CONTENTS

1 Introduction 3

2 Common mistakes: bad configuration parameters 5
   2.1 The quadratic parameters ........................................ 5
   2.2 Maximum distance ............................................... 6
   2.3 pointsource_distance ........................................... 7
   2.4 The linear parameters: width_of_mfd_bin and intensity levels .......... 9
   2.5 concurrent_tasks parameter ..................................... 9

3 Tips for running large hazard calculations 11
   3.1 Reducing a calculation .......................................... 11
   3.2 Classical PSHA for Europe ...................................... 12
   3.3 GMFs for California ............................................. 14

4 Tips for running large risk calculations 17
   4.1 Understanding the hazard ........................................ 17
   4.2 Collapsing of branches .......................................... 18
   4.3 Splitting the calculation into subregions ....................... 18
   4.4 Trimming of the logic-trees or sampling of the branches ........... 19
   4.5 Disabling the propagation of vulnerability uncertainty to losses .......... 19
   4.6 The asset loss table ............................................ 20
   4.7 The Probable Maximum Loss (PML) and the loss curves ............. 21
   4.8 Aggregating by multiple tags .................................... 23

5 The concept of effective realizations 25
   5.1 Reduction of the logic tree ...................................... 26
   5.2 How to analyze the logic tree of a calculation without running the calculation ... 27

6 Logic tree sampling strategies 29

7 Rupture sampling: how does it work? 31

8 Extra tips specific to event based calculations 35
20.3 Scenarios from ShakeMaps ........................................ 81

21 Developing with the engine ........................................ 85
   21.1 Prerequisites ..................................................... 85
   21.2 The first thing to do ............................................. 85
   21.3 Understanding the engine ...................................... 86
   21.4 Running calculations programmatically ..................... 87
   21.5 Case study: computing the impact of a source on a site .... 87
   21.6 Working with GMPEs directly: the ContextMaker ........... 89

22 Architecture of the OpenQuake engine ......................... 93
   22.1 Components of the OpenQuake Engine ....................... 93
   22.2 Design principles .............................................. 95

23 Some useful oq commands ........................................... 97
   23.1 oq zip .......................................................... 100
   23.2 Importing a remote calculation ............................... 101
   23.3 plotting commands ............................................ 101
   23.4 prepare_site_model ........................................... 102
   23.5 Reducing the source model .................................... 104
   23.6 Comparing hazard results ..................................... 104

24 Limitations of Floating-point Arithmetic ....................... 107
This manual is for advanced users, i.e. people who already know how to use the engine and have already read the official manual cover-to-cover. If you have just started on your journey of using and working with the OpenQuake engine, this manual is probably NOT for you. Beginners should study the official manual first. This manual is intended for users who are either running large calculations or those who are interested in programatically interacting with the datastore.

For the purposes of this manual a calculation is large if it cannot be run, i.e. if it runs out of memory, it fails with strange errors (rabbitmq errors, pickling errors, etc.) or it just takes too long to complete.

There are various reasons why a calculation can be too large. 90% of the times it is because the user is making some mistakes and she is trying to run a calculation larger than she really needs. In the remaining 10% of the times the calculation is genuinely large and the solution is to buy a larger machine, or to ask the OpenQuake developers to optimize the engine for the specific calculation that is giving issues.

The first things to do when you have a large calculation is to run the command `oq info --report job.ini`, that will tell you essential information to estimate the size of the full calculation, in particular the number of hazard sites, the number of ruptures, the number of assets and the most relevant parameters you are using. If generating the report is slow, it means that there is something wrong with your calculation and you will never be able to run it completely unless you reduce it.

The single most important parameter in the report is the number of effective ruptures, i.e. the number of ruptures after distance and magnitude filtering. For instance your report could contain numbers like the following:

```
#eff_ruptures 239,556
#tot_ruptures 8,454,592
```

This is an example of a computation which is potentially large - there are over 8 million ruptures generated by the model - but that in practice will be very fast, since 97% of the ruptures will be filtered away. The report gives a conservative estimate, in reality even more ruptures will be discarded.
It is very common to have an unhappy combinations of parameters in the `job.ini` file, like discretization parameters that are too small. Then your source model will contain millions and millions of ruptures and the computation will become impossibly slow or it will run out of memory. By playing with the parameters and producing various reports, one can get an idea of how much a calculation can be reduced even before running it.

Now, it is a good time to read the section about common mistakes.
By far, the most common source of problems with the engine is the choice of parameters in the job.ini file. It is very easy to make mistakes, because users typically copy the parameters from the OpenQuake demos. However, the demos are meant to show off all of the features of the engine in simple calculations, they are not meant for getting performance in large calculations.

2.1 The quadratic parameters

In large calculations, it is essential to tune a few parameters that are really important. Here is a list of parameters relevant for all calculators:

**maximum_distance:** The larger the maximum_distance, the more sources and ruptures will be considered; the effect is quadratic, i.e. a calculation with maximum_distance=500 km could take up to 6.25 times more time than a calculation with maximum_distance=200 km.

**region_grid_spacing:** The hazard sites can be specified by giving a region and a grid step. Clearly the size of the computation is quadratic with the inverse grid step: a calculation with region_grid_spacing=1 will be up to 100 times slower than a computation with region_grid_spacing=10.

**area_source_discretization:** Area sources are converted into point sources, by splitting the area region into a grid of points. The area_source_discretization (in km) is the step of the grid. The computation time is inversely proportional to the square of the discretization step, i.e. calculation with area_source_discretization=5 will take up to four times more time than a calculation with area_source_discretization=10.

**rupture_mesh_spacing:** Fault sources are computed by converting the geometry of the fault into a mesh of points; the rupture_mesh_spacing is the parameter determining the size of the mesh. The computation time is quadratic with the inverse mesh spacing. Using a rupture_mesh_spacing=2 instead of rupture_mesh_spacing=5 will make
your calculation up to 6.25 times slower. Be warned that the engine may complain if the rupture_mesh_spacing is too large.

**complex_fault_mesh_spacing**: The same as the rupture_mesh_spacing, but for complex fault sources. If not specified, the value of rupture_mesh_spacing will be used. This is a common cause of problems; if you have performance issue you should consider using a larger complex_fault_mesh_spacing. For instance, if you use a rupture_mesh_spacing=2 for simple fault sources but complex_fault_mesh_spacing=10 for complex fault sources, your computation can become up to 25 times faster, assuming the complex fault sources are dominating the computation time.

### 2.2 Maximum distance

The engine gives users a lot of control on the maximum distance parameter. For instance, you can have a different maximum distance depending on the tectonic region, like in the following example:

```python
maximum_distance = {'Active Shallow Crust': 200, 'Subduction': 500}
```

You can also have a magnitude-dependent maximum distance:

```python
maximum_distance = [(5, 0), (6, 100), (7, 200), (8, 300)]
```

In this case, given a site, the engine will completely discard ruptures with magnitude below 5, keep ruptures up to 100 km for magnitudes between 5 and 6, keep ruptures up to 200 km for magnitudes between 6 and 7, keep ruptures up to 300 km for magnitudes over 7.

You can have both trt-dependent and mag-dependent maximum distance:

```python
maximum_distance = {
    'Active Shallow Crust': [(5, 0), (6, 100), (7, 200), (8, 300)],
    'Subduction': [(6.5, 300), (9, 500)]
}
```

Given a rupture with tectonic region type trt and magnitude mag, the engine will ignore all sites over the maximum distance md(trt, mag). The precise value is given via linear interpolation of the values listed in the job.ini; you can determine the distance as follows:

```python
>>> from openquake.hazardlib.calc.filters import MagDepDistance
>>> md = MagDepDistance.new('[(5, 0), (6, 100), (7, 200), (8, 300)]')
>>> md('TRT', 4.5)
0.0
>>> md('TRT', 5.5)
50.0
>>> md('TRT', 6.5)
150.0
```
2.3 pointsource_distance

PointSources (and MultiPointSources and AreaSources, which are split into PointSources and therefore are effectively the same thing) are not pointwise for the engine: they actually generate ruptures with rectangular surfaces which size is determined by the magnitude scaling relationship. The geometry and position of such rectangles depends on the hypocenter distribution and the nodal plane distribution of the point source, which are used to model the uncertainties on the hypocenter location and on the orientation of the underlying ruptures.

Is the effect of the hypocenter/nodal planes distributions relevant? Not always: in particular, if you are interested in points that are far away from the rupture the effect is minimal. So if you have a nodal plane distribution with 20 planes and a hypocenter distribution with 5 hypocenters, the engine will consider 20 x 5 ruptures and perform 100 times more calculations than needed, since at large distance the hazard will be more or less the same for each rupture.

To avoid this performance problem there is a pointsource_distance parameter: you can set it in the job.ini as a dictionary (tectonic region type -> distance in km) or as a scalar (in that case it is converted into a dictionary {"default": distance} and the same distance is used for all TRTs). For sites that are more distant than the pointsource_distance from the point source, the engine (starting from release 3.11) creates an average rupture by taking weighted means of the parameters strike, dip, rake and depth from the nodal plane and hypocenter distributions and by rescaling the occurrence rate. For closer points, all the original ruptures are considered. This approximation (we call it rupture collapsing because it essentially reduces the number of ruptures) can give a substantial speedup if the model is dominated by PointSources and there are several nodal planes/hypocenters in the distribution. In some situations it also makes sense to set

pointsource_distance = 0

to completely remove the nodal plane/hypocenter distributions. For instance the Indonesia model has 20 nodal planes for each point sources; however such model uses the so-called equivalent distance approximation which considers the point sources to be really pointwise. In this case the contribution to the hazard is totally independent from the nodal plane and by using pointsource_distance = 0 one can get exactly the same numbers and run the model in 1 hour instead of 20 hours. Actually, starting from engine 3.3 the engine is smart enough to recognize the cases where the equivalent distance approximation is used and automatically set pointsource_distance = 0.

Even if you not using the equivalent distance approximation, the effect of the nodal plane/hypocenter distribution can be negligible: I have seen cases when setting setting

2.3. pointsource_distance
pointsource_distance = 0 changed the result in the hazard maps only by 0.1% and gained an order of magnitude of speedup. You have to check on a case by case basis.

There is a good example of use of the `pointsource_distance` in the MultiPointClassicalPSHA demo. Here we will just show a plot displaying the hazard curve without `pointsource_distance` (with ID=-2) and with `pointsource_distance`=200 km (with ID=-1). As you see they are nearly identical but the second calculation is ten times faster.

The `pointsource_distance` is also crucial when using the point source gridding approximation: then it can be used to speedup calculations even when the nodal plane and hypocenter distributions are trivial and no speedup would be expected.

**NB:** the `pointsource_distance` approximation has changed a lot across engine releases and you should not expect it to give always the same results. In particular, in engine 3.8 it has been extended to take into account the fact that small magnitudes will have a smaller collapse distance. For instance, if you set `pointsource_distance=100`, the engine will collapse the ruptures over 100 km for the maximum magnitude, but for lower magnitudes the engine will consider a (much) shorter collapse distance and will collapse a lot more ruptures. This is possible because given a tectonic region type the engine knows all the GMPEs associated to that tectonic region and can compute an upper limit for the maximum intensity generated by a rupture at any distance. Then it can invert the curve and given the magnitude and the maximum intensity can determine the
collapse distance for that magnitude.

In engine 3.11, contrarily to all previous releases, finite side effects are not ignored for distance sites, they are simply averaged over. This gives a better precision. In some case (i.e. the Alaska model) versions of the engine before 3.11 could give a completely wrong hazard on some sites. This is now fixed.

Note: setting `pointsource_distance=0` does not completely remove finite size effects. If you want to replace point sources with points you need to also change the magnitude-scaling relationship to `PointMSR`. Then the area of the underlying planar ruptures will be set to 1E-4 squared km and the ruptures will effectively become points.

### 2.4 The linear parameters: width_of_mfd_bin and intensity levels

The number of ruptures generated by the engine is controlled by the parameter `width_of_mfd_bin`; for instance if you raise it from 0.1 to 0.2 you will reduce by half the number of ruptures and double the speed of the calculation. It is a linear parameter, at least approximately. Classical calculations are also roughly linear in the number of intensity measure types and levels. A common mistake is to use too many levels. For instance a configuration like the following one:

```python
intensity_measure_types_and_levels = {
    "PGA": logscale(0.001,4.0, 100),
    "SA(0.3)": logscale(0.001,4.0, 100),
    "SA(1.0)": logscale(0.001,4.0, 100)
}
```

requires computing the PoEs on 300 levels. Is that really what the user wants? It could very well be that using only 20 levels per each intensity measure type produces good enough results, while potentially reducing the computation time by a factor of 5.

### 2.5 concurrent_tasks parameter

There is a last parameter which is worthy of mention, because of its effect on the memory occupation in the risk calculators and in the event based hazard calculator.

**concurrent_tasks**: This is a parameter that you should not set, since in most cases the engine will figure out the correct value to use. However, in some cases, you may be forced to set it. Typically this happens in event based calculations, when computing the ground motion fields. If you run out of memory, increasing this parameter will help, since the engine will produce smaller tasks. Another case when it may help is when computing hazard statistics with lots of sites and realizations, since by increasing this parameter the tasks will contain less sites.
Notice that if the number of `concurrent_tasks` is too big the performance will get worse and the data transfer will increase: at a certain point the calculation will run out of memory. I have seen this to happen when generating tens of thousands of tasks. Again, it is best not to touch this parameter unless you know what you are doing.
TIPS FOR RUNNING LARGE HAZARD CALCULATIONS

Running large hazard calculations, especially ones with large logic trees, is an art, and there are various techniques that can be used to reduce an impossible calculation to a feasible one.

3.1 Reducing a calculation

The first thing to do when you have a large calculation is to reduce it so that it can run in a reasonable amount of time. For instance you could reduce the number of sites, by considering a small geographic portion of the region interested, or by increasing the grid spacing. Once the calculation has been reduced, you can run it and determine what are the factors dominating the run time.

As we discussed in section common mistakes, you may want to tweak the quadratic parameters (maximum_distance, area_source_discretization, rupture_mesh_spacing, complex_fault_mesh_spacing). Also, you may want to choose different GMPEs, since some are faster than others. You may want to play with the logic tree, to reduce the number of realizations: this is especially important, in particular for event based calculation were the number of generated ground motion fields is linear with the number of realizations.

Once you have tuned the reduced computation, you can have an idea of the time required for the full calculation. It will be less than linear with the number of sites, so if you reduced your sites by a factor by a number of 100, the full computation will take a lot less than 100 times the time of the reduced calculation (fortunately). Still, the full calculation can be impossible because of the memory/data transfer requirements, especially in the case of event based calculations. Sometimes it is necessary to reduce your expectations. The examples below will discuss a few concrete cases. But first of all, we must stress an important point:

3.2 Classical PSHA for Europe

Suppose you want to run a classical PSHA calculation for the latest model for Europe and that it turns out to be too slow to run on your infrastructure. Let’s say it takes 4 days to run. How do you proceed to reduce the computation time?

The first thing that comes to mind is to tune the area_source_discretization parameter, since the calculation (as most calculations) is dominated by area sources. For instance, by doubling it (say from 10 km to 20 km) we would expect to reduce the calculation time from 4 days to just 1 day, a definite improvement.

But how do we check if the results are still acceptable? Also, how we check that in less than 4+1=5 days? As we said before we have to reduce the calculation and the engine provides several ways to do that.

If you want to reduce the number of sites, IMTs and realizations you can:

- manually change the sites.csv or site_model.csv files
- manually change the region_grid_spacing
- manually change the intensity_measure_types_and_levels variable
- manually change the GMPE logic tree file by commenting out branches
- manually change the source logic tree file by commenting out branches
- use the environment variable OQ_SAMPLE_SITES
- use the environment variable OQ_REDUCE

Starting from engine 3.11 the simplest approach is to use the OQ_REDUCE environment variable than not only reduce reduces the number of sites, but also reduces the number of intensity measure types (it takes the first one only) and the number of realizations to just 1 (it sets number_of_logic_tree_samples=1) and if you are in an event based calculation reduces the parameter ses_per_logic_tree_path too. For instance the command:

```
$ OQ_REDUCE=.01 oq engine --run job.ini
```

will reduce the number of sites by 100 times by random sampling, as well a reducing to 1 the number of IMTs and realizations. As a result the calculation will be very fast (say 1 hour instead of 4 days) and it will possible to re-run it multiple times with different parameters. For instance, you can test the impact of the area source discretization parameter by running:

```
$ OQ_REDUCE=.01 oq engine --run job.ini --param area_source_discretization=20
```

Then the engine provides a command oq compare to compare calculations; for instance:
$ oq compare hmaps PGA -2 -1 --atol .01

will compare the hazard maps for PGA for the original (ID=-2, area_source_discretization=10 km) and the new calculation (ID=-2, area_source_discretization=20 km) on all sites, printing out the sites where the hazard values are different more than .01 g (--atol means absolute tolerance). You can use `oq compare --help` to see what other options are available.

If the call to `oq compare` gives a result:

```
There are no differences within the tolerances atol=0.01, rtol=0%, ...
```

it means that within the specified tolerance the hazard is the same on all the sites, so you can safely use the area discretization of 20 km. Of course, the complete calculation will contain 100 times more sites, so it could be that in the complete calculation some sites will have different hazard. But that’s life. If you want absolute certitude you will need to run the full calculation and to wait. Still, the reduced calculation is useful, because if you see that are already big differences there, you can immediately assess that doubling the `area_source_discretization` parameter is a no go and you can try other strategies, like for instance doubling the `width_of_mfd_bin` parameter.

As of version 3.11, the `oq compare hmaps` command will give an output like the following, in case of differences:

```
site_id calc_id 0.5 0.1 0.05 0.02 0.01 0.005
=========================== =======
767 -2 0.10593 0.28307 0.37808 0.51918 0.63259 0.76299
767 -1 0.10390 0.27636 0.36955 0.50503 0.61676 0.74079
==========
```

```
poe rms-diff
====== =====
0.5 1.871E-04
0.1 4.253E-04
0.05 5.307E-04
0.02 7.410E-04
0.01 8.856E-04
0.005 0.00106
====== =====
```

This is an example with 6 hazard maps, for poe = .5, .1, .05, .02, .01 and .005 respectively. Here the only site that shows some discrepancy if the site number 767. If that site is in Greenland where nobody lives one can decide that the approximation is good anyway ;-) The engine also report the RMS-differences by considering all the sites, i.e.

\[
\text{rms-diff} = \sqrt{\text{avg}(\text{hmap1} - \text{hmap2})^2}\]

As to be expected, the differences are larger for maps with a smaller poe, i.e. a larger return period. But even in the worst case the RMS difference is only of 1E-3 g, which is not much. The complete
calculation will have more sites, so the RMS difference will likely be even smaller. If you can check the few outlier sites and convince yourself that they are not important, you have succeeded in doubling the speed on your computation. And then you can start to work on the other quadratic and linear parameter and to get an ever bigger speedup!

### 3.3 GMFs for California

We had an user asking for the GMFs of California on 707,920 hazard sites, using the UCERF mean model and an investigation time of 100,000 years. Is this feasible or not? Some back of the envelope calculations suggests that it is unfeasible, but reality can be different.

The relevant parameters are the following:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>707,920 hazard sites</td>
</tr>
<tr>
<td>E</td>
<td>$10^5$ estimated events of magnitude greater than 5.5 in the investigation time of 100,000 years</td>
</tr>
<tr>
<td>B</td>
<td>1 number of branches in the UCERF logic tree</td>
</tr>
<tr>
<td>G</td>
<td>5 number of GSIMS in the GMPE logic tree</td>
</tr>
<tr>
<td>I</td>
<td>6 number of intensity measure types</td>
</tr>
<tr>
<td>S1</td>
<td>13 number of bytes used by the engine to store a single GMV</td>
</tr>
</tbody>
</table>

The maximum size of generated GMFs is

$$N \times E \times B \times G \times I \times S1 = 25 \text{ TB (terabytes)}$$

Storing and sharing 25 TB of data is a big issue, so the problem seems without solution. However, most of the ground motion values are zero, because there is a maximum distance of 300 km and a rupture cannot affect all of the sites. So the size of the GMFs should be less than 25 TB. Moreover, if you want to use such GMFs for a damage analysis, you may want to discard very small shaking that will not cause any damage to your buildings. The engine has a parameter to discard all GMFs below a minimum threshold, the `minimum_intensity` parameter. The higher the threshold, the smaller the size of the GMFs. By playing with that parameter you can reduce the size of the output by orders of magnitudes. Terabytes could easily become gigabytes with a well chosen threshold.

In practice, we were able to run the full 707,920 sites by splitting the sites in 70 tiles and by using a minimum intensity of 0.1 g. This was the limit configuration for our cluster which has 5 machines with 128 GB of RAM each.

The full calculation was completed in only 4 hours because our calculators are highly optimized. The total size of the generated HDF5 files was of 400 GB. This is a lot less than 25 TB, but still too large for sharing purposes.

Another way to reduce the output is to reduce the number of intensity measure types. Currently in your calculations there are 6 of them (PGA, SA(0.1), SA(0.2), SA(0.5), SA(1.0), SA(2.0)) but if you restrict yourself to only PGA the computation and the output will become 6 times smaller.
Also, there are 5 GMPEs: if you restrict yourself to 1 GMPE you gain a factor of 5. Similarly, you can reduce the investigation period from 100,000 year to 10,000 years, thus gaining another order of magnitude. Also, raising the minimum magnitude reduces the number of events significantly.

But the best approach is to be smart. For instance, we know from experience that if the final goal is to estimate the total loss for a given exposure, the correct way to do that is to aggregate the exposure on a smaller number of hazard sites. For instance, instead of the original 707,920 hazard sites we could aggregate on only ~7,000 hazard sites and we would a calculation which is 100 times faster, produces 100 times less GMFs and still produces a good estimate for the total loss.

In short, risk calculations for the mean field UCERF model are routines now, in spite of what the naive expectations could be.
Scenario risk calculations usually do not pose a performance problem, since they involve a single rupture and a limited geography for analysis. Some event-based risk calculations, however, may involve millions of ruptures and exposures spanning entire countries or even larger regions. This section offers some practical tips for running large event based risk calculations, especially ones involving large logic trees, and proposes techniques that might be used to make an intractable calculation tractable.

4.1 Understanding the hazard

Event-based calculations are typically dominated by the hazard component (unless there are lots of assets aggregated on few hazard sites) and therefore the first thing to do is to estimate the size of the hazard, i.e. the number of GMFs that will be produced. Since we are talking about a large calculation, first of all we need reduce it to a size that is guaranteed to run quickly. The simplest way to do that is to reduce the parameters directly affecting the number of ruptures generated, i.e.

- investigation_time
- ses_per_logic_tree_path
- number_of_logic_tree_samples

For instance, if you have $\text{ses}_{\text{per_logic_tree_path}} = 10,000$ reduce it to 10, run the calculation and you will see in the log something like this:

```
[2018-12-16 09:09:57,689 #35263 INFO] Received
{'gmfdata': '752.18 MB', 'hcurves': '224 B', 'indices': '29.42 MB'}
```

The amount of GMFs generated for the reduced calculation is 752.18 MB; and since the calculation has been reduced by a factor of 1,000, the full computation is likely to generate around 750 GB of GMFs. Even if you have sufficient disk space to store this large quantity of GMFs, most likely you will run out of memory. Even if the hazard part of the calculation manages to run to completion, the risk part of the calculation is very likely to fail — managing 750 GB of GMFs is beyond the current capabilities of the engine. Thus, you will have to find ways to reduce the size of the computation.
A good start would be to carefully set the parameters `minimum_magnitude` and `minimum_intensity`:

- `minimum_magnitude` is a scalar or a dictionary keyed by tectonic region; the engine will discard ruptures with magnitudes below the given thresholds
- `minimum_intensity` is a scalar or a dictionary keyed by the intensity measure type; the engine will discard GMFs below the given intensity thresholds

Choosing reasonable cutoff thresholds with these parameters can significantly reduce the size of your computation when there are a large number of small magnitude ruptures or low intensity GMFs being generated, which may have a negligible impact on the damage or losses, and thus could be safely discarded.

### 4.2 Collapsing of branches

When one is not interested so much in the uncertainty around the loss estimates, but more interested simply in the mean estimates, all of the source model branches can be “collapsed” into one branch. Using the collapsed source model should yield the same mean hazard or loss estimates as using the full source model logic tree and then computing the weighted mean of the individual branch results.

Similarly, the GMPE logic tree for each tectonic region can also be “collapsed” into a single branch. Using a single collapsed GMPE for each TRT should also yield the same mean hazard estimates as using the full GMPE logic tree and then computing the weighted mean of the individual branch results. This has become possible through the introduction of AvgGMPE feature in version 3.9.

### 4.3 Splitting the calculation into subregions

If one is interested in propagating the full uncertainty in the source models or ground motion models to the hazard or loss estimates, collapsing the logic trees into a single branch to reduce computational expense is not an option. But before going through the effort of trimming the logic trees, there is an interim step that must be explored, at least for large regions like the entire continental United States. This step is to geographically divide the large region into logical smaller subregions, such that the contribution to the hazard or losses in one subregion from the other subregions is negligibly small or even zero. The effective realizations in each of the subregions will then be much fewer than when trying to cover the entire large region in a single calculation.
4.4 Trimming of the logic-trees or sampling of the branches

Trimming or sampling may be necessary if the following two conditions hold:

1. You are interested in propagating the full uncertainty to the hazard and loss estimates; only the mean or quantile results are not sufficient for your analysis requirements, AND
2. The region of interest cannot be logically divided further as described above; the logic-tree for your chosen region of interest still leads to a very large number of effective realizations.

Sampling is the easier of the two options now. You only need to ensure that you sample a sufficient number of branches to capture the underlying distribution of the hazard or loss results you are interested in. The drawback of random sampling is that you may still need to sample hundreds of branches to capture well the underlying distribution of the results.

Trimming can be much more efficient than sampling, because you pick a few branches such that the distribution of the hazard or loss results obtained from a full-enumeration of these branches is nearly the same as the distribution of the hazard or loss results obtained from a full-enumeration of the entire logic-tree.

4.5 Disabling the propagation of vulnerability uncertainty to losses

The vulnerability functions using continuous distributions (such as the lognormal distribution or beta distribution) to characterize the uncertainty in the loss ratio conditional on the shaking intensity level, specify the mean loss ratios and the corresponding coefficients of variation for a set of intensity levels. They are used to build the so called epsilon matrix within the engine, which is how loss ratios are sampled from the distribution for each asset.

There is clearly a performance penalty associated with the propagation of uncertainty in the vulnerability to losses. The epsilon matrix has to be computed and stored, and then the worker processes have to read it, which involves large quantities of data transfer and memory usage.

Setting

```
ignore_covs = true
```

in your `job.ini` file will result in the engine using just the mean loss ratio conditioned on the shaking intensity and ignoring the uncertainty. This tradeoff of not propagating the vulnerability uncertainty to the loss estimates can lead to a significant boost in performance and tractability.
4.6 The asset loss table

When performing an event based risk calculation the engine keeps in memory a table with the losses for each asset and each event, for each loss type. It is usually impossible to fully store such table, because it is extremely large; for instance, for 1 million assets, 1 million events, 2 loss types and 4 bytes per loss ~8 TB of disk space would be required. It is true that many events will produce zero losses because of the maximum_distance and minimum_intensity parameters, but still the asset loss table is prohibitively large and for many years could not be stored. In engine 3.8 we made a breakthrough: we decided to store a partial asset loss table, obtained by discarding small losses, by leveraging on the fact that loss curves for long enough return periods are dominated by extreme events, i.e. there is no point in saving all the small losses.

To that aim, the engine honors a parameter called minimum_asset_loss which determine how many losses are discarded when storing the asset loss table. The rule is simple: losses below minimum_asset_loss are discarded. By choosing the threshold properly in an ideal world

1. the vast majority of the losses would be discarded, thus making the asset loss table storable;
2. the loss curves would still be nearly identical to the ones without discarding any loss, except for small return periods.

It is the job of the user to verify if 1 and 2 are true in the real world. He can assess that by playing with the minimum_asset_loss in a small calculation, finding a good value for it, and then extending to the large calculation. Clearly it is a matter of compromise: by sacrificing precision it is possible to reduce enormously the size of the stored asset loss table and to make an impossible calculation possible.

Starting from engine 3.11 the asset loss table is stored if the user specifies

aggregate_by = id

in the job.ini file. In large calculations it extremely easy to run out of memory or the make the calculation extremely slow, so we recommend not to store the asset loss table. The functionality is there for the sole purpose of debugging small calculations, for instance to see the effect of the minimum_asset_loss approximation at the asset level.

For large calculations usually one is interested in the aggregate loss table, which contains the losses per event and per aggregation tag (or multi-tag). For instance, the tag occupancy has the three values “Residential”, “Industrial” and “Commercial” and by setting

aggregate_by = occupancy

the engine will store a pandas DataFrame called risk_by_event` with a field `agg_id with 4 possible value: 0 for “Residential”, 1 for “Industrial”, 2 for “Commercial” and 3 for the full aggregation.

NB: if the parameter aggregate_by is not specified, the engine will still compute the aggregate loss table but then the agg_id field will have a single value 0 corresponding to the total portfolio losses.
4.7 The Probable Maximum Loss (PML) and the loss curves

Given an effective investigation time and a return period, the engine is able to compute a PML for each aggregation tag. It does so by using the function `openquake.risklib.scientific.losses_by_period` which takes in input an array of cumulative losses associated to the aggregation tag, a list of or return periods, and the effective investigation time. If there is a single return period the function returns the PML; if there are multiple return periods it returns the loss curve. The two concepts are essentially the same thing, since a loss curve is just an array of PMLs, one for each return period. For instance

```python
>>> from openquake.risklib.scientific import losses_by_period
>>> losses = [3, 2, 3.5, 4, 3, 23, 11, 2, 1, 4, 5, 7, 8, 9, 13, 0]
>>> [PML_500y] = losses_by_period(losses, [500], eff_time=1000)
>>> PML_500y
13.0
```

computes the Probably Maximum Loss at 500 years for the given losses with an effective investigation time of 1000 years. The algorithm works by ordering the losses (suppose there are $E > 1$ losses) generating $E$ time periods $\text{eff_time}/E$, $\text{eff_time}/(E-1)$, ..., $\text{eff_time}/1$ and log-interpolating the loss at the return period. Of course this works only if the condition $\text{eff_time}/E < \text{return_period} < \text{eff_time}$ is respected. In this example there are $E=16$ losses, so the return period must be in the range 62.5 .. 1000 years. If the return period is too small the PML will be zero

```python
>>> losses_by_period(losses, [50], eff_time=1000)
array([0.])
```

while if the return period is outside the investigation range we will refuse the temptation to extrapolate and we will return NaN instead:

```python
>>> losses_by_period(losses, [1500], eff_time=1000)
array([nan])
```

The rules above are the reason while you will see zeros or NaNs in the loss curves generated by the engine sometimes, especially when there are too few events: the valid range will be small and some return periods may slip outside the range.

In order to compute aggregate loss curves you must set the `aggregate_by` parameter in the `job.ini` to one or more tags over which you wish to perform the aggregation. Your exposure must contain the specified tags with values for each asset. We have an example for Nepal in our event based risk demo. The exposure for this demo contains various tags and in particular a geographic tag called NAME1 with values “Mid-Western”, “Far-Western”, “West”, “East”, “Central”, and the `job_eb.ini` file defines
aggregate_by = NAME_1

When running the calculation you will see something like this:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Aggregate Event Losses</td>
</tr>
<tr>
<td>1</td>
<td>Aggregate Loss Curves</td>
</tr>
<tr>
<td>2</td>
<td>Aggregate Loss Curves Statistics</td>
</tr>
<tr>
<td>3</td>
<td>Aggregate Losses</td>
</tr>
<tr>
<td>4</td>
<td>Aggregate Losses Statistics</td>
</tr>
<tr>
<td>5</td>
<td>Average Asset Losses Statistics</td>
</tr>
<tr>
<td>11</td>
<td>Earthquake Ruptures</td>
</tr>
<tr>
<td>6</td>
<td>Events</td>
</tr>
<tr>
<td>7</td>
<td>Full Report</td>
</tr>
<tr>
<td>8</td>
<td>Input Files</td>
</tr>
<tr>
<td>10</td>
<td>Realizations</td>
</tr>
<tr>
<td>12</td>
<td>Total Loss Curves</td>
</tr>
<tr>
<td>13</td>
<td>Total Loss Curves Statistics</td>
</tr>
<tr>
<td>14</td>
<td>Total Losses</td>
</tr>
<tr>
<td>15</td>
<td>Total Losses Statistics</td>
</tr>
</tbody>
</table>

Exporting the Aggregate Loss Curves Statistics output will give you the mean and quantile loss curves in a format like the following one:

<table>
<thead>
<tr>
<th>annual_frequency_of_exceedence,return_period,loss_type,loss_value,loss_ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.00000E-01,2,nonstructural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>5.00000E-01,2,structural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>2.00000E-01,5,nonstructural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>2.00000E-01,5,structural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>1.00000E-01,10,nonstructural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>1.00000E-01,10,structural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>5.00000E-02,20,nonstructural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>5.00000E-02,20,structural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>2.00000E-02,50,nonstructural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>2.00000E-02,50,structural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>1.00000E-02,100,nonstructural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>1.00000E-02,100,structural,0.00000E+00,0.00000E+00</td>
</tr>
<tr>
<td>5.00000E-03,200,nonstructural,1.35279E+05,1.26664E-06</td>
</tr>
<tr>
<td>5.00000E-03,200,structural,2.36901E+05,9.02027E-03</td>
</tr>
<tr>
<td>2.00000E-03,500,nonstructural,1.74918E+06,1.63779E-05</td>
</tr>
<tr>
<td>2.00000E-03,500,structural,2.99670E+06,1.14103E-01</td>
</tr>
<tr>
<td>1.00000E-03,1000,nonstructural,6.92401E+06,6.48308E-05</td>
</tr>
<tr>
<td>1.00000E-03,1000,structural,1.15148E+07,4.38439E-01</td>
</tr>
</tbody>
</table>

If you do not set the aggregate_by parameter you will still able to compute the total loss curve (for the entire portfolio of assets), and the total average losses.
4.8 Aggregating by multiple tags

The engine also supports aggregation by multiple tags. For instance the second event based risk demo (the file `job_eb.ini`) has a line

```plaintext
aggregate_by = NAME_1, taxonomy
```

and it is able to aggregate both on geographic region (`NAME_1`) and on taxonomy. There are 25 possible combinations, that you can see with the command:

```bash
$ oq show agg_keys
```

<table>
<thead>
<tr>
<th>agg_keys</th>
<th>NAME_1_</th>
<th>taxonomy_</th>
<th>NAME_1</th>
<th>taxonomy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Mid-Western</td>
<td>Wood</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Mid-Western</td>
<td>Adobe</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Mid-Western</td>
<td>Stone-Masonry</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>Mid-Western</td>
<td>Unreinforced-Brick-Masonry</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>Mid-Western</td>
<td>Concrete</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Far-Western</td>
<td>Wood</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Far-Western</td>
<td>Adobe</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Far-Western</td>
<td>Stone-Masonry</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Far-Western</td>
<td>Unreinforced-Brick-Masonry</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Far-Western</td>
<td>Concrete</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>West</td>
<td>Wood</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>West</td>
<td>Adobe</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>West</td>
<td>Stone-Masonry</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>West</td>
<td>Unreinforced-Brick-Masonry</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>West</td>
<td>Concrete</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>East</td>
<td>Wood</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>East</td>
<td>Adobe</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>East</td>
<td>Stone-Masonry</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>East</td>
<td>Unreinforced-Brick-Masonry</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>East</td>
<td>Concrete</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Central</td>
<td>Wood</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Central</td>
<td>Adobe</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>Central</td>
<td>Stone-Masonry</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>Central</td>
<td>Unreinforced-Brick-Masonry</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Central</td>
<td>Concrete</td>
<td></td>
</tr>
</tbody>
</table>

The lines in this table are associated to the *generalized aggregation ID*, `agg_id` which is an index going from 0 (meaning aggregate assets with `NAME_1`=*Mid-Western* and `taxonomy`=*Wood*) to 24 (meaning aggregate assets with `NAME_1`=*Mid-Western* and `taxonomy`=*Wood*); moreover `agg_id=25` means full aggregation.

The `agg_id` field enters in `risk_by_event` and in outputs like the aggregate losses; for instance:
$ oq show agg_losses-rlzs

<table>
<thead>
<tr>
<th>agg_id</th>
<th>rlz</th>
<th>loss_type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nonstructural</td>
<td>2_327_008</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>structural</td>
<td>937_852</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>nonstructural</td>
<td>100_199_448</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>structural</td>
<td>157_885_648</td>
</tr>
</tbody>
</table>

The exporter (oq export agg_losses-rlzs) converts back the agg_id to the proper combination of tags; agg_id=25, i.e. full aggregation, is replaced with the string *total*.

By knowing the number of events, the number of aggregation keys and the number of loss types, it is possible to give an upper limit to the size of risk_by_event. In the demo there are 1703 events, 26 aggregation keys and 2 loss types, so risk_by_event contains at most

\[1703 \times 26 \times 2 = 88,556\] rows

This is an upper limit, since some combination can produce zero losses and are not stored, especially if the minimum_asset_loss feature is used. In the case of the demo actually only 20,877 rows are nonzero:

$ oq show risk_by_event

  event_id  agg_id  loss_id     loss  variance

...  

[20877 rows x 5 columns]
CHAPTER FIVE

THE CONCEPT OF EFFECTIVE REALIZATIONS

The management of the logic trees is the most complicated thing in the OpenQuake engine. It is important to manage the logic trees in an efficient way, by avoiding redundant computation and storage, otherwise the engine will not be able to cope with large computations. To that aim, it is essential to understand the concept of *effective realizations*.

The crucial point is that in many calculations it is possible to reduce the full logic tree (the tree of the potential realizations) to a much smaller one (the tree of the effective realizations).

First, it is best to give some terminology.

1. for each source model in the source model logic tree there is potentially a different GMPE logic tree
2. the total number of realizations is the sum of the number of realizations of each GMPE logic tree
3. a GMPE logic tree is *trivial* if it has no tectonic region types with multiple GMPEs
4. a GMPE logic tree is *simple* if it has at most one tectonic region type with multiple GMPEs
5. a GMPE logic tree is *complex* if it has more than one tectonic region type with multiple GMPEs.

Here is an example of trivial GMPE logic tree, in its XML input representation:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nrml xmlns:gml="http://www.opengis.net/gml"
     xmlns="http://openquake.org/xmlns/nrml/0.4">
  <logicTree logicTreeID='lt1'>
    <logicTreeBranchSet uncertaintyType="gmpeModel" branchSetID="bs1">
      <logicTreeBranch branchID="b1">
        <uncertaintyModel>SadighEtAl1997</uncertaintyModel>
        <uncertaintyWeight>1.0</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>  
  </logicTree>
</nrml>
```
The logic tree is trivial since there is a single branch (“b1”) and GMPE (“SadighEtAl1997”) for each tectonic region type (“active shallow crust”). A logic tree with multiple branches can be simple, or even trivial if the tectonic region type with multiple branches is not present in the underlying source model. This is the key to the logic tree reduction concept.

### 5.1 Reduction of the logic tree

The simplest case of logic tree reduction is when the actual sources do not span the full range of tectonic region types in the GMPE logic tree file. This happens very often. For instance, in the SHARE calculation for Europe the GMPE logic tree potentially contains 1280 realizations coming from 7 different tectonic region types:

- **Active_Shallow**: 4 GMPEs (b1, b2, b3, b4)
- **Stable_Shallow**: 5 GMPEs (b21, b22, b23, b24, b25)
- **Shield**: 2 GMPEs (b31, b32)
- **Subduction_Interface**: 4 GMPEs (b41, b42, b43, b44)
- **Subduction_InSlab**: 4 GMPEs (b51, b52, b53, b54)
- **Volcanic**: 1 GMPE (b61)
- **Deep**: 2 GMPEs (b71, b72)

The number of paths in the logic tree is $4 \times 5 \times 2 \times 4 \times 4 \times 1 \times 2 = 1280$, pretty large. We say that there are 1280 potential realizations per source model. However, in most computations, the user will be interested only in a subset of them. For instance, if the sources contributing to your region of interest are only of kind **Active_Shallow** and **Stable_Shallow**, you would consider only $4 \times 5 = 20$ effective realizations instead of 1280. Doing so may improve the computation time and the needed storage by a factor of $1280 / 20 = 64$, which is very significant.

Having motivated the need for the concept of effective realizations, let explain how it works in practice. For sake of simplicity let us consider the simplest possible situation, when there are two tectonic region types in the logic tree file, but the engine contains only sources of one tectonic region type. Let us assume that for the first tectonic region type (T1) the GMPE logic tree file contains 3 GMPEs (A, B, C) and that for the second tectonic region type (T2) the GMPE logic tree file contains 2 GMPEs (D, E). The total number of realizations (assuming full enumeration) is

$$total\_num\_rlzs = 3 \times 2 = 6$$
The realizations are identified by an ordered pair of GMPEs, one for each tectonic region type. Let’s number the realizations, starting from zero, and let’s identify the logic tree path with the notation \(<GMPE\ of\ first\ region\ type>_\langle GMPE\ of\ second\ region\ type\rangle\):

<table>
<thead>
<tr>
<th>#</th>
<th>lt_path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A_D</td>
</tr>
<tr>
<td>1</td>
<td>B_D</td>
</tr>
<tr>
<td>2</td>
<td>C_D</td>
</tr>
<tr>
<td>3</td>
<td>A_E</td>
</tr>
<tr>
<td>4</td>
<td>B_E</td>
</tr>
<tr>
<td>5</td>
<td>C_E</td>
</tr>
</tbody>
</table>

Now assume that the source model does not contain sources of tectonic region type T1, or that such sources are filtered away since they are too distant to have an effect: in such a situation we would expect to have only 2 effective realizations corresponding to the GMPEs in the second tectonic region type. The weight of each effective realizations will be three times the weight of a regular representation, since three different paths in the first tectonic region type will produce exactly the same result. It is not important which GMPE was chosen for the first tectonic region type because there are no sources of kind T1. In such a situation there will be 2 effective realizations coming from a total of 6 total realizations. It means that there will be three copies of the outputs, i.e. three identical outputs for each effective realization.

Starting from engine 3.9 the logic tree reduction must be performed manually, by discarding the irrelevant tectonic region types; in this example the user must add in the job.ini a line

```
discard_trts = Shield, Subduction_Interface, Subduction_InSlab, Volcanic, Deep
```

If not, multiple copies of the same outputs will appear.

### 5.2 How to analyze the logic tree of a calculation without running the calculation

The engine provides some facilities to explore the logic tree of a computation without running it. The command you need is the `oq info` command.

Let’s assume that you have a zip archive called `SHARE.zip` containing the SHARE source model, the SHARE source model logic tree file and the SHARE GMPE logic tree file as provided by the SHARE collaboration, as well as a job.ini file. If you run

```
$ oq info SHARE.zip
```

all the files will be parsed and the full logic tree of the computation will be generated. This is very fast, it runs in exactly 1 minute on my laptop, which is impressive, since the XML of the SHARE source models is larger than 250 MB. Such speed come with a price: all the sources are parsed, but...
they are not filtered, so you will get the complete logic tree, not the one used by your computation, which will likely be reduced because filtering will likely remove some tectonic region types.

The output of the info command will start with a CompositionInfo object, which contains information about the composition of the source model. You will get something like this:

```
<CompositionInfo
b1, area_source_model.xml, trt=[0, 1, 2, 3, 4, 5, 6], weight=0.500:
  →1280 realization(s)
b2, faults_backg_source_model.xml, trt=[7, 8, 9, 10, 11, 12, 13],
  →weight=0.200: 1280 realization(s)
b3, seifa_model.xml, trt=[14, 15, 16, 17, 18, 19], weight=0.300: 640
  →realization(s)>
```

You can read the lines above as follows. The SHARE model is composed by three submodels:

- `area_source_model.xml` contains 7 Tectonic Region Types numbered from 0 to 7 and produces 1280 potential realizations;
- `faults_backg_source_model.xml` contains 7 Tectonic Region Types numbered from 7 to 13 and produces 1280 potential realizations;
- `seifa_model.xml` contains 6 Tectonic Region Types numbered from 14 to 19 and produces 640 potential realizations;

In practice, you want to know if your complete logic tree will be reduced by the filtering, i.e. you want to know the effective realizations, not the potential ones. You can perform that check by using the `--report` flag. This will generate a report with a name like `report_<calc_id>.rst`:

```
$ oq info --report SHARE.zip
...
[2020-04-14 11:11:50 #2493 WARNING] No sources for some TRTs: you
  →should set
discard_trts = Subduction_InSlab, Deep
...
Generated /home/michele/report_2493.rst
```

If you open that file you will find a lot of useful information about the source model, its composition, the number of sources and ruptures and the effective realizations.

Depending on the location of the points and the maximum distance, one or more submodels could be completely filtered out and could produce zero effective realizations, so the reduction effect could be even stronger.

In any case the warning tells the user what she should do in order to remove the duplication and reduce the calculation only to the effective realizations, i.e. which are the TRTs to discard in the `job.ini` file.

28 Chapter 5. The concept of effective realizations
Stating from version 3.10, the OpenQuake engine supports 4 different strategies for sampling the logic tree. They are called, respectively, `early_weights`, `late_weights`, `early_latin`, `late_latin`. Here we will discuss how they work.

First of all, we must point out that logic tree sampling is controlled by three parameters in the `job.ini`:

- `number_of_logic_tree_samples` (default 0, no sampling)
- `sampling_method` (default `early_weights`)
- `random_seed` (default 42)

When sampling is enabled `number_of_logic_tree_samples` is a positive number, equal to the number of branches to be randomly extracted from full logic tree of the calculation. The precise why the random extraction works depends on the sampling method.

**early_weights** With this sampling method, the engine randomly choose branches depending on the weights in the logic tree; having done that, the hazard curve statistics (mean and quantiles) are computed with equal weights.

**late_weights** With this sampling method, the engine randomly choose branches ignoring the weights in the logic tree; however, the hazard curve statistics are computed by taking into account the weights.

**early_latin** With this sampling method, the engine randomly choose branches depending on the weights in the logic tree by using an hypercube latin sampling; having done that, the hazard curve statistics are computed with equal weights.

**late_latin** With this sampling method, the engine randomly choose branches ignoring the weights in the logic tree, but still using an hypercube sampling; then, the hazard curve statistics are computed by taking into account the weights.

More precisely, the engine calls something like the function:

```python
openquake.hazardlib.lt.random_sample(
    branchsets, num_samples, seed, sampling_method)
```
You are invited to play with it; in general the latin sampling produces samples much closer to the expected weights even with few samples. Here in an example with two branchsets with weights [.4, .6] and [.2, .3, .5] respectively.

```python
>>> bsets = [[('X', .4), ('Y', .6)], [('A', .2), ('B', .3), ('C', .5)]]
```

With 100 samples one would expect to get the path XA 8 times, XB 12 times, XC 20 times, YA 12 times, YB 18 times, YC 30 times. Instead we get:

```python
>>> paths = random_sample(bsets, 100, 42, 'early_weights')
>>> collections.Counter(paths)
Counter({'YC': 26, 'XC': 24, 'YB': 17, 'XA': 13, 'YA': 10, 'XB': 10})
```

```python
>>> paths = random_sample(bsets, 100, 42, 'late_weights')
>>> collections.Counter(paths)
Counter({'XA': 20, 'YA': 18, 'XB': 17, 'XC': 15, 'YB': 15, 'YC': 15})
```

```python
>>> paths = random_sample(bsets, 100, 42, 'early_latin')
>>> collections.Counter(paths)
Counter({'YC': 31, 'XC': 19, 'YB': 17, 'XB': 13, 'YA': 12, 'XA': 8})
```

```python
>>> paths = random_sample(bsets, 100, 45, 'late_latin')
>>> collections.Counter(paths)
Counter({'YC': 18, 'XA': 18, 'XC': 16, 'YA': 16, 'XB': 16, 'YB': 16})
```
RUPTURE SAMPLING: HOW DOES IT WORK?

In this section we will explain how the sampling of ruptures in event based calculations works, at least for the case of poissonian sources. As an example, we will consider the following point source:

```python
>>> from openquake.hazardlib import nrml
>>> src = nrml.get('''
... <pointSource id="1" name="Point Source"
...   tectonicRegion="Active Shallow Crust">
...   <pointGeometry>
...     <gml:Point><gml:pos>179.5 0</gml:pos></gml:Point>
...     <upperSeismoDepth>0</upperSeismoDepth>
...     <lowerSeismoDepth>10</lowerSeismoDepth>
...   </pointGeometry>
...   <magScaleRel>WC1994</magScaleRel>
...   <ruptAspectRatio>1.5</ruptAspectRatio>
...   <ruptGutenbergRichterMFD aValue="3" bValue="1" minMag="5"
...     maxMag="7"/>
...   <nodalPlaneDist>
...     <nodalPlane dip="30" probability="1" strike="45" rake="90">
...     </nodalPlane>
...   </nodalPlaneDist>
...   <hypoDepthDist>
...     <hypoDepth depth="4" probability="1"/>
...   </hypoDepthDist>
... </pointSource>''', investigation_time=1, width_of_mfd_bin=1.0)
```

The source here is particularly simple, with only one seismogenic depth and one nodal plane. It generates two ruptures, because with a `width_of_mfd_bin` of 1 there are only two magnitudes in the range from 5 to 7:

```python
>>> [(mag1, rate1), (mag2, rate2)] = src.get_annual_occurrence_rates()
>>> mag1
5.5
>>> mag2
31
```
The occurrence rates are respectively 0.009 and 0.0009. So, if we set the number of stochastic event sets to 1,000,000

```python
>>> num_ses = 1_000_000
```

we would expect the first rupture (the one with magnitude 5.5) to occur around 9000 times and the second rupture (the one with magnitude 6.5) to occur around 900 times. Clearly the exact numbers will depend on the stochastic seed; if we set

```python
>>> import numpy.random
>>> numpy.random.seed(42)
```

then we will have (for investigation_time=1)

```python
>>> numpy.random.poisson(rate1 * num_ses * 1)
8966
>>> numpy.random.poisson(rate2 * num_ses * 1)
921
```

These are the number of occurrences of each rupture in the effective investigation time, i.e. the investigation time multiplied by the number of stochastic event sets and the number of realizations (here we assumed 1 realization).

The total number of events generated by the source will be

\[
\text{number_of_events} = \sum (\text{n_occ for each rupture})
\]

i.e. 8966 + 921 = 9887, with ~91% of the events associated to the first rupture and ~9% of the events associated to the second rupture.

Since the details of the seed algorithm changes at each release of the engine, if you run an event based calculation with the same parameters you will not get exactly the same number of events, but something close. After running the calculation inside the datastore, in the ruptures dataset you will find the two ruptures, their occurrence rates and their integer number of occurrences (n_occ).

If the effective investigation time is large enough the relation

\[
\text{n_occ} \sim \text{occurrence_rate} \times \text{eff_investigation_time}
\]

will hold. If the effective investigation time is not large enough, or the occurrence rate is extremely small, then there will be big differences between the expected number of occurrences and n_occ, as well as a strong seed dependency.

It is important to notice than in order to determine the effective investigation time the engine takes into account also the logic tree and the correct formula to use is

\[
\text{eff_investigation_time} = \text{investigation_time} \times \text{num_ses} \times \text{num_rlzs}
\]

where num_rlzs is the number of realizations in the logic tree.
Just to be concrete, if you run a calculation with the same parameters as described before, but with two GMPEs instead of one (and number_of_logic_tree_samples=0), then the total number of paths admitted by the logic tree will be 2 and you should expect to get about twice the number of occurrences for each rupture.

Users wanting to know the nitty-gritty details should look at the code, inside hazardlib/source/base.py, in the method `src.sample_ruptures(eff_num_ses, ses_seed)`. 
EXTRA TIPS SPECIFIC TO EVENT BASED CALCULATIONS

Event based calculations differ from classical calculations because they produce visible ruptures, which can be exported and made accessible to the user. In classical calculations, instead, the underlying ruptures only live in memory and are normally not saved in the datastore, nor are exportable. The limitation is fundamentally a technical one: in the case of an event based calculation only a small fraction of the ruptures contained in a source are actually generated, so it is possible to store them. In a classical calculation all ruptures are generated and there are so many millions of them that it is impractical to save them, unless there are very few sites. For this reason they live in memory, they are used to produce the hazard curves and immediately discarded right after. The exception if for the case of few sites, i.e. if the number of sites is less than the parameter max_sites_disagg which by default is 10.

8.1 Sampling of the logic tree

There are real life examples of very large logic trees, like the model for South Africa which features 3,194,799,993,706,229,268,480 branches. In such situations it is impossible to perform a computation with full enumeration. However, the engine allows to sample the branches of the complete logic tree. More precisely, for each branch sampled from the source model logic tree, a branch of the GMPE logic tree is chosen randomly, by taking into account the weights in the GMPE logic tree file.

It should be noticed that even if source model path is sampled several times, the model is parsed and sent to the workers only once. In particular if there is a single source model (like for South America) and number_of_logic_tree_samples =100, we generate effectively 1 source model realization and not 100 equivalent source model realizations, as we did in past (actually in the engine version 1.3). The engine keeps track of how many times a model has been sampled (say Ns) and in the event based case it produce ruptures (with different seeds) by calling the appropriate hazardlib function Ns times. This is done inside the worker nodes. In the classical case, all the ruptures are identical and there are no seeds, so the computation is done only once, in an efficient way.
8.2 Convergency of the GMFs for non-trivial logic trees

In theory, the hazard curves produced by an event based calculation should converge to the curves produced by an equivalent classical calculation. In practice, if the parameters `number_of_logic_tree_samples` and `ses_per_logic_tree_path` (the product of them is the relevant one) are not large enough they may be different. The engine is able to compare the mean hazard curves and to see how well they converge. This is done automatically if the option `mean_hazard_curves = true` is set. Here is an example of how to generate and plot the curves for one of our QA tests (a case with bad convergence was chosen on purpose):

```bash
$ oq engine --run event_based/case_7/job.ini
WARNING:root:Relative difference with the classical mean curves for IMT=SA(0.1): 51%
WARNING:root:Relative difference with the classical mean curves for IMT=PGA: 49%
$s oq plot /tmp/cl/hazard.pik /tmp/hazard.pik --sites=0,1,2
```

The relative difference between the classical and event based curves is computed by computing the relative difference between each point of the curves for each curve, and by taking the maximum, at least for probabilities of exceedence larger than 1% (for low values of the probability the convergency may be bad). For the details I suggest you to look at the code.
CHAPTER NINE

THE CONCEPT OF “MEAN” GROUND MOTION FIELD

The engine has at least three different kinds of mean ground motion field, computed differently and used in different situations:

1. Mean ground motion field by GMPE, used to reduce disk space and make risk calculations faster.

2. Mean ground motion field by event, used for debugging/plotting purposes.

3. Single-rupture hazardlib mean ground motion field, used for analysis/plotting purposes.

9.1 Mean ground motion field by GMPE

This is the most useful concept for people doing risk calculations. To be concrete, suppose you are running a scenario_risk calculation on a region where you have a very fine site model (say at 1 km resolution) and a sophisticated hazard model (say with 16 different GMPEs): then you can easily end up with a pretty large calculation. For instance one of our users was doing such a calculation with an exposure of 1.2 million assets, 50,000+ hazard sites, 5 intensity measure levels and 1000 simulations, corresponding to 16,000 events given that there are 16 GMPEs. Given that each ground motion value needs 4 bytes to be stored as a 32 bit float, the math tells us that such calculation will generate 50000 x 16000 x 5 x 4 ~ 15 GB of data (it could be a bit less by using the minimum_intensity feature, but you get the order of magnitude). This is very little for the engine that can store such an amount of data in less than 1 minute, but it is a huge amount of data for a database. If you a (re)insurance company and your workflow requires ingesting the GMFs in a database to compute the financial losses, that’s a big issue. The engine could compute the hazard in just an hour, but the risk part could easily take 8 days. This is a no-go for most companies. They have deadlines and cannot way 8 days to perform a single analysis. At the end they are interested only in the mean losses, so they would like to have a single effective mean field producing something close to the mean losses that more correctly would be obtained by considering all 16 realizations. With a single effective realization the data storage would drop under 1 GB and more significantly the financial model software would complete the calculation in 12 hours instead of 8 days, something a lot more reasonable.
For this kind of situations hazardlib provides an `AvgGMPE` class, that allows to replace a set of GMPEs with a single effective GMPE. More specifically, the method `AvgGMPE.get_means_and_stddevs` calls the methods `.get_means_and_stddevs` on the underlying GMPEs and performs a weighted average of the means and a weighted average of the variances using the usual formulas:

\[
\mu = \sum_i w_i \mu_i \\
\sigma^2 = \sum_i w_i (\sigma_i)^2
\]

where the weights sum up to 1. It is up to the user to check how big is the difference in the risk between the complete calculation and the mean field calculation. A factor of 2 discrepancies would not be surprising, but we have also seen situations where there is no difference within the uncertainty due to the random seed choice.

### 9.2 Mean ground motion field by event

Using the `AvgGMPE` trick does not solve the issue of visualizing the ground motion fields, since for each site there are still 1000 events. A plotting tool has still to download 1 GB of data and then one has to decide which event to plot. The situation is the same if you are doing a sensitivity analysis, i.e. you are changing some parameter (it could be a parameter of the underlying rupture, or even the random seed) and you are studying how the ground motion fields change. It is hard to compare two sets of data of 1 GB each. Instead, it is a lot easier to define a “mean” ground motion field obtained by averaging on the events and then compare the mean fields of the two calculations: if they are very different, it is clear that the calculation is very sensitive to the parameter being studied. Still, the tool performing the comparison will need to consider 1000 times less data and will be 1000 times faster, also downloading 1000 times less data from the remote server where the calculation has been performed.

For this kind of analysis the engine provides an internal output `avg_gmf` that can be plotted with the command `oq plot avg_gmf <calc_id>`. It is also possible to compare two calculations with the command

\$ oq compare avg_gmf imt <calc1> <calc2>

Since `avg_gmf` is meant for internal usage and for debugging it is not exported by default and it is not visible in the WebUI. It is also not guaranteed to stay the same across engine versions. It is available starting from version 3.11. It should be noted that, consistently with how the `AvgGMPE` works, the `avg_gmf` output is computed in log space, i.e. it is geometric mean, not the usual mean. If the distribution was exactly lognormal that would also coincide with the median field.

However, you should remember that in order to reduce the data transfer and to save disk space the engine discards ground motion values below a certain minimum intensity, determined explicitly by the user or inferred from the vulnerability functions when performing a risk calculation: there is no point in considering ground motion values below the minimum in the vulnerability functions, since they would generate zero losses. Discarding the values below the threshold breaks the log normal distribution.
To be concrete, consider a case with a single site, and single intensity measure type (PGA) and a minimum_intensity of 0.05g. Suppose there are 1000 simulations and that you have a normal distribution of the logarithms with μ=2 and σ=.5; then the ground motion values that you could obtain would be as follows:

```python
>>> import numpy
>>> numpy.random.seed(42) # fix the seed
>>> gmvs = numpy.random.lognormal(mean=-2.0, sigma=.5, size=1000)
```

As expected, the variability of the values is rather large, spanning more than one order of magnitude:

```python
>>> gmvs.min(), numpy.median(gmvs), gmvs.max()
(0.026765710489091852, 0.1370582013790309, 0.9290114132955762)
```

Also mean and standard deviation of the logarithms are very close to the expected values μ=2 and σ=.5:

```python
>>> numpy.log(gmvs).mean()
-1.9903339720888376
>>> numpy.log(gmvs).std()
0.4893631038736771
```

The geometric mean of the values (i.e. the exponential of the mean of the logarithms) is very close to the median, as expected for a lognormal distribution:

```python
>>> numpy.exp(numpy.log(gmvs).mean())
0.13664978061122787
```

All these properties are broken when the ground motion values are truncated below the minimum_intensity:

```python
>>> gmvs[gmvs < .05] = .05
>>> numpy.log(gmvs).mean()
-1.9876078473466177
>>> numpy.log(gmvs).std()
0.4828063046779523
>>> numpy.exp(numpy.log(gmvs).mean())
0.13702281319482504
```

In this case the difference is minor, but if the number of simulations is small and/or the σ is large the mean and standard deviation obtained from the logarithms of the ground motion fields could be quite different from the expected ones.

Finally, it should be noticed that the geometric mean can be orders of magnitude different from the usual mean and it is purely a coincidence that in this case they are close (~0.137 vs ~0.155).
9.3 Single-rupture estimated median ground motion field

The mean ground motion field by event discussed above is an \textit{a posteriori} output: \textit{after} performing the calculation, some statistics are performed on the stored ground motion fields. However, in the case of a single rupture it is possible to estimate the geometric mean and the geometric standard deviation \textit{a priori}, using hazardlib and without performing a full calculation. It is enough to instantiate the rupture, the site collection and the GMPE (that can be an \texttt{AvgGMPE} in the case of multiple GMPEs‘) and to call directly the method \texttt{.get_mean_and_stddevs}. That is easy and nice but it should be noticed that it comes with some limitation:

1. it only works when there is a single rupture
2. you have to manage the \texttt{minimum_intensity} manually if you want to compare with a concrete engine output
3. it is good for estimates, it gives you the theoretical ground ground motion field but not the ones concretely generated by the engine fixed a specific seed

It should also be noticed that there is a shortcut to compute the single-rupture hazardlib “mean” ground motion field without writing any code; just set in your \texttt{job.ini} the following values:

\begin{verbatim}
truncation_level = 0
ground_motion_fields = 1
\end{verbatim}

Setting \texttt{truncation_level = 0} effectively replaces the lognormal distribution with a delta function, so the generated ground motion fields will be all equal, with the same value for all events: this is why you can set \texttt{ground_motion_fields = 1}, since you would just waste time and space by generating multiple copies.

Finally let’s warn again on the term hazardlib “mean” ground motion field: in log space it is truly a mean, but in terms of the original GMFs it is a geometric mean - which is the same as the median since the distribution is lognormal - so you can also call this the hazardlib \textit{median} ground motion field.
SITE-SPECIFIC CLASSICAL CALCULATIONS

We plan to introduce special optimizations for site-specific classical calculations. Preliminary investigations show that the GMPEs could be made 30x faster by parallelizing on the ruptures. However the work is currently stopped for lack of funding. Please get in contact with us if you are willing to sponsor work in this direction.
The engine has a relatively large set of predefined outputs, that you can get in various formats, like CSV, XML or HDF5. They are all documented in the manual and they are the recommended way of interacting with the engine, if you are not tech-savvy.

However, sometimes you must be tech-savvy: for instance if you want to post-process hundreds of GB of ground motion fields produced by an event based calculation, you should not use the CSV output, at least if you care about efficiency. To manage this case (huge amounts of data) there a specific solution, which is also able to manage the case of data lacking a predefined exporter: the Extractor API.

There are actually two different kind of extractors: the simple Extractor, which is meant to manage large data sets (say > 100 MB) and the WebExtractor, which is able to interact with the WebAPI and to extract data from a remote machine. The WebExtractor is nice, but cannot be used for large amount of data for various reasons; in particular, unless your Internet connection is ultra-fast, downloading GBs of data will probably send the web request in timeout, causing it to fail. Even if there is no timeout, the WebAPI will block, everything will be slow, the memory occupation and disk space will go up, and at certain moment something will fail.

The WebExtractor is meant for small to medium outputs, things like the mean hazard maps - an hazard map containing 100,000 points and 3 PoEs requires only 1.1 MB of data at 4 bytes per point. Mean hazard curves or mean average losses in risk calculation are still small enough for the WebExtractor. But if you want to extract all of the realizations you must go with the simple Extractor: in that case your postprocessing script must run in the remote machine, since it requires direct access to the datastore.

Here is an example of usage of the Extractor to retrieve mean hazard curves:

```python
>>> from openquake.calculators.extract import Extractor
>>> calc_id = 42  # for example
>>> extractor = Extractor(calc_id)
>>> obj = extractor.get('hcurves?kind=mean&imt=PGA')  # returns an
   ~ArrayWrapper
>>> obj.array.shape  # an example with 10,000 sites and 20 levels per
   ~PGA
   (10000, 20)
```
Here is an example of using the WebExtractor to retrieve hazard maps. Here we assumes that there is available in a remote machine where there is a WebAPI server running, a.k.a. the Engine Server. The first thing to is to set up the credentials to access the WebAPI. There are two cases:

1. you have a production installation of the engine in /opt
2. you have a development installation of the engine in a virtualenv

In both case you need to create a file called openquake.cfg with the following format:

```
[webapi]
server = http(s)://the-url-of-the-server(:port)
username = my-username
password = my-password
```

username and password can be left empty if the authentication is not enabled in the server, which is the recommended way, if the server is in your own secure LAN. Otherwise you must set the right credentials. The difference between case 1 and case 2 is in where to put the openquake.cfg file: if you have a production installation, put it in your $HOME, if you have a development installation, put it in your virtualenv directory.

The usage then is the same as the regular extractor:

```
>>> from openquake.calculators.extract import WebExtractor
>>> extractor = WebExtractor(calc_id)
>>> obj = extractor.get('hmaps?kind=mean&imt=PGA')  # returns an
ArrayWrapper
>>> obj.array.shape  # an example with 10,000 sites and 4 PoEs
(10000, 4)
>>> extractor.close()
```

If you do not want to put your credentials in the openquake.cfg file, you can do so, but then you need to pass them explicitly to the WebExtractor:

```
>>> extractor = WebExtractor(calc_id, server, username, password)
```

### 11.1 Plotting

The (Web)Extractor is used in the `oq plot` command: by configuring openquake.cfg it is possible to plot things like hazard curves, hazard maps and uniform hazard spectra for remote (or local) calculations. Here are three examples of use:
$ oq plot 'hcurves?kind=mean&imt=PGA&site_id=0' <calc_id>
$ oq plot 'hmaps?kind=mean&imt=PGA' <calc_id>
$ oq plot 'uhs?kind=mean&site_id=0' <calc_id>

The `site_id` is optional; if missing only the first site (`site_id=0`) will be plotted. If you want to plot all the realizations you can do:

$ oq plot 'hcurves?kind=rlzs&imt=PGA' <calc_id>

If you want to plot all statistics you can do:

$ oq plot 'hcurves?kind=stats&imt=PGA' <calc_id>

It is also possible to combine plots. For instance if you want to plot all realizations and also the mean the command to give is:

$ oq plot 'hcurves?kind=rlzs&kind=mean&imt=PGA' <calc_id>

If you want to plot the mean and the median the command is:

$ oq plot 'hcurves?kind=quantile-0.5&kind=mean&imt=PGA' <calc_id>

assuming the median (i.e. `quantile-0.5`) is available in the calculation. If you want to compare say `rlz-0` with `rlz-2` and `rlz-5` you can just just say so:

$ oq plot 'hcurves?kind=rlz-0&kind=rlz-2&kind=rlz-5&imt=PGA' <calc_id>

You can combine as many kinds of curves as you want. Clearly if your are specifying a kind that is not available you will get an error.

### 11.2 Extracting ruptures

Here is an example for the event based demo:

$ cd oq-engine/demos/hazard/EventBasedPSHA/
$ oq engine --run job.ini
$ oq shell
IPython shell with a global object "o"
In [1]: from openquake.calculators.extract import Extractor
In [2]: extractor = Extractor(calc_id=-1)
In [3]: aw = extractor.get('rupture_info?min_mag=5')
In [4]: aw
Out[4]: <ArrayWrapper(1511,)>  
In [5]: aw.array
Out[5]:

---

11.2 Extracting ruptures
array([  0, 1, 5.05, 0.08456118, 0.15503392, 5., b'Active Shallow\nCrust', 0.0000000e+00, 90. , 0.),
      (  1, 1, 5.05, 0.08456119, 0.15503392, 5., b'Active Shallow\nCrust', 4.4999969e+01, 90. , 0.),
      (  2, 1, 5.05, 0.08456118, 0.15503392, 5., b'Active Shallow\nCrust', 3.5999997e+02, 49.999985, 0.),
...,
      (1508, 2, 6.15, 0.26448786, -0.7442877 , 5., b'Active Shallow\nCrust', 0.0000000e+00, 90. , 0.),
      (1509, 1, 6.15, 0.26448786, -0.74428767, 5., b'Active Shallow\nCrust', 2.2499924e+02, 50.000004, 0.),
      (1510, 1, 6.85, 0.26448786, -0.74428767, 5., b'Active Shallow\nCrust', 4.9094699e-04, 50.000046, 0.)],
dtype=[('rup_id', '<u4'), ('multiplicity', '<u2'), ('mag', '<f4 '),
      ('centroid_lon', '<f4'), ('centroid_lat', '<f4'), ('centroid_depth', '<f4'), ('trt ', 'S50'), ('strike', '<f4'),
      ('dip', '<f4'), ('rake', '<f4')])
In [6]: extractor.close()
If you are a scientist familiar with Pandas, you will be happy to know that it is possible to process the engine outputs with it. Here we will give an example involving hazard curves.

Suppose you ran the AreaSourceClassicalPSHA demo, with calculation ID=42; then you can process the hazard curves as follows:

```python
>>> from openquake.commonlib.datastore import read
>>> dstore = read(42)
>>> df = dstore.read_df('hcurves-stats', index='lvl',
...                     sel=dict(imt='PGA', stat='mean', site_id=0))
```

<table>
<thead>
<tr>
<th>site_id</th>
<th>stat</th>
<th>imt</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>b'mean'</td>
<td>b'PGA'</td>
<td>0.999982</td>
</tr>
<tr>
<td>1</td>
<td>b'mean'</td>
<td>b'PGA'</td>
<td>0.999949</td>
</tr>
<tr>
<td>2</td>
<td>b'mean'</td>
<td>b'PGA'</td>
<td>0.999850</td>
</tr>
<tr>
<td>3</td>
<td>b'mean'</td>
<td>b'PGA'</td>
<td>0.999545</td>
</tr>
<tr>
<td>4</td>
<td>b'mean'</td>
<td>b'PGA'</td>
<td>0.998634</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>44</td>
<td>b'mean'</td>
<td>b'PGA'</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

The dictionary `dict(imt='PGA', stat='mean', site_id=0)` is used to select subsets of the entire dataset: in this case hazard curves for mean PGA for the first site.

If you do not like pandas, or for some reason you prefer plain numpy arrays, you can get a slice of hazard curves by using the `.sel` method:

```python
>>> arr = dstore.sel('hcurves-stats', imt='PGA', stat='mean', site_id=0)
>>> arr.shape  # (num_sites, num_stats, num_imts, num_levels)
(1, 1, 1, 45)
```

Notice that the `.sel` method does not reduce the number of dimensions of the original array (4 in this case), it just reduces the number of elements. It was inspired by a similar functionality in xarray.
12.1 Example: how many events per magnitude?

When analyzing an event based calculation, users are often interested in checking the magnitude-frequency distribution, i.e. to count how many events of a given magnitude present in the stochastic event set for a fixed investigation time and a fixed `ses_per_logic_tree_path`. You can do that with code like the following:

```python
def print_events_by_mag(calc_id):
    # open the DataStore for the current calculation
dstore = datastore.read(calc_id)
    # read the events table as a Pandas dataset indexed by the event ID
events = dstore.read_df('events', 'id')
    # find the magnitude of each event by looking at the 'ruptures' table
    events['mag'] = dstore['ruptures']['mag'][events['rup_id']]
    # group the events by magnitude
    for mag, grp in events.groupby(['mag']):
        print(mag, len(grp))  # number of events per group
```

If you want to now the number of events per realization and per stochastic event set you can just refine the `groupby` clause, using the list `['mag', 'rlz_id', 'ses_id']` instead of simply `['mag']`.

Given an event, it is trivial to extract the ground motion field generated by that event, if it has been stored (warning: events producing zero ground motion are not stored). It is enough to read the `gmf_data` table indexed by event ID, i.e. the `eid` field:

```python
>>> eid = 20  # consider event with ID 20
>>> gmf_data = dstore.read_df('gmf_data', index='eid')  # engine>3.11
>>> gmf_data.loc[eid]  # first site
   sid     gmv_0
eid
20      93  0.113241
20      102  0.114756
20      121  0.242828
20      142  0.111506
```

The `gmv_0` refers to the first IMT; here I have shown an example with a single IMT, in presence of multiple IMTs you would see multiple columns `gmv_0`, `gmv_1`, `gmv_2`, .... The `sid` column refers to the site ID.

As a following step, you can compute the hazard curves at each site from the ground motion values by using the function `gmvs_to_poes`, available since engine 3.10:

```python
>>> from openquake.commonlib.calc import gmvs_to_poes
>>> gmf_data = dstore.read_df('gmf_data', index='sid')
>>> df = gmf_data.loc[0]  # first site
```
```python
>>> gmvs = [df[col].to_numpy() for col in df.columns
...   if col.startswith('gmv_')]  # list of M arrays
>>> oq = dstore['oqparam']   # calculation parameters
>>> poes = gmvs_to_poes(gmvs, oq.imtls, oq.ses_per_logic_tree_path)
```

This will return an array of shape (M, L) where M is the number of intensity measure types and L the number of levels per IMT. This works when there is a single realization; in presence of multiple realizations one has to collect together set of values corresponding to the same realization (this can be done by using the relation event_id -> rlz_id) and apply gmvs_to_poes to each set.

NB: another quantity one may want to compute is the average ground motion field, normally for plotting purposes. In that case special care must be taken in the presence of zero events, i.e. events producing a zero ground motion value (or below the minimum_intensity); since such values are not stored you have to enlarge the gmvs arrays with the missing zeros, the number of which can be determined from the events table for each realization. The engine is able to compute the avg_gmf correctly, however, since it is an expensive operation, it is done only for small calculations.
Most of the Ground Motion Prediction Equations (GMPEs) in hazardlib are classes that can be instantiated without arguments. However, there is now a growing number of exceptions. Here I will describe some of the parametric GMPEs we have, as well as give some guidance for authors wanting to implement a parametric GMPE.

13.1 Signature of a GMPE class

The more robust way to define parametric GMPEs is to use a **kwargs signature (robust against subclassing):

```python
from openquake.hazardlib.gsim.base import GMPE

class MyGMPE(GMPE):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # doing some initialization here
```

The call to `super().__init__` will set a `self.kwargs` attribute and perform a few checks, like raising a warning if the GMPE is experimental. In absence of parameters `self.kwargs` is the empty dictionary, but in general it is non-empty and it can be arbitrarily nested, with only one limitation: it must be a `dictionary of literal Python objects` so that it admits a TOML representation.

TOML is a simple format similar to the .ini format but hierarchical (see https://github.com/toml-lang/toml#user-content-example). It is used by lots of people in the IT world, not only in Python. The advantage of TOML is that it is a lot more readable than JSON and XML and simpler than YAML: moreover, it is perfect for serializing into text literal Python objects like dictionaries and lists. The serialization feature is essential for the engine since the GMPEs are read from the GMPE logic tree file which is a text file, and because the GMPEs are saved into the datastore as text, in the dataset `full_lt/gsim_lt`.

The examples below will clarify how it works.
13.2 GMPETable

Historically, the first parametric GMPE was the GMPETable, introduced many years ago to support the Canada model. The GMPETable class has a single parameter, called `gmpe_table`, which is a (relative) pathname to an .hdf5 file with a fixed format, containing a tabular representation of the GMPE, numeric rather than analytic.

You can find an example of use of GMPETables in the test openquake/qa_tests_data/case_18, which contains three tables in its logic tree:

```xml
<logicTreeBranch branchID="b11">
  <uncertaintyModel>
    [GMPETable]
    gmpe_table = "Wcrust_low_rhypo.hdf5"
  </uncertaintyModel>
  <uncertaintyWeight>0.16</uncertaintyWeight>
</logicTreeBranch>

<logicTreeBranch branchID="b12">
  <uncertaintyModel>
    [GMPETable]
    gmpe_table = "Wcrust_med_rhypo.hdf5"
  </uncertaintyModel>
  <uncertaintyWeight>0.68</uncertaintyWeight>
</logicTreeBranch>

<logicTreeBranch branchID="b13">
  <uncertaintyModel>
    [GMPETable]
    gmpe_table = "Wcrust_high_rhypo.hdf5"
  </uncertaintyModel>
  <uncertaintyWeight>0.16</uncertaintyWeight>
</logicTreeBranch>
```

As you see, the TOML format is used inside the `uncertaintyModel` tag; the text:

```toml
[GMPETable]
gmpe_table = "Wcrust_low_rhypo.hdf5"
```

is automatically translated into a dictionary `{ 'GMPETable': { 'gmpe_table': "Wcrust_low_rhypo.hdf5" } }` and the `.kwargs` dictionary passed to the GMPE class is simply

```python
{'gmpe_table': "Wcrust_low_rhypo.hdf5"}
```

NB: you may see around old GMPE logic files using a different syntax, without TOML:

```xml
<logicTreeBranch branchID="b11">
  <uncertaintyModel gmpe_table="Wcrust_low_rhypo.hdf5">
```

Chapter 13. Parametric GMPEs
This is a legacy syntax, which is still supported and will likely be supported forever, but we recom-
mend to use the new TOML-based syntax, which is more general. The old syntax has the limitation
of being non-hierarchic, making it impossible to define MultiGMPEs involving parametric GM-
PEs: this is why we switched to TOML.

### 13.3 File-dependent GMPEs

It is possible to define other GMPEs taking one or more filenames as parameters. Everything will
work provided you respect the following rules:

1. there is a naming convention on the file parameters, that must end with the suffix _file or _table
2. the files must be read at GMPE initialization time (i.e. in the __init__ method)
3. they must be read with the GMPE.open method, NOT with the open builtin;
4. in the gsim logic tree file you must use relative path names

The constraint on the argument names makes it possible for the engine to collect all the files
required by the GMPEs; moreover, since the path names are relative, the _oq zip _command can
work making it easy to ship runnable calculations. The engine also stores in the datastore a copy
of all of the required input files. Without the copy, it would not be possible from the datastore to
reconstruct the inputs, thus making it impossible to dump and restore calculations from a server to
a different machine.

The constraint about reading at initialization time makes it possible for the engine to work on a
cluster. The issue is that GMPEs are instantiated in the controller and used in the worker nodes,
which *do not have access to the same filesystem*. If the files are read after instantiation, you will get a file not found error when running on a cluster.

The reason why you cannot use the standard open builtin to read the files is that the engine must be able to read the GMPE inputs from the datastore copies (think of the case when the calc_XXX.hdf5 has been copied to a different machine). In order to do that, there is some magic based on the naming convention. For instance, if your GMPE must read a text file with argument name `text_file` you should write the following code:

```python
class GMPEWithTextFile(GMPE):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        with self.open(kwargs['text_file']) as myfile:
            self.text = myfile.read().decode('utf-8')
```

You should NOT write the following, because it will break the engine, for instance by making it impossible to export the results of a calculation:

```python
class GMPEWithTextFile(GMPE):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        with open(kwargs['text_file']) as myfile:
            self.text = myfile.read()
```

NB: writing

```python
class GMPEWithTextFile(GMPE):
    def __init__(self, text_file):
        super().__init__(text_file=text_file)
        with self.open(text_file) as myfile:
            self.text = myfile.read().decode('utf-8')
```

would work but it is discouraged. It is best to keep the `**kwargs` signature so that the call to `super().__init__(**kwargs)` will work out-of-the-box even if in the future subclasses of `GMPEWithTextFile` with different parameters will appear: this is defensive programming.

### 13.4 MultiGMPE

Another example of parametric GMPE is the MultiGMPE class. A MultiGMPE is a dictionary of GMPEs, keyed by Intensity Measure Type. It is useful in geotechnical applications and in general in any situation where you have GMPEs depending on the IMTs. You can find an example in our test openquake/qa_tests_data/classical/case_1:

```xml
<logicTreeBranch branchID="b1">
  <uncertaintyModel>
```
Here the engine will use the GMPE AkkarBommer2010 for PGA and SadighEtAl1997 for SA(0.1). The .kwargs passed to the MultiGMPE class will have the form:

```python
{'PGA': {'AkkarBommer2010': {}},
'SA(0.1)': {'SadighEtAl1997': {}}}
```

The beauty of the TOML format is that it is hierarchic, so if we wanted to use parametric GMPEs in a MultiGMPE we could. Here is an example using the GMPETable Wcrust_low_rhypo.hdf5 for PGA and Wcrust_med_rhypo.hdf5 for SA(0.1) (the example has no physical meaning, it is just an example):

```xml
<logicTreeBranch branchID="b1">
  <uncertaintyModel>
    [MultiGMPE."PGA".GMPETable]
    gmpe_table = "Wcrust_low_rhypo.hdf5"
    [MultiGMPE."SA(0.1)".GMPETable]
    gmpe_table = "Wcrust_med_rhypo.hdf5"
  </uncertaintyModel>
  <uncertaintyWeight>1.0</uncertaintyWeight>
</logicTreeBranch>
```

### 13.5 GenericGmpeAvgSA

In engine 3.4 we introduced a GMPE that manages a range of spectral accelerations and acts in terms of an average spectral acceleration. You can find an example of use in openquake/qa_tests/data/classical/case_34:

```xml
<logicTreeBranch branchID="b1">
  <uncertaintyModel>
    [GenericGmpeAvgSA]
    gmpe_name = "BooreAtkinson2008"
    avg_periods = [0.5, 1.0, 2.0]
    corr_func = "baker_jayaram"
  </uncertaintyModel>
  <uncertaintyWeight>1.0</uncertaintyWeight>
</logicTreeBranch>
```

As you see, the format is quite convenient when there are several arguments of different types.
here we have two strings (gmpe_name and corr_func) and a list of floats (avg_periods). The dictionary passed to the underlying class will be

```python
{'gmpe_name': 'BooreAtkinson2008',
'avg_periods': [0.5, 1.0, 2.0],
'corr_func': 'baker_jayaram'}
```

### 13.6 ModifiableGMPE

In engine 3.10 we introduced a `ModifiableGMPE` class which is able to modify the behavior of an underlying GMPE. Here is an example of use in the logic tree file:

```xml
<uncertaintyModel>
  [ModifiableGMPE]
  gmpe.AkkarEtAlRjb2014 = {}
  set_between_epsilon.epsilon_tau = 0.5
</uncertaintyModel>
```

Here `set_between_epsilon` is simply shifting the mean with the formula `mean -> mean + epsilon_tau * inter_event`. In the future `ModifiableGMPE` will likely grow more methods. If you want to understand how it works you should look at the source code:

Starting from version 2.5, the OpenQuake Engine is able to manage MultiPointSources, i.e. collections of point sources with specific properties. A MultiPointSource is determined by a mesh of points, a MultiMFD magnitude-frequency-distribution and 9 other parameters:

1. tectonic region type
2. rupture mesh spacing
3. magnitude-scaling relationship
4. rupture aspect ratio
5. temporal occurrence model
6. upper seismogenic depth
7. lower seismogenic depth
8. NodalPlaneDistribution
9. HypoDepthDistribution

The MultiMFD magnitude-frequency-distribution is a collection of regular MFD instances (one per point); in order to instantiate a MultiMFD object you need to pass a string describing the kind of underlying MFD (‘arbitraryMFD’, ‘incrementalMFD’, ‘truncGutenbergRichterMFD’ or ‘YoungsCoppersmithMFD’), a float determining the magnitude bin width and few arrays describing the parameters of the underlying MFDs. For instance, in the case of an ‘incrementalMFD’, the parameters are min_mag and occurRates and a MultiMFD object can be instantiated as follows:

```python
mmfd = MultiMFD('incrementalMFD',
                   size=2,
                   bin_width=[2.0, 2.0],
                   min_mag=[4.5, 4.5],
                   occurRates=[[.3, .1], [.4, .2, .1]])
```

In this example there are two points and two underlying MFDs; the occurrence rates can be different for different MFDs: here the first one has 2 occurrence rates while the second one has 3 occurrence rates.
Having instantiated the `MultiMFD`, a `MultiPointSource` can be instantiated as in this example:

```python
npd = PMF([(0.5, NodalPlane(1, 20, 3)),
          (0.5, NodalPlane(2, 2, 4))])
hd = PMF([(1, 4)])
mesh = Mesh(numpy.array([0, 1]), numpy.array([0.5, 1]))
tom = PoissonTOM(50.)
rms = 2.0
rar = 1.0
usd = 10
lsd = 20
mps = MultiPointSource('mp1', 'multi point source',
                        'Active Shallow Crust',
                        mmfd, rms, PeerMSR(), rar,
                        tom, usd, lsd, npd, hd, mesh)
```

There are two major advantages when using `MultiPointSources`:

1. the space used is a lot less than the space needed for an equivalent set of PointSources (less memory, less data transfer)
2. the XML serialization of a MultiPointSource is a lot more efficient (say 10 times less disk space, and faster read/write times)

At computation time MultiPointSources are split into PointSources and are indistinguishable from those. The serialization is the same as for other source typologies (call `write_source_model(fname, [mps])` or `nrml.to_python(fname, sourceconverter)`) and in XML a `multiPointSource` looks like this:

```xml
<multiPointSource
  id="mp1"
  name="multi point source"
  tectonicRegion="Stable Continental Crust">
  <multiPointGeometry>
    <gml:posList>
      0.0 1.0 0.5 1.0
    </gml:posList>
    <upperSeismoDepth>
      10.0
    </upperSeismoDepth>
    <lowerSeismoDepth>
      20.0
    </lowerSeismoDepth>
  </multiPointGeometry>
  <magScaleRel>
    PeerMSR
  </magScaleRel>
  <ruptAspectRatio>
    1.0
  </ruptAspectRatio>
</multiPointSource>
```
The node `<lengths>` contains the lengths of the occurrence rates, 2 and 3 respectively in this example. This is needed since the serializer writes the occurrence rates sequentially (in this example they are the 5 floats 0.10 0.05 0.40 0.20 0.10) and the information about their grouping would be lost otherwise.

There is an optimization for the case of homogeneous parameters; for instance in this example the `bin_width` and `min_mag` are the same in all points; then it is possible to store these as one-element lists:

```python
mmfd = MultiMFD('incrementalMFD',
    size=2,
    bin_width=[2.0],
    min_mag=[4.5],
    occurRates=[[[.3, .1], [.4, .2, .1]]])
```

This saves memory and data transfer, compared to the version of the code above.

Notice that writing `bin_width=2.0` or `min_mag=4.5` would be an error: the parameters must be
vector objects; if their length is 1 they are treated as homogeneous vectors of size \( size \). If their length is different from 1 it must be equal to \( size \), otherwise you will get an error at instantiation time.
Starting from version 3.2, the OpenQuake Engine is able to manage MultiFaultSources, which are special sources used for UCERF-like models.

Unlike regular sources, MultiFaultSources are not self-consistent: they exist only in relation to a geometryModel which is a list of surfaces (only planarSurfaces and kiteSurfaces are supported), each one with an ID. MultiFaultSources are lists of nonparametric ruptures with values mag, rake, probs_occur and IDs (called sectionIndexes) taken from the geometryModel.

In order to understand MultiFaultSources, the example to study is in the directory qa_tests_data/classical/case_65 in the engine repository. In this example the geometryModel (the file sections.xml) contains two kiteSurfaces with IDs s1 and s2 respectively; there is a single MultiFaultSource with the following content:

The probs_occur must sum up to 1 and the number of probabilities must be uniform across the MultiFaultSource; in this example there always two probs_occur for each rupture.

When the engine reads a multiFaultSource XML file, a MultiFaultSource object is instantiated. Such object cannot be used until the method .create_inverted_index(sections) is called. That is the essential method associating the sectionIndexes to the section objects. The reading happens in three steps:

1. read and instantiate all MultiFaultSources
2. read the geometryFile and instantiate a dictionary of sections section_id -> surface
3. call src.set_sections(sections) for each MultiFaultSource: this will create an attribute .sections pointing to the sections dictionary

Only after the .sections dictionary has been set it is possible to call .iter_ruptures and to instantiate the underlying NonParametricProbabilisticRupture objects. Each rupture will have a surface; if the rupture contains a single section index the surface will be the surface in the geometryModel; if the ruptures contains multiple section indexes the surface will be a MultiSurface built from the corresponding surfaces in the geometryModel.

In our example the first rupture will be associated to the KiteSurface s1, the third rupture to the KiteSurface s2 and the second rupture will be associated to the MultiSurface obtained from the KiteSurfaces s1 and s2.
CHAPTER
SIXTEEN

HOW THE PARALLELIZATION WORKS IN THE ENGINE

The engine builds on top of existing parallelization libraries. Specifically, on a single machine it is based on multiprocessing, which is part of the Python standard library, while on a cluster it is based on the combination celery/rabbitmq, which are well-known and maintained tools.

While the parallelization used by the engine may look trivial in theory (it only addresses embarrassingly parallel problems, not true concurrency) in practice it is far from being so. For instance a crucial feature that the GEM staff requires is the ability to kill (revoke) a running calculation without affecting other calculations that may be running concurrently.

Because of this requirement, we abandoned concurrent.futures, which is also in the standard library, but is lacking the ability to kill the pool of processes, which is instead available in multiprocessing with the Pool.shutdown method. For the same reason, we discarded dask, which is a lot more powerful than celery but lacks the revoke functionality.

Using a real cluster scheduling mechanism (like SLURM) would be of course better, but we do not want to impose on our users a specific cluster architecture. celery/rabbitmq have the advantage of being simple to install and manage. Still, the architecture of the engine parallelization library is such that it is very simple to replace celery/rabbitmq with other parallelization mechanisms: people interested in doing so should just contact us.

Another tricky aspects of parallelizing large scientific calculations is that the amount of data returned can exceed the 4 GB limit of Python pickles: in this case one gets ugly runtime errors. The solution we found is to make it possible to yield partial results from a task: in this way instead of returning say 40 GB from one task, one can yield 40 times partial results of 1 GB each, thus bypassing the 4 GB limit. It is up to the implementor to code the task carefully. In order to do so, it is essential to have in place some monitoring mechanism measuring how much data is returned back from a task, as well as other essential informations like how much memory is allocated and how long it takes to run a task.

To this aim the OpenQuake engine offers a Monitor class (located in openquake.baselib.performance) which is perfectly well integrated with the parallelization framework, so much that every task gets a Monitor object, a context manager that can be used to measure time and memory of specific parts of a task. Moreover, the monitor automatically measures time and memory for the whole task, as well as the size of the returned output (or outputs). Such information
is stored in an HDF5 file that you must pass to the monitor when instantiating it. The engine automatically does that for you by passing the pathname of the datastore.

In OpenQuake a task is just a Python function (or generator) with positional arguments, where the last argument is a Monitor instance. For instance the rupture generator task in an event based calculation is coded more or less like this:

```python
def sample_ruptures(sources, num_samples, monitor):
    ebruptures = []
    for src in sources:
        for rup, n_occ in src.sample_ruptures(num_samples):
            ebr = EBRupture(rup, src.id, grp_id, n_occ)
            ebruptures.append(ebr)
        if len(ebruptures) > MAX_RUPTURES:
            # yield partial result to avoid running out of memory
            yield ebruptures
            ebruptures.clear()
    if ebruptures:
        yield ebruptures
```

If you know that there is no risk of running out of memory and/or passing the pickle limit you can just use a regular function and return a single result instead of yielding partial results. This is the case when computing the hazard curves, because the algorithm is considering one rupture at the time and it is not accumulating ruptures in memory, differently from what happens when sampling the ruptures in event based.

If you have ever coded in celery, you will see that the OpenQuake engine concept of task is different: there is no @task decorator and while at the end engine tasks will become celery tasks this is hidden to the developer. The reason is that we needed a celery-independent abstraction layer to make it possible to use different kinds of parallelization frameworks.

From the point of view of the coder, in the engine there is no difference between a task running on a cluster using celery and a task running locally using multiprocessing.Pool: they are coded the same, but depending on a configuration parameter in openquake.cfg (distribute=celery or distribute=processpool) the engine will treat them differently. You can also set an environment variable OQ_DISTRIBUTE, which takes the precedence over openquake.cfg, to specify which kind of distribution you want to use (celery or processpool): this is mostly used when debugging, when you typically insert a breakpoint in the task and then run the calculation with

```
$ OQ_DISTRIBUTE=no oq run job.ini
```

no is a perfectly valid distribution mechanism in which there is actually no distribution and all the tasks run sequentially in the same core. Having this functionality is invaluable for debugging.

Another tricky bit of real life parallelism in Python is that forking does not play well with the HDF5 library: so in the engine we are using multiprocessing in the spawn mode, not in fork mode: fortunately this feature has become available to us in Python 3 and it made our life a lot happier. Before it was extremely easy to incur unspecified behavior, meaning that reading an HDF5 file
from a forked process could

1. work perfectly well
2. read bogus numbers
3. cause a segmentation fault

and all of the three things could happen unpredictably at any moment, depending on the machine where the calculation was running, the load on the machine, and any kind of environmental circumstances.

Also, while with the newest HDF5 libraries it is possible to use a Single Writer Multiple Reader architecture (SWMR), and we are actually using it - even if it is sometimes tricky to use it correctly - the performance is not always good. So, when it matters, we are using a two files approach which is simple and very effective: we read from one file (with multiple readers) and we write on the other file (with a single writer). This approach bypasses all the limitations of the SWMR mode in HDF5 and did not require a large refactoring of our existing code.

Another tricky point in cluster situations is that rabbitmq is not good at transferring gigabytes of data: it was meant to manage lots of small messages, but here we are perverting it to manage huge messages, i.e. the large arrays coming from a scientific calculations.

Hence, since recent versions of the engine we are no longer returning data from the tasks via celery/rabbitmq: instead, we use zeromq. This is hidden from the user, but internally the engine keeps track of all tasks that were submitted and waits until they send the message that they finished. If one tasks runs out of memory badly and never sends the message that it finished, the engine may hang, waiting for the results of a task that does not exist anymore. You have to be careful. What we did in our cluster is to set some memory limit on the celery user with the cgroups technology, so that an out of memory task normally fails in a clean way with a Python MemoryError, sends the message that it finished and nothing particularly bad happens. Still, in situations of very heavy load the OOM killer may enter in action aggressively and the main calculation may hang: in such cases you need a good sysadmin having put in place some monitor mechanism, so that you can see if the OOM killer entered in action and then you can kill the main process of the calculation by hand. There is not much more that can be done for really huge calculations that stress the hardware as much as possible. You must be prepared for failures.

16.1 How to use openquake.baselib.parallel

Suppose you want to code a character-counting algorithm, which is a textbook exercise in parallel computing and suppose that you want to store information about the performance of the algorithm. Then you should use the OpenQuake Monitor class, as well as the utility openquake.baselib.commonlib.hdf5new that build an empty datastore for you. Having done that, the openquake.baselib.parallel.Starmap class can take care of the parallelization for you as in the following example:
```
import os
import sys
import pathlib
import collections
from openquake.baselib.performance import Monitor
from openquake.baselib.parallel import Starmap
from openquake.commonlib.datastore import hdf5new

def count(text):
c = collections.Counter()
for word in text.split():
c += collections.Counter(word)
return c

def main(dirname):
dname = pathlib.Path(dirname)
with hdf5new() as hdf5:
    monitor = Monitor('count', hdf5)
    iterargs = ((open(dname/fname, encoding='utf-8').read(),)
                for fname in os.listdir(dname)
                if fname.endswith('.rst'))
    c = collections.Counter()
    for counter in Starmap(count, iterargs, monitor):
c += counter
print(c)
print('Performance info stored in', hdf5)

if __name__ == '__main__':
    main(sys.argv[1])
```

The name Starmap was chosen it looks very similar to multiprocessing.Pool.starmap works, the only apparent difference being in the additional monitor argument:

```
pool.starmap(func, iterargs) -> Starmap(func, iterargs, monitor)
```

In reality the Starmap has a few other differences:

1. it does not use the multiprocessing mechanism to returns back the results, it uses zmq instead;
2. thanks to that, it can be extended to generator functions and can yield partial results, thus overcoming the limitations of multiprocessing
3. the Starmap has a .submit method and it is actually more similar to concurrent.futures than to multiprocessing.
Here is how you would write the same example by using `.submit`:

```python
def main(dirname):
    dname = pathlib.Path(dirname)
    with hdf5new() as hdf5:
        smap = Starmap(count, monitor=Monitor('count', hdf5))
        for fname in os.listdir(dname):
            if fname.endswith('.rst'):
                smap.submit(open(dname/fname, encoding='utf-8').read())
        c = collections.Counter()
        for counter in smap:
            c += counter
```

The difference with concurrent.futures is that the Starmap takes care for of all submitted tasks, so you do not need to use something like concurrent.futures.completed, you can just loop on the Starmap object to get the results from the various tasks.

The `.submit` approach is more general: for instance you could define more than one Starmap object at the same time and submit some tasks with a starmap and some others with another starmap: this may help parallelizing complex situations where it is expensive to use a single starmap. However, there is limit on the number of starmaps that can be alive at the same moment.

Moreover the Starmap has a `.shutdown` methods that allows to shutdown the underlying pool.

The idea is to submit the text of each file - here I am considering .rst files, like the ones composing this manual - and then loop over the results of the Starmap. This is very similar to how concurrent.futures works.
CHAPTER

SEVENTEEN

SPECIAL FEATURES OF THE ENGINE

There are few rarely used feature of the engine that are not documented in the manual, since their usage is quite specific. They are documented here.

17.1 The minimum_distance parameter

GMPEs have a range of validity. In particular they may give wrong results for points too close to the rupture. To avoid this problem the engine recognizes a minimum_distance parameter: if it is set, then for distances below the minimum the GMPEs return the value at the minimum distance. This avoids producing huge unphysical values of the hazard for small distances. The minimum distance is somewhat heuristic. It may be useful to experiment with different values of the minimum_distance, to see how the hazard and risk change.

17.2 GMPE logic trees with weighted IMTs

Our Canadian users asked us to implement GMPE logic trees with a different weight per each IMT. For instance you could have a GMPE applicable to PGA with a certain level of uncertainty, to SA(0.1) with another and to SA(1.0) with still another one. The user may want to give an higher weight to the IMTs were the GMPE has a small uncertainty and a lower weight to the IMTs with a large uncertainty. Moreover the GMPE could not be applicable for some period, and in that case the user can assign to it a zero weight, to ignore it. This is useful when you have a logic tree with multiple GMPEs per branchset, some of which are applicable for some IMTs and not for others. Here is an example:

```
<logicTreeBranchSet uncertaintyType="gmpeModel" branchSetID="bs1"
applyToTectonicRegionType="Volcanic">
<logicTreeBranch branchID="BooreEtAl1997GeometricMean">
  <uncertaintyModel>BooreEtAl1997GeometricMean</uncertaintyModel>
  <uncertaintyWeight>0.33</uncertaintyWeight>
  <uncertaintyWeight imt="PGA">0.25</uncertaintyWeight>
  <uncertaintyWeight imt="SA(0.5)">0.5</uncertaintyWeight>
```
Clearly the weights for each IMT must sum up to 1, otherwise the engine will complain. Note that this feature only works for the classical and disaggregation calculators: in the event based case only the default uncertaintyWeight (i.e. the first in the list of weights, the one without imt attribute) would be taken for all IMTs.

17.3 Equivalent Epicenter Distance Approximation

The equivalent epicenter distance approximation (reqv for short) was introduced in engine 3.2 to enable the comparison of the OpenQuake engine with time-honored Fortran codes using the same approximation.

You can enable it in the engine by adding a [reqv] section to the job.ini, like in our example in openquake/qa_tests_data/classical/case_2/job.ini:

```
reqv_hdf5 = {'active shallow crust': 'lookup_asc.hdf5',
             'stable shallow crust': 'lookup_sta.hdf5'}
```
For each tectonic region type to which the approximation should be applied, the user must provide a lookup table in .hdf5 format containing arrays \textit{mags} of shape \( M \), \textit{repi} of shape \( N \) and \textit{reqv} of shape \((M, N)\).

The examples in openquake/qa_tests_data/classical/case_2 will give you the exact format required. \( M \) is the number of magnitudes (in the examples there are 26 magnitudes ranging from 6.05 to 8.55) and \( N \) is the number of epicenter distances (in the examples ranging from 1 km to 1000 km).

Depending on the tectonic region type and rupture magnitude, the engine converts the epicentral distance \textit{repi} into an equivalent distance by looking at the lookup table and use it to determine the \textit{rjb} and \textit{rrup} distances, instead of the regular routines. This means that within this approximation ruptures are treated as pointwise and not rectangular as the engine usually does.

Notice that the equivalent epicenter distance approximation only applies to ruptures coming from \texttt{PointSources/AreaSources/MultiPointSources}, fault sources are untouched.

## 17.4 Ruptures in CSV format

Since engine v3.10 there is a way to serialize ruptures in CSV format. The command to give is:

```
$ oq extract "ruptures?min_mag=<mag>" <calc_id>
```

For instance, assuming there is an event based calculation with ID 42, we can extract the ruptures in the datastore with magnitude larger than 6 with \texttt{oq extract "ruptures?min_mag=6" 42}: this will generate a CSV file. Then it is possible to run scenario calculations starting from that rupture by simply setting

\[
\text{rupture\_model\_file} = \text{ruptures\_min\_mag=6\_42.csv}
\]

in the \texttt{job.ini} file. The format is provisional and may change in the future, but it will stay a CSV with JSON fields. Here is an example for a planar rupture, i.e. a rupture generated by a point source:

```
#,,,,,,,,,,"trts=['Active Shallow Crust']"
seed,mag,rake,lon,lat,dep,multiplicity,trt,kind,mesh,extra
24,5.050000E+00,0.000000E+00,0.08456,0.15503,5.000000E+00,1,Active Shallow Crust,ParametricProbabilisticRupture PlanarSurface,"[[[0.08456, 0.08456, 0.08456, 0.08456]], [[0.13861, 0.17145, 0.13861, 0.17145]], [[3.17413, 3.17413, 6.82587, 6.82587]]],",""occurrence_rate": 4e-05}"
```

The format is meant to support all kind of ruptures, including ruptures generated by simple and complex fault sources, characteristic sources, nonparametric sources and new kind of sources that could be introduced in the engine in the future. The header will be the same for all kind of ruptures that will be stored in the same CSV. Here is description of the fields as they are named now (engine 3.11):
seed  the random seed used to compute the GMFs generated by the rupture
mag  the magnitude of the rupture
rake  the rake angle of the rupture surface in degrees
lon  the longitude of the hypocenter in degrees
lat  the latitude of the hypocenter in degrees
dep  the depth of the hypocenter in km
multiplicity  the number of occurrences of the rupture (i.e. number of events)
trt  the tectonic region type of the rupture; must be consistent with the trts listed in the pre-header of the file
kind  a space-separated string listing the rupture class and the surface class used in the engine
mesh  3 times nested list with lon, lat, dep of the points of the discretized rupture geometry for each underlying surface
extra  extra parameters of the rupture as a JSON dictionary, for instance the rupture occurrence rate

Notice that using a CSV file generated with an old version of the engine is inherently risky: for instance if we changed the ParametricProbabilisticRupture class or the PlanarSurface classes in an incompatible way with the past, then a scenario calculation starting with the CSV would give different results in the new version of the engine. We never changed the rupture classes or the surface classes, but we changed the seed algorithm often, and that too would cause different numbers to be generated (hopefully, statistically consistent). A bug fix or change of logic in the calculator can also change the numbers across engine versions.

17.5 max_sites_disagg

There is a parameter in the job.ini called max_sites_disagg, with a default value of 10. This parameter controls the maximum number of sites on which it is possible to run a disaggregation. If you need to run a disaggregation on a large number of sites you will have to increase that parameter. Notice that there are technical limits: trying to disaggregate 100 sites will likely succeed, trying to disaggregate 100,000 sites will most likely cause your system to go out of memory or out of disk space, and the calculation will be terribly slow. If you have a really large number of sites to disaggregate, you will have to split the calculation and it will be challenging to complete all the subcalculations.

The parameter max_sites_disagg is extremely important not only for disaggregation, but also for classical calculations. Depending on its value and then number of sites (N) your calculation can be in the few sites regime or the many sites regime.
In the **few sites regime** \((N \leq \text{max\_sites\_disagg})\) the engine stores information for each rupture in the model (in particular the distances for each site) and therefore uses more disk space. The problem is mitigated since the engine uses a relatively aggressive strategy to collapse ruptures, but that requires more RAM available.

In the **many sites regime** \((N > \text{max\_sites\_disagg})\) the engine does not store rupture information (otherwise it would immediately run out of disk space, since typical hazard models have tens of millions of ruptures) and uses a much less aggressive strategy to collapse ruptures, which has the advantage of requiring less RAM.

### 17.6 extendModel

Starting from engine 3.9 there is a new feature in the source model logic tree: the ability to define new branches by adding sources to a base model. An example will explain it all:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nrml xmlns:gml="http://www.opengis.net/gml"
      xmlns="http://openquake.org/xmlns/nrml/0.4">
  <logicTree logicTreeID="lt1">
    <logicTreeBranchSet uncertaintyType="sourceModel" branchSetID="bs0">
      <logicTreeBranch branchID="b01">
        <uncertaintyModel>common1.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="b02">
        <uncertaintyModel>common2.xml</uncertaintyModel>
        <uncertaintyWeight>0.4</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>
    <logicTreeBranchSet uncertaintyType="extendModel" applyToBranches="b01" branchSetID="bs1">
      <logicTreeBranch branchID="b11">
        <uncertaintyModel>extra1.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="b12">
        <uncertaintyModel>extra2.xml</uncertaintyModel>
        <uncertaintyWeight>0.4</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>
  </logicTree>
</nrml>
```

In this example there are two base source models, named `common1.xml` and `common2.xml`;
the branchset with uncertaintyType = "extendModel" is telling the engine to generate two effective source models by extending common1.xml first with extra1.xml and then with extra2.xml. If we removed the constraint applyToBranches="b01" then two additional effective source models would be generated by applying extra1.xml and extra2.xml to common2.xml.
CHAPTER
EIGHTEEN

THE POINT SOURCE GRIDDING APPROXIMATION

WARNING: *the point source gridding approximation is used only in classical calculations, not in event based calculations!*

Most hazard calculations are dominated by distributed seismicity, i.e. area sources and multipoint sources that for the engine are just regular point sources. In such situations the parameter governing the performance is the grid spacing: a calculation with a grid spacing of 50 km produces 25 times less ruptures and it is expected to be 25 times faster than a calculation with a grid spacing of 10 km.

The *point source gridding approximation* is a smart way of raising the grid spacing without losing too much precision and without losing too much performance.

The idea is two use two kind of point sources: the original ones and a set of “effective” ones (instances of the class `CollapsedPointSource`) that essentially are the original sources averaged on a larger grid, determined by the parameter `ps_grid_spacing`.

The plot below should give the idea, the points being the original sources and the squares with ~25 sources each being associated to the collapsed sources:
For distant sites it is possible to use the large grid (i.e. the CollapsePointSources) without losing much precision, while for close points the original sources must be used.

The engine uses the parameter `pointsource_distance` to determine when to use the original sources and when to use the collapsed sources.

If the `maximum_distance` has a value of 500 km and the `pointsource_distance` a value of 50 km, then \((50/500)^2 = 1\%\) of the sites will be close and 99\% of the sites will be far. Therefore you will able to use the collapsed sources for 99\% percent of the sites and a huge speedup is to big expected.
HOW THE HAZARD SITES ARE DETERMINED

There are several ways to specify the hazard sites in an engine calculation.

1. The user can specify the sites directly in the job.ini using the sites parameter (e.g. sites = -122.4194, 37.7749, -118.2437, 34.0522, -117.1611, 32.7157). This method is perhaps most useful when the analysis is limited to a handful of sites.

2. Otherwise the user can specify the list of sites in a CSV file (i.e. sites_csv = sites.csv).

3. Otherwise the user can specify a grid via the region and region_grid_spacing parameters.

4. Otherwise the sites can be inferred from the exposure, if any, in two different ways:
   (a) if region_grid_spacing is specified, a grid is implicitly generated from the convex hull of the exposure and used
   (b) otherwise the locations of the assets are used as hazard sites

5. Otherwise the sites can be inferred from the site model file, if any.

It must be noted that the engine rounds longitudes and latitudes to 5 decimal places (or approximately 1 meter spatial resolution), so sites that differ only at the 6th decimal place or beyond will end up being considered as duplicated sites by the engine, and this will be flagged as an error.

Having determined the sites, a SiteCollection object is generated by associating the closest parameters from the site model (if any) or using the global site parameters, if any. If the site model is specified, but the closest site parameters are too distant from the sites, a warning is logged for each site.

There are a number of error situations:

1. If both site model and global site parameters are missing, the engine raises an error.
2. If both site model and global site parameters are specified, the engine raises an error.
3. Specifying both the sites.csv and a grid is an error.
4. Specifying both the sites.csv and a site_model.csv is an error. If you are in such situation you should consider using the command `oq prepare_site_model` to manually prepare a site model on the location of the sites.

5. Having duplicates (i.e. rows with identical lon, lat up to 5 digits) in the site model is an error. If you want to compute the hazard on the locations specified by the site model and not on the exposure locations, you can split the calculation in two files: `job_haz.ini` containing the site model and `job_risk.ini` containing the exposure. Then the risk calculator will find the closest hazard to each asset and use it. However, if the closest hazard is more distant than the `asset_hazard_distance` parameter (default 15 km) an error is raised.
NEW RISK FEATURES

Here we document some new risk features which have not yet made it into the engine manual. These are new features and cannot be considered fully-tested and stable yet.

20.1 Taxonomy mapping

In an ideal world, for every building type represented in the exposure model, there would be a unique matching function in the vulnerability or fragility models. However, often it may not be possible to have a one-to-one mapping of the taxonomy strings in the exposure and those in the vulnerability or fragility models. For cases where the exposure model has richer detail, many taxonomy strings in the exposure would need to be mapped onto a single vulnerability or fragility function. In other cases where building classes in the exposure are more generic and the fragility or vulnerability functions are available for more specific building types, a modeller may wish to assign more than one vulnerability or fragility function to the same building type in the exposure with different weights.

We may encode such information into a `taxonomy_mapping.csv` file like the following:

<table>
<thead>
<tr>
<th>taxonomy</th>
<th>conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wood Type A</td>
<td>Wood</td>
</tr>
<tr>
<td>Wood Type B</td>
<td>Wood</td>
</tr>
<tr>
<td>Wood Type C</td>
<td>Wood</td>
</tr>
</tbody>
</table>

Using an external file is convenient, because we can avoid changing the original exposure. If in the future we will be able to get specific risk functions, then we will just remove the taxonomy mapping. This usage of the taxonomy mapping (use proxies for missing risk functions) is pretty useful, but there is also another usage which is even more interesting.

Consider a situation where there are doubts about the precise composition of the exposure. For instance we may know than in a given geographic region 20% of the building of type “Wood” are of “Wood Type A”, 30% of “Wood Type B” and 50% of “Wood Type C”, corresponding to different risk functions, but do not know building per building what it its precise taxonomy, so we
just use a generic “Wood” taxonomy in the exposure. We may encode the weight information into a `taxonomy_mapping.csv` file like the following:

<table>
<thead>
<tr>
<th>taxonomy</th>
<th>conversion</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wood</td>
<td>Wood Type A</td>
<td>0.2</td>
</tr>
<tr>
<td>Wood</td>
<td>Wood Type B</td>
<td>0.3</td>
</tr>
<tr>
<td>Wood</td>
<td>Wood Type C</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The engine will read this mapping file and when performing the risk calculation will use all three kinds of risk functions to compute a single result with a weighted mean algorithm. The sums of the weights must be 1 for each exposure taxonomy, otherwise the engine will raise an error. In this case the taxonomy mapping file works like a risk logic tree.

Internally both the first usage and the second usage are treated in the same way, since the first usage is a special case of the second when all the weights are equal to 1.

Currently, this taxonomy mapping feature has been tested only for the scenario and event based risk calculators.

### 20.2 Extended consequences

Scenario damage calculations produce the so called asset damage table, i.e. a set of damage distributions (the probability of being in each damage state listed in the fragility functions) per each asset, event and loss type. From the asset damage table it is possible to compute generic consequences by multiplying for known coefficients specified in what are called consequence model. For instance, from the probability of being in the collapsed damage state, one might estimate the number of fatalities, given the right multiplicative coefficient.

By default the engine allows to compute economic losses; in that case the coefficients depends on the taxonomy and the consequence model can be represented as a CSV file like the following:

The first field in the header is the name of a tag in the exposure; in this case it is the taxonomy but it could be any other tag — for instance, for volcanic ash-fall consequences, the roof-type might be more relevant, and for recovery time estimates, the occupancy class might be more relevant. The framework is meant to be used for generic consequences, not necessarily limited to earthquakes, because since version 3.6 the engine provides a multi-hazard risk calculator.

The second field of the header, the consequence, is a string identifying the kind of consequence we are considering. It is important because it is associated to the name of the plugin function to use to compute the consequence.

The other fields in the header are the loss_type and the damage states. For instance the coefficient 0.25 for “moderate” means that the cost to bring a structure in “moderate damage” back to its undamaged state is 25% of the total replacement value of the asset.
20.3 Scenarios from ShakeMaps

Beginning with version 3.1, the engine is able to perform scenario_risk and scenario_damage calculations starting from the GeoJSON feed for ShakeMaps provided by the United States Geological Survey (USGS). Furthermore, starting from version 3.12 it is possible to use ShakeMaps from other sources like the local filesystem or a custom URL.

20.3.1 Running the Calculation

In order to enable this functionality one has to prepare a parent calculation containing the exposure and risk functions for the region of interest, say Peru. To that aim the user will need to write a prepare_job.ini file like this one:

```
[general]
description = Peru - Preloading exposure and vulnerability
calculation_mode = scenario
exposure_file = exposure_model.xml
structural_vulnerability_file = structural_vulnerability_model.xml
```

By running the calculation

```
$ oq engine --run prepare_job.ini
```

The exposure and the risk functions will be imported in the datastore.

This example only includes vulnerability functions for the loss type structural, but one could also have in this preparatory job file the functions for nonstructural components and contents, and occupants, or fragility functions if damage calculations are of interest.

It is essential that each fragility/vulnerability function in the risk model should be conditioned on one of the intensity measure types that are supported by the ShakeMap service – MMI, PGV, PGA, SA(0.3), SA(1.0), and SA(3.0). If your fragility/vulnerability functions involves an intensity measure type which is not supported by the ShakeMap system (for instance SA(0.6)) the calculation will terminate with an error.

Let’s suppose that the calculation ID of this ‘pre’ calculation is 1000. We can now run the risk calculation starting from a ShakeMap. For that, one need a job.ini file like the following:

```
[general]
description = Peru - 2007 M8.0 Pisco earthquake losses
calculation_mode = scenario_risk
number_of_ground_motion_fields = 10
truncation_level = 3
shakemap_id = usp000fjta
spatial_correlation = yes
cross_correlation = yes
```
This example refers to the 2007 Mw8.0 Pisco earthquake in Peru (see https://earthquake.usgs.gov/earthquakes/eventpage/us000fjta#shakemap). The risk can be computed by running the risk job file against the prepared calculation:

```bash
$ oq engine --run job.ini --hc 1000
```

Starting from version 3.12 it is also possible to specify the following sources instead of a `shakemap_id`:

```python
# (1) from local files:
shakemap_uri = {
    "kind": "usgs_xml",
    "grid_url": "relative/path/file.xml",
    "uncertainty_url": "relative/path/file.xml"
}

# (2) from remote files:
shakemap_uri = {
    "kind": "usgs_xml",
    "grid_url": "https://url.to/grid.xml",
    "uncertainty_url": "https://url.to/uncertainty.zip"
}

# (3) both files in a single archive
# containing grid.xml, uncertainty.xml:
shakemap_uri = {
    "kind": "usgs_xml",
    "grid_url": "relative/path/grid.zip"
}
```

While it is also possible to define absolute paths, it is advised not to do so since using absolute paths will make your calculation not portable across different machines.

The files must be valid .xml USGS ShakeMaps (1). One or both files can also be passed as .zip archives containing a single valid xml ShakeMap (2). If both files are in the same .zip, the archived files must be named `grid.xml` and `uncertainty.xml`.

Also starting from version 3.12 it is possible to use ESRI Shapefiles in the same manner as ShakeMaps. Polygons define areas with the same intensity levels and assets/sites will be associated to a polygon if contained by the latter. Sites outside of a polygon will be discarded. Shapefile inputs can be specified similar to ShakeMaps:

```python
shakemap_uri = {
    "kind": "shapefile",
    "fname": "path_to/file.shp"
}
```

It is only necessary to specify one of the available files, and the rest of the files will be expected
to be in the same location. It is also possible to have them contained together in a *.zip file. There are at least a *.shp-main file and a *.dbf-dBASE file required. The record field names, intensity measure types and units all need to be the same as with regular USGS ShakeMaps.

Irrespective of the input, the engine will perform the following operations:

1. download the ShakeMap and convert it into a format suitable for further processing, i.e. a ShakeMaps array with lon, lat fields
2. the ShakeMap array will be associated to the hazard sites in the region covered by the ShakeMap
3. by using the parameters truncation_level and number_of_ground_motion_fields a set of ground motion fields (GMFs) following the truncated Gaussian distribution will be generated and stored in the datastore
4. a regular risk calculation will be performed by using such GMFs and the assets within the region covered by the shakemap.

20.3.2 Correlation

By default the engine tries to compute both the spatial correlation and the cross correlation between different intensity measure types. Please note that if you are using MMI as intensity measure type in your vulnerability model, it is not possible to apply correlations since those are based on physical measures.

For each kind of correlation you have three choices, that you can set in the job.ini, for a total of nine combinations:

```
- spatial_correlation = yes, cross_correlation = yes  # the default
- spatial_correlation = no, cross_correlation = no  # disable
- spatial_correlation = yes, cross_correlation = no
- spatial_correlation = no, cross_correlation = yes
- spatial_correlation = full, cross_correlation = full
- spatial_correlation = yes, cross_correlation = full
- spatial_correlation = no, cross_correlation = full
- spatial_correlation = full, cross_correlation = no
- spatial_correlation = full, cross_correlation = yes
```

*yes* means using the correlation matrix of the Silva-Horspool paper; *no* mean using no correlation; *full* means using an all-ones correlation matrix.

Apart from performance considerations, disabling either the spatial correlation or the cross correlation (or both) might be useful to see how significant the effect of the correlation is on the damage/loss estimates.

In particular, due to numeric errors, the spatial correlation matrix - that by construction contains only positive numbers - can still produce small negative eigenvalues (of the order of -1E-15) and
the calculation fails with an error message saying that the correlation matrix is not positive defined. Welcome to the world of floating point approximation! Rather than magically discarding negative eigenvalues the engine raises an error and the user has two choices: either disable the spatial correlation or reduce the number of sites because that can make the numerical instability go away. The easiest way to reduce the number of sites is setting a region_grid_spacing parameter in the prepare_job.ini file, then the engine will automatically put the assets on a grid. The larger the grid spacing, the fewer the number of points, and the closer the calculation will be to tractability.

## 20.3.3 Performance Considerations

The performance of the calculation will be crucially determined by the number of hazard sites. For instance, in the case of the Pisco earthquake the ShakeMap has 506,142 sites, which is a significantly large number of sites. However, the extent of the ShakeMap in longitude and latitude is about 6 degrees, with a step of 10 km the grid contains around 65 x 65 sites; most of the sites are without assets because most of the grid is on the sea or on high mountains, so actually there are around ~500 effective sites. Computing a correlation matrix of size 500 x 500 is feasible, so the risk computation can be performed.

Clearly in situations in which the number of hazard sites is too large, approximations will have to be made such as using a larger region_grid_spacing. Disabling spatial AND cross correlation makes it possible run much larger calculations. The performance can be further increased by not using a truncation_level.

When applying correlation, a soft cap on the size of the calculations is defined. This is done and modifiable through the parameter cholesky_limit which refers to the number of sites multiplied by the number of intensity measure types used in the vulnerability model. Raising that limit is at your own peril, as you might run out of memory during calculation or may encounter instabilities in the calculations as described above.

If the ground motion values or the standard deviations are particularly large, the user will get a warning about suspicious GMFs.

Moreover, especially for old ShakeMaps, the USGS can provide them in a format that the engine cannot read.

Thus, this feature is not expected to work in 100% of the cases.

Note: on macOS make sure to run the script located under /Applications/Python 3.6/ Install Certificates.command, after Python has been installed, to update the SSL certificates and to avoid SSL errors when downloading the ShakeMaps from the USGS website (see the FAQ)
Some advanced users are interested in developing with the engine, usually to contribute new GMPEs and sometimes to submit a bug fix. There are also users interested in implementing their own customizations of the engine. This part of the manual is for them.

### 21.1 Prerequisites

It is assumed here that you are a competent scientific Python programmer, i.e. that you have a good familiarity with the Python ecosystem (including pip and virtualenv) and its scientific stack (numpy, scipy, h5py, ...). It should be noticed that since engine v2.0 there is no need to know anything about databases and web development (unless you want to develop on the WebUI part) so the barrier for contribution to the engine is much lower than it used to be. However, contributing is still nontrivial, and it absolutely necessary to know git and the tools of Open Source development in general, in particular about testing. If this is not the case, you should do some study on your own and come back later. There is a huge amount of resources in the net about these topics. This manual will focus solely on the OpenQuake engine and it assume that you already know how to use it, i.e. you have read the User Manual first.

Before starting, it may be useful to have an idea of the architecture of the engine and its internal components, like the DbServer and the WebUI. For that you should read the *Architecture of the OpenQuake engine* document.

There are also external tools which are able to interact with the engine, like the QGIS plugin to run calculations and visualize the outputs and the IPT tool to prepare the required input files (except the hazard models). Unless you are developing for such tools you can safely ignore them.

### 21.2 The first thing to do

The first thing to do if you want to develop with the engine is to remove any non-development installation of the engine that you may have. While it is perfectly possible to install on the same machine both a development and a production instance of the engine (it is enough to configure
the ports of the DbServer and WebUI) it is easier to work with a single instance. In that way you will have a single code base and no risks of editing the wrong code. A development installation the engine works as any other development installation in Python: you should clone the engine repository, create and activate a virtualenv and then perform a `pip install -e` from the engine main directory, as normal. You can find the details here:


It is also possible to develop on Windows (https://github.com/gem/oq-engine/blob/master/doc/installing/development.md) but very few people in GEM are doing that, so you are on your own, should you encounter difficulties. We recommend Linux, but Mac also works.

Since you are going to develop with the engine, you should also install the development dependencies that by default are not installed. They are listed in the setup.py file, and currently (January 2020) they are pytest, flake8, pdbpp, silx and ipython. They are not required but very handy and recommended. It is the stack we use daily for development.

## 21.3 Understanding the engine

Once you have the engine installed you can run calculations. We recommend starting from the demos directory which contains example of hazard and risk calculations. For instance you could run the area source demo with the following command:

```bash
$ oq run demos/hazard/AreaSourceClassicalPSHA/job.ini
```

You should notice that we used here the command `oq run` while the engine manual recommend the usage of `oq engine --run`. There is no contradiction. The command `oq engine --run` is meant for production usage, it will work with the WebUI and store the logs of the calculation in the engine database. But here we are doing development, so the recommended command is `oq run` which will not interact with the database, will be easier to debug and accept the essential flag `--pdb`, which will start the python debugger should the calculation fail. Since during development is normal to have errors and problems in the calculation, so this ability is invaluable.

Then, if you want to understand what happened during the calculation you should generate the associated .rst report, which can be seen with the command

```bash
$ oq show fullreport
```

There you will find a lot of interesting information that it is worth studying and we will discuss in detail in the rest of this manual. The most important section of the report is probably the last one, titled “Slowest operations”. For that one can understand the bottlenecks of the calculation and, with experience, he can understand which part of the engine he needs to optimize. Also, it is very useful to play with the parameters of the calculation (like the maximum distance, the area discretization, the magnitude binning, etc etc) and see how the performance change. There is also a command to plot hazard curves and a command to compare hazard curves between different
calculations: it is common to be able to get big speedups simply by changing the input parameters in the job.ini of the model, without changing much the results.

There a lot of oq commands: if you are doing development you should study all of them. They are documented here.

## 21.4 Running calculations programmatically

Starting from engine 3.12 the recommended way to run a job programmaticaly is the following:

```python
from openquake.commonlib import logs
from openquake.calculators.base import calculators
with logs.init('job', 'job_ini') as log: # initialize logs
    calc = calculators(log.get_oqparam(), log.calc_id)
    calc.run() # run the calculator
```

Then the results can be read from the datastore by using the extract API:

```python
from openquake.calculators.extract import extract
extract(calc.datastore, 'something')
```

## 21.5 Case study: computing the impact of a source on a site

As an exercise showing off how to use the engine as a library, we will solve the problem of computing the hazard on a given site generated by a given source, with a given GMPE logic tree and a few parameters, i.e. the intensity measure levels and the maximum distance.

The first step is to specify the site and the parameters; let’s suppose that we want to compute the probability of exceeding a Peak Ground Accelation (PGA) of 0.1g by using the ToroEtAl2002SHARE GMPE:

```python
from openquake.commonlib import readinput
oq = readinput.get_oqparam(dict(
    calculation_mode='classical',
    sites='15.0 45.2',
    reference_vs30_type='measured',
    reference_vs30_value='600.0',
    intensity_measure_types_and_levels="{'PGA': [0.1]}",
    investigation_time='50.0',
    gsim='ToroEtAl2002SHARE',
    maximum_distance='200.0'))
```
Then we need to specify the source:

```python
>>> from openquake.hazardlib import nrml
>>> src = nrml.get('''
... <areaSource
...   id="126"
...   name="HRAS195"
... >
...   <areaGeometry discretization="10">
...     <gml:Polygon>
...       <gml:exterior>
...         <gml:LinearRing>
...           <gml:posList>
...             1.5026169E+01 4.5773603E+01
...             1.5650548E+01 4.6176279E+01
...             1.6273108E+01 4.6083465E+01
...             1.6398742E+01 4.6024744E+01
...             1.5947759E+01 4.5648318E+01
...             1.5677179E+01 4.5422577E+01
...           </gml:posList>
...         </gml:LinearRing>
...       </gml:exterior>
...     </gml:Polygon>
...   </areaGeometry>
...   <upperSeismoDepth>0</upperSeismoDepth>
...   <lowerSeismoDepth>30</lowerSeismoDepth>
...   <magScaleRel>WC1994</magScaleRel>
...   <ruptAspectRatio>1</ruptAspectRatio>
...   <incrementalMFD binWidth=".2" minMag="4.7">
...     <occurRates>
...       1.4731083E-02 9.2946848E-03 5.864596E-03
...       3.7002807E-03 2.3347193E-03 1.4731083E-03
...       9.2946848E-04 5.864596E-04 3.7002807E-04
...       2.3347193E-04 1.4731083E-04 9.2946848E-05
...       1.758846E-05 1.1097568E-05 2.3340307E-06
...     </occurRates>
...   </incrementalMFD>
...   <nodalPlaneDist>
...     <nodalPlane dip="5.7596810E+01" probability="1"
...       rake="0" strike="6.9033586E+01"/>
...   </nodalPlaneDist>
...   <hypoDepthDist>
...     <hypoDepth depth="1.0200000E+01" probability="1"/>
...   </hypoDepthDist>
... </areaSource>
'''

Then the hazard curve can be computed as follows:
21.6 Working with GMPEs directly: the ContextMaker

If you are a hazard scientist, you will likely want to interact with the GMPE library in openquake.hazardlib.gsim. The recommended way to do so is in terms of a ContextMaker object.

```python
from openquake.hazardlib.contexts import ContextMaker

param = dict(maximum_distance=oq.maximum_distance, imtls=oq.imtls,
              truncation_level=oq.truncation_level,
              investigation_time=oq.investigation_time)

cmaker = ContextMaker(src.tectonic_region_type, gsims, param)
```

Then you can use the ContextMaker to generate context objects from the sources:

```python
ctxs = cmaker.from_srcs([src], sitecol)
```

There is a context for each rupture in the source. In our example, there are 15 magnitudes

```python
len(src.get_annual_occurrence_rates())
```

15

and the area source contains 47 point sources

```python
len(list(src))
```

47

so in total there are 15 x 47 = 705 ruptures:

```python
len(ctxs)
```

705

The ContextMaker takes care of the maximum_distance filtering, so in general the number of contexts is lower than the total number of ruptures, since some ruptures are normally discarded, being distant from the sites.
The contexts contains all the rupture, site and distance parameters. Consider for instance the first context:

```python
>>> ctx = ctxs[0]
```

Then you have

```python
>>> ctx.mag
4.7
>>> ctx.rrup
array([106.40112646])
>>> ctx.rjb
array([105.8963247])
```

In this example, the GMPE `ToroEtAl2002SHARE` does not require site parameters, so calling `ctx.vs30` will raise an ‘`AttributeError`’ but in general the contexts contains also arrays of site parameters. There is also an array of indices telling which are the sites affected by the rupture associated to the context:

```python
>>> ctx.sids
array([0], dtype=uint32)
```

Once you have the contexts, the `ContextMaker` is able to compute means and standard deviations from the underlying GMPEs on each context. For instance, suppose you are only interested in the total standard deviation:

```python
>>> from openquake.hazardlib.const import StdDev
```

Then you can get a list of arrays containing mean and total standard deviation, with an array for each underlying gsim:

```python
>>> all_mean_std = cmaker.get_mean_stds(ctxs, StdDev.TOTAL)
```

Since in this example there is a single gsim you can do the following:

```python
>>> mean, std = all_mean_std[0]
>>> mean.shape
(1, 705)
>>> std.shape
(1, 705)
```

The shape of the arrays is $(M, C)$ where $M$ is the number of intensity measure types (in this example there is only one, PGA) and $C$ is the total size of the contexts. Since this is an example with a single site, each context has size 1, therefore $C = 705 \times 1 = 705$. In general if there are multiple sites a context $C$ is the total number of affected sites. For instance if there are two contexts and the first affect 1 sites and the second 2 sites then $C$ would be $1 + 2 = 3$. This example correspond to $1 + 1 + \ldots + 1 = 705$. 

90 Chapter 21. Developing with the engine
From the mean and standard deviation is possible to compute the probabilities of exceedence. The ContextMaker provides a method to compute directly the probability map, which internally calls `cmaker.get_mean_stds(ctxs, StdDev.TOTAL):

```python
>>> cmaker.get_pmap(ctxs)
{0: <ProbabilityCurve [[0.00508693]>>}
```

This is exactly the result provided by `calc_hazard_curve(sitecol, src, gsims, oq)` in the section before.

If you want to know exactly how `get_pmap` works you are invited to look at the source code in `openquake.hazardlib.contexts`. 

21.6. Working with GMPEs directly: the ContextMaker
ARCHITECTURE OF THE OPENQUAKE ENGINE

The engine is structured as a regular scientific application: we try to perform calculations as much as possible in memory and when it is not possible intermediate results are stored in HDF5 format. We try work as much as possible in terms of arrays which are efficiently manipulated at C/Fortran speed with a stack of well established scientific libraries (numpy/scipy).

CPU-intensity calculations are parallelized with a custom framework (the engine is ten years old and predates frameworks like dask or ray) which however is quite easy to use and mostly compatible with Python multiprocessing or concurrent.futures. The concurrency architecture is the standard Single Writer Multiple Reader (SWMR), used at the HDF5 level: only one process can write data while multiple processes can read it. The engine runs seamlessly on a single machine or a cluster, using as many cores as possible.

In the past the engine had a database-centric architecture and was more class-oriented than numpy-oriented: some remnants of such dark past are still there, but they are slowly disappearing. Currently the database is only used for storing accessory data and it is a simple SQLite file. It is mainly used by the WebUI to display the logs.

22.1 Components of the OpenQuake Engine

The OpenQuake Engine suite is composed of several components:

- a set of support libraries addressing different concerns like reading the inputs and writing the outputs, implementing basic geometric manipulations, managing distributed computing and generic programming utilities
- the hazardlib and risklib scientific libraries, providing the building blocks for hazard and risk calculations, notably the GMPEs for hazard and the vulnerability/fragility functions for risk
- the hazard and risk calculators, implementing the core logic of the engine
- the datastore, which is an HDF5 file working as a short term storage/cache for a calculation; it is possible to run a calculation starting from an existing datastore, to avoid recomputing everything every time; there is a separate datastore for each calculation
• the *database*, which is a SQLite file working as a long term storage for the calculation metadata; the database contains the start/stop times of the computations, the owner of a calculation, the calculation descriptions, the performances, the logs, etc; the bulk scientific data (essentially big arrays) are kept in the datastore

• the *DbServer*, which is a service mediating the interaction between the calculators and the database

• the *WebUI* is a web application that allows to run and monitor computations via a browser; multiple calculations can be run in parallel

• the *oq* command-line tool; it allows to run computations and provides an interface to the underlying database and datastores so that it is possible to list and export the results

• the engine can run on a cluster of machines: in that case a minimal amount of configuration is needed, whereas in single machine installations the engine works out of the box

• since v3.8 the engine does not depend anymore from celery and rabbitmq, but can still use such tools until they will be deprecated

This is the full stack of internal libraries used by the engine. Each of these is a Python package containing several modules or event subpackages. The stack is a dependency tower where the higher levels depend on the lower levels but not vice versa:

• level 9: commands (subcommands of oq)
• level 8: server (database and Web UI)
• level 7: engine (command-line tool, export, logs)
• level 6: calculators (hazard and risk calculators)
• level 5: commonlib (configuration, logic trees, I/O)
• level 4: risklib (risk validation, risk models, risk inputs)
• level 3: hmtk (catalogues, plotting, …)
• level 2: hazardlib (validation, read/write XML, source and site objects, geospatial utilities, GSIM library)
• level 1: baselib (programming utilities, parallelization, monitoring, hdf5…)

*baselib* and *hazardlib* are very stable and can be used outside of the engine; the other libraries are directly related to the engine and are likely to be affected by backward-incompatible changes in the future, as the code base evolves.

The GMPE library in *hazardlib* and the calculators are designed to be extensible, so that it is easy to add a new GMPE class or a new calculator. We routinely add several new GMPEs per release; adding new calculators is less common and it requires more expertise, but it is possible and it has been done several times in the past. In particular it is often easier to add a specific calculator optimized for a given use case rather than complicating the current calculators.
The results of a computation are automatically saved in the datastore and can be exported in a portable format, such as CSV (or XML, for legacy reasons). You can assume that the datastore of version X of the engine will not work with version X + 1. On the contrary, the exported files will likely be same across different versions. It is important to export all of the outputs you are interested in before doing an upgrade, otherwise you would be forced to downgrade in order to be able to export the previous results.

The WebUI provides a REST API that can be used in third party applications: for instance a QGIS plugin could download the maps generated by the engine via the WebUI and display them. There is lot of functionality in the API which is documented here: https://github.com/gem/oq-engine/blob/master/doc/web-api.md. It is possible to build your own user interface for the engine on top of it, since the API is stable and kept backward compatible.

### 22.2 Design principles

The main design principle has been simplicity: everything has to be as simple as possible (but not simplest). The goal has been to keep the engine simple enough that a single person can understand it, debug it, and extend it without tremendous effort. All the rest comes from simplicity: transparency, ability to inspect and debug, modularity, adaptability of the code, etc. Even efficiency: in the last three years most of the performance improvements came from free, just from removing complications. When a thing is simple it is easy to make it fast. The battle for simplicity is never ending, so there are still several things in the engine that are more complex than they should: we are working on that.

After simplicity the second design goal has been performance: the engine is a number crunching application after all, and we need to run massively parallel calculations taking days or weeks of runtime. Efficiency in terms of computation time and memory requirements is of paramount importance, since it makes the difference between being able to run a computation and being unable to do it. Being too slow to be usable should be considered as a bug.

The third requirement is reproducibility, which is the same as testability: it is essential to have a suite of tests checking that the calculators are providing the expected outputs against a set of predefined inputs. Currently we have thousands of tests which are run multiple times per day in our Continuous Integration environments (avis, GitLab, Jenkins), split into unit tests, end-to-end tests and long running tests.
CHAPTER

TWENTYTHREE

SOME USEFUL OQ COMMANDS

The oq command-line script is the entry point for several commands, the most important one being oq engine, which is documented in the manual.

The commands documented here are not in the manual because they have not reached the same level of maturity and stability. Still, some of them are quite stable and quite useful for the final users, so feel free to use them.

You can see the full list of commands by running oq --help:

```
$ oq --help
usage: oq [--version]
       {workerpool,webui,dbserver,info,ltcsv,dump,export,celery,
        plot_losses,restore,plot_assets,reduce_sm,check_input,plot_ac,
        upgrade_nrml,shell,plot_pyro,nrml_to,postzip,show,workers,abort,
        engine,reaggregate,db,compare,renumber_sm,download_shakemap,
        importcalc,purge,tidy,from_shapefile,zip,checksum,to_shapefile,to_hdf5,extract,reset,run,show_attrs,prepare_site_model,sample,plot}
...

positional arguments:
  {workerpool,webui,dbserver,info,ltcsv,dump,export,celery,plot_losses,
   restore,plot_assets,reduce_sm,check_input,plot_ac,upgrade_nrml,shell,
   plot_pyro,nrml_to,postzip,show,workers,abort,engine,reaggregate,db,
   compare,renumber_sm,download_shakemap,importcalc,purge,tidy,from_shapefile,zip,checksum,to_shapefile,to_hdf5,extract,reset,run,show_attrs,prepare_site_model,sample,plot}
available subcommands; use oq <subcmd> --help

optional arguments:
  -h, --help    show this help message and exit
  -v, --version show program's version number and exit
```

This is the output that you get at the present time (engine 3.11); depending on your version of the engine you may get a different output. As you see, there are several commands, like purge,
show.attrs, export, restore, … You can get information about each command with oq <command> --help; for instance, here is the help for purge:

```bash
$ oq purge --help
usage: oq purge [-h] [-f] calc_id

Remove the given calculation. If you want to remove all calculations, use oq reset.

positional arguments:
calc_id    calculation ID

optional arguments:
    -h, --help  show this help message and exit
    -f, --force ignore dependent calculations
```

Some of these commands are highly experimental and may disappear; others are meant for debugging and are not meant to be used by end-users. Here I will document only the commands that are useful for the general public and have reached some level of stability.

Probably the most important command is oq info. It has several features.

1. It can be invoked with a `job.ini` file to extract information about the logic tree of the calculation.
2. When invoked with the --report option, it produces a `.rst` report with several important informations about the computation. It is ESSENTIAL in the case of large calculations, since it will give you an idea of the feasibility of the computation without running it. Here is an example of usage:

```bash
$ oq info --report job.ini
Generated /tmp/report_1644.rst
<Monitor info, duration=10.910529613494873s, memory=60.16 MB>
```

You can open `/tmp/report_1644.rst` and read the informations listed there (1644 is the calculation ID, the number will be different each time).

3. It can be invoked without a `job.ini` file, and it that case it provides global information about the engine and its libraries. Try, for instance:

```bash
$ oq info calculators # list available calculators
$ oq info gsims # list available GSIMs
$ oq info views # list available views
$ oq info exports # list available exports
$ oq info parameters # list all job.ini parameters
```

The second most important command is oq export. It allows customization of the exports from the datastore with additional flexibility compared to the oq engine export commands. In the future the oq engine exports commands might be deprecated and oq export might become the official export command, but we are not there yet.
Here is the usage message:

```bash
$ oq export --help
usage: oq export [-h] [-e csv] [-d .] datastore_key [calc_id]

Export an output from the datastore.

positional arguments:
  datastore_key  datastore key
  calc_id       number of the calculation [default: -1]

optional arguments:
  -h, --help    show this help message and exit
  -e csv, --exports csv
                export formats (comma separated)
  -d ., --export-dir .  export directory
```

The list of available exports (i.e. the datastore keys and the available export formats) can be extracted with the `oq info exports` command; the number of exporters defined changes at each version:

```bash
$ oq info exports
agg_curves-rlzs ['csv']
agg_curves-stats ['csv']
agg_loss_table ['csv']
agg_losses-rlzs ['csv']
agg_losses-stats ['csv']
agg_risk ['csv']
agg_curves ['csv']
agg_losses ['csv']
aggregate_by ['csv']
asset_risk ['csv']
avg_gmf ['csv']
avg_losses-rlzs ['csv']
avg_losses-stats ['csv']
bcr-rlzs ['csv']
bcr-stats ['csv']
damages-rlzs ['npz', 'csv']
damages-stats ['csv']
disagg ['csv', 'xml']
disagg_traditional ['csv']
events ['csv']
fullreport ['rst']
gmf_data ['csv', 'hdf5']
hcurves ['csv', 'xml', 'npz']
hmaps ['csv', 'xml', 'npz']
input ['zip']
```
loss_curves ['csv']
loss_curves-rlzs ['csv']
loss_curves-stats ['csv']
loss_maps-rlzs ['csv', 'npz']
loss_maps-stats ['csv', 'npz']
losses_by_asset ['npz']
realizations ['csv']
ruptures ['xml', 'csv']
src_loss_table ['csv']
uhs ['csv', 'xml', 'npz']

There are 47 exporters defined.

At the present the supported export types are xml, csv, rst, npz and hdf5. xml has been deprecated for some outputs and is not the recommended format for large exports. For large exports, the recommended formats are npz (which is a binary format for numpy arrays) and hdf5. If you want the data for a specific realization (say the first one), you can use:

```bash
$ oq export hcurves/rlz-0 --exports csv
$ oq export hmaps/rlz-0 --exports csv
$ oq export uhs/rlz-0 --exports csv
```

but currently this only works for csv and xml. The exporters are one of the most time-consuming parts on the engine, mostly because of the sheer number of them; the are more than fifty exporters and they are always increasing. If you need new exports, please [add an issue on GitHub](https://github.com/gem/oq-engine/issues).

### 23.1 oq zip

An extremely useful command if you need to copy the files associated to a computation from a machine to another is oq zip:

```bash
$ oq zip --help
usage: oq zip [-h] [-r] what [archive_zip]

positional arguments:
  what         path to a job.ini, a ssmLT.xml file, or an ...
  exposure.xml
  archive_zip  path to a non-existing .zip file [default: '']

optional arguments:
  -h, --help          show this help message and exit
  -r, --risk-file     optional file for risk
```

For instance, if you have two configuration files `job_hazard.ini` and `job_risk.ini`, you can zip all the files they refer to with the command:
$ oq zip job_hazard.ini -r job_risk.ini

{oq zip} is actually more powerful than that; other than job.ini files, it can also zip source models:

$ oq zip ssmLT.xml

and exposures:

$ oq zip my_exposure.xml

### 23.2 Importing a remote calculation

The use-case is importing on your laptop a calculation that was executed on a remote server/cluster. For that to work you need to create a file called openquake.cfg in the virtualenv of the engine (the output of the command oq info venv, normally it is in $HOME/openquake) with the following section:

```ini
[webapi]
server = https://oq1.wilson.openquake.org/    # change this
username = michele  # change this
password = PWD     # change this
```

Then you can import any calculation by simply giving its ID, as in this example:

```sh
$ oq importcalc 41214
INFO:root:Saving /home/michele/oqdata/calc_41214.hdf5
Downloaded 58,118,085 bytes
{"checksum32": 1949258781,
 "date": '2021-03-18T15:25:11',
 "engine_version": '3.12.0-gita399903317'}
INFO:root:Imported calculation 41214 successfully
```

### 23.3 plotting commands

The engine provides several plotting commands. They are all experimental and subject to change. They will always be. The official way to plot the engine results is by using the QGIS plugin. Still, the {oq} plotting commands are useful for debugging purposes. Here I will describe the plot_assets command, which allows to plot the exposure used in a calculation together with the hazard sites:
This is particularly interesting when the hazard sites do not coincide with the asset locations, which is normal when gridding the exposure.

Very often, it is interesting to plot the sources. While there is a primitive functionality for that in `oq plot`, we recommend to convert the sources into .gpkg format and use QGIS to plot them:

```
$ oq nrml_to --help
usage: oq nrml_to [-h] [-o .] [-c] {csv,gpkg} fnames [fnames ...]

Convert source models into CSV files or a geopackage.

positional arguments:
    {csv,gpkg}           csv or gpkg
    fnames               source model files in XML

optional arguments:
    -h, --help           show this help message and exit
    -o ., --outdir .     output directory
    -c, --chatty         display sources in progress
```

For instance

```
$ oq nrml_to gpkg source_model.xml -o source_model.gpkg
```

will convert the sources in .gpkg format while

```
$ oq nrml_to csv source_model.xml -o source_model.csv
```

will convert the sources in .csv format. Both are fully supported by QGIS. The CSV format has the advantage of being transparent and easily editable; it also can be imported in a geospatial database like Postgres, if needed.

### 23.4 prepare_site_model

The command `oq prepare_site_model`, introduced in engine 3.3, is quite useful if you have a vs30 file with fields lon, lat, vs30 and you want to generate a site model from it. Normally this feature
is used for risk calculations: given an exposure, one wants to generate a collection of hazard sites covering the exposure and with vs30 values extracted from the vs30 file with a nearest neighbour algorithm:

```bash
$ oq prepare_site_model -h
usage: oq prepare_site_model [-h] [-1] [-2] [-3]
    [-e [EXPOSURE_XML [EXPOSURE_XML ...]]]
    [-s SITES_CSV] [-g 0] [-a 5] [-o site_model.csv]
    vs30_csv [vs30_csv ...]
```

Prepare a site_model.csv file from exposure xml files/site csv files, vs30 csv files and a grid spacing which can be 0 (meaning no grid). For each site the closest vs30 parameter is used. The command can also generate (on demand) the additional fields z1pt0, z2pt5 and vs30measured which may be needed by your hazard model, depending on the required GSIMs.

Positional arguments:
- vs30_csv files with lon,lat,vs30 and no header

Optional arguments:
- -h, --help show this help message and exit
- -1, --z1pt0
- -2, --z2pt5 build the z2pt5
- -3, --vs30measured build the vs30measured
- -e [EXPOSURE_XML [EXPOSURE_XML ...]], --exposure-xml [EXPOSURE_XML [, [EXPOSURE_XML ...]]] exposure(s) in XML format
- -s SITES_CSV, --sites-csv SITES_CSV
- -g 0, --grid-spacing 0 grid spacing in km (the default 0 means no grid)
- -a 5, --assoc-distance 5 sites over this distance are discarded
- -o site_model.csv, --output site_model.csv output file

The command works in two modes: with non-gridded exposures (the default) and with gridded exposures. In the first case the assets are aggregated in unique locations and for each location the vs30 coming from the closest vs30 record is taken. In the second case, when a `grid_spacing` parameter is passed, a grid containing all of the exposure is built and the points with assets are associated to the vs30 records. In both cases if the closest vs30 record is over the `site_param_distance` - which by default is 5 km - a warning is printed.
In large risk calculations, it is quite preferable *to use the gridded mode* because with a well spaced grid,

1. the results are the nearly the same than without the grid and
2. the calculation is a lot faster and uses a lot less memory.

Gridding of the exposure makes large calculations more manageable. The command is able to manage multiple Vs30 files at once. Here is an example of usage:

```bash
$ oq prepare_site_model Vs30/Ecuador.csv Vs30/Bolivia.csv -e Exposure/
  Exposure_Res_Ecuador.csv Exposure/Exposure_Res_Bolivia.csv --grid-
  spacing=10
```

## 23.5 Reducing the source model

Source models are usually large, at the continental scale. If you are interested in a city or in a small region, it makes sense to reduce the model to only the sources that would affect the region, within the integration distance. To fulfil this purpose there is the `oq reduce_sm` command. The suggestion is run a preclassical calculation (i.e. set `calculation_mode=preclassical` in the job.ini) with the full model in the region of interest, keep track of the calculation ID and then run:

```bash
$ oq reduce_sm <calc_id>
```

The command will reduce the source model files and add an extension `.bak` to the original ones.

```bash
$ oq reduce_sm -h
usage: oq reduce_sm [-h] calc_id

Reduce the source model of the given (pre)calculation by discarding all sources that do not contribute to the hazard.

positional arguments:
  calc_id    calculation ID

optional arguments:
  -h, --help  show this help message and exit
```

## 23.6 Comparing hazard results

If you are interested in sensitivity analysis, i.e. in how much the results of the engine change by tuning a parameter, the `oq compare` command is useful. For the moment it is able to compare hazard curves and hazard maps. Here is the help message:

```bash
$ oq compare -h
usage: oq compare [-h] calc_id1 calc_id2

Compare two hazard results.

positional arguments:
  calc_id1    calculation ID
  calc_id2    calculation ID

optional arguments:
  -h, --help    show this help message and exit
```
$ oq compare --help
usage: oq compare [-h] [-f] [-s 100] [-r 0] [-a 0.001] [-t 0.01]
   {hcurves,hmaps} imt calc_ids [calc_ids ...]

Compare the hazard curves or maps of two or more calculations

positional arguments:
  {hcurves,hmaps} hmaps or hcurves
  imt intensity measure type to compare
  calc_ids calculation IDs

optional arguments:
  -h, --help show this help message and exit
  -f, --files write the results in multiple files
  -s 100, --samplesites 100 sites to sample (or fname with site IDs)
  -r 0, --rtol 0 relative tolerance
  -a 0.001, --atol 0.001 absolute tolerance
  -t 0.01, --threshold 0.01 ignore the hazard curves below it
LIMITATIONS OF FLOATING-POINT ARITHMETIC

Most practitioners of numeric calculations are aware that addition of floating-point numbers is non-associative; for instance

```python
>>> (.1 + .2) + .3
0.6000000000000001
```

is not identical to

```python
>>> .1 + (.2 + .3)
0.6
```

Other floating-point operations, such as multiplication, are also non-associative. The order in which operations are performed plays a role in the results of a calculation.

Single-precision floating-point variables are able to represent integers between \([-16777216, 16777216]\) exactly, but start losing precision beyond that range; for instance:

```python
>>> numpy.float32(16777216)
16777216.0
```

```python
>>> numpy.float32(16777217)
16777216.0
```

This loss of precision is even more apparent for larger values:

```python
>>> numpy.float32(123456786432)
123456780000.0
```

```python
>>> numpy.float32(123456786433)
123456790000.0
```

These properties of floating-point numbers have critical implications for numerical reproducibility of scientific computations, particularly in a parallel or distributed computing environment.
For the purposes of this discussion, let us define numerical reproducibility as obtaining bit-wise identical results for different runs of the same computation on either the same machine or different machines.

Given that the OpenQuake engine works by parallelizing calculations, numerical reproducibility cannot be fully guaranteed, even for different runs on the same machine (unless the computation is run using the –no-distribute or –nd flag).

Consider the following strategies for distributing the calculation of the asset losses for a set of events, followed by aggregation of the results for the portfolio due to all of the events. The assets could be split into blocks with each task computing the losses for a particular block of assets and for all events, and the partial losses coming in from each task is aggregated at the end. Alternatively, the assets could be kept as a single block, splitting the set of events/ruptures into blocks instead; once again the engine has to aggregate partial losses coming in from each of the tasks. The order of the tasks is arbitrary, because it is impossible to know how long each task will take before the computation actually begins.

For instance, suppose there are 3 tasks, the first one producing a partial loss of 0.1 billion, the second one of 0.2 billion, and the third one of 0.3 billion. If we run the calculation and the order in which the results are received is 1-2-3, we will compute a total loss of (.1 + .2) + .3 = 0.6000000000000001 billion. On the other hand, if for some reason the order in which the results arrive is 2-3-1, say for instance, if the first core is particularly hot and the operating system decides to enable some throttling on it, then the aggregation will be (.2 + .3) + .1 = 0.6 billion, which is different from the previous value by 1.11E-7 units. This example assumes the use of Python’s IEEE-754 “double precision” 64-bit floats.

However, the engine uses single-precision 32-bit floats rather than double-precision 64-bit floats in a tradeoff necessary for reducing the memory demand (both RAM and disk space) for large computations, so the precision of results is less than in the above example. 64-bit floats have 53 bits of precision, and this why the relative error in the example above was around 1.11E-16 (i.e. 2^-53). 32-bit floats have only 24 bits of precision, so we should expect a relative error of around 6E-8 (i.e. 2^-24), which for the example above would be around 60 units. Loss of precision in representing and storing large numbers is a factor that must be considered when running large computations.

By itself, such differences may be negligible for most computations. However, small differences may accumulate when there are hundreds or even thousands of tasks handling different parts of the overall computation and sending back results for aggregation.

Anticipating these issues, some adjustments can be made to the input models in order to circumvent or at least minimize surprises arising from floating-point arithmetic. Representing asset values in the exposure using thousands of dollars as the unit instead of dollars could be one such defensive measure.

This is why, as an aid to the interested user, starting from version 3.9, the engine logs a warning if it finds inconsistencies beyond a tolerance level in the aggregated loss results.

Recommended readings:

• [https://docs.python.org/3/tutorial/floatingpoint.html](https://docs.python.org/3/tutorial/floatingpoint.html)
