PyRE Documentation

Release 5.0.1

Juergen Hackl

May 11, 2013

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

Date 11 May 2013

Authors Jürgen Hackl

Contact hackl.j@gmx.at

Web site http://github.com/hackl/pyre

License PyRe is released under the GNU General Public Licence.

Version 5.0.1

Warning: This is a preliminary program code!
I wrote `pyre` for my Master's Thesis. The work isn't finished yet, so changes in the code are very likely!

Note: If you have any problems, found bugs in the code or have feature request comments or questions, please feel free to send a mail to Jürgen Hackl.

1.1 Purpose

PyRe (python Reliability) is a python module for structural reliability analysis. Its flexibility and extensibility make it applicable to a large suite of problems. Along with core reliability analysis functionality, PyRe includes methods for summarizing output.

Note: At the moment only First-Order Reliability Methods are supported! Second-Order Reliability Methods and Monte Carlo Simulation will hopefully follow soon :)

1.2 History

The FERUM (Finite Element Reliability Using Matlab) project was initiated in 1999 at the University of California, Berkeley, by Terje Haukaas and Armen Der Kiureghian, primarily for pedagogical purposes aimed at teaching and learning structural reliability and stochastic finite elements methods. [DerKiureghian2006] This code consists of an open-source Matlab toolbox, featuring various structural reliability methods. The latest available version (version 3.1), which can be downloaded from FERUM. Since 2003, this code is no longer officially maintained. [Bourinet2010]

A new version of this open-source code (FERUM 4.x) based on a work carried out at the Institut Français de Mécanique Avancée (IFMA) in Clermont-Ferrand, France. This version offers improved capabilities such as simulation-based technique (Subset Simulation), Global Sensitivity Analysis (based on Sobol's indices), Reliability-Based Design Optimization (RBDO) algorithm, Global Sensitivity Analysis and reliability assessment based on Support Vector Machine (SVM) surrogates, etc. Beyond the new methods implemented in this Matlab code. [Bourinet2009]

Of the purpose, to use structural reliability analysis for the project "Risk based decision framework for the optimal management of aging reinforced concrete structures" [Hackl2013] a python version of FERUM has been created.

The focus here lies on the reliability analysis and not more on the finite element method, so only the core function of FERUM are implemented.

1.3 Features

PyRe provides functionalities to make structural reliability analysis as easy as possible. Here is a short list of some of its features:

- Perform reliability analysis with First-Order Reliability Methods
- Includes a large suite of well-documented statistical distributions.
- Uses NumPy for numerics wherever possible.
- No limitation on the limit state function
- Correlation between the random variables are possible
- Traces can be saved to the disk as plain text.
- PyRe can be embedded in larger programs, and results can be analyzed with the full power of Python.

1.4 Getting started

This guide provides all the information needed to install PyRe, code a reliability model, run the sampler, save and visualize the results. In addition, it contains a list of the statistical distributions currently available.

CHAPTER

TWO

INSTALLATION

Date 11 May 2013

Authors Jürgen Hackl

Contact hackl.j@gmx.at

Web site http://github.com/hackl/pyre

Version 5.0.1

PyRe is known to run on Mac OS X, Linux and Windows, but in theory should be able to work on just about any platform for which Python, a Fortran compiler and the NumPy SciPy, and Math modules are available. However, installing some extra depencies can greatly improve PyRe's performance and versatility. The following describes the required and optional dependencies and takes you through the installation process.

2.1 Dependencies

PyMC requires some prerequisite packages to be present on the system. Fortunately, there are currently only a few hard dependencies, and all are freely available online.

- Python version 2.6 or later.
- NumPy : The fundamental scientific programming package, it provides a multidimensional array type and many useful functions for numerical analysis.
- SciPy : Library of algorithms for mathematics, science and engineering.
- IPython (optional): An enhanced interactive Python shell and an architecture for interactive parallel computing.

2.2 Installation using EasyInstall

Not available at the moment.

2.3 Installing from pre-built binaries

Not available at the moment.

2.4 Compiling the source code

First download the source code from GitHub and unpack it. Then move into the unpacked directory and follow the platform specific instructions.

2.4.1 Windows

Not available at the moment.

2.4.2 Mac OS X or Linux

Not available at the moment.

2.5 Development version

You can check out the development version of the code from the GitHub repository:

```
git clone git://github.com/hackl/pyre.git
```

2.6 Bugs and feature requests

Report problems with the installation, bugs in the code or feature request at the issue tracker. Comments and questions are welcome and should be addressed to Jürgen Hackl.

CHAPTER

THREE

TUTORIAL

This tutorial will guide you through a typical PyMC application. Familiarity with Python is assumed, so if you are new to Python, books such as [Lutz2007] or [Langtangen2009] are the place to start. Plenty of online documentation can also be found on the Python documentation page.

3.1 An example reliability model

Consider the following random variables:

$$X_1 \sim \text{Logormal}(500, 100) \tag{3.1}$$
$$X_2 \sim \text{Normal}(2000, 400) \tag{3.2}$$
$$X_3 \sim \text{Uniform}(5, 0.5) \tag{3.3}$$

Additionaly those variables are relatet to each other. Therefore the correlationmatrix $\mathbf{C}$ is given:

$$\mathbf{C} = \begin{pmatrix} 1.0 & 0.3 & 0.2 \\ 0.3 & 1.0 & 0.2 \\ 0.2 & 0.2 & 1.0 \end{pmatrix} \tag{3.4}$$

Now, we like to compute the reliability index β and the failure probability P_f, by given limit state function $g(X_1, X_2, X_3)$:

$$g(X_1, X_2, X_3) = 1 - \frac{X_2}{1000 \cdot X_3} - \left(\frac{X_1}{200 \cdot X_3}\right)^2 \tag{3.5}$$

3.2 Let's model

Before we start with the modeling, we have to import the `pyre` package. Therefore are two different methods available:

In case 1 we load `pyre` like a normal library:

```
import pyre
```

here we must write for each command `pyre.my_command()`. A much nicer way to load the package is case 2:

```
# import pyre library
from pyre import *
```

here, we import all available objects from `pyre`.

To define the random variables from (3.1) we can use following syntax:

```
# Define random variables
X1 = Lognormal('X1',500,100)
X2 = Normal('X2',2000,400)
X3 = Uniform('X3',5,0.5)
```

The first parameter is the name of the random variable. The name has to be a string, so the input looks like `'X3'`.

By default, the next to values are the first and second moment of the distribution, here mean and standard deviation. Are mean and standard deviation unknown but the distribution parameter known, then the `input_type` has to be changed.

For example random variable X_3 is uniform distributed. Above we assume that X_3 is defined by mean and standard deviation. But we can describe the distribution with the parameter a and b. In this case the code will look like:

```
X3 = Uniform('X3',4.133974596215562, 5.866025403784438, 1)
```

to get the same results as before. To see which parameters are needed and in which order the must insert, take a look at Chapter *Probability distributions*. There are all currently available distributions listed.

In the same way, we can add the correlation matrix to our model:

```
# Define Correlation Matrix
Corr = CorrelationMatrix([[1.0, 0.3, 0.2],
                          [0.3, 1.0, 0.2],
                          [0.2, 0.2, 1.0]])
```

Are the variables uncorrelated, you don't have to add a correlation matrix to the model.

At least we have to define the limit state function. Therefore are two ways:

- Direct in the `main` code,
- in a separate `function`.

In the first case the input will look like:

```
# Define limit state function
# - case 1: define directly
g = LimitStateFunction('1 - X2*(1000*X3)**(-1) - (X1*(200*X3)**(-1))**2')
```

and in the second case like this:

```
# Define limit state function
# - case 2: define load function, wich is defined in function.py
g = LimitStateFunction('function(X1,X2,X3)')
```

The function `'function(X1,X2,X3)'` can be found in `'function.py'`. This case can be useful if the limit state function is quiet complex or need more then one line to define it. Here `'function.py'` is defined as:

```
def function(X1,X2,X3):
  g = 1 - X2*(1000*X3)**(-1) - (X1*(200*X3)**(-1))**2
  return g
```

At this stage our model is complete defined and we can start the analysis.

3.3 Reliability Analysis

To store the results from the analysis an object must be initialized:

```
# Performe FORM analysis
Analysis = Form()
```

Now the code can be compiled and the FORM analysis will be preformed. In this example we will get following results:

```
==================================================

 RESULTS FROM RUNNING FORM RELIABILITY ANALYSIS

 Number of iterations:      17
 Reliability index beta:    1.75397614074
 Failure probability:       0.039717297753
 Number of calls to the limit-state function: 164

==================================================
```

If we don't like to see the results in the terminal the option `printResults(False)` has set to be `False`. There are also some other options which can be modified (see *Reliability Model*).

To change some options, a object must be initialized which stores the customized options.

```
# Set some options (optional)
options = AnalysisOptions()
options.printResults(False)
```

and This options must be implemented in our analysis:

```
# Performe FORM analysis
Analysis = Form(options)
```

To use the results for further calculations, plots etc. the results can get by some getter methods (see *Calculations*)

```
# Some single results:
beta = Analysis.getBeta()
failure = Analysis.getFailure()
```

3.4 Finally...

This was a short introduction how to use `pyre`. The tutorial above is also available on GitHub under `example.py`.

Let's have fun ;)

CHAPTER

FOUR

THEORETICAL BACKGROUND

4.1 Reliability

Structural reliability analysis is concerned with the rational treatment of uncertainties [Melchers1999]. These uncertainties could be broadly grouped into three. Thoft- Christensen and Baker (1982) classified them into physical uncertainties, statistical uncertainties and model uncertainties.

The order of the listed uncertainties corresponds approximately to the decreasing level of current knowledge and available theoretical tools for their description and consideration in design. Most of the uncertainties can never be eliminated absolutely and must be taken into account by engineers when designing any construction work [Melchers1999].

4.1.1 First-Order Reliability Method (FORM)

First-Order Reliability Method (FORM) aims at using a first-order approximation of the limit-state function in the standard space at the so-called Most Probable Point (MPP) of failure P^* (ordesign point), which is the limit-state surface closest point to the origin. Finding the coordinates $\mathbf{u}^*$ of the MPP consists in solving the following constrained optimization problem:

$$\mathbf{u} = \arg\min \{ \|\mathbf{u}\| \, | g(\mathbf{x}(\mathbf{u}), \theta_g) = G(\mathbf{u}, \theta_g) = 0 \}$$

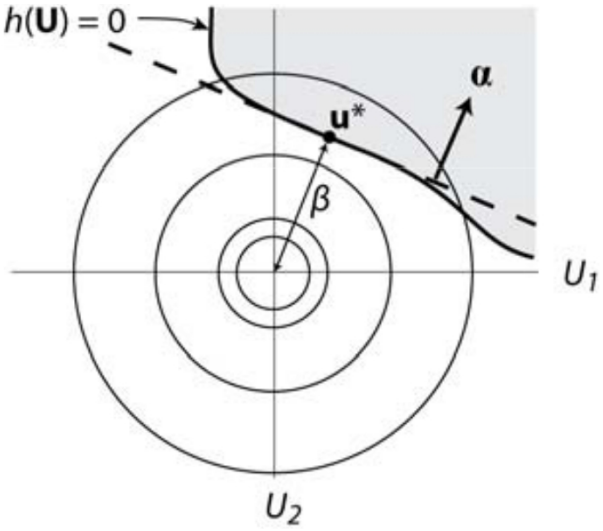
Once the MPP P^* is obtained, the Hasofer and Lind reliability index β is computed as $\beta = \alpha^T \mathbf{u}^*$ where $\alpha = -\nabla_u G(\mathbf{u}^*)/\|\nabla_u G(\mathbf{u}^*)\|$ is the negative normalized gradient vector at the MPP P*. It represents the distance from the origin to the MPP in the standard space. The first-order approximation of the failure probability is then given by $p_{f1} = \Phi(-\beta)$, where $\Phi(\cdot)$ is the standard normal cdf. The same technique is applied to step size evaluation with Armijo rule, where all corresponding g-calls are sent simultaneously. [Bourinet2010]

4.1.2 Armijo Rule

Denote a univariate function ϕ restricted to the direction $\mathbf{p}_k$ as $\phi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{p}_k)$. A step length α_k is said to satisfy the Wolfe conditions if the following two inequalities hold:

$$i) f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k)$$

$$ii) \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \geq c_2 \mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k)$$



with $0 < c_1 < c_2 < 1$. (In examining condition (ii), recall that to ensure that $\mathbf{p}_k$ is a descent direction, we have $\mathbf{p}_k^{\mathrm{T}} \nabla f(\mathbf{x}_k) < 0$.)

c_1 is usually chosen to be quite small while c_2 is much larger; Nocedal gives example values of $c_1 = 10^{-4}$ and $c_2 = 0.9$ for Newton or quasi-Newton methods and $c_2 = 0.1$ for the nonlinear conjugate gradient method. Inequality i) is known as the Armijo rule and ii) as the curvature condition; i) ensures that the step length α_k decreases f 'sufficiently', and ii) ensures that the slope has been reduced sufficiently.

4.2 Cholesky decomposition

The Cholesky decomposition of a Hermitian positive-definite matrix A is a decomposition of the form

$$\mathbf{A} = \mathbf{L}\mathbf{L}^*$$

where L is a lower triangular matrix with positive diagonal entries, and L* denotes the conjugate transpose of L. Every Hermitian positive-definite matrix (and thus also every real-valued symmetric positive-definite matrix) has a unique Cholesky decomposition.

If the matrix A is Hermitian and positive semi-definite, then it still has a decomposition of the form A = LL* if the diagonal entries of L are allowed to be zero.

When A has real entries, L has real entries as well.

The Cholesky decomposition is unique when A is positive definite; there is only one lower triangular matrix L with strictly positive diagonal entries such that A = LL*. However, the decomposition need not be unique when A is positive semidefinite.

The converse holds trivially: if A can be written as LL* for some invertible L, lower triangular or otherwise, then A is Hermitian and positive definite.

4.3 Gauss base points and weight factors

using the algorithm given by Davis and Rabinowitz in 'Methods of Numerical Integration', page 365, Academic Press, 1975.

CHAPTER

FIVE

RELIABILITY MODEL

5.1 Inputparameter

class pyre.model.**AnalysisOptions**

Ro_method = None
Method for computation of the modified Nataf correlation matrix

Ro_methods:

- 0: use of approximations from ADK's paper (no longer supported)
- 1: exact, solved numerically

block_size = None
Block size

Number of g-calls to be sent simultaneously

differentation_modus = None
Kind of differentiation

- 'ddm': direct differentiation,
- 'ffd': forward finite difference

e1 = None
Tolerance on how close design point is to limit-state surface

e2 = None
Tolerance on how accurately the gradient points towards the origin

ffdpara = None
Parameter for computation

Parameter for computation of FFD estimates of gradients - Perturbation = stdv/analysisopt.ffdpara

Recommended values:

- 1000 for basic limit-state functions,
- 50 for FE-based limit-state functions

flag_sens = None
Flag for computation of sensitivities

w.r.t. means, standard deviations, parameters and correlation coefficients

- 1: all sensitivities assessed,

- 0: no sensitivities assessment

i_max = None
Maximum number of iterations allowed in the search algorithm

multi_proc = None
Amount of g-calls

1: block_size g-calls sent simultaneously

- gfunbasic.m is used and a vectorized version of gfundata.expression is available. The number of g-calls sent simultaneously (block_size) depends on the memory available on the computer running FERUM.
- gfunxxx.m user-specific g-function is used and able to handle block_size computations sent simultaneously, on a cluster of PCs or any other multiprocessor computer platform.

0: g-calls sent sequentially

print_output = None
Print comments during calculation

•True: FERUM interactive mode,

•False: FERUM silent mode

step_size = None
Step size

0: step size by Armijo rule, otherwise: given value is the step size

transf_type = None
Type of joint distribution

transf_types:

- 1: jointly normal (no longer supported)
- 2: independent non-normal (no longer supported)
- 3: Nataf joint distribution (only available option)

class pyre.model.**LimitState**

evaluator = None
Type of limit-state function evaluator:

'basic': the limit-state function is defined by means of an analytical expression or a Matlab m-function, using gfundata(lsf).expression. The function gfun.m calls gfunbasic.m, which evaluates gfundata(lsf).expression.

'xxx': the limit-state function evaluation requires a call to an external code. The function gfun.m calls gfunxxx.m, which evaluates gfundata(lsf).expression where gext variable is a result of the external code.

expression = None
Expression of the limit-state function

flag_sens = None
Flag for computation of sensitivities

w.r.t. thetag parameters of the limit-state function

- 1: all sensitivities assessed,

- 0: no sensitivities assessment

class pyre.model.**StochasticModel**

CHAPTER

SIX

CALCULATIONS

6.1 Reliability

6.1.1 First-Order Reliability Method (FORM)

class pyre.form.**Form**(*analysis_options=None*, *limit_state=None*, *stochastic_model=None*)

6.1.2 Second-Order Reliability Method (FORM)

6.2 Matrices Operators

6.2.1 Cholesky decomposition

6.2.2 Transformation

6.3 Numerical Integration

CHAPTER

SEVEN

PROBABILITY DISTRIBUTIONS

PyRe provides a large suite of built-in probability distributions. For each distribution, it provides:

- A function that evaluates its marginal distribution.
- A function that evaluates its probability density function.
- A function that evaluates its cumulative distribution function.
- A function that compute its jacobian.
- A function that transform from from u to x space.
- A function that transform from from x to u space.

7.1 Normal distribution

class pyre.distributions.normal.**Normal**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)
Normal distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean
- stdv (float): Standard deviation
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Normal.**cdf**(*x*, *mean=None*, *stdv=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod Normal.**inv_cdf**(*P*)
inverse cumulative distribution function

classmethod Normal.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Normal.**pdf**(*x*, *mean=None*, *stdv=None*, *var_3=None*, *var_4=None*)
probability density function

Normal.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Normal.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Normal.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.2 Lognormal distribution

class pyre.distributions.lognormal.**Lognormal**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)
Lognormal distribution

Arguments

- name (str): Name of the random variable
- mean (float): Mean or lamb
- stdv (float): Standard deviation or zeta
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Lognormal.**cdf**(*x*, *lamb=None*, *zeta=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod Lognormal.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Lognormal.**pdf**(*x*, *lamb=None*, *zeta=None*, *var_3=None*, *var_4=None*)
probability density function

Lognormal.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Lognormal.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Lognormal.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.3 Gamma distribution

class pyre.distributions.gamma.**Gamma**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)
Gamma distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or lamb
- stdv (float): Standard deviation or k
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Gamma.**cdf**(*x*, *lamb=None*, *k=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod Gamma.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Gamma.**pdf**(*x*, *lamb=None*, *k=None*, *var_3=None*, *var_4=None*)
probability density function

Gamma.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Gamma.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Gamma.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.4 Shifted exponential distribution

class pyre.distributions.shiftedexponential.**ShiftedExponential**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)

Shifted exponential distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or lamb
- stdv (float): Standard deviation or x_zero
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod ShiftedExponential.**cdf**(*x*, *lamb=None*, *x_zero=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod ShiftedExponential.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod ShiftedExponential.**pdf**(*x*, *lamb=None*, *x_zero=None*, *var_3=None*, *var_4=None*)
probability density function

ShiftedExponential.**setMarginalDistribution**()
Compute the marginal distribution

classmethod ShiftedExponential.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod ShiftedExponential.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.5 Shifted Rayleigh distribution

class pyre.distributions.shiftedrayleigh.**ShiftedRayleigh**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)

Shifted Rayleigh distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or a
- stdv (float): Standard deviation or x_zero
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod ShiftedRayleigh.**cdf**(*x*, *a=None*, *x_zero=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod ShiftedRayleigh.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod ShiftedRayleigh.**pdf**(*x*, *a=None*, *x_zero=None*, *var_3=None*, *var_4=None*)
probability density function

ShiftedRayleigh.**setMarginalDistribution**()
Compute the marginal distribution

classmethod ShiftedRayleigh.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod ShiftedRayleigh.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.6 Uniform distribution

class pyre.distributions.uniform.**Uniform**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)

Uniform distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or a
- stdv (float): Standard deviation or b
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Uniform.**cdf**(*x*, *a=None*, *b=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod Uniform.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Uniform.**pdf**(*x*, *a=None*, *b=None*, *var_3=None*, *var_4=None*)
probability density function

Uniform.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Uniform.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Uniform.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.7 Beta distribution

class pyre.distributions.beta.**Beta**(*name*, *mean*, *stdv*, *a=0*, *b=1*, *input_type=None*, *startpoint=None*)
Beta distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or q
- stdv (float): Standard deviation or r
- a (float): Lower boundary
- b (float): Uper boundary
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Beta.**cdf**(*x*, *q=None*, *r=None*, *a=None*, *b=None*)
cumulative distribution function

classmethod Beta.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Beta.**pdf**(*x*, *q=None*, *r=None*, *a=None*, *b=None*)
probability density function

Beta.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Beta.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Beta.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.8 Chi square distribution

class pyre.distributions.chisquare.**ChiSquare**(*name*, *mean*, *stdv=None*, *input_type=None*, *startpoint=None*)
Chi-Square distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or nu
- stdv (float): Standard deviation
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod ChiSquare.**cdf**(*x*, *nu=None*, *var_2=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod ChiSquare.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod ChiSquare.**pdf**(*x*, *nu=None*, *var_2=None*, *var_3=None*, *var_4=None*)
probability density function

ChiSquare.**setMarginalDistribution**()
Compute the marginal distribution

classmethod ChiSquare.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod ChiSquare.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.9 Type I largest value distribution

class pyre.distributions.typeilargestvalue.**TypeIlargestValue**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)

Type I largest value distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or u_n
- stdv (float): Standard deviation or a_n
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod TypeIlargestValue.**cdf**(*x*, *u_n=None*, *a_n=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod TypeIlargestValue.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod TypeIlargestValue.**pdf**(*x*, *u_n=None*, *a_n=None*, *var_3=None*, *var_4=None*)
probability density function

TypeIlargestValue.**setMarginalDistribution**()
Compute the marginal distribution

classmethod TypeIlargestValue.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod TypeIlargestValue.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.10 Type I smallest value distribution

class pyre.distributions.typeismallestvalue.**TypeIsmallestValue**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)

Type I smallest value distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or u_1
- stdv (float): Standard deviation or a_1
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod TypeIsmallestValue.**cdf**(*x*, *u_1=None*, *a_1=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod TypeIsmallestValue.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod TypeIsmallestValue.**pdf**(*x*, *u_1=None*, *a_1=None*, *var_3=None*, *var_4=None*)
probability density function

TypeIsmallestValue.**setMarginalDistribution**()
Compute the marginal distribution

classmethod TypeIsmallestValue.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod TypeIsmallestValue.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.11 Type II largest value distribution

class pyre.distributions.typeiilargestvalue.**TypeIIlargestValue**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)

Type II largest value distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or u_n
- stdv (float): Standard deviation or k
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod TypeIIlargestValue.**cdf**(*x*, *u_n=None*, *k=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod TypeIIlargestValue.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod TypeIIlargestValue.**pdf**(*x*, *u_n=None*, *k=None*, *var_3=None*, *var_4=None*)
probability density function

TypeIIlargestValue.**setMarginalDistribution**()
Compute the marginal distribution

classmethod TypeIIlargestValue.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod TypeIIlargestValue.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.12 Type III smallest value distribution

class pyre.distributions.typeiiismallestvalue.**TypeIIIsmallestValue**(*name*, *mean*, *stdv*, *epsilon=0*, *input_type=None*, *startpoint=None*)

Type III smallest value distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or u_1
- stdv (float): Standard deviation or k
- epsilon (float): Epsilon
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod TypeIIIsmallestValue.**cdf**(*x*, *u_1=None*, *k=None*, *epsilon=None*, *var_4=None*)
cumulative distribution function

classmethod TypeIIIsmallestValue.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod TypeIIIsmallestValue.**pdf**(*x*, *u_1=None*, *k=None*, *epsilon=None*, *var_4=None*)
probability density function

TypeIIIsmallestValue.**setMarginalDistribution**()
Compute the marginal distribution

classmethod TypeIIIsmallestValue.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod TypeIIIsmallestValue.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.13 Gumbel distribution

class pyre.distributions.gumbel.**Gumbel**(*name*, *mean*, *stdv*, *input_type=None*, *startpoint=None*)
Gumbel distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or u_n
- stdv (float): Standard deviation or a_n
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Gumbel.**cdf**(*x*, *u_n=None*, *a_n=None*, *var_3=None*, *var_4=None*)
cumulative distribution function

classmethod Gumbel.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Gumbel.**pdf**(*x*, *u_n=None*, *a_n=None*, *var_3=None*, *var_4=None*)
probability density function

Gumbel.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Gumbel.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Gumbel.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

7.14 Weibull distribution

class pyre.distributions.weibull.**Weibull**(*name*, *mean*, *stdv*, *epsilon=0*, *input_type=None*, *startpoint=None*)
Weibull distribution

Attributes

- name (str): Name of the random variable
- mean (float): Mean or u_1
- stdv (float): Standard deviation or k
- epsilon (float): Epsilon
- input_type (any): Change meaning of mean and stdv
- startpoint (float): Start point for seach

classmethod Weibull.**cdf**(*x*, *u_1=None*, *k=None*, *epsilon=None*, *var_4=None*)
cumulative distribution function

classmethod Weibull.**jacobian**(*u*, *x*, *marg*, *J=None*)
Compute the Jacobian

classmethod Weibull.**pdf**(*x*, *u_1=None*, *k=None*, *epsilon=None*, *var_4=None*)
probability density function

Weibull.**setMarginalDistribution**()
Compute the marginal distribution

classmethod Weibull.**u_to_x**(*u*, *marg*, *x=None*)
Transformation from u to x

classmethod Weibull.**x_to_u**(*x*, *marg*, *u=None*)
Transformation from x to u

CHAPTER

EIGHT

LIST OF REFERENCES

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Bourinet2009] J.-M. Bourinet, C. Mattrand, and V Dubourg. A review of recent features and improvements added to FERUM software. In Proc. of the 10th International Conference on Structural Safety and Reliability (ICOSSAR'09), Osaka, Japan, 2009.

[Bourinet2010] J.-M. Bourinet. FERUM 4.1 User's Guide, 2010.

[DerKiureghian2006] 1. Der Kiureghian, T. Haukaas, and K. Fujimura. Structural reliability software at the University of California, Berkeley. Structural Safety, 28(1-2):44–67, 2006.

[Hackl2013] 10. Hackl. Risk based decision framework for the optimal management of aging reinforced concrete structures. Master's thesis, Norwegian University of Science and Technolog, Trondheim, Norway, 2013. (in work)

[Langtangen2009] Hans Petter Langtangen. Python Scripting for Computational Science. Springer-Verlag, 2009.

[Lutz2007] 13. Lutz. Learning Python. O'Reilly, 2007.

[Melchers1999] R.D. Melchers. structural reliability analysis and prediction Ellishorwood limited, Manchester, 1999.

PYTHON MODULE INDEX

p