"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*

Disclaimer
The "Risk Modeller's Tookit - User Guide" is distributed in the hope that it will be useful, but without any warranty: without even the implied warranty of merchantability or fitness for a particular purpose. While every precaution has been taken in the preparation of this document, in no event shall the authors of the manual and the GEM Foundation be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of information contained in this document or from the use of programs and source code that may accompany it, even if the authors and GEM Foundation have been advised of the possibility of such damage. The Book provided hereunder is on as "as is" basis, and the authors and GEM Foundation have no obligations to provide maintenance, support, updates, enhancements, or modifications.
The current version of the book has been revised only by members of the GEM model facility and it must be considered a draft copy.

First printing, June 2015

Contents

Preface

To be completed by Vitor.

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

To be completed by Anirudh.

1.2 Current features

To be completed by Anirudh.

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

1. 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. 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. To be completed by Anirudh.

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) Spo2ida (Vamvatsikos and Cornell, 2006) R-/*mu*-T-based (Dolsek and Fajfar, 2004)
Non-linear Dynamic	DPM-based (Silva et al. 2013) Ida-postprocessing (Vamvatsikos and Cornell, 2002)

Table 1.1 – *Current algorithms in the HMTK*

2. Plotting

The OpenQuake-engine is capable of generating several seismic hazard and risk outputs, such as loss exceedance curves, seismic hazard curves, loss and hazard maps, damage statistics, amongst others. Most of these outputs are stored using the Natural Risk Markup Language (nrml), or simple comma separated value (csv) files. The Plotting module of the Risk Modellers Toolkit allows users to visualize the majority of the OpenQuake-engine results, as well as to convert them into other formats compatible with GIS software (e.g. QGIS). Despite the default styling of the maps and curves defined within the Risk Modellers Toolkit, it is important to state that any user can adjust the features of each output by modifying the original scripts.

2.1 Plotting damage distribution

Using the Scenario Damage Calculator (Silva et al., 2014) of the OpenQuake-engine, it is possible to assess the distribution of damage for a collection of assets considering a single seismic event. These results are comprised of damage per building typology, total damage distribution, and distribution of collapses in the region of interest.

2.1.1 Plotting damage distribution

This feature of the Plotting module allows users to plot the distribution of damage across the various vulnerability classes, as well as to the total damage distribution. For what concerns the former result, it is necessary to set the path to the output file using the parameter `tax_dmg_dist_file`. It is also possible to specify which vulnerability classes should be considered, using the parameter `taxonomy_list`. However, if a user wishes to consider all of the vulberability classes, then this parameter should be left empty. It is also possible to specify is a 3D plot containing all of the vulnerability classes should be generated, or a instead a 2D plot per vulnerability class. For follow the former option, the parameter `plot_3d` should be set to `True`. It is important to understand that this option leads to a plot of damage fractions for each vulnerability class, instead of the number of assets in each damage state. An example of this output is illustrated in Figure 2.1.

In order to plot the total damage distribution (considering the entire colection of assets), it is necessary to use the parameter `total_dmg_dist_file` to define the path to the respective output file.Figure 2.2 presents an example of this type of output.

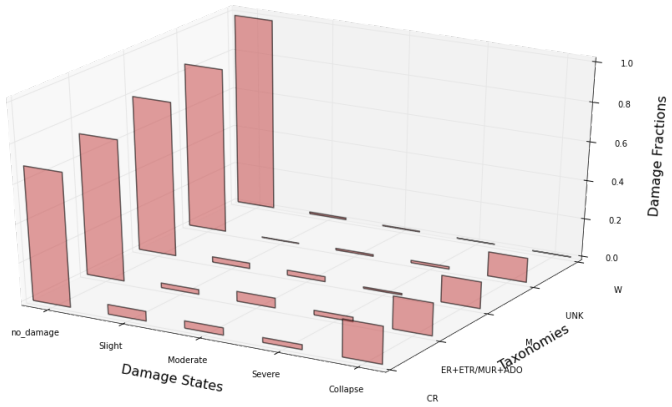

Figure 2.1 – *Damage disaggregation per vulnerability class.*

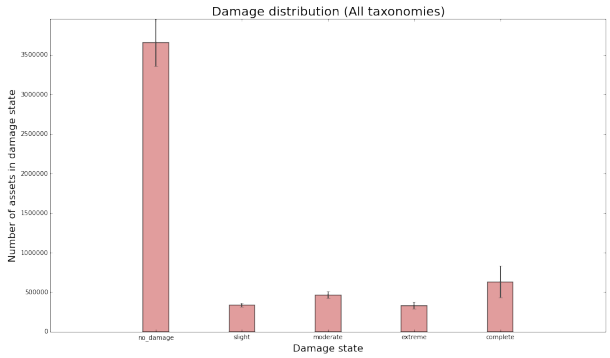

Figure 2.2 – *Total damage distribution.*

2.1.2 Plotting collapse maps

The OpenQuake-engine also generates an output defining the spatial distribution of the mean (and associated standard deviation) of assets in the last damage state (usually representing collapse or complete damage). The location of this output needs to be specified using the parameter `collapse_map`. Then, it is necessary to specify whether the user desires a map with the aggregated number of collapses (i.e. at each location, the mean number of collapses across all of the vulnerability classes are summed) or a map for each vulnerability class. Thus, the following options are permitted:

1. Aggregated collapse map only.
2. Collapse maps per taxonomy only.
3. Both aggregated and taxonomy-based.

The plotting option should be specified using the parameter `plotting_type`, and the location of the exposure model used to perform the calculations must be defined using the variable `exposure_model`. A number of other parameters can also be adjusted to modify the style of the resulting collapse map as follows:

- `bounding_box`: If set to 0, the Plotting module will calculate the geographical distribution of the assets, and adjust the limits of the map accordingly. Alternatively, a user can also specify the minimum/maximum latitude and longitude that should be used i nthe creation of the map.

- `marker_size`: This attribute can be used to adjust the size of the markers in the map.
- `log_scale`: If set to `True`, it will apply a logarithmic scale on the color scheme of the map, pottentially allowing a better visualization of the variation of the numbers of collapses in the region of interest.

An example of a collapse map is presented in Figure 2.3.

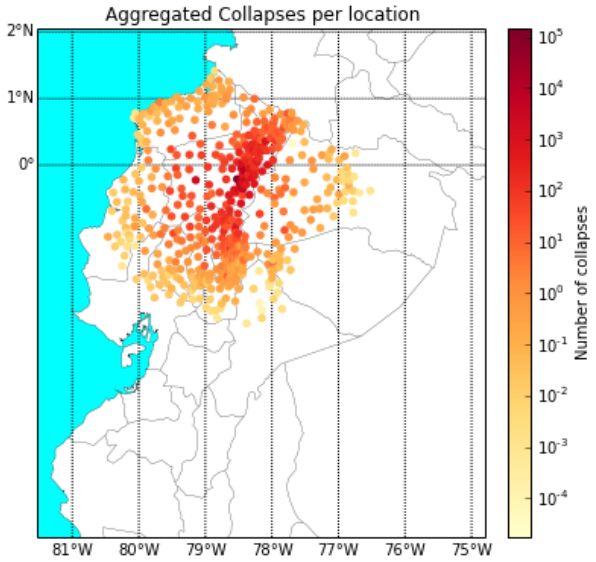

Figure 2.3 – *Spatial distribution of the mean number of collapses.*

2.2 Plotting hazard and loss curves

Using the Classical PSHA-based or Probabilistic Event-based Calculators (Silva et al., 2014, Pagani et al., 2014) of the OpenQuake-engine, it is possible to calculate seismic hazard curves for a number of locations, or loss exceedance curves considering a collection patially distributted assets.

2.2.1 Plotting hazard curves and uniform hazard spectra

A seismic hazard curve defines the probability of exceeding a number of intensity measure levels (e.g. peak ground acceleration or spectral acceleration) for a given interval of time (e.g. 50 years). In order to plot these curves, it is necessary to define the path to the output file in the parameter `hazard_curve_file`. Then, since each output file might contained a great number of haard curves, it is necessary to establish the location for each the hazard curve will be extracted. To visualize the list of locations comprised in the output file, the function `hazard_curves.loc_list` can be employed. Then, the choosen location must be provided to the plotting function (e.g. `hazard_curves.plot("81.213823|29.761172")`). An example of a seismic hazard curve is provided in Figure 2.4.

To plot uniform hazard spectra (UHS), a similar approach should be followed. The output file containing the uniform hazard spectra should be defined using the parameter `uhs_file`, and then a location must be provided to the plotting function (e.g.`uhs.plot("81.213823|29.761172")`). An example of uniform hazard spectra is illustrated in Figure 2.5.

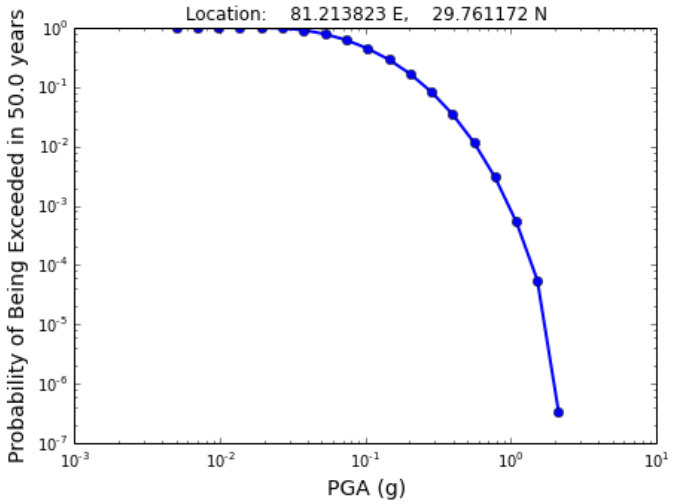

Figure 2.4 – *Seismic hazard curve for peak ground acceleration (PGA).*

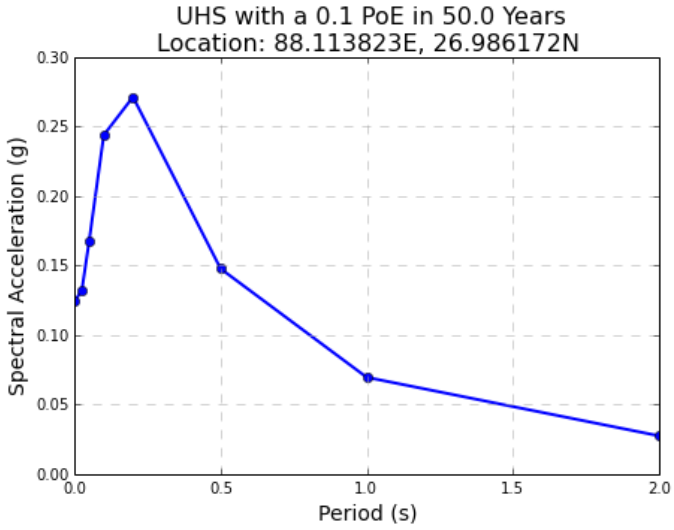

Figure 2.5 – *Uniform Hazard Spectra for a probability of exceedance of 10% in 50 years.*

2.2.2 Plotting loss curves

A loss exceedance curve defines the relation between a set of loss levels and the corresponding probability of exceedance within a given time span (e.g. a year). In order to plot these curves, it is necessary to define the location of the output file using the parameter `loss_curves_file`. Since each output file may contains a large number of loss exceedance curves, it is necessary to define for which assets will the loss curves be extracted. The parameter `assets_list` should be employed to define all of the chosen asset ids. These ids can be visualize directly on the loss curve output file, or on the exposure model used for the risk calculations. It is also possible to define a logarithimic scale for the x and y axis using the parameters `log_scale_x` and `log_scale_y`. A loss exceedance curve for a single asset is depicted in Figure 2.6.

2.3 Plotting hazard and loss maps

The OpenQuake-engine offers the possibility of calculating seismic hazard and loss (or risk) maps. To do so, it utilizes the seismic hazard or loss exceedances curves, to estimate the

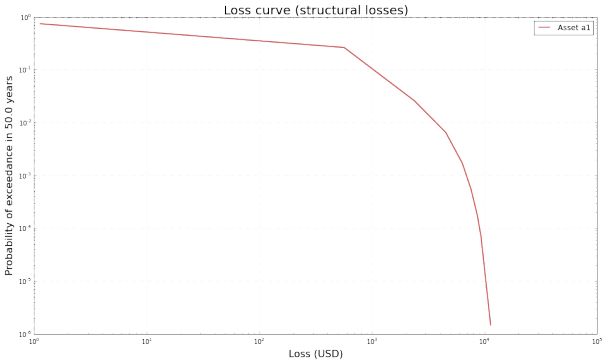

Figure 2.6 – *Loss exceedance curve.*

corresponding hazard or loss for the pre-defined return period (or probability of exceedance within a given interval of time).

2.3.1 Plotting hazard maps

A seismic hazard map provides the expected ground motion (e.g. peak ground acceleration or spectral acceleration) at each location, for a certain return period (or probability of exceedance within a given interval of time). To plot this type of maps, it is necessary to specify the location of the output file using the parameter `hazard_map_file`. An example hazard map is displayed in Figure 2.4.

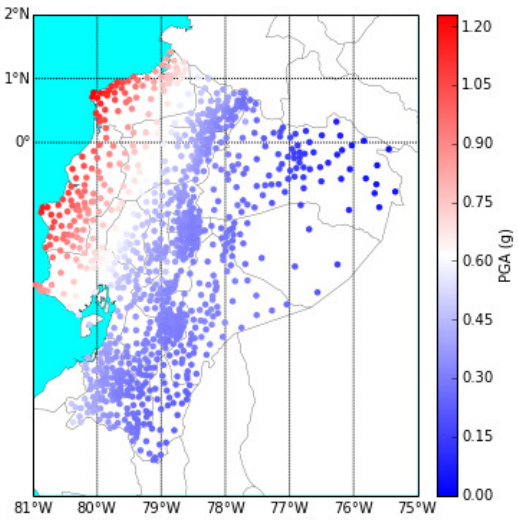

Figure 2.7 – *Seismic hazard map for a probability of exceedance of 10% in 50 years.*

2.3.2 Plotting loss maps

A loss map provides the estimated losses for a collection of assets, for a certain return period (or probability of exceedance within a given interval of time). It is important to understand that these maps are not providing the distribution of losses for a seismic event for the chosen return period, nor the losses whose sum would correspond to the aggreagted loss for the same return period. This type of maps is simply providing the expected loss for a specified level of frequency for each asset. To use this feature, it is necessary to define the parth ot the output file using the parameter `loss_map_file`, as well as the exposure model used to perform the risk

calculations through the parameter `exposure_model`. Then, similarly to what was explained in section 2.1.2 for collapse maps, it is possible to follow three approaches to generate the loss maps:

1. Aggregated loss map only.
2. Loss maps per taxonomy only.
3. Both aggregated and taxonomy-based.

Then, there are a number of options that can be used to modify the style of the maps. These include the size of the marker of the map (`marker_size`), the geographical limits of the map (`bounding_box`), and the employment of a logarithmic spacing for the color scheme (`log_scale`). An example loss map for a single vulnerability class is presented in Figure 2.8.

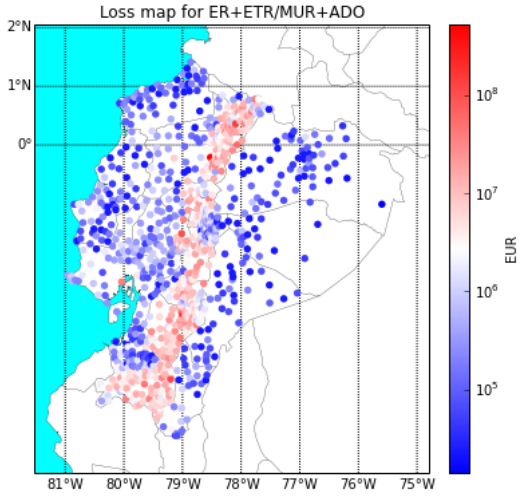

Figure 2.8 – *Loss (economic) map for a probability of exceedance of 10% in 50 years.*

As mentioned on the introductory section, it is also possible to convert any of the maps into a format (csv) easily readable by GIS software. To do so, it is necessary to set the parameter `export_map_to_csv` to `True`. As an example, a map containing the average annual losses for Ecuador has been converted to the `csv` format, and introduced into the QGIS software to produce the map presented in Figure 2.9.

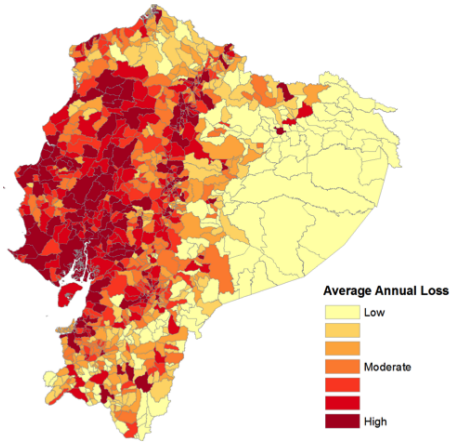

Figure 2.9 – *Average annual (economic) losses for Ecuador.*

3. Risk

The OpenQuake-engine currently generates the most common seismic hazard and risk results (e.g. hazard maps, loss curves, average annual losses). However, it is recognized that there are a number of other metrics that might not be of interest of the general GEM community, but fundamental for specific users. This module of the Risk Modellers Toolkit aims to provide users with additional risk results and funtionalities.

3.1 Deriving Probable Maximum Losses (PML)

The Probabilistic Event-based Risk calculator ((Silva et al., 2014)) of the OpenQuake-engine is capable of calculating event loss tables, which contain a list of earthquake ruptures and associarted losses. These losses my refer to specific assets, or the sum of the losses from the entire building portfolio (aggregated loss curves). Using this module, it is possible to derive a probable maximum loss (PML) curves (i.e. relation between a set of loss levels and corresponding return peridos of exceedance), as illustrated in Figure 3.1.

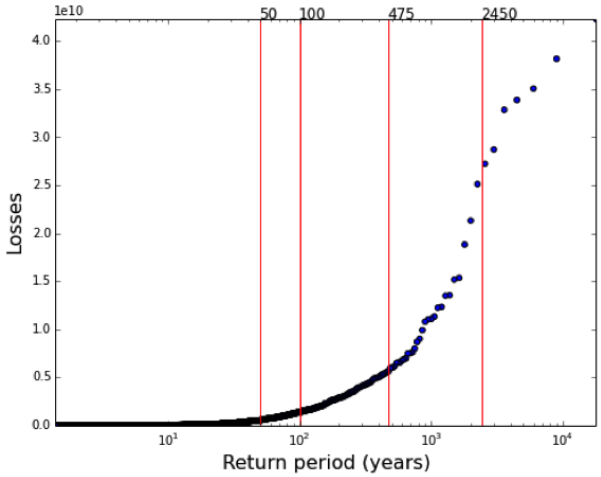

Figure 3.1 – *Probable Maximum Loss (PML) curve.*

To use this feature, it is necessary to use the parameter `event_loss_table_folder` to

specify the location of the folder that contains the set of event loss tables and stochastic event sets. Then, it is also necessary to provide the total economic value of the building portfolio (using the variable `total_cost`) and the list of return periods of interest (using the variable `return_periods`). This module also offers the possibility of saving all of the information in `csv` files, which can be used in other software (e.g. Microsoft Excel) for other purposes. To do so, the parameters `save_elt_csv` and `save_ses_csv` should be set to `True`.

3.2 Selecting a logic tree branch

To be completed by Anirudh.

4. Vulnerability

4.1 Introduction

To be completed by Anirudh.

4.2 Definition input models

To be completed by Anirudh.

4.3 Model generator

To be completed by Vitor.

4.3.1 Generation of capacity curves using DBELA

To be completed by Vitor.

4.3.2 Generation of capacity curves using SP-BELA

To be completed by Vitor.

4.3.3 Generation of capacity curves using point dispersion

To be completed by Vitor.

4.4 Conversion from MDOF to SDOF

To be completed by Vitor.

4.4.1 Conversion based on one mode of vibration

To be completed by Anirudh.

4.4.2 Conversion using an adaptive approach

To be completed by Anirudh.

4.5 Derivation of fragility

As explained by Lin and Miranda (2008), performance-based evaluation of structures requires the estimation of a demand spectrum that can properly take into account the inelastic behaviour of the structure(s) under analysis, namely, their capacity to dissipate seismic energy through hysteresis (Monteiro, 2011). In order to achieve this, two main approaches exist. The first one estimates the maximum inelastic deformation of a structure by applying a modification factor to its maximum linear elastic deformation. This factor can be damping-based or ductility-based. The former case reduces both the displacement and acceleration spectral coordinates by a reduction factor B, which is based on the equivalent viscous damping of the system. On the other hand, the latter reduce the spectral acceleration ordinates of a 5%-damped elastic response spectrum by a factor dependant on the ductility of the system.

The second approach estimates the maximum deformation of a nonlinear structure as the maximum deformation of an equivalent linear system with longer period of vibration (i.e. more flexible lateral stiffness) and higher viscous damping than those of the "original" structure. These parameters can be dependant either on the displacement ductility of the system (which implies an iterative process) or on the strength ratio R (or strength reduction factor).

4.5.1 SPO2IDA (Vamvatsikos and Cornell 2006)

To be completed by Chiara.

4.5.2 Dolsek and Fajfar 2004

To be completed by Chiara.

4.5.3 Ruiz Garcia and Miranda 2007

To be completed by Chiara.

4.5.4 Vidic and Fajfar 1994

To be completed by Vitor.

4.5.5 Lin and Miranda 2008

To be completed by Vitor.

4.5.6 Miranda (2000) for firm soils

To be completed by Vitor.

4.5.7 N2 (EC8, CEN 2005)

To be completed by Vitor.

4.5.8 Capacity Spectrum Method (FEMA, 2005)

To be completed by Vitor.

4.5.9 DBELA (Silva et al. 2013)

To be completed by Vitor.

4.5.10 Nonlinear time-history analysis in Single Degree of Freedom (SDOF) Oscilators

To be completed by Vitor.

Part I

Appendices

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!