"OpenQuake: Calculate, share, explore"

# Risk Modeller's Toolkit - User Guide

PUBLISHED BY GEM FOUNDATION

GLOBALQUAKEMODEL.ORG/OPENQUAKE

**Citation**
Please cite this document as:
Casotto, C. (2014) OpenQuake Risk Modeller's Toolkit - User Guide. *Global Earthquake Model (GEM). Technical Report*

*First printing, November 2014*

# Contents

# Preface

The goal of this book is to provide a comprehensive and transparent description of the methodologies adopted and implemented in the Risk Modeller's Toolkit (RMTK).

# 1. Introduction

## 1.1 Introduction

The Risk Modeller's Toolkit (*rmtk* hereafter) is a Python 2.7 library of functions written by scientists at the GEM Model Facility, which is intended to provide scientists and engineers with the tools to help creating the vulnerability input models that go into the OpenQuake risk engine and managing the generated output files. The intention of this software is to provide scientists and engineers with the means to apply some of the most commonly used algorithms for preparing vulnerability models using structural analysis data and to facilitate the visualisation and the use of the outputs from OpenQuake. In forthcoming versions will hope to make available more methodologies for the process indicated here, and to integrate new functionalities for i) building structural models of different levels of complexity within the *rmtk* in combination with a structural analysis software, ii) running dynamic and static analysis within the *rmtk* in combination with a structural analysis software, iii) deriving vulnerability curves directly applying engineering demand parameters-to-loss functions to structural analysis results.

### 1.1.1 Getting Started and Running the Software

The Modeller's Toolkit is designed for execution from the command line. As with OpenQuake engine, the preferred environment is Ubuntu Linux (12.04 or later), however, the *rmtk* is a library that can be run using any python environment for other operating systems (i.e. Mac, Windows). A careful effort has been made to keep the number of additional dependencies to a minimum. No packaged version of the software has been released at the time of writing, so the user must install Python 2.7 and the dependencies manually. The current dependencies are:

- Numpy and Scipy (included in the standard OpenQuake installation)
- Matplotlib (http://matplotlib.org/)
- Os

The Numpy, Scipy, Matplotlib, and Os dependencies are installed in the library for the demos, but once Python 2.7 has been installed they can be easily installed from the command line by:

```
~$ sudo pip install numpy
~$ sudo pip install scipy
~$ sudo pip install matplotlib
~$ sudo pip install os
```

To enable usage of the *rmtk* within any location in the operating system, OSX and Linux users should add the folder location (set the path) manually to the command line profile file. This can be done as follows:

1. Using a command line text editor (e.g. VIM), open the /.profile folder as follows:

```
~$ vim ~/.profile
```

2. At the bottom of the profile file (if one does not exist it will be created) add the line:

```
export PYTHONPATH=/path/to/rmtk/folder/:$PYTHONPATH
```

   Where `/path/to/rmtk/folder/` is the system path to the location of the *rmtk* folder (use the command `pwd` from within the *rmtk* folder to view the full system path).
3. Re-source the bash shell via the command

```
~$ source ~/.profile
```

### Windows Installation

Although this installation has been primarily tested in a Linux/Unix environment it is possible to install natively in Windows using the following process. This assumes that no other version of Python is installed in your windows environment.

The easiest way to install all of the dependencies needed is by virtue of the PythonXY program http://code.google.com/p/pythonxy/, a free and open python user interface, which will bring in all the dependencies nedeed automatically. The installer for the latest version of PythonXY can be downloaded from here: http://code.google.com/p/pythonxy/wiki/Downloads?tm=2.

Click on the executable and follow the instructions (the installation may take up to half an hour or more, depending on the system). It is strongly recommended that the use opt for the "**FULL**" installation, which should bring in all of the dependencies needed for the *rmtk*.

Now, download the zipped folder of the *rmtk* from the github repository and unzip to a folder of your choosing. To allow for usage of the *rmtk* throughout your operating system, do the following:

1. From the desktop, right-click **My Computer** and open **Properties**
2. In the "System Properties" window click on the **Advanced** tab.
3. From the "Advanced" section open the **Environment Variables**.
4. In the "Environment Variables" you will see a list of "System Variables", select "Path" and "Edit".
5. Add the path to the *rmtk* directory to the list of folders then save.

After this process it may be necessary to restart PythonXY.

## 1.2 Current features

The Risk Modeller's Toolkit is currently divided into two sections:

1. **Vulnerability** These functions are intended to address the modeller's needs for defining vulnerability curves, implementing methodologies differing for level of complexity and for the input data available for the buildings under study. GEM analytical vulnerability guidelines have been integrated in this tool and some of the methodologies indicated have been already implemented in the library.
2. **Plotting** These functions are intended to address the needs of visualising the results of the calculations performed with the OpenQuake engine.

## 1.3 Organization

This manual is designed to explain the various functions in the toolkit, to provide the theoretical background behind them, and to guide the modeller in the use of the rmtk within the "IPython Notebook" environment. This novel tool implements Python inside a web-browser environment, permitting the user to execute real Python workflows, whilst allowing for images and text to be embedded. Its use is encouraged especially for beginner python users for a more visual application of the rmtk.

The IPython Notebook comes installed from version 1.0 of IPython, that can be installed from the python package repository by entering:

```
~$ sudo pip install ipython
```

A notebook session can be started via the command:

```
~$ ipython notebook --pylab inline
```

The tutorial itself does not specifically require a working knowledge of Python. However, an understanding of the basic python data types is highly desirable. Users who are new to Python are recommended to familiarise themselves with Appendix A of this tutorial.

The *rmtk* is currently subdivided into two classes of tools, the Vulnerability and Plotting tools, presented in Chapter 2 and Chapter 3 of this tutorial respectively. In the Vulnerability chapter the vulnerability methodologies implemented are classified in Non-linear Static (NLS) and Non-linear Dynamic (NLD) according to the structural analysis type performed to assess the response of the building. These two main sections (NLS and NLD) are organised as follows:

- General Introduction.
- Getting Started, where it is explained what files need to be executed to start the vulnerability analysis, and what options are available to call the preferred methodology and to input the preferred data type.
- Description of the methodologies.

Within the description of each methodology the user can find the following subsections:

- Theoretical description of the method.
- Description and examples of the inputs.
- Description of the workflow.

A summary of the algorithms available in the present version is given in Table 1.1.

| Feature | Algorithm |
|---|---|
| **Non-linear Static** | Cr-based (**Ruiz-Garcia and Miranda**; **2007** )<br>Spo2ida (**Vamvatsikos and Cornell**; **2006** )<br>R-$/mu$-T-based (**Dolsek and Fajfar**; **2004** ) |
| **Non-linear Dynamic** | DPM-based (**Silva et al.**; **2013** )<br>Ida-postprocessing (**Vamvatsikos and Cornell**; **2002** ) |

**Table 1.1** – *Current algorithms in the HMTK*

# 2. Vulnerability

## 2.1 Non-linear Static (NLS) Methods

Nonlinear Static Methods are based on the use of capacity curves resulting from nonlinear static pushover analysis to determine the median seismic intensity values $\hat{s}_c$ corresponding to the attainment of a certain damage state threshold (limit state) and the corresponding dispersion $\beta_{sc}$. These parameters are used to represent a fragility curve as the probability of the limit state capacity C being exceeded by the demand D, both expressed in terms of intensity levels (s$_c$ and s respectively), as shown in the following equation:

$$P_{LS}(s) = P(C < D|s) = \Phi(\frac{lns - ln\hat{s}_c}{\beta_{sc}}) \tag{2.1}$$

The methodology implemented so far in the RMTK allows to consider different shapes of the pushover curve, multilinear and bilinear, record-to-record dispersion, dispersion in the damage state thresholds.

Different input types can be inserted depending on whether the user has already at his disposal an idealised pushover curve or it has to be derived from the raw results of a pushover analysis. Fragility and vulnerability functions can be derived for a single building or for a class of buildings, simply inserting data for many buildings in the inputs.

The intensity measure to be used is S$_a$ and a mapping between any engineering demand parameter (EDP), assumed to describe the damage state thresholds, and the roof displacement should be available from the pushover analysis.

Ruiz-Garcia and Miranda (2007) study on inelastic displacement demand estimation, Vamvatsikos and Cornell (2006) and Dolsek and Fajfar (2004) work on seismic demand estimation with multilinear static pushover curves, have been integrated in three nonlinear static procedures, C$_R$-based, spo2ida-based and R-*mu*-T-based, within the same script. In this way the user has the chance to select the procedure with the degree of accuracy consistent with the available input and the type of structural analyses performed.

In section 2.1.1 the main information necessary to start the analysis are presented. In sections 2.1.2, 2.1.3 and 2.1.4 the three procedures are explained respectively, from the point of view of the necessary scientific background behind and their step-by-step implementation in the python script.

### 2.1.1 Using the NLS module

To start using the nonlinear static procedure with record-to-record variability a command line text editor should be used to enter manually the folder location where the *rmtk* has been saved, as shown in the example below:

```
~$ cd path/to/rmtk/folder/rmtk/vulnerability
```

Where /path/to/rmtk/folder/ is the system path to the location of the *rmtk* folder. From the text editor iPython browser page can be opened with the following command line:

```
~$ ipython-2.7 notebook --pylab=inline
```

Once the iPython page is opened on the browser, the python scripts contained in the *vulnerability* directory will be visible. The file *NSM_dispersion.ipynb* should be selected to start the calculations.

In the initial section of the script "Define Options" the user should set the options, while the input corresponding to the defined options should be entered in the folder *NSP/input*. The main options to define are the following:

- Type of procedure to perform: either $C_R$-based, spo2ida-based or R-$\mu$-T-based. The main difference between the three is that $C_R$-based procedure is applicable to elasto-plastic idealised capacity curve only, while spo2ida-based and R-$\mu$-T-based procedure fit any kind of multilinear curve. Among the two procedures for multilinear capacity curves the main difference lies in the spo2ida-based ability to compute more accurately the record-to-record dispersion.
- Type of input: either results of a pushover analysis in terms of displacement vs base shear at each time step or idealised pushover curve, as shown in Figures 2.1 and 2.2.

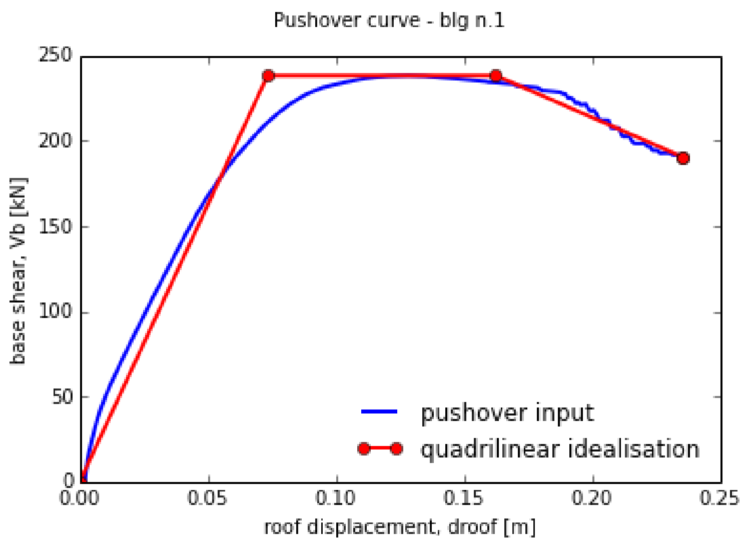

**Figure 2.1** – *Pushover curve.*

- Type of output: either fragility curve (probability of exceedance of a set of limit states vs seismic intensity, as shown in Figure 2.3) or vulnerability curve (loss ratio vs seismic intensity, as shown in Figure 2.4).

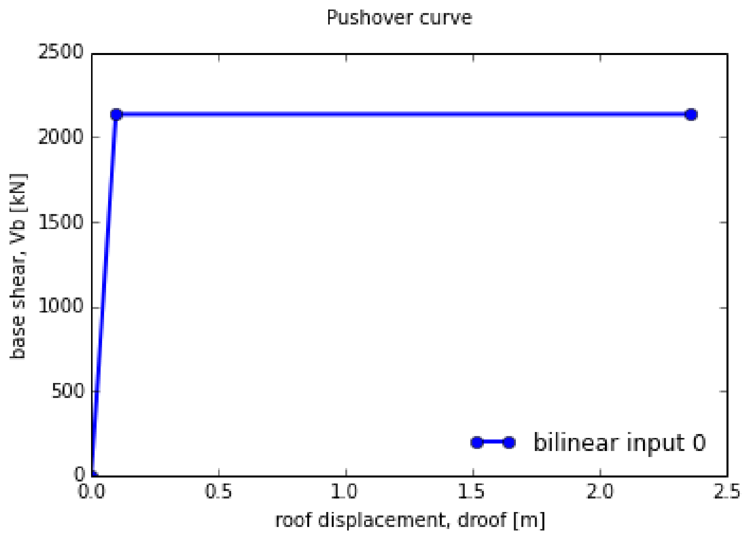

**Figure 2.2** – *Idealised pushover curve.*

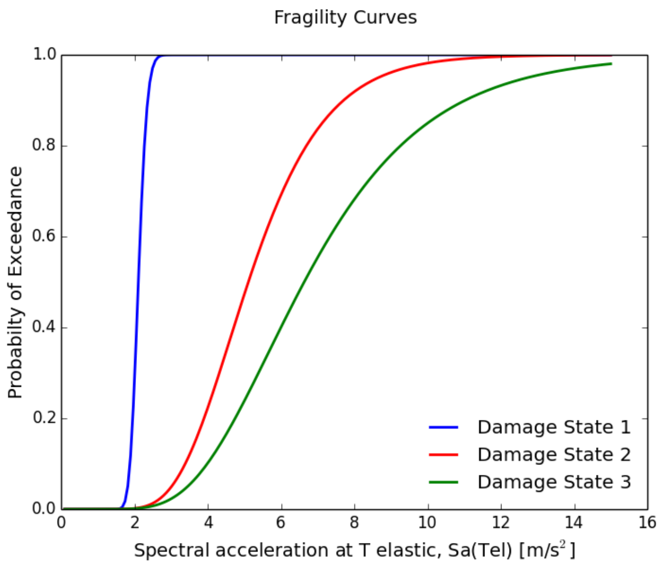

**Figure 2.3** – *Output: fragility curve*

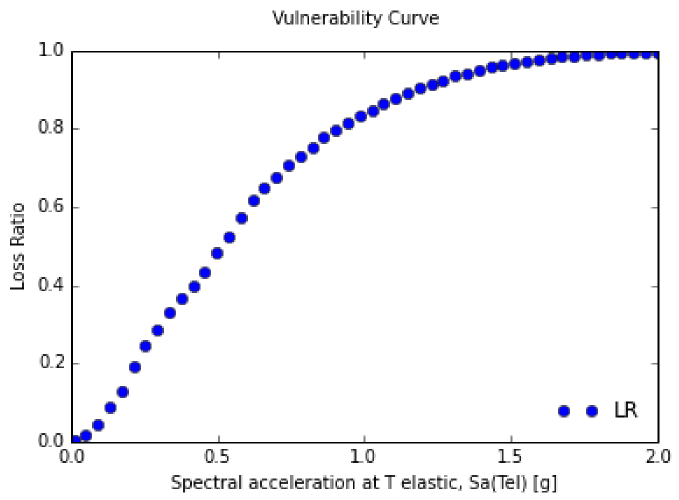

**Figure 2.4** – *Output: vulnerability curve*

The outputs could be found in the *outputs* folder, located in the *vulnerability* directory. The nonlinear static procedure with dispersion produces results in terms of spectral acceleration in units of g. Therefore the mean spectral acceleration of the lognormal fragility curves and the spectral acceleration levels corresponding to each loss ratio in the vulnerability curve are in units of g. These options and others need to be defined in the initial section of the script "Define Options". In section 2.1.1 the alternatives values that the initial variables can assume and their meaning are described in detail, while the parameters to be inserted in the input files are listed below. They are fully described in section 2.1.2 section 2.1.3 and section 2.1.4, within the presentation of each of the three procedures.

## Options

The type of procedure to be performed and the type of inputs at disposal, are set with the variables *an_type* and *in_type* respectively. With the variable *an_type* the user can choose between:

```
an_type = 0 #Cr-based procedure (Ruiz-Garcia and Miranda, 2007)
an_type = 1 #spo2ida-based procedure (Vamvatsikos and Cornell, 2006)
an_type = 2 #R-m-T-based procedure (Dolsek and Fajfar, 2004)
```

With the variable *in_type* the user can choose between:

```
in_type = 0 # idealised pushover curve
in_type = 1 # raw results from a pushover analysis
```

The variable *vuln* instead gives the opportunity to decide the type of outputs, whether to stop the process at the derivation of the fragility curves, or to go all the way up to the vulnerability curve definition, applying damage-to-loss functions.

```
vuln = 0 # derive fragility curves
vuln = 1 # derive vulnerability curve
```

The variable *g* serves the purpose of defining the units that are being used. A floating number must be assigned to the gravity acceleration, compatible with the units used for the period of vibration and for the displacements (if the period is expressed in seconds and displacements are in meters, then g = 9.81). The variable *iml* is a numpy array that identifies the intensity measure levels for which loss ratios are computed and provided in the vulnerability curve. The variable *MC* fix the number of Monte Carlo simulations to account for uncertainty in damage thresholds.

```
g = 9.81
iml = np.linspace(0.1,15,100)
MC = 25
```

The variable *plotflag* allows or inhibits the displaying of plots. It is a python list composed of 4 integers, each one controlling a different plot: idealised pushover curve, 16%-50%-84% ida curves, fragility curves and vulnerability curve respectively. Each integer can take as value either zero or one, whether the corresponding graph has to be displayed or not:

```
plotflag = [1, 1, 1, 1] # plot all the graphs
plotflag = [0, 0, 0, 0] # do not plot any graph
```

The following variables set some of the characteristics of the plots:

- *linew*: integer for defining lines width.
- *fontsize*: fontsize used for labels, graphs etc.
- *units*: list of 3 strings defining displacements, forces and Spectral acceleration units, as ['[kN]', '[m]', '[m/s$^2$]'], to be displayed on the axes of the plots.

The variable *N* is needed for spo2ida-based procedure only and it represents the number of points per segment of IDA curve derived with spo2ida.

The last set of variables is needed for R-$\mu$-T-based procedure only:

- *Tc*: constant accel-constant velocity corner period of a Newmark-Hall type spectrum. Default value is 0.5. *Td*: constant velocity-constant displacement corner period of a Newmark-Hall type spectrum. Default value is 1.8.

### 2.1.2 C$_R$-based procedure (Ruiz-Garcia and Miranda, 2007)

The aim of this procedure, proposed by Vamvatsikos (2014), is the estimation of the median spectral acceleration value $\hat{S}_{a,ds}$, that brings the structure to the attainment of a set of damage states *ds*, and the corresponding dispersion beta $\beta_{S_a}$, the parameters needed for the mathematical representation of fragility in equation 2.1. The aim is achieved making use of the work by Ruiz-Garcia and Miranda (2007), where the inelastic displacement demand is related to the elastic displacement with a simple relationship, and it can thus be easily estimated through a response spectrum analysis and a capacity curve.

The C$_R$-based procedure presented herein is applicable to bilinear elasto-plastic capacity curve only, and it is suitable for single building fragility curve estimation, as described in section 2.1.2. However the fragility curves derived for single buildings can be combined in a unique fragility curve, which considers the inter-building uncertainty, as described in the following sections.

#### Single Building Fragility and Vulnerability function

This procedure provides a simple relationship between median damage state threshold, expressed in terms of top displacement IS THIS DISPLACEMENT OR DRIFT?? $\hat{d}_{roof,ds}$, at each damage state threshold *ds*, and the corresponding median elastic Spectral displacement value $\hat{S}_{d,ds}(T_1)$.

$$\hat{d}_{roof,ds} = C_R \hat{S}_{d,ds}(T_1)\Gamma_1\Phi_1 \tag{2.2}$$

where $\Gamma_1\Phi_1$ is the first mode participation factor estimated for the first-mode shape normalised by the roof displacement, and C$_R$ is the inelastic displacement ratio (inelastic over elastic spectral displacement), computed by Ruiz-Garcia and Miranda (2007) for nonlinear SDoF systems having hysteretic behaviour representative of the analysed structure, which is a function of the first-mode period of vibration and the relative lateral strength of the system R. Therefore the median Spectral acceleration at the fundamental period of vibration $\hat{S}_{a,ds}(T_1)$ turns out to be expressed as a function of the roof displacement according to the following equation:

$$\hat{S}_{a,ds}(T_1) = \frac{4\pi^2}{\hat{C}_R T^2 \Gamma_1 \Phi_1} \hat{d}_{roof,ds} \tag{2.3}$$

Default values of $\hat{C}_R$ parameter estimates are provided by Ruiz-Garcia and Miranda (2007), as result of nonlinear regression analysis of three different measures of central tendency computed from 240 ground motions:

$$\hat{C}_R = 1 + \frac{\hat{R}-1}{79.12 T_1^{1.98}} \tag{2.4}$$

and values for $\hat{R}$ are given as:

$$\hat{R}_{ds} = max(0.425(1-c+\sqrt{c^2+2c(2\hat{\mu}_{ds}-1)+1}),1) \tag{2.5}$$

where c = 79.12 T$^{1.98}$, and $\hat{\mu}_{ds}$ is the median ductility level at the damage state threshold of interest.

For what concerns $\beta_{S_a}$, the dispersion of $\hat{S}_{a,ds}$, it can be computed either in a simplified way or with a Monte Carlo sampling procedure, only if the dispersion due to uncertainty in the limit state definition $\beta_{\theta c}$ is different from zero.

In the simplified approach the following relationship between $S_a$ and the median EDP damage threshold $\hat{\theta}$ (Cornell, 2002) is used:

$$\hat{\theta} = a S_a^b \tag{2.6}$$

so that $\beta_{S_a}$ can be easily derived from the dispersion of $\theta$ due to record-to-record variability, $\beta_{\theta d}$, as in the following:

$$\beta_{S_a} = \frac{1}{b}\beta_{\theta d} \tag{2.7}$$

$\beta_{\theta d}$ can be obtained assuming that top drift $d_{roof}$ and $\theta$ are proportional, and they thus share the same dispersion. Moreover the dispersion of $d_{roof}$ is the same as the dispersion of C$_R$, since they are also proportional. Finally $\beta_{\theta d}$ can be computed with the following equation, which represents Ruiz-Garcia and Miranda's (2007) estimate of C$_R$ dispersion:

$$\sigma_{\ln(C_R)} = \sigma_{\ln(d_{roof})} = \beta_{\theta d} = 1.975[\frac{1}{5.876} + \frac{1}{11.749(T+0.1)}][1-\exp(-0.739(R-1))] \tag{2.8}$$

Uncertainty in the damage state can also be accounted for combining the dispersion of $\theta$ due to uncertainty in the damage state with the dispersion due to record-to-record variability as in the following equation:

$$\beta_{S_a} = \frac{1}{b}\sqrt{\beta_{\theta d}^2 + \beta_{\theta c}^2} \tag{2.9}$$

In the Monte Carlo approach first of all three polylines corresponding to the $16^{th}$, $50^{th}$ and $84^{th}$ fractiles of R need to be drawn. This is done by computing $\beta_{\theta d}$ for each median $\hat{\mu}_{ds}$ with eq. 2.8, and obtaining from this value the $16^{th}$ and $84^{th}$ fractiles of $\mu_{ds}$, $\mu_{16\%}$ and $\mu_{84\%}$ according to the following equations:

$$\mu_{ds,16} = \hat{\mu}_{ds} e^{-\beta_{\theta d,ds}} \tag{2.10}$$

$$\mu_{ds,16} = \hat{\mu}_{ds} e^{\beta_{\theta d,ds}} \tag{2.11}$$

The median $\hat{R}_{ds}$ values have been already found from eq. 2.5, and $mu_{16\%} - \hat{R}$, $\hat{\mu} - \hat{R}$ and $\mu_{84\%} - \hat{R}$ curves can be drawn. Different values of ductility limit state are now sampled from the lognormal distribution with median the median value of the ductility limit state, and dispersion the input $\beta_{\theta c}$. For each of these ductilities the corresponding $\hat{R}$,$R_{16\%}$, and $R_{84\%}$ values are found interpolating the aforementioned curves, and converted into $\hat{S}_{a,ds}$ and $\beta_{\theta d}$ according to the following equations:

$$\hat{S}_{a,ds} = \hat{R}(\mu_{ds}) S_{ay} \tag{2.12}$$

$$\beta_{R(\mu)} = \frac{\ln R(\mu)_{84\%} - \ln R(\mu)_{16\%}}{2} \tag{2.13}$$

where

$$S_{ay} = \frac{4\pi^2 d_{roof,y}}{T_1^2 g \Gamma_1} \tag{2.14}$$

N random S$_a$ for each of the N sampled ductility limit states are computed using $\hat{S}_{a,ds}$ and $\beta_{\theta d}$, and their median and the dispersion are estimated. These parameters constitute the median $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ for the considered damage state. The procedure is repeated for each damage state.

To derive a discrete vulnerability function at certain intensity measure levels, the input damage-to-loss factors are applied to the probability of occurance of each damage state, extracted from the probability of exceedance of each damage state described by the fragility function. A value of loss ratio is thus defined for the vector of selected intensity measure levels.

### Multiple-Building Fragility and Vulnerability function

If multiple buildings have been input to derive fragility function for a class of buildings all $\hat{S}_{a,blg}$ and $\beta_{S_{a,blg}}$ are combined in a single lognormal curve. A minimum of 5 buildings should be considered to obtain reliable results for the class. A new issue arises when multiple buildings are considered: the S$_a$ at the fundamental period of each building should be converted to a common intensity measure, to be able to combine the different fragility functions. A common intensity measure is selected to be S$_a$ at the period T$_{av}$, which is a weighted average of the individual buildings fundamental period T$_1$. Then each individual fragility needs to be expressed in terms of the common S$_a$(T$_{av}$), using a spectrum. FEMA P-695 far field set of 44 accelerograms (22 records for the two directions) was used to derive a mean uniform hazard spectrum, and the ratio between the S$_a$ at different periods is used to scale the fragility functions. It can be noted that the actual values of the spectrum are not important, but just the spectral shape. The median $\hat{S}_a$ is converted to the mean $\mu_{ln(S_a)}$ of the corresponding normal distribution ($\mu_{ln(S_a)} = ln(\hat{S}_a)$) and, simply scaled to the common intensity measure as follows:

$$\mu_{ln(S_a),blg} = \mu_{ln(S_a),blg} S(T_{av})/S(T_{1,blg}) \tag{2.15}$$

$$\beta_{S_a,blg} = \beta_{S_a,blg} S(T_{av})/S(T_{1,blg}) \tag{2.16}$$

Finally the parameters of the single lognormal curve for the class of buildings, mean and dispersion, can be computed as the weighted mean of the single means and the weighted SRSS of the inter-building and intra-building standard deviation, the standard deviation of the single means and the single dispersions respectively, as shown in the following equations:

$$\mu_{ln(S_a),tot} = \sum_{i=0}^{n.blg} w_{blg-i} \mu_{ln(S_a),blg-i} \tag{2.17}$$

$$\beta_{S_a,tot} = \sqrt{\sum_{i=0}^{n.blg} w_{blg-i}((\mu_{ln(S_a),blg-i} - \mu_{ln(S_a),tot})^2 + \beta^2_{S_a,blg-i})} \tag{2.18}$$

The mean $\mu_{ln(S_a)}$ and total dispersion $\beta_{S_a}$ of the fragility function of the class are converted to logarithmic mean $\mu_{S_a}$ and logarithmic covariance $cov_{S_a}$ (standard deviation $\sigma_{S_a}$ over $\mu_{S_a}$), according to the following equations:

$$\hat{S}_a = e^{\mu_{ln(S_a)}} \tag{2.19}$$

$$\mu_{S_a} = \hat{S}_a e^{\frac{\beta^2_{S_a}}{2}} \tag{2.20}$$

$$\sigma_{S_a} = \sqrt[2]{(\beta^2_{S_a} - 1) e^{2 \ln \hat{S}_a + \beta^2_{S_a}}} \tag{2.21}$$

$$cov_{S_a} = \frac{\sigma_{S_a}}{\mu_{S_a}} \tag{2.22}$$

A single vulnerability function can be also obtained, from the single building vulnerability functions. The input damage-to-loss function is applied to the fragility function derived for each building. For the selected intensity measure levels a value of loss ratio $LR_{blg}$ is thus defined for each building. A discrete vulnerability function for the entire class of buildings is represented at each iml by a mean LR, $\mu_{LR,tot}$, equal tot the weighted $LR_{blg}$, and a standard deviation, $\sigma_{LR,tot}$, equal to the weighted standard deviation of all the computed $LR_{blg}$. The $\sigma_{LR,tot}$ of the fragility function of the class is converted to covariance $cov_{LR}$ (standard deviation $\sigma_{LR,tot}$ over $\mu_{LR,tot}$).

### Inputs

The inputs must be formatted as comma-separated value files (.csv), and saved in the folder *input*, contained in the NSP directory. If any other environment different from Windows is used make sure that the "comma separated values Windows" is selected as saving option when creating the input files.

If multiple buildings want to be analysed to consider the inter-building uncertainty the parameters relative to each building should be added as additional lines in the input tables, as shown in the examples below, otherwise a single line must be input.

If the user has already at disposal an idealised elasto-plastic pushover curve for each building, that is to say that the variable *in_type* has been set to 0, the following data need to be provided in the corresponding csv files:

1. First period of vibration $T_1$, corresponding modal participation factor $\Gamma_1$, normalised with respect to the roof displacement, and weight for the combination of different buildings, input in *building_parameters.csv*, as in the example below:

| **n.building** | $\mathbf{T_1}$ | $\Gamma_1$ | **weights** |
|---|---|---|---|
| 1 | 0.32 | 1.23 | 0.2 |
| 2 | 0.40 | 1.25 | 0.3 |
| ... | ... | ... | ... |

2. Roof displacement at each limit state LS and corresponding dispersion $\beta_{\theta c}$ input in *displacement_profile.csv*, as shown in the example below. If dispersion is unknown, $\beta_{\theta c}$ can be set equal to zero at each LS.

| **n.building** | $\mathbf{LS_1}$ | $\mathbf{LS_2}$ | $\mathbf{LS_3}$ |
|---|---|---|---|
| 1 | 0.066 | 0.169 | 0.23 |
| $\beta_{\theta d,1}$ | 0.1 | 0.3 | 0.4 |
| 2 | 0.08 | 0.172 | 0.25 |
| $\beta_{\theta d,2}$ | 0.1 | 0.3 | 0.4 |

3. Idealised pushover curve, input in *idealised_curve.csv* as shown below. The only required parameters are the yielding displacement $d_y$, the ultimate displacement $d_u$ and the yielding force $F_y$.

| **n.building** | $\mathbf{d_y}$ | $\mathbf{d_u}$ | $\mathbf{F_y}$ |
|---|---|---|---|
| 1 | 0.09 | 0.3 | 523 |
| 2 | 0.12 | 0.35 | 400 |
| ... | ... | ... | ... |

4. Consequence model (loss ratio per each damage state) consistent with the defined set of damage states, input in *consequence.csv*, as in the example below. A single consequence model can be input. This input is needed only if the variable *vuln* has been set to 1.

| $\mathbf{DS_1}$ | $\mathbf{DS_2}$ | $\mathbf{DS_3}$ |
|---|---|---|
| 0.2 | 0.5 | 1 |

If the idealised curve are not available, *in_type* = 0 can be selected and the displacements vs base shear at each time step results from a pushover analysis can be input instead. The following data need to be provided in the corresponding csv files:

1. $T_1$ and corresponding $\Gamma_1$, weight for the combination of different buildings, number of storeys and height of each storey, input in *building_parameters.csv*, as in the example below:

| **n.building** | **$T_1$** | **$\Gamma_1$** | **weights** | **n.Storey** | **$H_1$** | **$H_2$** | **...** | **$H_n$** |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.32 | 1.23 | 0.2 | 4 | 3 | 3 | ... | 3 |
| 2 | 0.40 | 1.25 | 0.3 | 4 | 4 | 2.7 | ... | 2.7 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

2. Displacements at each storey, at each incremental step of the pushover analysis, input in *displacements_pushover.csv*, as in the example below:

| **n.building** | **n.Storey** | **Step1** | **Step 2** | **Storey 3** | **...** | **Step n** |
|---|---|---|---|---|---|---|
| 1 | 1 | 0.0001 | 0.0005 | 0.001 | ... | 0.01 |
|  | 2 | 0.0003 | 0.0010 | 0.002 | ... | 0.02 |
|  | 3 | 0.0004 | 0.0016 | 0.003 | ... | 0.03 |
|  | 4 | 0.0006 | 0.0021 | 0.004 | ... | 0.04 |
| 2 | 1 | 0.0001 | 0.0005 | 0.001 | ... | 0.01 |
|  | 2 | 0.0005 | 0.0012 | 0.002 | ... | 0.03 |
|  | ... | ... | ... | ... | ... | ... |

3. Base shear at each incremental step of the pushover analysis input in *reactions_pushover.csv*, as in the example below:

| **n.building** | **Step1** | **Step 2** | **Storey 3** | **...** | **Step n** |
|---|---|---|---|---|---|
| 1 | 0.35 | 0.69 | 1.04 | ... | 29.12 |
| 2 | 0.45 | 0.78 | 2.05 | ... | 40.00 |
| ... | ... | ... | ... | ... | ... |

4. Drift limit state and corresponding dispersion $\beta_{\theta c}$ input in *limits.csv*. If dispersion is unknown, $\beta_{\theta c}$ can be set equal to zero at each limit state.

| **n.building** | **$LS_1$** | **$LS_2$** | **$LS_3$** |
|---|---|---|---|
| 1 | 0.01 | 0.025 | 0.0337 |
| $\beta_{\theta d,1}$ | 0.1 | 0.2 | 0.25 |
| 2 | 0.014 | 0.030 | 0.0430 |
| $\beta_{\theta d,2}$ | 0.1 | 0.2 | 0.25 |

5. Consequence model (loss ratio per each damage state) consistent with the defined set of damage states, input in *consequence.csv*, as in the example below. A single consequence model can be input. This input is needed only if the variable *vuln* has been set to 1.

| **$DS_1$** | **$DS_2$** | **$DS_3$** |
|---|---|---|
| 0.2 | 0.5 | 1 |

### Calculation Steps

The overall workflow of $C_R$-based procedure is summarised in this section. The option *an_type* must be set equal to 0 and the option *in_type* according to the input at disposal. The corresponding inputs should follow the requirements described in section 2.1.2. At this point the code proceeds with the following steps:

1. (a) If *in_type* = 0 roof displacement at limit states and idealised pushover are extracted from *displacement_profile.csv* and *idealised_curve.csv* respectively.
   (b) If *in_type* = 1 results from pushover analysis are extracted from *displacements_pushover.csv* and *reactions_pushover.csv*, and drift limit states from *limits.csv*. The idealised pushover curve is then derived in the *idealisation* function, where the idealisation process is conducted according to FEMA-440. The elastic stiffness is defined as the tangent stiffness passing through the point of the pushover curve where 60% of the maximum base shear is reached, and the perfectly plastic branch is set at an height equal to the maximum base shear. The yielding point is found as the interception between the elastic and the plastic branch.
2. The csv input files are parsed with the function *read_data* according to the defined options. The parameters essential to the analysis are return together with a graphical visualisation of the inputs if the variable *plotflag*[0] is equal to 1.
3. The parameters extracted are used in the *simplified_bilinear* function, within the *fragility_process* function, to derive ductility levels $\mu_{ds}$, median spectral acceleration $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ at each limit state through the following steps:
   - The idealised MDoF system is transformed into an equivalent SDoF system, using $\Gamma_1$.
   - Ductility levels $\mu_{ds}$ corresponding to each damage threshold, are defined.
   - R and C$_R$ are computed, using eq. 2.5 and 2.4 respectively.
   - $\hat{S}_{a,ds}$ and the corresponding dispersion $\beta_{\theta d}$ are computed using eq. 2.3 and 2.7 respectively.
   - If dispersion due to uncertainty in the limit state $\beta_{\theta c}$ is different from zero different ductility limit states are sampled for each median ductility level $\mu_{ds}$ and corresponding values of $\hat{S}_{a,ds}$ and $\beta_{S_{a,d}}$ are computed, as described in section 2.1.2, but not yet combined together..
4. All $\hat{S}_{a,ds}(T_1)$ are converted to mean $\mu_{ln(S_{a,ds})}(T_1)$ and then to the intensity measure in common with the rest of the buildings, $\mu_{ln(S_{a,ds}(T_{av}))}$, according to eq. 2.16.
5. Step 3 and 4. are repeated for the number of input buildings.
6. (a) If *vuln* = 0: All $\hat{S}_{a,ds}$ and $\beta_{S_{a,d}}$ from all the buildings and all the sampled ductility limit states are combined in a single lognormal curve, as described in section 2.1.3. Mean $\mu_{ln(S_a)}$ and total dispersion $\beta_{S_a}$ are then converted to logarithmic mean $\mu_{S_a}$ and logarithmic covariance $cov_{S_a}$, according to equations 2.20 and 2.21 respectively. Fragility curves for the class of buildings are displayed if the variable *plotflag*[2] = 1, and logarithmic $\mu_{S_a}$ and $cov_{S_a}$ are exported in the *outputs* folder.
   (b) If *vuln* =1: For the intensity measure levels defined in the variable *iml* a value of loss ratio $\mu_{LR,iml,blg}$ is defined for each building and a standard deviation $\sigma_{LR,iml,blg}$, if dispersion due to uncertainty in the limit state $\beta_{\theta c}$ is different from zero. They are finally combined in a single mean and standard deviationas described in section 2.1.3. Vulnerability curve for the class of buildings is displayed if the variable *plotflag*[3] = 1, and $\mu_{LR}$ and $cov_{LR}$ at each iml are exported in the *outputs* folder.

### 2.1.3 spo2ida-based procedure (Vamvatsikos and Cornell, 2006)

The aim of this procedure is the estimation of the median spectral acceleration value $\hat{S}_{a,ds}$, that brings the structure to the attainment of a set of damage states ds, and the corresponding dispersion beta $\beta_{S_a}$, the parameters needed for the mathematical representation of fragility in equation 2.1. The aim is achieved making use of the tool spo2ida (Vamvatsikos and Cornell, 2006), where static pushover curves are converted into 16%, 50% and 84% ida curves, using empirical relationships from a large database of incremental dynamic analysis results, as shown in Figure 2.5.

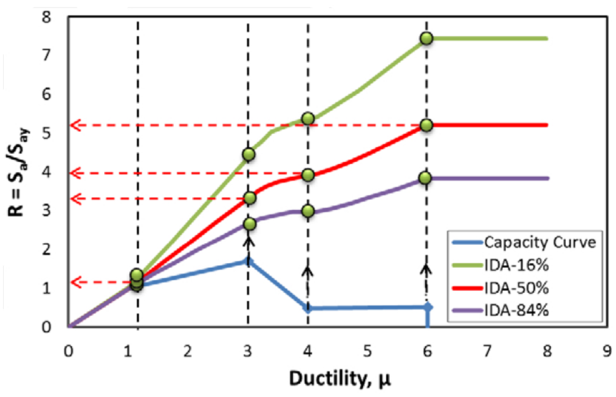

**Figure 2.5** – *spo2ida tool: IDA curves derived from Pushover curve.*

The spo2ida-based procedure presented herein is applicable to any kind of multi-linear capacity curve, and it is suitable for single building fragility curve estimation, as described in section 2.1.3. However the fragility curves derived for single buildings can be combined in a unique fragility curve, which considers the inter-building uncertainty, as described in section 2.1.3.

### Single-building Fragility and Vulnerability function

Given the idealised capacity curve the spo2ida tool uses an implicit R-$\mu$-T relation to correlate nonlinear displacement, expressed in terms of ductility $\mu$ to the corresponding medians capacity in terms of the parameters R. R is the lateral strength ratio, defined as the ratio between the spectral spectral acceleration S$_a$ and the yielding capacity of the system S$_{ay}$.

Each branch of the capacity curve, hardening, softening and residual plateau, is converted to a corresponding branch of the three ida curves, using the R-$\mu$-T relation, which is a function of the hardening stiffness, the softening stiffness and the residual force. These parameters are derived from the idealised pushover capacity expressed in $\mu$-R terms, as well as the ductility levels at the onset of each branch. If some of the branches of the pushover curve are missing because of the seismic behaviour of the system, spo2ida can equally work with bilinear, trilinear and quadrilinear idealisations.

The result of the spo2ida routine is thus a list of ductility levels and corresponding R values at 50%, 16% and 84% percentiles. The distribution of R values at each ductility level, due to the record-to-record variability, is assumed to be lognormal and it can be easily converted to the dispersion of the lognormal distribution with the eq. 2.13.
$\beta_{R(\mu)}$ represents also the record-to-record variability of S$_a$ at different ductility levels $\beta_{S_a,d}$. Median R and its dispersion at ductility levels corresponding to the damage thresholds can thus be determined, and $\hat{S}_{a,ds}$ can be easily extracted simply multiplying $\hat{R}(\mu_{ds})$ by the yielding capacity of the system $S_{ay}$, as shown in eq. 2.12 and eq. 2.14. Since $\hat{R}$ and $\hat{S}_a$ are proportional they share the same dispersion.

If dispersion due to uncertainty in the limit state definition $\beta_{\theta c}$ is different from zero it can not be combined directly with the record-to-record dispersion, but a Monte Carlo sampling of the limit state needs to be performed instead. Different values of ductility limit state are sampled from the lognormal distribution with median the median value of the ductility limit state, and dispersion the input $\beta_{\theta c}$. For each of these ductilities the corresponding median $\hat{R}$ and $R_{16\%}$, $R_{84\%}$ are found and converted into $\hat{S}_{a,ds}$ and $\beta_{\theta d}$ according to equation 2.12 and 2.13. N random $S_a$ for each the N sampled ductility limit states are computed, and their median and the dispersion are estimated. These parameters constitute the median $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ for the considered damage state. The procedure is repeated for each damage state.

To derive a discrete vulnerability function at certain intensity measure levels, the input damage-to-loss factors are applied to the probability of occurance of each damage state, extracted from the probability of exceedance of each damage state described by the set of fragility curves.

If dispersion due to uncertainty in the limit state is different from zero a vulnerability function is derived for the N sets of sampled ductility limit states. It results in N loss ratios for each defined intensity measure levels. Finally a lognormal distribution of the loss ratios is assumed at each iml and the vulnerability curve is defined at each iml by the mean and the standard deviation of all the computed loss ratios.

### Multiple-Building Fragility and Vulnerability function

If multiple buildings have been input to derive a set of fragility curves for a class of buildings all $\hat{S}_{a,blg}$ and $\beta_{S_a,blg}$ are combined in a single lognormal curve for each damage state. A minimum of 5 buildings should be considered to obtain reliable results for the class. The procedure to get $\mu_{S_a,tot}$ and $cov_{S_a,tot}$ for the class of building is the same described in section 2.1.2, but the $\hat{S}_{a,blg}$ and $\beta_{S_a,blg}$ are those derived from each sampled set of ductility limit state.

A single vulnerability curve can also be obtained, from the single building vulnerability curves. If no dispersion in the limit state is defined, the method is the same described in section 2.1.2. Otherwise a vulnerability curve is derived for each building as explained in section **??**, considering the sampled set of ductility limit states, that is to say that the mean loss ratio and its standard deviation at each iml, $\mu_{LR,iml,blg}$ and $\sigma_{LR,iml,blg}$ respectively, are found for each building. Finally the mean loss ratio and its standard deviation, $\mu_{LR,iml}$ and $\sigma_{LR,iml}$, are found for the entire class of buildings as the weighted mean of the single $\mu_{LR,iml,blg}$ and the weighted SRSS of the inter-building and intra-building standard deviation, the standard deviation of the single means $\mu_{LR,iml,blg}$ and the single dispersions $\sigma_{LR,iml,blg}$ respectively, as described in eq 2.18, substituting loss ratio to spectral acceleration.

### Inputs

The inputs must be formatted as comma-separated value files (.csv), and saved in the folder *input*, contained in the NSP directory. If any other environment different from Windows is used make sure that the "comma separated values Windows" is selected as saving option when creating the input files.

If multiple buildings want to be analysed to consider the inter-building uncertainty the parameters relative to each building should be added as additional lines in the tables, as shown in the examples below, otherwise a single line must be input.

If the user has already at disposal an idealised multilinear pushover curve for each building, that is to say that the variable *in_type* has been set to 0, the following data need to be provided in the corresponding csv files:

1. First period of vibration $T_1$, corresponding modal participation factor $\Gamma_1$, normalised with respect to the roof displacement, and weight for the combination of different buildings, input in *building_parameters.csv*, as in section 2.1.2, input n. 1.
2. Top displacement at each damage state threshold and corresponding dispersion $\beta_{\theta c}$ input in *displacement_profile.csv*, as in section 2.1.2, input n. 2.

3. Idealised pushover curve, input in *idealised_curve.csv* as shown below. The parameters needed to describe the idealised pushover curve are: yielding displacement $d_y$, displacement at the onset of degradation $d_s$, displacement at the onset of residual force $d_{min}$, ultimate displacement $d_u$, maximum force $F_{max}$, residual force $F_{min}$. These parameters are represented in Figure 2.6.

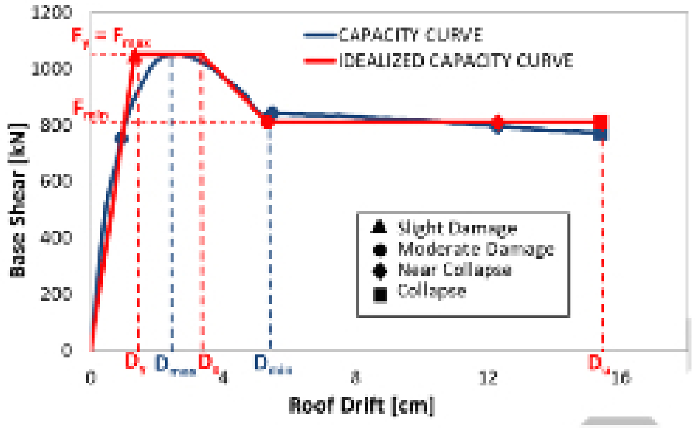

**Figure 2.6** – *Idealisation of capacity curve using multilinear elasto-plastic form.*

| n.building | $d_y$ | $d_s$ | $d_{min}$ | $d_u$ | $F_{max}$ | $F_{min}$ |
|---|---|---|---|---|---|---|
| 1 | 0.09 | 0.3 | 0.4 | 0.5 | 523 | 430 |
| 2 | 0.12 | 0.35 | 0.43 | 0.5 | 400 | 305 |

4. Consequence model (loss ratio per each damage state) consistent with the defined set of damage states, input in *consequence.csv*, as in section 2.1.2, input n. 5.

If these data are not available, *in_type* = 0 can be selected and the "raw" results from a pushover analysis can be input instead. The same data as in section 2.1.2 for *in_type* = 0 can be input.

### Calculation Steps

The overall workflow of spo2ida-based procedure is summarised in this section. The option *an_type* must be set equal to 1 and the option *in_type* according to the input at disposal. The corresponding inputs should follow the requirements described in section 2.1.3. At this point the code proceeds with the following steps:

1. (a) If *in_type* = 0, the roof displacement at each limit state and the idealised pushover curve parameters are extracted from *displacement_profile.csv* and *idealised_curve.csv* respectively.
   (b) If *in_type* = 1 the results from a pushover analysis are extracted from *displacements_pushover.csv* and *reactions_pushover.csv* and drift limit states from limits.csv. The idealised pushover curve is then derived in the *idealisation* function, where the idealisation process is conducted according to the Gem Analytical Vulnerability Guidelines.
2. The parameters extracted are used to derive ductility levels $\mu_{ds}$, median spectral acceleration $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ at each damage threshold through the following steps:
   - The idealised MDoF system is transformed into an equivalent SDoF system, using $\Gamma_1$, and SDoF capacity curve is expressed in terms of $\mu$-R.

- The variables for spo2ida tool are extracted from the capacity curve and they are used as input to get the 16%-50%-84% ida curves.
- The ductility levels $\mu_{ds}$ corresponding to each damage threshold are defined, and the corresponding $R_{16\%}$-$R_{50\%}$-$R_{84\%}$ are found in ida outputs.
- $\hat{S}_{a,ds}$ and the corresponding dispersion $\beta_{S_{a,d}}$ are computed using eq. 2.12 and eq. 2.13, respectively.
- If dispersion due to uncertainty in the limit state $\beta_{\theta c}$ is different from zero different ductility limit states are sampled for each median ductility level $\mu_{ds}$ and corresponding values of $\hat{S}_{a,ds}$ and $\beta_{S_{a,d}}$ are computed, as described in section 2.1.3, but not yet combined together.

3. All $\hat{S}_{a,ds}(T_1)$ are converted to mean $\mu_{ln(S_{a,ds})}(T_1)$ and then to the intensity measure in common with the rest of the buildings, $\mu_{ln(S_{a,ds}(T_{av}))}$, according to eq. 2.16.
4. Step 2. and 3. are repeated for the number of input buildings.
5. (a) If *vuln* = 0: All $\hat{S}_{a,ds}$ and $\beta_{S_{a,d}}$ from all the buildings and all the sampled ductility limit states are combined in a single lognormal curve, as described in section 2.1.3. Mean $\mu_{ln(S_a)}$ and total dispersion $\beta_{S_a}$ are then converted to logarithmic mean $\mu_{S_a}$ and logarithmic covariance $cov_{S_a}$, according to equations 2.20 and 2.21 respectively. Fragility curves for the class of buildings are displayed if the variable *plotflag*[2] = 1, and logarithmic $\mu_{S_a}$ and $cov_{S_a}$ are exported in the *outputs* folder.
   (b) If *vuln* =1: For the intensity measure levels defined in the variable *iml* a value of loss ratio $\mu_{LR,iml,blg}$ is defined for each building and a standard deviation $\sigma_{LR,iml,blg}$, if dispersion due to uncertainty in the limit state $\beta_{\theta c}$ is different from zero. They are finally combined in a single mean and standard deviationas described in section 2.1.3. Vulnerability curve for the class of buildings is displayed if the variable *plotflag*[3] = 1, and $\mu_{LR}$ and $cov_{LR}$ at each iml are exported in the *outputs* folder.

### 2.1.4 $R-mu-T$-based procedure (Dolsek and Fajfar, 2004)

The aim of this procedure is the estimation of the median spectral acceleration value $\hat{S}_a$, that brings the structure to the attainment of a set of damage states, and the corresponding dispersion beta $\beta_{S_a}$, the parameters needed for the mathematical representation of fragility in equation 2.1. The aim is achieved making use of a R-$\mu$-T relationship, between the reduction factor R, the ductility $\mu$ and period T, which is based on the work of Dolsek and Fajfar (2004). The R-$\mu$-T-based procedure presented herein is applicable to any kind of multi-linear capacity curve, and it is suitable for single building fragility curve estimation, as described in section 2.1.4. However the fragility curves derived for single buildings can be combined in a unique fragility curve, which considers the inter-building uncertainty, as described in section 2.1.4.

#### Single Building Fragility and Vulnerability function

The spectral value at each damage state threshold *ds* $\hat{S}_{a,ds}$ is found from the roof displacement limit state for that *ds* $\hat{d}_{roof,ds}$, as explained in $C_R$_based procedure and reported by the following equation:

$$\hat{S}_{a,ds}(T_1) = \frac{4\pi^2}{\hat{C}_R T^2 \Gamma_1 \Phi_1} \hat{d}_{roof,ds} \tag{2.23}$$

The value of $C_R$, the ratio between the inelastic and the elastic spectral displacement, is found from the equation:

$$\hat{C}_R = \frac{\mu_{ds}}{R_{ds}} \tag{2.24}$$

where $\mu_{ds}$ and $R_{ds}$ are the ductility level and the reduction factor at the attainment of the *ds*. According to the results of an extensive parametric study using three different sets of recorded and semi-artificial ground motions Dolsek and Fajfar (2004) related the ductility demand, $\mu$ , and reduction factor, R , through the following formula:

$$\mu = \frac{1}{c}(R - R_0) + \mu_0 \tag{2.25}$$

In the proposed model, $\mu$ is linearly dependent on R within two reduction factor intervals. The parameter c defines the slope of the R–$\mu$ relation, and depends on the initial period of the system T, the ratio $r_u$, the reduction factor R and the corner periods $T_c$ and $T_d$. $T_c$ and $T_d$ are the corner periods between the constant acceleration and constant velocity part of the idealized elastic spectrum, and between the constant velocity and constant displacement part of the idealized elastic spectrum respectively.

Given the parameters of the multilinear pushover curves ($R_{\mu_c}$, $\mu_c$, $r_u$) and T, the median R-$\mu$ curve, similar to an ida curve, can be construct using the aforementioned relationship. A multilinear capacity curve and the corresponding $R_{\mu_c}$, $\mu_c$ and $r_u$ parameters are shown in Figure 2.7.

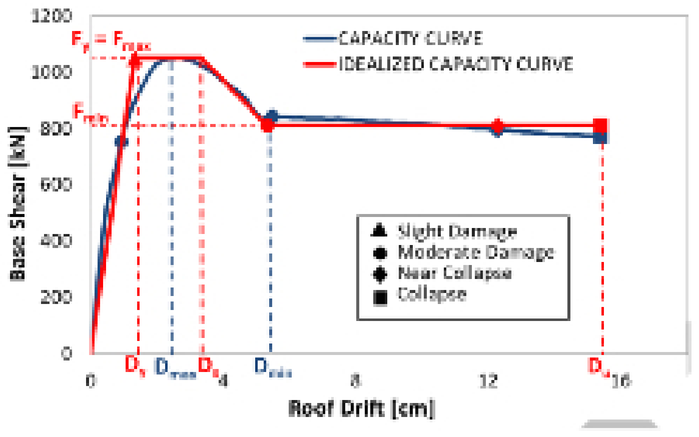

**Figure 2.7** – *Multilinear capacity curve and parameters for the definition of R-μ relation.*

On the median "ida curve" the $\mu$-R pairs corresponding to the limit states ($R_{ds}$ and $\mu_{ds}$) can be found and turned into median spectral acceleration values for that limit state $\hat{S}_a$, to be used in equation 2.23.

Once median $\hat{R}_{ds}$ and $\hat{\mu}_{ds}$ are found, the 84th and 16th fractiles of $\mu$ are extracted using top displacement record-to-record dispersion $\sigma_{ln(d_{roof})}$ from equation 2.8, by Ruiz-Garcia and Miranda (2007). The steps to derive $\mu_{16\%}$ and $\mu_{84\%}$ are the following:

$$ln(d_{roof})_{16\%} = ln(\hat{d_{roof}}) - \sigma_{ln(d_{roof})} \tag{2.26}$$

$$ln(d_{roof})_{84\%} = ln(\hat{d_{roof}}) + \sigma_{ln(d_{roof})} \tag{2.27}$$

$$\mu_{16\%} = \hat{\mu}\exp - \sigma_{ln(d_{roof})} \tag{2.28}$$

$$\mu_{84\%} = \hat{\mu} \exp \sigma_{ln(d_{roof})} \tag{2.29}$$

Given the linear relationship between R and $\mu$, the R$_{16\%}$ and R$_{84\%}$ values at $\mu_{ds}$ are just linearly interpolated, and the record-to-record dispersion of the spectral acceleration $\beta_{S_{a,d}}$ at each $\mu_{ds}$ coincides with the dispersion of R, computed from the percentiles values, as in the following equation:

$$\beta_{S_{a,d}} = \beta_{R(\mu)} = \frac{\ln R(\mu)_{84\%} - \ln R(\mu)_{16\%}}{2} \tag{2.30}$$

If dispersion due to uncertainty in the limit state definition $\beta_{\theta c}$ is different from zero it can be combined with the record-to-record dispersion $\beta_{\theta d}$ either in a simplified way or with a Monte Carlo sampling procedure similar to what is done in section 2.1.2.

In the simplified method the dispersion of $S_a$ due to uncertainty in the damage state threshold $\beta_{S_{a,c}}$ can be found converting the dispersion on the damage threshold $\beta_{\theta c}$, as explaind in section 2.1.2 and reported in the following equation:

$$\beta_{S_{ac}} = \frac{1}{b}\beta_{\theta c} \tag{2.31}$$

In order to derive the b values, which represent the slope of the R-$\mu$ relation in the log-space, a further step needs to be made, because the R-$\mu$-T is suggested by the authors as conservative, since it is not based on the median but on the mean $\mu$ given R. An attempt was made to correct it reducing the median R curve by 15%, $\hat{R}_{corrected}$, as shown in equation 2.32, and extrapolating the corresponding $\hat{\mu}_{corrected}$.

$$\hat{R}_{corrected} = 0.85\hat{R} \tag{2.32}$$

$$b = \frac{ln(\hat{\mu}_{corrected})}{ln(\hat{R}_{corrected})} \tag{2.33}$$

Finally the dispersion of $S_a$ due to record-to-record variability, can be combined with the dispersion of $S_a$ due to uncertainty in the damage state threshold, as in the following equation:

$$\beta_{S_a} = \sqrt{\beta^2_{S_{a,c}} + \beta^2_{S_{a,d}}} \tag{2.34}$$

In the Monte Carlo approach different values of ductility limit state are now sampled from the lognormal distribution with median the median value of the ductility limit state, and dispersion the input $\beta_{\theta c}$. For each of these ductilities the corresponding $\hat{R}$, $R_{16\%}$, and $R_{84\%}$ values are found interpolating the aforementioned ida curves, and converted into $\hat{S}_{a,ds}$ and $\beta_{\theta d}$ according to the equations 2.12 and 2.13. N random S$_a$ for each of the N sampled ductility limit states are computed using $\hat{S}_{a,ds}$ and $\beta_{\theta d}$, and their median and the dispersion are estimated. These parameters constitute the median $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ for the considered damage state. The procedure is repeated for each damage state.

To derive a discrete vulnerability function at certain intensity measure levels, the input damage-to-loss factors are applied to the probability of occurance of each damage state, extracted from the probability of exceedance of each damage state described by the fragility function. A value of loss ratio is thus defined for the vector of selected intensity measure levels.

### Multiple Building Fragility and Vulnerability function

If multiple buildings have been input to derive fragility function for a class of buildings all $\hat{S}_{a,blg}$ and $\beta_{S_a,blg}$ are combined in a single lognormal curve as described in section 2.1.2. The same holds for vulnerability function, as described in the same section.

### Inputs

The data the user needs provided and the their format is described in section 2.1.3

### Calculation Steps to be changed

The overall workflow of R-$\mu$-T-based procedure is summarised in this section. The option *an_type* must be set equal to 2 and the option *in_type* according to the input at disposal. The corresponding inputs should follow the requirements described in section 2.1.3. At this point the code proceeds with the following steps:

1. (a) If *in_type* = 0, the roof displacement at each limit state and the idealised pushover curve parameters are extracted from *displacement_profile.csv* and *idealised_curve.csv* respectively.
   (b) If *in_type* = 1 the results from a pushover analysis are extracted from *displacements_pushover.csv* and *reactions_pushover.csv* and drift limit states from limits.csv. The idealised pushover curve is then derived in the *idealisation* function, where the idealisation process is conducted according to the Gem Analytical Vulnerability Guidelines.
2. The csv input files are parsed with the function *read_data* according to the defined options. The parameters essential to the analysis are return together with a graphical visualisation of the inputs if the variable *plotflag*[0] is equal to 1.
3. The parameters extracted are used in the *DFfragility* function, within the *fragility_process* function, to derive ductility levels $\mu_{ds}$, median spectral acceleration $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ at each limit state through the following steps:
   - The idealised MDoF system is transformed into an equivalent SDoF system, using $\Gamma_1$.
   - Ductility levels $\mu_{ds}$ corresponding to each damage threshold, are defined.
   - R and C$_R$ are computed, using eq. 2.25 and 2.24 respectively.
   - $\hat{S}_{a,ds}$ and the corresponding dispersion $\beta_{S_{a,d}}$ are computed using eq. 2.23 and 2.30 respectively.
   - R$_{corrected}$ curve are found reducing by 15% the median R curve, and the corresponding $\mu_{corrected}$ corrected are extrapolated.
   - b value is found from R and $\mu$ according to eq. **??**.
     (a) If the variable *MC* = 0 uncertainty in the model is expressed in terms of dispersion in S$_a$ $\beta_{S_{a,c}}$ according to eq. 2.31 and combined with $\beta_{S_{a,d}}$ to get the total dispersion $\beta_{S_a}$, using eq. 2.34.
     (b) If the variable *MC* is different from 0 different ductility limit states are sampled for each median ductility level $\mu_{ds}$ and for each of these MC values of $\hat{S}_{a,ds}$ and $\beta_{S_{a,d}}$ are computed, as described in section 2.1.4. Their median and the dispersion are estimated. These parameters constitute the median $\hat{S}_{a,ds}$ and the total dispersion $\beta_{S_a}$ for the considered damage state.
   - All $\hat{S}_{a,ds}(T_1)$ are converted to mean $\mu_{ln(S_{a,ds})}(T_1)$ and then to the intensity measure in common with the rest of the buildings, $\mu_{ln(S_{a,ds}(T_{av}))}$, according to eq. 2.16.
4. Step 2. and 3. are repeated for the number of input buildings.
5. (a) If *vuln* = 0: All $\hat{S}_{a,ds}$ and $\beta_{S_{a,d}}$ from all the buildings and all the sampled ductility limit states are combined in a single lognormal curve, as described in section 2.1.3. Mean $\mu_{ln(S_a)}$ and total dispersion $\beta_{S_a}$ are then converted to logarithmic mean $\mu_{S_a}$

and logarithmic covariance $cov_{S_a}$, according to equations 2.20 and 2.21 respectively. Fragility curves for the class of buildings are displayed if the variable *plotflag*[2] = 1, and logarithmic $\mu_{S_a}$ and $cov_{S_a}$ are exported in the *outputs* folder.

(b) If *vuln* =1: For the intensity measure levels defined in the variable *iml* a value of loss ratio $\mu_{LR,iml,blg}$ is defined for each building and a standard deviation $\sigma_{LR,iml,blg}$, if dispersion due to uncertainty in the limit state $\beta_{\theta c}$ is different from zero. They are finally combined in a single mean and standard deviationas described in section 2.1.3. Vulnerability curve for the class of buildings is displayed if the variable *plotflag*[3] = 1, and $\mu_{LR}$ and $cov_{LR}$ at each iml are exported in the *outputs* folder.

### 2.1.5 NLS methods without record-to-record dispersion

Still to implement

## 2.2 Non-linear Dynamic (NLD) Methods

Nonlinear Dynamic Methods are based on the results of many dynamic analyses, which relate the seismic response of a structure, represented by an Engineering Demand Parameter (EDP), like maximum top displacement, maximum inter-storey drift ratio, maximum top drift etc., to the Intensity Measure Level (IML) of the input accelerograms. Many methods exists in literature to perform a series of dynamic analysis and to post-process the results in order to derive fragility curves. Some of them treat a single building to estimate directly the median seismic intensity value corresponding to the attainment of different damage state threshold (limit state), and the corresponding dispersion (Vamvatsikos and Cornell, 2002, Ellingwood and Kinali, 2009). Others treat a class of buildings, and lead to the evaluation of the probabilities of different damage states for a series of IMLs and thus to the set up of a damage probability matrix (Singhal and Kiremidjian, 1996, Silva et al., 2013).

The last approach have been implemented in the DPM-based procedure, explained in section 2.2.2, from the point of view of the necessary scientific background behind and their step-by-step implementation in the python script.

### 2.2.1 Using the NLD module

To start using the nonlinear dynamic method a command line text editor should be used to enter manually the folder location where the *rmtk* has been saved, as shown in the example below:

```
cd path/to/rmtk/folder/RMTK
```

From the text editor iPython browser page can be opened with the following command line:

```
ipython-2.7 notebook --pylab=inline
```

Once the iPython page is opened on the browser, the python scripts contained in the *rmtk* directory will be visible. The file *NDM.ipynb* should be selected to start the calculations.

In the initial section of the script "Define Options" the user needs to set the options and to enter the input corresponding to the defined options in the folder *NDP/input*. In section 2.2.2 the alternatives values that the initial variables can assume and their meaning are described in detail, while the parameters to be inserted in the input files are fully described in section 2.2.2.

### 2.2.2 Damage probability matrix DPM-based procedure

This procedure performs the post-processing part of a set of dynamic analyses to first assemble a damage probability matrix, and then use this data for the derivation of a fragility function. Therefore the results of a set of dynamic analyses previously run have to be input to start the process. A list of intensity measure associated to each accelerogram, and corresponding EDP for each structure of the building class can be input, along with the set of limit states expressed in terms of the same EDP. The EDPs resulting from the dynamic analyses are compared with the limit state displacements and a global damage state is assigned to each structure. Thus, for each record, the number of frames in each damage state can be obtained. The distribution of buildings in each damage state is organised in the damage probability matrix, with a number of rows equal to the number of ground motion records and a number of columns equal to the number of damage states.

The processing of the data continue with the estimation of the cumulative fraction of structures in each damage state, summing the percentages of frames belonging to all the subsequent damage states. A lognormal cumulative distribution function, expressing the probability of exceeding each damage state in a continuous fashion, is then fit to these results, leading to the statistical parameters of the fragility curves. The regression analysis is carried out using the maximum likelihood method.

This function have the advantage of accounting for the record-to-record variability by the use of many ground motion records, and the inter-building variability subjecting to the same set of accelerograms hundreds of structures representing the entire building class.

To derive a discrete vulnerability function at certain intensity measure levels, the input damage-to-loss factors are applied to the probability of occurance of each damage state, extracted from the probability of exceedance of each damage state described by the fragility function. For the vector of selected intensity measure levels a value of loss ratio is thus defined.

#### Options

In the Options the user has to define first of all the type of inputs at disposal, setting the variable *in_ type*, choosing between entering a damage count matrix, which corresponds to a damage probability matrix, where the probabilities of each damage state are substituted by the count of buildings in that damage state, or IML and corresponding EDPs for each dynamic analysis

```
in_type = 0 # damage count matrix
in_type = 1# IMLs and EDPs
```

The variable *vuln* instead gives the opportunity to decide the type of outputs, whether to stop the process at the derivation of the fragility curves, or to go all the way up to the vulnerability curve definition, applying damage-to-loss functions.

```
vuln = 0 # derive fragility curves
vuln = 1 # derive vulnerability curve
```

The variable g has to be set to the value of the gravity acceleration, expressed consistently with the units used for the intensity measure input data. For example if the intensity measure used is PGA expressed in g, the variable g will be set to 1, if PGA is expressed in m/s$^2$ instead g will be set to 9.81.

The variable *iml* is a numpy array that identifies the intensity measure levels for which loss ratios are computed and provided in the vulnerability curve.

```
iml = np.linspace(0.1,15,100)
```

The variable *plotflag* allows or inhibits the displaying of plots. It is a python list composed of 2 integers, each one controlling a different plot: fragility and vulnerability function respectively. Each integer can take as value either zero or one, whether the corresponding graph has to be displayed or not:

```
plotflag = [1, 1] # plot all the graphs
plotflag = [0, 0] # do not plot any graph
```

The following variables set some of the characteristics of the plots:

- IMlabel: list of one strings defining the IM on the x axis as ['Sa(Tel)-m/s$^2$']
- linew: integer for defining lines width.
- fontsize : fontsize used for labels, graphs etc.

### Inputs

The inputs must be formatted as comma-separated value files (.csv), and saved in the folder *input*, contained in the NDP directory. If any other environment different from Windows is used make sure that the "comma separated values Windows" is selected as saving option when creating the input files.

Two types of input can be entered, whether the results of the set of dynamic analyses performed have already been organised in a damage probability matrix for or not. In the former case the variable *in_type* should be set to 0 and the damage count matrix should be input in the csv file *dcm.csv*. The first two columns refer to the number of record and the corresponding intensity measure level, the following columns report the number of buildings in each damage state, as shown in the following table:

| **n.records** | **Intensity Measure Level** | **DS$_0$** | **DS$_1$** | **DS$_2$** |
|---|---|---|---|---|
| 1 | 49.852 | 30 | 16 | 54 |
| 2 | 47.056 | 54 | 15 | 31 |
| 3 | 33.012 | 59 | 10 | 31 |
| 4 | 82.125 | 24 | 26 | 50 |
| ... | ... | ... | ... | ... |
| 5 | 37.499 | 58 | 5 | 37 |

In the latter case the variable *in_type* should be set to 1, and the results of the set of dynamic analyses should be entered in the *edp.csv* file in the following fashion: number of record, corresponding IML, and corresponding EDPs for each building subjected to that record.

| **n.records** | **IML** | **edp$_{blg,1}$** | **edp$_{blg,2}$** | **...** | **edp$_{blg,n}$** |
|---|---|---|---|---|---|
| 1 | 69.209 | -0.00069 | 0.00031 | ... | 0.00131 |
| 2 | 75.470 | 0.00102 | 0.00202 | ... | 0.00302 |
| 3 | 62.233 | -0.00090 | 0.00010 | ... | 0.00110 |
| 4 | 168.47 | -0.00246 | -0.00146 | ... | -0.00046 |
| 5 | 67.612 | 0.00095 | 0.00195 | ... | 0.00295 |
| ... | ... | ... | ... | ... | ... |
| n | 34.484 | 0.00036 | 0.00136 | ... | 0.00236 |

In this case also the limit states must be input in the *limits.csv* file. Each line of the file corresponds to the limit states of a building, but if all the buildings share the same limits a single line can be input.

| **n.building** | **LS$_1$** | **LS$_2$** | **LS$_3$** | **LS$_4$** |
|---|---|---|---|---|
| 1 | 0.003 | 0.010 | 0.025 | 0.0337 |
| 2 | 0.004 | 0.015 | 0.020 | 0.035 |
| 3 | 0.002 | 0.019 | 0.027 | 0.032 |
| ... | ... | ... | ... | ... |
| n | 0.0024 | 0.016 | 0.025 | 0.03 |

#### Calculations

The overall workflow of the DPM-based procedure is summarised in this section. The option *in_type* should be set according to the input at disposal. The corresponding inputs should follow the requirements described in the previous section. At this point the code proceeds with the following steps:

1. In the function *read_data* the inputs are read and the damage count matrix is returned.
   (a) If *in_type* = 0, the damage count matrix is extracted directly from the *dcm.csv* file.
   (b) If *in_type* = 1 the IMLs of the records used in the dynamic analyses and the corresponding EDPs are extracted from *edp.csv* and the limit states for each building, expressed in terms of the same EDP, are extracted from *limits.csv*. These data are converted into a damage count matrix according to the method described in section 2.2.2.
2. The parameters extracted are used to derive the Probability of Exceedance (PoE) of each limit state for each IML, as described in section 2.2.2.
3. The PoEs are fitted with a lognormal function using the maximum likelihood method. The mean $\mu_{lnIML}$ and standard deviation $\sigma_{lnIML}$ of the corresponding normal distribution for the entire class of buildings are found for each limit state.
4. The $\mu_{lnIML}$ and $\sigma_{lnIML}$ are converted to logarithmic $\mu_{IML}$ and $\sigma_{IML}$. The fragility curves for the class of buildings are displayed if the variable *plotflag*[0] = 1, and the logarithmic $\mu_{IML}$ and $\sigma_{IML}$ are exported in the *outputs* folder.
5. If *vuln* =1: For the intensity measure levels defined in the variable *iml* a value of loss ratio is defined, according to section 2.2.2. A vulnerability curve for the class of buildings is displayed if the variable *plotflag*[1] = 1, and the loss ratios at each iml are exported in the *outputs* folder.

### 2.2.3 Post-processing IDA

Post processing IDA

# 3. Plotting

# A. The 10 Minute Guide to Python!

The HMTK is intended to be used by scientists and engineers without the necessity of having an existing knowledge of Python. It is hoped that the examples contained in this manual should provide enough context to allow the user to understand how to use the tools for their own needs. In spite of this, however, an understanding of the fundamentals of the Python programming language can greatly enhance the user experience and permit the user to join together the tools in a workflow that best matches their needs.

The aim of this appendix is therefore to introduce some fundamentals of the Python programming language in order to help understand how, and why, the HMTK can be used in a specific manner. If the reader wishes to develop their knowledge of the Python programming language beyond the examples shown here, there is a considerable body of literature on the topic from both a scientific and developer perspective.

## A.1 Basic Data Types

Fundamental to the use of the HMTK is an understanding of the basic data types Python recognises:

### A.1.1 Scalar Parameters

- **float** A floating point (decimal) number. If the user wishes to enter in a floating point value then a decimal point must be included, even if the number is rounded to an integer.

```
>> a = 3.5
>> print a, type(a)
3.5 <type 'float'>
```

- **integer** An integer number. If the decimal point is omitted for a floating point number the number will be considered an integer

```
>> b = 3
>> print b, type(b)
3 <type 'int'>
```

The functions `float()` and `int()` can convert an integer to a float and vice-versa. Note that taking `int()` of a fraction will round the fraction down to the nearest integer

```
>> float(b)
3
>> int(a)
3
```

- **string** A text string (technically a “list” of text characters). The string is indicated by the quotation marks ”something” or ’something else’

```
>> c = "apples"
>> print c, type(c)
apples <type 'str'>
```

- **bool** For logical operations python can recognise a variable with a boolean data type (`True` / `False`).

```
>> d = True
>> if d:
        print "y"
    else:
        print "n"
y
>> d = False
>> if d:
        print "y"
    else:
        print "n"
n
```

*Care should be taken in Python as the value 0 and 0.0 are both recognised as False if applied to a logical operation. Similarly, booleans can be used in arithmetic where True and False take the values 1 and 0 respectively*

```
>> d = 1.0
>> if d:
        print "y"
    else:
        print "n"
y
>> d = 0.0
>> if d:
        print "y"
    else:
        print "n"
n
```

## Scalar Arithmetic

Scalars support basic mathematical operations (# indicates a comment):

```
>> a = 3.0
>> b = 4.0
>> a + b # Addition
7.0
>> a * b # Multiplication
12.0
>> a - b # Subtraction
-1.0
>> a / b # Division
0.75
>> a ** b  # Exponentiation
81.0
# But integer behaviour can be different!
>> a = 3; b = 4
>> a / b
0
>> b / a
1
```

### A.1.2 Iterables

Python can also define variables as lists, tuples and sets. These data types can form the basis for iterable operations. It should be noted that unlike other languages, such as Matlab or Fortran, Python iterable locations are zero-ordered (i.e. the first location in a list has an index value of 0, rather than 1).

- **List** A simple list of objects, which have the same or different data types. Data in lists can be re-assigned or replaced

```
>> a_list = [3.0, 4.0, 5.0]
>> print a_list
[3.0, 4.0, 5.0]
>> another_list = [3.0, "apples", False]
>> print another_list
[3.0, 'apples', False]
>> a_list[2] = -1.0
a_list = [3.0, 4.0, -1.0]
```

- **Tuples** Collections of objects that can be iterated upon. As with lists, they can support mixed data types. However, objects in a tuple cannot be re-assigned or replaced.

```
>> a_tuple = (3.0, "apples", False)
>> print a_tuple
(3.0, 'apples', False)
# Try re-assigning a value in a tuple
>> a_tuple[2] = -1.0
TypeError                        Traceback (most recent call last)
<ipython-input-43-644687cfd23c> in <module>()
----> 1 a_tuple[2] = -1.0

TypeError: 'tuple' object does not support item assignment
```

- **Range** A range is a convenient function to generate arithmetic progressions. They are called with a `start`, a `stop` and (optionally) a `step` (which defaults to 1 if not specified)

```
>> a = range(0, 5)
>> print a
[0, 1, 2, 3, 4]   # Note that the stop number is not
                  # included in the set!
>> b = range(0, 6, 2)
>> print b
[0, 2, 4]
```

- **Sets** A set is a special case of an iterable in which the elements are unordered, but contains more enhanced mathematical set operations (such as intersection, union, difference, etc.)

```
>> from sets import Set
>> x = Set([3.0, 4.0, 5.0, 8.0])
>> y = Set([4.0, 7.0])
>> x.union(y)
Set([3.0, 4.0, 5.0, 7.0, 8.0])
>> x.intersection(y)
Set([4.0])
>> x.difference(y)
Set([8.0, 3.0, 5.0]) # Notice the results are not ordered!
```

### Indexing

For some iterables (including lists, sets and strings) Python allows for subsets of the iterable to be selected and returned as a new iterable. The selection of elements within the set is done according to the index of the set.

```
>> x = range(0, 10)  # Create an iterable
>> print x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> print x[0] # Select the first element in the set
0            # recall that iterables are zero-ordered!
>> print x[-1] # Select the last element in the set
9
>> y = x[:] # Select all the elements in the set
>> print y
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> y = x[:4]  # Select the first four element of the set
>> print y
[0, 1, 2, 3]
>> y = x[-3:] # Select the last three elements of the set
>> print y
[7, 8, 9]
>> y = x[4:7] # Select the 4th, 5th and 6th elements
>> print y
[4, 5, 6]
```

## A.1.3 Dictionaries

Python is capable of storing multiple data types associated with a map of variable names inside a single object. This is called a "Dictionary", and works in a similar manner to a "data structure" in languages such as Matlab. Dictionaries are used frequently in the HMTK as ways of structuring inputs to functions that share a common behaviour but may take different numbers and types of parameters on input.

```
>> earthquake = {"Name": "Parkfield",
                 "Year": 2004,
                 "Magnitude": 6.1,
                 "Recording Agencies" = ["USGS", "ISC"]}
# To call or view a particular element in a dictionary
>> print earthquake["Name"], earthquake["Magnitude"]
Parkfield 6.1
```

## A.1.4 Loops and Logicals

Python's syntax for undertaking logical operations and iterable operations is relatively straightforward.

### Logical

A simple logical branching structure can be defined as follows:

```
>> a = 3.5
>> if a <= 1.0:
       b = a + 2.0
   elif a > 2.0:
       b = a - 1.0
   else:
       b = a ** 2.0
>> print b
2.5
```

Boolean operations can are simply rendered as `and`, `or` and `not`.

```
>> a = 3.5
>> if (a <= 1.0) or (a > 3.0):
        b = a - 1.0
   else:
        b = a ** 2.0
>> print b
2.5
```

### Looping

There are several ways to apply looping in python. For simple mathematical operations, the simplest way is to make use of the **range** function:

```
>> for i in range(0, 5):
        print i, i ** 2
0   0
1   1
2   4
3   9
4   16
```

The same could be achieved using the `while` function (though possibly this approach is far less desirable depending on the circumstance):

```
>> i = 0
>> while i < 5:
        print i, i ** 2
        i += 1
0   0
1   1
2   4
3   9
4   16
```

A `for` loop can be applied to any iterable:

```
>> fruit_data = ["apples", "oranges", "bananas", "lemons",
                 "cherries"]
>> i = 0
>> for fruit in fruit_data:
        print i, fruit
        i += 1
0   apples
1   oranges
2   bananas
3   lemons
4   cherries
```

The same results can be generated, arguably more cleanly, by making use of the `enumerate` function:

```
>> fruit_data = ["apples", "oranges", "bananas", "lemons",
                 "cherries"]
>> for i, fruit in enumerate(fruit_data):
        print i, fruit
0   apples
1   oranges
2   bananas
3   lemons
4   cherries
```

As with many other programming languages, Python contains the statements `break` to break out of a loop, and `continue` to pass to the next iteration.

```
>> i = 0
>> while i < 10:
        if i == 3:
            i += 1
            continue
        elif i == 5:
            break
        else:
            print i, i ** 2
        i += 1
0   0
1   1
2   4
4   16
```

## A.2 Functions

Python easily supports the definition of functions. A simple example is shown below. *Pay careful attention to indentation and syntax!*

```
>> def a_simple_multiplier(a, b):
        """
        Documentation string - tells the reader the function
        will multiply two numbers, and return the result and
        the square of the result
        """
        c = a * b
        return c, c ** 2.0

>> x = a_simple_multiplier(3.0, 4.0)
>> print x
(12.0, 144.0)
```

In the above example the function returns two outputs. If only one output is assigned then that output will take the form of a tuple, where the elements correspond to each of the two outputs. To assign directly, simply do the following:

```
>> x, y = a_simple_multiplier(3.0, 4.0)
>> print x
12.0
>> print y
144.0
```

## A.3 Classes and Inheritance

Python is one of many languages that is fully object-oriented, and the use (and terminology) of objects is prevalent throughout the HMTK and this manual. A full treatise on the topic of object oriented programming in Python is beyond the scope of this manual and the reader is referred to one of the many textbooks on Python for more examples

### A.3.1 Simple Classes

A class is an object that can hold both attributes and methods. For example, imagine we wish to convert an earthquake magnitude from one scale to another; however, if the earthquake occurred after a user-defined year we wish to use a different formula. This could be done by a method, but we can also use a class:

```
>> class MagnitudeConverter(object):
        """
        Class to convert magnitudes from one scale to another
        """
        def __init__(self, converter_year):
            """
            """
            self.converter_year = converter_year

        def convert(self, magnitude, year):
            """
            Converts the magnitude from one scale to another
            """
            if year < self.converter_year:
                converted_magnitude = -0.3 + 1.2 * magnitude
            else:
                converted_magnitude = 0.1 + 0.94 * magnitude
            return converted_magnitude

>> converter1 = MagnitudeConverter(1990)
>> mag_1 = converter1.convert(5.0, 1987)
>> print mag_1
5.7
>> mag_2 = converter1.convert(5.0, 1994)
>> print mag_2
4.8
# Now change the conversion year
>> converter2 = MagnitudeConverter(1995)
>> mag_1 = converter2.convert(5.0, 1987)
>> print mag_1
5.7
>> mag_2 = converter2.convert(5.0, 1994)
>> print mag_2
5.7
```

In this example the class holds both the attribute `converter_year` and the method to convert the magnitude. The class is created (or "instantiated") with only the information regarding the cut-off year to use the different conversion formulae. Then the class has a method to convert a specific magnitude depending on its year.

## A.3.2 Inheritance

Classes can be useful in many ways in programming. One such way is due to the property of inheritance. This allows for classes to be created that can inherit the attributes and methods of another class, but permit the user to add on new attributes and/or modify methods.

In the following example we create a new magnitude converter, which may work in the same way as the `MagnitudeConverter` class, but with different conversion methods.

```
>> class NewMagnitudeConverter(MagnitudeConverter):
        """
        A magnitude converter using different conversion
        formulae
        """
        def convert(self, magnitude, year):
            """
            Converts the magnitude from one scale to another
            - differently!!!
            """
            if year < self.converter_year:
                converted_magnitude = -0.1 + 1.05 * magnitude
            else:
                converted_magnitude = 0.4 + 0.8 * magnitude
            return converted_magnitude
# Now compare converters
>> converter1 = MagnitudeConverter(1990)
>> converter2 = NewMagnitudeConverter(1990)
>> mag1 = converter1.convert(5.0, 1987)
>> print mag1
5.7
>> mag2 = converter2.convert(5.0, 1987)
>> print mag2
5.15
>> mag3 = converter1.convert(5.0, 1994)
>> print mag3
4.8
>> mag4 = converter2.convert(5.0, 1994)
>> print mag4
4.4
```

## A.3.3 Abstraction

Inspection of the HMTK code (https://github.com/GEMScienceTools/hmtk) shows frequent usage of classes and inheritance. This is useful in our case if we wish to make available different methods for the same problem. In many cases the methods may have similar logic, or may provide the same types of outputs, but the specifics of the implementation may differ. Functions or attributes that are common to all methods can be placed in a “Base Class”, permitting each implementation of a new method to inherit the “Base Class” and its functions/attributes/behaviour. The new method will simply modify those aspects of the base class that are required for the specific method in question. This allows functions to be used interchangeably, thus allowing for a "mapping" of data to specific methods.

An example of abstraction is shown using our two magnitude converters shown previously. Imagine that a seismic recording network (named "XXX") has a model for converting from their locally recorded magnitude to a reference global scale (for the purposes of narrative, imagine that a change in recording procedures in 1990 results in a change of conversion model). A different recording network (named “YYY”) has a different model for converting their local magnitude to a reference global scale (and we imagine they also changed their recording procedures, but they did so in 1994). We can create a mapping that would apply the correct conversion for each locally recorded magnitude in a short catalogue, provided we know the local magnitude, the year and the recording network.

```
>> CONVERSION_MAP = {"XXX": MagnitudeConverter(1990),
                     "YYY": NewMagnitudeConverter(1994)}
>> earthquake_catalogue = [(5.0, "XXX", 1985),
                           (5.6, "YYY", 1992),
                           (4.8, "XXX", 1993),
                           (4.4, "YYY", 1997)]
>> for earthquake in earthquake_catalogue:
       converted_magnitude = \ # Line break for long lines!
           CONVERSION_MAP[earthquake[1]].convert(earthquake[0],
                                                 earthquake[2])
       print earthquake, converted_magnitude
(5.0, "XXX", 1985) 5.7
(5.6, "YYY", 1992) 5.78
(4.8, "XXX", 1993) 4.612
(4.4, "YYY", 1997) 3.92
```

So we have a simple magnitude homogenisor that applies the correct function depending on the network and year. It then becomes a very simple matter to add on new converters for new agencies; hence we have a "toolkit" of conversion functions!

## A.4 Numpy/Scipy

Python has two powerful libraries for undertaking mathematical and scientific calculation, which are essential for the vast majority of scientific applications of Python: Numpy (for multi-dimensional array calculations) and Scipy (an extensive library of applications for maths, science and engineering). Both libraries are critical to both OpenQuake and the HMTK. Each package is so extensive that a comprehensive description requires a book in itself. Fortunately there is abundant documentation via the online help for Numpy www.numpy.org and Scipy www.scipy.org, so we do not need to go into detail here.

The particular facet we focus upon is the way in which Numpy operates with respect to vector arithmatic. Users familiar with Matlab will recognise many similarities in the way the Numpy package undertakes array-based calculations. Likewise, as with Matlab, code that is well vectorised is signficantly faster and more efficient than the pure Python equivalent.

The following shows how to undertake basic array arithmetic operations using the Numpy library

```
>> import numpy as np
# Create two vectors of data, of equal length
>> x = np.array([3.0, 6.0, 12.0, 20.0])
>> y = np.array([1.0, 2.0, 3.0, 4.0])
# Basic arithmetic
>> x + y    # Addition (element-wise)
np.array([4.0, 8.0, 15.0, 24.0])
>> x + 2    # Addition of scalar
np.array([5.0, 8.0, 14.0, 22.0])
>> x * y    # Multiplication (element-wise)
np.array([3.0, 12.0, 36.0, 80.0])
>> x * 3.0    # Multiplication by scalar
np.array([9.0, 18.0, 36.0, 60.0])
>> x - y    # Subtraction (element-wise)
np.array([2.0, 4.0, 9.0, 16.0])
>> x - 1.0    # Subtraction of scalar
np.array([2.0, 5.0, 11.0, 19.0])
>> x / y    # Division (element-wise)
np.array([3.0, 3.0, 4.0, 5.0])
>> x / 2.0    # Division over scalar
np.array([1.5, 3.0, 6.0, 10.0])
>> x ** y     # Exponentiation (element-wise)
np.array([3.0, 36.0, 1728.0, 160000.0])
>> x ** 2.0    # Exponentiation (by scalar)
np.array([9.0, 36.0, 144.0, 400.0])
```

Numpy contains a vast set of mathematical functions that can be operated on a vector (e.g.):

```
>> x = np.array([3.0, 6.0, 12.0, 20.0])
>> np.exp(x)
np.array([2.00855369e+01, 4.03428793e+02, 1.62754791e+05,
          4.85165195e+08])
# Trigonometry
>> theta = np.array([0., np.pi / 2.0, np.pi, 1.5 * np.pi])
>> np.sin(theta)
np.array([0.0000, 1.0000, 0.0000, -1.0000])
>> np.cos(theta)
np.array([1.0000, 0.0000, -1.0000, 0.0000])
```

Some of the most powerful functions of Numpy, however, come from its logical indexing:

```
>> x = np.array([3.0, 5.0, 12.0, 21.0, 43.0])
>> idx = x >= 10.0    # Perform a logical operation
>> print idx
np.array([False, False, True, True, True])
>> x[idx]    # Return an array consisting of elements
             # for which the logical operation returned True
np.array([12.0, 21.0, 43.0])
```

Create, index and slice n-dimensional arrays:

```
>> x = np.array([[3.0,  5.0, 12.0, 21.0, 43.0],
                 [2.0,  1.0,  4.0, 12.0, 30.0],
                 [1.0, -4.0, -2.1,  0.0, 92.0]])
>> np.shape(x)
(3, 5)
>> x[:, 0]
np.array([3.0, 2.0, 1.0])
>> x[1, :]
np.array([2.0, 1.0, 4.0, 12.0, 30.0])
>> x[:, [1, 4]]
np.array([[ 5.0, 43.0],
          [ 1.0, 30.0],
          [-4.0, 92.0]])
```

The reader is referred to the online documentation for the full set of functions!