PyPBEE User Manual

v1.0.0

Angshuman Deb
angshuman311@gmail.com
Joel P Conte
jpconte@ucsd.edu

Table of Contents

1 Introduction

PyPBEE is an open-source Python framework designed for performance-based earthquake engineering (PBEE) research. It enables probabilistic analyses across all stages of the PBEE methodology (from seismic hazard to loss assessment) through a modular, object-oriented design.

The framework supports custom model integration, uncertainty quantification, high-performance computing (HPC) execution, and direct interfacing with finite element (FE) platforms such as OpenSeesPy and OpenSeesTcl.

This manual provides step-by-step instructions for installing, configuring, and running PyPBEE using the supplied Bridge Column Example, which demonstrates the full PBEE workflow:

PrelimAnalysis → PSHA → GMS → NLTHA → PSDemHA → PSDamHA

The architecture, mathematical formulation, and theoretical background are described in the companion journal paper.

This manual focuses on practical usage and reproducibility

---

2 Installation and Setup

2.1 System Requirements

- **Python:** Version >=3.12, <3.13
- **Supported Platforms:** Windows, macOS, Linux
- **Dependencies:** NumPy, SciPy, Matplotlib, Pathos, PyDOE3, Benedict, PyGMM, OpenSeesPy, and others (installed automatically via requirements files)

2.2 Installation Overview

It is **highly recommended** to install PyPBEE inside a **new isolated environment** to prevent dependency conflicts.
PyPBEE can be set up using **five installation routes**. The first four assume you have forked and cloned the repository; the fifth installs directly from PyPI.

Route	Platform	Tool	Description
1. Windows (venv)	Windows	Virtual environment (venv)	Uses `setup_venv.bat` to create and configure a venv automatically
2. Windows (Conda)	Windows	Conda	Uses the provided `environment.yml` to set up a Conda environment
3. macOS (venv)	macOS	Virtual environment (venv)	Manual venv setup using `python3 -m venv`
4. macOS (Conda)	macOS	Conda	Uses the provided `environment.yml` to create a

Route	Platform	Tool	Description
			Conda environment
5. PyPI	All	pip	Install directly from PyPI (recommended for general users)

2.3 Forking and Cloning the Repository

1. Install **Git** from git-scm.com.

2. Go to https://github.com/angshuman311/PyPBEE and click **Fork** (top-right corner).
   This creates your own copy of the repository under your GitHub account.

3. Open **Git Bash** (Windows) or **Terminal** (macOS).

4. Navigate to your desired directory:

   ```
   cd path/to/your/projects
   ```

5. Clone **your fork** instead of the main repo:

   ```
   git clone https://github.com/<your-github-username>/PyPBEE
   ```

2.4 Installing on Windows using Virtual Environment (venv)

1. Install **Python ≥ 3.12 and < 3.13** from python.org.

2. Locate `setup_venv.bat` inside your cloned repository directory.

3. Double-click `setup_venv.bat`.

4. When prompted, browse to your `python.exe` location.

5. The script will:
   - Create `venv\\pypbee` inside the repository directory.
   - Install all required packages from `requirements.txt`.

6. Point your **IDE's Python interpreter** (e.g., in VS Code, PyCharm, or Spyder) to the **`python.exe`** inside `venv\\pypbee`.

2.5 Installing on Windows (Conda)

1. Install the **Anaconda** distribution.

2. Open **Anaconda Prompt**.

3. Navigate to your cloned repository directory:

   ```
   cd path\\to\\your\\PyPBEE
   ```

4. Create the environment from the provided file:

   ```
   conda env create -f environment.yml
   ```

5. Point your **IDE's interpreter** to the Conda environment you just created (`pypbee`).

2.6 Installing on macOS using Virtual Environment (venv)

1. Install **Python ≥ 3.12 and < 3.13** from python.org.

2. Open **Terminal**.

3. Navigate to your cloned repository directory:

   ```
   cd path/to/your/PyPBEE
   ```

4. Create and activate the environment:

   ```
   python3 -m venv venv/pypbee
   source venv/pypbee/bin/activate
   ```

5. Upgrade pip and install dependencies:

   ```
   python -m pip install --upgrade pip
   pip install -r requirements.txt
   ```

   Note: If installation fails due to `numba`, use the alternate file:

   ```
   pip install -r requirements-no-numba.txt
   ```

6. Point your **IDE's Python interpreter** to the virtual environment binary at `venv/pypbee/bin/python`.

2.7 Installing on macOS (Conda)

1. Install the **Anaconda** distribution.

2. Open **Terminal**.

3. Navigate to your cloned repository directory:

   ```
   cd path/to/your/PyPBEE
   ```

4. Create the environment:

   ```
   conda env create -f environment.yml
   ```

Note: If numba installation issues arise, use:

```
conda env create -f environment-no-numba.yml
```

5. Activate the environment:

   ```
   conda activate pypbee
   ```

6. Point your **IDE's interpreter** to the Conda environment (pypbee).

2.8 Installing Directly from PyPI

PyPBEE is also available on **PyPI**.

> ⚠️ **Important:**
> Always install PyPBEE inside a **new isolated virtual or Conda environment**.
> Do **not** install it in your base environment to avoid dependency conflicts.

Step 1 – Install Python or Anaconda

Ensure **Python 3.12 ≤ version < 3.13** is installed:
- Download from python.org and check "Add Python to PATH" during installation,
or install the **Anaconda** distribution (Conda will manage Python automatically).

Step 2 – Create an Environment

Using **virtual env**:

```
python -m venv pypbee-venv
source pypbee-venv/bin/activate    # macOS/Linux
pypbee-venv\\Scripts\\activate       # Windows
```

Using **Conda** (run inside Anaconda Prompt or Terminal):

```
conda create -n pypbee python=3.12.10
conda activate pypbee
```

Step 3 – Install PyPBEE

Without Numba:

```
pip install pypbee
```

With Numba acceleration:

```
pip install "pypbee[numba]"
```

2.9 Verifying the Installation

Run the following to confirm installation:

```
import pypbee
print(pypbee.__version__)
```

If no errors appear, your installation is successful and your environment is correctly configured.

2.10 Example Directory Structure

The example files provided in this release demonstrate the complete workflow for a Bridge Column. The structure below shows the recommended organization of files and folders used in PyPBEE examples.

```
Bride_Column_Work_Dir/
├── (will be populated by results from different PyPBEE analysis stages)

PyPBEE/
│
├── /scripts/Bridge_Column
│       ├── analysis_driver.py  # Executes all PBEE stages
│       ├── create_objects.py   # Defines inputs and creates analysis objects
│       ├── ops_model.py
│       ├── model_info.py
│
├── /pypbee/                    # PyPBEE source library (core modules)
│       ├── analysis.py
│       ├── psha.py, gms.py, nltha.py
│       ├── psdemha.py, psdamha.py
│       ├── structure.py, im.py, edp.py, ds.py
│       ├── structural_analysis_platform.py
│       └── utility.py, mixture.py, multivariate_nataf.py, ...
```

This structure ensures a clear separation between source code (/pypbee), scripts used for running analyses (/scripts), site hazard data, and output directories. The working directory (Bride_Column_Work_Dir/) is dynamically populated as you run the full PBEE workflow, automatically organizing outputs for each analysis stage (e.g., PSHA, GMS, NLTHA, PSDemHA, PSDamHA).

---

3 FE Modeling

This section documents the configuration file used to define the probabilistic, design, and model attributes for the structural model to be analyzed within PyPBEE. In this manual, these parameters are illustrated using a simplified structural system: a cantilever reinforced concrete bridge column. This example acts as a representative system for demonstrating how PyPBEE manages FE model parameters, random-variable sampling, design inputs, and different model forms throughout the modeling workflow. At present, the manual describes the model file only in Python using OpenSeesPy. PyPBEE fully supports model construction using OpenSeesTcl as well, and equivalent Tcl-based documentation will be added in a future release.

3.1 Bridge Column Model Description

The structural model used in this manual is a single reinforced concrete bridge column idealized as a cantilever. The column has a circular cross section discretized into fiber regions representing confined core concrete, unconfined cover concrete, and longitudinal reinforcing steel. The longitudinal reinforcement is distributed around the perimeter using bar clusters, allowing spatial variability in steel properties while maintaining a realistic reinforcement layout.

The base of the column is fully fixed, representing a rigid connection to the foundation or pile cap. The top of the column is free and connected to an idealized deck node, which carries a lumped tributary mass corresponding to the portion of superstructure dead load supported by the column. Gravity loads from both the column self-weight and the supported deck mass are applied consistently at the nodal level.

Nonlinear behavior is captured using beam-column formulations with user-defined fiber sections specified at multiple stations along the column height. Depending on the selected model form, the column may be represented by a single force-based beam-column element with multiple integration points, or by multiple displacement-based beam-column elements concentrated over a plastic hinge region and the remaining elastic height. This setup allows the model to represent curvature development, stiffness degradation, axial load interaction, and the spread of plasticity under seismic loading.

The model supports bidirectional lateral response, axial force effects, and material nonlinearity in both concrete and steel fibers. Section shear and torsional responses are incorporated through section aggregation, enabling coupled axial-flexural-shear behavior consistent with typical reinforced concrete bridge column response assumptions used in performance-based earthquake engineering studies.

<Add a figure>

3.2 Model Information Configuration File

This Python file defines all probabilistic, design, and model level parameters for the cantilever bridge column used in the example. It does not build an FE model itself. Instead, it supplies structured inputs that PyPBEE uses when it constructs and analyzes OpenSees models.

3.2.1 Baseline material properties and variability

The first block defines the nominal or expected values of key material properties for concrete, reinforcing steel, unit weight, and damping:

- Concrete
    - `fpc_e`: mean concrete compressive strength
    - `Ec_e`: mean concrete elastic modulus
- Steel
    - `fy_e`: mean steel yield strength
    - `fu_e`: mean steel ultimate strength
    - `Es_e`: mean steel elastic modulus
    - `b_e`: mean strain hardening ratio

- Mass and damping
    - `w_e`: concrete unit weight in kip per cubic inch
    - `xi_e`: reference damping ratio

Each of these has an associated coefficient of variation:

- *_COV values set the relative dispersion for `fpc`, `Ec`, `fy`, `fu`, `Es`, `b`, `w`, and `xi`.

For steel strengths, lower and upper bounds are also specified:

- `fy_min`, `fy_max`
- `fu_min`, `fu_max`

Shift parameters `b_shift` and `xi_shift` are used later when defining lognormal distributions so that the random variables remain positive.

Correlation coefficients define how some of these variables co-vary:

- `corr_fpc_Ec`, `corr_fy_fu`, `corr_fy_b`, `corr_fu_b`, `corr_Es_b`

These encode, for example, that concrete modulus is positively correlated with concrete strength, and that steel hardening b is correlated with `fy`, `fu`, and `Es`.

```
import numpy as np

fpc_e = 1.3 * 5  # Caltrans SDC v1.7
Ec_e = (40000. * np.sqrt(fpc_e * 1000) + 1e6) / 1000  # Caltrans SDC v1.7
fy_e = 69.144  # Darwin
fu_e = 95.1969  # Darwin
Es_e = 29200  # Lee Mosalam (2006)
b_e = 0.01  # calculated
w_e = 143.96 * 0.001 / (12 ** 3)  # Caltrans SDC v1.7
xi_e = 0.01  # Self

fpc_COV = 0.16  # Restrepo
Ec_COV = 0.10  # Restrepo
fy_COV = 0.04  # Carreno
fu_COV = fy_COV / 1.4  # Restrepo
Es_COV = 0.033  # Lee Mosalam (2006)
b_COV = 0.30  # calculatedf
w_COV = 0.04  # Restrepo, Weldon, Conte
xi_COV = 0.50  # Astroza

fy_min = 60  # Darwin
fy_max = 85.4  # Darwin

fu_min = 80  # Darwin
fu_max = 116  # Darwin

b_shift = 0.005
xi_shift = 0.003

corr_fpc_Ec = 0.40  # Restrepo
corr_fy_fu = 0.85  # Lee Mosalam (2006)
corr_fy_b = 0.40  # calculated
```

```
corr_fu_b = 0.50  # calculated
corr_Es_b = -0.13  # calculated
```

3.2.2 Probability distributions

The next block constructs a list of SciPy distribution objects in a fixed order:

- Index 0: `fpc` (`normal`)
- Index 1: `Ec` (`normal`)
- Index 2: `fy` (`beta`, bounded between `fy_min` and `fy_max`)
- Index 3: `fu` (`beta`, bounded between `fu_min` and `fu_max`)
- Index 4: `Es` (`normal`)
- Index 5: `b` (`lognormal`, shifted by `b_shift`)
- Index 6: `w` (`normal`)
- Index 7: `xi` (`lognormal`, shifted by `xi_shift`)

Shape parameters for the beta and lognormal distributions are computed using helper functions in `pypbee.Utility`, which take the specified mean, standard deviation, and bounds (for beta) or shift (for lognormal) and return the appropriate SciPy parameterization.

```
from pypbee.utility import Utility
from scipy.stats import beta, lognorm, norm

prob_dist_list = list()

# prob_dists
# 0: fpc
# 1: Ec
# 2: fy
# 3: fu
# 4: Es
# 5: b
# 6: w
# 7: xi

prob_dist_list.append(norm(fpc_e, fpc_COV * abs(fpc_e)))
prob_dist_list.append(norm(Ec_e, Ec_COV * abs(Ec_e)))
prob_dist_list.append(beta(a_fy, b_fy, fy_min, fy_max - fy_min))
prob_dist_list.append(beta(a_fu, b_fu, fu_min, fu_max - fu_min))
prob_dist_list.append(norm(Es_e, Es_COV * abs(Es_e)))
prob_dist_list.append(lognorm(*Utility.get_lognormal_dist_params(b_e, b_COV * abs(b_e), loc=b_shift)))
prob_dist_list.append(norm(w_e, w_COV * abs(w_e)))
prob_dist_list.append(lognorm(*Utility.get_lognormal_dist_params(xi_e, xi_COV * abs(xi_e), loc=xi_shift)))
```

A correlation matrix `corr_matrix` is then built as an identity matrix modified by the specified correlation coefficients. Only entries consistent with the chosen correlations are nonzero.

```
corr_matrix = np.eye(len(prob_dist_list))
corr_matrix[0, 1] = corr_fpc_Ec
corr_matrix[1, 0] = corr_fpc_Ec

corr_matrix[2, 3] = corr_fy_fu
corr_matrix[3, 2] = corr_fy_fu
corr_matrix[2, 5] = corr_fy_b
```

```
corr_matrix[5, 2] = corr_fy_b
corr_matrix[3, 5] = corr_fu_b
corr_matrix[5, 3] = corr_fu_b
corr_matrix[4, 5] = corr_Es_b
corr_matrix[5, 4] = corr_Es_b
```

The list `diag_blocks = [[0, 1], [2, 3, 4, 5], [6], [7]]` partitions the eight variables into blocks that can be treated as correlated groups during Nataf type sampling.

```
diag_blocks = [[0, 1], [2, 3, 4, 5], [6], [7]]
```

3.2.3 FE (random) model parameters

This section explains how random variables are mapped to specific model parameter names that the OpenSees model builder will use.

The central object is the list `rv_list`. Each entry in `rv_list` is a dictionary that defines a group of related random variables, specifying:

- which distributions to sample from
- how many independent sets are needed
- which parameter names in the model they should populate
- what the corresponding expected (mean) values are

Each rv_list entry has the form:

```
rv_list_entry = {
    'prob_dist_ind_list': […],
    'count': …,
    'name_list': […],
    'expected_value_list': […]
}
rv_list.append(rv_list_entry)
```

where:

- `prob_dist_ind_list` is a list of indices into `prob_dist_list`, identifying which underlying distributions are used for this group (for example `[0, 1]` for concrete strength and modulus).
- `count` is the number of independent samples (sets) of this group that are needed in the model. It equals the number of structural locations where this group of variables is required, such as sections, bar clusters, abutments, or shear keys.
- `name_list` is a flattened list of parameter names that will receive the sampled values. The length of name_list must equal `count * len(prob_dist_ind_list)`.
- `expected_value_list` contains the mean values corresponding to the distributions in `prob_dist_ind_list`, in the same order.

This setup is both specific to the bridge column example and generalizable. The same distributions can be reused across multiple components (columns, abutments, bearings, shear keys) simply by adding additional `rv_list_entry` blocks that point to the same `prob_dist_ind_list`, while providing different `name_list` and `count`.

Below, we describe the entries for the bridge column example.

3.2.3.1 Concrete parameters

The first entry in `rv_list` handles concrete strength and modulus for the column:

```
expected_value_list = [fpc_e, Ec_e]
name_list = []
for i_col in range(1, num_cols_total + 1):
    for i_sec in range(1, num_secdef_per_col + 1):
        name_list.append(f'fpc_col_{i_col}_secdef_{i_sec}')
        name_list.append(f'Ec_col_{i_col}_secdef_{i_sec}')

rv_list_entry = {
    'prob_dist_ind_list': [0, 1],
    'count': num_cols_total * num_secdef_per_col,
    'name_list': name_list,
    'expected_value_list': expected_value_list
}
rv_list.append(rv_list_entry)
```

Interpretation:

- `prob_dist_ind_list = [0, 1]` points to the distributions for `fpc` and `Ec` in `prob_dist_list`.
- `count = num_cols_total * num_secdef_per_col` specifies how many section definitions per column require concrete properties. For a single column with three section definitions, `count = 3`.
- `name_list` contains pairs of names for each section, for example:
    - `fpc_col_1_secdef_1, Ec_col_1_secdef_1`
    - `fpc_col_1_secdef_2, Ec_col_1_secdef_2`
    - `fpc_col_1_secdef_3, Ec_col_1_secdef_3`

At sampling time, PyPBEE draws count independent sets of (`fpc, Ec`) and assigns them to these names. Each section definition in the column thus gets its own concrete strength and modulus.

3.2.3.2 Steel parameters

The second entry handles steel parameters for all bar clusters in all section definitions:

```
expected_value_list = [fy_e, fu_e, Es_e, b_e]
name_list = []
for i_col in range(1, num_cols_total + 1):
    for i_sec in range(1, num_secdef_per_col + 1):
        for i_bc in range(1, num_bar_clusters + 1):
            name_list.append(f'fy_col_{i_col}_secdef_{i_sec}_barcluster_{i_bc}')
            name_list.append(f'fu_col_{i_col}_secdef_{i_sec}_barcluster_{i_bc}')
            name_list.append(f'Es_col_{i_col}_secdef_{i_sec}_barcluster_{i_bc}')
            name_list.append(f'b_col_{i_col}_secdef_{i_sec}_barcluster_{i_bc}')

rv_list_entry = {
    'prob_dist_ind_list': [2, 3, 4, 5],
    'count': num_cols_total * num_secdef_per_col * num_bar_clusters,
```

```
    'name_list': name_list,
    'expected_value_list': expected_value_list
}
rv_list.append(rv_list_entry)
```

Interpretation:

- `prob_dist_ind_list = [2, 3, 4, 5]` selects the steel distributions for `fy`, `fu`, `Es`, and `b`.
- `count = num_cols_total * num_secdef_per_col * num_bar_clusters` is the total number of bar clusters in the model. For one column, three section definitions, and four bar clusters per section, `count = 12`.
- For each of the 12 clusters, a 4-component vector (`fy, fu, Es, b`) is sampled and stored under four names:
    - `fy_col_1_secdef_s_barcluster_c`
    - `fu_col_1_secdef_s_barcluster_c`
    - `Es_col_1_secdef_s_barcluster_c`
    - `b_col_1_secdef_s_barcluster_c`

This yields 12 × 4 = 48 distinct steel random variables for the column.

3.2.3.3 Mass and damping

Global mass density and damping ratio are treated as a simple one-location group:

```
expected_value_list = [w_e, xi_e]
name_list = ['wconc_all', 'damp_ratio']

rv_list_entry = {
    'prob_dist_ind_list': [6, 7],
    'count': 1,
    'name_list': name_list,
    'expected_value_list': expected_value_list
}
rv_list.append(rv_list_entry)
```

Interpretation:

- `prob_dist_ind_list = [6, 7]` selects the distributions for unit weight and damping ratio.
- `count = 1` because these variables are global, not repeated per section or component.
- `name_list` has exactly two entries: `wconc_all` and `damp_ratio`.

One sample of (`wconc_all, damp_ratio`) is drawn and used throughout the model.

3.2.3.4 Reusing Complex Distributions

An important advantage of this `rv_list` design is that the underlying probability distributions (whether simple, mixture-based, or Bayesian posterior distributions) only need to be defined once in

`prob_dist_list`. Any component in the structural model can then reference those distributions by index through `prob_dist_ind_list`, without redefining or reconstructing them.

This becomes especially valuable when:

- the distribution for a parameter is a mixture model (e.g., combining lab data and field data),
- the distribution is a Bayesian posterior obtained from calibration,
- or the distribution relies on expensive pre-processing, such as kernel density estimation or parameter inference.

Under this structure:

- A complex distribution might take significant time to assemble or evaluate.
- But once it exists in `prob_dist_list`, it can be reused efficiently for any model component simply by referencing its index.
- New components, such as shear keys, backwalls, springs, or diaphragms, can obtain their random variables from the same source distribution without duplication.

For example, if the structural model included shear keys, the entry

```
num_shear_key = 4
expected_value_list = [fpc_e, fy_e, Es_e]
name_list = list()
for i_sk in list(range(1, num_shear_key + 1)):
    name_list.append(f'fpc_sk_{i_sk}')
    name_list.append(f'fy_sk_{i_sk}')
    name_list.append(f'Es_sk_{i_sk}')

rv_list_entry = {'prob_dist_ind_list': [0, 2, 4],
                 'count': num_shear_key,
                 'name_list': name_list,
                 'expected_value_list': expected_value_list
                 }
rv_list.append(rv_list_entry)
```

would allow the shear key properties to reuse the same distributions already defined for concrete strength `fpc` (index 0), steel yield strength `fy` (index 2), and steel modulus `Es` (index 4). Because these distributions are created once in `prob_dist_list`, they do not need to be regenerated for each component. This ensures consistent material variability across the model and avoids unnecessary computation, which is particularly helpful when the underlying distributions are mixture models or Bayesian posterior estimates.

3.2.3.5 Random model parameter container

Once the distributions, correlations, and random variable mappings have been defined, they are collected into a single dictionary called `random_model_params`:

```
random_model_params = dict()
random_model_params['rv_list'] = rv_list
random_model_params['prob_dist_list'] = prob_dist_list
random_model_params['corr_matrix'] = corr_matrix
random_model_params['diag_blocks'] = diag_blocks
random_model_params['sample_size'] = sample_size
```

```
random_model_params['sampling_method'] = sampling_method
random_model_params['estimation_sample_size'] = estimation_sample_size
random_model_params['dist_params_sample_size'] = dist_params_sample_size
```

This dictionary is the master container for all information needed to generate random realizations of the model parameters:

- `rv_list`
  List of group entries, where each entry describes which distributions to use, how many times to sample them (`count`), and which model parameter names should receive the sampled values.

- `prob_dist_list`
  Ordered list of the SciPy distribution objects (normal, beta, lognormal, mixtures, posteriors, etc.) from which all random variables are drawn.

- `corr_matrix`
  Correlation matrix that encodes linear correlations among the base distributions in `prob_dist_list`. Used by the sampling engine to generate correlated samples.

- `diag_blocks`
  List of index blocks that specify which subsets of variables are treated as correlated groups during sampling, for example `[[0, 1], [2, 3, 4, 5], [6], [7]]`.

- `sample_size`
  Number of Monte Carlo samples to generate for each design point.

- `sampling_method`
  Name of the sampling scheme to use. In this example it is `'lhs'` (Latin Hypercube Sampling), but other methods could be supported.

- `estimation_sample_size`
  Internal sample size used by PyPBEE for tasks such as parameter estimation uncertainty quantification.

- `dist_params_sample_size`
  Internal sample size used when estimating or validating distribution parameters, for example when fitting mixture models.

Downstream, the analysis driver receives `random_model_params` and uses it to generate consistent random samples for all parameter names that appear in `rv_list`.

3.2.4 Primary design variables

The block defining `primary_design_vars` specifies the design space for parametric studies. In PyPBEE, the user is free to choose any set of design variables. PyPBEE does not impose what the design variables must be; it only requires that they be provided in a structured form.

In the bridge column example, the chosen primary design variables are:

- column diameter (in feet)

- longitudinal reinforcement ratio

The first entry:

```
primary_design_vars = {'1': [5.51, 0.02]}
col_dia = [5, 6, 7, 8]
rho_long = [0.01, 0.015, 0.02, 0.025, 0.03]
```

represents the as-designed configuration of the bridge column. If the user wishes to perform a parametric design study, additional design points can be generated programmatically. In this example, the script constructs a grid of column diameters and reinforcement ratios:

```
design_num = 2
for i_col_dia in range(len(col_dia)):
    for i_rho_long in range(len(rho_long)):
        primary_design_vars.update(
            {f'{design_num}': [col_dia[i_col_dia], rho_long[i_rho_long]]}
        )
        design_num += 1
```

This loop fills `primary_design_vars` with multiple combinations of:

- entries from `col_dia = [5, 6, 7, 8]`
- entries from `rho_long = [0.01, 0.015, 0.02, 0.025, 0.03]`

creating a full factorial grid for parametric exploration.

However, a parametric grid is not required. PyPBEE operates perfectly well with only one design point, such as the as-designed configuration. The user may choose to add as many or as few design points as desired, depending on whether performance comparisons, trade-off studies, or automated design sweeps are needed.

The design points are then packaged into a container called `primary_design_params`:

```
n_dp = len(primary_design_vars.keys())
primary_design_params = dict()
primary_design_params['value_list_dict'] = primary_design_vars
primary_design_params['name_list'] = ['all_col_dia_in_ft', 'all_rho_long']
# to define grid point vs non-grid-point
primary_design_params['design_point_qualifier_dict'] = dict(
    zip(
        list(primary_design_vars.keys()),
        ['as-designed non-gp'] + ['gp'] * (n_dp - 1)
    )
)
```

Here:

- `value_list_dict` stores all design points and their associated design variable values
- `name_list` labels the design-variable vector [column_diameter, reinforcement_ratio]
- `design_point_qualifier_dict` marks design point "1" as “as-designed non-gp”, and all others as “gp” (grid points)

This structure allows PyPBEE to treat every design point uniformly during sampling and analysis, regardless of whether the user is analyzing the as-built configuration only or conducting an extensive parametric study across many designs.

3.2.5 Model Form Specification

PyPBEE allows multiple model-form options to be declared for a given structural model. These model forms may describe alternative damping formulations, element types, soil–structure interaction assumptions, or other modeling decisions. Each option is provided as a list. The first entry in the list is always treated as the default.

If no model-form study is intended, the list should contain exactly one item, ensuring deterministic model construction. This ensures no cross-model combinatorial expansion.

Model-form selection is controlled through the dictionary `model_attributes.`

```
model_attributes = dict()
model_attributes['vertical_gm'] = [0, 1]
model_attributes['ssi_springs'] = [0, 1]
model_attributes['steel_material'] = ['SteelMPF', 'Steel01']
model_attributes['col_elem_type'] = ['1', '2', '3']
model_attributes['damping'] = ['rayleigh_damping', 'modal_damping']
```

Each key corresponds to a specific modeling decision:

- `vertical_gm`: include or exclude vertical ground motion
- `ssi_springs`: activate or deactivate soil–structure interaction springs
- `steel_material`: select the steel constitutive model
- `col_elem_type`: choose among multiple element formulations
- `damping`: specify which damping formulation(s) to consider by providing the "`name`" of the damping model definition (see next section)

Model-form combinatorics occur only when a list contains more than one entry, and PyPBEE evaluates the full Cartesian product of all model-form lists. For example, if `col_elem_type` contains three items and `vertical_gm` contains two items (0: false, 1: true), PyPBEE will generate $3 \times 2 = 6$ distinct model-form combinations. If every model-form category contains multiple entries, PyPBEE explores the complete product of all choices across all lists.

3.2.5.1 Damping Model Forms

Two damping model forms are declared:

1. Rayleigh Damping (two-mode calibration)
2. Modal Damping (direct per-mode assignment)

```
damping_models = list()

damping_model = dict()
damping_model['name'] = 'rayleigh_damping'
damping_model['xi_i'] = ['damp_ratio'] * 2
damping_model['i'] = [1, 2]
damping_model['w_i'] = [None, None]
damping_models.append(damping_model)

damping_model = dict()
damping_model['name'] = 'modal_damping'
```

```
damping_model['xi_i'] = ['damp_ratio'] * 4 + list(np.linspace(0.01, 0.10, 10))
damping_models.append(damping_model)
```

Rayleigh damping defines

$$C = \alpha M + \beta K$$

and requires:

- two target modal damping ratios `xi_i`
- two mode indices `i`
- the corresponding modal frequencies `w_i` (computed at runtime)

The sampled values of `damp_ratio` (random variable defined above) are used to determine the α, β coefficients.

Modal damping assigns damping directly to each mode, without Rayleigh calibration:

$$\xi_i = \text{specified value}$$

The list `xi_i` may contain:

- string references to random variables (e.g., `damp_ratio`)
- numeric fixed damping values

The example:

`['damp_ratio'] * 4 + [0.01, 0.02, ..., 0.10]`

means:

- the first 4 modes use the sampled global damping ratio
- higher modes use fixed modal values

Because modal damping is direct assignment, it does not require mode indices `i`.

3.2.6 Other Model Parameters

The section:

```
other_model_params = dict()
other_model_params['value_list'] = [num_bar_clusters, num_secdef_per_col]
other_model_params['name_list'] = ['num_bar_clusters', 'num_secdef_per_col']
```

stores additional scalar parameters that are needed during model construction but are not random variables and are not part of the primary design space.

In the bridge column example, these parameters include:

- `num_bar_clusters`: the number of reinforcing bar clusters around the column perimeter
- `num_secdef_per_col`: the number of section definitions along the column height

These parameters are provided here so that the model-building script has access to structural bookkeeping quantities that may be useful during assembly. They are not inherently required by PyPBEE, but supplying them through `other_model_params` allows the model file to incorporate them whenever desired, for example, when determining how many sections to define, how many steel bar clusters to generate, or how to organize material groups. This keeps the model builder flexible: it can rely on the `other_model_params` values when needed, but it is not constrained to use them.

More generally, `other_model_params` is the place to pass deterministic model-construction parameters that accompany the model-parameter definitions established earlier (e.g., in `rv_list`, `primary_design_params`, and `damping_models`).

Any structural model that requires additional fixed integers or flags, such as number of bearings, number of shear keys, number of abutments, number of link elements, number of decks, etc., can store them here using the same pattern.

3.2.7 Model Parameters Container

All of the parameter groups defined above are finally assembled into a single dictionary named `model_params`:

```
model_params = dict()
model_params['primary_design_params'] = primary_design_params
model_params['random_model_params'] = random_model_params
model_params['other_model_params'] = other_model_params
model_params['damping_models'] = damping_models
model_params['model_attributes'] = model_attributes
```

This container acts as the central configuration object for the structural model. It gathers every piece of information needed by PyPBEE's model-building and analysis routines and passes them forward in a structured, unified format. Its components are:

- `primary_design_params`
  Describes the design space (column geometry, reinforcement ratio, and any user-defined design variables), including both the as-designed configuration and optional parametric design points.

- `random_model_params`
  Encodes all random-variable definitions: distributions, correlations, RV groupings (`rv_list`), sampling settings, and any additional inputs required for probabilistic sampling.

- `other_model_params`
  Holds extra deterministic parameters that the model builder may use, such as the number of bar clusters or the number of section definitions, allowing the model to incorporate these values when helpful.

- `damping_models`
  A library of available damping formulations, with names and configuration fields needed for Rayleigh or modal damping. These definitions are referenced by the entries in `model_attributes['damping']`.

- `model_attributes`
  Specifies which model-form options to use or explore (e.g., damping formulation, element type, vertical GM inclusion, SSI settings). The first entry in each list serves as the default, while additional entries enable model-form combinatorics.

Together, these dictionaries form a complete, self-contained specification of the structural model, its uncertainty description, its design alternatives, and its modeling assumptions. All subsequent steps in the PyPBEE workflow (hazard analysis, ground-motion selection, nonlinear analysis, and damage/loss assessment) draw from this unified configuration.

3.2.8 Site and Location Information

The final container specifies the minimal site metadata needed for the hazard calculations used in the current version of PyPBEE:

```
location_info = dict()
location_info['mechanism'] = 'U'
location_info['latitude'] = 37.7531
location_info['longitude'] = -121.1427
location_info['v_s30'] = 216.8470
location_info['region'] = 'california'
```

At present, the seismic hazard curve is obtained from the output of the OpenSHA Hazard Curve Calculator. Because the hazard values are pre generated outside PyPBEE, only a small set of inputs is required. The `latitude`, `longitude`, `v_s30`, and `region` label are included primarily for record keeping and for consistency with the site information used when creating the hazard curves in OpenSHA. Readers who wish to understand the procedure used in PyPBEE for reconstructing hazard curves from the OpenSHA deaggregation outputs are referred to Deb et al.(2021), which documents the full formulation.

In future versions, we plan to incorporate direct hazard calculations and ground motion selection using the OpenQuake engine. When that capability is added, this section of the manual will be updated to describe the additional inputs and options needed for fully integrated hazard analysis.

3.3 Model File

In PyPBEE, the structural model itself must be written in one of the two supported analysis platforms:

- OpenSees Tcl
- OpenSeesPy (Python)

Regardless of the platform, the model is expected to be written in a parametric form. All quantities that are to be treated as random variables, primary design variables, or other model parameters should appear in the model as named parameters, and those names should match the names used in the configuration file. For example, if the configuration file defines random variables called `fpc_col_1_secdef_1` and `damp_ratio`, then the OpenSees model should refer to those same quantities through parameters with exactly those names. The same convention applies to primary design parameters such as `col_dia` and `rho_long`, and to any additional parameters stored in `other_model_params`.

This naming contract is the key link between the configuration file and the model file. The configuration file defines the statistical and deterministic description of each parameter, while the model file consumes the realized values of those parameters to build the finite element model as described in the subsequent sections.

3.3.1 OpenSeesPy Model Builder: ops_model.py

`ops_model.py` is the user-authored OpenSeesPy model builder that constructs the finite element model for each analysis run. PyPBEE does not auto-generate this file. Users must code the model.

Before writing your own model, read and run the Bridge Column example and the driver scripts in the repository scripts folder (in particular, `scripts/examples/Bridge_Column`).

- **Required entry point:** `ops_model.py` must define `main(model_input_dict)`. Function `main` is responsible for building the OpenSees domain for the current realization, including nodes, elements, constraints, masses, and loads. The function should unpack `model_input_dict["model_param_vals"]` (realized random variables, primary design variables, and other model parameters) and `model_input_dict["model_attributes"]` (model-form options), then construct the model accordingly. The example `ops_model.py` for the Bridge Column is provided under `scripts/examples/Bridge_Column.`

- **Naming contract (critical)**: PyPBEE auto-generates a `model_input_dict` for each analysis run and passes it into `main(model_input_dict)` in `ops_model.py`. This dictionary has the form:

```
model_input_dict = {
    'model_info_directory': model_info_directory,
    'write_model_files': 1,
    'model_param_vals': model_param_vals,
    'model_attributes': model_attributes,
     …
}
```

  `model_info_directory` is the realization-specific working folder for the current design number, random-parameter realization, and model-form choice. This path is automatically created based on the `model_work_dir_path` provided when instantiating the `Structure` object (described later). Users should write any auxiliary files (for example fiber-section definition files, rebar material tag mappings, and other EDP-support files) into this directory when `write_model_files == 1` (this is automatic in `PrelimAnalysis`, and typically 0 in `NLTHA`, both described later).

  Most importantly, `model_param_vals` and `model_attributes` are the bridge between `model_info.py` and `ops_model.py`. `model_param_vals` contains (i) realized random model parameters (from `rv_list`), (ii) primary design parameters, and (iii) other model parameters; `model_attributes` contains the selected model-form options obtained by combinatorics over the attribute lists specified in `model_info.py`. The string keys in `model_param_vals` are the contract: `ops_model.py` must request values using exactly the same parameter names defined in `model_info.py`. PyPBEE will not infer or repair naming mismatches. If you rename a parameter in `model_info.py`, you must update `ops_model.py` and any dependent EDP or damage models accordingly.

- **File outputs for downstream post-processing.** If `model_input_dict["write_model_files"]` is enabled, `main` should write any auxiliary files required by the user-defined EDP and damage models to `model_input_dict["model_info_directory"]`. Inherited EDP classes (from the abstract base class EDP) commonly rely on section and material bookkeeping written here. Examples include fiber-section definition files (e.g., `fib_secdef_sec_<sectionTag>.txt`), material definition summaries (e.g., `material_data.txt`), and mappings of reinforcing-bar clusters to steel material tags (e.g., `col_rebar_mat_info.txt`). If you change file names or formats, update your EDP evaluators accordingly.

3.3.2 OpenSeesTcl Model Builder: main.tcl

TO BE ADDED IN A FUTURE RELEASE

---

4 Defining PBEE Workflow Objects

This section describes how the PBEE workflow objects are instantiated for the Bridge Column example. These objects collectively define the full probabilistic analysis pipeline, from structural model configuration through demand and damage hazard evaluation. At this stage, the objects are *defined and connected* but not yet executed.

4.1 Imports

The core PyPBEE classes that correspond to different PBEE entities and analyses are imported:

```
from pypbee.utility import Utility
from pypbee.structure import OSB
from pypbee.structural_analysis_platform import OpenSeesPy
from pypbee.prelim_analysis import PrelimAnalysis
from pypbee.sa_t import SaT
from pypbee.psha import PSHA
from pypbee.gms import GMS
from pypbee.edp import MaxColRebarStrain
from pypbee.nltha import NLTHA
from pypbee.psdemha import PSDemHA
from pypbee.ds import DS
from pypbee.psdamha import PSDamHA
from pypbee.interp_exterp_model import InterpExterpModel
from pypbee.pygmm_extension.boore_atkinson_2008 import BooreAtkinson2008
import pygmm.baker_jayaram_2008
from scipy.stats import lognorm
```

- `Utility`: helper utilities.
- `OSB`: the structural system container for an OpenSees bridge model (your "`Structure`" object).
- `OpenSeesPy`: the analysis platform wrapper that knows how to run OpenSeesPy using your local Python executable and model files.
- `PrelimAnalysis`: pre-analysis stage (typically gravity + modal info, periods, mode shapes, etc).
- `SaT`: intensity measure (`IM`) object for spectral acceleration at a specified period.
- `PSHA`: probabilistic seismic hazard analysis stage.
- `GMS`: ground motion selection stage.
- `MaxColRebarStrain`: an EDP definition.
- `NLTHA`: nonlinear time history analysis stage.
- `PSDemHA`: probabilistic seismic demand hazard analysis stage.

- DS: a damage-state definition.
- PSDamHA: probabilistic seismic damage hazard analysis stage.
- `InterpExterpModel`: interpolation and extrapolation methods used when processing hazard relationships.
- `BooreAtkinson2008, BakerJayaram2008`: the ground motion model and an `IM` correlation function used in the `IM` object.
- `lognorm`: `scipy` distribution used in hazard fitting and fragility specification.

4.2 Global run configuration

```
haz_lev_list = ['1', '2', '3', '4', '5', '6']
mrp_list = [72, 224, 475, 975, 2475, 4975]
```

- `haz_lev_list` is a list of hazard level identifiers (numeric strings).
- `mrp_list` gives the matching mean return periods (years) for those hazard levels.

```
name = 'Bridge_Column'
analysis_case = '100'
n_gm_list = [50] * len(mrp_list)
rng_seed = 'unique 6'
```

- `name` identifies the example/model folder (Bridge_Column).
- `analysis_case` is a user-facing switch that controls what sources of uncertainty are included in the run. In PyPBEE this maps to descriptive labels:

  '100' → Deterministic_FE_Model

  '200' → Incl_FE_Model_Param_Uncertainty

  '300' → Deterministic_FE_Model_Incl_Model_Form_Error

  '400' → Incl_Prob_Dist_Param_Estimation_Uncertainty

  In practice, this affects which iterators and sampling engines are activated when generating realizations and running downstream stages. For example, a deterministic model uses a single parameter set per design point, while parameter uncertainty uses sampled realizations from `rv_list`, and model-form error expands runs across the combinatoric model-attribute options.
- `n_gm_list` requests 50 ground motions per hazard level (so 50 for each MRP).
- `rng_seed` is a seed policy or seed tag used for reproducibility. It controls how random draws are seeded for reproducibility.

  If `rng_seed` is a string starting with "unique" (for example "unique 6"), PyPBEE will generate a reproducible integer seed by multiplying a seed multiplier (here 6) by an integer encoding of unique design numbers, model option numbers, model realization numbers, and hazard level numbers.

  Otherwise, PyPBEE uses the provided integer `rng_seed` directly.

This gives predictable, traceable randomization across design points and other iterator dimensions, while still ensuring different designs and analysis cases do not share identical random samples.

4.3 Paths

```
base_dir_path = r"C:\Users\adeb\Work\PyPBEE"
base_work_dir_path = r"C:\Users\adeb\Work\PyPBEE_Work_Dir"
model_files_path = os.path.join(base_dir_path, 'scripts', 'examples', name)
model_work_dir_path = os.path.join(base_work_dir_path, f"{name}_Work_Dir")
local_opensees_path = os.path.join(base_dir_path, "vendor", "OpenSees_Windows", "OpenSees")
local_python_path = os.path.join(base_dir_path, "venv", "pypbee", "Scripts", "python.exe")
gm_database_dir_path = r"C:\Users\adeb\Work\NGA_PEER_EQ_DB"
```

- `model_files_path` points to the folder that contains your user-authored model files, especially `model_info.py` and `ops_model.py`.
- `model_work_dir_path` is the root output directory where PyPBEE will auto-create realization-specific subfolders for each stage.
- `local_python_path` is the Python executable used to run OpenSeesPy in the analysis subprocesses. Typically point it to your virtual environment's or conda environment's `python.exe`
- `gm_database_dir_path` points to your local NGA/PEER ground motion database used by the GMS stage. This folder should .AT2 files inside directories like LANDERS/FAI095.AT2, LANDERS/FAI185.AT2, LANDERS/FAI-UP.AT2, NORTHR/SAN090.AT2, NORTHR/SAN180.AT2, NORTHR/SAN-UP.AT2, and so on.

4.4 Create the Structure Object

```
model_params = Utility.import_attr_from_module(
    model_files_path, f"model_info", 'model_params'
)
location_info = Utility.import_attr_from_module(
    model_files_path, f"model_info", 'location_info'
)
structural_analysis_platform = OpenSeesPy(model_files_path,
local_python_path)
osb = OSB(
    name,
    location_info,
    model_files_path,
    model_work_dir_path,
    model_params,
    structural_analysis_platform
)
design_num_list = osb.get_design_num_list('all')
```

- `model_params` is the full container that includes primary design variables, random model parameters, other deterministic parameters, damping models, and model attributes.
- `location_info` holds site metadata needed for hazard and ground motion selection (lat, lon, Vs30, region, etc).
- `OpenSeesPy(...)` tells PyPBEE: "this Structure is analyzed using OpenSeesPy, with model files located here, executed using this Python."

- `OSB(...)` builds the central structure container. This is the object that will:
    - manage design points and realizations
    - manage the auto-generated working directories
    - act as the common dependency shared by IM, EDP, and analysis stages
- `design_num_list` collects the design numbers you want to run (here: all defined design points). The `get_design_num_list(…)` also works with a list of primary design points. For, example:

```
osb.get_design_num_list([[5.51, 0.02], [5, 0.01]])
```

  This will return [‘1’, ‘2’] based on the design points defined in `model_info.py`

4.5 Define the Intensity Measure (IM): Sa(T)

```
gmm = BooreAtkinson2008
correl_func = pygmm.baker_jayaram_2008.calc_correls
im = SaT(osb, gmm, correl_func, 'T_1')
```

- This defines the IM as spectral acceleration at a period (here keyed by 'T_1' meaning the first mode period which will be extracted internally). If the period is known, a floating point value can also be provided instead of ‘T_1’.
- `gmm` is the ground motion model used to compute `IM` hazard and selection quantities. `BooreAtkinson2008` is a custom GMPE implementation in PyPBEE implemented by subclassing the `pygmm.model.GroundMotionModel` and defining the specific equations and coefficients of the GMPE.
- `correl_func` provides cross-period correlation behavior (useful when you work with multiple periods or spectral shape selection). The `calc_correls` function in `pygmm.baker_jayaram_2008` is used.

4.6 Engineering Demand Parameters and their Demand-Model Requirements

This block defines (1) the `EDP`s to be extracted from `NLTHA` and (2) the hazard-modeling rules used later when converting NLTHA response samples into continuous demand models (for `PSDemHA`). In the Bridge Column example, `MaxColRebarStrain` is one of the EDP classes used downstream in `NLTHA, PSDemHA,` and `PSDamHA`.

4.6.1 Defining the EDP list

```
edp_list = [
    MaxColRebarStrain(
        max_what='compression',
        frame_structure=osb,
        tag='1', recorder_file_storage='shared'
    ),
    MaxColRebarStrain(
        max_what='tension',
        frame_structure=osb,
```

```
        tag='2',
        recorder_file_storage='shared'
    ),
]
```

Each `edp_list` entry creates an EDP object that tells PyPBEE what response quantity to extract from analysis results:

- `MaxColRebarStrain('compression', ...)`
  Maximum compressive longitudinal rebar strain.
- `MaxColRebarStrain('tension', ...)`
  Maximum tensile longitudinal rebar strain.

Common arguments:

- `osb`: the Structure object.
- `tag='1', '2'`: EDP identifiers. These are commonly used to tag outputs, organize folders, and map results to EDP-specific processing.
- `recorder_file_storage`: recorder-sharing mode. `'shared'` means these EDPs read from the same recorder file set (one recorder folder), instead of each EDP writing and reading its own recorder outputs. This is appropriate when multiple EDPs can be computed from the same recorded response quantities. The other option is `'separate'`.

4.6.2 Defining `haz_req`: how conditional demand is modeled

```
haz_req = dict()
haz_req['fit_dist'] = lognorm
haz_req['transf_on_dist_params_list'] = [
    lambda x, y, z: x,
    lambda x, y, z: y,
    lambda x, y, z: z
]
haz_req['transf_to_dist_params_list'] = [
    lambda x, y, z: x,
    lambda x, y, z: y,
    lambda x, y, z: z
]
haz_req['interp_exterp_function_list'] = [
    InterpExterpModel.piecewise_linear_interp_constant_extrap,
    lambda x, y, xq: 0 * xq,
    InterpExterpModel.power_law_regression_lin
]
[edp.set_haz_req(haz_req) for edp in edp_list]
```

`haz_req` is attached to each EDP and is used later during PSDemHA. Conceptually, PSDemHA needs to construct a model for the conditional distribution:

$$\text{EDP} \mid \text{IM}$$

This is typically done by fitting a distribution at a set of IM levels (or IM bins), then interpolating or extrapolating the fitted parameters to any IM value required during hazard calculations.

4.6.2.1 *fit_dist*

This specifies that the conditional distribution for demand is modeled as a SciPy lognormal distribution.

In SciPy, scipy.stats.lognorm is parameterized using three parameters:

1. shape (usually denoted `s`, and equal to the standard deviation of ln(EDP))
2. loc (location shift)
3. scale (equal to $\exp(\mu)$, where μ is the mean of ln(EDP))

So, when PyPBEE fits lognorm, the distribution parameter vector is treated as a 3-tuple in SciPy's order:

(shape, loc, scale)

4.6.2.2 *transf_on_dist_params_list and transf_to_dist_params_list*

These lists define parameter transformations between:

- the parameterization used internally for modeling and interpolation, and
- the parameterization required by the SciPy distribution object.

In the above example, each transformation is the identity map. That means PyPBEE uses the SciPy parameterization directly for interpolation and exterpolation, without converting the parameters to an alternative form. For example, if the interpolation and extrapolation were instead to be performed in terms of the parameter triplet (shape, loc, μ), where μ = ln(scale), rather than SciPy's native (shape, loc, scale), the transformation lists would be defined explicitly as mappings between these two representations as follows:

```
haz_req['transf_on_dist_params_list'] = [
    lambda x, y, z: x,           # shape -> shape
    lambda x, y, z: y,           # loc   -> loc
    lambda x, y, z: np.log(z),   # scale -> mu
]
haz_req['transf_to_dist_params_list'] = [
    lambda x, y, z: x,           # shape -> shape
    lambda x, y, z: y,           # loc   -> loc
    lambda x, y, z: np.exp(z),   # mu    -> scale
]
```

4.6.2.3 *Interpolation and extrapolation functions for the distribution parameters*

This list contains one function per SciPy lognormal parameter, in the same order:

1. function for shape
2. function for loc
3. function for scale

Each function must implement the same interface:

- `x`: known `IM` values (where parameter values have been fit)

- y: parameter values at those x
- xq: query IM values where parameters are needed

It returns yq: parameter values at xq. `InterpExterpModel` is the designated module for these interpolation and extrapolation functions in the workflow. The ones used in the example are:

- `InterpExterpModel.piecewise_linear_interp_constant_extrap(x, y, xq)` for shape:
    - Interpolates linearly in log(IM)
    - Extrapolates as constants outside the fitted IM range.
    - This is a stable choice for a parameter you do not want to grow unbounded when extrapolating outside the calibration IM range.
- `lambda x, y, xq: 0 * xq` for loc:
    - This returns zeros at all query points. In practice this enforces a constant loc parameter of 0.
    - This is consistent with a common lognormal modeling assumption in PBEE demand modeling: loc = 0, meaning the distribution is supported on positive EDP values without an additive shift.
- `InterpExterpModel.power_law_regression_lin(x, y, xq)` for scale:
    - This fits a power-law model: $y = ax^b$. It does so by performing linear regression in log space (least squares on $\log(x)$ and $\log(y)$), then evaluates the fitted power law at xq.
    - This is often a good choice for parameters that are expected to vary smoothly and monotonically with IM, while staying positive.

Finally,

```
[edp.set_haz_req(haz_req) for edp in edp_list]
```

attaches the same hazard-modeling specification to every EDP in `edp_list`. After this line:

- all three EDPs will be modeled using a SciPy lognormal conditional distribution
- all three will use the same parameter transformation rules (identity)
- all three will interpolate/extrapolate the three lognormal parameters using the specified per-parameter functions

---

4.7 Damage States

Damage states translate engineering demand parameters (EDPs) into discrete damage events by comparing demand to a capacity or limit state. In PyPBEE, a damage state is defined by combining:

- a governing EDP,
- a capacity (limit-state) model,

- and a damage-specific uncertainty model.

In this example, two damage states are defined, corresponding to the two previously defined rebar-strain EDPs.

```
ds_list = list()
ds_list.append(
    DS(
        edp_list[0],
        lambda x: 0.004,
        haz_req={
            'normalized_fragility_dist': lognorm(0.326, 0, 1.02),
            'estimation_sample_size': 5
        },
        ds_type='col_rebar_strain_damage'
    )
)
ds_list.append(
    DS(
        edp_list[1],
        lambda x: 0.03 + 700 * x[1] * x[2] / x[3] - 0.1 * x[8] / (x[4] * x[5]),
        haz_req={
            'normalized_fragility_dist': lognorm(0.201, 0, 1.05),
            'estimation_sample_size': 5
        },
        ds_type='col_rebar_strain_damage'
    )
)
```

- Each DS object is associated with a single EDP:
    - The first damage state uses `edp_list[0]`, corresponding to maximum compressive longitudinal rebar strain.
    - The second damage state uses `edp_list[1]`, corresponding to maximum tensile longitudinal rebar strain.
- The second argument to DS is a Python callable defined in terms of `x`, where `x` is the vector read from predictor-information files written by the OpenSees model during model construction and gravity analysis. Each entry `x[i]` corresponds to the `i`-th parameter (1-based index) written by the model, in the order defined by the user in ops_model.py. This defines a predictive capacity model used to normalize and denormalize fragility functions as described in Deb et al. (2021) and Deb et al. (2022).
- Each damage state also includes a DS-specific `haz_req` dictionary:

```
haz_req={
    'normalized_fragility_dist': lognorm(…),
    'estimation_sample_size': 5
}
```

    - `normalized_fragility_dist` defines uncertainty in the capacity model using a normalized distribution. The normalized fragility models uncertainty in capacity in terms of a normalized EDP (EDP divided by predictive capacity). For a given structural realization, this fragility is denormalized using the predictive capacity computed above, yielding a capacity distribution for the structure being analyzed in physical EDP units. This procedure is described in the above-referenced literature (Deb et al. 2021, 2022).
    - `estimation_sample_size` controls internal Monte Carlo sampling to account for small sample uncertainty (`analysis_case = '400'`).

- Finally, `ds_type = 'col_rebar_strain_damage'` is a user-defined label that identifies the damage mechanism. The structural model must include this string verbatim in the predictor information file name written during model construction.

  Predictor files are read using the pattern `predictor_info_{ds_type}_{ds_str}.txt`, where `ds_str` identifies the structural location and is set equal to `edp_str`. The string `edp_str` is generated by the governing EDP through its "concrete" subclass implementation of the method `get_edp_strings()`. (discussed later in the public interface of PBEE entities section). Consequently, the structural model must also include this string verbatim in the predictor file name.

  In the bridge column example, `edp_str` (and hence `ds_str`) is of the form "`col_1_edge_1`", since there is one column and one edge where EDPs are evaluated. This follows directly from the model configuration in model_info.py:

```
num_cols_total = 1
col_edge_list = [1]
```

  Because `edp_str` is generated by the specific EDP subclass, the linkage between the predictor file written by the model and the damage state definition is explicit and user-defined. PyPBEE does not infer or reconcile this mapping; the user must ensure that the predictor file names written by the structural model are consistent with both `ds_type` and the EDP's `edp_str` generation logic.

4.8 Analysis Objects

This block defines the **analysis-stage objects** that together execute the PBEE workflow, along with several auxiliary parameters that control resolution, numerical behavior, and computational performance. These objects are defined here but are executed later by the analysis driver.

The following objects define the **analysis stages** of the PBEE workflow. Each object represents one stage in the standard PBEE sequence and is instantiated here with explicit dependencies on the previously defined Structure, IM, EDPs, and DSs.

```
prelim_analysis = PrelimAnalysis(osb, num_modes)
psha = PSHA(im)
gms = GMS(im)
nltha = NLTHA(edp_list, im)
psdemha = PSDemHA(edp_list, im)
psdamha = PSDamHA(ds_list, im=im, sol_type='numerical')
```

`PrelimAnalysis` defines the preliminary structural analysis stage. It operates on the `Structure` object and typically includes gravity analysis and modal analysis up to `num_modes`. This stage is also responsible for writing realization-specific auxiliary outputs, such as modal properties and predictor information files, which are required by downstream demand and damage evaluations.

`PSHA` defines the probabilistic seismic hazard analysis stage for the chosen intensity measure (`IM`). It organizes hazard information in terms of the `IM` and provides the hazard representation needed for ground motion selection and subsequent analyses.

`GMS` defines the ground motion selection stage. Using the hazard information associated with the `IM`, it selects suites of ground motions that are consistent with the specified hazard levels and ground motion model assumptions.

NLTHA defines the nonlinear time history analysis stage. It executes OpenSees analyses for each selected ground motion and extracts the engineering demand parameters listed in `edp_list`. Recorder outputs are written according to the storage rules defined by each EDP.

PSDemHA defines the probabilistic seismic demand hazard analysis stage. It fits probabilistic demand models for each EDP as a function of the `IM`, using the hazard requirements attached to the EDP objects, and computes demand hazard relationships.

PSDamHA defines the probabilistic seismic damage hazard analysis stage. It propagates demand hazard through the damage-state definitions in `ds_list` to compute damage hazard curves. The argument `sol_type='numerical'` specifies that the damage hazard integral is evaluated numerically. Other options include `'cf-cornell'` and `'cf-vamvatsikos'` following closed-form approximations of damage hazard integral proposed by Cornell et al. (2002) and Vamvatsikos (2013), respectively.

---

5 Analysis Driver

The analysis driver script is responsible for orchestrating execution of the workflow objects defined in `create_objects.py`. It imports the configured analysis case, iterator lists, shared settings (for example RNG strategy and database paths), and the instantiated analysis-stage objects, then runs the desired stages in sequence.

```
from create_objects import analysis_case, rng_seed, \
    gm_database_dir_path, \
    design_num_list, haz_lev_list, mrp_list, n_gm_list, \
    prelim_analysis, psha, gms, nltha, psdemha, psdamha

if __name__ == "__main__":

    pool_size = 12

    prelim_analysis.setup(analysis_case, design_num_list, rng_seed=rng_seed)
    prelim_analysis.run(analysis_case, pool_size)
    prelim_analysis.wrap_up(analysis_case)
```

This block defines the execution entry point for the analysis driver and controls how the preliminary analysis stage is executed.

The `if __name__ == "__main__":` guard is required to safely enable parallel execution in Python. PyPBEE uses multiprocessing for several analysis stages, and on platforms such as Windows and macOS (using the spawn start method), each worker process starts by importing the main module. Without this guard, the driver script would be re-executed in every worker process, causing unintended recursive process creation, duplicated analyses, or runtime errors. Wrapping the execution logic inside the `__main__` block ensures that setup, execution, and cleanup code is run only in the main process, while worker processes import the module without triggering execution.

`pool_size = 12` specifies the number of local worker processes to be used for parallel execution. This value controls how many realizations can be processed concurrently and should be chosen based on available CPU resources.

5.1 PrelimAnalysis

The preliminary analysis stage is then executed using a standardized three-step lifecycle:

- `setup(...)` prepares the analysis stage by expanding the iterator space based on `analysis_case` and `design_num_list`, initializing realization-specific working directories, and configuring random number seeds according to `rng_seed`.
- `run(...)` performs the actual computations for the stage. For `PrelimAnalysis`, this typically includes gravity and modal analyses. The computation is parallelized using the specified `pool_size`.
- `wrap_up(...)` finalizes the stage by performing bookkeeping tasks, consolidating outputs, and ensuring that all realization-specific artifacts required by downstream stages (such as predictor files and modal properties) are complete.

Together, this block cleanly separates execution control, parallel configuration, and stage lifecycle management, ensuring that the preliminary analysis is run safely, reproducibly, and efficiently as the first step of the PBEE workflow.

5.2 PSHA

5.2.1 PSHA Prerequisites and OpenSHA Setup

In the current implementation, PyPBEE relies on the output of OpenSHA's Hazard Curve Calculator to obtain the seismic hazard information required for subsequent stages. The PSHA stage in PyPBEE does not perform a full seismic source characterization internally. The seismic source characterization is obtained by performing a dummy hazard analysis and hazard disaggregation using OpenSHA. This is explained in Deb et al. (2021)

5.2.1.1 Hazard curve calculation in OpenSHA

The user should run OpenSHA's Hazard Curve Calculator for the site of interest, using:

- Intensity Measure: Spectral Acceleration (Sa)
- Period: 1.0 s
- GMPE: Boore and Atkinson (2008)
- Site parameters: Latitude, longitude, and Vs30 corresponding to the structure location

Under the ERF & Time Span tab, it is essential to set:

- Duration (Years) = 1.0

This setting ensures that the resulting hazard values are expressed as mean annual rates (MARs), which is the form required by PyPBEE.

5.2.1.2 Disaggregation setup

After computing the hazard curve, the user must perform magnitude–distance (M–R) disaggregation. This information is required by PyPBEE to reconstruct seismic hazard in terms of the target intensity measure used in the workflow.

In the Control Panel, select Disaggregation, and configure the following:

- Disaggregate IML: enabled
- IML level: a *very small* spectral acceleration value, for example
- Sa = 1e-4 g
- Magnitude bins:
- M = 5.0 to 9.0, increment 0.1
- Distance bins:
- R = 5 km to 200 km, increment 1 km

A fine bin resolution is recommended to accurately capture the contribution of individual earthquake scenarios.

5.2.1.3 Rationale (brief)

The theoretical justification for this procedure is described in detail in Deb et al. (2021) and is only summarized here. At very small spectral acceleration values (e.g., 1e-4 g), exceedance is virtually guaranteed for any earthquake scenario. Consequently, the mean annual rate of exceedance at this level converges to the total rate of all contributing earthquake scenarios at the site.

By combining:

- the MAR at very small Sa,
- the M–R disaggregation results,
- and a GMPE independent of explicit fault characteristics (such as Boore and Atkinson, 2008),

the seismic hazard integral can be evaluated in a discrete and computationally tractable form, allowing PyPBEE to reconstruct hazard curves for the desired intensity measure without re-performing full PSHA internally.

5.2.1.4 Saving required OpenSHA outputs

After clicking Compute in the main OpenSHA window and once the disaggregation results are displayed, the user must save two outputs:

1. Mean/Mode tab
    - Save the contents to a file named `deagg_summary_small_sa.txt`

      Location:

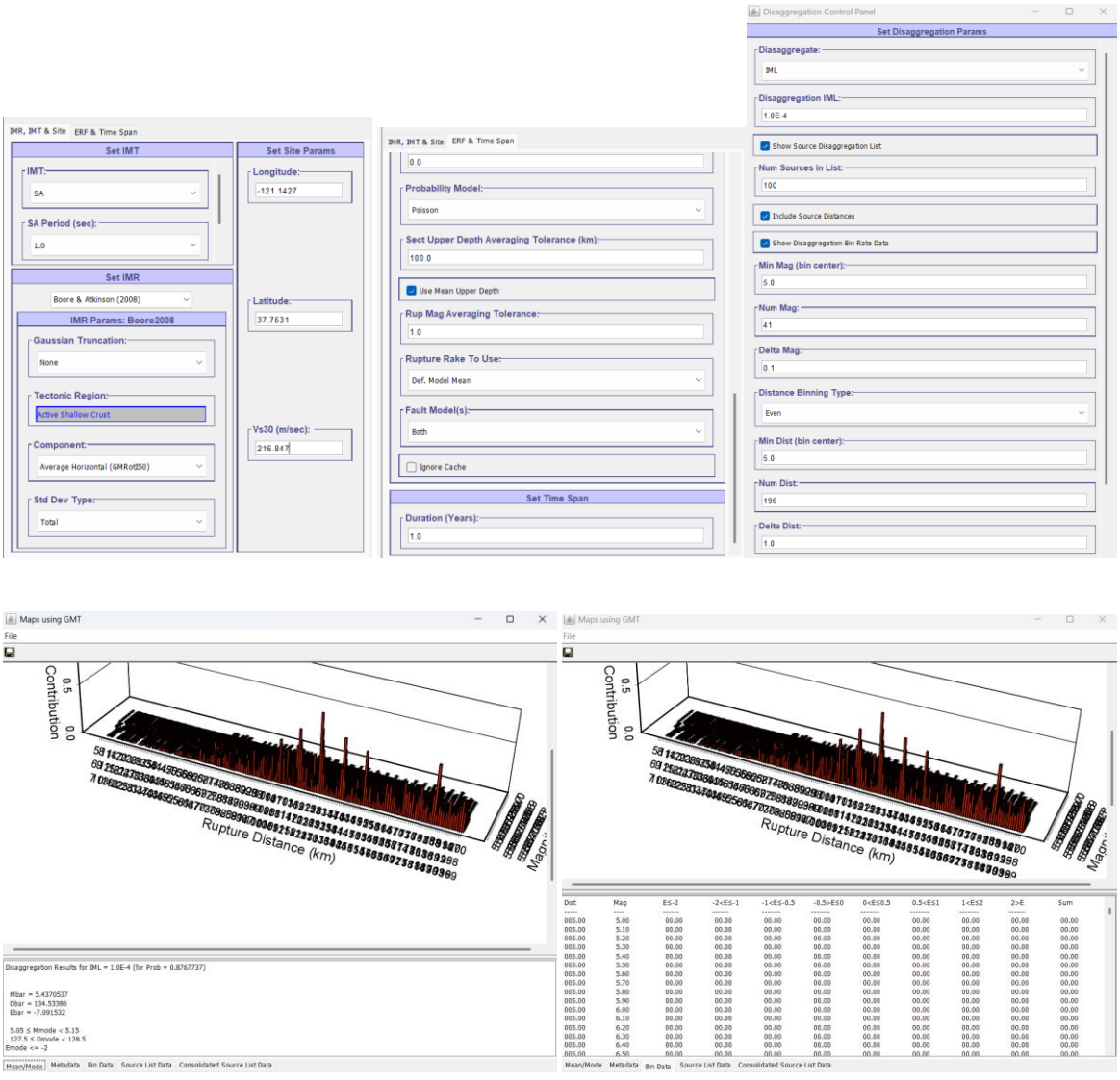
`<model_files_path>/Site_Hazard/deagg_summary_small_sa.txt`

2. Bin Data tab
    - Save the contents to a file named `deagg_data_small_sa.txt`

      Location:

      `<model_files_path>/Site_Hazard/deagg_data_small_sa.txt`

The following figure shows the console inputs and the information to be saved for OpenSHA's Hazard Curve Calculator:



After this step, the directory structure should include:

```
PyPBEE/scripts/Bridge_Column/
├── ops_model.py
├── model_info.py
```

```
├── Site_Hazard/
│       ├── deagg_data_small_sa.txt      # from OpenSHA Hazard application
│       ├── deagg_summary_small_sa.txt   # from OpenSHA Hazard application
```

These two files constitute the minimum PSHA input required by PyPBEE. The PSHA object reads this information to construct scenario rates and thereby compute seismic hazard integrals.

Failure to generate or correctly place these files will prevent the PSHA stage from running correctly. This OpenSHA setup must therefore be completed before executing the PyPBEE analysis driver.

5.2.2 Running PSHA

The probabilistic seismic hazard analysis stage is executed using the same standardized three-step lifecycle:

- `setup(...)` prepares the PSHA stage by expanding the iterator space based on `analysis_case` and `design_num_list`, initializing realization-specific working directories, and registering the required site hazard inputs (i.e., the OpenSHA-derived disaggregation files) for each design point.
- `run(...)` performs the seismic hazard computations for the stage. In the current PyPBEE implementation, this involves reading the precomputed OpenSHA disaggregation outputs, constructing the discrete scenario-rate representation required by the framework, and performing numerical seismic hazard integral computation. The computations are parallelized using the specified `pool_size`.
- `wrap_up(...)` finalizes the PSHA stage by completing bookkeeping tasks, writing processed hazard information to disk, and ensuring that all hazard-related artifacts required by ground motion selection and subsequent analyses are available and consistently organized.

Together, this block enforces a clean separation between stage preparation, parallel hazard processing, and post-processing, ensuring that the seismic hazard inputs to the PBEE workflow are generated in a reproducible and modular manner before proceeding to ground motion selection.

5.3 GMS → NLTHA → PSDemHA → PSDamHA

5.3.1 GMS

After completion of the PSHA stage, all remaining analysis stages follow the same standardized lifecycle consisting of `setup → run → wrap_up`. This uniform interface is a core design feature of PyPBEE and ensures consistency, modularity, and clear separation of responsibilities across analysis stages.

```
gms.setup(analysis_case, design_num_list, haz_lev_list, mrp_list, n_gm_list)
gms.run(analysis_case, pool_size, rng_seed=rng_seed)
gms.wrap_up(analysis_case)
```

The GMS stage is first prepared using `setup(...)`, which expands the iterator space over design points, hazard levels (each hazard level is associated with its target mean return period), and number of ground motions per hazard level, and prepares realization-specific directories.

The `run(...)` method performs the actual selection of ground motion suites, using the hazard information generated by PSHA and the specified random seed for reproducibility. The current approach implemented for `IM SaT` is the Conditional Mean Spectrum approach.

Finally, `wrap_up(...)` finalizes the stage by organizing selected motions and ensuring that all metadata required for NLTHA is available.

5.3.2 NLTHA

```
nltha.setup(
    analysis_case, pool_size, design_num_list, haz_lev_list,
    gm_database_directory_path
)
nltha.run(analysis_case, pool_size)
nltha.wrap_up(analysis_case)
```

The NLTHA stage is then executed. During `setup(...)`, PyPBEE prepares the analysis jobs by associating selected ground motions with the corresponding structural realizations.

The `run(...)` method executes the OpenSees analyses in parallel using the specified `pool_size`.

The initial `wrap_up(...)` performs bookkeeping and records convergence status for each analysis.

During the NLTHA stage, PyPBEE calls a user-provided function named `analysis_time_hist(model_input_dict)` to execute the nonlinear time history analysis for a single realization and ground motion. This function must be implemented by the user in ops_model.py, alongside `main()`, `analysis_prelim()`, and `analysis_gravity()`.

`analysis_time_hist` is responsible for:

- reading the ground motion files,
- defining the dynamic excitation (e.g., uniform excitation in one or more directions),
- configuring the transient analysis parameters and numerical solution strategy,
- running the transient analysis with appropriate fallback strategies for convergence,
- and returning a Boolean flag indicating whether the analysis completed successfully.

PyPBEE does not impose a fixed implementation of this function. Instead, it provides the surrounding execution logic (parallelization, retries, EDP extraction, and bookkeeping), while the numerical details of the time history analysis are fully controlled by the user. This design allows advanced users to customize integration schemes, convergence tests, solver algorithms, and ground motion handling.

An example implementation of `analysis_time_hist` is provided in the Bridge Column example `ops_model.py`. Users are strongly encouraged to start from this example and adapt it to their own structural models and numerical requirements, while preserving the required function signature and return behavior.

If some NLTHA realizations do not converge, the workflow allows the user to modify numerical settings (for example solver algorithm or tolerances) and re-run the NLTHA stage without repeating earlier stages. This second `run` → `wrap_up` cycle attempts to complete the remaining unconverged analyses.

```
nltha.run(analysis_case, pool_size)
nltha.wrap_up(analysis_case)
# After all analyses converges:
nltha.collect_edps(analysis_case, pool_size)
```

Once all NLTHA analyses have converged, `collect_edps(...)` is called to extract and aggregate the engineering demand parameters defined by the EDP objects. This step consolidates recorder outputs into EDP realizations that will be used by downstream demand and damage hazard analyses.

5.3.3 PSDemHA

```
psdemha.setup(analysis_case, design_num_list, haz_lev_list)
psdemha.run(analysis_case, pool_size)
psdemha.wrap_up(analysis_case)
```

The probabilistic seismic demand hazard analysis (`PSDemHA`) stage follows the same lifecycle. The `setup` phase prepares demand modeling tasks for each design point and hazard level. The `run` phase fits demand models and evaluates demand hazard, and the `wrap-up` phase finalizes outputs required by damage hazard calculations.

5.3.4 PSDamHA

```
psdamha.setup(
    analysis_case, design_num_list, haz_lev_list, n_gm_list,
    rng_seed=rng_seed
)
psdamha.run(analysis_case, pool_size)
psdamha.wrap_up(analysis_case)
```

Finally, the probabilistic seismic damage hazard analysis (`PSDamHA`) stage propagates demand hazard through the defined damage states. Its `setup` phase registers damage states, hazard levels, and sampling requirements. The `run` phase evaluates damage hazard numerically, and the `wrap-up` phase consolidates final damage hazard results.

In summary, all analysis stages in PyPBEE adhere to the same lifecycle structure. This design enables predictable execution, easy debugging, selective re-execution of individual stages (such as `NLTHA`), and clear separation between preparation, computation, and post-processing across the entire PBEE workflow.

---

6 Plotting and visualizing results

PyPBEE provides plotting helpers on the `Structure`, `IM`, `EDP`, and `DS` objects to quickly inspect intermediate and final outputs from each stage of the workflow. Typical visualizations include:

- Seismic hazard curves in terms of the chosen `IM` (for example `SaT`), including annotated values at target return periods.
- Seismic hazard deaggregation (magnitude–distance contribution) at a specified hazard level or MRP.
- Selected ground motion response spectra with probabilistic bounds.
- Conditional demand models p(`EDP` | `IM`) and fitted distribution parameters across `IM`.
- Damage and risk summaries, such as MRP histograms and empirical CDFs derived from damage-state exceedance (for FE Model Parameter Uncertainty `analysis_case = '200'`).

Example calls (as shown in the figure) include:

```
# Plotting results
im = psha.im
edp1 = nltha.edp_list[0]
ds1 = psdamha.ds_list[0]
im.plot_shc(for_which=['100', '1', '1', '1', '3'])
im.plot_seismic_hazard_deagg(mrp=475, for_which=['100', '1', '1', '1', '3'])
im.plot_gm_psa_spectra(for_which=['100', '1', '1', '1', '3'], plot_uhs=True)

edp1.plot_conditional_demand_model_3d(
    for_which=['100', '1', '1', '1'],
    edp_str='col_1_edge_1',
    haz_lev_list=haz_lev_list,
    im=im,
    im_lim=(1e-8, 1.),
    edp_lim=(1e-8, 0.125)
)
```

6.1 The `for_which` parameter

Most plotting functions need to know which specific analysis scenario to read from disk. PyPBEE stores results in a directory structure indexed by the workflow iterators, and `for_which` is the user-facing way to select one combination.

In general, `for_which` is a list of numeric strings that identifies:

1. analysis case number (for example '100', '200', ...)
2. structural design number (for example '1')
3. model option number (the selected model-form combination number)
4. model realization number (random model parameter realization number)
5. hazard level (only for hazard-level dependent plots)
6. Ground motion record number (only for ground-motion–specific operations)

So, for example:

- `for_which=['100','1','1','1','3']` means: analysis case 100, design 1, model option 1, realization 1, hazard level 3.
- Some plots are not hazard-level dependent and therefore use a shorter selector such as `for_which=['100','1','1','1']`.
- Some DS-level plots may only require a subset of identifiers (for example `for_which=['200','1','1']`), depending on what the plotting method needs to locate.

Important: `for_which` is not merely for plotting convenience. It is the explicit handle that maps workflow iterators to the on-disk directory structure and tells PyPBEE which directory to query for results. Users should treat it as part of the naming and organization contract that links workflow iterators to stored outputs.

For example, the final level, the ground motion record number, is particularly useful for detailed NLTHA inspection. It allows the user to isolate and investigate a single nonlinear time history analysis run. For example:

```
nltha.extract_single_run(
    for_which=['100', '1', '1', '1', '3', '7'],
    num_modes=prelim_analysis.num_modes,
    gm_database_dir_path=gm_database_dir_path
)
```

This extracts the complete, self-contained model and analysis setup for the specified run, including the realized model parameters, selected model attributes, damping configuration, and fully defined EDP recorders. All files are written into a directory named `<structure.name>_extract_<structural_analysis_platform_name>` located at the same directory level as the model definition folder (i.e., as a sibling directory to the folder represented by `structure.model_files_path`). These extracted files can be executed independently, enabling detailed debugging, verification, or post-processing outside the full PyPBEE workflow.

---

7 Public Interface for PyPBEE Entity Classes

PyPBEE is intentionally designed as a code-first framework. Users are expected to implement concrete subclasses of a small set of core abstract base classes to define their structural system, intensity measures, engineering demand parameters, and damage states. This design enforces clarity, extensibility, and explicit control over all modeling assumptions.

The four fundamental entity classes are:

- `Structure`
- `IM` (Intensity Measure)
- `EDP` (Engineering Demand Parameter)
- `DS` (Damage State)

Each class exposes a public interface that PyPBEE relies on internally. Users implement concrete subclasses by overriding specific methods that correspond to well-defined physical or computational tasks. PyPBEE then orchestrates these objects through the PBEE workflow without attempting to infer or modify user logic.

7.1 Structure

The Structure class represents the physical system and its uncertainty model. A concrete Structure subclass is responsible for:

- defining design points and model attributes,
- generating and retrieving random model parameter realizations,
- managing site hazard information,
- writing model parameters and attributes to analysis directories,
- and exposing structural properties (periods, damping, element connectivity) to other entities.

The provided OSB class is a concrete implementation tailored to bridge systems, extending `FrameStructure`. It adds bridge-specific logic such as identifying the first transverse mode and specialized damping definitions.

Key responsibilities exposed through the public interface include:

- `get_design_num_list`, `get_design_pts`
- `generate_random_model_param_vals`, `get_random_model_param_vals`
- `set_site_hazard_info`, `get_site_hazard_info`
- `write_model_param_vals`, `write_model_attributes`
- `get_periods`, `get_period_range`, `get_rayleigh_damping_params`

7.2 IM

The `IM` class defines how seismic hazard is measured and how hazard information is propagated through the workflow. An `IM` object defines the intensity-measure used for seismic hazard, disaggregation, and record selection. Users implement concrete `IM`s by specifying:

- how the ground motion model is evaluated,
- how hazard integrals are computed,
- how ground motions are selected,
- and how conditional or unconditional spectra are constructed.

The abstract `IM` interface enforces implementation of:

- `evaluate_gmm`
- `select_ground_motion_records`
- `compute_seismic_hazard_integral`

Concrete subclasses such as `SaT` (spectral acceleration at a single period) and `AvgSa` (average spectral acceleration over a period range) demonstrate how different `IM` definitions can reuse the same framework while changing only the physics.

The `IM` public interface also exposes plotting and post-processing utilities, including:

- hazard curves,
- hazard deaggregation,
- uniform hazard spectra,
- conditional spectra,
- and selected ground motion spectra.

These functions all rely on a consistent `for_which` selector to locate results.

7.3 EDP

An EDP object defines a scalar response quantity that PyPBEE will extract from NLTHA results (for example, peak rebar strain, peak drift, etc).

EDP entities serve two roles:

1. They define what to record and how to compute demand for each NLTHA run.
2. They define how demand is modeled probabilistically for PSDemHA (demand hazard).

Concrete subclasses of EDP must implement three critical methods:

- `generate_recorder(...)`
  Defines how OpenSees recorders are written for this EDP.
- `write_evaluate_edp_and_helpers(...)`
  Defines how recorded responses are reduced to a scalar EDP value.
- `get_edp_strings(...)`
  Defines the structural locations where the EDP is evaluated (for example "`col_1_edge_1`").

The provided `MaxColRebarStrain` class illustrates this pattern clearly: it defines recorder generation for rebar strain fibers, evaluates peak strain, and enumerates EDP locations based on column and edge indices.

The EDP public interface also includes:

- EDP collection across realizations (`collect`)
- demand hazard computation (`compute_demand_hazard_integral`)
- retrieval of EDP values at hazard levels
- extensive plotting utilities for conditional demand models

EDPs are the bridge between NLTHA and probabilistic demand modeling, and their implementation is necessarily problem-specific.

7.4 DS

A DS object maps an EDP into a damage/capacity model and ultimately produces damage hazard using demand hazard (from PSDemHA) and capacity uncertainty. A concrete DS object combines:

- a governing EDP,
- a predictive capacity model (`lambda x: ...`),
- and a normalized fragility distribution.

The public interface exposes methods to:

- read predictor information written by the structural model,
- denormalize fragilities using realization-specific predictive capacities,

- compute damage hazard integrals (numerically or via closed-form approximations),
- and aggregate system-level damage metrics.

Crucially, DS relies on explicit naming contracts:

- Predictor files are read as `predictor_info_{ds_type}_{ds_str}.txt`
- `ds_type` is user-defined.
- `ds_str (= edp_str)` is generated by the governing EDP.

7.5 Design philosophy

In PyPBEE:

- Abstract base classes define **what must be implemented**.
- Concrete subclasses define **how it is implemented**.
- The workflow engine defines **when it is executed**.

Users are expected to code in PyPBEE, not merely configure it. This approach enables:

- nonstandard `IM` definitions,
- custom EDP extraction logic,
- advanced damage models,
- and research-grade experimentation with PBEE formulations.

The provided example implementations (OSB, SaT, AvgSa, MaxColRebarStrain, and custom DS definitions) are intended as templates, not constraints. Advanced users are encouraged to extend these patterns while preserving the public interface contracts described above.

In short: **PyPBEE gives you the framework, not the answers.**

8 Using HPC/HTC Resources

TO BE ADDED IN A FUTURE RELEASE

PyPBEE was originally developed and validated with support for HPC execution on the Stampede2 system at the Texas Advanced Computing Center (TACC). Several components of the workflow, including large-scale NLTHA execution and parameter sweeps, were designed with high-throughput and distributed computing in mind.

At present, Stampede2 is no longer available, and as a result the documentation related to HPC and HTC execution has been temporarily deferred. PyPBEE continues to support local and workstation-level multiprocessing, and the core architecture remains compatible with future HPC backends.

The user manual is a work in progress, and we are actively updating and reorganizing the documentation. Guidance on deploying PyPBEE on modern HPC platforms and cloud-based HTC environments will be added in a subsequent release.

9 References

Cornell, C. A., F. Jalayer, R. O. Hamburger, and D. A. Foutch. 2002. “Probabilistic basis for 2000 SAC federal emergency management agency steel moment frame guidelines.” *Journal of structural engineering*, 128 (4): 526–533. ASCE.

Deb, A., J. P. Conte, and J. I. Restrepo. 2022. “Comprehensive treatment of uncertainties in risk-targeted performance-based seismic design and assessment of bridges.” *Earthquake Engineering & Structural Dynamics*, 51 (14): 3272–3295. John Wiley & Sons, Ltd. https://doi.org/https://doi.org/10.1002/eqe.3722.

Deb, A., A. L. Zha, Z. A. Caamaño-Withall, J. P. Conte, and J. I. Restrepo. 2021. “Updated Probabilistic Seismic Performance Assessment Framework for Ordinary Standard Bridges in California.” *Earthquake Engineering & Structural Dynamics*, 50(9): 2551–2570. https://doi.org/https://doi.org/10.1002/eqe.3459.

Vamvatsikos, D. 2013. “Derivation of new SAC/FEMA performance evaluation solutions with second-order hazard approximation.” *Earthquake Engineering & Structural Dynamics*, 42 (8): 1171–1188. Wiley Online Library.