vivarium$_{c}luster_{t}oolsDocumentation$

Release 1.2.3

The vivarium developers

Mar 08, 2021

Contents

Vivarium cluster tools is a python package that makes running `vivarium` simulations at scale on a Univa Grid Engine cluster easy.

CHAPTER 1

Installation

You can install this package with

```
pip install vivarium-cluster-tools
```

In addition, this tool needs the redis client. This must be installed using conda.

```
conda install redis
```

CHAPTER 2

Running simulations in parallel

Once you successfully create a simulation specification and branch file it is time to use the distributed runner. Recall, we ran a single simulation with a model specification file in this way

```
simulate run /path/to/your/model/specification
```

Very similar to this, `vivarium-cluster-tools` includes a command for simulating in parallel

```
psimulate run /path/to/your/model/specification  /path/to/your/branch
```

By default, output will be saved in `/share/costeffectiveness/results`. If you want to save the results somewhere else you can specify your output directory as an optional argument

```
psimulate run /path/to/your/model/specification /path/to/your/branch -o /path/to/
↪output
```

Another optional argument is the cluster project under which to run the simulations. By default, the cluster project used is `proj_cost_effect`. To use a different project, specify it with the `-P` flag

```
psimulate run /path/to/your/model/specification /path/to/your/branch -P proj_csu
```

Currently, the projects that simulation science has access to are `proj_cost_effect`, `proj_cost_effect_diarrhea`, `proj_cost_effect_dcpn`, `proj_cost_effect_conic`, and `proj_csu`. Only these projects may be used.

If your `psimulate run` has failed to complete you can restart the failed jobs by specifying which output directory includes the partially completed jobs using `restart`

```
psimulate restart /path/to/the/previous/results/
```

For `psimulate restart` you can also choose a project with optional flag `-P`.

If you wish to expand a previous `psimulate run` by adding additional input draws and/or random seeds, you can do so using `expand`.

```
psimulate expand /path/to/the/previous/results/ --add-draws 10 --add-seeds 5
```

You can use one or both of `--add-draws` and `--add-seeds` to expand your simulation. Any previous results will not be overwritten, but any additional simulations resulting from the new input draws and/or random seeds will be run.

`psimulate expand` also supports choosing a project via the option flag `-P`.

CHAPTER 3

YAML Basics

YAML is a simple, human-readable data serialization format that is often used for specification files. The extensions of a file can be **.yaml** or **.yml**, both of which are accepted throughout the Vivarium framework. The following are general rules to keep in mind when writing and interpreting Vivarium YAML files. Examples use snippets from Vivarium model configurations but do not go in-depth about that topic. For more information about model configurations, please see that section of this documentation.

3.1 Structure

YAML files are structured by lines and space indentations. Indentation levels should be either 2 or 4 spaces, and **tabs are not valid**. For example, a configuration file that sets parameters for a BMI drug treatment component looks like the following:

```
configuration:
    bmi_treatment:
        age_cutoff: 20
        bmi_cutoff: 30
        adherence_proportion: 0.92
        treatment_proportion: 1.0
        treatment_available:
            year: 2019
```

(continues on next page)

(continued from previous page)

```
            month: 7
            day: 15
```

3.2 Comments

YAML comments are denoted with the pound symbol #, and can be placed anywhere, but must be separated from the preceding token by a space. For example, adding a comment to the configuration from above looks like this:

```
configuration:
    bmi_treatment:
        age_cutoff: 20
        bmi_cutoff: 30
        adherence_proportion: 0.92  #  Proportion of population selected who continue␣
↪treatment
        treatment_proportion: 1.0  # Proportion of population selected to be treated
        treatment_available:
            year: 2019
            month: 7
            day: 15
```

3.3 Mappings

A mapping, or key-value pairing, is formed using a colon :. This corresponds to an entry from the `dictionary` data structure from python, and there is no notion of ordering. Mappings can be specified in block format or inline, however we recommend block format so that is what we will show an example of here. In block format, mappings are separated onto new lines, and indentation forms a parent-child relationship. For example, below is a snippet from a configuration that specifies configuration parameters for a simulation population as mappings. Each colon below begins a mapping.

```
configuration:
    population:
        population_size: 1000
        age_start: 0
        age_end: 30
```

The interpretation of this configuration into python is shown below . You may have noticed that the above example contains nested mappings, this is valid YAML syntax and it relies on whitespace indentation. Also, the inner most block (population_size, age_start, age_end) is unordered.

```
{configuration: {
    population: {
        population_size: 1000,
        age_start: 0,
        age_end: 30
        }
    }
}
```

3.4 Lists

An in-line list in YAML is formed by a comma-separated set of items inside square brackets, similar to a python list. For example, below is a YAML configuration snippet that defines a list of years in which a hypothetical drug treatment is available in a simulation.

```
configuration:
    drug_treatment:
        available_years: [2015, 2016, 2017]
```

This will be interpreted in python as

```
{configuration:
    drug_treatment: {
        available_years: [2015, 2016, 2017]
    }
}
```

You may sometimes see a list in block format, which is also valid YAML syntax. Such a list is formed using a hyphen - and with each entry appearing on a new line with the same indentation level. The YAML example below is interpreted equivalently in python to the previous YAML example.

```
configuration:
    drug_treatment:
        available_years:
            - 2015
            - 2016
            - 2017
```

3.5 Composite Data

Lists and Mappings can be nested together to make more complicated structures. In fact, the previous mapping and list examples were taken from Vivarium model specifications and included nested mappings and lists. Vivarium model specifications will generally always take the form of these nested mappings, where some values are lists.

CHAPTER 4

The Branches File

When investigating a research question with the Vivarium framework, it usually becomes necessary to vary aspects of a *model specification* in order to evaluate the uncertainty of model outputs or to explore different scenarios based on model parameters. Without any extra tooling this would require manually manipulating the model specification file and re-running for each desired change, which would quickly get out of hand. The *branch configuration* helps us do this in a convenient way. This section will detail the common ways simulations are varied and the different aspects of a branch configuration that help us do this.

4.1 Uncertainty

Generating uncertainty for results is a core tenant of IHME and this is no different for simulation science. We are primarily concerned with two kinds of uncertainy in our model – *parameter uncertainy* and *stochastic uncertainty*. The branch configuration can help us explore both sources of uncertainty by varying both the *input draw* of the parameter data and the *seed* of the simulation's random number generator.

4.1.1 Parameter Uncertainty

Our simulations primarily rely on results from the Global Burden of Disease (GBD). GBD results are produced with *uncertainty* represented as *draws*. Once we have a model we trust, we typically want to capture our uncertainty in the input data by running the simulation model for several different input draws.

Note: A draw is a statistical term related to Bayesian statistics that has a specific meaning in the context of the GBD. The implementation details vary, but the purpose is for some quantity or measure of interest, a draw is a member of a full set of results such that, when taken together, the set of draws describes at least some of the uncertainty surrounding the quantity as a result of the modeling process, data uncertainty, etc. Generally, GBD results are produced in sets of 1000 draws.

To do this, we can use the `input_draw_count` key in a *branch configuration*. This key refers to an integer that represents the number of different input draws to generate simulations from.

Listing 1: parameter_uncertainty_branches.yaml

```
input_draw_count: 10
```

When we use this branch configuration along with the original *model specification*, we'll launch 10 simulations in parallel, each using a different set of input parameters represented by the draw number.

```
psimulate run /path/to/model_specification.yaml /path/to/parameter_uncertainty_
↪branches.yaml
```

Note: `psimulate` randomly selects the input draws it uses from the range [0, 999]. The selection happens without replacement, so specifying an `input_draw_count` of 10 guarantees you 10 unique input draws.

4.1.2 Stochastic Uncertainty

Vivarium simulations are probabilistic in nature. They use Monte Carlo sampling techniques to make decisions about who gets sick, who goes to the hospital, who dies, etc. This usage of randomness means our models have to consider the impact of *stochastic uncertainty* on its outputs.

There are two ways to handle stochastic uncertainty. The first is to increase the size of the population you're simulating. This will wash out outlier cases that might heavily skew your results. This works fine up to a point, but simulation run time scales directly with the size of the population you're simulating. Alternatively, you can run multiple simulations with different *random seeds* and aggregate your results across those simulations. This second approach takes advantage of parallel computing to keep run times under control.

Note: Random seeds are a convenient way to scale up a simulation's population in parallel. For example, running a simulation with one million simulants and a single random seed is equivalent to running the same simulation with ten thousand people and 100 random seeds. Because simulations specified with different seeds will be run in parallel, the latter run strategy is often preferable.

To run our simulation for multiple random seeds, we use the `random_seed_count` key in a *branch configuration*. This key specifies an integer that represents the number of different random seeds to use, each generated randomly and run in a separate simulation.

Listing 2: stochastic_uncertainty_branches.yaml

```
random_seed_count: 100
```

When we use this branch configuration along with the original *model specification*, we'll launch 100 simulations in parallel, each using a different random seed.

```
psimulate run /path/to/model_specification.yaml /path/to/stochastic_uncertainty_
↪branches.yaml
```

4.1.3 Combining Draws and Seeds

Since specifying either *input draws* or *random seeds* will result in multiple simulations being run, it is important to understand how *branch configurations* are parsed into simulations when both keys are specified. Specifying both an `input_draw_count` and a `random_seed_count` will result in a set of input draws and a set of random seeds being independently generated. Simulations will then be run for each unique combination of input draw and random seed (the Cartesian product of the two sets).

An example may make this clearer, so consider the following model specification.

Listing 3: combined_uncertainty_branches.yaml

```
input_draw_count: 100
random_seed_count: 10
```

It combines the two configuration keys we just learned about. Taken separately, the `input_draw_count` mapping would lead to 100 simulations on 100 draws of input data while the `random_seed_count` mapping would lead to ten simulations on with identical input data but a different seed for the random number generation. With both specified, the result is 1,000 total simulations, one for each member of the Cartesian product of those sets. That is, we would run ten simulations with the ten random seeds for each of the 100 input data draws.

4.2 Configuration Variations

A major function of *branch configurations* is to enable easy manipulation of the *configuration parameters* of a *model specification*. These parameters generally govern interesting features of an intervention, such as its target coverage or efficacy.

Within a branch configuration, you can specify several variations of these parameters to generate different scenarios or examine the sensitivity of a model to changes in a specific parameter. In the following sections we will describe a number of ways you can construct different scenarios and explain how to compute the number of simulations that will be run for a particular branch configuration.

Note: The following examples that alter configuration parameters all lie under a `branches` key. This is the only other top level key (besides `input_draw_count` and `random_seed_count`) that `psimulate` understands how to parse.

4.2.1 Single Parameter Variation

In order to illustrate the variation of a single *parameter*, let's assume you have defined a *model specification* that includes a dietary intervention of egg supplementation and that this intervention is parameterized by the proportion

of the population that is recruited into the intervention program. We may want to run simulations on several different proportions including full recruitment and no recruitment, which would function as a baseline. We can easily do this with the following branches file.

Listing 4: egg_intervention_branches.yaml

```yaml
branches:
        - egg_intervention:
                recruitment:
                    proportion: [0.0, 0.4, 0.8, 1.0]
```

The `branches` block specifies changes to values found in the configuration block of the original model specification YAML. The block found in the branches file must exactly match the block from the original model specification. Here, the YAML list [0.0, 0.4, 0.8, 1.0] dictates specific recruitment proportions to be simulated. Thus, you can expect four separate simulations to be run, one for each variation.

Warning: Varying the time step, start or end time, or the population size of a simulation will make profiling very difficult and runs the risk of breaking our output writing tools.

4.2.2 Interaction with Uncertainty

As touched upon in the section on *combining draws and seeds*, each of the top level keys in a *branch configuration* can be independently produce a set of simulations to be run. To find the total set of simulations to be run from a branch configuration file, we need to count the Cartesian product of the top level keys. We'll use a slight alteration of our intervention configuration as an example.

Listing 5: egg_intervention_with_parameter_uncertainty_branches.yaml

```yaml
input_draw_count: 100
random_seed_count: 4

branches:
        - egg_intervention:
                recruitment:
                    proportion: [0.0, 0.4, 0.8, 1.0]
```

This branch configuration will produce 400 simulations. First we consider the space of *configuration parameters* the simulation will be run for: one scenario for each of the four recruitment proportions. For each scenario, we will run a simulation for each combination of *input draw* and *random seed* specified by the `input_draw_count` and `random_seed_count` keys. So we'll have: `(Number of input draws) * (Number of random seeds) * (Number of scenarios) = 100 * 4 * 4 = 1600` simulations to run from this branch configuration.

4.2.3 Multi-parameter Variation

Branch configurations really shine when you want to vary a lot of aspects of your model.

Let's add another *parameter* to create scenarios along a new dimension. Say, for instance, we were also interested in the implementing the egg intervention by recruiting people only once they pass a certain age threshold. Provided components were available that can implement this, we could add a variety of starting ages to our branches file like so:

Listing 6: egg_intervention_with_ages_branches.yaml

```yaml
input_draw_count: 100

branches:
        - egg_intervention:
                recruitment:
                    proportion: [0.0, 0.4, 0.8, 1.0]
                    age_start: [10.0, 25.0, 45.0, 65.0]
```

This will result in scenarios encompassing every combination of recruitment proportion and starting age. Additionally, it will result in 100 simulations for each one of the scenarios, one for each of the *input draws*. This means the total number of simulations is given by `(Number of input draws) * (Number of recruitment proportions) * (Number of starting ages)` giving a total of 1600 simulations.

4.2.4 Complex Configurations

Let's look at a final example with a bit more going on. Note that in our last example *branch configuration* we did significantly more work than we needed to. When our recruitment proportion is `0`, it doesn't matter what age we start recruiting people at. This caused us to run 300 more simulations than we needed to. How do we write a better branch configuration?

Listing 7: better_egg_intervention_with_ages_branches.yaml

```yaml
input_draw_count: 100
random_seed_count: 4

branches:
        # Baseline scenario
        - egg_intervention:
            recruitment:
                proportion: 0.0
        # Intervention variations
        - egg_intervention:
            recruitment:
                proportion: [0.4, 0.8, 1.0]
                age_start: [10.0, 25.0, 45.0, 65.0]
```

The *YAML List* underneath the `branches` key denotes two different simulation scenario branches each with a set of *configuration parameters*. We resolve each one of the list items under the `branches` key separately. The first block resolves to a single baseline scenario. The second block resolves to three different recruitment proportions for four different ages, which produces a total of 12 intervention scenarios. Thus the entire `branches` block resolves to 13 different sets of configuration parameters.

Following the same logic as in the previous section, we compute the total number of simulations to be run as `(Number of input draws) * (Number of random seeds) * (Number of scenarios) = 100 * 4 * 13 = 5200`.

CHAPTER 5

Glossary

Model Specification A yaml file that details all components and configuration necessary to run a particular model.

Branch Configuration A yaml file that lists the count of input data draws and random seeds as well as a set of simulation configuration options. When coupled with a model specification, this file defines a set of different simulation scenarios that can be run in parallel with the `psimulate` command line utility.

Parameter Uncertainty Parameter uncertainty is uncertainty due to the input data. The Global Burden of Disease represents uncertainy distributions around the parameters it produces with *draws*. By running a simulation with several different draws of the input data, we can propagate the parameter uncertainty through our model and to our outputs.

Stochastic Uncertainty Stochastic uncertainty is uncertainty due to the inherent variability in the model. This variability is represented in a variety of places by using random numbers to sample from distributions. Our simulations control stochastic uncertainty with *random seeds*. Stochastic uncertainty in simulations is deeply related to sampling uncertainty in a study. By holding the input parameters constant and varying the random seed, we can produce many realizations (or samples) of the same underlying population and use that to minimize the stochastic uncertainty.

Input Draw A way of representing uncertainty in the input parameters. Rather than having an explicit distribution, each input draw of a parameter is a sample from the underlying distribution of that parameter in a population. Taken collectively, all the input draws form a numeric representation of the original parameter distribution.

Random Seed A number used to seed the simulation's random number generator. Two simulations run with the same random seed will produce the same random numbers for each decision made in the simulation on each time step for each person.

Configuration Parameter A parameter of a given model specified in the `configuration` block of the *model specification*. Typically determined early on in the model development process, these parameters determine what scenarios can be run with a given model. Common configuration parameters include the target coverage of an intervention, what subset of a population the intervention targets, and how effective an intervention is.

Index