
pliMean.run()
```

```
marginal = 2
fig, ax = pliMean.drawContourIndices(marginal)
fig.show()
```



Variance perturbation

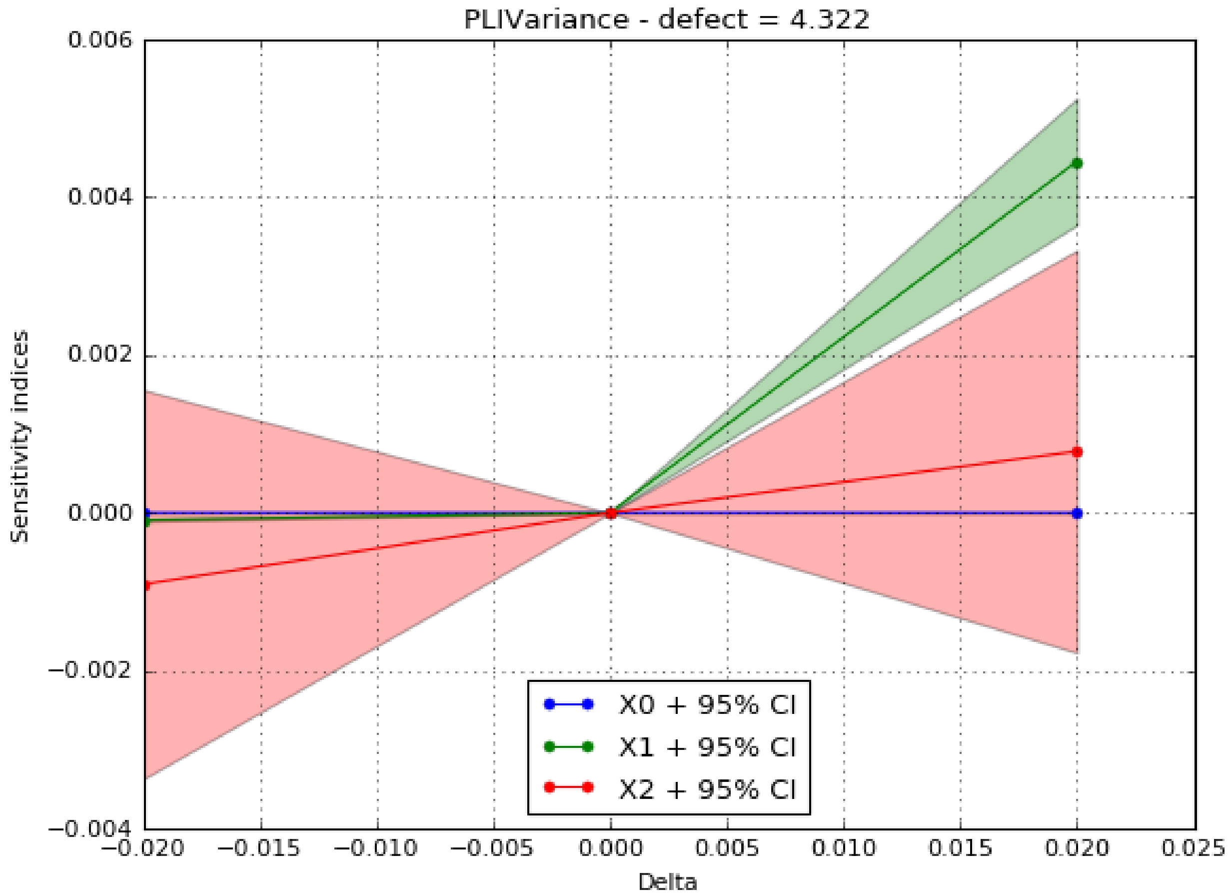
For the variance perturbation, the delta values must be greater than 0. The delta values corresponds to: - the new variance if *coefScaled* = *False* - $newCov = delta + cov$ if *covScaled* = *True*, this increases the coefficient of variation by delta. The new variance is computed such that the mean does not change.

It is also possible to define the delta values independently for each margin.

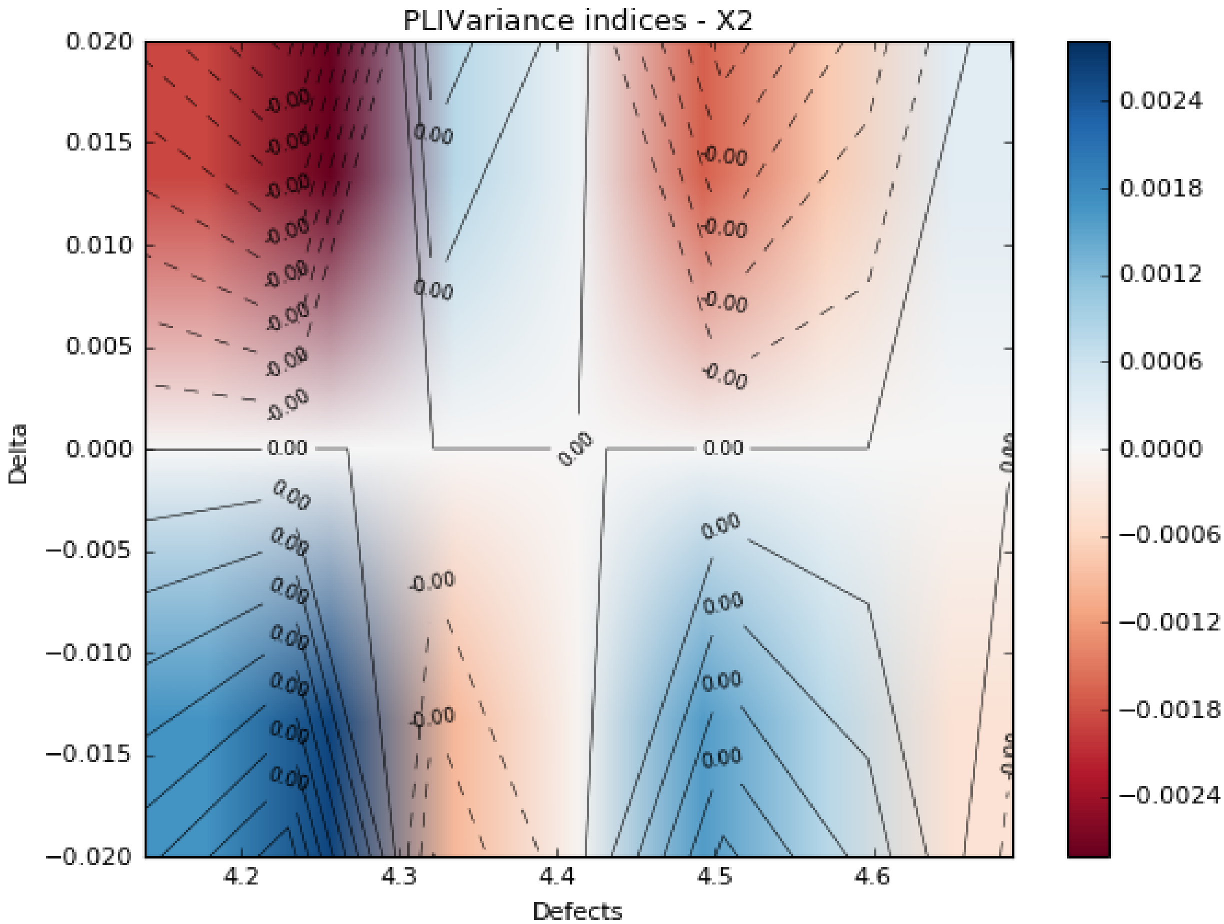
```
# the delta values
deltas = np.linspace(-0.02, 0.02, 3)

# with coef scaled
pliVar = otpod.PLIVariance(krigingPOD, deltas, covScaled=True)
pliVar.run()
```

With the Hellinger distance, we can see that the perturbed variance is not equivalent for all margins.

```
idefect = 4
fig, ax = pliVar.drawIndices(idefect, confidenceLevel=0.95,
                             hellinger=False)
ax.legend(loc='lower center')
fig.show()
```




```
marginal = 2
fig, ax = pliVar.drawContourIndices(marginal)
fig.show()
```


PLIVariance indices - X2
Delta
Defects
0.020
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
4.2
4.3
4.4
4.5
4.6
0.0024
0.0018
0.0012
0.0006
0.0000
−0.0006
−0.0012
−0.0018
−0.0024

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

A

C

D

F

G

K

P

Q

R

S

U