Overview of OpenTURNS and its graphical user interface

J. Pelamatti [1], M. Baudin [1], T. Delage [1]

[1]EDF R&D. 6, quai Watier, 78401, Chatou Cedex - France, julien.pelamatti@edf.fr

Journées du réseau MEXICO, 29-30 Novembre 2021, INRAE

Contents

Introduction

A few OpenTURNS functionalities and Examples

PERSALYS, the graphical user interface

OpenTURNS: `www.openturns.org`

OpenTURNS

An Open source initiative for the Treatment of Uncertainties, Risks'N Statistics

- C++ library with a Python interface
- Numerical tools dedicated to the treatment of uncertainties
- Generic coupling to any type of physical model
- Open source, LGPL licensed

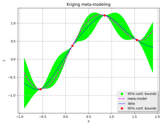
Baudin, Michaël, et al. *Open TURNS: An industrial software for uncertainty quantification in simulation.*

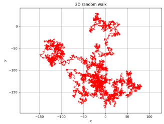
OpenTURNS: `www.openturns.org`

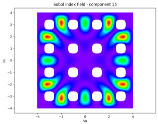
- Linux, Windows
- First release : 2007
- 5 full time developers
- 370 000 Total Conda downloads
- Project size (2018) : 720 classes, more than 6000 services

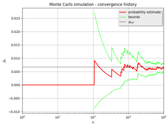
OpenTURNS: content

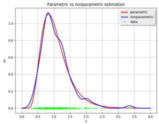
- Data analysis
  - Distribution fitting
  - Statistical tests
  - Estimate dependency and copulas
  - Estimate stochastic processes
- Probabilistic modeling
  - Dependence modeling
  - Univariate distributions
  - Multivariate distrbutions
  - Copulas
  - Processes
  - Covariance models
- Meta modeling
  - Linear regression
  - Polynomial chaos expansion
  - Gaussian process regression
  - Spectral methods
  - Low rank tensors
  - Fields metamodel
- Reliability, sensitivity
  - Sampling methods
  - Approximation methods
  - Sensitivity analysis
  - Design of experiments
- Calibration
  - Least squares calibration
  - Gaussian calibration
  - Bayesian calibration
- Numerical methods
  - Optimization
  - Integration
  - Least squares
  - Meshing
  - Coupling with external codes



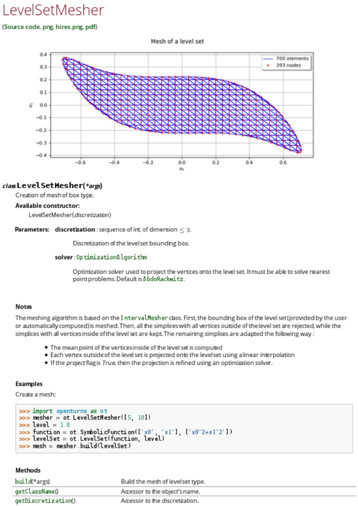








OpenTURNS: documentation



LevelSetMesher

(Source code, png, hires.png, pdf)

class **`LevelSetMesher`**(**args*)

Creation of mesh of box type.

Available constructor:

LevelSetMesher(*discretization*)

Parameters: **discretization** : sequence of int, of dimension ≤ 3.

Discretization of the level set bounding box.

solver : `OptimizationAlgorithm`

Optimization solver used to project the vertices onto the level set. It must be able to solve nearest point problems. Default is `AbdoRackwitz`.

Notes

The meshing algorithm is based on the `IntervalMesher` class. First, the bounding box of the level set (provided by the user or automatically computed) is meshed. Then, all the simplices with all vertices outside of the level set are rejected, while the simplices with all vertices inside of the level set are kept. The remaining simplices are adapted the following way :

- The mean point of the vertices inside of the level set is computed
- Each vertex outside of the level set is projected onto the level set using a linear interpolation
- If the *project* flag is *True*, then the projection is refined using an optimization solver.

Examples

Create a mesh:

```
>>> import openturns as ot
>>> mesher = ot.LevelSetMesher([5, 10])
>>> level = 1.0
>>> function = ot.SymbolicFunction(['x0', 'x1'], ['x0^2+x1^2'])
>>> levelSet = ot.LevelSet(function, level)
>>> mesh = mesher.build(levelSet)
```

Methods

`build`(*args)	Build the mesh of levelset type.
`getClassName`()	Accessor to the object's name.
`getDiscretization`()	Accessor to the discretization.

- Content:
  - Programming interface (API)
  - Examples
  - Theory
- *All* classes and methods are documented, partly automatically.
- Examples are automatically tested at *each* update of the code and outputs are checked.

OpenTURNS: modules

- Different python modules based on the OpenTURNS core classes
- Developed and maintained by the consortium
- Coded either in C++ or directly in Python

Main modules

- **otagrum** : create a distribution from a Bayesian Network using aGrUM
- **otfftw** : Fast Fourier Transform algorithm (e.g. for stochastic processes) using FFTW
- **otfmi** : FMI models manipulation using PyFMI
- **otmixmod** : build mixtures of a multivariate Normal distribution from a sample
- **otmorris** : Morris screening method module
- **otpmml** : manages PMML files for meta-modeling exchanges
- **otpod**: A module to build Probability of Detection for Non Destructive Testing
- **otrobopt**: robust optimization
- **otsubsetinverse**: inverse subset simulation
- **otsvm** : Support Vector regression and classification with libsvm
- **otwrapy** : Python wrapper tools

Additional modules

- **otbenchmark** : benchmark problems for reliability and sensitivity analysis
- **othdrplot** : high density region algorithm for functional outlier detection
- **otsurrogate** : surrogate metamodels
- **otsklearn** : metamodels with the scikit-learn estimator API (fit/predict)
- **otusecases**: use cases suitable for OpenTURNS (functions and datasets)
- **otmarkov** : simulates Markov chains (experimental)
- **otsensitivity** : sensitivity analysis with density based measures

OpenTURNS: practical use

- Compatibility with most popular python packages
  - Numpy
  - Scipy
  - Matplotlib
  - scikit-learn

- Parallel computational with shared memory (TBB)

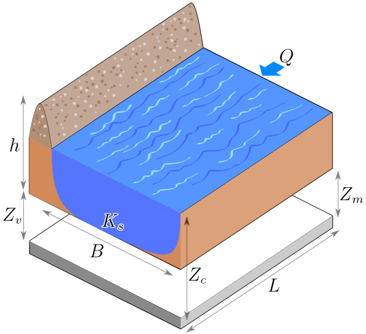
- Optimized linear algebra with LAPACK and BLAS

- Possibility to interface with a computation cluster

- Focused towards handling numerical data

- Installation through conda, pip and source code

Example : Flood Analysis test-case



Iooss, Bertrand, and Paul Lemaître. "A review on global sensitivity analysis methods." Uncertainty management in simulation-optimization of complex systems. Springer, Boston, MA, 2015. 101-122.

We consider the following random parameters:

- Q : Rate of flow(m^3/s)
- Ks : Strickler's coefficient ($m^{1/3}/s$)
- Zv/Zm : Downstream/Upstream elevation (m)

Additionally, we consider the following parameters:

- River section length $L = 5000$ (m)
- River Width $B = 300$ (m) / Dam height $H_d = 58.5$

We approximate :

$$\alpha = \frac{Z_m - Z_v}{L},$$

Therefore :

$$H = \left(\frac{Q}{K_s B \sqrt{\alpha}}\right)^{0.6},$$

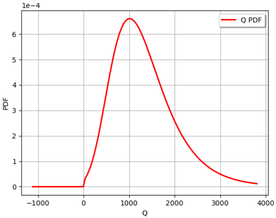
We want to analyse the overflow :

$$S = H + Zv - H_d$$

Probabilistic modeling

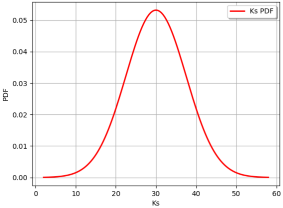
Random variables distributions :

Q : Gumbel(scale=558, mode=1013)>0



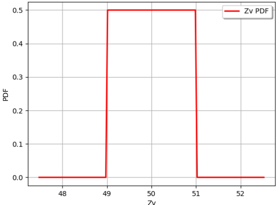
```
Dist = ot.Gumbel(558,1013)
Q = ot.TruncatedDistribution(Dist, 0.,
ot.TruncatedDistribution.LOWER)
```

Ks : Normal(mean=30, std=7.5)>0

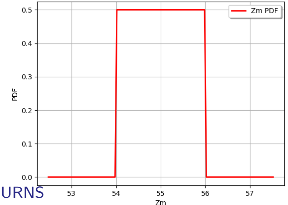


```
Dist = ot.Normal(30.,7.5)
Ks = ot.TruncatedDistribution(Dist, 0., ot.
    TruncatedDistribution.LOWER)
```

Zv : Uniform(min=49, max=51)

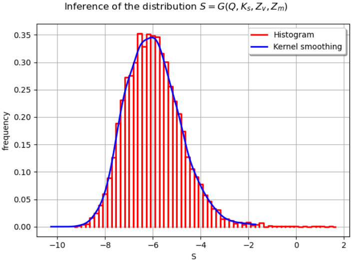


Zm : Uniform(min=54, max=56)



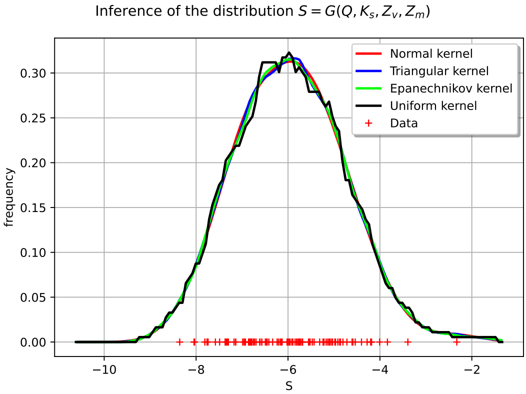
Monte-Carlo sampling

- The input distribution and relative output value are evaluated 10000 times
- The output distribution can be infered through histogram or kernel smoothing methods



```
Distribution = ot.ComposedDistribution([Q,Ks,Zv,Zm])

#Python model
def floodFunction(X) :
    Q, Ks, Zv, Zm = X
    alpha = (Zm - Zv)/5.0e3
    H = (Q/(300.0*Ks*np.sqrt(alpha)))**0.6
    S = [H + Zv - 58.5]
    return S

fun = ot.PythonFunction(4,1,floodFunction)

#We define the output as a random vector
inputVector = ot.RandomVector(Distribution)
outputVector = ot.CompositeRandomVector(fun,
 inputVector)

#We sample and infere the output distribution
size = 10000
sampleY = outputVector.getSample(size)
graph = ot.HistogramFactory().build(sampleY).drawPDF
     ()
loiKS = ot.KernelSmoothing().build(sampleY)
graph2 = loiKS.drawPDF()
graph.add(graph2)
graph.setTitle(r'Inference of the distribution
 $S=G(Q, K_s, Z_v, Z_m)$')
graph.setXTitle('S')
graph.setYTitle('frequency')
graph.setLegends(['Histogram', 'Kernel smoothing'])
graph.setColors(['red', 'blue'])
view = View(graph)
```

Distribution inference



- Parametric ($1d - Nd$) distribution inference
- Non-parametric ($1d - Nd$) distribution inference
- Parametric copula inference
- Non-parametric copula inference (Bernstein copula)
- Resampling w.r.t. infered distributions

```
size = 100
sampleY = outputVector.getSample(
     size)
graph = ot.KernelSmoothing(ot.
     Normal()).build(sampleY).
     drawPDF()
loiKS = ot.KernelSmoothing(ot.
     Triangular()).build(sampleY)
graph2 = loiKS.drawPDF()
graph.add(graph2)
loiKS = ot.KernelSmoothing(ot.
     Epanechnikov()).build(sampleY)
graph2 = loiKS.drawPDF()
graph.add(graph2)
loiKS = ot.KernelSmoothing(ot.
     Uniform()).build(sampleY)
graph2 = loiKS.drawPDF()
graph.add(graph2)
graph.add(ot.Cloud(sampleY,ot.
     Sample(100,1)))
graph.setTitle(r’Inference of the
     distribution $S=G(Q, K_s, Z_v,
      Z_m)$’)
graph.setXTitle(’S’)
graph.setYTitle(’frequency’)
graph.setLegends([’Normal kernel’,
     ’Triangular kernel’, ’
     Epanechnikov kernel’, ’Uniform
      kernel’,’Data’])
graph.setColors([’red’, ’blue’, ’
     green’, ’black’])
view = View(graph)
```

Iterative Monte-Carlo : Central tendency anaysis

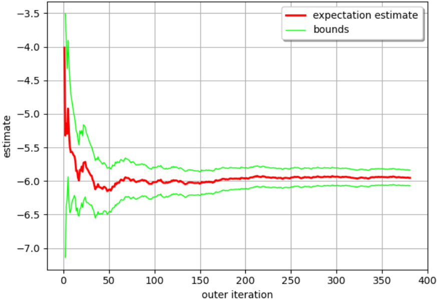
- The expected value and associated standard deviation are computed iteratively
- Different stopping criteria can be used : e.g., variation coefficient $= 0.01$
- Batch computation can be used

$$\widehat{m}_y = \frac{1}{N}\sum_1^N G(\mathbf{X}_i)$$

$$\widehat{\sigma}_y = \sqrt{\frac{1}{N}\sum_1^N (G(\mathbf{X}_i) - \hat{m}_y)^2}$$

$$\widehat{\sigma}_{m_y} = \widehat{\sigma}_y/\sqrt{N}$$

Expectation convergence graph at level 0.95



```
algo = ot.ExpectationSimulationAlgorithm(
 outputVector)
algo.setMaximumOuterSampling(100000)
algo.setBlockSize(1)
algo.setCoefficientOfVariationCriterionType(
 'MAX')
algo.setMaximumCoefficientOfVariation(0.01)
algo.run()
graph = algo.drawExpectationConvergence()
view = View(graph)
```

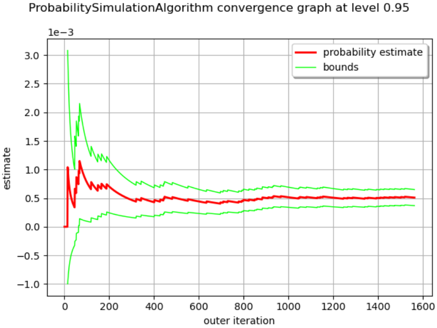
Iterative Monte-Carlo : Reliability analysis

- We now consider the probability of flooding : $(P(S > 0.))$
- Same as before, but the function $\mathbb{I}_{G(\mathbf{x}_i)>0}$ is considered

$$\widehat{p}_y = \frac{1}{N}\sum_{1}^{N} \mathbb{I}_{G(\mathbf{x}_i)>0}$$

$$\widehat{\sigma} = \sqrt{\frac{1}{N}\sum_{1}^{N} (\mathbb{I}_{G(\mathbf{x}_i)>0} - \hat{p}_y)^2}$$

$$\widehat{\sigma}_{p_y} = \widehat{\sigma}/\sqrt{N}$$



```
eventF = ot.ThresholdEvent(outputVector, ot.
     GreaterOrEqual(), 0.0)
exp = ot.MonteCarloExperiment()
algo = ot.ProbabilitySimulationAlgorithm(
     eventF, exp)
algo.setMaximumOuterSampling(100000)
algo.setMaximumCoefficientOfVariation(0.01)
algo.setBlockSize(10)
algo.run()
graph = algo.drawProbabilityConvergence()
view = View(graph)
```

FORM/SORM reliability analysis

- We estimate the probability of flooding through FORM/SORM procedures
- MC estimation requires ≃ 1500 function evaluations
- FORM and SORM only use ≃ 150

- Estimated probability :
  - MC : 5.09999999999998 1e-4
  - FORM : 5.340929030055227 1e-4
  - SORM : 6.793780433482759 1e-4

Also :

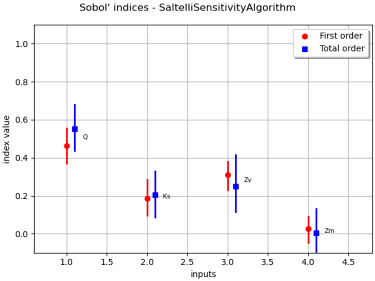
- Directional sampling
- Importance sampling
- Subset sampling

```
#FORM
OptAlgo = ot.Cobyla()
startingPoint = Distribution.getMean()
algoFORM = ot.FORM(OptAlgo, eventF,
startingPoint)
algoFORM.run()

#SORM
OptAlgo = ot.Cobyla()
startingPoint = Distribution.getMean()
algoSORM = ot.SORM(OptAlgo, eventF,
startingPoint)
algoSORM.run()
```

Different types and parameterizations of finite difference gradient computation are available

Sensitivity analysis : Sobol indices

- We want to identify the most influential random input parameters
- Different sensitivity analysis methods : Sobol, SRC, Morris
- We generate an appropriate design of experiment and use if to compute the Sobol indices of the 4 inputs
- Different estimators are available



$$S_i = \frac{\mathbb{V}[\mathbb{E}[Y|X_i]]}{\mathbb{Y}}$$

```
size = 1000
computeSecondOrder = False
sie = ot.SobolIndicesExperiment(Distribution
      , size, computeSecondOrder)
inputDesign = sie.generate()

outputDesign = fun(inputDesign)
sensitivityAnalysis = ot.
      SaltelliSensitivityAlgorithm(
      inputDesign, outputDesign, size)
sensitivityAnalysis.setBootstrapSize(300)
graph = sensitivityAnalysis.draw()
view = View(graph)
```

Sensitivity analysis : HSIC indices

- Probabilistic modeling of d input physical variables and black-box model:

$$\mathbf{X} = (X_1, X_2, \ldots, X_d)^\top \sim P_{\mathbf{X}} \quad \text{over} \quad \mathcal{X} = \times_{i=1}^{d} \mathcal{X}_i$$

$$\mathcal{M} : \left| \begin{array}{rcl} \mathcal{X} \subseteq \mathbb{R}^d & \longrightarrow & \mathcal{Y} \subseteq \mathbb{R} \\ \mathbf{X} & \longrightarrow & Y = \mathcal{M}(\mathbf{X}) \end{array} \right.$$

- **Learning sample** → a n-size i.i.d. sample of the couple $(\mathbf{X}, Y)$:

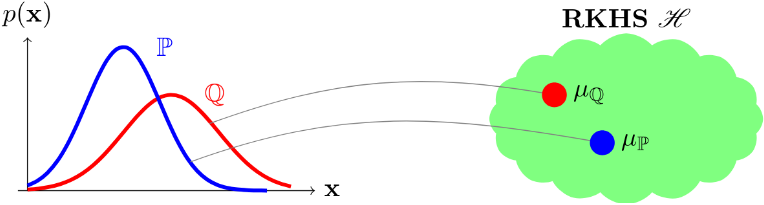
$$\left(\mathbf{X}^{(j)}, Y^{(j)}\right)_{(1 \leq j \leq n)} = \left(X_1^{(j)}, X_2^{(j)}, \ldots, X_d^{(j)}, Y^{(j)}\right)_{(1 \leq j \leq n)}$$

with $P_{\mathbf{X}^{(j)}} = P_{\mathbf{X}}$ and $Y^{(j)} = \mathcal{M}\left(X_1^{(j)}, X_2^{(j)}, \ldots, X_d^{(j)}\right), \forall j \in \{1, \ldots, n\}$

Sensitivity analysis : HSIC indices

- Let us now consider $\mathcal{V}$, an RKHS over $\mathcal{X} \times \mathcal{Y}$ with kernel $v(\cdot,\cdot)$
- We consider the mean embedding of $P_Y P_{X_i}$ and P_{Y,X_i} in $\mathcal{V}$
  - $\mu[P_Y P_{X_i}] = \mathbb{E}_Y \mathbb{E}_{X_i}[v((Y,X_i),\cdot)]$
  - $\mu[P_{Y,X_i}] = \mathbb{E}_{Y,X_i}[v((Y,X_i),\cdot)]$
- We now have a dependence measure under the form of:

$$\Delta := ||\mu[P_{Y,X_i}] - \mu[P_Y P_{X_i}]|| \longleftrightarrow HSIC(Y,X_i) = \Delta^2$$



Sensitivity analysis : HSIC indices

Estimators

- Different estimators, e.g., V-statistics

$$\widehat{\mathrm{HSIC}}(X_i, Y) = \frac{1}{n^2} \mathrm{Tr}(L_i H L H)$$

where L_i and L are Gram matrices, and H a shift matrix

→ Kernel-based estimator

- **A plug-in estimator of a normalized sensitivity index**

$$\widehat{R^2_{\mathrm{HSIC},i}} = \frac{\widehat{\mathrm{HSIC}}(X_i, Y)}{\sqrt{\widehat{\mathrm{HSIC}}(X_i, X_i)\ \widehat{\mathrm{HSIC}}(Y, Y)}} \qquad (1)$$

Sensitivity analysis : HSIC indices

A few advantages

- Data efficient
- Given data estimators
- Computational cost scales linearly with the problem dimension
- Associated statistical test
- Can be used when dealing with non continuous variables
    - discrete/categorical variables
    - graphs
    - stochastic codes
- Formulation holds in case of dependence between inputs
    - Interpretation of results becomes complicated!

Sensitivity analysis : HSIC indices

Statistical hypothesis testing with HSIC

- We consider the following statistical test $\mathcal{T}$

$$\mathcal{T} : \text{Test} \quad "(\mathcal{H}_{0,i}) : \mathrm{HSIC}(X_i, Y) = 0" \quad \text{vs.} \quad "(\mathcal{H}_{1,i}) : \mathrm{HSIC}(X_i, Y) > 0"$$

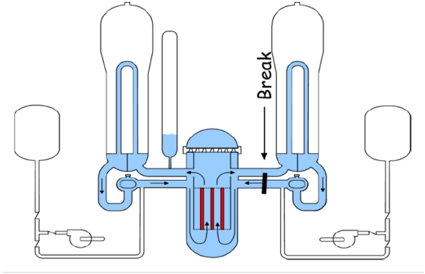
$\mathcal{H}_{0,i}$: X_i and Y are independent

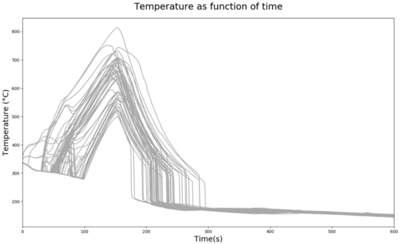
- $\widehat{S}_{\mathcal{T}} := n \times \widehat{\mathrm{HSIC}}(X_i, Y)$ is the test statistic
- **p-value** associated to the test $\mathcal{T}$:

$$p_{\mathrm{val}} = \mathbb{P}\left(\widehat{S}_{\mathcal{T}} \geq \widehat{S}_{\mathcal{T},\mathrm{obs}} \mid \mathcal{H}_{0,i}\right)$$

  - Asymptotic estimator (smal data set)
  - Permutation-based estimator (large data set)

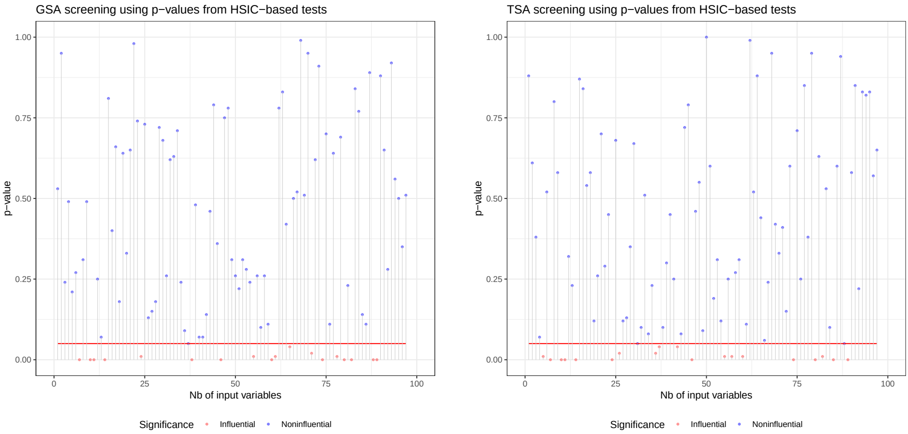
Sensitivity analysis : HSIC indices





- Illustrative scheme of an IBLOCA scenario (@CEA)
- Simulation trajectories of the Peak Cladding Temperature (PCT) (@EDF).

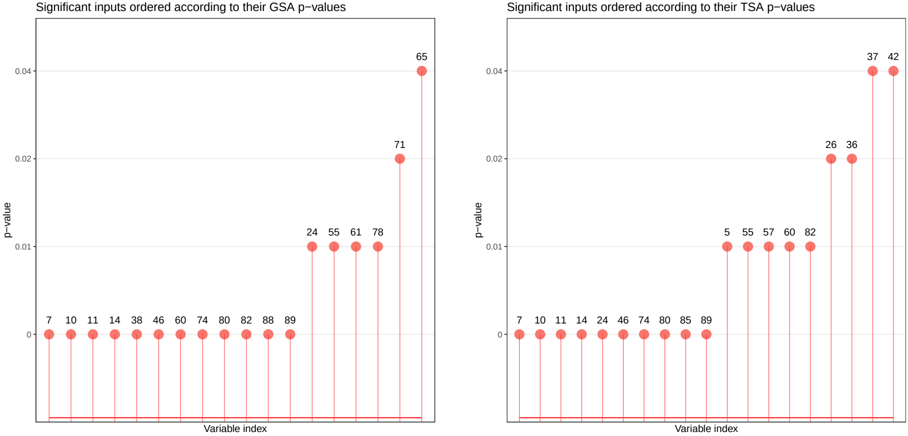
Sensitivity analysis : HSIC indices



GSA-oriented screening. TSA-oriented screening.

Screening for GSA & TSA using p-values.

Sensitivity analysis : HSIC indices



GSA-oriented ranking

TSA-oriented ranking

Ranking for GSA & TSA using p-values and R^2_{HSIC}

Sensitivity analysis : HSIC indices

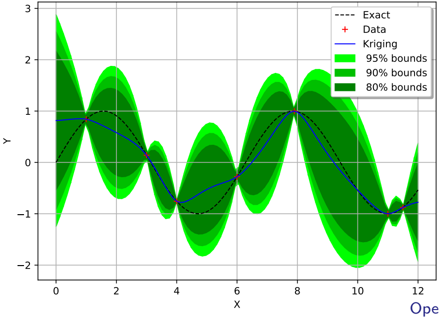
Work in progress!

Coming soon in OpenTURNS 1.19 ($\sim$ May 2022)

Surrogate modeling : Kriging

- Different surrogate modeling methods are available
  - Kriging
  - Polynomial chaos expansion
  - Linear regression
  - Low rank tensors

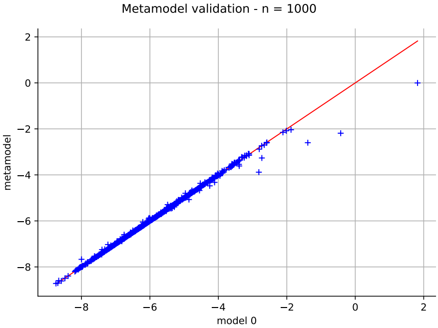
- Kriging
  - Different types of covariance functions and function basis can be used
  - User-defined options are also available
  - MLE optimization can be parameterized
  - Large number of optimization algorithms available



```
inputSample = Distribution.getSample(100)
outputSample = fun(inputSample)

dimension = 4
basis = ot.ConstantBasisFactory(dimension).
     build()
covarianceModel = ot.SquaredExponential()

algo = ot.KrigingAlgorithm(inputSample,
     outputSample, covarianceModel, basis)

algo.run()
result = algo.getResult()
KrigingMM = result.getMetaModel()
```

Surrogate modeling : Validation

- Automated validation of surrogate models with respect to a user provided data set
- Predictivity factor : $Q2 = 0.992615$
- $y - \hat{y}$ plot:



```
inputValidation = Distribution.getSample
     (1000)
outputValidation = fun(inputValidation)
validation = ot.MetaModelValidation(
     inputValidation,outputValidation,
     KrigingMM)
graph = validation.drawValidation()
view = View(graph)
print(validation.computePredictivityFactor()
     )
```

Optimization

- OpenTURNS provides an interface with several optimization libraries

- AUGLAG
- AUGLAG_EQ
- GD_MLSL
- GD_MLSL_LDS
- GD_STOGO
- GD_STOGO_RAND
- GN_AGS
- GN_CRS2_LM
- GN_DIRECT
- GN_DIRECT_L
- GN_DIRECT_L_NOSCAL
- GN_DIRECT_L_RAND
- GN_DIRECT_L_RAND
- GN_DIRECT_NOSCAL
- GN_ESCH,GN_ISRES
- GN_MLSL
- GN_MLSL_LDS
- GN_ORIG_DIRECT
- GN_ORIG_DIRECT_L
- G_MLSL
- G_MLSL_LDS
- LD_AUGLAG
- LD_AUGLAG_EQ
- LD_CCSAQ
- LD _LBFGS
- LD_MMA
- LD_SLSQP
- LD_TNEWTON
- LD_TNEWTON_PRECOND
- LD_VAR1
- LD_VAR2
- LN_AUGLAG
- LN_AUGLAG_EQ
- LN_BOBYQA
- LN_COBYLA
- LN_NELDERMEAD
- LN_NEWUOA
- LN_NEWUOA_BOUND
- LN_PRAXIS
- LN_SBPLX

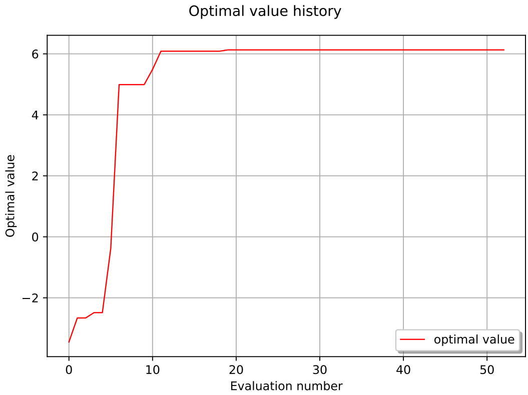
- Constrained and unconstrained optimization
- Bound and unbound optimization
- Multi-start wrapper

Optimization

We want to maximize the value of the overflow in order to identify the most dangerous configuration of input variables

```
#Setting the optimization bounds and starting point distribution
lbounds = [1e-3, 1e-3, 49, 54]
ubounds = [4000,60,51,56]
MSDistribution = []

for i in range(dimension):
    MSDistribution.append(ot.Uniform(lbounds[i],ubounds[i]))

MSDistribution = ot.ComposedDistribution(MSDistribution)
bounds = ot.Interval(ot.Point(lbounds),ot.Point(ubounds))

#Defining the optimization problem
problem = ot.OptimizationProblem(fun)
problem.setMinimization(False)
problem.setBounds(bounds)

#Defining and parameterizing the solver
algo = ot.MultiStart(ot.NLopt('LN_COBYLA'), MSDistribution.getSample(10))
algo.setProblem(problem)
algo.setMaximumRelativeError(1e-6)

#Running the optimization
algo.run()
```

Optimization



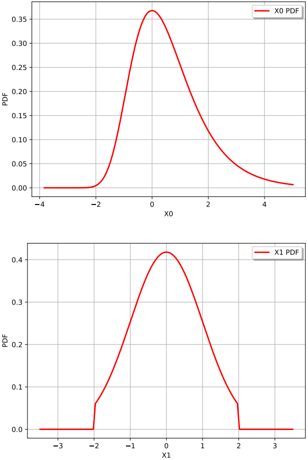
```
res = algo.getResult()
OptValue = res.getOptimalValue()
OptPoint = res.getOptimalPoint()
graph = res.drawOptimalValueHistory
      ()

print(OptValue)
[1.09192]

print(OptPoint)
[3638.55,13.5605,50.9784,54.0316]

print(lbounds)
[69.78551963374207,
      13.560460816677406,
      49.02524785938163,
      54.031603399244695]

print(ubounds)
[3638.549738548708,
      48.154203046100534,
      50.978367971379114,
      55.94973015924193]
```

- As expected, we hit the min/max bounds in order to find the maximal value of overflow

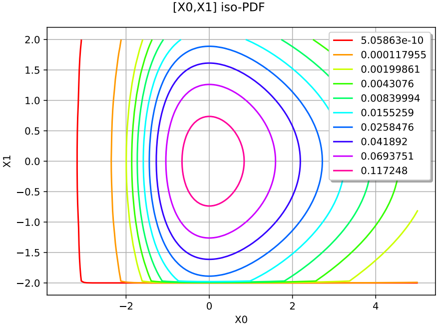
Design of experiments

- Different Design of experiment types are available
- We consider a 2-dimensional distribution with the following marginals :
  - Gumbel(min = -1, max = 1)
  - Truncated normal (mean = 0, std = 1, min = -2, max = 2)

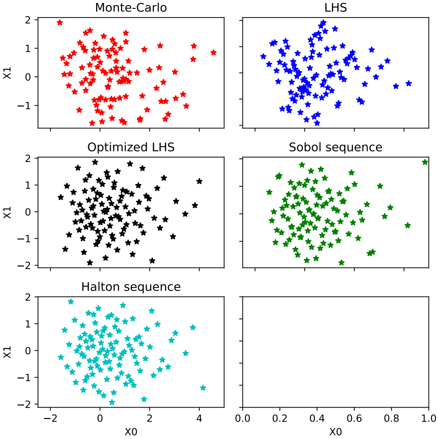
Marginals



Joint distributions

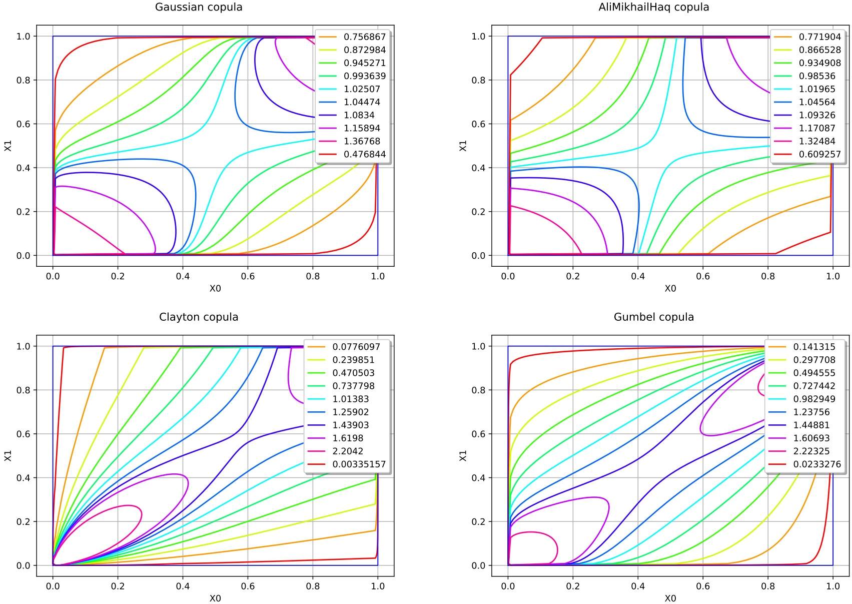

[X0,X1] iso-PDF

Design of experiments

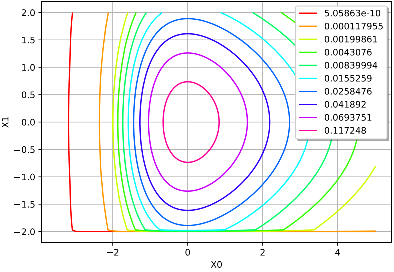


```
dim = 2
X = [ot.Gumbel(),ot.TruncatedNormal
     (0,1,-2,2)]
distribution = ot.ComposedDistribution(X)
bounds = distribution.getRange()
sampleSize = 100

sample1 = distribution.getSample(sampleSize)

experiment = ot.LHSExperiment(distribution,
  sampleSize, False, False)
sample2 = experiment.generate()

lhs = ot.LHSExperiment(distribution,
 sampleSize)
lhs.setAlwaysShuffle(True) # randomized
space_filling = ot.SpaceFillingC2()
temperatureProfile = ot.GeometricProfile
     (10.0, 0.95, 1000)
algo = ot.SimulatedAnnealingLHS(lhs,
 space_filling, temperatureProfile)
sample3 = algo.generate()

sequence = ot.SobolSequence(dim)
experiment = ot.LowDiscrepancyExperiment(
  sequence, distribution, sampleSize, False)
sample4 = experiment.generate()

sequence = ot.HaltonSequence(dim)
experiment = ot.LowDiscrepancyExperiment(
  sequence, distribution, sampleSize, False)
sample5 = experiment.generate()
```

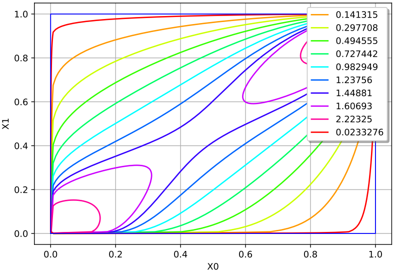
Beyond independent marginals : Copulas



Composing marginal distributions and copulas

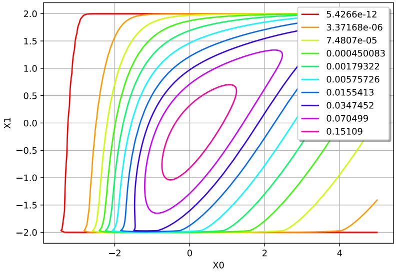


[X0,X1] iso-PDF



Gumbel copula

We obtain :



Composed Gumbel copula

```
distribution =
 [ot.Uniform(),ot.TruncatedNormal(0,1,-2,2)]
composed = ot.ComposedDistribution(X,copula)
graph = composed.drawPDF()
graph.setTitle('Composed Gumbel copula')
viewer.View(graph)
```

Different copulas can be used to link the various dimensions of the model

Example : Viscous free fall

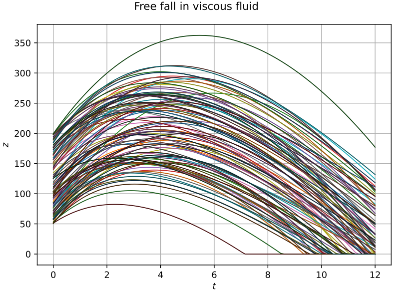
- We consider here the free fall of a sphere in a viscous fluid.
- We model the vertical trajectory of the ball as a function of 4 random input parameters :
    - z_0 : Initial height : Uniform(mean = 50.0, std = 200.0)
    - v_0 : Initial vertical speed : Normal(min 55.0, max = 10.0)
    - m : Ball mass : Normal(mean = 80.0, std = 8.0)
    - c : Fluid viscosity : Uniform(min = 0.0, max = 30.0)
- The model can be seen as a field function with the following expression :

$$\begin{aligned} z(t) &= z_0 + v_{inf} t + \tau (v_0 - v_{inf})(1 - e^{-t/\tau}) \quad \forall t \in [0, t_{max}] \\ \tau &= m/c \\ v_{inf} &= -m * g * c \end{aligned}$$

- If the model inputs are random, the output can be seen as a stochastic process

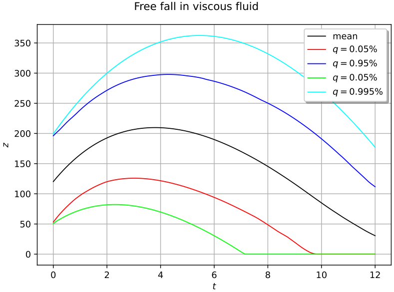
Steven C. Chapra. Applied numerical methods with Matlab for engineers and scientists, Third edition. 2012

Field function sampling


Free fall in viscous fluid

```
def AltiFunc(X):
    g  = 9.81
    z0,v0,m,c = X
    tau=m/c
    vinf=-m*g/c
    t = np.array(mesh.getVertices().asPoint())
    z=z0+vinf*t+tau*(v0-vinf)*(1-np.exp(-t/tau))
    z=np.maximum(z,0.0)
    return ot.Field(mesh, [[zeta] for zeta in z])

tmin=0.
tmax=12.
gridsize=100
mesh = ot.IntervalMesher([gridsize-1]).build(
ot.Interval(tmin, tmax))

alti = ot.PythonPointToFieldFunction(4, mesh,  1,
      AltiFunc)

distZ0 = ot.Uniform(50.0, 200.0)
distV0 = ot.Normal(55.0, 10.0)
distM = ot.Normal(80.0, 8.0)
distC = ot.Uniform(0.0, 30.0)
distX = ot.ComposedDistribution([distZ0, distV0,
 distM, distC])

size = 100
inputSample = distX.getSample(size)
outputField = alti(inputSample)
```

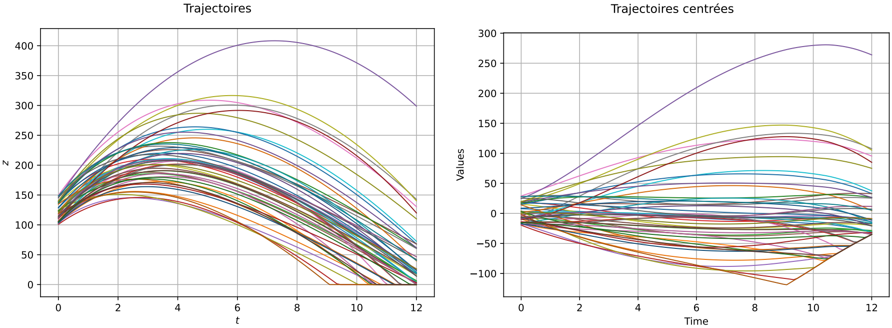
Field function analysis



```
meanField = outputField.computeMean()
graph = meanField.drawMarginal(0)
graph.setTitle(’Free fall in viscous fluid’)
graph.setXTitle(r’$t$’)
graph.setYTitle(r’$z$’)

quantileField_005 = outputField.computeQuantile
 PerComponent(0.05)
graph.add(quantileField_005.drawMarginal(0))

quantileField_095 = outputField.computeQuantile
 PerComponent(0.95)
graph.add(quantileField_095.drawMarginal(0))

quantileField_0005 = outputField.computeQuantile
 PerComponent(0.005)
graph.add(quantileField_0005.drawMarginal(0))

quantileField_0995 = outputField.computeQuantile
 PerComponent(0.995)
graph.add(quantileField_0995.drawMarginal(0))
```

Field function analysis

We center the trajectories with respect to the mean field:

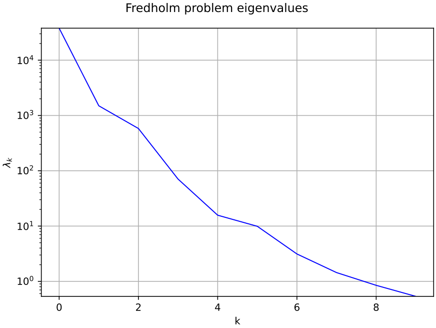


```
meanFunction = P1LagrangeEvaluation(meanField)
trend = TrendTransform(meanFunction, myMesh)
invTrend = trend.getInverse()
outputFieldCentered = invTrend(outputField)
```

Dimension reduction : Karhunen-Loève decomposition

- We wish to reduce the dimension of the problem from a infinite dimensional output to a finite dimensional one
- We can perform a Karhunen-Loève decomposition with a finite truncature
- This requires to solve a Fredholm's problem in order to identify the eigenfunctions and associated eigenvalues of the considered process

$$Y(\omega,\underline{t}) = \sum_{k=1}^{\infty} \sqrt{\lambda_k}\xi_k(\omega)\underline{\varphi}_k(\underline{t}) \rightarrow \tilde{Y}(\omega,\underline{t}) = \sum_{k=1}^{p} \sqrt{\lambda_k}\xi_k(\omega)\underline{\varphi}_k(\underline{t})$$

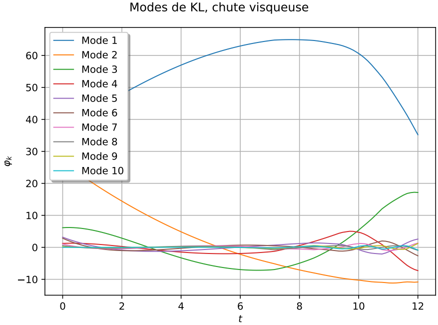


```
meanFunction = ot.P1LagrangeEvaluation(
 meanField)
trend = ot.TrendTransform(meanFunction,
     myMesh)
invTrend = trend.getInverse()
outputFieldCentered = invTrend(outputField)

truncThreshold = 1.0e-5
algo = ot.KarhunenLoeveSVDAlgorithm(
 outputFieldCentered, truncThreshold)
algo.run()
KLResult = algo.getResult()

eigenValues = KLResult.getEigenValues()
```

Dimension reduction : Karhunen-Loève decomposition

$$\tilde{Y}(\omega, \underline{t}) = \sum_{k=1}^{p} \sqrt{\lambda_k} \xi_k(\omega) \underline{\varphi}_k(\underline{t})$$

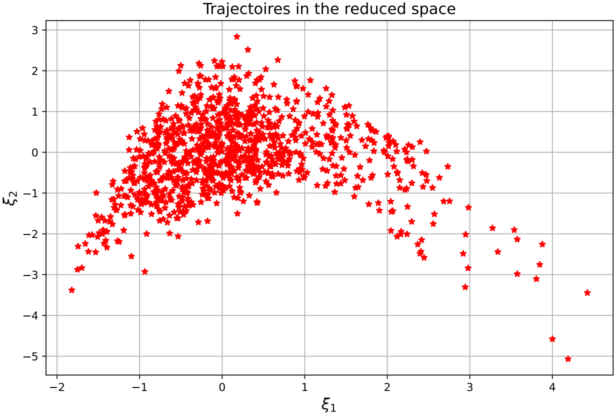
Main modes :



```
scaledModes =
 KLResult.getScaledModesAsProcessSample()
graph = scaledModes.drawMarginal(0)
graph.setTitle('Modes de KL, chute visqueuse
     ')
graph.setXTitle(r'$t$')
graph.setYTitle(r'$\varphi_k$')
leg = ot.Description([ 'Mode '+str(i +1) for
  i in range(eigenValues.getDimension()) ])
graph.setLegends(leg)
graph.setLegendPosition('topleft')
view=View(graph)
```

Dimension reduction : Karhunen-Loève decomposition

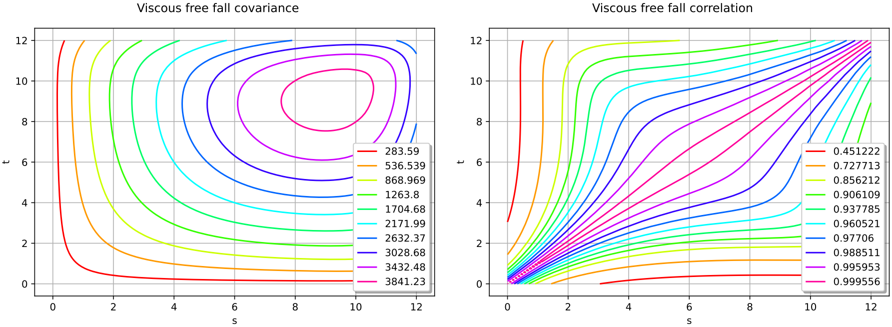
We only consider the first 2 terms of the decomposition :



```
projectionFunction = ot.KarhunenLoeveProjection(KLResult)
sampleKsi = projectionFunction(outputFieldCentered)
sampleKsi = sampleKsi[:,:2]
```

Field function analysis

We center the trajectories with respect to the mean field:



```
cov = KLResult.getCovarianceModel()

# As a covariance function
isStationary = False
asCorrelation = False
graph = cov.draw(0, 0, tmin, tmax, 128, isStationary, asCorrelation)
view=View(graph)

# As a correlation function
asCorrelation = True
graph = cov.draw(0, 0, tmin, tmax, 128, isStationary, asCorrelation)
view=View(graph)
```

Field function surrogate model

We can combine the Karhunen-Loève decomposition and a polynomial chaos regression in order to predict new fields :

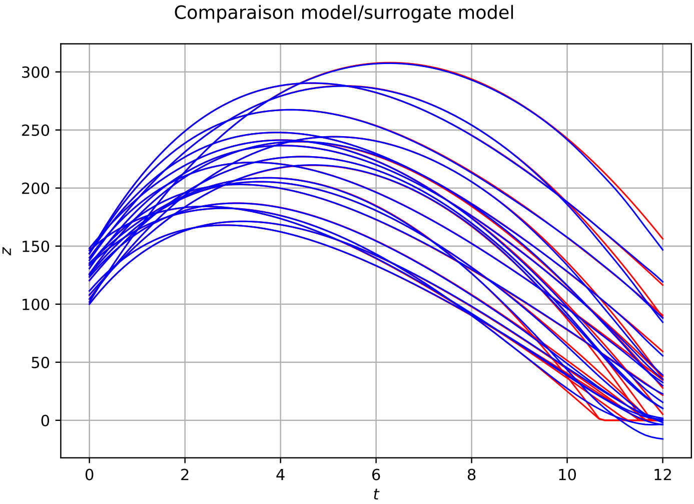
1. Train a projection function on a training data set : Field space → Reduced space
2. Define the lifting function : Reduced space → Field space
3. Train a polynimial chaos regressor : Input space → Reduced space
4. Generate a new sample in the Input space
5. Predict its image in the reduced space → Lift the result in the Field space

$$\begin{tikzcd}
X \arrow[r, "g"] \arrow[dr, "g_\xi"] & Y \arrow[d, shift right, "\pi"'] \\
& (\xi_k)_{k\in\mathbb{N}} \arrow[u, shift right, "(\pi')^{-1}"']
\end{tikzcd}$$

Field function surrogate model

```
# Creation of the orthonormal polynomial basis
enumerateFunction = ot.HyperbolicAnisotropicEnumerateFunction(distX.getDimension(), 0.7)
basis = ot.OrthogonalProductPolynomialFactory([ot.StandardDistributionPolynomialFactory(
     distX.getMarginal(i)) for i in range(distX.getDimension())])

# Creation of the basis selection strategy
degree = 7
adaptive = ot.FixedStrategy(basis, enumerateFunction.getStrataCumulatedCardinal(degree))

# Input space to Reduced space PCE
algo = FunctionalChaosAlgorithm(inputSample, sampleKsi, distX, adaptive)
algo.run()

polyChaos = algo.getResult().getMetaModel()
polyResults = algo.getResult()

# Creation of the Input to Field space surrogate model
procZ_centre = ot.PointToFieldConnection(liftFunction, polyChaos)
metaModel = ot.PointToFieldConnection(trend, procZ_centre)
```

Field function surrogate model

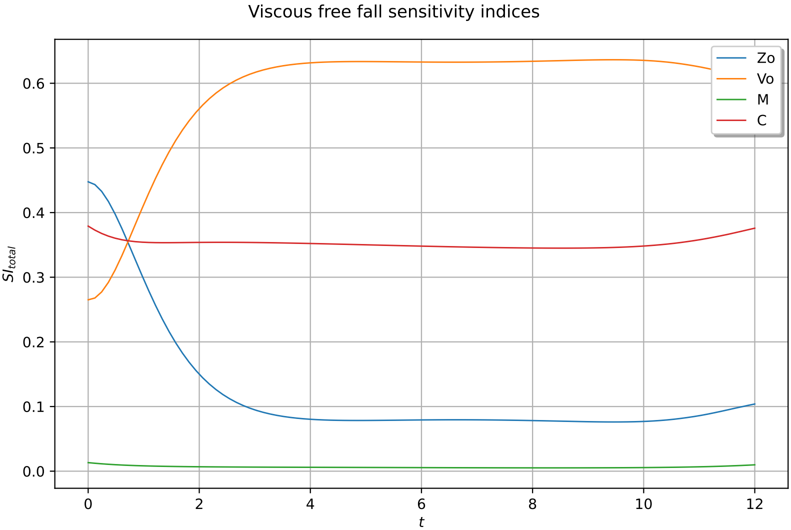


Time dependent sensitivity analysis

The Polynomial chaos expansion provides the Sobol indices as a free byproduct of the surrogate model training. We can use this feature in order to compute the sensitivity indices as a function of time

```
chaosSI = FunctionalChaosSobolIndices(polyResults)
chaosRV = FunctionalChaosRandomVector(polyResults)
Modes = KLResult.getModesAsProcessSample()

SIField = ProcessSample()

VarFieldValues = 0.
for i in range(eigenValues.getSize()):
    VarFieldValues = VarFieldValues + eigenValues[i] * ( np.array( Modes.getMarginal(0)[i
     ] )**2 * chaosRV.getCovariance()[i,i] )

for j in range( distX.getDimension()):
    VarFieldValuesX = 0.
    for i in range(eigenValues.getSize()):
        VarFieldValuesX = VarFieldValuesX + eigenValues[i] * ( np.array( Modes.
     getMarginal(0)[i] )**2 * chaosRV.getCovariance()[i,i] * chaosSI.getSobolTotalIndex(j
     ,i) )
    SIField.add( Field(myMesh, VarFieldValuesX / VarFieldValues) )

graph = SIField.drawMarginal()
graph.setTitle(r'Viscous free fall sensitivity indices')
graph.setXTitle(r'$t$')
graph.setYTitle(r'$SI_{total}$')
graph.setLegends(distX.getDescription())
view = View(graph)
```

Time dependent sensitivity analysis

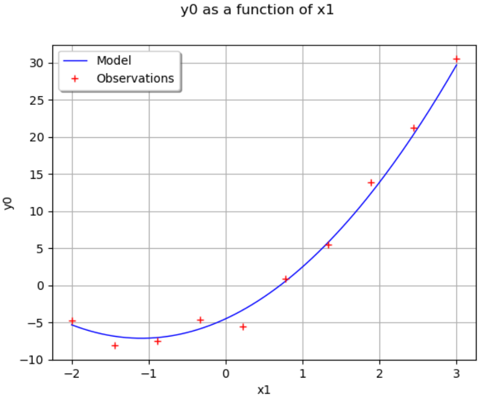


Calibration of a computer code

- The objective is to determine the optimal parameters of a computer code by using the available data
- We consider the following model :

$$y = \theta_0 + x\theta_1 + x^2\theta_2 + \varepsilon$$

- True values : $\theta = \{-4.5, 4.8, 2.2\}$
- 10 observations :



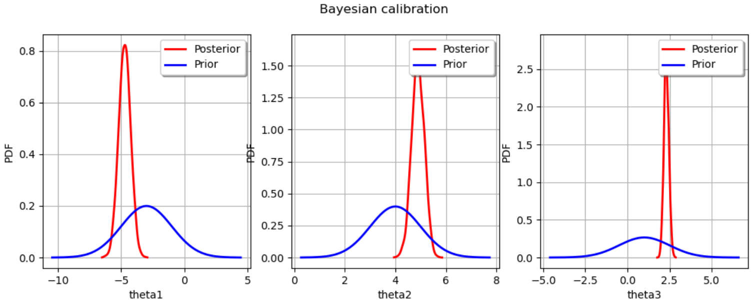
- Linear least squares
- Non-linear least squares
- Gaussian calibration
- Bayesian calibration

Bayesian calibration

- We consider a multivariate Gaussian prior on θ :

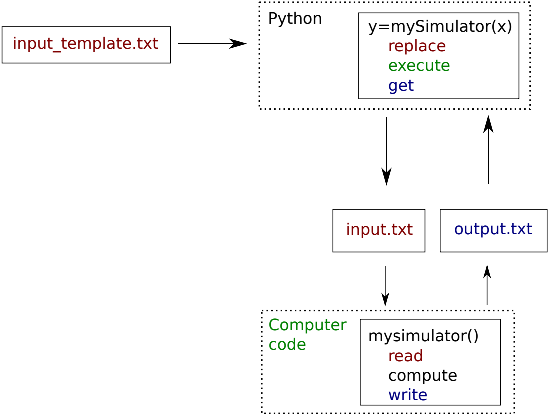
$$\theta \sim \mathcal{N}\left(\mu \begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix}, \sigma = \begin{bmatrix} 2^2 & 0 & 0 \\ 0 & 1^2 & 0 \\ 0 & 0 & 1.5^2 \end{bmatrix}\right)$$

- We can then use a MCMC to sample from the conditioned posterior distribution (1000 samples + Kernel smoothing):



Coupling OpenTURNS with computer codes

OpenTURNS provides a text file exchange based interface in order to perform analyses on complex computer codes



- Replaces the need for input/output text parsers
- Wraps a simulation code under the form of a standard python function
- Allows to interface OpenTURNS with a cluster

Support, discussion and contribution

- github repository : `github.com/openturns/openturns`
  - Bug report
  - Enhancement suggestions
  - Contribute
  - Review contributions
- Discourse forum : `https://openturns.discourse.group/`
  - Practical questions
  - Theoretical questions
  - Feature request
  - Forum layout
- Gitter chat : `https://gitter.im/openturns`
  - Practical questions
  - Theoretical questions
  - Feature request
  - Chat layout

PERSALYS, the graphical user interface of OpenTURNS

- Graphical interface of OpenTURNS
- Features :
  - Probabilistic modeling
  - Distribution fitting
  - Central tendency
  - Sensitivity analysis
  - Probability estimate
  - Meta-modeling (polynomial chaos, kriging)
  - Screening (Morris)
  - optimization
  - Design of experiments
  - 1-D field analysis
- Partners : EDF, Phiméca
- Licence : LGPL
- `persalys.fr`

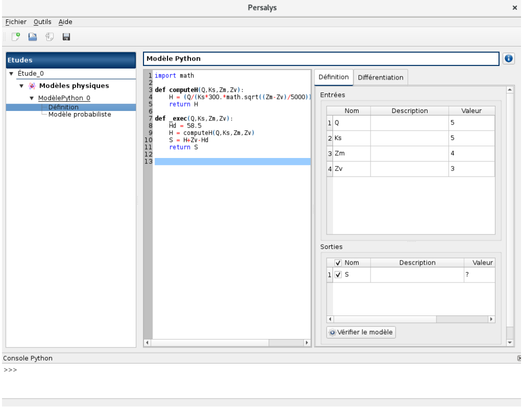
  `salome-platform.org`

Baudin, Michaël, et al. "The graphical user interface of OpenTURNS, a UQ software in simulation." (2017).

PERSALYS: model definition

Different options for defining the model :

- Analytical formula
- Python script
- Numerical code called through a file exchange system
- FMU models
- Data import



```python
import math

def computeH(Q,Ks,Zm,Zv):
    H = (Q/(Ks*300.*math.sqrt((Zm-Zv)/5000))
    return H

def _exec(Q,Ks,Zm,Zv):
    Hd = 58.5
    H = computeH(Q,Ks,Zm,Zv)
    S = H+Zv-Hd
    return S
```

Entrées

	Nom	Description	Valeur
1	Q		5
2	Ks		5
3	Zm		4
4	Zv		3

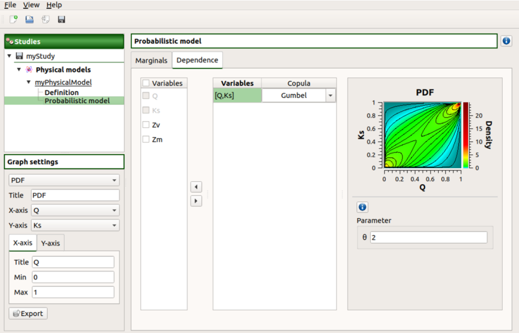
Sorties

	☑ Nom	Description	Valeur
1	☑ S		?

PERSALYS: probabilistic model

The probabilistic model can be defined in different ways:

- Independent parametric marginals
- Marginal coupled through copulas
- Inference of marginals from data
- Inference of copulas from data



Variables	Copula
[Q,Ks]	Gumbel

θ 2

PDF

Title PDF

X-axis Q

Y-axis Ks

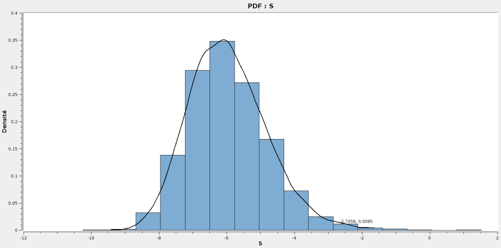
Title Q

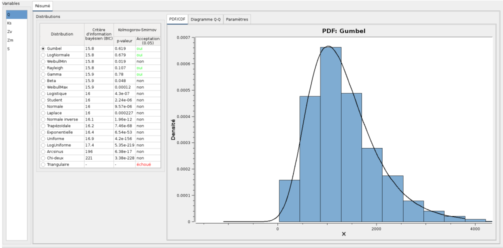
Min 0

Max 1

PERSALYS: data analysis

- Parametric and non-parametric distribution inference



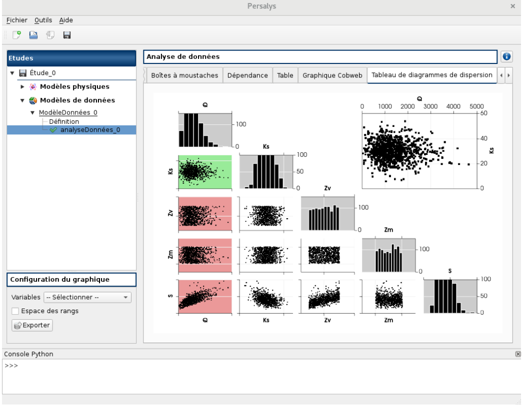


Distribution	Critère d'information bayésien (BIC)	Kolmogorov-Smirnov	
		p-valeur	Acceptation (0.05)
● Gumbel	15.8	0.619	oui
○ LogNormale	15.8	0.679	oui
○ WeibullMin	15.8	0.019	non
○ Rayleigh	15.8	0.107	oui
○ Gamma	15.9	0.78	oui
○ Beta	15.9	0.048	non
○ WeibullMax	15.9	0.00012	non
○ Logistique	16	4.3e-07	non
○ Student	16	2.24e-06	non
○ Normale	16	9.57e-06	non
○ Laplace	16	0.000227	non
○ Normale inverse	16.1	1.96e-12	non
○ Trapézoidale	16.2	7.46e-68	non
○ Exponentielle	16.4	6.54e-53	non
○ Uniforme	16.9	4.2e-156	non
○ LogUniforme	17.4	5.35e-219	non
○ Arcsinus	196	6.38e-17	non
○ Chi-deux	221	3.38e-228	non
○ Triangulaire	-	-	échoué

PERSALYS: data analysis

Visual data analysis :

- Pair-wise scatter plots
- Empirical histogram
- Interactive and linked views
- Physical and rank spaces



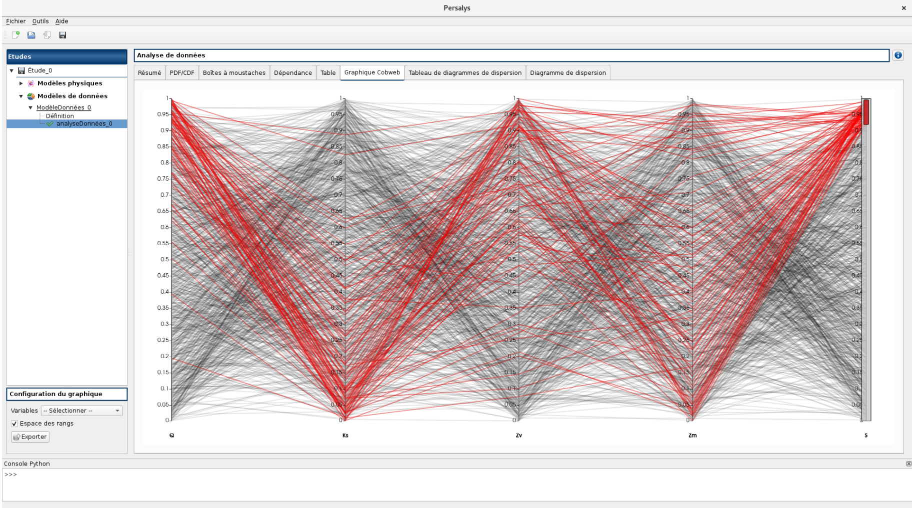
Variables -- Sélectionner --

☐ Espace des rangs

```
>>>
```

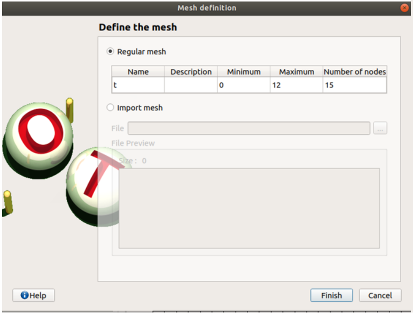
PERSALYS: data analysis

Interactive graphical data analysis



PERSALYS: 1D fields

- Mesh definition and visualization
- Import from text or csv file



Define the mesh

● Regular mesh

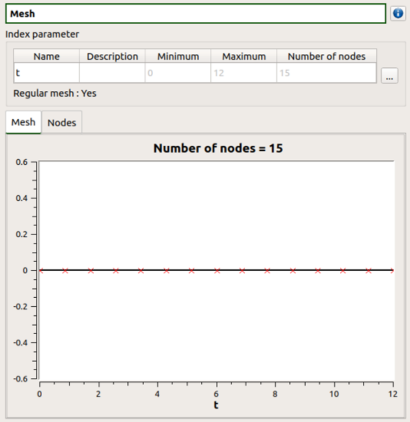
Name	Description	Minimum	Maximum	Number of nodes
t		0	12	15

○ Import mesh

File

File Preview

Size : 0



Mesh

Index parameter

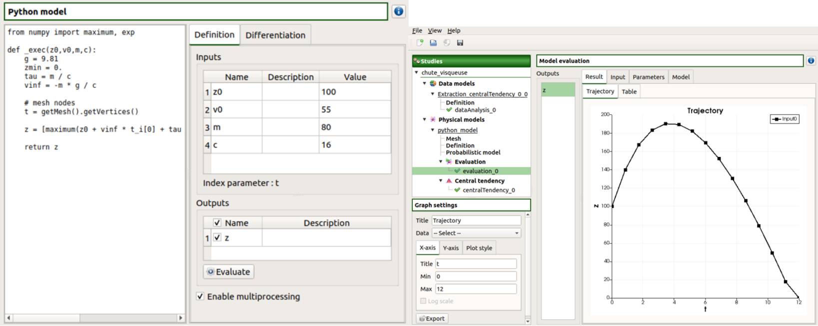
Name	Description	Minimum	Maximum	Number of nodes
t		0	12	15

Regular mesh : Yes

Number of nodes = 15

PERSALYS: 1D fields

- Functional model definition and probabilistic model
- Python or symbolic



Python model

```
from numpy import maximum, exp

def _exec(z0,v0,m,c):
    g = 9.81
    zmin = 0.
    tau = m / c
    vinf = -m * g / c

    # mesh nodes
    t = getMesh().getVertices()

    z = [maximum(z0 + vinf * t_i[0] + tau

    return z
```

Inputs

	Name	Description	Value
1	z0		100
2	v0		55
3	m		80
4	c		16

Index parameter : t

Outputs

	☑ Name	Description
1	☑ z	

☑ Enable multiprocessing

Graph settings

Title: Trajectory

Data: -- Select --

Title: t

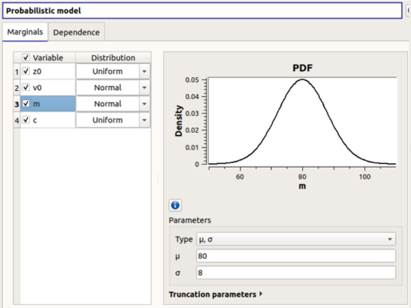
Min: 0

Max: 12

☐ Log scale

PERSALYS: 1D fields

- Probabilistic model
- Uncertainty propagation with simple Monte-Carlo sampling



Probabilistic model

Marginals | Dependence

	☑ Variable	Distribution
1	☑ z0	Uniform
2	☑ v0	Normal
3	☑ m	Normal
4	☑ c	Uniform

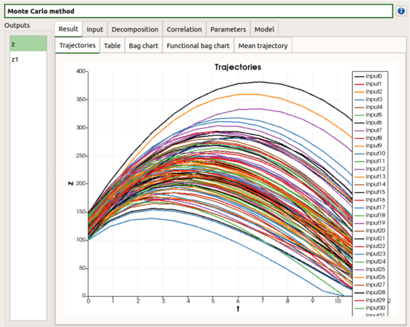
Parameters

Type: μ, σ

μ: 80

σ: 8

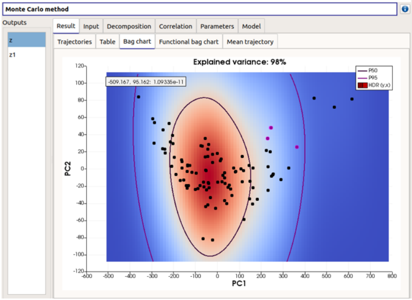
Truncation parameters

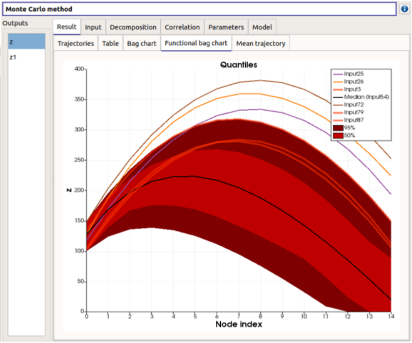


Monte Carlo method

PERSALYS: 1D fields

- BagChart and Functional Bagchart (from Paraview) based on High Density Regions (Hyndman, 1996).
- To do this, Paraview uses a principal component analysis decomposition.
- Linked and interactive selections in the views.





Hyndman, Rob J., and Han Lin Shang. "Rainbow plots, bagplots, and boxplots for functional data." Journal of Computational and Graphical Statistics 19.1 (2010): 29-45.

Support, discussion and contribution

- github repository : `https://github.com/persalys/persalys`
  - Bug report
  - Enhancement suggestions
  - Mirror of theinternal development git repository

- Discourse forum : `https://persalys.discourse.group/`
  - Practical questions
  - Theoretical questions
  - Feature request
  - Forum layout

Thank you for your attention! :)