OpenQuake for Advanced Users Documentation

Release 3.17.2

Michele Simionato, Anirudh Rao

Jul 31, 2023

CONTENTS

PDF version 3.17: https://docs.openquake.org/oq-engine/3.17/PDF/OpenQuakeforAdvancedUsers.pdf

Contents:

GENERAL

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 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 with contain millions and mil-

lions 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.

COMMON MISTAKES: BAD CONFIGURATION PARAMETERS

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:

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

You can also have a magnitude-dependent maximum distance:

```
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 (the maximum distance in this magnitude range will vary linearly between 0 and 100), keep ruptures up to 200 km for magnitudes between 6 and 7 (with *maximum_distance* increasing linearly from 100 to 200 km from magnitude 6 to magnitude 7), keep ruptures up to 300 km for magnitudes between 7 and 8 (with *maximum_distance* increasing linearly from 200 to 300 km from magnitude 7 to magnitude 8) and discard ruptures for magnitudes over 8.

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

```
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:

```
>>> from openquake.hazardlib.calc.filters import IntegrationDistance
>>> idist = IntegrationDistance.new('[(4, 0), (6, 100), (7, 200), (8.5,␣
↪300)]')
>>> interp = idist('TRT')
>>> interp([4.5, 5.5, 6.5, 7.5, 8])
array([ 25.        ,  75.        , 150.        , 233.33333333,
       266.66666667])
```

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* plus the rupture radius from the point source, the engine 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 `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.

SA(2.0), site 0, inv_time=1y
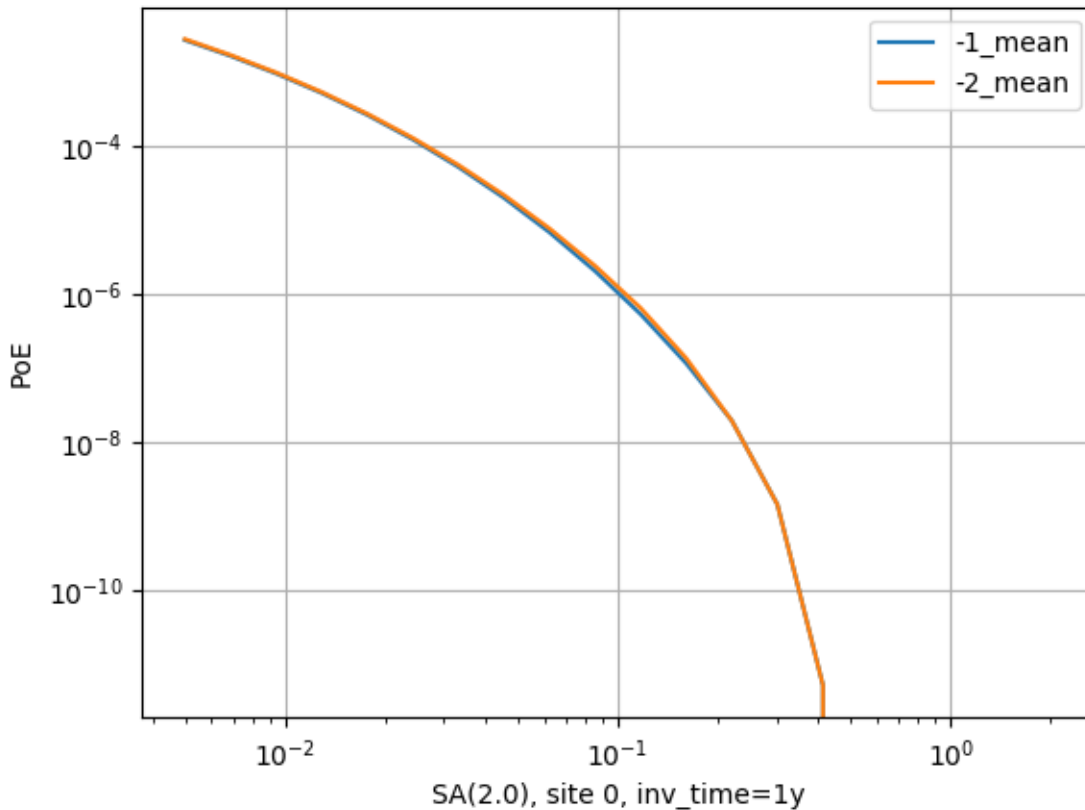
The `pointsource_distance` is also crucial when using the point source gridding approximation: then it can be used to speedup calculations even when the nodal plane and hypocenter distributions are trivial and no speedup would be expected.

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:

```
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.

DEVELOPING WITH THE ENGINE

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.

3.1 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 on the net about these topics. This manual will focus solely on the OpenQuake engine and it assumes 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.

3.2 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:

https://github.com/gem/oq-engine/blob/engine-3.17/doc/installing/development.md

It is also possible to develop on Windows (https://github.com/gem/oq-engine/blob/engine-3.17/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.

3.3 Understanding the engine

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

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

You should notice that we used here the command `oq run` while the engine manual recommends the usage of `oq engine --run`. There is no contradiction. The command `oq engine --run` is meant for production usage, but here we are doing development, so the recommended command is `oq run` which will will be easier to debug thanks to the 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, this ability is invaluable.

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

```
$ 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 a 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 changes. 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.

3.4 Running calculations programmatically

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

```
>> 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:

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

3.5 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:

```
>>> 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',
```

(continues on next page)

```
... intensity_measure_types_and_levels="{'PGA': [0.1]}",
... investigation_time='50.0',
... gsim='ToroEtAl2002SHARE',
... truncation_level='99.0',
... maximum_distance='200.0'))
```

Then we need to specify the source:

```
>>> 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>
...                     <upperSeismoDepth>0</upperSeismoDepth>
...                     <lowerSeismoDepth>30</lowerSeismoDepth>
...                 </areaGeometry>
...                 <magScaleRel>WC1994</magScaleRel>
...                 <ruptAspectRatio>1</ruptAspectRatio>
...                 <incrementalMFD binWidth=".2" minMag="4.7">
...                     <occurRates>
...                         1.4731083E-02 9.2946848E-03 5.8645496E-03
...                         3.7002807E-03 2.3347193E-03 1.4731083E-03
...                         9.2946848E-04 5.8645496E-04 3.7002807E-04
...                         2.3347193E-04 1.4731083E-04 9.2946848E-05
...                         1.7588460E-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:

```
>>> from openquake.hazardlib.calc.hazard_curve import calc_hazard_curve
>>> from openquake.hazardlib import valid
>>> sitecol = readinput.get_site_collection(oq)
>>> gsims = readinput.get_gsim_lt(oq).values['*']
>>> calc_hazard_curve(sitecol, src, gsims, oq)
<ProbabilityCurve
[[0.00507997]]>
```

3.6 Working with GMPEs directly: the ContextMaker

If you are an 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.

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

In order to instantiate a `ContextMaker` you first need to populate a dictionary of parameters:

```
>>> 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 arrays from the sources:

```
>>> [ctx] = cmaker.from_srcs([src], sitecol)
```

In our example, there are 15 magnitudes

```
>>> len(src.get_annual_occurrence_rates())
15
```

and the area source contains 47 point sources

```
>>> len(list(src))
47
```

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

```
>>> len(ctx)
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 contain all the rupture, site and distance parameters.

Then you have

```
>>> ctx.mag[0]
4.7
>>> round(ctx.rrup[0], 1)
106.4
>>> round(ctx.rjb[0], 1)
105.9
```

In this example, the GMPE `ToroEtAl2002SHARE` does not require site parameters, so calling `ctx.vs30` will raise an `AttributeError` but in general the contexts contain 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:

```
>>> import numpy
>>> numpy.unique(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 as follows (for engine version >= 3.13):

```
>>> mean, sig, tau, phi = cmaker.get_mean_stds([ctx])
```

Since in this example there is a single gsim and a single IMT you will get:

```
>>> mean.shape
(1, 1, 705)
>>> sig.shape
(1, 1, 705)
```

The shape of the arrays in general is (G, M, N) where G is the number of GSIMs, M the number of

intensity measure types and N the total size of the contexts. Since this is an example with a single site, each context has size 1, therefore N = 705 * 1 = 705. In general if there are multiple sites a context M 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 N would be 1 + 2 = 3. This example correspond to 1 + 1 + … + 1 = 705.

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_pmap([ctx])` which gives 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`.

3.7 Working with verification tables

Hazard scientists implementing a new GMPE must provide verification tables, i.e. CSV files containing inputs and expected outputs.

For instance, for the Atkinson2015 GMPE (chosen simply because is the first GMPE in lexicographic order in hazardlib) the verification table has a structure like this:

```
rup_mag,dist_rhypo,result_type,pgv,pga,0.03,0.05,0.1,0.2,0.3,0.5
2.0,1.0,MEAN,5.50277734e-02,3.47335058e-03,4.59601700e-03,7.71361460e-03,
→9.34624779e-03,4.33207607e-03,1.75322233e-03,3.44695521e-04
2.0,5.0,MEAN,6.43850933e-03,3.61047741e-04,4.57949482e-04,7.24558049e-04,
→9.44495571e-04,5.11252304e-04,2.21076069e-04,4.73435138e-05
...
```

The columns starting with `rup_` contain rupture parameters (the magnitude in this example) while the columns starting with `dist_` contain distance parameters. The column `result_type` is a string in the set {"MEAN", "INTER_EVENT_STDDEV", "INTRA_EVENT_STDDEV", "TOTAL_STDDEV"}. The remaining columns are the expected results for each intensity measure type; in the the example the IMTs are PGV, PGA, SA(0.03), SA(0.05), SA(0.1), SA(0.2), SA(0.3), SA(0.5).

Starting from engine version 3.13, it is possible to instantiate a ContextMaker and the associated contexts from a GMPE and its verification tables with a few simple steps. First of all one must instantiate the GMPE:

```
>>> from openquake.hazardlib import valid
>>> gsim = valid.gsim("Atkinson2015")
```

Second, one can determine the path names to the verification tables as follows (they are in a subdirectory of *hazardlib/tests/gsim/data*):

```
>>> import os
>>> from openquake.hazardlib.tests.gsim import data
>>> datadir = os.path.join(data.__path__[0], 'ATKINSON2015')
>>> fnames = [os.path.join(datadir, f) for f in ["ATKINSON2015_MEAN.csv",
...             "ATKINSON2015_STD_INTER.csv", "ATKINSON2015_STD_INTRA.csv",
...             "ATKINSON2015_STD_TOTAL.csv"]]
```

Then it is possible to instantiate the ContextMaker associated to the GMPE and a pandas DataFrame associated to the verification tables in a single step:

```
>>> from openquake.hazardlib.tests.gsim.utils import read_cmaker_df, gen_
↪ctxs
>>> cmaker, df = read_cmaker_df(gsim, fnames)
>>> list(df.columns)
['rup_mag', 'dist_rhypo', 'result_type', 'damping', 'PGV', 'PGA', 'SA(0.
↪03)', 'SA(0.05)', 'SA(0.1)', 'SA(0.2)', 'SA(0.3)', 'SA(0.5)', 'SA(1.0)',
↪ 'SA(2.0)', 'SA(3.0)', 'SA(5.0)']
```

Then you can immediately compute mean and standard deviations and compare with the values in the verification table:

```
>>> mean, sig, tau, phi = cmaker.get_mean_stds(gen_ctxs(df))
```

sig refers to the "TOTAL_STDDEV", *tau* to the "INTER_EVENT_STDDEV" and *phi* to the "INTRA_EVENT_STDDEV". This is how the tests in hazardlib are implemented. Interested users should look at the code in https://github.com/gem/oq-engine/blob/engine-3.17/openquake/hazardlib/tests/gsim/utils.py.

3.8 Running the engine tests

If you are a hazard scientist contributing a bug fix to a GMPE (or any other kind of bug fix) you may need to run the engine tests and possibly change the expected files if there is a change in the numbers. The way to do it is to start the dbserver and then run the tests from the repository root:

```
$ oq dbserver start
$ pytest -vx openquake/calculators
```

If you get an error like the following:

```
openquake/calculators/tests/__init__.py:218: in assertEqualFiles
    raise DifferentFiles('%s %s' % (expected, actual))
E   openquake.calculators.tests.DifferentFiles: /home/michele/oq-engine/
↪openquake/qa_tests_data/classical/case_1/expected/hazard_curve-PGA.csv /
↪tmp/tmpkdvdhlq5/hazard_curve-mean-PGA_27249.csv
```

you need to change the expected file, i.e. copy the file `/tmp/tmpkdvdhlq5/`
`hazard_curve-mean-PGA_27249.csv` over `classical/case_1/expected/`
`hazard_curve-PGA.csv`.

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 to 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.

4.1 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 meta-data; 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/engine-3.17/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.

4.2 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, adaptablity 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 (GitHub Actions, GitLab Pipelines), split into unit tests, end-to-end tests and long running tests.

FIVE

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 zeromq, which is a well-known and maintained library.

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*.

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. Zeromq has 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 zeromq with other parallelization mechanisms: people interested in doing so should just contact us.

Another tricky aspect 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 information 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):  # simplified code
    ebruptures = []
    for src in sources:
        for ebr in src.sample_ruptures(num_samples):
            eb_ruptures.append(ebr)
        if len(eb_ruptures) > MAX_RUPTURES:
            # yield partial result to avoid running out of memory
            yield eb_ruptures
            eb_ruptures.clear()
    if ebruptures:
        yield eb_ruptures
```

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.

From the point of view of the coder, in the engine there is no difference between a task running on a cluster using zeromq and a task running locally using `multiprocessing.Pool`: they are coded the same, but depending on a configuration parameter in openquake.cfg (`distribute=zmq` 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 (`zmq` 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 we need to transfer gigabytes of data, i.e. the large arrays coming from scientific calculations, rather than lots of small messages. Hence, we are returning data from the tasks via zeromq instead of using celery/rabbitmq as we did in the remote past. 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 task 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 openquake 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.

5.1 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 builds 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
```

```python
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:  # create a new datastore
        monitor = Monitor('count', hdf5)  # create a new monitor
        iterargs = ((open(dname/fname, encoding='utf-8').read(),)
                    for fname in os.listdir(dname)
                    if fname.endswith('.rst'))  # read the docs
        c = collections.Counter()  # intially empty counter
        for counter in Starmap(count, iterargs, monitor):
            c += counter
        print(c)  # total counts
        print('Performance info stored in', hdf5)


if __name__ == '__main__':
    main(sys.argv[1])  # pass the directory where the .rst files are
```

The name `Starmap` was chosen because it looks very similar to how `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 return 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` method 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.

SIX

EXTRACTING DATA FROM CALCULATIONS

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 is 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:

```
>> 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.mean.shape  # an example with 10,000 sites, 20 levels per PGA
```

(continues on next page)

```
(10000, 1, 20)
>> extractor.close()
```

If in the calculation you specified the flag `individual_rlzs=true`, then it is also possible to retrieve a specific realization

>> dic = vars(extractor.get('hcurves?kind=rlz-0')) >> dic['rlz-000'] # array of shape (num_sites, num_imts, num_levels)

or even all realizations:

>> dic = vars(extractor.get('hcurves?kind=rlzs'))

Here is an example of using the *WebExtractor* to retrieve hazard maps. Here we assume that in a remote machine 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.mean.shape   # an example with 10,000 sites and 4 PoEs
(10000, 1, 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)
```

6.1 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 median and the mean 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.

6.2 Extracting disaggregation outputs

Disaggregation outputs are particularly complex and they are stored in the datastore in different ways depending on the engine version. Here we will give a few examples for the Disaggregation Demo, which has the flag `individual_rlzs` set. If you run the demos with a recent enough version of the engine (>=3.17) you will see two disaggregation outputs:

1. Disaggregation Outputs Per Realization

2. Statistical Disaggregation Outputs

Such outputs can be exported as usual in CSV format and will generate several files. Users can be interested in extracting a subset of the outputs programmatically, thus avoiding the overhead of exporting more data than needed and having to read the CSV. The way to go is to define an extractor:

```
>> extractor = Extractor(calc_id)
```

and five parameters:

1. kind: the kind of outputs, like Mag, Mag_Dist, Mag_Dist_Eps, etc

2. imt: the IMT, like PGA, SA(1.0), etc

3. site_id: the site ordinal number, like 0, 1, etc

4. poe_id: the ordinal of the PoE, like 0, 1, etc

5. spec: the specifier string, one of "rlzs", "stats", "rlzs-traditional", "stats-traditional"

Here is an example:

```
>> ex = 'disagg?kind=Mag_Dist&imt=PGA&site_id=0&poe_id=0&spec=rlzs-
↪traditional'
>> dic = extractor.get(ex)
```

The dictionary here contains the following keys:

```
>> dic["mag"] # lenght 4
array([5., 6., 7., 8.])
>> dic["dist"] # lenght 21
array([  0.,  10.,  20.,  30.,  40.,  50.,  60.,  70.,  80.,  90., 100.,
       110., 120., 130., 140., 150., 160., 170., 180., 190., 200.])
>> dic["array"].shape
(4, 21, 1, 1)
```

6.3 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]:
array([(   0, 1, 5.05, 0.08456118,  0.15503392, 5., b'Active Shallow Crust
↪', 0.0000000e+00, 90.       , 0.),
       (   1, 1, 5.05, 0.08456119,  0.15503392, 5., b'Active Shallow Crust
↪', 4.4999969e+01, 90.       , 0.),
       (   2, 1, 5.05, 0.08456118,  0.15503392, 5., b'Active Shallow Crust
↪', 3.5999997e+02, 49.999985, 0.),
       ...,
       (1508, 2, 6.15, 0.26448786, -0.7442877 , 5., b'Active Shallow Crust
↪', 0.0000000e+00, 90.       , 0.),
       (1509, 1, 6.15, 0.26448786, -0.74428767, 5., b'Active Shallow Crust
↪', 2.2499924e+02, 50.000004, 0.),
       (1510, 1, 6.85, 0.26448786, -0.74428767, 5., b'Active Shallow Crust
↪', 4.9094699e-04, 50.000046, 0.)],
      dtype=[('rup_id', '<i8'), ('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()
```

READING OUTPUTS WITH PANDAS

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:

```
>> 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))
     site_id stat       imt      value
lvl
0        0  b'mean'  b'PGA'   0.999982
1        0  b'mean'  b'PGA'   0.999949
2        0  b'mean'  b'PGA'   0.999850
3        0  b'mean'  b'PGA'   0.999545
4        0  b'mean'  b'PGA'   0.998634

..     ...     ...      ...        ...
44       0  b'mean'  b'PGA'   0.000000
```

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:

```
>> 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.

7.1 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 are 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 know 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:

```
>> eid = 20  # consider event with ID 20
>> gmf_data = dstore.read_df('gmf_data', index='eid') # engine>3.11
>> gmf_data.loc[eid]
     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
>> 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.

LIMITATIONS OF FLOATING-POINT ARITHMETIC

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

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

is not identical to

```
>>> .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:

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

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

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

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

```
>>> 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:

- Goldberg, D. (1991). What every computer scientist should know about floating-point arith-

metic. *ACM Computing Surveys (CSUR)*, 23(1), 5-48. Reprinted at https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

- https://docs.python.org/3/tutorial/floatingpoint.html

- https://en.wikipedia.org/wiki/Single-precision_floating-point_format

- https://en.wikipedia.org/wiki/Double-precision_floating-point_format

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 [-h] [-v]
          {shell,upgrade_nrml,reduce_smlt,show_attrs,prepare_site_model,
↪nrml_from,shakemap2gmfs,importcalc,run,show,purge,renumber_sm,workers,
↪postzip,plot_assets,db,dbserver,tidy,extract,sample,to_hdf5,ltcsv,
↪reaggregate,restore,mosaic,check_input,dump,info,zip,abort,nrml_to,
↪engine,reset,checksum,export,webui,compare,plot,reduce_sm}
          ...

positional arguments:
  {shell,upgrade_nrml,reduce_smlt,show_attrs,prepare_site_model,nrml_from,
↪shakemap2gmfs,importcalc,run,show,purge,renumber_sm,workers,postzip,
↪plot_assets,db,dbserver,tidy,extract,sample,to_hdf5,ltcsv,reaggregate,
↪restore,mosaic,check_input,dump,info,zip,abort,nrml_to,engine,reset,
↪checksum,export,webui,compare,plot,reduce_sm}
                        available subcommands; use oq <subcmd> --help


options:
  -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 3.17); 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*:

```
$ 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 important information 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:

```
$ 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 information 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:

```
$ 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:

```
$ 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:

```
$ oq info exports
? "aggregate_by" ['csv']
? "disagg_traditional" ['csv']
? "loss_curves" ['csv']
? "losses_by_asset" ['npz']
Aggregate Asset Losses "agglosses" ['csv']
Aggregate Loss Curves Statistics "agg_curves-stats" ['csv']
Aggregate Losses "aggrisk" ['csv']
Aggregate Risk Curves "aggcurves" ['csv']
Aggregated Risk By Event "risk_by_event" ['csv']
Asset Loss Curves "loss_curves-rlzs" ['csv']
Asset Loss Curves Statistics "loss_curves-stats" ['csv']
Asset Loss Maps "loss_maps-rlzs" ['csv', 'npz']
Asset Loss Maps Statistics "loss_maps-stats" ['csv', 'npz']
Asset Risk Distributions "damages-rlzs" ['npz', 'csv']
Asset Risk Statistics "damages-stats" ['csv']
Average Asset Losses "avg_losses-rlzs" ['csv']
Average Asset Losses Statistics "avg_losses-stats" ['csv']
Average Ground Motion Field "avg_gmf" ['csv']
Benefit Cost Ratios "bcr-rlzs" ['csv']
Benefit Cost Ratios Statistics "bcr-stats" ['csv']
Disaggregation Outputs "disagg" ['csv']
Earthquake Ruptures "ruptures" ['csv']
Events "events" ['csv']
```

(continues on next page)

```
Exposure + Risk "asset_risk" ['csv']
Full Report "fullreport" ['rst']
Ground Motion Fields "gmf_data" ['csv', 'hdf5']
Hazard Curves "hcurves" ['csv', 'xml', 'npz']
Hazard Maps "hmaps" ['csv', 'xml', 'npz']
Input Files "input" ['zip']
Mean Conditional Spectra "cs-stats" ['csv']
Realizations "realizations" ['csv']
Source Loss Table "src_loss_table" ['csv']
Total Risk "agg_risk" ['csv']
Uniform Hazard Spectra "uhs" ['csv', 'xml', 'npz']
There are 44 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:

```
$ 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; there are more than fifty exporters and they are always increasing. If you need new exports, please add an issue on GitHub.

9.1 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*:

```
$ 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
```

9.2 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 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:

```
[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:

```
$ oq importcalc 41214
INFO:root:POST https://oq2.wilson.openquake.org//accounts/ajax_login/
INFO:root:GET https://oq2.wilson.openquake.org//v1/calc/41214/extract/
↪oqparam
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
```

9.3 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:

```
$ oq plot_assets --help
usage: oq plot_assets [-h] [calc_id]

Plot the sites and the assets

positional arguments:
  calc_id     a computation id [default: -1]

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

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.

9.4 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:

```
$ 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)
```

(continues on next page)

```
-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:

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

9.5 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:

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

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

```
$ 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
```

9.6 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. It is able to compare many things, depending on the engine version. Here are a few examples:

```
$ oq compare hcurves --help
usage: oq compare hcurves [-h] [-f] [-s] [-r 0] [-a 0.001] imt calc_ids␣
 ↪[calc_ids ...]

Compare the hazard curves of two or more calculations.

positional arguments:
  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 , --samplesites     sites to sample (or fname with site IDs)
  -r 0, --rtol 0         relative tolerance
  -a 0.001, --atol 0.001
                         absolute tolerance

$ oq compare hmaps --help
usage: oq compare hmaps [-h] [-f] [-s] [-r 0] [-a 0.001] imt calc_ids␣
 ↪[calc_ids ...]

Compare the hazard maps of two or more calculations.

positional arguments:
  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 , --samplesites    sites to sample (or fname with site IDs)
  -r 0, --rtol 0        relative tolerance
  -a 0.001, --atol 0.001
                        absolute tolerance


$ oq compare uhs --help
usage: oq compare uhs [-h] [-f] [-s] [-r 0] [-a 0.001] calc_ids [calc_ids␣
→...]


Compare the uniform hazard spectra of two or more calculations.


positional arguments:
  calc_ids              calculation IDs


optional arguments:
  -h, --help            show this help message and exit
  -f, --files           write the results in multiple files
  -s , --samplesites    sites to sample (or fname with site IDs)
  -r 0, --rtol 0        relative tolerance
  -a 0.001, --atol 0.001
                        absolute tolerance
```

Notice the `compare uhs` is able to compare all IMTs at once, so it is the most convenient to use if there are many IMTs.

9.7 Showing calculation attributes

The command *oq show_attrs* offers a convenient way to retrieve the attributes of a calculation without needing to open the datastore with any external tools:

```
$ oq show_attrs -h
usage: oq show_attrs [-h] key [calc_id]


Show the attributes of a HDF5 dataset in the datastore.


positional arguments:
  key          key of the datastore
  calc_id      calculation ID [default: -1]
```

```
options:
  -h, --help  show this help message and exit
```

If the key / is requested, the root attributes are retrieved. For instance:

```
$ oq show_attrs / 4

checksum32 1572793419
date 2023-04-25T08:19:33
engine_version 3.17.0-gitcae0748
input_size 4021
```

If the calculation id is not specified, the value of the requested key is retrieved for the latest calculation.

9.8 Mosaic-related commands

Mosaic-related commands are defined as subcommands of the *oq mosaic* command:

```
$ oq mosaic -h
usage: oq mosaic [-h] {run_site,sample_rups,sample_gmfs} ...

positional arguments:
  {run_site,sample_rups,sample_gmfs}
                        available subcommands; use oq mosaic <subcmd> --
↪help

options:
  -h, --help            show this help message and exit
```

9.8.1 Running a PSHA analysis

The *oq mosaic run_site* subcommand gives the possibility to run a PSHA analysis for a site on the given longitude and latitude, or for multiple sites specified in a CSV file:

```
$ oq mosaic run_site -h
usage: oq mosaic run_site [-h] [--hc HC] [-s SLOWEST] [-c 8] lonlat_or_
↪fname
```

```
Run a PSHA analysis on the given lon, lat

positional arguments:
  lonlat_or_fname         lon,lat of the site to analyze or CSV file

options:
  -h, --help              show this help message and exit
  --hc HC                 previous calculation ID
  -s SLOWEST, --slowest SLOWEST
                          profile and show the slowest operations
  -c 8, --concurrent-jobs 8
                          maximum number of concurrent jobs
```

If a CSV file is provided, it must contain in each row a site identifier starting with the 3-character code of the mosaic model that covers it, and the longitude and latitude of the site, separated by commas.

9.8.2 Sampling ruptures

The *oq mosaic sample_rups* subcommand gives the possibility to sample the ruptures of the given model in the mosaic with an effective investigation time of 100,000 years:

```
$ oq mosaic sample_rups -h
usage: oq mosaic sample_rups [-h] [-s SLOWEST] model

Sample the ruptures of the given model in the mosaic with an effective␣
↪investigation time of 100,000 years

positional arguments:
  model                   3-letter name of the model

options:
  -h, --help              show this help message and exit
  -s SLOWEST, --slowest SLOWEST
                          profile and show the slowest operations
```

9.8.3 Sampling ground motion fields

The *oq mosaic sample_gmfs* subcommand gives the possiblity to sample the gmfs of the given model in the mosaic with an effective investigation time of 100,000 years:

```
$ oq mosaic sample_gmfs -h
usage: oq mosaic sample_gmfs [-h] [-t -1] [-m 0.0] [-e 1000.0] [--hc HC]␣
↪[-s SLOWEST] model

Sample the gmfs of the given model in the mosaic with an effective␣
↪investigation time of 100,000 years

positional arguments:
  model                   3-letter name of the model

options:
  -h, --help              show this help message and exit
  -t -1, --trunclevel -1
                          truncation level (default: the one in job_vs30.
↪ini)
  -m 0.0, --mindist 0.0
                          minimum_distance (default: 0)
  -e 1000.0, --extreme-gmv 1000.0
                          threshold above which a GMV is extreme
  --hc HC                 previous hazard calculation
  -s SLOWEST, --slowest SLOWEST
                          profile and show the slowest operations
```

TEN

LOGIC TREES

Logic trees are documented in the OpenQuake manual (section "Defining Logic Trees"). However some features are only mentioned without giving examples (such as `applyToBranches`) and some recent developments are missing, in particular the `extendModel` feature. Here we will document both.

10.1 extendModel

Starting from engine 3.9 it is possible to define logic trees by adding sources to one or more base models. An example will make things clear:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nrml xmlns:gml="http://www.opengis.net/gml"
      xmlns="http://openquake.org/xmlns/nrml/0.5">
  <logicTree logicTreeID="lt1">
    <logicTreeBranchSet uncertaintyType="sourceModel"
                        branchSetID="bs0">
      <logicTreeBranch branchID="A">
        <uncertaintyModel>common1.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="B">
        <uncertaintyModel>common2.xml</uncertaintyModel>
        <uncertaintyWeight>0.4</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>
    <logicTreeBranchSet uncertaintyType="extendModel" branchSetID="bs1">
      <logicTreeBranch branchID="C">
        <uncertaintyModel>extra1.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
```

(continues on next page)

```xml
        <logicTreeBranch branchID="D">
          <uncertaintyModel>extra2.xml</uncertaintyModel>
          <uncertaintyWeight>0.2</uncertaintyWeight>
        </logicTreeBranch>
        <logicTreeBranch branchID="E">
          <uncertaintyModel>extra3.xml</uncertaintyModel>
          <uncertaintyWeight>0.2</uncertaintyWeight>
        </logicTreeBranch>
      </logicTreeBranchSet>
    </logicTree>
</nrml>
```

In this example there are two base source models, named `commom1.xml` and `common2.xml` and three possibile extensions `extra1.xml`, `extra2.xml` and `extra3.xml`. The engine will generate six effective source models by extending first `common1.xml` and then `common2.xml` with `extra1.xml`, then with `extra2.xml` and then with `extra3.xml` respectively. Notice that `extra1.xml`, `extra2.xml` and `extra3.xml` can be different versions of the same sources with different parameters or geometries, so `extendModel` can be used to implement correlated uncertainties.

Since engine 3.15 it is possible to describe logic trees as python lists (one list for each branchset) and to programmatically generate the realizations by using a simplified logic tree implementation in hazardlib. This is extremely useful. For instance, the logic tree above would be written as follows:

```python
>>> from openquake.hazardlib.lt import build
>>> logictree = build(
...     ['sourceModel', [], ['A', 'common1.xml', 0.6],
...                         ['B', 'common2.xml', 0.4]],
...     ['extendModel', [], ['C', 'extra1.xml', 0.6],
...                         ['D', 'extra2.xml', 0.2],
...                         ['E', 'extra3.xml', 0.2]])
```

and the 6 possible paths can be extracted as follows:

```python
>>> logictree.get_all_paths()   # 2 x 3 paths
['AC', 'AD', 'AE', 'BC', 'BD', 'BE']
```

The empty square brackets means that the branchset should be applied to all branches in the previous branchset and correspond to the `applyToBranches` tag in the XML version of the logic tree. If `applyToBranches` is missing, the logic tree is multiplicative and the total number of paths can be obtained simply by multiplying the number of paths in each branchset. When `applyToBranches` is used, the logic tree becomes additive and the total number of paths can be obtained by summing the number of paths in the different subtrees. For instance, let us extend the previous example by adding another `extendModel` branchset and by using `applyToBranches`:

```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="A">
        <uncertaintyModel>common1.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="B">
        <uncertaintyModel>common2.xml</uncertaintyModel>
        <uncertaintyWeight>0.4</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>
    <logicTreeBranchSet uncertaintyType="extendModel" branchSetID="bs1"
                        applyToBranches="A">
      <logicTreeBranch branchID="C">
        <uncertaintyModel>extra1.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="D">
        <uncertaintyModel>extra2.xml</uncertaintyModel>
        <uncertaintyWeight>0.2</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="E">
        <uncertaintyModel>extra3.xml</uncertaintyModel>
        <uncertaintyWeight>0.2</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>
    <logicTreeBranchSet uncertaintyType="extendModel" branchSetID="bs2"
                        applyToBranches="B">
      <logicTreeBranch branchID="F">
        <uncertaintyModel>extra4.xml</uncertaintyModel>
        <uncertaintyWeight>0.6</uncertaintyWeight>
      </logicTreeBranch>
      <logicTreeBranch branchID="G">
        <uncertaintyModel>extra5.xml</uncertaintyModel>
        <uncertaintyWeight>0.4</uncertaintyWeight>
      </logicTreeBranch>
    </logicTreeBranchSet>
  </logicTree>
</nrml>
```

In this case only 3 + 2 = 5 paths are considered. You can see which are the combinations by building the logic tree:

```
>>> logictree = build(
...      ['sourceModel', [], ['A', 'common1.xml', 0.6],
...                          ['B', 'common2.xml', 0.4]],
...      ['extendModel', ['A'], ['C', 'extra1.xml', 0.6],
...                             ['D', 'extra2.xml', 0.2],
...                             ['E', 'extra3.xml', 0.2]],
...      ['extendModel', ['B'], ['F', 'extra4.xml', 0.6],
...                             ['G', 'extra5.xml', 0.4]])
>>> logictree.get_all_paths()  # 3 + 2 paths
['AC.', 'AD.', 'AE..', 'BF.', 'BG.']
```

`applyToBranches` can be used in different ways. For instance you can attach the second `extendModel` to everything and get 8 paths:

```
>>> logictree = build(
...      ['sourceModel', [], ['A', 'common1.xml', 0.6],
...                          ['B', 'common2.xml', 0.4]],
...      ['extendModel', ['A'], ['C', 'extra1.xml', 0.6],
...                             ['D', 'extra2.xml', 0.2],
...                             ['E', 'extra3.xml', 0.2]],
...      ['extendModel', [], ['F', 'extra4.xml', 0.6],
...                          ['G', 'extra5.xml', 0.4]])
>>> logictree.get_all_paths()  # 3 * 2 + 2 paths
['ACF', 'ACG', 'ADF', 'ADG', 'AEF', 'AEG', 'B.F', 'B.G']
```

The complete realizations can be obtained by not specifying `applyToBranches`:

```
>>> logictree = build(
...      ['sourceModel', [], ['A', 'common1.xml', 0.6],
...                          ['B', 'common2.xml', 0.4]],
...      ['extendModel', [], ['C', 'extra1.xml', 0.6],
...                          ['D', 'extra2.xml', 0.2],
...                          ['E', 'extra3.xml', 0.2]],
...      ['extendModel', [], ['F', 'extra4.xml', 0.6],
...                          ['G', 'extra5.xml', 0.4]])
>>> logictree.get_all_paths() # 12 paths
['ACF', 'ACG', 'ADF', 'ADG', 'AEF', 'AEG', 'BCF', 'BCG', 'BDF', 'BDG',
 ↪'BEF', 'BEG']
```

10.2 The logic tree demo

As another example we will consider the demo `LogicTreeCase2ClassicalPSHA` in the engine distribution; the logic tree has the following structure:

```
>>> lt = build(
...     ['sourceModel', [], ['b11', 'source_model.xml', .333]],
...     ['abGRAbsolute', [], ['b21', '4.6 1.1', .333],
...                          ['b22', '4.5 1.0', .333],
...                          ['b23', '4.4 0.9', .334]],
...     ['abGRAbsolute', [], ['b31', '3.3 1.0', .333],
...                          ['b32', '3.2 0.9', .333],
...                          ['b33', '3.1 0.0', .334]],
...     ['maxMagGRAbsolute', [], ['b41', 7.0, .333],
...                              ['b42', 7.3, .333],
...                              ['b43', 7.6, .334]],
...     ['maxMagGRAbsolute', [], ['b51', 7.5, .333],
...                              ['b52', 7.8, .333],
...                              ['b53', 8.0, .334]],
...     ['Active Shallow Crust', [], ['c11', 'BA08', .5],
...                                  ['c12', 'CY12', .5]],
...     ['Stable Continental Crust', [], ['c21', 'TA02', .5],
...                                      ['c22', 'CA03', .5]])
```

Since the demo is using full enumeration there are 1*3*3*3*3*2*2 = 324 realizations in total that you can build as follows:

```
>>> import numpy
>>> paths = numpy.array(lt.get_all_paths())
>>> for row in paths.reshape(36, 9):
...     print(' '.join(row))
AADGJMO AADGJMP AADGJNO AADGJNP AADGKMO AADGKMP AADGKNO AADGKNP AADGLMO
AADGLMP AADGLNO AADGLNP AADHJMO AADHJMP AADHJNO AADHJNP AADHKMO AADHKMP
AADHKNO AADHKNP AADHLMO AADHLMP AADHLNO AADHLNP AADIJMO AADIJMP AADIJNO
AADIJNP AADIKMO AADIKMP AADIKNO AADIKNP AADILMO AADILMP AADILNO AADILNP
AAEGJMO AAEGJMP AAEGJNO AAEGJNP AAEGKMO AAEGKMP AAEGKNO AAEGKNP AAEGLMO
AAEGLMP AAEGLNO AAEGLNP AAEHJMO AAEHJMP AAEHJNO AAEHJNP AAEHKMO AAEHKMP
AAEHKNO AAEHKNP AAEHLMO AAEHLMP AAEHLNO AAEHLNP AAEIJMO AAEIJMP AAEIJNO
AAEIJNP AAEIKMO AAEIKMP AAEIKNO AAEIKNP AAEILMO AAEILMP AAEILNO AAEILNP
AAFGJMO AAFGJMP AAFGJNO AAFGJNP AAFGKMO AAFGKMP AAFGKNO AAFGKNP AAFGLMO
AAFGLMP AAFGLNO AAFGLNP AAFHJMO AAFHJMP AAFHJNO AAFHJNP AAFHKMO AAFHKMP
AAFHKNO AAFHKNP AAFHLMO AAFHLMP AAFHLNO AAFHLNP AAFIJMO AAFIJMP AAFIJNO
AAFIJNP AAFIKMO AAFIKMP AAFIKNO AAFIKNP AAFILMO AAFILMP AAFILNO AAFILNP
ABDGJMO ABDGJMP ABDGJNO ABDGJNP ABDGKMO ABDGKMP ABDGKNO ABDGKNP ABDGLMO
```

```
ABDGLMP ABDGLNO ABDGLNP ABDHJMO ABDHJMP ABDHJNO ABDHJNP ABDHKMO ABDHKMP
ABDHKNO ABDHKNP ABDHLMO ABDHLMP ABDHLNO ABDHLNP ABDIJMO ABDIJMP ABDIJNO
ABDIJNP ABDIKMO ABDIKMP ABDIKNO ABDIKNP ABDILMO ABDILMP ABDILNO ABDILNP
ABEGJMO ABEGJMP ABEGJNO ABEGJNP ABEGKMO ABEGKMP ABEGKNO ABEGKNP ABEGLMO
ABEGLMP ABEGLNO ABEGLNP ABEHJMO ABEHJMP ABEHJNO ABEHJNP ABEHKMO ABEHKMP
ABEHKNO ABEHKNP ABEHLMO ABEHLMP ABEHLNO ABEHLNP ABEIJMO ABEIJMP ABEIJNO
ABEIJNP ABEIKMO ABEIKMP ABEIKNO ABEIKNP ABEILMO ABEILMP ABEILNO ABEILNP
ABFGJMO ABFGJMP ABFGJNO ABFGJNP ABFGKMO ABFGKMP ABFGKNO ABFGKNP ABFGLMO
ABFGLMP ABFGLNO ABFGLNP ABFHJMO ABFHJMP ABFHJNO ABFHJNP ABFHKMO ABFHKMP
ABFHKNO ABFHKNP ABFHLMO ABFHLMP ABFHLNO ABFHLNP ABFIJMO ABFIJMP ABFIJNO
ABFIJNP ABFIKMO ABFIKMP ABFIKNO ABFIKNP ABFILMO ABFILMP ABFILNO ABFILNP
ACDGJMO ACDGJMP ACDGJNO ACDGJNP ACDGKMO ACDGKMP ACDGKNO ACDGKNP ACDGLMO
ACDGLMP ACDGLNO ACDGLNP ACDHJMO ACDHJMP ACDHJNO ACDHJNP ACDHKMO ACDHKMP
ACDHKNO ACDHKNP ACDHLMO ACDHLMP ACDHLNO ACDHLNP ACDIJMO ACDIJMP ACDIJNO
ACDIJNP ACDIKMO ACDIKMP ACDIKNO ACDIKNP ACDILMO ACDILMP ACDILNO ACDILNP
ACEGJMO ACEGJMP ACEGJNO ACEGJNP ACEGKMO ACEGKMP ACEGKNO ACEGKNP ACEGLMO
ACEGLMP ACEGLNO ACEGLNP ACEHJMO ACEHJMP ACEHJNO ACEHJNP ACEHKMO ACEHKMP
ACEHKNO ACEHKNP ACEHLMO ACEHLMP ACEHLNO ACEHLNP ACEIJMO ACEIJMP ACEIJNO
ACEIJNP ACEIKMO ACEIKMP ACEIKNO ACEIKNP ACEILMO ACEILMP ACEILNO ACEILNP
ACFGJMO ACFGJMP ACFGJNO ACFGJNP ACFGKMO ACFGKMP ACFGKNO ACFGKNP ACFGLMO
ACFGLMP ACFGLNO ACFGLNP ACFHJMO ACFHJMP ACFHJNO ACFHJNP ACFHKMO ACFHKMP
ACFHKNO ACFHKNP ACFHLMO ACFHLMP ACFHLNO ACFHLNP ACFIJMO ACFIJMP ACFIJNO
ACFIJNP ACFIKMO ACFIKMP ACFIKNO ACFIKNP ACFILMO ACFILMP ACFILNO ACFILNP
```

The engine is computing all such realizations; after running the calculations you will see an output called "Realizations". If you export it, you will get a CSV file with the following structure:

```
#,,"generated_by='OpenQuake engine 3.13...'"
rlz_id,branch_path,weight
0,AAAAA~AA,3.0740926e-03
1,AAAAA~AB,3.0740926e-03
...
322,ACCCC~BA,3.1111853e-03
323,ACCCC~BB,3.1111853e-03
```

For each realization there is a `branch_path` string which is split in two parts separated by a tilde. The left part describes the branches of the source model logic tree and the right part the branches of the gmpe logic tree. In past versions of the engine the branch path was using directly the branch IDs, so it was easy to assess the correspondence between each realization and the associated branches.

Unfortunately, we had to remove that direct correspondence in engine 3.11. The reason is that engine is used in situations where the logic tree has billions of billions of billions ... of billions potential realizations, with hundreds of branchsets. If you have 100 branchsets and the branch IDs are 10 characters long, each branch path will be 1000 characters long and impossible to display. The

compact representation requires only 1-character per branchset instead. It is possible to pass from the compact representation to the original branch IDs by using the command `oq show branches`:

```
$ oq show branches
| branch_id | abbrev | uvalue             |
|-----------+--------+--------------------|
| b11       | A0     | source_model.xml   |
| b21       | A1     | 4.60000 1.10000    |
| b22       | B1     | 4.50000 1.00000    |
| b23       | C1     | 4.40000 0.90000    |
| b31       | A2     | 3.30000 1.00000    |
| b32       | B2     | 3.20000 0.90000    |
| b33       | C2     | 3.10000 0.80000    |
| b41       | A3     | 7.00000            |
| b42       | B3     | 7.30000            |
| b43       | C3     | 7.60000            |
| b51       | A4     | 7.50000            |
| b52       | B4     | 7.80000            |
| b53       | C4     | 8.00000            |
| b11       | A0     | [BooreAtkinson2008] |
| b12       | B0     | [ChiouYoungs2008]  |
| b21       | A1     | [ToroEtAl2002]     |
| b22       | B1     | [Campbell2003]     |
```

The first character of the `abbrev` specifies the branch number ("A" means the first branch, "B" the second, etc) while the other characters are the branch set number starting from zero. The format works up to 184 branches per branchset, using printable UTF8 characters. For instance the realization #322 has the following branch path in compact form:

```
ACCCC~BA
```

which will expand to the following abbreviations (considering that fist "A" corresponds to the branchset 0, the first "C" to branchset 1, the second "C" to branchset 2, the third "C" to branchset 3, the fourth "C" to branchset 4, "B" to branchset 0 of the GMPE logic tree and the last "A" to branchset 1 of the GMPE logic tree):

```
A0 C1 C2 C3 C4 ~ B0 A1
```

and then, using the correspondence table `abbrev->uvalue`, to:

```
"source_model.xml" "4.4 0.9" "3.1 0.8" "7.6" "8.0" ~
"[ChiouYoungs2008]" "[ToroEtAl2002]"
```

For convenience, the engine provides a simple command to display the content of a realization, given the realization number:

```
$ oq show rlz:322
| uncertainty_type        | uvalue            |
|-------------------------+-------------------|
| sourceModel             | source_model.xml  |
| abGRAbsolute            | 4.40000 0.90000   |
| abGRAbsolute            | 3.10000 0.80000   |
| maxMagGRAbsolute        | 7.60000           |
| maxMagGRAbsolute        | 8.00000           |
| Active Shallow Crust    | [ChiouYoungs2008] |
| Stable Continental Crust | [ToroEtAl2002]   |
```

NB: the commands *oq show branches* and *oq show rlz* are new in engine 3.13: they may change in the future and the string representation of the branch path may change too. It has already changed twice in engine 3.11 and engine 3.12. You cannot rely on it across engine versions.

10.3 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'>
```

(continues on next page)

```
            <logicTreeBranchSet uncertaintyType="gmpeModel" branchSetID=
↪"bs1"
                applyToTectonicRegionType="active shallow crust">

            <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.

10.4 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 * 5 * 2 * 4 * 4 * 1 * 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 * 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

$total\_num\_rlzs = 3 * 2 = 6$

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>_<GMPE of second region type>*:

#	lt_path
0	*A_D*
1	*B_D*
2	*C_D*
3	*A_E*
4	*B_E*
5	*C_E*

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.

10.5 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
```

(continues on next page)

```
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.

ELEVEN

SOURCE SPECIFIC LOGIC TREES

There are situations in which the hazard model is comprised by a small number of sources, and for each source there is an individual logic tree managing the uncertainty of a few parameters. In such situations we say that we have a *Source Specific Logic Tree*.

Such situation is esemplified by the demo that you can find in the directory `demos/hazard/` `LogicTreeCase2ClassicalPSHA`, which has the following logic tree, in XML form:

As you can see, each branchset has an `applyToSources` attribute, pointing to one of the two sources in the hazard model, therefore we have a source specific logic tree.

In compact form we can represent the logic tree as the composition of two source specific logic trees with the following branchsets:

```
src "1": [<abGRAbsolute(3)>, <maxMagGRAbsolute(3)>]
src "2": [<abGRAbsolute(3)>, <maxMagGRAbsolute(3)>]
```

The `(X)` notation denotes the number of branches for each branchset and multiplying such numbers we can deduce the size of the full logic tree (ignoring the gsim logic tree for sake of simplificity):

```
(3 x 3 for src "1") x (3 x 3 for src "2") = 81 realizations
```

It is possible to see the full logic tree as the product of two source specific logic trees each one with 9 realizations. The interesting thing it that the engine will require storage and computational power proportional to 9 + 9 = 18 basic components and not to the 9 * 9 = 81 final realizations. In general if there are N source specific logic trees, each one generating R_i realizations with i in the range 0..N-1, the number of basic components and final realizations are respectively:

```
C = sum(R_i)
R = prod(R_i)
```

In the demo the storage is over 4 times less (18 vs 81); in more complex cases the gain than can be much more impressive. For instance the ZAF model in our mosaic (the national model for South Africa) contains a source specific logic tree with 22 sources that can be decomposed as follows:

In other words, by storing only 186 components we can save enough information to build 24_959_374_950_829_916_160 realizations, with a gain of over 10^17!

11.1 Extracting the hazard curves

While it is impossible to compute the hazard curves for 24_959_374_950_829_916_160 realizations, it is quite possible to get the source-specific hazard curves. To this end the engine provides a class `HcurvesGetter` with a method `.get_hcurves` which is able to retrieve all the curves associated to the realizations of the logic tree associated to a specific source. Here is the usage:

```python
from openquake.commonlib.datastore import read
from openquake.calculators.getters import HcurvesGetter

getter = HcurvesGetter(read(-1))
print(getter.get_hcurves('1', 'PGA'))  # array of shape (Rs, L)
```

Looking at the source-specific realizations is useful to assess if the logic tree can be collapsed.

11.2 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.

LOGIC TREE SAMPLING STRATEGIES

Stating from version 3.10, the OpenQuake engine suppports 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 Wit 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:

```
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.

```
>>> import collections
>>> from openquake.hazardlib.lt import random_sample
>>> 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:

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

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

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

```
>>> paths = random_sample(bsets, 100, 45, 'late_latin')
>>> collections.Counter(paths)
Counter({'YC': 18, 'XA': 18, 'XC': 16, 'YA': 16, 'XB': 16, 'YB': 16})
```

12.1 GMPE logic trees with weighted IMTs

In order to support Canada's 5th Generation seismic hazard model, the engine now has the ability to manage GMPE logic trees where the weight assigned to each GMPE may be different for each IMT. For instance you could have a particular GMPE applied to PGA with a certain weight, to SA(0.1) with a different weight, and to SA(1.0) with yet another weight. The user may want to assign a higher weight to the IMTs where the GMPE has a small uncertainty and a lower weight to the IMTs with a large uncertainty. Moreover a particular GMPE may not be applicable for some periods, and in that case the user can assign to a zero weight for those periods, in which case the engine will ignore it entirely for those IMTs. 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">
```

(continues on next page)

```xml
            <uncertaintyModel>BooreEtAl1997GeometricMean</uncertaintyModel>
            <uncertaintyWeight>0.33</uncertaintyWeight>
            <uncertaintyWeight imt="PGA">0.25</uncertaintyWeight>
            <uncertaintyWeight imt="SA(0.5)">0.5</uncertaintyWeight>
            <uncertaintyWeight imt="SA(1.0)">0.5</uncertaintyWeight>
            <uncertaintyWeight imt="SA(2.0)">0.5</uncertaintyWeight>
        </logicTreeBranch>
        <logicTreeBranch branchID="SadighEtAl1997">
            <uncertaintyModel>SadighEtAl1997</uncertaintyModel>
            <uncertaintyWeight>0.33</uncertaintyWeight>
            <uncertaintyWeight imt="PGA">0.25</uncertaintyWeight>
            <uncertaintyWeight imt="SA(0.5)">0.5</uncertaintyWeight>
            <uncertaintyWeight imt="SA(1.0)">0.5</uncertaintyWeight>
            <uncertaintyWeight imt="SA(2.0)">0.5</uncertaintyWeight>
        </logicTreeBranch>
        <logicTreeBranch branchID="MunsonThurber1997Hawaii">
            <uncertaintyModel>MunsonThurber1997Hawaii</uncertaintyModel>
            <uncertaintyWeight>0.34</uncertaintyWeight>
            <uncertaintyWeight imt="PGA">0.25</uncertaintyWeight>
            <uncertaintyWeight imt="SA(0.5)">0.0</uncertaintyWeight>
            <uncertaintyWeight imt="SA(1.0)">0.0</uncertaintyWeight>
            <uncertaintyWeight imt="SA(2.0)">0.0</uncertaintyWeight>
        </logicTreeBranch>
        <logicTreeBranch branchID="Campbell1997">
            <uncertaintyModel>Campbell1997</uncertaintyModel>
            <uncertaintyWeight>0.0</uncertaintyWeight>
            <uncertaintyWeight imt="PGA">0.25</uncertaintyWeight>
            <uncertaintyWeight imt="SA(0.5)">0.0</uncertaintyWeight>
            <uncertaintyWeight imt="SA(1.0)">0.0</uncertaintyWeight>
            <uncertaintyWeight imt="SA(2.0)">0.0</uncertaintyWeight>
        </logicTreeBranch>
</logicTreeBranchSet>
```

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 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.

CLASSICAL PSHA

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.

13.1 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, of 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 about 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 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:

```
Our experience tells us that THE PERFORMANCE BOTTLENECKS OF THE
REDUCED CALCULATION ARE TOTALLY DIFFERENT FROM THE BOTTLENECKS OF
THE FULL CALCULATION. Do not trust your performance intuition.
```

13.2 Classical PSHA for Europe (SHARE)

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 tolerence).
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%, sids=[.
↪..]
```

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.

rms-diff = sqrt<(hmap1 - hmap2)^2> # mediating on all the sites

As to be expected, the differences are larger for maps with a smaller poe, i.e. a larger return period.

13.2. Classical PSHA for Europe (SHARE) 79

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!

13.3 Collapsing the GMPE logic tree

Some hazard models have GMPE logic trees which are insanely large. For instance the GMPE logic tree for the latest European model (ESHM20) contains 961,875 realizations. This causes two issues:

1. it is impossible to run a calculation with full enumeration, so one must use sampling

2. when one tries to increase the number of samples to study the stability of the mean hazard curves, the calculation runs out of memory

Fortunately, it is possible to compute the *exact mean hazard curves* by collapsing the GMPE logic tree. This is a simple as listing the name of the branchsets in the GMPE logic tree that one wants to collapse. For instance in the case of ESHM20 model there are the following 6 branchsets:

1. Shallow_Def (19 branches)

2. CratonModel (15 branches)

3. BCHydroSubIF (15 branches)

4. BCHydroSubIS (15 branches)

5. BCHydroSubVrancea (15 branches)

6. Volcanic (1 branch)

By setting in the job.ini the following parameters

```
number_of_logic_tree_samples = 0
collapse_gsim_logic_tree = Shallow_Def CratonModel BCHydroSubIF␣
↪BCHydroSubIS BCHydroSubVrancea Volcanic
```

it is possible to collapse completely the GMPE logic tree, i.e. going from 961,875 realizations to 1. Then the memory issues are solved and one can assess the correct values of the mean hazard curves. Then it is possible to compare with the value produce with sampling and assess how much they can be trusted.

NB: the `collapse_gsim_logic_tree` feature is rather old but only for engine versions >=3.13 it produces the exact mean curves (using the `AvgPoeGMPE`); otherwise it will produce a different kind of collapsing (using the `AvgGMPE`).

FOURTEEN

PARAMETRIC GMPES

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.

14.1 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.

14.2 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:

```
<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:

```
[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

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

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

```
<logicTreeBranch branchID="b11">
    <uncertaintyModel gmpe_table="Wcrust_low_rhypo.hdf5">
        GMPETable
    </uncertaintyModel>
    <uncertaintyWeight>0.16</uncertaintyWeight>
</logicTreeBranch>
    <logicTreeBranch branchID="b12">
    <uncertaintyModel gmpe_table="Wcrust_med_rhypo.hdf5">
        GMPETable
    </uncertaintyModel>
    <uncertaintyWeight>0.68</uncertaintyWeight>
</logicTreeBranch>
    <logicTreeBranch branchID="b13">
    <uncertaintyModel gmpe_table="Wcrust_high_rhypo.hdf5">
        GMPETable
    </uncertaintyModel>
    <uncertaintyWeight>0.16</uncertaintyWeight>
</logicTreeBranch>
```

This is a legacy syntax, which is still supported and will likely be supported forever, but we recommend 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 GMPEs: this is why we switched to TOML.

14.3 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:  # good
            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:  # bad
            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:  # good
            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.

14.4 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:

```
<logicTreeBranch branchID="b1">
   <uncertaintyModel>
     [MultiGMPE."PGA".AkkarBommer2010]
     [MultiGMPE."SA(0.1)".SadighEtAl1997]
   </uncertaintyModel>
   <uncertaintyWeight>1.0</uncertaintyWeight>
 </logicTreeBranch>
```

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:

```
{'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):

```
<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>
```

14.5 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 open-quake/qa_tests/data/classical/case_34:

```
<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

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

14.6 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:

```
<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:

https://github.com/gem/oq-engine/blob/engine-3.17/openquake/hazardlib/gsim/mgmpe/modifiable_gmpe.py

MULTIPOINTSOURCES

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:

```
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>
    <multiMFD
    kind="incrementalMFD"
    size=2
    >
        <bin_width>
            2.0 2.0
        </bin_width>
        <min_mag>
            4.5 4.5
        </min_mag>
        <occurRates>
            0.10 0.05 0.40 0.20 0.10
        </occurRates>
        <lengths>
            2 3
        </lengths>
    </multiMFD>
    <nodalPlaneDist>
        <nodalPlane dip="20.0" probability="0.5" rake="3.0" strike="1.0"/>
        <nodalPlane dip="2.0" probability="0.5" rake="4.0" strike="2.0"/>
    </nodalPlaneDist>
    <hypoDepthDist>
        <hypoDepth depth="14.0" probability="1.0"/>
    </hypoDepthDist>
</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:

```
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.

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 kinds 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:
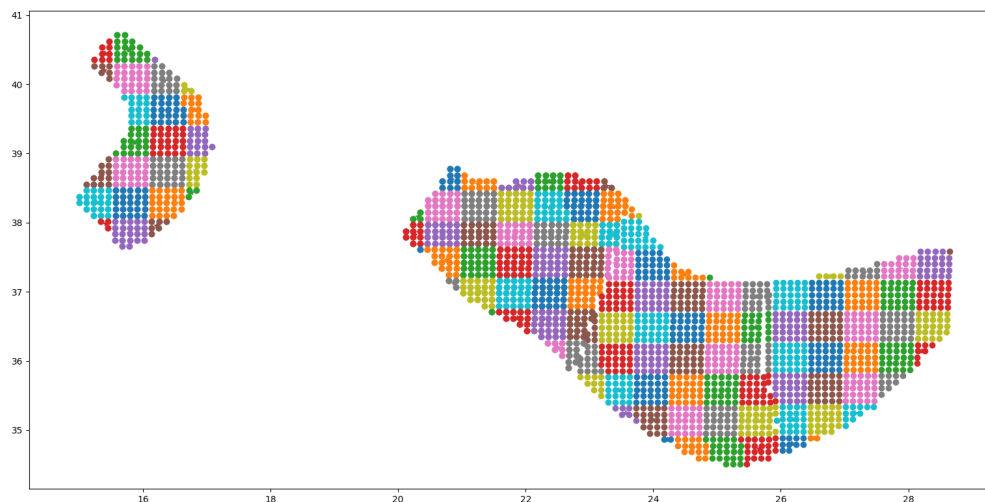
For distant sites it is possible to use the large grid (i.e. the CollapsePointSources) without losing much precision, while for close points the original sources must be used.

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 (in reality things are a bit more complicated, since the engine also consider the fact that ruptures have a finite size, but you get the idea).

16.1 Application: making the Canada model 26x faster

In order to give a concrete example, I ran the Canada 2015 model on 7 cities by using the following `site_model.csv` file:

custom_site_id	lon	lat	vs30	z1pt0	z2pt5
montre	-73	45	368	393.6006	1.391181
calgar	-114	51	451	290.6857	1.102391
ottawa	-75	45	246	492.3983	2.205382
edmont	-113	53	372	389.0669	1.374081
toront	-79	43	291	465.5151	1.819785
winnip	-97	50	229	499.7842	2.393656
vancou	-123	49	600	125.8340	0.795259

Notice that we are using a `custom_site_id` field to identify the cities. This is possible only in engine versions >= 3.13, where `custom_site_id` has been extended to accept strings of at most 6 characters, while before only integers were accepted (we could have used a zip code instead).

If no special approximations are used, the calculation is extremely slow, since the model is extremely large. On the the GEM cluster (320 cores) it takes over 2 hours to process the 7 cities. The dominating operation, as of engine 3.13, is "computing mean_std" which takes, in total, 925,777 seconds split across the 320 cores, i.e. around 48 minutes per core. This is way too much and it would make impossible to run the full model with ~138,000 sites. An analysis shows that the calculation time is totally dominated by the point sources. Moreover, the engine prints a warning saying that I should use the `pointsource_distance` approximation. Let's do so, i.e. let us set

`pointsource_distance = 50`

in the job.ini file. That alone triples the speed of the engine, and the calculation times in "computing mean_std" goes down to 324,241 seconds, i.e. 16 minutes per core, in average. An analysis of the hazard curves shows that there is practically no difference between the original curves and the ones computed with the approximation on:

```
$ oq compare hcurves PGA <first_calc_id> <second_calc_id>
There are no differences within the tolerances atol=0.001, rtol=0%,␣
 ↪sids=[0 1 2 3 4 5 6]
```

However, this is not enough. We are still too slow to run the full model in a reasonable amount of time. Enters the point source gridding. By setting

```
ps_grid_spacing=50
```

we can spectacularly reduce the calculation time to 35,974s, down by nearly an order of magnitude! This time `oq compare hcurves` produces some differences on the last city but they are minor and not affecting the hazard maps:

```
$ oq compare hmaps PGA <first_calc_id> <third_calc_id>
There are no differences within the tolerances atol=0.001, rtol=0%,␣
 ↪sids=[0 1 2 3 4 5 6]
```

The following table collects the results:

operation	calc_time	approx	speedup
computing mean_std	925_777	no approx	1x
computing mean_std	324_241	pointsource_distance	3x
computing mean_std	35_974	ps_grid_spacing	26x

It should be noticed that if you have 130,000 sites it is likely that there will be a few sites where the point source gridding approximation gives results quite different for the exact results. The commands `oq compare` allows you to figure out which are the problematic sites, where they are and how big is the difference from the exact results.

You should take into account that even the "exact" results have uncertainties due to all kind of reasons, so even a large difference can be quite acceptable. In particular if the hazard is very low you can ignore any difference since it will have no impact on the risk.

Points with low hazard are expected to have large differences, this is why by default *oq compare* use an absolute tolerance of 0.001g, but you can raise that to 0.01g or more. You can also give a relative tolerance of 10% or more. Internally `oq compare` calls the function `numpy.allclose` see https://numpy.org/doc/stable/reference/generated/numpy.allclose.html for a description of how the tolerances work.

By increasing the `pointsource_distance` parameter and decreasing the `ps_grid_spacing` parameter one can make the approximation as precise as wanted, at the expense of a larger runtime.

NB: the fact that the Canada model with 7 cities can be made 26 times faster does not mean that the same speedup apply when you consider the full 130,000+ sites. A test with `ps_grid_spacing=pointsource_distance=50` gives a speedup of 7 times, which is still very significant.

16.2 How to determine the "right" value for the ps_grid_spacing parameter

The trick is to run a sensitivity analysis on a reduced calculation. Set in the job.ini something like this:

```
sensitivity_analysis = {'ps_grid_spacing': [0, 20, 40, 60]}
```

and then run:

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

This will run sequentially 4 calculations with different values of the `ps_grid_spacing`. The first calculation, the one with `ps_grid_spacing=0`, is the exact calculation, with the approximation disabled, to be used as reference.

Notice that setting the environment variable `OQ_SAMPLE_SITES=.01` will reduced by 100x the number of sites: this is essential in order to make the calculation times acceptable in large calculations.

After running the 4 calculations you can compare the times by using `oq show performance` and the precision by using `oq compare`. From that you can determine which value of the `ps_grid_spacing` gives a good speedup with a decent precision. Calculations with plenty of nodal planes and hypocenters will benefit from lower values of `ps_grid_spacing` while calculations with a single nodal plane and hypocenter for each source will benefit from higher values of `ps_grid_spacing`.

If you are interested only in speed and not in precision, you can set `calculation_mode=preclassical`, run the sensitivity analysis in parallel very quickly and then use the `ps_grid_spacing` value corresponding to the minimum weight of the source model, which can be read from the logs. Here is the trick to run the calculations in parallel:

```
$ oq engine --multi --run job.ini -p calculation_mode=preclassical
```

And here is how to extract the weight information, in the example of Alaska, with job IDs in the range 31692-31695:

```
$ oq db get_weight 31692
<Row(description=Alaska{'ps_grid_spacing': 0}, message=tot_weight=1_929_
↪504, max_weight=120_594, num_sources=150_254)>
$ oq db get_weight 31693
<Row(description=Alaska{'ps_grid_spacing': 20}, message=tot_weight=143_
↪748, max_weight=8_984, num_sources=22_727)>
$ oq db get_weight 31694
<Row(description=Alaska{'ps_grid_spacing': 40}, message=tot_weight=142_
↪564, max_weight=8_910, num_sources=6_245)>
```

```
$ oq db get_weight 31695
<Row(description=Alaska{'ps_grid_spacing': 60}, message=tot_weight=211_
↪542, max_weight=13_221, num_sources=3_103)>
```

The lowest weight is 142_564, corresponding to a `ps_grid_spacing` of 40km; since the weight
is 13.5 times smaller than the weight for the full calculation (1_929_504), this is the maximum
speedup that we can expect from using the approximation.

Note 1: the weighting algorithm changes at every release, so only relative weights at a fixed release
are meaningful and it does not make sense to compare weights across engine releases.

Note 2: the precision and performance of the `ps_grid_spacing` approximation change at every
release: you should not expect to get the same numbers and performance across releases even if the
model is the same and the parameters are the same.

SEVENTEEN

DISAGG_BY_SRC

Given a system of various sources affecting a specific site, one very common question to ask is: what are the more relevant sources, i.e. which sources contribute the most to the mean hazard curve? The engine is able to answer such question by setting the `disagg_by_src` flag in the job.ini file. When doing that, the engine saves in the datastore a 4-dimensional ArrayWrapper called `mean_rates_by_src` with dimensions (site ID, intensity measure type, intensity measure level, source ID). From that it is possible to extract the contribution of each source to the mean hazard curve (interested people should look at the code in the function `check_disagg_by_src`). The ArrayWrapper `mean_rates_by_src` can also be converted into a pandas DataFrame, then getting something like the following:

```
>> dstore['mean_rates_by_src'].to_dframe().set_index('src_id')
              site_id  imt  lvl        value
ASCTRAS407          0  PGA    0  9.703749e-02
IF-CFS-GRID03       0  PGA    0  3.720510e-02
ASCTRAS407          0  PGA    1  6.735009e-02
IF-CFS-GRID03       0  PGA    1  2.851081e-02
ASCTRAS407          0  PGA    2  4.546237e-02
...               ...  ...  ...           ...
IF-CFS-GRID03       0  PGA   17  6.830692e-05
ASCTRAS407          0  PGA   18  1.072884e-06
IF-CFS-GRID03       0  PGA   18  1.275539e-05
ASCTRAS407          0  PGA   19  1.192093e-07
IF-CFS-GRID03       0  PGA   19  5.960464e-07
```

The `value` field here is the probability of exceedence in the hazard curve. The `lvl` field is an integer corresponding to the intensity measure level in the hazard curve.

In engine 3.15 we introduced the so-called "colon convention" on source IDs: if you have many sources that for some reason should be collected together - for instance because they all account for seismicity in the same tectonic region, or because they are components of a same source but are split into separate sources by magnitude - you can tell the engine to collect them into one source in the `mean_rates_by_src` matrix. The trick is to use IDs with the same prefix, a colon, and then a numeric index. For instance, if you had 3 sources with IDs `src_mag_6.65`, `src_mag_6.`

75, `src_mag_6.85`, fragments of the same source with different magnitudes, you could change their IDs to something like `src:0`, `src:1`, `src:2` and that would reduce the size of the matrix `mean_rates_by_src` by 3 times by collecting together the contributions of each source. There is no restriction on the numeric indices to start from 0, so using the names `src:665`, `src:675`, `src:685` would work too and would be clearer: the IDs should be unique, however.

If the IDs are not unique and the engine determines that the underlying sources are different, then an extension "semicolon + incremental index" is automatically added. This is useful when the hazard modeler wants to define a model where the more than one version of the same source appears in one source model, having changed some of the parameters, or when varied versions of a source appear in each branch of a logic tree. In that case, the modeler should use always the exact same ID (i.e. without the colon and numeric index): the engine will automatically distinguish the sources during the calculation of the hazard curves and consider them the same when saving the array `mean_rates_by_src`: you can see an example in the test `qa_tests_data/classical/case_20/job_bis.ini` in the engine code base. In that case the `source_info` dataset will list 7 sources `CHAR1;0 CHAR1;1 CHAR1;2 COMFLT1;0 COMFLT1;1 SFLT1;0 SFLT1;1` but the matrix `mean_rates_by_src` will see only three sources `CHAR1 COMFLT1 SFLT1` obtained by composing together the versions of the underlying sources.

In version 3.15 `mean_rates_by_src` was extended to work with mutually exclusive sources, i.e. for the Japan model. You can see an example in the test `qa_tests_data/classical/case_27`. However, the case of mutually exclusive ruptures - an example is the New Madrid cluster in the USA model - is not supported yet.

In some cases it is tricky to discern whether use of the colon convention or identical source IDs is appropriate. The following list indicates several possible cases that a user may encounter, and the appropriate approach to assigning source IDs. Note that this list includes the cases that have been tested so far, and is not a comprehensive list of all cases that may arise.

1. Sources in the same source group/source model are scaled alternatives of each other. For example, this occurs when for a given source, epistemic uncertainties such as occurrence rates or geometries are considered, but the modeller has pre-scaled the rates rather than including the alternative hypothesis in separate logic tree branches.

   Naming approach: identical IDs.

2. Sources in different files are alternatives of each other, e.g. each is used in a different branch of the source model logic tree.

   Naming approach: identical IDs.

3. A source is defined in OQ by numerous sources, either in the same file or different ones. For example, one could have a set of non-parametric sources, each with many rutpures, that are grouped together into single files by magnitude. Or, one could have many point sources that together represent the seismicity from one source.

   Naming approach: colon convention

4. One source consists of many mutually exclusive sources, as in `qa_tests_data/classical/case_27`.

Naming approach: colon convention

Cases 1 and 2 could include include more than one source typology, as in `qa_tests_data/classical/case_79`.

NB: `disagg_by_src` can be set to true only if the `ps_grid_spacing` approximation is disabled. The reason is that the `ps_grid_spacing` approximation builds effective sources which are not in the original source model, thus breaking the connection between the values of the matrix and the original sources.

THE POST-PROCESSING FRAMEWORK AND VECTOR-VALUED PSHA CALCULATIONS

Since version 3.17 the OpenQuake engine has special support for custom postprocessors. A post-processor is a Python module located in the directory `openquake/calculators/postproc` and containing a `main` function with signature:

```python
def main(dstore, [csm], ...):
    ...
```

Post-processors are called after a classical or preclassical calculation: the `dstore` parameter is a DataStore instance corresponding to the calculation, while the `csm` parameter is a Composite-SourceModel instance (it can be omitted if not needed).

The `main` function is called when the user sets in the job.ini file the parameters `postproc_func` and `postproc_args`. `postproc_func` is the dotted name of the postprocessing function (in the form `modulename.funcname` where `funcname` is normally `main`) and `postproc_args` is a dictionary of literal arguments that get passed to the function; if not specified the empty dictionary is passed. This happens for istance for the conditional spectrum post-processor since it does not require additional arguments with respect to the ones in `dstore['oqparam']`.

The post-processing framework was put in place in order to run VPSHA calculations. The user can find an example in `qa_tests_data/postproc/case_mrd`. In the job.ini file there are the lines:

```
postproc_func = compute_mrd.main
postproc_args = {
  'imt1': 'PGA',
  'imt2': 'SA(0.05)',
  'cross_correlation': 'BakerJayaram2008',
  'seed': 42,
  'meabins': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6],
  'sigbins': [0.2, 0.3, 0.4, 0.5, 0.6, 0.7],
  'method': 'indirect'}
```

while the postprocessor module `openquake.calculators.postproc.compute_mrd` contains the function

```
# inside openquake.calculators.postproc.compute_mrd
def main(dstore, imt1, imt2, cross_correlation, seed, meabins, sigbins,
         method='indirect'):
    ...
```

Inside `main` there is code to create the dataset `mrd` which contains the Mean Rate Distribution as an array of shape L1 x L1 x N where L1 is the number of levels per IMT minus 1 and N the number of sites (normally 1).

While the postprocessing part for VPSHA calculations is computationally intensive, it is much more common to have a light postprocessing, i.e. faster than the classical calculation it depends on. In such situations the postprocessing framework really shines, since it is possible to reuse the original calculation via the standard `--hc` switch, i.e. you can avoid repeating multiple times the same classical calculation if you are interested in running the postprocessor with different parameters. In that situation the `main` function will get a DataStore instance with an attribute `parent` corresponding to the DataStore of the original calculation.

The postprocessing framework also integrates very well with interactive development (think of Jupyter notebooks). The following lines are all you need to create a child datastore where the postprocessing function can store its results after reading the data from the calculation datastore:

```
>> from openquake.commonlib.datastore import read, build_dstore_log
>> from openquake.calculators.postproc import mypostproc
>> dstore, log = build_dstore_log(parent=read(calc_id))
>> with log:
..     mypostproc.main(dstore)
```

18.1 The conditional spectrum post-processor

Since version 3.17 the engine includes an experimental post-processor which is able to compute the conditional spectrum.

The implementation was adapted from the paper *Conditional Spectrum Computation Incorporating Multiple Causal Earthquakes and Ground-Motion Prediction Models* by Ting Lin, Stephen C. Harmsen, Jack W. Baker, and Nicolas Luco (http://citeseerx.ist.psu.edu/viewdoc/download?doi= 10.1.1.845.163&rep=rep1&type=pdf) and it is rather sophisticated.

In order to perform a conditional spectrum calculation you need to specify, in addition to the usual parameter of a classical calculation:

1. a reference intensity measure type (i.e. `imt_ref = SA(0.2)`)

2. a cross correlation model (i.e. `cross_correlation = BakerJayaram2008`)

3. a set of poes (i.e. `poes = 0.01 0.1`)

The engine will compute a mean conditional spectrum for each `poe` and site, as well as the usual mean uniform hazard spectra. The following restrictions are enforced:

1. the IMTs can only be of type `SA` and `PGA`

2. the source model cannot contain mutually exclusive sources (i.e. you cannot compute the conditional spectrum for the Japan model)

An example can be found in the engine repository, in the directory openquake/qa_tests_data/conditional_spectrum/case_1. If you run it, you will get something like the following:

```
$ oq engine --run job.ini
...
 id | name
261 | Full Report
262 | Hazard Curves
260 | Mean Conditional Spectra
263 | Realizations
264 | Uniform Hazard Spectra
```

Exporting the output 260 will produce two files `conditional-spectrum-0.csv` and `conditional-spectrum-1.csv`; the first will refer to the first `poe`, the second to the second `poe`. Each file will have a structure like the following:

```
#,,,,"generated_by='OpenQuake engine 3.13.0-gitd78d717e66', start_date=
↪'2021-10-13T06:15:20', checksum=3067457643, imls=[0.99999, 0.61470],␣
↪site_id=0, lon=0.0, lat=0.0"
sa_period,val0,std0,val1,std1
0.00000E+00,1.02252E+00,2.73570E-01,7.53388E-01,2.71038E-01
1.00000E-01,1.99455E+00,3.94498E-01,1.50339E+00,3.91337E-01
2.00000E-01,2.71828E+00,9.37914E-09,1.84910E+00,9.28588E-09
3.00000E-01,1.76504E+00,3.31646E-01,1.21929E+00,3.28540E-01
1.00000E+00,3.08985E-01,5.89767E-01,2.36533E-01,5.86448E-01
```

The number of columns will depend from the number of sites. The conditional spectrum calculator, like the disaggregation calculator, is meant to be run on a very small number of sites, normally one. In this example there are two sites 0 and 1 and the columns `val0` and `val` give the value of the conditional spectrum on such sites respectively, while the columns `std0` and `std1` give the corresponding standard deviations.

Conditional spectra for individual realizations are also computed and stored for debugging purposes, but they are not exportable.

Chapter 18. The post-processing framework and Vector-valued PSHA calculations

NINETEEN

EVENT BASED AND SCENARIOS

Scenario risk calculations usually do not pose a performance problem, since they involve a single rupture and limited geographical region for analysis. Some event-based risk calculations, however, may involve millions of ruptures and exposures spanning entire countries or even continents. 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 otherwise intractable calculation tractable.

19.1 Understanding the hazard

Event-based calculations are typically dominated by the hazard component (unless there are lots of assets aggregated on a 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 to 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 `ses_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.

19.2 region_grid_spacing

In our experience, the most common error made by our users is to compute the hazard at the sites of the exposure. The issue is that it is possible to have exposures with millions of assets on millions of distinct hazard sites. Computing the GMFs for millions of sites is hard or even impossible (there is a limit of 4 billion rows on the size of the GMF table in the datastore). Even in the cases when computing the hazard is possible, then computing the risk starting from an extremely large amount of GMFs will likely be impossible, due to memory/runtime constraints.

The second most common error is using an extremely fine grid for the site model. Remember that if you have a resolution of 250 meters, a square of 250 km x 250 km will contain one million sites, which is definitely too much. The engine was designed when the site models had resolutions around 5-10 km, i.e. of the same order of the hazard grid, while nowadays the vs30 fields have a much larger resolution.

Both problems can be solved in a simple way by specifying the `region_grid_spacing` parameter. Make it large enough that the resulting number of sites becomes reasonable and you are done. You will lose some precision, but that is preferable to not being able to run the calculation. You will need to run a sensitivity analysis with different values of `region_grid_spacing` parameter to make sure that you get consistent results, but that's it.

Once a `region_grid_spacing` is specified, the engine computes the convex hull of the exposure sites and builds a grid of hazard sites, associating the site parameters from the closest site in the site model and discarding sites in the region where there are no assets (i.e. more distant than `region_grid_spacing * sqrt(2)`). The precise logic is encoded in the function `openquake.commonlib.readinput.get_sitecol_assetcol`, if you want to know the specific implementation details.

Our recommendation is to use the command `oq prepare_site_model` to apply such logic before starting a calculation and thus producing a custom site model file tailored to your exposure (see the section *prepare_site_model*).

19.3 Collapsing of branches

When one is not interested in the uncertainty around the loss estimates and cares more about 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.

19.4 Using `collect_rlzs=true` in the risk calculation

Since version 3.12 the engine recognizes a flag `collect_rlzs` in the risk configuration file. When the flag is set to true, then the hazard realizations are collected together *when computing the risk results* and considered as one.

Setting `collect_rlzs=true` is possible only when the weights of the realizations are all equal, otherwise, the engine raises an error. Collecting the realizations makes the calculation of the average losses and loss curves much faster and more memory efficient. It is the recommended way to proceed when you are interested only in mean results. When you have a large exposure and many realizations (say 5 million assets and 1000 realizations, as it is the case for Chile) setting `collect_rlzs=true` can make possible a calculation that otherwise would run out of memory.

Note 1: when using sampling, `collect_rlzs` is implicitly set to `True`, so if you want to export the individual results per realization you must set explicitly `collect_rlzs=false`.

Note 2: `collect_rlzs` is not the inverse of the `individual_rlzs` flag. The `collect_rlzs` flag indicates to the engine that it should pool together the hazard realizations into a single collective bucket that will then be used to approximate the branch-averaged risk metrics directly, without going through the process of first computing the individual branch results and then getting the weighted average results from the branch results. Whereas the `individual_rlzs` flag indicates to the engine that the user is interested in storing and exporting the hazard (or risk) results for every realization. Setting `individual_rlzs` to `false` means that the engine will store only the statistics (mean and quantile results) in the datastore.

Note 3: `collect_rlzs` is completely ignored in the hazard part of the calculation, i.e. it does not affect at all the computation of the GMFs, only the computation of the risk metrics.

19.5 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.

19.6 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.

19.7 ignore_covs vs ignore_master_seed

The vulnerability functions using continuous distributions (lognormal/beta) to characterize the uncertainty in the loss ratio, specify the mean loss ratios and the corresponding coefficients of variation for a set of intensity levels.

There is clearly a performance/memory penalty associated with the propagation of uncertainty in the vulnerability to losses. You can completely remove it by setting

```
ignore_covs = true
```

in the *job.ini* file. Then the engine would compute just the mean loss ratios by ignoring the uncertainty i.e. the coefficients of variation. Since engine 3.12 there is a better solution: setting

```
ignore_master_seed = true
```

in the *job.ini* file. Then the engine will compute the mean loss ratios but also store information about the uncertainty of the results in the asset loss table, in the column "variance", by using the formulae

$$variance = \Sigma_i \sigma_i^2 \ for \ asset\_correl = 0$$
$$variance = (\Sigma_i \sigma_i)^2 \ for \ asset\_correl = 1$$

in terms of the variance of each asset for the event and intensity level in consideration, extracted from the asset loss and the coefficients of variation. People interested in the details should look at the implementation in https://github.com/gem/oq-engine/blob/engine-3.17/openquake/risklib/scientific.py.

TWENTY

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 a 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 honours a parameter called `minimum_asset_loss` which determines 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 it 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 is extremely easy to run out of memory or 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 of 0 corresponding to the total portfolio losses.

20.1 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 as input an array of cumulative losses associated with the aggregation tag, a list of the 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:

```
>>> 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 losses, E > 1) generating E time periods `eff_time/E, eff_time/(E-1), ... eff_time/1`, and log-interpolating the loss at the return period. Of course this works only if the condition `eff_time/E < return_period < 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

```
>>> 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:

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

The rules above are the reason why 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).

20.1.1 Aggregate loss curves

In some cases the computation of the PML is particularly simple and you can do it by hand: this happens when the ratio `eff_time/return_period` is an integer. Consider for instance a case with `eff_time=10,000` years and `return_period=2,000` years; suppose there are the following 10 losses aggregating the commercial and residential buildings of an exposure:

```
>>> import numpy as np
>>> losses_COM = np.array([123, 0, 400, 0, 1500, 200, 350, 0, 700, 600])
>>> losses_RES = np.array([0, 800, 200, 0, 500, 1200, 250, 600, 300, 150])
```

The loss curve associate the highest loss to 10,000 years, the second highest to 10,000 / 2 years, the third highest to 10,000 / 3 years, the fourth highest to 10,000 / 4 years, the fifth highest to 10,000 / 5 years and so on until the lowest loss is associated to 10,000 / 10 years. Since the return period is 2,000 = 10,000 / 5 to compute the MPL it is enough to take the fifth loss ordered in descending order:

```
>>> MPL_COM = [1500, 700, 600, 400, 350, 200, 123, 0, 0, 0][4] = 350
>>> MPL_RES = [1200, 800, 600, 500, 300, 250, 200, 150, 0, 0][4] = 300
```

Given this algorithm, it is clear why the MPL cannot be additive, i.e. MPL(COM + RES) != MPL(COM) + MPL(RES): doing the sums before or after the ordering of the losses is different. In this example by taking the fifth loss of the sorted sums

```
>>> sorted(losses_COM + losses_RES, reverse=True)
[2000, 1400, 1000, 800, 750, 600, 600, 600, 123, 0]
```

one gets `MPL(COM + RES) = 750` which is different from `MPL(RES) + MPL(COM) = 350 + 300 = 650`.

The engine is able to compute aggregate loss curves correctly, i.e. by doing the sums before the ordering phase. In order to perform aggregations, you need to set the `aggregate_by` parameter in the `job.ini` by specifying tags over which you wish to perform the aggregation. Your exposure must contain the specified tags 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 NAME_1 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:

```
 Calculation 1 finished correctly in 17 seconds
id | name
 9 | Aggregate Event Losses
 1 | Aggregate Loss Curves
 2 | Aggregate Loss Curves Statistics
 3 | Aggregate Losses
 4 | Aggregate Losses Statistics
 5 | Average Asset Losses Statistics
11 | Earthquake Ruptures
 6 | Events
 7 | Full Report
 8 | Input Files
10 | Realizations
12 | Total Loss Curves
13 | Total Loss Curves Statistics
14 | Total Losses
15 | Total Losses Statistics
```

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

```
annual_frequency_of_exceedence,return_period,loss_type,loss_value,loss_
 ↪ratio
5.00000E-01,2,nonstructural,0.00000E+00,0.00000E+00
5.00000E-01,2,structural,0.00000E+00,0.00000E+00
2.00000E-01,5,nonstructural,0.00000E+00,0.00000E+00
2.00000E-01,5,structural,0.00000E+00,0.00000E+00
1.00000E-01,10,nonstructural,0.00000E+00,0.00000E+00
1.00000E-01,10,structural,0.00000E+00,0.00000E+00
5.00000E-02,20,nonstructural,0.00000E+00,0.00000E+00
5.00000E-02,20,structural,0.00000E+00,0.00000E+00
2.00000E-02,50,nonstructural,0.00000E+00,0.00000E+00
2.00000E-02,50,structural,0.00000E+00,0.00000E+00
1.00000E-02,100,nonstructural,0.00000E+00,0.00000E+00
1.00000E-02,100,structural,0.00000E+00,0.00000E+00
5.00000E-03,200,nonstructural,1.35279E+05,1.26664E-06
5.00000E-03,200,structural,2.36901E+05,9.02027E-03
2.00000E-03,500,nonstructural,1.74918E+06,1.63779E-05
2.00000E-03,500,structural,2.99670E+06,1.14103E-01
1.00000E-03,1000,nonstructural,6.92401E+06,6.48308E-05
1.00000E-03,1000,structural,1.15148E+07,4.38439E-01
```

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

20.2 Aggregating by multiple tags

The engine also supports aggregation by multiple tags. Multiple tags can be indicated as multi-tag and/or various single-tag aggregations:

```
aggregate_by = NAME_1, taxonomy
```

or

```
aggregate_by = NAME_1; taxonomy
```

Comma `,` separated values will generate keys for all the possible combinations of the indicated tag values, while semicolon `;` will generate keys for the single tags.

For instance the second event based risk demo (the file `job_eb.ini`) has a line

```
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 *oq show agg_keys*:

```
$ oq show agg_keys
| NAME_1_ | taxonomy_ | NAME_1      | taxonomy                  |
+---------+-----------+-------------+---------------------------+
| 1       | 1         | Mid-Western | Wood                      |
| 1       | 2         | Mid-Western | Adobe                     |
| 1       | 3         | Mid-Western | Stone-Masonry             |
| 1       | 4         | Mid-Western | Unreinforced-Brick-Masonry |
| 1       | 5         | Mid-Western | Concrete                  |
| 2       | 1         | Far-Western | Wood                      |
| 2       | 2         | Far-Western | Adobe                     |
| 2       | 3         | Far-Western | Stone-Masonry             |
| 2       | 4         | Far-Western | Unreinforced-Brick-Masonry |
| 2       | 5         | Far-Western | Concrete                  |
| 3       | 1         | West        | Wood                      |
| 3       | 2         | West        | Adobe                     |
| 3       | 3         | West        | Stone-Masonry             |
| 3       | 4         | West        | Unreinforced-Brick-Masonry |
| 3       | 5         | West        | Concrete                  |
| 4       | 1         | East        | Wood                      |
| 4       | 2         | East        | Adobe                     |
| 4       | 3         | East        | Stone-Masonry             |
| 4       | 4         | East        | Unreinforced-Brick-Masonry |
| 4       | 5         | East        | Concrete                  |
| 5       | 1         | Central     | Wood                      |
| 5       | 2         | Central     | Adobe                     |
| 5       | 3         | Central     | Stone-Masonry             |
```

```
| 5         | 4          | Central       | Unreinforced-Brick-Masonry |
| 5         | 5          | Central       | Concrete                   |
```

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=*Central* and taxonomy=*Concrete*); 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
| agg_id | rlz | loss_type     | value       |
+--------+-----+--------------+-------------+
| 0        | 0   | nonstructural | 2_327_008   |
| 0        | 0   | structural    | 937_852     |
+--------+-----+--------------+-------------+
| ...      + ... + ...          + ...         +
+--------+-----+--------------+-------------+
| 25       | 1   | nonstructural | 100_199_448 |
| 25       | 1   | structural    | 157_885_648 |
```

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*.

It is possible to see the `agg_id` field with the command `$ oq show agg_id`.

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

$$1703 * 26 * 2 = 88,556 \text{ rows}$$

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]
```

It is also possible to perform the aggregation by various single-tag aggregations, using the `;` separator instead of `,`. For example, a line like

```
aggregate_by = NAME_1; taxonomy
```

would produce first the aggregation by geographic region (`NAME_1`), then by `taxonomy`. In this

case, instead of producing 5 x 5 combinations, only 5 + 5 outputs would be obtained.

TWENTYONE

RUPTURE SAMPLING: HOW DOES IT WORK?

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

```
>>> 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>
...     <truncGutenbergRichterMFD aValue="3" bValue="1" minMag="5" maxMag=
↪"7"/>
...     <nodalPlaneDist>
...         <nodalPlane dip="30" probability="1" strike="45" rake="90" />
...     </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:

```
>>> [(mag1, rate1), (mag2, rate2)] = src.get_annual_occurrence_rates()
>>> mag1
5.5
>>> mag2
6.5
```

The occurrence rates are respectively 0.009 and 0.0009. So, if we set the number of stochastic event sets to 1,000,000

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

we would expect the first rupture (the one with magnitude 5.5) to occur around 9,000 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

```
>>> np.random.seed(42)
```

then we will have (for `investigation_time = 1`)

```
>>> np.random.poisson(rate1 * num_ses * 1)
8966
>>> np.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

```
number_of_events = sum(n_occ for each rupture)
```

i.e. 8,966 + 921 = 9,887, 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 can change with updates to the the engine, if you run an event based calculation with the same parameters with different versions of the engine, you may not get exactly the same number of events, but something close given a reasonably long effective investigation time. 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 then the relation

```
n_occ ~ occurrence_rate * eff_investigation_time
```

will hold. If the effective investigation time is not large enough, or the occurrence rate is extremely small, then you should expect to see larger 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 ground motion logic tree and the correct formula to use is

```
eff_investigation_time = investigation_time * num_ses * num_rlzs
```

where `num_rlzs` is the number of realizations in the ground motion 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)`.

21.1 The difference between full enumeration and sampling

Users are often confused about the difference between full enumeration and sampling. For this reason the engine distribution comes with a pedagogical example that considers an extremely simplified situation comprising a single site, a single rupture, and only two GMPEs. You can find the example in the engine repository under the directory *openquake/qa_tests_data/event_based/case_3*. If you look at the ground motion logic tree file, the two GMPEs are AkkarBommer2010 (with weight 0.9) and SadighEtAl1997 (with weight 0.1).

The parameters in the job.ini are:

```
investigation_time = 1
ses_per_logic_tree_path = 5_000
number_of_logic_tree_paths = 0
```

Since there are 2 realizations, the effective investigation time is 10,000 years. If you run the calculation, you will generate (at least with version 3.13 of the engine, though the details may change with the version) 10,121 events, since the occurrence rate of the rupture was chosen to be 1. Roughly half of the events will be associated with the first GMPE (AkkarBommer2010) and half with the second GMPE (SadighEtAl1997). Actually, if you look at the test, the precise numbers will be 5,191 and 4,930 events, i.e. 51% and 49% rather than 50% and 50%, but this is expected and by increasing the investigation time you can get closer to the ideal equipartition. Therefore, even if the AkkarBommer2010 GMPE is assigned a relative weight that is 9 times greater than SadighEtAl1997, *this is not reflected in the simulated event set*. It means that when performing a computation (for instance to compute the mean ground motion field, or the average loss) one has to keep the two realizations distinct, and only at the end to perform the weighted average.

The situation is the opposite when sampling is used. In order to get the same effective investigation time of 10,000 years you should change the parameters in the job.ini to:

```
investigation_time = 1
ses_per_logic_tree_path = 1
number_of_logic_tree_paths = 10_000
```

Now there are 10,000 realizations, not 2, and they *all have the same weight .0001*. The number of events per realization is still roughly constant (around 1) and there are still 10,121 events, however now *the original weights are reflected in the event set*. In particular there are 9,130 events associated

---

to the AkkarBommer2010 GMPE and 991 events associated to the SadighEtAl1997 GMPE. There is no need to keep the realizations separated: since they have all the same weigths, you can trivially compute average quantities. AkkarBommer2010 will count more than SadighEtAl1997 simply because there are 9 times more events for it (actually 9130/991 = 9.2, but the rate will tend to 9 when the effective time will tend to infinity).

NB: just to be clear, normally realizations are not in one-to-one correspondence with GMPEs. In this example, it is true because there is a single tectonic region type. However, usually there are multiple tectonic region types, and a realization is associated to a tuple of GMPEs.

TWENTYTWO

RUPTURE SAMPLING: HOW TO GET IT WRONG

Rupture samplings is *much more complex than one could expect* and in many respects *surprising*. In the many years of existence of the engine, multiple approached were tried and you can expect the details of the rupture sampling mechanism to be different nearly at every version of the engine.

Here we will discuss some tricky points that may help you understand why different versions of the engine may give different results and also why the comparison between the engine and other software performing rupture sampling is nontrivial.

We will start with the first subtlety, the *interaction between sampling and filtering*. The short version is that you should *first sample and then filter*.

Here is the long version. Consider the following code emulating rupture sampling for poissonian ruptures:

```python
import numpy

class FakeRupture:
    def __init__(self, mag, rate):
        self.mag = mag
        self.rate = rate

def calc_n_occ(ruptures, eff_time, seed):
    rates = numpy.array([rup.rate for rup in ruptures])
    return numpy.random.default_rng(seed).poisson(rates * eff_time)

mag_rates = [(5.0, 1e-5), (5.1, 2e-5), (5.2, 1e-5), (5.3, 2e-5),
             (5.4, 1e-5), (5.5, 2e-5), (5.6, 1e-5), (5.7, 2e-5)]
fake_ruptures = [FakeRupture(mag, rate) for mag, rate in mag_rates]
eff_time = 50 * 10_000
seed = 42
```

Running this code will give you the following numbers of occurrence for the 8 ruptures considered:

```
>> calc_n_occ(fake_ruptures, eff_time, seed)
[ 8  9  6 13  7  6  6 10]
```

Here we did not consider the fact that engine has a `minimum_magnitude` feature and it is able to discard ruptures below the minimum magnitude. But how should it work? The natural approach to follow, for performance-oriented applications, would be to first discard the low magnitudes and then perform the sampling. However, that would have effects that would be surprising for most users. Consider the following two alternative:

```
def calc_n_occ_after_filtering(ruptures, eff_time, seed, min_mag):
    mags = numpy.array([rup.mag for rup in ruptures])
    rates = numpy.array([rup.rate for rup in ruptures])
    return numpy.random.default_rng(seed).poisson(
        rates[mags >= min_mag] * eff_time)

def calc_n_occ_before_filtering(ruptures, eff_time, seed, min_mag):
    mags = numpy.array([rup.mag for rup in ruptures])
    rates = numpy.array([rup.rate for rup in ruptures])
    n_occ = numpy.random.default_rng(seed).poisson(rates * eff_time)
    return n_occ[mags >= min_mag]
```

Most users would expect that removing a little number of ruptures has a little effect; for instance, if we set `min_mag = 5.1` such that only the first rupture is removed from the total 8 ruptures, we would expect a minor change. However, if we follow the filter-early approach the user would get completely different occupation numbers:

```
>> calc_n_occ_after_filtering(fake_ruptures, eff_time, seed, min_mag)
[13  6  9  6 13  7  6]
```

It is only by using the filter-late approach that the occupation numbers are consistent with the no-filtering case:

```
>> calc_n_occ_before_filtering(fake_ruptures, eff_time, seed, min_mag)
[ 9  6 13  7  6  6 10]
```

The problem with the filtering is absolutely general and not restricted only to the magnitude filtering: it is exactly the same also for distance filtering. Suppose you have a `maximum_distance` of 300 km and than you decide that you want to increase it to 301 km. One would expect this change to have a minor impact; instead, you may end up sampling a very different set of ruptures.

It is true that average quantities like the hazard curves obtained from the ground motion fields will converge for long enough effective time, however in practice you are always in situations were

1. you cannot perform the calculation for a long enough effective time since it would be computationally prohibitive

2. you are interested on quantities which are strongly sensitive to aany change, like the Maximum Probable Loss at some return period

In such situations changing the site collection (or changing the maximum distance which is akin to changing the site collection) can change the sampling of the ruptures significantly, at least for engine versions lower than 3.17.

Users wanting to compare the GMFs or the risk on different site collections should be aware of this effect; the solution is to first sample the ruptures without setting any site collection (i.e. disabling the filtering) and then perform the calculation with the site collection starting from the sampled ruptures.

The problem is discussed also in the section about *Risk profiles* which discusses why you should not split the hazard calculation in different countries when running a continental scale calculation.

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.

23.1 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):

```
$ oq engine --run event_based/case_7/job.ini
<snip>
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%
<snip>
$ oq plot /tmp/cl/hazard.pik /tmp/hazard.pik --sites=0,1,2
```
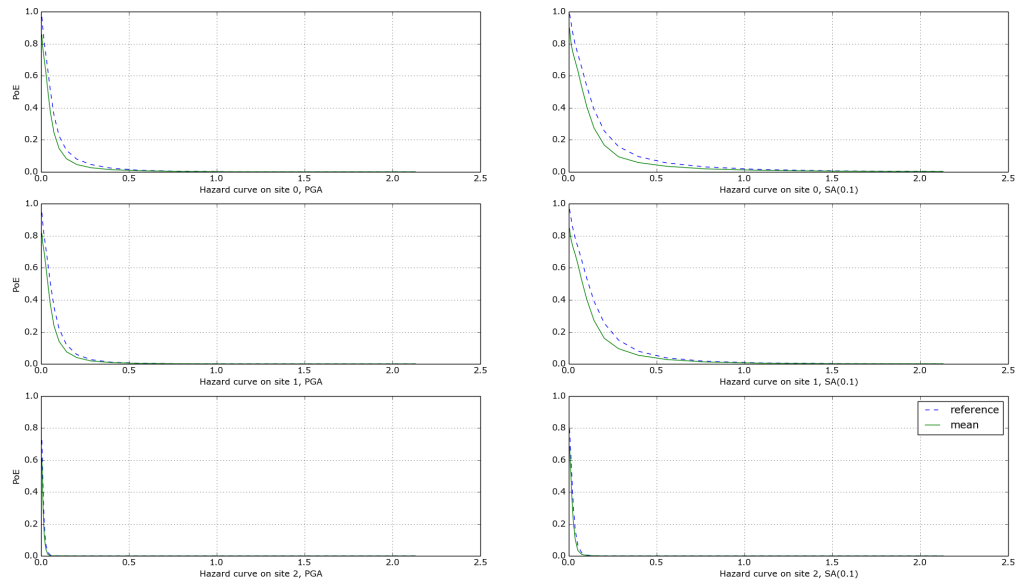
The relative difference between the classical and event based curves is computed by computing the relative difference between each point of the curves for each curve, and by taking the maximum, at least for probabilities of exceedence larger than 1% (for low values of the probability the convergency may be bad). For the details I suggest you to look at the code.

TWENTYFOUR

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.

24.1 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 but 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:

$$\mu = \Sigma_i w_i \mu_i$$
$$\sigma^2 = \Sigma_i w_i (\sigma_i)^2$$

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.

24.2 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 downloding 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 threshould 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 logaritms with μ sigma`=.5; then the ground motion values that you could obtain would be as follows:

```
>>> import numpy
>>> np.random.seed(42) # fix the seed
>>> gmvs = np.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:

```
>>> numpy.round([gmvs.min(), np.median(gmvs), gmvs.max()], 6)
array([0.026766, 0.137058, 0.929011])
```

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

```
>>> round(np.log(gmvs).mean(), 6)
-1.990334
>>> round(np.log(gmvs).std(), 6)
0.489363
```

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:

```
>>> round(np.exp(np.log(gmvs).mean()), 6)
0.13665
```

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

```
>> gmvs[gmvs < .05] = .05
>> round(np.log(gmvs).mean(), 6)
-1.987608
>> round(np.log(gmvs).std(), 6)
0.4828063
>> round(np.exp(np.log(gmvs).mean()), 6)
0.137023
```

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).

24.2. Mean ground motion field by event **131**

24.3 Single-rupture estimated median ground motion field

The mean ground motion field by event discussed above is an *a posteriori* output: *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 *a priori*, using hazardlib and without performing a full calculation. However, there are some limitations to this approach:

1. it only works when there is a single rupture

2. you have to manage the `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 `job.ini` the following values:

```
truncation_level = 0
ground_motion_fields = 1
```

Setting `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 `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 *median* ground motion field.

24.4 Case study: 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:

```
N = 707,920 hazard sites
E = 10^5 estimated events of magnitude greater then 5.5 in the␣
→investigation
    time of 100,000 years
```

```
B = 1 number of branches in the UCERF logic tree
G = 5 number of GSIMS in the GMPE logic tree
I = 6 number of intensity measure types
S1 = 13 number of bytes used by the engine to store a single GMV
```

The maximum size of generated GMFs is

```
N * E * B * G * I * S1 = 25 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.

EXTENDED CONSEQUENCES

Scenario damage calculations produce damage distributions, i.e. arrays containing the number of buildings in each damage state defined in the fragility functions. There is a damage distribution per each asset, event and loss type, so you can easily produce *billions* of damage distributions. This is why the engine provide facilities to compute results based on aggregating the damage distributions, possibly multiplied by suitable coefficients, i.e. *consequences*.

For instance, from the probability of being in the collapsed damage state, one may estimate the number of fatalities, given the right multiplicative coefficient. Another commonly computed consequence is the economic loss; in order to estimated it, one need a different multiplicative coefficient for each damage state and for each taxonomy. The table of coefficients, a.k.a. the *consequence model*, can be represented as a CSV file like the following:

taxonomy	conse-quence	loss_type	slight	moder-ate	exten-sive	com-plete
CR_LFINF-DUH_H2	losses	struc-tural	0.05	0.25	0.6	1
CR_LFINF-DUH_H4	losses	struc-tural	0.05	0.25	0.6	1
MCF_LWAL-DNO_H3	losses	struc-tural	0.05	0.25	0.6	1
MR_LWAL-DNO_H1	losses	struc-tural	0.05	0.25	0.6	1
MR_LWAL-DNO_H2	losses	struc-tural	0.05	0.25	0.6	1
MUR_LWAL-DNO_H1	losses	struc-tural	0.05	0.25	0.6	1
W-WS_LPB-DNO_H1	losses	struc-tural	0.05	0.25	0.6	1
W-WWD_LWAL-DNO_H1	losses	struc-tural	0.05	0.25	0.6	1
MR_LWAL-DNO_H3	losses	struc-tural	0.05	0.25	0.6	1

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 consequence 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 function to use to compute the consequence. It is rather easy to write an additional function in case one needed to support a new kind of consequence. You can show the list of consequences by the version of the engine that you have installed with the command:

```
$ oq info consequences  # in version 3.12
The following 5 consequences are implemented:
losses
collapsed
injured
fatalities
homeless
```

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. The loss type refers to the fragility model, i.e. `structural` will mean that the coefficients apply to damage distributions obtained from the fragility functions defined in the file `structural_fragility_model.xml`.

25.1 discrete_damage_distribution

Damage distributions are called discrete when the number of buildings in each damage is an integer, and continuous when the number of buildings in each damage state is a floating point number. Continuous distributions are a lot more efficient to compute and therefore that is the default behavior of the engine, at least starting from version 3.13. You can ask the engine to use discrete damage distribution by setting the flag in the job.ini file

`discrete_damage_distribution = true`

However, it should be noticed that setting `discrete_damage_distribution = true` will raise an error if the exposure contains a floating point number of buildings for some asset. Having a floating point number of buildings in the exposure is quite common since the "number" field is often estimated as an average.

Even if the exposure contains only integers and you have set `discrete_damage_distribution = true` in the job.ini, the aggregate damage distributions will normally contains floating point numbers, since they are obtained by summing integer distributions for all seismic events of a given hazard realization and dividing by the number of events of that realization.

By summing the number of buildings in each damage state one will get the total number of buildings for the given aggregation level; if the exposure contains integer numbers than the sum of the numbers will be an integer, apart from minor differences due to numeric errors, since the engine stores even discrete distributions as floating point numbers.

25.2 The EventBasedDamage demo

Given a source model, a logic tree, an exposure, a set of fragility functions and a set of consequence functions, the `event_based_damage` calculator is able to compute results such as average consequences and average consequence curves. The `scenario_damage` calculator does the same, except it does not start from a source model and a logic tree, but rather from a set of predetermined ruptures or ground motion fields, and the averages are performed on the input parameter `number_of_ground_motion_fields` and not on the effective investigation time.

In the engine distribution, in the folders `demos/risk/EventBasedDamage` and `demos/risk/ScenarioDamage` there are examples of how to use the calculators.

Let's start with the EventBasedDamage demo. The source model, the exposure and the fragility functions are much simplified and you should not consider them realistic for the Nepal, but they permit very fast hazard and risk calculations. The effective investigation time is

```
eff_time = 1 (year) x 1000 (ses) x 50 (rlzs) = 50,000 years
```

and the calculation is using sampling of the logic tree. Since all the realizations have the same weight, on the risk side we can effectively consider all of them together. This is why there will be a single output (for the effective risk realization) and not 50 outputs (one for each hazard realization) as it would happen for an `event_based_risk` calculation.

Normally the engine does not store the damage distributions for each asset (unless you specify `aggregate_by=id` in the `job.ini` file).

By default it stores the aggregate damage distributions by summing on all the assets in the exposure. If you are interested only in partial sums, i.e. in aggregating only the distributions associated to a certain tag combination, you can produce the partial sums by specifying the tags. For instance `aggregate_by = taxonomy` will aggregate by taxonomy, `aggregate_by = taxonomy, region` will aggregate by taxonomy and region, etc. The aggregated damage distributions (and aggregated consequences, if any) will be stored in a table called `risk_by_event` which can be accessed with pandas. The corresponding DataFrame will have fields `event_id`, `agg_id` (integer referring to which kind of aggregation you are considering), `loss_id` (integer referring to the loss type in consideration), a column named `dmg_X` for each damage state and a column for each consequence. In the EventBasedDamage demo the exposure has a field called `NAME_1` and representing a geographic region in Nepal (i.e. "East" or "Mid-Western") and there is an `aggregate_by = NAME_1, taxonomy` in the `job.ini`.

Since the demo has 4 taxonomies ("Wood", "Adobe", "Stone-Masonry", "Unreinforced-Brick-Masonry") there 4 x 2 = 8 possible aggregations; actually, there is also a 9th possibility corre-

sponding to aggregating on all assets by disregarding the tags. You can see the possible values of the the `agg_id` field with the following command:

```
$ oq show agg_id
                          taxonomy          NAME_1
agg_id
0                             Wood            East
1                             Wood     Mid-Western
2                            Adobe            East
3                            Adobe     Mid-Western
4                    Stone-Masonry            East
5                    Stone-Masonry     Mid-Western
6       Unreinforced-Brick-Masonry            East
7       Unreinforced-Brick-Masonry     Mid-Western
8                          *total*         *total*
```

Armed with that knowledge it is pretty easy to understand the `risk_by_event` table:

```
>> from openquake.commonlib.datastore import read
>> dstore = read(-1)   # the latest calculation
>> df = dstore.read_df('risk_by_event', 'event_id')
          agg_id  loss_id  dmg_1  dmg_2  dmg_3  dmg_4           losses
event_id
472            0        0    0.0    1.0    0.0    0.0      5260.828125
472            8        0    0.0    1.0    0.0    0.0      5260.828125
477            0        0    2.0    0.0    1.0    0.0      6368.788574
477            8        0    2.0    0.0    1.0    0.0      6368.788574
478            0        0    3.0    1.0    1.0    0.0      5453.355469
...          ...      ...    ...    ...    ...    ...              ...
30687          8        0   56.0   53.0   26.0   16.0    634266.187500
30688          0        0    3.0    6.0    1.0    0.0     14515.125000
30688          8        0    3.0    6.0    1.0    0.0     14515.125000
30690          0        0    2.0    0.0    1.0    0.0      5709.204102
30690          8        0    2.0    0.0    1.0    0.0      5709.204102
[8066 rows x 7 columns]
```

The number of buildings in each damage state is integer (even if stored as a float) because the exposure contains only integers and the *job.ini* is setting explicitly `discrete_damage_distribution` = `true`.

It should be noted that while there is a CSV exporter for the `risk_by_event` table, it is designed to export only the total aggregation component (i.e. `agg_id=9` in this example) for reasons of backward compatibility with the past, the time when the only aggregation the engine could perform was the total aggregation. Since the `risk_by_event` table can be rather large, it is recommmended to interact with it with pandas and not to export in CSV.

There is instead a CSV exporter for the aggregated damage distributions (together with the aggregated consequences) that you may call with the command `oq export aggrisk`; you can also see the distributions directly:

```
$ oq show aggrisk
   agg_id  rlz_id  loss_id        dmg_0      dmg_1      dmg_2      dmg_3      ␣
→dmg_4        losses
0       0       0        0    18.841061  0.077873  0.052915  0.018116  0.
→010036     459.162567
1       3       0        0   172.107361  0.329445  0.591998  0.422925  0.
→548271   11213.121094
2       5       0        0     1.981786  0.003877  0.005539  0.004203  0.
→004594     104.431755
3       6       0        0   797.826111  1.593724  1.680134  0.926167  0.
→973836   23901.496094
4       7       0        0    48.648529  0.120687  0.122120  0.060278  0.
→048386    1420.059448
5       8       0        0  1039.404907  2.125607  2.452706  1.431690  1.
→585123   37098.269531
```

By summing on the damage states one gets the total number of buildings for each aggregation level:

```
agg_id dmg_0 + dmg_1 + dmg_2 + dmg_3 + dmg_4 aggkeys
0           19.000039 ~ 19                   Wood,East
3          173.999639 ~ 174                  Wood,Mid-Western
5            2.000004 ~ 2                     Stone-Masonry,Mid-Western
6          802.999853 ~ 803                  Unreinforced-Brick-Masonry,
→East
7           48.999971 ~ 49                   Unreinforced-Brick-Masonry,
→Mid-Western
8         1046.995130 ~ 1047                 Total
```

25.3 The ScenarioDamage demo

The demo in `demos/risk/ScenarioDamage` is similar to the EventBasedDemo (it still refers to Nepal) but it uses a much large exposure with 9063 assets and 5,365,761 building. Moreover the configuration file is split in two: first you should run `job_hazard.ini` and then run `job_risk.ini` with the `--hc` option.

The first calculation will produce 2 sets of 100 ground motion fields each (since `job_hazard.ini` contains `number_of_ground_motion_fields = 100` and the gsim logic tree file contains two GMPEs). The second calculation will use such GMFs to compute aggregated damage distributions. Contrarily to event based damage calculations, scenario damage calculations normally use

full enumeration, since there are very few realizations (only two in this example), thus the scenario damage calculator is able to distinguish the results by realization.

The main output of a `scenario_damage` calculation is still the `risk_by_event` table which has exactly the same form as for the EventBasedDamage demo. However there is a difference when considering the `aggrisk` output: since we are using full enumeration we will produce a damage distribution for each realization:

```
$ oq show aggrisk
   agg_id  rlz_id  loss_id        dmg_0  ...  dmg_4         losses
0       0       0        0  4173405.75  ...  452433.40625  7.779261e+09
1       0       1        0  3596234.00  ...  633638.37500  1.123458e+10
```

The sum over the damage states will still produce the total number of buildings, which will be independent from the realization:

```
rlz_id dmg_0 + dmg_1 + dmg_2 + dmg_3 + dmg_4
0        5365761.0
1        5365761.0
```

In this demo there is no `aggregate_by` specified, so the only aggregation which is performed is the total aggregation. You are invited to specify `aggregate_by` and study how `aggrisk` changes.

25.4 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:

taxonomy	conversion
Wood Type A	Wood
Wood Type B	Wood
Wood Type C	Wood

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:

taxonomy	conversion	weight
Wood	Wood Type A	0.2
Wood	Wood Type B	0.3
Wood	Wood Type C	0.5

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.

CORRELATION OF GROUND MOTION FIELDS

There are multiple different kind of correlation on the engine, so it is extremely easy to get confused. Here I will list all possibilities, in historical order.

1. Spatial correlation of ground motion fields has been a feature of the engine from day one. The available models are JB2009 and HM2018.

2. Cross correlation in ShakeMaps has been available for a few years. The model used there is hard-coded an the user cannot change it, only disable it. The models list below (3. and 4.) *have no effect on ShakeMaps*.

3. Since version 3.13 the engine provides the BakerJayaram2008 cross correlation model, however at the moment it is used only in the conditional spectrum calculator.

4. Since version 3.13 the engine provides the GodaAtkinson2009 cross correlation model and the FullCrossCorrelation model which can be used in scenario and event based calculations.

Earthquake theory tells us that ground motion fields depend on two different lognormal distributions with parameters (μ, τ) and (μ, ϕ) respectively, which are determined by the GMPE (Ground Motion Prediction Equal). Given a rupture, a set of M intensity measure types and a collection of N sites, the parameters μ, τ and ϕ are arrays of shape (M, N). μ is the mean of the logarithms and τ the between-event standard deviation, associated to the cross correlation, while ϕ is the within-event standard deviation, associated to the spatial correlation. math:*tau* and ϕ are normally N-independent, i.e. each array of shape (M, N) actually contains N copies of the same M values read from the coefficient table of the GMPE.

In the OpenQuake engine each rupture has associated a random seed generated from the parameter `ses_seed` given in the job.ini file, therefore given a fixed number E of events it is possible to generate a deterministic distribution of ground motion fields, i.e. an array of shape (M, N, E). Technically such feature is implemented in the class `openquake.hazardlib.calc.gmf.GmfComputer`. The algorithm used there is to generate two arrays of normally distributed numbers called ϵ_τ (of shape (M, E)) and ϵ_ϕ (of shape (M, N, E)), one using the between-event standard deviation τ and the other using the within-event standard deviation ϕ, while keeping the same mean μ. Then the ground motion fields are generated as an array of shape (M, N, E) with the formula

$$gmf = exp(\mu + crosscorrel(\epsilon_\tau) + spatialcorrel(\epsilon\phi))$$

The details depend on the form of the cross correlation model and of the spatial correlation model and you have to study the source code if you really want to understand how it works, in particular how the correlation matrices are extracted from the correlation models. By default, if no cross correlation nor spatial correlation are specified, then there are no correlation matrices and $crosscorrel(\epsilon_\tau)$ and $spatialcorrel(\epsilon\phi)$ are computed by using `scipy.stats.truncnorm`. Otherwise `scipy.stats.multivariate_normal` with a correlation matrix of shape (M, M) is used for cross correlation and `scipy.stats.multivariate_normal` distribution with a matrix of shape (N, N) is used for spatial correlation. Notice that the truncation feature is lost if you use correlation, since scipy does not offer at truncated multivariate_normal distribution. Not truncating the normal distribution can easily generated non-physical fields, but even if the truncation is on it is very possible to generate exceedingly large ground motion fields, so the user has to be *very* careful.

Correlation is important because its presence normally causes the risk to increase, i.e. ignoring the correlation will under-estimate the risk. The best way to play with the correlation is to consider a scenario_risk calculation with a single rupture and to change the cross and spatial correlation models. Possibilities are to specify in the job.ini all possible combinations of

cross_correlation = FullCrossCorrelation cross_correlation = GodaAtkinson2009 ground_motion_correlation_model = JB2009 ground_motion_correlation_model = HM2018

including removing one or the other or all correlations.

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.

27.1 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/usp000fjta#shakemap). The risk can be computed by running the risk job file against the prepared calculation:

```
$ 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*:

```
# (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:

```
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.

27.2 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 everything
- spatial_correlation = yes, cross_correlation = no
```

(continues on next page)

```
- 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.

27.3 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 all cases.

TWENTYEIGHT

REINSURANCE CALCULATIONS

Starting from engine 3.16 reinsurance loss estimates for traditional property contracts are available for event-based and scenario risk calculations.

The current implementation considers multiple layers of both proportional and non-proportional treaties.

Proportional treaties (Pro-Rata)

- Quota Share

- Surplus

- Facultative

NOTE: proportional treaties may have a parameter "max_cession_event" limiting the total losses per event that can be ceded to the reinsurer. The excess of loss generated by events that exceed the maximum cession per event (overspill losses) is going back to the insurer.

Non-proportional treaties

- Working excess of loss per risk, WXL/R (`wxlr`). The unit of loss under this treaty is the "risk". The engine aggregates the losses per "risk" at the policy level, which can include single or multiple assets.

- Catastrophic excess of loss per event, CatXL (`catxl`). The unit of loss under this treaty is the "event".

- When combined with *proportional* treaties, the *non-proportional* layers are applied over the net loss retention coming from the proportional layers; first the `wxlr` are estimated, and then the successive layers of CatXL are applied over the net loss retention

NOTE: The CatXL is applied over the net loss retention per event coming from the proportional layers and therefore it includes the overspill losses.

Reinsurance calculations provide, in addition to the ground up losses, the losses allocated to different treaties during a single event or during multiple events over a given time window. Outputs include average losses and aggregated loss curves at policy and portfolio level for the retention and cession under the different treaties.

28.1 Input files

To run reinsurance calculations, in addition to the required files for performing event-based or scenario risk calculations, it is required to adjust the exposure information, and to include two additional files:

1. Reinsurance information: an `.xml` file defining the characteristics of the reinsurance treaties (e.g., "reinsurance.xml").

2. Policy information: a `.csv` file with details of each policy indicated in the exposure model and the associated reinsurance treaties (e.g., "policy.csv").

The insurance information includes the allocation of assets into a given policy, the liability and the deductible. The deductible can be defined at asset level (therefore indicated in the exposure model `csv` file), or in at the policy level (therefore indicated in the policy `csv` file). The current implementation only supports liability at policy level.

28.1.1 Exposure file

The exposure input file (csv and xml with metadata) needs to be adjusted to include a `policy` tag that indicates the type of policy (and therefore the reinsurance contracts) associated with each asset.

Policies can be defined for single or multiple assets. When multiple assets are allocated to the same policy, losses are aggregated at the policy level before applying the insurance and reinsurance deductions.

Below we present an example of an exposure model considering the policy information and its associated metadata:

`exposure_model.csv`

id	lon	lat	tax-onomy	num-ber	struc-tural	con-tents	non-struc-tural	busi-ness_interruption		night	tag_1	pol-icy
a1	-122	38.113	tax1	1	10000	5000	15000	2000		6	zone_1	p1_a1
a2	-122.114	38.113	tax1	1	10000	5000	15000	2000		6	zone_1	p1_a2
a3	-122.57	38.113	tax1	1	10000	5000	15000	2000		6	zone_1	p1_a3
a4	-122	38	tax1	1	10000	5000	15000	2000		6	zone_2	p2
a5	-122	37.91	tax1	1	10000	5000	15000	2000		6	zone_2	p2
a6	-122	38.225	tax1	1	10000	5000	15000	2000		6	zone_2	p2
a7	-121.886	38.113	tax1	1	10000	5000	15000	2000		6	zone_2	p2

exposure.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nrml xmlns="http://openquake.org/xmlns/nrml/0.4">
  <exposureModel id="ex1" category="buildings" taxonomySource="GEM␣
↪taxonomy">
    <description>exposure model</description>
    <conversions>
      <costTypes>
        <costType name="structural" type="aggregated" unit="USD"/>
        <costType name="nonstructural" type="aggregated" unit="USD"/>
        <costType name="contents" type="aggregated" unit="USD"/>
      </costTypes>
    </conversions>
    <tagNames>tag_1 policy</tagNames>
    <occupancyPeriods>night </occupancyPeriods>
    <assets>
      exposure_model.csv
    </assets>
  </exposureModel>
</nrml>
```

This example presents 7 assets (a1 to a7) with 4 associated policies. Notice that the column `policy` is mandatory, as well as the line `<tagNames>policy</tagNames>` in the xml. Additional tags can be included as needed.

Starting from OpenQuake 3.17, the exposure module can also include deductibles at asset level (called `ideductible`). In this case, the deductions are applied at asset level, and later aggregated at the policy level before applying the liability and reinsurance allocations.

NOTE: It is not possible to have a policy with ``ideductible`` at asset and ``deductible`` at policy level. The engine only accepts one value.

28.1.2 Insurance and reinsurance information (`reinsurance.xml`)

The insurance and reinsurance information is defined by a `reinsurance.xml` that includes the metadata and treaty characteristics for each treaty specified in the policy information.

The following example facilitates the understanding of the input file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nrml xmlns="http://openquake.org/xmlns/nrml/0.5"
      xmlns:gml="http://www.opengis.net/gml">
  <reinsuranceModel>
    <description>reinsurance model</description>

    <fieldMap>
      <field oq="liability" input="Limit" />
      <field oq="deductible" input="Deductible" />

      <field input="treaty_1" type="prop" max_cession_event="400" />
      <field input="treaty_2" type="prop" max_cession_event="400" />
      <field input="xlr1" type="wxlr" deductible="200" limit="1000" />
    </fieldMap>
  <policies>policy.csv</policies>
  </reinsuranceModel>
</nrml>
```

reinsurance.xml parameters:

The reinsurance information must include, at least, a <`description`> and a list of files that contain the <`policies`>. The <`fieldMap`> block is used to define the reinsurance treaties and their parameters.

The `oq` and `input` parameters are used to specify the *key* used in the engine (`oq`) and its equivalent column header in the policy file (`input`). All reinsurance calculations must include, at least, the insurance characteristics of each policy: liability and deductible (that can be at asset or policy level, depending on the portfolio characteristics). Then, the definition of reinsurance treaties depends on the treaty type: proportional or non proportional.

Proportional treaties are identified by the parameter `type="prop"`. The fraction of losses ceded to each treaty is specified for each policy covered by the treaty, and the retention is calculated as 1

minus all the fractions specified in the multiple layers of proportional treaties. For each proportional treaty it is possible to define the `max_cession_event`.

Non-proportional treaties are identified by the parameter `type="wxlr"` or `type="catxl"`. For each treaty it is required to indicate the `deductible` and `limit`.

Note: treaties must be written in a given order, keeping proportional ones first, then non-proportional ones of type "wxlr" and finally those of type "catxl".

- **insurance deductible**: the amount (economic value) that the insurer will "deduct" from the ground up losses before paying up to its policy limits. The units of the deductible must be compatible with the units indicated in the exposure model (e.g. USD dollars or Euros). The deductible can be specified at policy (`deductible`) or asset level (`ideductible`) depending on the insurance contract.

  The claim is calculated as `claim = ground_up_loss - deductible` for policies with deductibles defined at the policy level, or `claim = ground_up_loss - ideductible` for policies with deductibles defined at the asset level.

- **insurance liability**: the maximum economic amount that can be covered by the insurance, according to the policy characteristics. The liability is also known as limit or maximum coverage.

- **type**: parameter that specifies the type of treaty. There are three supported types: `prop` (for proportional treaties), `wxlr`, or `catxl`.

- **max_cession_event**: the maximum cession per event is an optional parameter for *proportional* reinsurance treaties. It refers to the maximum cession accepted by the reinsurance company for the aggregated losses produced by a single event. If the aggregated losses exceed this threshold, then the cession in excess is reported as an `overspill`.

- **reinsurance deductible**: only applicable to *non-proportional* treaties, the maximum retention (also known as "first loss") is the limit above which the reinsurer becomes liable for losses up to the upper limit of cover.

- **reinsurance limit**: in *non-proportional* treaties it refers to the upper limit of cover or ceiling. The *reinsurance_cover* is the amount between the `deductible` (deductible) and the upper limit of cover.

Note: the current engine implementation does not support an "annual aggregate limit" for non-proportional reinsurance treaties.

28.1.3 Policy information (`policy.csv`)

The policy input file indicates, for each policy, the insurance values (deductible and liability), as well as the reinsurance contracts associated with each policy present in the exposure model.

For **proportional** treaties, the values indicated in each columns refer to the fraction of cession under the reinsurance treaty. On the other hand, for **non-proportional** treaties, the values are indicates as 1 for policies covered within the treaty and zero when they are not part of the treaty.

The table below presents an example of the four policies indicated in the example of the exposure model and the reinsurance presented above:

`policy.csv`

policy	Limit	Deductible	treaty_1	treaty_2	xlr1
p1_a1	2000	400	0.1	0.2	1
p1_a2	1000	200	0.3	0.1	1
p1_a3	1000	100	0	0.7	1
p2	2000	500	0	.4	1

The `policy` column must contain the same identifiers as the ones specified by the `policy` field in the exposure model.

In this example the `Limit` corresponds to the `liability` of each policy, while the `Deductible` refers to the `deductible` in the engine. Both columns indicate the absolute values using the same units as the exposed values in the exposure model. There are two proportional reinsurance treaties (namely `QuotaShare` and `Surplus`), and the values indicated in each column represent the fraction of cession under each treaty. For example, for "pol_1" the "QuotaShare" ceeds 0.1 of the losses and there is no cession under the "Surplus" treaty; therefore the retention corresponding to the proportional treaties for "pol_1" will be (1 - 0.1 - 0. = 0.9). In the case of non-proportional treaties, "pol_1" is allocated to the `WXLR` (an excess of loss per risk) treaty, and to the `CatXL1` (a catastrophic excess of loss per event) treaty. This policy is not covered by the `CatXL2` treaty.

Note: treaties must be written in a given order, keeping proportional ones first, then non-proportional ones of type "wxlr" and finally those of type "catxl".

28.1.4 Configuration file `job.ini`

Reinsurance losses can be calculated for event-based and scenario risk calculations. To do so, the configuration file, `job.in`, needs to specify the parameters presented below, in addition to the parameters generally indicated for these type of calculations:

```
[risk_calculation]
aggregate_by = policy
```

```
reinsurance_file = {'structural+contents': 'reinsurance.xml'}
total_losses = structural+contents
```

Additional comments:

- `aggregate_by`: it is possible to define multiple aggregation keys (see *Aggregating by multiple tags*). However, for reinsurance calculations the `policy` key must be present, otherwise an error message will be raised. In the following example, multiple aggregation keys are used:

  ```
  aggregate_by = policy; tag1
  ```

  In this case, aggregated loss curves will be produced also for `tag1` and `policy`, while reinsurance outputs will only be produced for the policy.

- `reinsurance_file`: This dictionary associates the reinsurance information to a given the loss_type (the engine supports structural, nonstructural, contents or its sum). The insurance and reinsurance calculations are applied over the indicated loss_types, i.e. to the sum of the ground up losses associated with the specified loss_types.

  NOTE: The current implementation works only with a single reinsurance file.

- `total_losses`: (or total exposed value) needs to be specified when the reinsurance needs to be applied over the sum of two or more loss types (e.g. `structural+contents`). The definition of total losses is also reflected in the risk outputs of the calculation. NB: if there is a single loss type (e.g. `structural`) there is no need to specify this parameter, just write `reinsurance_file = {'structural': 'reinsurance.xml'}`

28.2 Output files

The reinsurance calculations generates estimates of retention and cession under the different reinsurance treaties. The following output files are produced:

1. `Reinsurance by event`: aggregated estimated per event for the claim, retention, cession and overspills under each reinsurance treaty.

event_id	retention	claim	treaty_1	treaty_2	xlr1	overspill_treaty_2	year
0	738.429	1833.73	142.206	400.000	553.096	180.819	1
1	319.755	701.219	51.7092	179.292	150.463	0.00000	1
2	1226.97	3210.91	282.622	400.000	1301.32	474.357	1
3	1318.88	3600.81	294.502	400.000	1587.42	629.187	1

2. `Reinsurance curves`: reinsurance loss exceedance curves describe the probabilities of exceeding a set of loss ratios or loss values, within a given time span (or investigation interval).

The curves are generated for the claim, retention, cession and overspills under each reinsurance treaty.

rlz_id	return_period	retention	claim	treaty_1	treaty_2	xlr1	overspill_treaty_2
0	50.0000	319.755	701.219	51.7092	179.292	150.463	0.00000
0	100.000	1226.97	3210.91	282.622	400.000	1301.32	474.357
0	200.000	1318.88	3600.81	294.502	400.000	1587.42	629.187

3. `Average reinsurance losses`: the average reinsurance losses indicates the expected value within the time period specified by risk_investigation_time for the claim, retention, and cessions under each reinsurance treaty for all policies in the Exposure Model.

rlz_id	retention	claim	treaty_1	treaty_2	xlr1	overspill_treaty_2
0	1.80202E+01	4.67333E+01	3.85520E+00	6.89646E+00	1.79615E+01	6.42181E+00

4. `Aggregated reinsurance by policy`: the average reinsurance losses for each policy, by ignoring the overspill logic.

rlz_id	policy_id	retention	claim	treaty_1	treaty_2	xlr1
0	p1_a1	4.61304	19.0934	1.90934	3.81867	8.75232
0	p1_a2	3.01643	6.48621	1.94586	0.648621	0.875298
0	p1_a3	38.9468	1.29823	0.00000	0.908759	0.00000
0	p2	3.57945	19.8555	0.00000	7.94221	8.33388

The parameters indicated in the previous outputs include:

- `policy`: identifier of the unique policies indicated in the exposure model and policy files.

- `claim`: ground up losses minus the deductible and up to the policy liability.

- `retention`: net losses that the insurance company keeps for its own account.

- `cession_i`: net losses that are ceded by the insurance company to the reinsurer(s) under treaty i. The cession is indicated by the treaty name defined in the reinsurance information.

- `overspill_treaty_i`: net losses that exceed the maximum cession per event ("max_cession_event") for *proportional* and/or *catxl* treaties.

NOTE: The sum of the claim is not equal to the ground up losses, since usually the deductible is nonzero. Moreover there could be "non-insured" losses corresponding to policies with no insurance contracts or that exceed the policy liability.

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:

    1. if `region_grid_spacing` is specified, a grid is implicitly generated from the convex hull of the exposure and used

    2. 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.

It is possible to specify both `sites.csv` and `site_model.csv`: in that case the sites in `sites.csv` are used, with parameters inferred from the closest site model parameters.

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. 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.

RISK PROFILES

The OpenQuake engine can produce risk profiles, i.e. estimates of average losses and maximum probable losses for all countries in the world. Even if you are interested in a single country, you can still use this feature to compute risk profiles for each province in your country.

However, the calculation of the risk profiles is tricky and there are actually several different ways to do it.

1. The least-recommended way is to run independent calculations, one for each country. The issue with this approach is that even if the hazard model is the same for all the countries (say you are interested in the 13 countries of South America), due to the nature of event based calculations, different ruptures will be sampled in different countries. In practice, when comparing Chile with Peru you will see differences due to the fact that the random sampling picked different ruptures in the two contries and not real differences. In theory, the effect should disappear if the calculations have sufficiently long investigation times, when all possible ruptures are sampled, but in practice, for finite investigation times there will always be different ruptures.

2. To avoid such issues, the country-specific calculations should ideally all start from the same set of precomputed ruptures. You can compute the whole stochastic event set by running an event based calculation without specifying the sites and with the parameter `ground_motion_fields` set to false. Currently, one must specify a few global site parameters in the precalculation to make the engine checker happy, but they will not be used since the ground motion fields will not be generated in the precalculation. The ground motion fields will be generated on-the-fly in the subsequent individual country calculations, but not stored in the file system. This approach is fine if you do not have a lot of disk space at your disposal, but it is still inefficient since it is quite prone to the slow tasks issue.

3. If you have plenty of disk space it is better to also generate the ground motion fields in the precalculation, and then run the contry-specific risk calculations starting from there. This is particularly convenient if you foresee the need to run the risk part of the calculations multiple times, while the hazard part remains unchanged. Using a precomputed set of GMFs removes the need to rerun the hazard part of the calculations each time. This workflow has been particularly optimized since version 3.16 of the engine and it is now quite efficient.

4. If you have a really powerful machine, the simplest is to run a single calculation consid-

ering all countries in a single job.ini file. The risk profiles can be obtained by using the
`aggregate_by` and `reaggregate_by` parameters. This approach can be much faster than
the previous ones. However, approaches #2 and #3 are cloud-friendly and can be preferred if
you have access to cloud-computing resources, since then you can spawn a different machine
for each country and parallelize horizontally.

Here are some tips on how to prepare the required job.ini files:

When using approach #1 you will have 13 different files (in the example of South America) with a
format like the following:

```
$ cat job_Argentina.ini
calculation_mode = event_based_risk
source_model_logic_tree_file = ssmLT.xml
gsim_logic_tree_file = gmmLTrisk.xml
site_model_file = Site_model_Argentina.csv
exposure_file = Exposure_Argentina.xml
...
$ cat job_Bolivia.ini
calculation_mode = event_based_risk
source_model_logic_tree_file = ssmLT.xml
gsim_logic_tree_file = gmmLTrisk.xml
site_model_file = Site_model_Bolivia.csv
exposure_file = Exposure_Bolivia.xml
...
```

Notice that the `source_model_logic_tree_file` and `gsim_logic_tree_file` will be the
same for all countries since the hazard model is the same; the same sources will be read 13 times
and the ruptures will be sampled and filtered 13 times. This is inefficient. Also, hazard parameters
like

```
truncation_level = 3
investigation_time = 1
number_of_logic_tree_samples = 1000
ses_per_logic_tree_path = 100
maximum_distance = 300
```

must be the same in all 13 files to ensure the consistency of the calculation. Ensuring this consis-
tency can be prone to human error.

When using approach #2 you will have 14 different files: 13 files for the individual countries and a
special file for precomputing the ruptures:

```
$ cat job_rup.ini
calculation_mode = event_based
source_model_logic_tree_file = ssmLT.xml
```

```
gsim_logic_tree_file = gmmLTrisk.xml
reference_vs30_value = 760
reference_depth_to_1pt0km_per_sec = 440
ground_motion_fields = false
...
```

The files for the individual countries will be as before, except for the parameter `source_model_logic_tree_file` which should be removed. That will avoid reading 13 times the same source model files, which are useless anyway, since the calculation now starts from precomputed ruptures. There are still a lot of repetitions in the files and the potential for making mistakes.

Approach #3 is very similar to approach #2: the only differences will be in the initial file, the one used to precompute the GMFs. Obviously it will require setting `ground_motion_fields = true`; moreover, it will require specifying the full site model as follows:

```
site_model_file =
  Site_model_Argentina.csv
  Site_model_Bolivia.csv
  ...
```

The engine will automatically concatenate the site model files for all 13 countries and produce a single site collection. The site parameters will be extracted from such files, so the dummy global parameters `reference_vs30_value`, `reference_depth_to_1pt0km_per_sec`, etc can be removed.

It is FUNDAMENTAL FOR PERFORMANCE to have reasonable site model files, i.e. you should not compute the hazard at the location of every single asset, but rather you should use a variable-size grid fitting the exposure.

The engine provides a command `oq prepare_site_model` which is meant to generate sensible site model files starting from the country exposures and the global USGS vs30 grid. It works by using a hazard grid so that the number of sites can be reduced to a manageable number. Please refer to the manual in the section about the oq commands to see how to use it, or try `oq prepare_site_model --help`.

For reference, we were able to compute the hazard for all of South America on a grid of half million sites and 1 million years of effective time in a few hours in a machine with 120 cores, generating half terabyte of GMFs. The difficult part is avoiding running out memory when running the risk calculation; huge progress in this direction was made in version 3.16 of the engine.

Approach #4 is the best, when applicable, since there is only a single file, thus avoiding entirely the possibily of having inconsistent parameters in different files. It is also the faster approach, not to mention the most convenient one, since you have to manage a single calculation and not 13. That makes the task of managing any kind of post-processing a lot simpler. Unfortunately, it is also the option that requires more memory and it can be infeasable if the model is too large and you do not

have enough computing resources. In that case your best bet might be to go back to options #2 or #3. If you have access to multiple small machines, approaches #2 and #3 can be more attractive than #4, since then you can scale horizontally. If you decide to use approach #4, in the single file you must specify the `site_model_file` as done in approach #3, and also the `exposure_file` as follows:

```
exposure_file =
  Exposure_Argentina.xml
  Exposure_Bolivia.xml
  ...
```

The engine will automatically build a single asset collection for the entire continent of South America. In order to use this approach, you need to collect all the vulnerability functions in a single file and the taxonomy mapping file must cover the entire exposure for all countries. Moreover, the exposure must contain the associations between asset<->country; in GEM's exposure models, this is typically encoded in a field called `ID_0`. Then the aggregation by country can be done with the option

```
aggregate_by = ID_0
```

Sometimes, one is interested in finer aggregations, for instance by country and also by occupancy (Residential, Industrial or Commercial); then you have to set

```
aggregate_by = ID_0, OCCUPANCY
reaggregate_by = ID_0
```

`reaggregate_by`` is a new feature of engine 3.13 which allows to go from a finer aggregation (i.e. one with more tags, in this example 2) to a coarser aggregation (i.e. one with fewer tags, in this example 1). Actually the command ``oq reaggregate` has been there for more than one year; the new feature is that it is automatically called at the end of a calculation, by spawning a subcalculation to compute the reaggregation. Without `reaggregate_by` the aggregation by country would be lost, since only the result of the finer aggregation would be stored.

30.1 Single-line commands

When using approach #1 your can run all of the required calculations with the command:

```
$ oq engine --run job_Argentina.csv job_Bolivia.csv ...
```

When using approach #2 your can run all of the required calculations with the command:

```
$ oq engine --run job_rup.ini job_Argentina.csv job_Bolivia.csv ...
```

When using approach #3 your can run all of the required calculations with the command:

```
$ oq engine --run job_gmf.ini job_Argentina.csv job_Bolivia.csv ...
```

When using approach #4 your can run all of the required calculations with the command:

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

Here `job_XXX.ini` are the country specific configuration files, `job_rup.ini` is the file generating the ruptures, `job_rup.ini` is the file generating the ruptures, `job_gmf.ini` is the file generating the ground motion files and `job_all.ini` is the file encompassing all countries.

Finally, if you have a file `job_haz.ini` generating the full GMFs, a file `job_weak.ini` generating the losses with a weak building code and a file `job_strong.ini` generating the losses with a strong building code, you can run the entire an analysis with a single command as follows:

```
$ oq engine --run job_haz.ini job_weak.ini job_strong.ini
```

This will generate three calculations and the GMFs will be reused. This is as efficient as possible for this kind of problem.

30.2 Caveat: GMFs are split-dependent

You should not expect the results of approach #4 to match exactly the results of approaches #3 or #2, since splitting a calculation by countries is a tricky operation. In general, if you have a set of sites and you split it in disjoint subsets, and then you compute the ground motion fields for each subset, you will get different results than if you do not split.

To be concrete, if you run a calculation for Chile and then one for Argentina, you will get different results than running a single calculation for Chile+Argentina, *even if you have precomputed the ruptures for both countries, even if the random seeds are the same and even if there is no spatial correlation*. Many users are surprised but this fact, but it is obvious if you know how the GMFs are computed. Suppose you are considering 3 sites in Chile and 2 sites in Argentina, and that the value of the random seed in 123456: if you split, assuming there is a single event, you will produce the following 3+2 normally distributed random numbers:

```
>>> np.random.default_rng(123456).normal(size=3)   # for Chile
array([ 0.1928212 , -0.06550702,  0.43550665])
>>> np.random.default_rng(123456).normal(size=2)   # for Argentina
array([ 0.1928212 , -0.06550702])
```

If you do not split, you will generate the following 5 random numbers instead:

```
>>> np.random.default_rng(123456).normal(size=5)
array([ 0.1928212 , -0.06550702,  0.43550665,  0.88235875,  0.37132785])
```

They are unavoidably different. You may argue that not splitting is the correct way of proceeding, since the splitting causes some random numbers to be repeated (the numbers 0.1928212 and -0.0655070 in this example) and actually breaks the normal distribution.

In practice, if there is a sufficiently large event-set and if you are interested in statistical quantities, things work out and you should see similar results with and without splitting. But you will *never produce identical results*. Only the classical calculator does not depend on the splitting of the sites, for event based and scenario calculations there is no way out.

SPECIAL FEATURES OF THE ENGINE

There are a few less frequently used features of the engine that are not documented in the general user's manual, since their usage is quite specific. They are documented here.

31.1 Sensitivity analysis

Running a sensitivity analysis study means to run multiple calculations by changing a parameter and to study how the results change. For instance, it is interesting to study the random seed dependency when running a calculation using sampling of the logic tree, or it is interesting to study the impact of the truncation level on the PoEs. The engine offers a special syntax to run a sensitivity analysis with respect to one (or even more than one) parameter; you can find examples in the demos, see for instance the MultiPointClassicalPSHA demo or the EventBasedDamage demo. It is enough to write in the job.ini a dictionary of lists like the following:

```
sensitivity_analysis = {"random_seed": [100, 200, 300]}
sensitivity_analysis = {'truncation_level': [2, 3]}
```

The first example with run 3 calculations, the second 2 calculations. The calculations will be sequential unless you specify the `--many` flag in `oq engine --run --many job.ini`. The descriptions of the spawned calculation will be extended to include the parameter, so you could have descriptions as follows:

```
Multipoint demo {'truncation_level': 2}
Multipoint demo {'truncation_level': 3}
```

NB: from version 3.17 the engine is also able to run sensitivity analysis on file parameters. For instance if you want to run a `classical_risk` calculation starting from three different hazard inputs you can write:

```
sensitivity_analysis = {
  "hazard_curves_file": ["hazard1.csv", "hazard2.csv", "hazard3.csv"]}
```

31.2 The `custom_site_id`

Since engine 3.13, it is possible to assign 6-character ASCII strings as unique identifiers for the sites (8-characters since engine 3.15). This can be convenient in various situations, especially when splitting a calculation in geographic regions. The way to enable it is to add a field called `custom_site_id` to the site model file, which must be unique for each site.

The hazard curve and ground motion field exporters have been modified to export the `custom_site_id` instead of the `site_id` (if present).

We used this feature to split the ESHM20 model in two parts (Northern Europe and Southern Europe). Then creating the full hazard map was as trivial as joining the generated CSV files. Without the `custom_site_id` the site IDs would overlap, thus making impossible to join the outputs.

A geohash string (see https://en.wikipedia.org/wiki/Geohash) makes a good `custom_site_id` since it can enable the unique identification of all potential sites across the globe.

31.3 The `minimum_distance` parameter

GMPEs often have a prescribed range of validity. In particular they may give unexpected results for points too close to ruptures. To avoid this problem the engine recognizes a `minimum_distance` parameter: if it is set, then for distances below the specified minimum distance, the GMPEs return the ground-motion value at the minimum distance. This avoids producing extremely large (and physically unrealistic) ground-motion values at 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.

31.4 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/logictree/case_02/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 `mags` of shape M, `repi` of shape N and `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 `repi` into an equivalent distance by looking at the lookup table and use it to determine the `rjb` and `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 PointSources/AreaSources/MultiPointSources, fault sources are untouched.

31.5 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 `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

```
rupture_model_file = ruptures-min_mag=6_42.csv
```

in the `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.

31.6 `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 `<=` `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 `>` `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.