14th International Conference on Applications of Statistics and Probability in Civil Engineering, ICASP14
Dublin, Ireland, July 9-13, 2023

PSimPy: GP emulation-based sensitivity analysis, uncertainty quantification and calibration of landslide simulators

Hu Zhao
Postdoctoral researcher, Methods for Model-based Development in Computational Engineering, RWTH Aachen University, Aachen, Germany

Anil Yildiz
Postdoctoral researcher, Methods for Model-based Development in Computational Engineering, RWTH Aachen University, Aachen, Germany

Nazanin Bagherinejad
Scientific assistant, Methods for Model-based Development in Computational Engineering, RWTH Aachen University, Aachen, Germany

Julia Kowalski
Professor, Methods for Model-based Development in Computational Engineering, RWTH Aachen University, Aachen, Germany

ABSTRACT: Computer simulations are widely used to study real-world systems in many fields of science and engineering, such as earth science, life science, energy engineering, civil engineering, etc. Such simulators may be subject to a variety of uncertainties resulting from many sources, e.g. uncertain input parameters. These uncertainties need to be properly quantified in order to achieve reliable simulation-based prediction and design. However, simulators are often computationally too expensive for uncertainty-related analyses, such as uncertainty quantification, global sensitivity analyses, and parameter calibration. In the recent decades, Gaussian process (GP) emulation has been shown to be effective in overcoming computational bottlenecks. This work presents the newly developed open-source Python package, PSimPy, for sensitivity analysis, uncertainty quantification, and parameter calibration of simulators using GP emulation. It is built upon recent progress in GP emulation for simulators with massive outputs, GP emulation-enabled global sensitivity analyses, and Bayesian active learning for parameter calibration. The structure of PSimPy is presented herein, and case studies from landslide run-out modelling are performed to demonstrate the feasibility of PSimPy for the above mentioned computationally costly tasks. Due to the data-driven nature of GP emulation, PSimPy is potentially applicable to computationally expensive simulators in many fields.

Various fields of science and engineering use computer simulations to describe and study real-world systems and processes. Notable examples include climate science, earth science, life science, energy engineering, civil engineering, and geohazards engineering, etc. Such simulation models, so called simulators, are subject to various uncertainties resulting from a variety of sources, e.g. uncertain input parameters and observation error (Kennedy and O'Hagan, 2001). The presence of uncertainties implies the need of uncertainty-related analyses including sensitivity analysis (Saltelli et al., 2010), uncertainty quantification (Dalbey et al., 2008), and parameter calibration (Allmaras et al., 2013). These analyses are often computationally expensive or even infeasible due to the large number of necessary simulator executions.

One way of overcoming this challenge is to utilize surrogate models that aim at substituting the simulator by a faster to evaluate approximation. Many surrogate modelling methods have been developed over last decades, see Asher et al. (2015) for an overview. Among them, Gaussian process (GP) emulation has been widely used due to its rich theoretical background (Girard et al., 2016) and robustness, e.g. Bounceur et al. (2015) and Sun et al. (2021). Recent progress includes GP emulation for simulators with massive outputs (Gu and Berger, 2016), GP emulation-enabled global sensitivity analyses (Le Gratiet et al., 2014; Zhao et al., 2021), and GP emulation-based Bayesian active learning for parameter calibration (Kandasamy et al., 2017; Wang and Li, 2018; Zhao and Kowalski, 2022). It is now desirable to leverage these new capabilities in a unified framework that can be easily employed by the community to efficiently perform above uncertainty-related computationally expensive analyses.

This work presents the newly developed open-source Python package, PSimPy. It integrates recent progress in the field of Gaussian process emulation and provides a user-friendly toolbox to facilitate uncertainty-related analyses. In section 2, the methods are briefly introduced. In section 3, the structure of PSimPy and its implementation are presented. Section 4 shows how it can be employed using a case study.

1. METHODOLOGY

1.1. Gaussian process emulation

A simulator, $y = f(\mathbf{x})$, defines a mapping of input $\mathbf{x} = (x_1, \ldots, x_p)^T \in \mathscr{X} \subset \mathbb{R}^p$ to output $y \in \mathbb{R}$. Note, that here we consider y as scalar output for simplicity. For each $\mathbf{x}$, a computer simulation needs to be run in order to obtain the corresponding output. As soon as uncertainty-related analyses are conducted, the simulator has to be evaluated many times, which renders a computationally infeasible task even for efficient simulators. GPs are introduced to speed up the evaluation. GP emulation treats the simulator as an unknown function and assumes that its output is unknown at any input before running the simulation. The simulator is then modeled by a Gaussian process

$$f(\cdot) \sim \mathscr{GP}(m(\cdot;\beta), K(\cdot,\cdot;\sigma^2,\gamma)), \tag{1}$$

where $m(\cdot)$ and $K(\cdot,\cdot)$ are the mean and kernel function respectively and β, σ^2, and γ are unknown hyperparameters of the Gaussian process.

From a Bayesian perspective, we can evaluate the simulator at a small number n^{tr} of input points to obtain so called training data (input-output pairs of n^{tr} simulator evaluations). Based on the training data, we can update the above Gaussian process, hence, specify the GP's hyperparameters. The updated Gaussian process provides the Gaussian process emulator which allows us to almost instantly predict output y^* at any new input $\mathbf{x}^*$. The new prediction y^* will be exact whenever the input $\mathbf{x}^*$ corresponds to one of the training data pairs, and otherwise yield an approximation to the simulator. The strength of Gaussian processes furthermore is, that they yield not only the output, but in fact a complete probability distribution, which will later be used for error-control.

There are different ways to learn the unknown hyperparameters β, σ^2, and γ, ranging from non-Bayesian methods to fully Bayesian methods, see Zhao (2021). PSimPy adopts the work of Gu et al. (2018) and Gu et al. (2019), which is a robust partial Bayesian method. For cases where the simulator output is not a scalar but high-dimensional, PSimPy relies on the parallel partial GP emulator developed by Gu and Berger (2016).

1.2. Sensitivity analysis

Given a simulator $y = f(\mathbf{x})$, the aim of a sensitivity analysis is to find out how sensitive the output varies with changing input. The Sobol' sensitivity analysis is a type of global sensitivity analysis method. It decomposes the overall variance of y into contributions resulting from the individual input parameters $x_i, i = 1, \ldots, p$ alone, as well as their interactions. The independent contribution of x_i is represented by its first-order Sobol' index

$$S_i = \frac{V_{x_i}[E_{\mathbf{x}_{-i}}[y|x_i]]}{V[y]}, \tag{2}$$

and the overall contribution of x_i is represented by its total-order Sobol' index

$$S_{T_i} = 1 - \frac{V_{\mathbf{x}_{-i}}[E_{x_i}[y|\mathbf{x}_{-i}]]}{V[y]}, \tag{3}$$

where V denotes the variance operator and E denotes the expectation operator; $\mathbf{x}_{-i}$ represents the collection of all input factors except x_i.

Equations (2)-(3) include tedious integrals, hence, are impossible to be solved analytically if the simulator is complex, which is often the case for real-world processes. PSimPy adopts a numerical method proposed by Saltelli et al. (2010). It requires $n_{base} \cdot (p+2)$ simulator executions where n_{base} denotes the base sample size. This is computationally expensive since n_{base} must be sufficiently large. Le Gratiet et al. (2014) proposed a GP emulation-based Sobol' analysis method which overcomes the computational bottleneck. It provides a means to account for additional uncertainty introduced by GP emulation. Zhao et al. (2021) extended the method to simulators with high-dimensional output by leveraging the parallel partial GP emulator. PSimPy adopts these recently developed methods.

1.3. Uncertainty quantification

The aim of a uncertainty quantification in a narrow view is to quantify the uncertainty in simulator output y induced by its uncertain input $\mathbf{x}$ using relevant statistics such as the mean

$$\mu_y = E[y] \tag{4}$$

and the standard deviation

$$\sigma_y = \sqrt{E[(y-\mu_y)^2]}. \tag{5}$$

Again, equations (4)-(5) are analytically impractical if the simulator is complex. In that case, different flavours of Monte Carlo methods are widely used to numerically approximate the statistics. They require evaluating the simulator at a large number of randomly picked input points and then compute the statistics based on corresponding simulation output values. To improve the computational efficiency, one can first build a GP emulator to substitute for the simulator and then perform the Monte Carlo approximation using the emulator.

1.4. Parameter calibration

In some fields, simulators are used in an inverse manner to learn unknown parameters $\mathbf{x}$ based on observation data d of concerned output y, known as parameter calibration or inference. Following the Bayes' theorem, a posterior probability distribution of the unknown parameters $\mathbf{x}$ can be derived based on prior knowledge of $\mathbf{x}$ and d, namely

$$p(\mathbf{x} \mid d) = \frac{L(\mathbf{x} \mid d)p(\mathbf{x})}{\int L(\mathbf{x} \mid d)p(\mathbf{x})d\mathbf{x}}, \tag{6}$$

where $p(\mathbf{x})$, $p(\mathbf{x} \mid d)$, and $L(\mathbf{x} \mid d)$ are prior probability distribution, posterior probability distribution, and likelihood function respectively.

The likelihood function contains the simulator and is needed to evaluate for the posterior. For a complex simulator, the posterior cannot be analytically computed and has to rely on numerical approach. Numerical approximation strategies, such as grid approximation or Markov Chain Monte Carlo (MCMC) methods, require evaluating the likelihood function (thus the simulator) at a large number of $\mathbf{x}$ values and are therefore computationally costly. Kandasamy et al. (2017) and Wang and Li (2018) combined Gaussian process emulation and active learning with Bayesian inference to improve the computational efficiency significantly. PSimPy implements these newly developed methods for parameter calibration.

2. STRUCTURE OF PSIMPY

PSimPy, standing for Predictive and Probabilistic Simulation with Python, is an open-source Python package. It implements a GP emulation-based framework to efficiently facilitate uncertainty-related analyses associated with simulators, including sensitivity analysis, uncertainty quantification, and parameter calibration. It is hosted at the GitLab project under the link `https://git-ce.rwth-aachen.de/mbd/psimpy`. The installation is straightforward following the guideline on the GitLab page.

Figure 1 shows the main structure of PSimPy, including modules *simulator*, *emulator*, *sensitivity*, *inference*, and *sampler*. As for *uncertainty quantification* (the dashed box), computing the statistics (section 1.3) can be easily done using functionalities provided by NumPy and therefore does not require a dedicated module. PSimPy is implemented in a modular way. One can conveniently combine different modules to realize a desired uncertainty-analyses workflow. It is also easily extendable. Namely, one can implement new classes or modules, such as other MCMC methods, and use them together with existing functionalities provided by PSimPy.

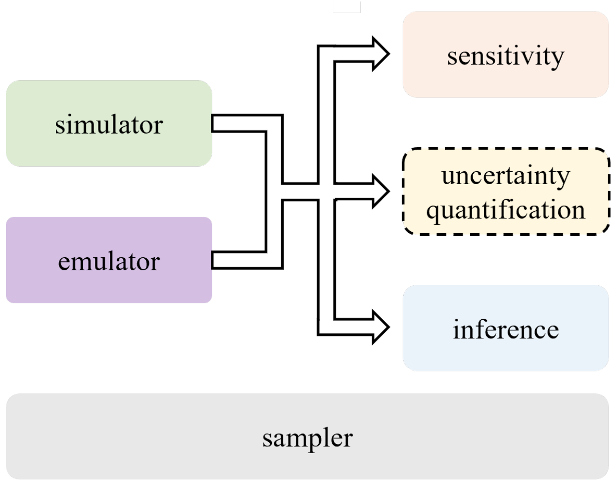

Figure 1: Main structure of PSimPy

2.1. *Sampler*

The *sampler* module hosts sampling methods for emulator training, Sobol' sensitivity analysis, and inference (parameter calibration). The following classes are implemented:

- `psimpy.sampler.LHS`: Latin hypercube sampling. It is a commonly used space-filling sampling method to draw training input points for Gaussian process emulation.
- `psimpy.sampler.Saltelli`: Saltelli's version of Sobol' sampling for Sobol' sensitivity analysis (Saltelli et al., 2010). This class is built upon Saltelli sampling implemented in the Sensitivity Analysis Library in Python (SALib) (Herman and Usher, 2017).
- `psimpy.sampler.MetropolisHastings`: Metropolis Hastings sampling. It is a straightforward MCMC method to numerically approximate the posterior in parameter calibration (section 1.4).

2.2. *Simulator*

The *simulator* module hosts functionality for running simulators. Users can implement simulators of their interest and use functionality provided by `psimpy.simulator.RunSimulator` to run multiple simulations either serially or in parallel. Two simulators frequently used in the field of landslide run-out simulation are also included in the *simulator* module as demonstrators. The following classes are currently available:

- `psimpy.simulator.RunSimulator`: Serial and parallel execution of simulators. The parallel execution relies on `ProcessPoolExecutor` class from Python `concurrent.futures` module.
- `psimpy.simulator.MassPointModel`: Mass point model for landslide run-out simulation.
- `psimpy.simulator.Ravaflow24Mixture`: Voellmy-type shallow flow model for landslide run-out simulation. This class is built upon the GIS-based open source mass flow simulation tool r.avaflow (Mergili et al., 2017).

2.3. *Emulator*

The *emulator* module hosts functionality for emulation methods. Currently implemented classes are:

- `psimpy.emulator.ScalarGaSP`: GP emulation for single-output simulators. Parameters of GP emulator can be robustly estimated following Gu et al. (2018).
- `psimpy.emulator.PPGaSP`: GP emulation for multi-output simulators. It is based on the parallel partial Gaussian process emulation developed by Gu and Berger (2016).

The implementation of classes `ScalarGaSP` and `PPGaSP` relies on the R package RobustGaSP (Gu et al., 2019) and the package rpy2 (Interface to use R from Python). `ScalarGaSP` and `PPGaSP` provide a Python interface to directly use RobustGaSP from within Python.

2.4. *Sensitivity*

The *sensitivity* module hosts functionality for computing sensitivity indices. Currently implemented class is:

- `psimpy.sensitivity.SobolAnalyze`: Compute Sobol' indices. This class is based on Sobol' analysis implemented in SALib.

2.5. *Inference*

The *inference* module hosts functionality for parameter calibration. It contains two Bayesian inference methods, namely grid estimation and Metropolis Hastings estimation. Moreover, active learning is implemented which can be combined with the Bayesian inference methods to realize Bayesian active learning methods (Kandasamy et al., 2017; Wang and Li, 2018). Currently available classes are:

- `psimpy.inference.GridEstimation`: Grid estimation to numerically approximate the posterior distribution. The denominator in equation (6) is estimated by numerical integration on a regular grid. This method is only suitable for low-dimensional problems.
- `psimpy.inference.MetropolisHastingsEstimation`: Metropolis Hastings estimation to numerically approximate the posterior distribution. The posterior is estimated by samples drawn from the unnormalized posterior (numerator of equation 6) using Metropolis Hastings sampling.
- `psimpy.inference.ActiveLearning`: Actively pick training input points to construct a GP emulator for the unnormalized posterior.

3. USAGE EXAMPLE

To demonstrate how PSimPy is used for uncertainty-related analyses, this section shows an example in the field of landslide run-out assessment. We assume a Voellmy-type shallow flow process model (Christen et al., 2010; Mergili et al., 2017). It is governed by balance laws for mass and momentum that describe flow dynamics (i.e., flow height and velocity) of triggered mass along a topography given initial mass distribution. Examples below are based on the 2017 Bondo landslide event (Zhao et al., 2021), see figure 2 for its topography and initial mass distribution.

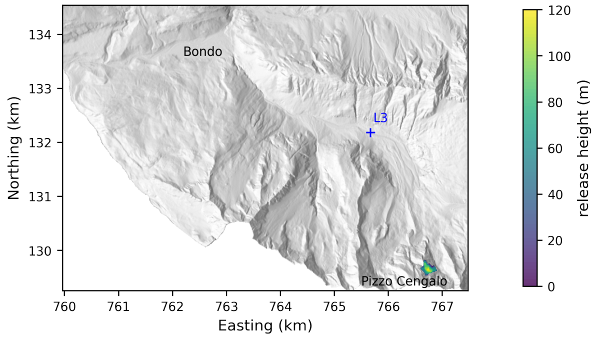

Figure 2: Topography and initial mass distribution of the 2017 Bondo landslide event.

3.1. *Sensitivity analysis and uncertainty quantification*

To set up the sensitivity analysis and uncertainty quantification example, consider the simulator $y = f(\mathbf{x}) = f(\mu, \xi, v_0)$, where y represents the output of interest, here the impact area. μ, ξ, and v_0 are Coulomb friction coefficient, turbulent friction coefficient, and release volume respectively. They are treated as uncertain input parameters. Ranges of μ, ξ, and v_0 are set as 0.02–0.3, 100–2200 m/s^2, and 1.5–4.5 million m^3 following Zhao et al. (2021).

A Sobol' sensitivity analysis assesses the contribution of each uncertain factor to the variation of the impact area (section 1.2). Listing 1 shows how a GP emulation-based Sobol' analysis may be performed using PSimPy:

```python
import numpy as np
from psimpy.sampler import LHS, Saltelli
from psimpy.simulator import RunSimulator
from psimpy.simulator import Ravaflow24Mixture
from psimpy.emulator import ScalarGaSP
from psimpy.sensitivity import SobolAnalyze

def voellmy_model(mu, xi, v0, ...):
    """Define Voellmy model based on Ravaflow24Mixture."""
    ...
    return impact_area

ndim = 3 # number of uncertain parameters
bounds = np.array([[0.02,0.3],[100,2200],[1.5,4.5]])
```

```
15 # Draw training input points using
     Latin hypercube sampling
16 lhs_sampler = LHS(ndim, bounds, ...)
17 lhs_samples = lhs_sampler.sample(
     nsamples=200)
18 # Run simulations at training input
     points and extract outputs
19 run_model = RunSimulator(simulator=
     voellmy_model, ...)
20 run_model.parallel_run(lhs_samples,
     ...)
21 impact_areas = np.array(run_model.
     outputs)
22 # Build GP emulator
23 emulator = ScalarGaSP(ndim, ...)
24 emulator.train(design=lhs_samples,
     response=impact_areas, ...)
25 # Draw samples for Sobol' analysis
26 saltelli_sampler = Saltelli(ndim,
     bounds, ...)
27 saltelli_samples = saltelli_sampler.
     sample(nbase=6000)
28 # Draw realizations of the simulator
     at saltelli_samples using the
     trained emulator
29 Y = emulator.sample(saltelli_samples,
     nsamples=50, ...)
30 # Perform Sobol' analysis
31 analyzer = SobolAnalyze(ndim, Y, ...)
32 sobol_indices = analyzer.run(...)
```

Listing 1: Code snippet of a GP emulation-based Sobol' analysis using PSimPy

In listing 1, lines 8–11 define the Voellmy type shallow flow model based on `psimpy.simulator.Ravaflow24Mixture`. It takes the μ, ξ, and v_0 triple as input and returns the simulated impact area. The rest is self-explanatory.

Figure 3 shows results of the Sobol' analysis. The workflow in listing 1 can be easily extended to multi-output simulators by replacing `ScalarGaSP` with `PPGaSP`, see Zhao et al. (2021) for how the results look like.

For GP emulation-based uncertainty quantification, the workflow is very similar. For example, to quantify the impact of uncertainties of the three unknown parameters on the impact area, one may replace lines 26–28 with a Monte Carlo sampling and lines 31–32 with NumPy commands to compute the statistics.

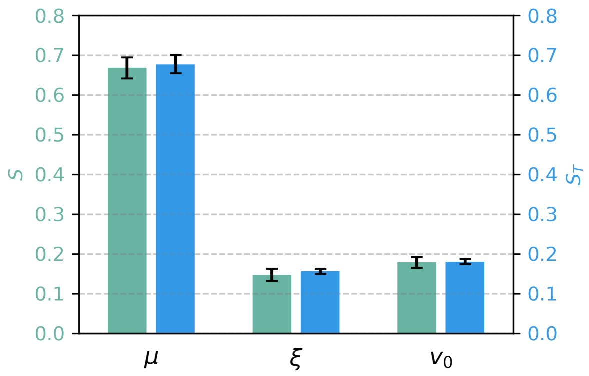

Figure 3: Sobol' indices for the impact area regarding three uncertain parameters μ, ξ, v_0. The green and blue bars represent first-order and total-effect Sobol' indices respectively. The Coulomb friction coefficient μ clearly contributes the most to the variation of the impact area.

3.2. *Inference*

The example of parameter calibration follows the setting of Zhao and Kowalski (2022). Namely, we consider the simulator $y = f(\mathbf{x}) = f(\mu, \xi)$, where y represents maximum flow velocity at location L3 (figure 2). In the context of parameter calibration, the aim is to use observed data to update our knowledge of unknown parameters using Bayes' theorem (section 1.4. Here, we would like to learn about Coulomb friction coefficient μ and turbulent friction coefficient ξ based on an (synthetic) observation of maximum flow velocity at L3 via Bayesian active learning. Listing 2 presents how such an analysis may be conducted using PSimPy:

```
import numpy as np
from psimpy.sampler import LHS
from psimpy.simulator import
    RunSimulator
from psimpy.simulator import
    Ravaflow24Mixture
from psimpy.emulator import ScalarGaSP
from psimpy.inference import
    ActiveLearning
from psimpy.inference import
    GridEstimation

def voellmy_model(mu, xi, ...):
    """Define Voellmy model based on
    Ravaflow24Mixture."""
    ...
    return maxv_L3
```

```python
14 # number of uncertain parameters which need to be calibrated
15 ndim = 2
16 bounds = np.array([[0.02,0.3],[100,2200]])
17 # define a Latin hypercube sampler to draw initial training input points
18 lhs_sampler = LHS(ndim, bounds, ...)
19 # create a RunSimulator object to run simulator
20 run_model = RunSimulator(simulator=voellmy_model, ...)
21 # create a ScalarGaSP object to train emulator
22 scalar_gasp = ScalarGaSP(ndim, ...)
23 # observed data for parameter calibration
24 data = ...
25 # define prior probability distribution
26 def prior(...):
27     ...
28     return prior_probability_density_value
29 # define likelihood function
30 def likelihood(...):
31     ...
32     return likelihood_value
33 # create an ActiveLearning object to actively train a GP emulator for the unnormalized posterior
34 active_learner = ActiveLearning(ndim, bounds, data, run_model, prior, likelihood, lhs_sampler, scalar_gasp, ...)
35 n0 = 40 # number of initial simulations
36 niter = 80 # number of iterative simulations
37 # run initial simulations
38 init_var_samples, init_sim_outputs = active_learner.initial_simulation(n0, ...)
39 # run iterative simulations and sequentially build and improve emulator
40 var_samples, sim_outputs, ln_pxl_values = active_learner.iterative_emulation(n0, init_var_samples, init_sim_outputs, niter, ...)
41 # use grid estimation to approximate the posterior based on the final GP emulator
42 grid_estimator = GridEstimation(ndim, bounds, ln_pxl=active_learner.approx_ln_pxl)
43 posterior, _ = grid_estimator.run(nbins=100)
```

Listing 2: Code snippet of a parameter calibration using PSimPy

In listing 2, lines 9–33 define required inputs to create an `ActiveLearning` object. Once it is created, we can call its `initial_simulation` method to prepare initial training data (line 38) and then call its `iterative_emulation` method to iteratively build the GP emulator for the unnormalized posterior (line 40). In lines 42–43, we use grid estimation to approximate the posterior based on the final GP emulator . Figure 4 shows the results.

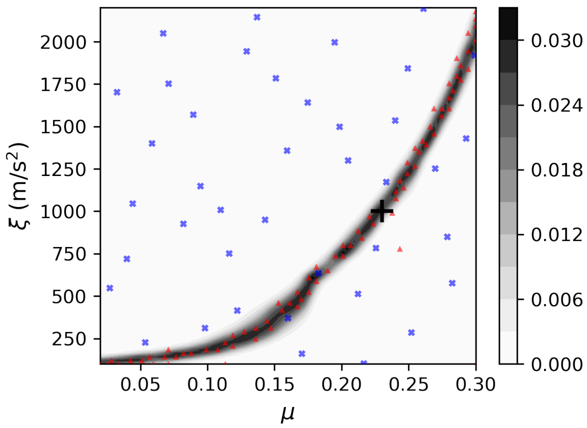

Figure 4: Bayesian active learning for calibrating μ and ξ based on an (synthetic) observation of maximum flow velocity at L3. The black cross represents the underlying truth values of μ and ξ and the colormap shows the estimated posterior. Blue asterisk denotes the initial training input points and red triangle denotes actively picked training input points. The resulting banana shaped posterior distribution is well known for this type of shallow flow based landslide models.

4. Conclusion

This paper presents the recently developed open-source Python package, PSimPy, for sensitivity analysis, uncertainty quantification, and parameter calibration of simulators using GP emulation. It takes advantage of recent GP emulation for simulators with massive outputs, GP emulation-enabled global sensitivity analyses, and Bayesian active learning for parameter calibration. PSimPy is implemented in a highly modular way which means different modules can be easily combined to realize desired workflows, as shown by the usage examples. The modular characteristic also makes it easily extendable. Users can simply implement their own classes or modules and combine them with existing building blocks of PSimPy. To obtain up-to-date information about the package, please refer to our GitLab project at `https://git-ce.rwth-aachen.de/mbd/psimpy`.

5. REFERENCES

Allmaras, M., Bangerth, W., Linhart, J. M., Polanco, J., Wang, F., Wang, K., Webster, J., and Zedler, S. (2013). "Estimating parameters in physical models through Bayesian inversion: A complete example." *SIAM Review*, 55(1), 149–167.

Asher, M. J., Croke, B. F. W., Jakeman, A. J., and Peeters, L. J. M. (2015). "A review of surrogate models and their application to groundwater modeling." *Water Resources Research*, 51(8), 5957–5973.

Bounceur, N., Crucifix, M., and Wilkinson, R. D. (2015). "Global sensitivity analysis of the climate–vegetation system to astronomical forcing: an emulator-based approach." *Earth System Dynamics*, 6(1), 205–224.

Christen, M., Kowalski, J., and Bartelt, P. (2010). "RAMMS: Numerical simulation of dense snow avalanches in three-dimensional terrain." *Cold Regions Science and Technology*, 63(1), 1–14.

Dalbey, K., Patra, A. K., Pitman, E. B., Bursik, M. I., and Sheridan, M. F. (2008). "Input uncertainty propagation methods and hazard mapping of geophysical mass flows." *Journal of Geophysical Research: Solid Earth*, 113(B5), B05203.

Girard, S., Mallet, V., Korsakissok, I., and Mathieu, A. (2016). "Emulation and Sobol' sensitivity analysis of an atmospheric dispersion model applied to the Fukushima nuclear accident." *Journal of Geophysical Research: Atmospheres*, 121(7), 3484–3496.

Gu, M. and Berger, J. O. (2016). "Parallel partial Gaussian process emulation for computer models with massive output." *Annals of Applied Statistics*, 10(3), 1317–1347.

Gu, M. Y., Palomo, J., and Berger, J. O. (2019). "Robustgasp: Robust Gaussian stochastic process emulation in R." *The R Journal*, 11(1), 112–136.

Gu, M. Y., Wang, X. J., and Berger, J. O. (2018). "Robust Gaussian stochastic process emulation." *Annals of Statistics*, 46(6A), 3038–3066.

Herman, J. and Usher, W. (2017). "SALib: An open-source Python library for sensitivity analysis." *The Journal of Open Source Software*, 2(9).

Kandasamy, K., Schneider, J., and Póczos, B. (2017). "Query efficient posterior estimation in scientific experiments via Bayesian active learning." *Artificial Intelligence*, 243, 45–56.

Kennedy, M. C. and O'Hagan, A. (2001). "Bayesian calibration of computer models." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(3), 425–464.

Le Gratiet, L., Cannamela, C., and Iooss, B. (2014). "A bayesian approach for global sensitivity analysis of (multifidelity) computer codes." *SIAM/ASA Journal on Uncertainty Quantification*, 2(1), 336–363.

Mergili, M., Fischer, J. T., Krenn, J., and Pudasaini, S. P. (2017). "r.avaflow v1, an advanced open-source computational framework for the propagation and interaction of two-phase mass flows." *Geoscientific Model Development*, 10(2), 553–569.

Saltelli, A., Annoni, P., Azzini, I., Campolongo, F., Ratto, M., and Tarantola, S. (2010). "Variance based sensitivity analysis of model output. Design and estimator for the total sensitivity index." *Computer Physics Communications*, 181(2), 259–270.

Sun, X. P., Zeng, P., Li, T. B., Wang, S., Jimenez, R., Feng, X. D., and Xu, Q. (2021). "From probabilistic back analyses to probabilistic run-out predictions of landslides: A case study of Heifangtai terrace, Gansu Province, China." *Engineering Geology*, 280, 105950.

Wang, H. Q. and Li, J. L. (2018). "Adaptive Gaussian process approximation for Bayesian inference with expensive likelihood functions." *Neural Computation*, 30(11), 3072–3094.

Zhao, H. (2021). "Gaussian processes for sensitivity analysis, Bayesian inference, and uncertainty quantification in landslide research." Ph.D. thesis, RWTH Aachen University, Aachen.

Zhao, H., Amann, F., and Kowalski, J. (2021). "Emulator-based global sensitivity analysis for flow-like landslide run-out models." *Landslides*, 18, 3299–3314.

Zhao, H. and Kowalski, J. (2022). "Bayesian active learning for parameter calibration of landslide run-out models." *Landslides*, 19, 2033–2045.