Computer Physics Communications 283 (2023) 108596

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Feature article

FabSim3: An automation toolkit for verified simulations using high performance computing ☆

Derek Groen [a,d,*], Hamid Arabnejad [a], Diana Suleimenova [a], Wouter Edeling [b], Erwan Raffin [c], Yani Xue [a], Kevin Bronik [d], Nicolas Monnier [c], Peter V. Coveney [d,e,f]

[a] Department of Computer Science, Brunel University London, UK
[b] Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands
[c] CEPP – Center for Excellence in Performance Programming, Atos, Rennes, France
[d] Department of Chemistry, University College London, UK
[e] Advanced Research Computing, University College London, UK
[f] Instituut voor Informatica, Universiteit van Amsterdam, the Netherlands

ARTICLE INFO

Article history:
Received 17 June 2022
Received in revised form 3 October 2022
Accepted 11 November 2022
Available online 17 November 2022

Keywords:
Automation
High-performance computing
Simulation
Materials
Migration
Climate
Biomedicine
Multiscale modelling

ABSTRACT

A common feature of computational modelling and simulation research is the need to perform many tasks in complex sequences to achieve a usable result. This will typically involve tasks such as preparing input data, pre-processing, running simulations on a local or remote machine, post-processing, and performing coupling communications, validations and/or optimisations. Tasks like these can involve manual steps which are time and effort intensive, especially when it involves the management of large ensemble runs. Additionally, human errors become more likely and numerous as the research work becomes more complex, increasing the risk of damaging the credibility of simulation results. Automation tools can help ensure the credibility of simulation results by reducing the manual time and effort required to perform these research tasks, by making more rigorous procedures tractable, and by reducing the probability of human error due to a reduced number of manual actions. In addition, efficiency gained through automation can help researchers to perform more research within the budget and effort constraints imposed by their projects.

This paper presents the main software release of FabSim3, and explains how our automation toolkit can improve and simplify a range of tasks for researchers and application developers. FabSim3 helps to prepare, submit, execute, retrieve, and analyze simulation workflows. By providing a suitable level of abstraction, FabSim3 reduces the complexity of setting up and managing a large-scale simulation scenario, while still providing transparent access to the underlying layers for effective debugging. The tool also facilitates job submission and management (including staging and curation of files and environments) for a range of different supercomputing environments. Although FabSim3 itself is application-agnostic, it supports a provably extensible plugin system where users automate simulation and analysis workflows for their own application domains. To highlight this, we briefly describe a selection of these plugins and we demonstrate the efficiency of the toolkit in handling large ensemble workflows.

Program summary

Program Title: FabSim3
CPC Library link to program files: https://doi.org/10.17632/6nfrwy7ptj.1
Licensing provisions: BSD 3-clause
Programming language: Python 3
Nature of problem: Many aspects are crucial for obtaining reproducible and robust simulation results. For instance, we need to curate all the inputs and outputs for later scrutiny, scrutinize the model behaviour under slightly perturbed circumstances, quantify the propagation of key uncertainties from input data and known parameters and analyze the sensitivity for any parameters for which the exact specification eludes us.

☆ The review of this paper was arranged by Prof. N.S. Scott.

* Corresponding author at: Department of Computer Science, Brunel University London, UK.
E-mail address: Derek.Groen@brunel.ac.uk (D. Groen).

https://doi.org/10.1016/j.cpc.2022.108596

Solution method: FabSim3 uses a range of methods to provide automation. These primarily include: (i) SSH + Fabric2 to enable remote execution of SSH commands, (ii) an internal parameter state space using primarily Python dict objects that can be customized with machine- plugin- and user-specific modifications, (iii) Python templating to quickly enable the insertion of state space variables into supercomputing scripts, (iv) multiprocessing and/or QCG-PilotJob to enable efficient submission and execution of job arrays and (v) a system of flexibly installable and modifiable Python3 plugins which allows users to create and customize application-specific functionalities without modifying the core code base. In addition to the written code, FabSim3 also relies on a set of user conventions to maintain a separation of concerns (particularly between machine-, user- and application-specific settings).
Additional comments including restrictions and unusual features: This paper serves as the definitive reference for FabSim3.

1. Introduction

Computational simulation and complex data analysis are essential in many scientific disciplines. They involve researchers going through tasks such as constructing models, preparing data, executing and monitoring simulation runs, fetching simulation output, post-processing results and performing further analysis and visualisation tasks. They often perform these tasks in an iterative manner, redoing previous sets of tasks in a similar (but not always identical) manner to investigate the problem at hand from a different angle, or to move onward to a newly derived problem. These iterative patterns return frequently in computational research, for instance during initial model development, during verification, validation and sensitivity analysis studies, during the debugging of simulation codes or new domain configurations, during the deployment of a code to a new infrastructure, and during the in-depth studies of the main scientific (production) problems. All these cases commonly require a complex combination of the aforementioned tasks, and in particular a sizable number of different simulations runs.

When manually working on these type of iterative use cases, or workflows, the number of human errors tends to scale with the complexity of the task at hand. And because a single mistake can render the full simulation useless, manual management simply becomes both intractable and highly effort-intensive for more complex computational research cases, particularly those involving large ensemble simulations or remote (HPC) resources. Another drawback of manual processes, or even semi-automated ones through individual scripts, is that these approaches are difficult to re-purpose: e.g. for use on new HPC platforms, for studies that use existing solutions in a different context, or for studies that rely on an alternative simulation solver.

We have developed FabSim3 to help computational researchers with these issues, to become more productive in general, and become more effective with handling complex applications. For instance, researchers may want to rerun an application among different platforms, run with different ranges of input parameters, or define new workflow steps for their applications using external libraries. Our aim is to do all these steps in a transparent fashion, so that it is possible for researchers to examine essential aspects of their application in detail, scrutinize individual runs and their environment, debug their application, and repeat or reproduce runs with ease.

In this paper, we will describe FabSim3 and a range of relevant related work in the remainder of this introductory section. In Section 2 we will present the architecture of FabSim3 in greater detail, while in Section 3 we present key performance characteristics of the toolkit. FabSim3 has been used across a wide range of scientific disciplines, and we summarize the key use cases (and their associated plugins) in Section 4. We then conclude the paper in Section 5 and provide some key documentation as part of the Appendices.

1.1. FabSim3 overview

FabSim3 is an automation toolkit that is intended to support computational researchers, particularly those who run simulations, but who also do development on these simulations and their workflows. To accomplish this, our objectives are to:

- Flexibly automate scientific workflow simulation processes to improve development and user productivity. Here “flexibly” means that users can easily modify existing automated tasks for re-use in new contexts. This can involve e.g. (a) preparation, (b) execution and monitoring, and (c) post-processing and analysis,
- Reduce the risk of human error, e.g. by automating data management tasks and the management of ensemble and replicated simulations.
- Make workflows easier to debug by transparently and systematically curating key inputs, outputs, scripts, local and remote environment variables.
- Reduce the required skill level and expertise for users to perform complex simulation workflows, including e.g. coupled or ensemble simulations, by introducing short-hand commands and flexible Python functions.
- Provide fully automated functionalities for important or common workflows, enabling users to perform these from preparation to post-processing and results analysis with a single command.
- Make workflows easier to port to different platforms by systematically organising platform-specific details of each application and workflow, and by pooling user-independent settings in a repository for shared use.
- Improve the reproducibility of research results by providing an organised environment that encapsulates all relevant scripts, configuration details, environment settings and inputs for key studies.

FabSim3 offers Application Programmable Interfaces (APIs) that allow developers to describe their application workflow in a manner that hides low-level details, such as preparation of simulation runs, job submission and management, and file transfer.

FabSim3 also allows developers to define bespoke application workflows through both an extensible plugin architecture and the API. It offers simple functions to launch simulations on remote (HPC) resources: these can be single simulation runs, replicated runs (with each

simulation run being identically configured), or ensembles of runs (with each simulation being configured slightly differently). In terms of HPC access, FabSim3 supports the use of either SSH or GSISSH, and it has working template examples for HPC job schedulers such as SLURM, PBS, QCG, SGE, and LSF. Adding support for a new job scheduler is as simple as modifying one of the existing templates to fit its specification.

To reduce human error and make these complex workflows easier to reproduce, FabSim3 provides an administrative framework for curating job information including inputs, outputs and environmental parameters. FabSim3 also supports a set of Verification and Validation Patterns (VVPs) and Uncertainty Quantification and sensitivity analysis Patterns (UQPs) to facilitate the Verification, Validation and Uncertainty Quantification (VVUQ) of specific applications. All these functionalities and plug-in systems are provided at a high level of abstraction and extensibility. By using the built-in FabSim3 APIs, users can extend the core functionalities by developing their own plug-ins. Moreover, FabSim3 is a community project: machine-specific configuration information is shared and once a new feature or bugfix is finalized by one developer, it can be embedded and used by all others in their own plugins.

FabSim3 is one of the main components in SEAVEAtk (formerly known as the VECMA toolkit), which is an application-agnostic software toolkit for the verification, validation and uncertainty quantification of high-performance computing simulation workflows.[1]

1.2. Related work

FabSim3 relates to a wide range of tools that automate and support complex workflows. Many of these tools are taken up in the multiscale computing community, where the workflows are generally more complex due to the presence of multiple coupled models. A systematic analysis of multiscale computing tools and their added value is provided by Groen et al. [1]. Many of the tools mentioned there relate to FabSim3 in one form or another. In this section we will highlight the most relevant tools.

The Automated Interactive Infrastructure and Database for Computational Science (AiiDA) [2] is perhaps one of the most closely related tools. Similarly to FabSim3, it provides an environment which helps to automate and curate complex workflows. AiiDA differs from FabSim3 in several areas. First, AiiDA is mainly geared towards use in materials science (as evidenced by the plugin list), while FabSim3 has a broad scope across domains. Second, the tool focuses on enabling workflows using directed acyclic graphs, whereas FabSim3 focuses on enabling both acyclic and cyclic workflows. Third, the tool is more extensive, as for example its API also incorporates a wide range of database functionalities. Fourth, FabSim3 enables users to efficient execute large ensembles of jobs through a pilotjob manager, while AiiDA relies solely on submitting jobs directly to supercomputer schedulers, and therefore can only support ensembles of limited size.

MaestroWF [3] is another, more generic tool for creating and managing simulation workflows. Similar to AiiDA, it supports a range of HPC scheduling systems, but does not (to our knowledge) interface to a pilotjob manager. In addition, MaestroWF relies on YML for simulation workflow definitions, while FabSim3 plugins have Python functions that define the workflows. Both approaches have their advantages, with our Python-based plugin approach being more flexible, while a YML-defined simulation workflow as used in MaestroWF is easier to verify syntactically.

Similarly, the A-/B-/O-MUSE frameworks [4,5] provide a multi-purpose simulation workflow environment in respectively the astrophysical, biological and oceanographic domain. No general-purpose implementation currently exists, but the underlying design is adaptable to different domains [5]. AMUSE and its variations provide a wide range of domain-specific added values, but share similarities with FabSim3 in that they also contribute towards making complex simulation workflows easier to execute and modify.

FabSim3 is built on top of Fabric2,[2] a Python library for executing shell commands remotely over SSH, and therefore shares automation and remote access features with Fabric[2] and other system-level automation tools such as Ansible[3] and Homebrew.[4] FabSim3 also has functionalities that enable models to be easily combined to form multi-model simulations, a feature which is common in *coupling tools* such as MUSCLE2 [6] and Oasis-MCT [7]. There are also a number of languages and standards that provide abstractions for remote access in the Cloud, such as TOSCA [8].

2. Architecture of FabSim3

Within this section we present the main architecture of FabSim3. The architecture is modular, and consists of four main parts: (a) base components, (b) backends, (c) plugins and (d) patterns. Fig. 1 represents a high-level architecture overview of FabSim3. Here, many of the application-agnostic core functionalities are provided by the base components. These rely in turn on the backends to provide seamless access and use of remote resources, and on the patterns to provide automated routines for VVUQ. Lastly, the plugins are user-customizable and application-specific components that can exploit the wide range of functionalities offered by the base components, or use the VVP/UQP pattern codes directly.

2.1. Base components

Base components deliver the application-agnostic and platform-agnostic functionalities in FabSim3. These include essential functionalities such as job submission, ensemble execution, file staging or local script execution. The functionalities in the base components are tied to FabSim3 itself, but it does leverage pattern code that is entirely tool-agnostic. These tool-agnostic functionalities are part of the patterns.

Base components rely on the functionalities provided by the Fabric2 library and serve to simplify machine selection, provide essential command-line tools, expose a unified API to external Python scripts, and provide building blocks for job submission and management for the other parts of the toolkit.

[1] https://www.seavea-project.org/seaveatk/.
[2] http://www.fabfile.org.
[3] http://www.ansible.com.
[4] http://brew.sh.

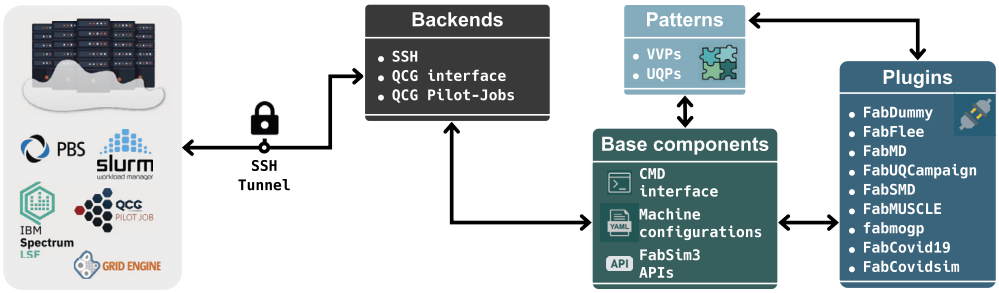


Fig. 1. Essential building blocks within the FabSim3. Arrows indicate direct interactions between components. Patterns do not have dependencies on other components and can be extracted and reused in other tools.

FabSim3/fabsim/deploy/machines.yml

```yaml
eagle_vecma: # machine name
  remote: "eagle.man.poznan.pl"
  home_path_template: "/home/$username"
  budget: "vecma2020"
  corespernode: 28
  partition_name: "standard"
  job_dispatch: "sbatch"
  cancel_job_command: "scancel $jobID"
  job_info_command : "squeue --jobs $jobID"
  stat: "squeue -u $username"
  ...
```

(a) machines.yml

FabSim3/fabsim/deploy/machines_user.yml

```yaml
eagle_vecma: # machine name
  username: "test"
  # format days-hours:minutes:seconds
  job_wall_time : "0-1:40:00"
  partition_name: "fast"
  ...
```

(b) machines_user.yml

FabSim3/fabsim/deploy/templates/slurm-eagle

```
#!/bin/bash

#SBATCH --nodes=$nodes
#SBATCH --tasks-per-node=$corespernode
#SBATCH --time=$job_wall_time
#SBATCH --account=$budget
#SBATCH --output=$job_results/JobID.output
#SBATCH --error=$job_results/JobID.error
#SBATCH --mem=$memory
#SBATCH --partition=$partition_name
```

(c) Sample SLURM based scheduler job script

submitted job script

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --tasks-per-node=28
#SBATCH --time=0-1:40:00
#SBATCH --account=vecma2020
#SBATCH --output=/home/test/JobID.output
#SBATCH --error=/home/test/JobID.error
#SBATCH --mem=6000
#SBATCH --partition=fast
```

(d) generated output script

Fig. 2. Sample remote machine configuration.

2.1.1. Machine configurations

Simulations can be run locally or remotely using HPC resources. A remote run involves tasks such as data staging, job submission, job monitoring and output retrieval on the target machine. To automate these steps, we require some information about the target machine, such as username, host internet address and the job submission command. Within FabSim3, we store machine-specific variables in three human readable YAML files: (1) `machines.yml` file which keeps general configurations which are applicable to all users of each remote machine, (2) `machines_user.yml` file, which contains user-specific information for each machine that is stored locally, and (3) `machines_<plugin_name>_user.yml` file, which stores machine configurations specific to a FabSim3 plugin. For example, the internet address for the ARCHER2 headnode would be stored in (1), the name of the user account on that machine in (2), and the installation directory for the LAMMPS solver in FabMD on the ARCHER2 supercomputer would be placed in (3).

We provide examples of SLURM-based remote machine configuration entries in `machines.yml` and `machines_user.yml` files in Fig. 2 (a) and (b).

2.1.2. Remote applications execution

To submit a job for execution, FabSim3 prepares a platform-specific job script using a two-phase template system, which inserts machine-specific instructions, followed by application-specific instructions (containing machine-specific variables). This script is then submitted to the remote batch system, such as SLURM or PBS. We use a Python3 built-in pyTemplate substitution system[5] to generate the required script for job submission. Similar to machine-specific configurations, we also need a proper template script for job submission for each type of machine and/or batch system. This template script contains scheduler-specific commands for job definition. By using the template system, FabSim3 loads and combines the required variables from the machine and user specific configurations `yaml` files,

[5] https://docs.python.org/3/library/string.html?highlight=template#string.Template.

Table 1
List of tasks commonly used in FabSim3.

Task Name	Brief description & Usage

`list`	prints the list of available tasks or remote machines.
	`>_ fabsim -list:tasks`
	`>_ fabsim -list:machines`

`avail_plugin`	prints the list of available tasks or remote machines. `>_ fabsim localhost avail_plugin`
`machine_config_info`	prints the defaults and user overwritten values of the target host. `>_ fabsim <target_host> machine_config_info`
`stat`	prints information about the submitted and unfinished jobs for user on the target host. `>_ fabsim <target_host> stat`
`cancel_job`	cancels jobs you are running under your account on the target host. `>_ fabsim <target_host> cancel_job:<jobID>`
`setup_ssh_keys`	sets up SSH key pairs for passwordless access to target host. `>_ fabsim <target_host> setup_ssh_keys`
`fetch_results`	fetches results from the target machine to local result store. The results will then be in the `results` folder inside FabSim3 directory. `>_ fabsim <target_host> fetch_results`

and generates the job script for submission to the target machine. This structure allows us to combine templates for executing specific applications with those for a HPC scheduler, without the need to define new templates for each combination of the two. In Fig. 2 (c) we present an example job script template for a SLURM-based remote machine.

FabSim3 has been used for a range of job scheduler including SLURM, PBS, QCG, SGE, and LSF. Support for any new scheduler can be added by adjusting the machine-specific configurations and creating a template file for the job submission script (see Appendix A for more details).

2.1.3. Command-line interface

FabSim3 exposes a range of command-line functionalities through the Fabric2 library.[6] It allows users to define their own FabSim commands as well, mapping Python function directly to the command-line interface. All FabSim3 commands adhere the following structure:

```
> fabsim <machine_name> <task name>[:<arg1=x1>,...,<argN=xN>]
```

When a user invokes a `fabsim` command, (a) the remote machine name, (b) the called task name, and (c) all input arguments for the task (if exists) are extracted from the context of that command. We describe a number of tasks commonly used in FabSim3 in Table 1.

2.1.4. FabSim3 APIs

In addition to one-liner commands, FabSim3 can also be used in Python programs through a Python3 API. The API can be accessed using `from fabsim.lib.fabsim3_cmd_api import fabsim`.

The API allows users to access any FabSim3 command-line task by calling the function `fabsim(command, arguments, machine)`. For instance, to run a dummy job on localhost, one can use

```
fabsim(command="dummy",arguments="dummy_test,cores=1",machine="localhost")
```

Within `fabsim3_cmd_api`, we also provide better-formatted shorthand functions for running ensembles and gathering samples, and we plan to expand the range of shorthand functions as and when the API becomes more widely adopted.

2.1.5. Support for simulation ensembles and replicas

When using simulations for the purpose of in-depth research, a simulation usually needs to be run multiple times to account for aleatory or epistemic uncertainties. In the case of aleatory uncertainty, the individual simulation runs will be configured identically, generating a set of simulation *replicas*. In the case of epistemic uncertainty, or when analysing parameter sensitivity, the individual runs will each have slightly different configuration, and form an *ensemble simulation*.

The preparation of ensemble/replica simulation runs is labour intensive when performed manually, and a simple mistake can render the full computation useless. FabSim3 provides functions to help automate creation, management, and submission of large ensemble/replica runs, allowing users to save time, effort, and reducing the risk for human error. These functions can automatically construct, execute, validate, analyse and curate a number of different models (and model executions) for each computation. FabSim3 also curates all inputs and keeps a copy of the local and remote environment variables, so that failed computations can be investigated and successful ones can more easily be repeated elsewhere.

Fig. 3 demonstrates the organization of files and folders when we call Fabsim3 ensemble/replicas functionalities on the execution side.

2.2. Backends

The Backends consist of a range of modules and interfaces that enable FabSim3 to launch, manage and monitor jobs on computational resources, both locally and remotely. They currently consist of three main functionalities: (1) an interface to use SSH (or GSISSH) to

[6] http://www.fabfile.org.

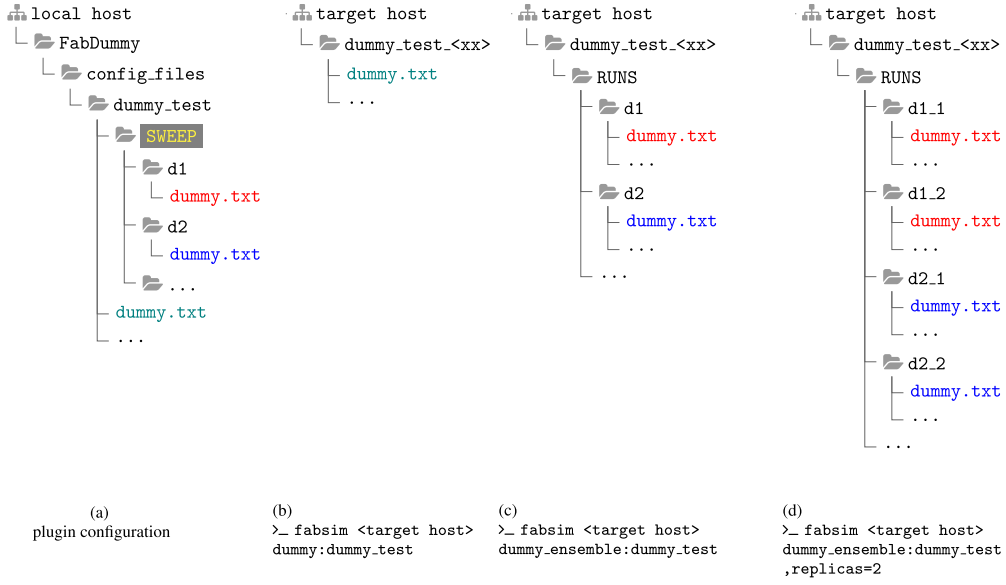


Fig. 3. An example to demonstrate how files and folders are organized and transfered to the target machine for execution with ensemble and replica option. (a): the local configuration, i.e., files and folders, for application (b): the generated files and folder for simple run, without ensemble/replica option (c): the generated files and folder for tow ensembles run of application (d): the generated files and folder for two ensembles and one replica runs of application. Please note that, by default, FabSim3 builds and constructs the required number of ensembles runs based on a default folder, namely `SWEEP`, located inside the application config directory.

execute commands remotely, (2) a QCG Pilot-Job interface to enable the flexible scheduling of large sets of simulation runs within single HPC resource allocations, and (3) a QCG interface to communicate with the cross-resource QCG Broker. We present these functionalities in greater detail below:

2.2.1. SSH and GSISSH interface

With FabSim3, we provide support using either grid certificates (through GSISSH) or using the SSH protocol (using the Paramiko library[7]). We choose these interfaces because they are widely supported standards for HPC access, particularly in the case of SSH. FabSim3 uses the Fabric2 library to allow users to perform remote operations using basic SSH as the transport middleware. As such, FabSim3's security model is essentially the SSH security model; public/private keys are used to authenticate FabSim3 operations on target resources, and the `~/.ssh/known_hosts` file is used to allow users to configure mutual authentication.

Unlike grid based X.509 authentication (which is also supported through GSISSH), the SSH setup is entirely controlled by the end user. Typically, on a grid using X.509 certificates, an administrator must set up details of a user's certificate on their resources while SSH based authentication requires that the user sets up their own keys on a target resource. Although key management does potentially increase the management overhead of setting up FabSim3 security, the ubiquity of SSH means that most FabSim3 users will have already taken steps to set up SSH keys for the resources that they wish to access. Lastly, FabSim3 can be configured with multiplexing to greatly reduce the number of times that users need to provide their account passwords.

2.2.2. QCG middleware interface

The QCG middleware[8] includes a number of key components, such as QCG-Client,[9] QCG-Broker,[10] and QCG-NOW,[11] for job management and execution in HPC centers. The QCG client allows users to access HPC infrastructure and perform calculations on a single or many computational resources in an easy and flexible way [9]. The QCG client functionalities is available via command line with the interface very similar to the interfaces of popular queuing systems (e.g. SLURM, PBS). It allows submission of different types of jobs, including complex workflows, parameter sweep tasks and array jobs, on single or multiple clusters. The similar functionalities are also available with the QCG-Now which is a desktop graphical program that can be easily installed on laptops or PCs and used to run computational tasks on the HPC infrastructure accessible with QCG services.

2.2.3. QCG pilot jobs interface

Submitting a large group of jobs to the HPC scheduler can result in a long completion time as each single job is scheduled independently and waits in a queue. Moreover the submission of a large number of jobs can be restricted or prohibited by administrative policies on HPC resources. A Pilot Job is a container which resides within a single HPC resource job allocation and facilitates many sub-jobs that can be started and managed, without having to wait individually for resources to become available.

One can argue that there are available job array mechanisms in many systems, however the traditional job array mechanism is often platform-specific, and normally does not support the heterogeneous and dynamic workflows that are required for use cases such as

[7] http://www.paramiko.org.
[8] http://www.qoscosgrid.org.
[9] http://www.qoscosgrid.org/trac/qcg/wiki/user_information.
[10] http://www.qoscosgrid.org/trac/qcg-broker.
[11] http://www.qoscosgrid.org/qcg-now/en.

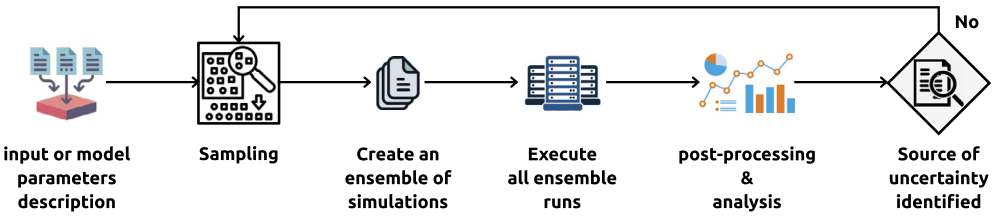


Fig. 4. Typical progression of a UQ procedure. FabSim3 relies on running multiple simulation instances to quantify uncertainty, with each instance relying on a different sample selection of input parameters. When an adaptive sampling method is used, the UQ procedure will be repeated until the selected uncertainties are accurately quantified.

multiscale simulations and/or adaptive sensitivity analysis. In FabSim3 we support pilot jobs by providing an integration with QCG Pilot Job[12] (QCG-PJ).

FabSim3 allows user to submit a large number of jobs within the pilot job mechanism. To use this functionality, users will need to install the QCG-PJ manager on the target remote machine. To do so, they can simply type:

```
>_ fabsim <machine_name> install_app:QCG-PilotJob,venv=True
```

Since the internet access inside the compute nodes is blocked on some supercomputers, we provide an installation process in an off-line mode, i.e., all required packages are downloaded and then transferred to the remote machine to be installed in a fresh python virtual environment.

2.3. Patterns

Computing patterns are intended to simplify specific aspects of the implementation of applications, to help identify commonalities between applications from different domains, and to encapsulate theoretical approaches that can be freely used by different applications, irrespective of their source domain. In the case of FabSim3, there are three patterns that are relevant to the design and implementation of the toolkit: multiscale computing patterns (MCPs), uncertainty quantification patterns (UQPs) and verification and validation patterns (VVPs).

MCPs are patterns that categorize multiscale applications into three main classes. To our knowledge, any multiscale application can be categorised to fit one of these three patterns, or a combination of these. The three patterns are described in detail by Alowayyed et al. [10], but are briefly defined as follows:

- Extreme Scaling (ES): here a primary single-scale model is coupled to a set of serial and/or parallel auxiliary models on any scale.
- Heterogeneous Multiscale Computing (HMC): this pattern represents a form of macro-micro coupling, where many micro-scale models are coupled to a single macro-scale model and launched on-demand. In this pattern, the macro-scale model then receives results from any number of micro-scale model instances, and uses this input to provide a more accurate result.
- Replica Computing (RC): this pattern contains a large number of individual model ensembles to produce statistically robust results. These ensembles/replicas are interdependent simulation runs and executed independently of each other. If a ensemble/replica (i.e. a simulation) fails, the RC patterns affords some level of fault tolerance, taking into account maintaining the overall statistics.

Most of the applications that use FabSim3 rely on RC, and FabSim3 provides specific support to make the execution of RC applications easier. The tool also provides support for ES applications, as is for instance demonstrated in the migration plugin example.

2.3.1. UQPs

Uncertainty quantification (UQ) is an increasingly prominent research area in scientific simulation. Its focus is to identify the sources of uncertainty in each component of the model, and quantify their magnitudes. A systematically quantified and controlled uncertainty is essential for reliable simulation results [11]. Within the VECMA project we developed Uncertainty Quantification Patterns (UQP) [12] which are generic patterns that facilitate uncertainty quantification and propagation or sensitivity analysis.[13] A general procedure/workflow for many UQPs can be found in Fig. 4 and be defined as follows:

- Create an ensemble of simulations, i.e., multiple individual simulations with different input or model parameters. Note that each simulation may consist of multiple (coupled) model executions that are provided with separate inputs, for instance in the case of multiscale simulations.
- Execute all ensemble runs
- Perform post-processing analysis by using statistical techniques, such as the Monte Carlo, Polynomial Chaos and the Stochastic Collocation methods, to measure the error distribution fore ach input or model parameters individually
- Refine and report the previous steps until the confidence in the simulation results is reached, i.e., identifying the source of uncertainty in the model parameters

[12] https://github.com/vecma-project/QCG-PilotJob.
[13] https://www.vecma.eu.

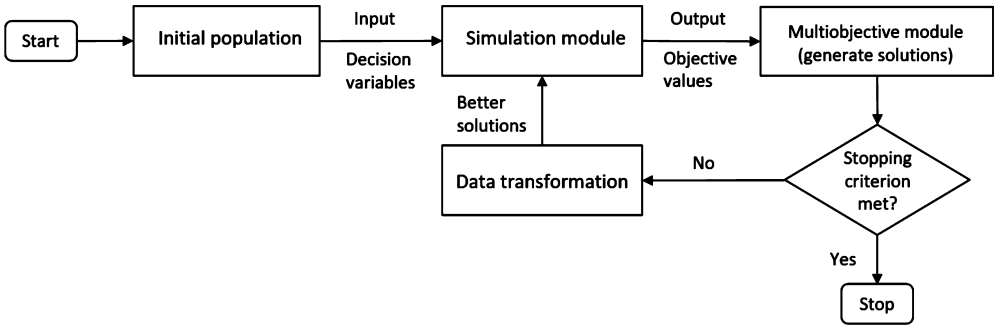


Fig. 5. Overview of a multiobjective optimisation workflow involving a simulation kernal.

2.3.2. VVPs

Verification and Validation (V&V) provide a framework for building confidence in predictions from computational simulation [13]. The verification process addresses the quality of the numerical treatment of the model used in the predication, and the validation process addresses the quality of the model [14]. Simulations that are systematically verified are more likely to be correctly implemented versions of their corresponding conceptual models, while simulations that are systematically validated are more likely to correspond to reality. Within VECMA we have identified four V&V patterns which can be applied to applications irrespective of their source domain, and irrespective of whether they are single- or multiscale. These patterns are (i) Stable Intermediate Forms (SIF); (ii) Level of Refinement (LoR); (iii) Ensemble Output Validation (EoV); and (iv) Quantity of interest Distribution Comparison (QDC).[14] We have example implementations in FabSim3 for three of the four VVPs, namely SIF, LoR and EoV.

2.3.3. Multiobjective optimization

A Multiobjective Optimization Problem (MOP) involves optimising at least two objectives/criteria simultaneously. Certain important decisions in humanitarian logistics can be formulated as MOPs. For example, the camp location selection problem focuses on determining the optimal camp locations where asylum-seekers/unrecognized refugees could arrive within a minimum time period or travel distance [15]. Other objectives may include the maximization of camp capacity and the minimization of average idle capacity [15]. The decisions related to camp location selection have an influence on the efficiency of providing humanitarian support to forcibly displaced persons and saving their lives.

In this release of FabSim3, we have established routines for multiobjective optimization and showcase their functionality specifically in the FabFlee plugin. In future releases, we intend to generalize this functionality into a pattern, making it available for use in other plugins as well.

Considering that forced migration is a complex phenomenon, a simulation-optimization approach has been developed to tackle the MOP involving uncertainty [15]. Fig. 5 shows the simulation-based optimization framework, which combines multiobjective evolutionary algorithms with an agent-based migration model (simulation module). To start with, a set of solutions are generated as initial population. Then the simulation module takes the decision variables as input and outputs objective values corresponding to each solution. Next, the multiobjective module is performed to compare solutions in the current population on the basis of their objective values and generate offspring of the current population for the next-generation evolution. The process iterates until a certain stopping criterion (e.g., the maximum number of generations) is satisfied.

2.4. Plugin system

An important evolution from the original version of FabSim [16] is the development of a flexible plugin system. In the original FabSim users would take the toolkit as a whole, and modify it to suit their own needs. Although this led to a good uptake in several domains initially, we discovered that it became hard to transfer improvements across differently modified versions, and that the modified versions themselves could differ in fundamental ways, making code maintenance and improvement even more difficult.

In FabSim3 we have therefore opted for a two-layer system where, first, a range of functionalities (Base, Backends and Patterns) are provided to all users, irrespective of their application domains. To complement this, FabSim3 offers a plugin system that allows users to introduce customisations and extensions in a modular and lightweight manner through plugin development. By default, any plugin code is written in Python3, but the plugins themselves can support applications using any language.

FabSim3 plugins serve a specific purpose and are often application-specific. Users define them to automate tasks such as code installation, model validation, coupled simulation executions for specific problems or iterative sensitivity analysis. In essence, the plugins provide application developers with the necessary sandbox to create and curate complex, reproducible and dynamic ensemble workflows and to make their simulations more robust by combining their workflows with VVUQ functionalities.

FabSim3 plugin development follows three main goals:

(a) *Provide Stability*:
- Plugins are non-intrusive in regards to simulation codes, and plugin functions act as a wrapper functionality around existing applications.
- Plugins can be reliably installed wherever FabSim3 is installed.

[14] For More details about each V&V patterns, see VECMA Deliverable 2.2.

Table 2
List of available tasks for FabSim3 plugin system.

Task Name	Brief description & Usage
`install_plugin`	Installs a FabSim3 plugin in your local system[a]. `>_ fabsim localhost install_plugin:<plugin_name>`
`avail_plugin`	Prints the list of available plugins. `>_ fabsim localhost avail_plugin`
`update_plugin`	Updates a specific FabSim3 plugin from its github repository. `>_ fabsim localhost update_plugin:<plugin_name>`
`remove_plugin`	Removes a specific FabSim3 plugin from your local system. `>_ fabsim localhost remove_plugin:<plugin_name>`

[a] The list of available plugins can be found in https://github.com/djgroen/FabSim3/blob/master/deploy/plugins.yml.

- Plugins provide a central point of implementation for key workflow definitions, to enable consistent use across the plugin user community.

(b) *Ease of Development*:
- Plugins are straightforward to develop, extend and support. As a starting point, we showcase this through the FabDummy plugin, which is a minimal platform that users can copy to begin developing their own plugin.
- Plugins have a sufficient level of simplicity for average user-developers, such that they can develop and improve plugin code without needing low-level Python programming experience.

(c) *Performance*:
- Plugin installation is fast, and occurs on the local machine only.
- Plugins support parallel processing, both locally and remotely.
- Plugins are able to exploit base functions in FabSim3 to improve application performance (e.g. ensemble simulation executions using Pilot Jobs).

We summarize the main commands to install, remove, and update FabSim3 plugins in Table 2.

2.5. Summary of differences with the original FabSim release

Aside from the introduction of a plugin system, described in the previous subsection, there have been a lot of functional and performance improvements since our original release in 2016 [16]. These include new features and capabilities, extended and revised the software documentation, and resolved technical issues reported by users. In Table 3 we present the main improvements and changes in FabSim3 toolkit relative to the original FabSim. In addition to all the changes described there, FabSim3 is developed using the Python3 platform and tools, while the original FabSim was developed using Python 2. To clearly communicate the fact that we now use Python 3 instead of 2, we chose to name the toolkit FabSim3, not FabSim2.

3. Performance analysis

Many scientific applications need to run large ensemble simulations (1000+ runs) to perform UQ and SA, which cannot be executed as individual jobs on most supercomputers due to scheduler constraints, and require a pilot job mechanism such as QCG-PJ. QCG-PJ has been shown to efficiently execute 10000 jobs with less than 10% overhead, even if those jobs only last for one second each.[15]

To improve the FabSim3 ensemble submission performance, we integrated it with QCG-PJ and enabled multi-processing job submission from the local machine. To demonstrate the benefit of this, we measured the total elapsed time of the job submission to a remote machine for the epidemiology[16] application. Owing to the limitation of maximum number of submitted jobs per user, i.e., 5000 jobs both in running and pending at a given time, on the Eagle supercomputer, we use ensembles sizes 500 and 5000 runs, and use up to 6 cores for the submission process. We have enabled pilot jobs for the larger ensemble for both large and small cases.

In Fig. 6 we present the time required to submit each job to a remote machine with FabSim3. We find that for a single core without QCG-PJ the job submission overhead is around 30 milliseconds. When using FabSim3 with QCG-PJ, the job submission per job reduces to around 1.7 milliseconds. For earlier FabSim3 test releases the job submission using a single core, without and with QCG-PJ was about 2466 milliseconds and 80 milliseconds, respectively. We have achieved a large reduction on job submission overhead by revising and greatly simplifying the job submission implementation. The benefits users by saving them time when a large ensemble of runs needs to be submitted to the remote machine.

4. Overview of FabSim3 plugins

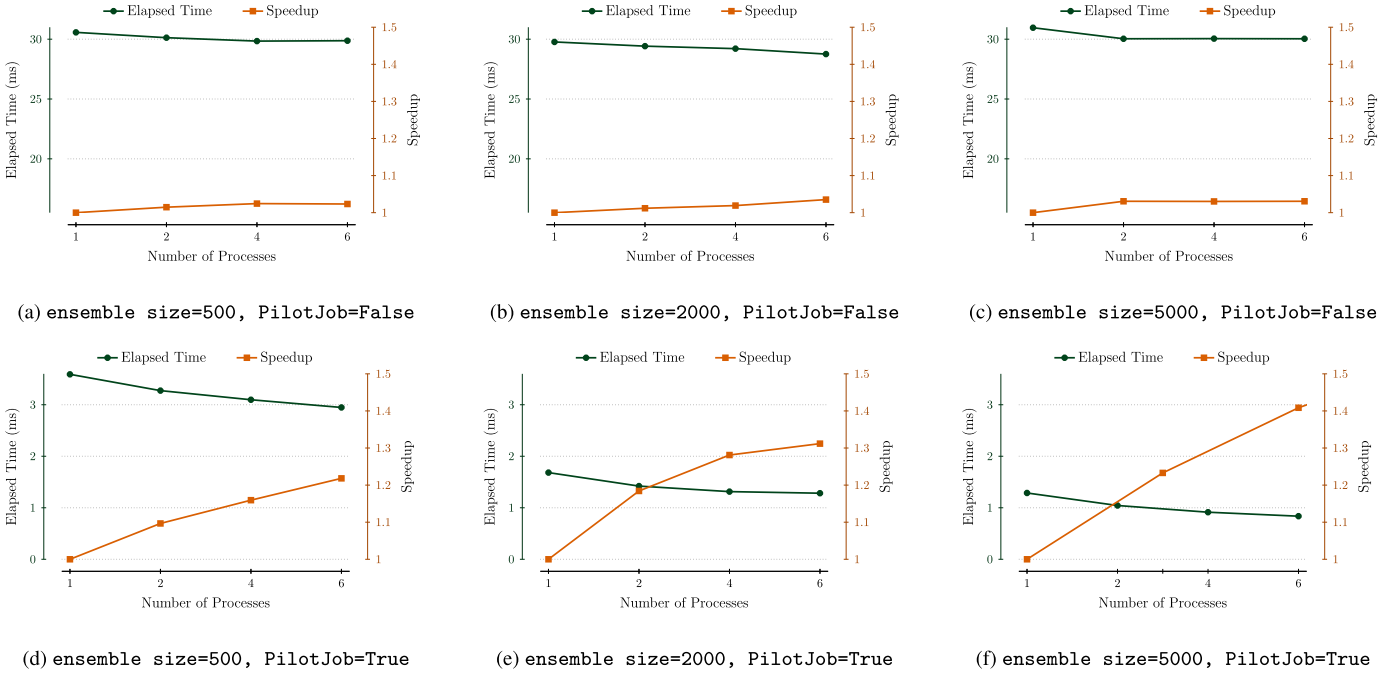
To illustrate the breadth of uptake of FabSim3, and highlight different examples of how to adopt it, we briefly present the main plugins that are currently available for FabSim3.

[15] Please see VECMA D5.2 for more details.
[16] FabCovid19 plugin: https://github.com/djgroen/FabCovid19.

Table 3
The list of modifications and enhancements in the development of FabSim3 since the original FabSim [16] release in 2016. The full change log can be found https://github.com/djgroen/FabSim3/releases.

	List of improvements and changes in development
Improved Functionalities	• Support for Eagle, Prometheus, ARCHER and SuperMUC supercomputers, among others. • Support for ensemble job execution. • Support for replica execution (multiple simulations with identical input data and input parameters). • Improved documentation, more examples. • Added support for job submission and monitoring. • Support for containerization (with FabSim3 deployable as a Docker and singularity image). • Added support multi-threaded job submission. • Revamped the documentation. • Added support for SSH multiplexing to simplify access.
Plugins	• FabDummy (https://github.com/djgroen/FabDummy) • FabFlee (https://github.com/djgroen/FabFlee) • FabNEPTUNE (https://github.com/UCL-CCS/FabNEPTUNE) • FabSCEMa (https://github.com/UCL-CCS/FabSCEMa) • FabMD (https://github.com/UCL-CCS/FabMD) • FabUQCampaign (https://github.com/wedeling/FabUQCampaign) • fabmogp (https://github.com/edaub/fabmogp) • FabMUSCLE (https://github.com/djgroen/FabMUSCLE) • FabSMD (https://github.com/potterton48/FabSMD) • FabCovid19 (https://github.com/djgroen/FabCovid19) • FabCovidSim (https://github.com/arabnejad/FabCovidSim)
Verification & Validation Patterns (VVP)*	To support the extension of single scale verification and validation procedures to multiscale settings. • Stable Intermediate Forms (SIF) • Level of Refinement (LoR) • Ensemble Output Validation (EoV) • Quantity of Interest (QoI)
Uncertainty Quantification & sensitivity analysis Patterns (UQPs)*	To automate routines for Uncertainty Quantification and Sensitivity Analysis. • Added FabUQCampaign plugin, which runs the UQ (uncertainty quantification) samples from an EasyVVUQ campaign via the `campaign2ensemble` subroutine of FabSim3, submiting the UQ samples as ensemble jobs to remote machines

* FabSim3 new features added after first release.



(a) `ensemble size=500, PilotJob=False` (b) `ensemble size=2000, PilotJob=False` (c) `ensemble size=5000, PilotJob=False`

(d) `ensemble size=500, PilotJob=True` (e) `ensemble size=2000, PilotJob=True` (f) `ensemble size=5000, PilotJob=True`

Fig. 6. Time required to submit per each job with FabSim3 (with/without QCG-PJ) relative to the number of cores used for job submission. Graph is made using an average of 10 repetition of each ensemble size. Please note that, here we only measure the job submission overhead, so queuing time and job execution on computing nodes are not considered in our test.

4.1. FabFlee

FabFlee[17] is the automation toolkit for the forced displacement simulation tool, namely Flee.[18] Flee is an agent-based simulation model, which forecasts the distribution of incoming refugees across destination camps under a range of different policy situations. By using FabFlee plugin, users are able to construct and modify refugee simulations, instantiate and execute multiple runs for different policy decisions, as well as to validate and visualize the obtained results against the existing data.

4.2. FabMD

FabMD[19] is FabSim3 plugin for the Molecular Dynamics (MD) simulation which used for understanding complex systems on the atomistic scale. A major use of MD simulation is to predict the binding affinity of a lead compound or drug with a protein target, of major relevance in drug discovery and personalized medicine. FabMD supports three MD simulation kernels at time of writing: LAMMPS, NAMD and GROMACS.

4.3. Fabmogp

fabmogp[20] is FabSim3 plugin for the `mogp`[21] (Multi-Output Gaussian Process) emulator to automate the simulation execution and Uncertainty Quantification (UQ) workflow of the earthquake simulation. The `mogp` emulator is a Python package for fitting Gaussian Process Emulators to computer simulation results. The code contains routines for fitting GP emulators to simulation results with a single or multiple target values, optimizing hyper parameter values, and making predictions on unseen data. The library also implements experimental design, dimension reduction, and calibration tools to enable modellers to understand complex computer simulations. To run the actual earthquake simulations, we use the `fdfault`[22] application, which is a high performance, parallelised finite difference code for simulation of frictional failure and wave propagation in elastic-plastic media.

4.4. FabCovid19

FabCovid19[23] is a FabSim3 plugin for the Flu And Coronavirus Simulator (FACS)[24] [17]. FACS is an agent-based modelling code that models the viral spread at the sub-national level, incorporating geospatial data sources to extract buildings and residential areas within a predefined region. By using FabCovid19, users are able to run different scenarios for different boroughs with one-liner bash commands. In addition, we are also currently using it to investigate how sensitive FACS forecasts are to some of FACS main assumptions.

4.5. FabUQCampaign

FabUQCampaign[25] is a general interface between EasyVVUQ and FabSim3. The former is a forward uncertainty propagation and sensitivity analysis toolkit. To perform this task, repeated sampling of the computational model is required. Hence, via FabUQCampaign the EasyVVUQ ensembles can be executed on remote supercomputers, including the data transfer to and from the remote host. A Python API is available as well, which allows one to execute FabSim3 command from within the EasyVVUQ Python script.

4.6. FabSCEMa

The tool in combination with an existing platform for verification, validation, and uncertainty quantification offers a scientific simulation environment and data processing workflows that enable the execution of single and ensemble simulations. It also supports the execution of remotely or locally submitted jobs through the plugin and helps the experts to do several specific annalistic tasks on a local machine or on a cluster or supercomputing platform within the EasyVVUQ and EasySurrogate architectures. It is a fully automated computational tool for the study of the uncertainty in a computational model of a heterogeneous multi-scale atomistic continuum coupling system, a publicly available open-source code SCEMa (https://github.com/UCL-CCS/SCEMa).

4.7. FabDummy

FabDummy[26] is a plugin that serves as an example for new plugin developers. It showcases basic functionalities such as remote job submission, ensemble job submission, as well as some of the VVP functionalities. FabDummy is not directly used for research itself; instead users typically clone it as a starting point whenever they want to begin writing their custom plugin.

[17] https://github.com/djgroen/FabFlee.
[18] https://github.com/djgroen/flee.
[19] https://fabmd.readthedocs.io.
[20] https://github.com/edaub/fabmogp.
[21] https://mogp-emulator.readthedocs.io.
[22] https://github.com/egdaub/fdfault.
[23] https://github.com/djgroen/FabCovid19.
[24] https://facs.readthedocs.io.
[25] https://github.com/djgroen/FabUQCampaign.
[26] https://github.com/djgroen/FabDummy.

4.8. FabSMD

FabSMD[27] is a FabSim3 plugin for *ensemble-based* Steered Molecular Dynamics (SMD) method [18]. Molecular dynamics attempts to recreate the physical motion of atoms by solving Newton's equations. SMD applies a force to the system by attaching a fictitious Newtonian spring to a selected group of atoms to predict how long the drug remains bound for (drug-target residence time). It is important to use *ensemble-based* approaches in SMD, especially when trying to obtain numerical values from the simulations, as the results are from a distribution of possible results. These results depend on the initial velocities of the atoms which are taken from the Maxwell-Boltzmann distribution.

5. Conclusion

Computational models have become prevalent in describing and predicting the behaviour of real-world processes and systems. In many cases, the computational models are based on theories and/or mathematical equations to represent problems and produce simulation outcomes. Typically, a simulation involves tasks such as preparation, submission and analysis, which are time expensive, especially where there are a large number ensembles or replicas in the scenario. Furthermore, the credibility and impact of a simulation can be greatly improved when it can easily be shared and reproduced. Automation tools therefore enhance the credibility of simulation results while simplifying the simulation workflow.

In this paper, we have presented FabSim3, a toolkit to reduce the complexity of tasks associated with computational research. We presented use cases spanning a comprehensive range of simulation domains, along with their respective FabSim3 plugins. The simplest plugin is FabDummy which focuses on automation of single or ensemble jobs to the target remote machine. The FabFlee plugin is the most advanced plugin in the collection, as it uses advanced FabSim3 APIs such as multiple VVPs and UQPs.

Overall, the FabSim3 plugin system provides several classes of benefits for users, including (a) automation and productivity, (b) built-in sharing and semi/full reproducibility, (c) lowered required developer skill level and expertise, and (d) diminished user error rate.

CRediT authorship contribution statement

Derek Groen: Conceptualization, Funding acquisition, Methodology, Software, Supervision, Validation, Writing – original draft, Writing – review & editing. **Hamid Arabnejad:** Methodology, Software, Visualization, Writing – original draft. **Diana Suleimenova:** Software, Supervision, Validation. **Wouter Edeling:** Methodology, Software, Writing – review & editing. **Erwan Raffin:** Funding acquisition, Software, Validation. **Yani Xue:** Software, Validation. **Kevin Bronik:** Software, Validation. **Nicolas Monnier:** Software, Validation. **Peter V. Coveney:** Conceptualization, Funding acquisition, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

We are grateful to Robin Richardson, Robert Sinclair, Maxime Vassaux, David Wright, James Suter, Daan Crommelin, David Coster, Tomasz Piontek, Bartosz Bosak, Alireza Jahani and Eric Daub for their valuable inputs, as well as to all other FabSim3 plugin developers and test users for their contributions. The scientific computations presented in this work have been performed using the Eagle supercomputer at the Poznan Supercomputing and Networking Centre in Poland. This work has been supported by the SEAVEA ExCALIBUR project, which has received funding from EPSRC under grant agreement EP/W007711/1, as well as by the VECMA and HiDALGO projects, which have received funding from the European Union Horizon 2020 research and innovation programme under grant agreement nos 800925 and 824115. In addition, FabFlee was supported by the ITFLOWS project and FabCovid19 by the STAMINA project, both of which have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 882986 and No 883441 respectively.

Appendix A. How to write a FabSim3 plugin

In this tutorial, we explain how to write a FabSim3 plugin from scratch. To keep simplicity, the basic functionalities are presented here, for more advanced and complicated functionalities, we suggest reader to have look at the current plugins presented in Section 4 in this work.

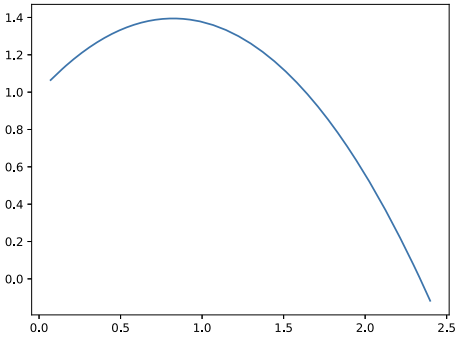
For this tutorial, a simple application, namely `cannon_app`, which calculates the range of a projectile fired at an angle is selected. By using simple physics rules, you can find how far a fired projectile will travel. The source code for this application, written in three of the most widely used languages: C,Java, and Python, is available here: https://github.com/arabnejad/cannon_app. The `cannon_app` reads the input parameters from a simple text file and calculate the distance until ball hits the round. How far the ball travels will depend on the input parameters such as: speed, angle, gravity, and air resistance. Fig. A.7 shows a sample input setting file and the generate output plot.

[27] https://github.com/potterton48/FabSMD.

simsetting.txt

```
"gravity", 9.8
"mass", 5.8
"velocity", 10.0
"angle", 0.79
"height", 1.0
"air_resistance", 0.2
"time_step", 0.01
```

(a)



(b)

Fig. A.7. (a) Sample input parameter file and (b) generated output plot by `cannon_app`.

A.1. Step 1 : FabSim3 installation

If you have already installed FabSim3, you can ignore this part and move to step 2. To install FabSim3 in your local machine:

1. Clone or download FabSim3 from https://github.com/djgroen/FabSim3. To clone the code from the GitHub repository, simply type:

```
>_ git clone https://github.com/djgroen/FabSim3.git
```

2. To install all required python packages automatically and configure yml files, please go to your FabSim3 directory, and type:

```
>_ cd FabSim3
>_ python3 configure_fabsim.py
```

3. After the installation process, the root FabSim3 directory should be added to both `$PATH` and `$PYTHONPATH` environment variables. To make these updates permanent, you need to update both `$PATH` and `$PYTHONPATH` variables in your bash shell script. The instruction to do that will be shown at the end of output of step 2.
4. To make the `fabsim` command available in you system, please restart the shell by opening a new terminal or just re-load your bash profile by `source` command, e.g., `source ~/.bashrc`.
5. To make sure that installation is done correctly and `fabsim` command available in you system, you can execute:

```
>_ which fabsim
<FabSim3 folder>/bin/fabsim
```

   make sure that `<FabSim3 folder>` is pointed to your local cloned or downloaded FabSim3 repo in your local machine.
6. if you have encountered any problems during the installation, please check the known issues and how to fix them here: https://fabsim3.readthedocs.io/en/latest/installation.html#known-issues

A.2. Step 2 : New plugin preparation

To create a new plugin for `cannon_app` application, you need to follow a files and folders structure to be used by FabSim3. To do that, follow these steps:

1. Create a folder, namely `FabCannonsim` under `plugins` in your local FabSim3 directory.
2. Create two sub-folders, `config_files` where we put the application there, and `templates` where all templates files are placed.
3. Clone or download the `cannon_app` application in the `FabCannonsim/config_files` folder that just created.

```
>_ cd FabSim3/plugins/FabCannonsim/config_files
>_ git clone https://github.com/arabnejad/cannon_app.git
```

4. in the `FabCannonsim` folder, create two empty files:
   (a) `FabCannonsim.py`, which contains the plugin source code.
   (b) `machines_FabCannonsim_user.yml`, which contains the plugin environmental variables.
5. Add a new plugin entry in `plugins.yml` file located in `FabSim3/deploy` folder.

```
FabCannonsim:
  repository: <empty>
```

For now, we left `repository` with empty value. Later, this can be filled by the github repo address of your plugin.

To summarize this part, by following above steps, the file and directory should be as shown in Fig. A.8

Fig. A.8. (a) demonstrates the directory tree structures, and (b) show the updated `plugins.yml` file located in `FabSim3/deploy` folder. Please note that, the folders name highlighted with red colour in (a) will be used by FabSim3 for job configuration and execution and should not be changed. Also, all three (1) the plugin name, here `FabCannonsim`, (2) the plugin fabric python file, and (3) the plugin entry in `plugins.yml` file, should be identical. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

```
1   from fabsim.base.fab import *
2   # Add local script, blackbox and template path.
3   add_local_paths("FabCannonsim")

4   @task
5   @load_plugin_env_vars("FabCannonsim")
6   def Cannonsim(app, **args):
7       # Submit a single job of Cannon_app
8       #       >_ fabsim <remote_machine>
        ↪  Cannonsim:cannon_app
9       # e.g., >_ fabsim localhost Cannonsim:cannon_app
10      update_environment(args)
11      with_config(app)
12      execute(put_configs, app)
13      env.script = "cannonsim"
14      job(args)
```

Fig. A.9. Single job submission function.

A.3. Step 3: Write application-specific functionalities

To call and execute a function from command line, it should be tagged as a Fabric task class. This part of tutorial explains how to write a function/task to execute a single or ensemble jobs of your application.

A.3.1. Single job execution

Fig. A.9 show a sample function for single job execution in `FabCannonsim.py`.
The following paragraphs will explain it line by line:

- **`from base.fab import *`**
  loads all FabSim3 pre-defined functions
- `add_local_paths("FabCannonsim")`
  sets the default location for templates, python scripts, and config files
- `@task`
  Marks the function, as a callable task, to be executed when it invoked by`fabsim` from command line.
- `@load_plugin_env_vars("FabCannonsim")`
  Loads all machine-specific configuration information that are specified by the user for the input plugin name. Fig. A.10 shows the sample machine-specific configuration yaml file for the `cannon_app` application.
- **`def`** `Cannonsim(app, **args)`
  Defines the task name. The defined task name can be called from command line alongside `fabsim` command, e.g., `>_ fabsim <remote/local machine> Cannonsim:<input parameters>`
- `update_environment(args)` is predefined FabSim3 function which updated the environmental variables that are used as a combination settings registry and shared inter-task data namespace. The complete list of FabSim3 environmental variables can be found in `machines.yml` and `machines_user.yml` located in `FabSim3/fabsim/deploy` folder.

```yaml
default:
  # require command for compile and execute C code version
  c_app_run_prefix: "gcc cannonsim.cpp -o cannonsim -lm"
  c_app_run_command: "./cannonsim"
  # require command for compile and execute python code version
  py_app_run_command: "python cannonsim.py"
  # require command for compile and execute JAVA code version
  java_app_run_prefix: "export CLASSPATH='java_libs/commons-cli-1.3.1.jar:.'"
  java_app_compile_command: "javac cannonsim.java"
  java_app_run_command: "java cannonsim"

localhost:

eagle_vecma:
  cores: 1
  job_wall_time : "0-0:10:00"
  partition_name: "fast"

  modules:
    loaded: ["python/3.7.3","java8/jdk1.8.0_40"]
    unloaded: ["python"]
```

Fig. A.10. Sample machine-specific configuration yaml file for the `cannon_app` application.

- `with_config(args)` augments the FabSim3 environment variable, such as the remote location variables where the config files for the job should be found or stored, with information regarding a particular configuration name.
- `execute(put_configs, app)` transfers the config files to the remote machine to be used in launching jobs.
- `env.script = "cannonsim"` the `env.script` variable contains the name of template script file to be used for execution of job on the target machine, which can be local host or HPC resources. This script will be called when the job execution starts, and contains all steps, such as set environment variable, or commands line to call/execute the application. Fig. A.11(a) shows the script file,[28] namely `cannonsim`, for the `cannon_app` application.
  At mentioned before, FabSim3 uses a template/variable substitution system[29] to easily generate the required script for executing job on the target local/remote machine. The used system is `$`-based substitutions, where `$var` will replaced by the actual value of the variable `var`, and `$$` is an escape and is replaced with a single `$`. Figs. A.11(b) and (c) are two generated sample for input script A.11(a) and the loaded environmental variables from Fig. A.10.
- `job(args)`
  is an internal low level job launcher defined in FabSim3.

To submit and execute a single `cannon_app` job,

```
# to execute on localhost
>_ fabsim localhost Cannonsim:cannon_app
# to execute on remote machine
>_ fabsim eagle_vecma Cannonsim:cannon_app
```

Please note that, the target machine, e.g., `localhost` or `eagle_vecma`, should be defined and available in both in `machines.yml` and `machines_user.yml` files. To see the machine configuration attribute, you can run: `fabsim <target_host> machine_config_info`. Additionally, you can overwrite or add new attributes to the target machine, tailored to your plugin, in `machines_<plugin_name>_user.yml` file.

A.3.2. An ensemble job execution

An ensemble-based simulation uses variation in input or output data, model parameters, and/or available versions of models to improve the simulation performance. For the `cannon_app` application, the input `simsetting.txt` file can be varied for different ensemble runs. To setup an ensemble simulation, first we need to create a `SWEEP`[30] folder in the root directory of application. Inside the `SWEEP` folder, each ensemble runs should represented by different folder name. To vary the input `simsetting.txt` file, we should follow the same relative path of that file inside each run directory in `SWEEP` folder. Fig. A.12 illustrates a sample files and folders structures with 3 ensemble runs.

Fig. A.13 show the sample function for an ensemble execution in `FabCannonsim.py` file. Most part of this code is already explained in the previous section, i.e., single job submission. The following paragraphs will explain the required lines for an ensemble functionality:

- `sweep_dir = find_config_file_path(app) + "/SWEEP"`
  set the `SWEEP` directory PATH. As it mentioned earlier, the `SWEEP` directory should be located in the root of the application. The API

[28] By default, FabSim3 loads all required scripts from `templates` folder located in plugin directory. Hence, the `cannonsim` file should be saved in `FabSim3/plugins/FabCannonsim/templates` directory.

[29] https://docs.python.org/3/library/string.html#template-strings.

[30] N.B. by default, FabSim3 builds and constructs the required number of ensembles runs based on a default folder, namely `SWEEP`, located inside the application config directory.

```
# change directory to where application is stored
cd $job_results
$run_prefix

OUTPUT_DIR="output_files"
INPUT_DIR="input_files"

# run c program
$c_app_run_prefix
$c_app_run_command

# run python program
$py_app_run_command

# run java program
$java_app_run_prefix
$java_app_compile_command
$java_app_run_command

# show output results
echo "Output results for python program :"
cat $$OUTPUT_DIR/py_output.txt

echo "Output results for C program :"
cat $$OUTPUT_DIR/c_output.txt

echo "Output results for Java program :"
cat $$OUTPUT_DIR/java_output.txt
```

(a) The `cannonsim` script file for job execution

```
# change directory to where application is stored
cd ~/FabSim3/results/cannon_app_localhost_1
/bin/true || true

OUTPUT_DIR="output_files"
INPUT_DIR="input_files"

# run c program
gcc cannonsim.cpp -o cannonsim -lm
./cannonsim

# run python program
python cannonsim.py

# run java program
export CLASSPATH='java_libs/commons-cli-1.3.1.jar:.'
javac cannonsim.java
java cannonsim

# show output results
echo "Output results for python program :"
cat $OUTPUT_DIR/py_output.txt

echo "Output results for C program :"
cat $OUTPUT_DIR/c_output.txt

echo "Output results for Java program :"
cat $OUTPUT_DIR/java_output.txt
```

(b) The generated script for executing the `cannon_app` application on the localhost

```
#!/bin/bash
#SBATCH --nodes 1
#SBATCH --tasks-per-node=28
#SBATCH --time=0-0:10:00
#SBATCH --account=vecma2020
#SBATCH --output=/home/hamid/FabSim3/results/cannon_app_ea
↪  gle_vecma_1/JobID-%j.output
#SBATCH --error=/home/hamid/FabSim3/results/cannon_app_eag
↪  le_vecma_1/JobID-%j.error
#SBATCH --mem=6000
#SBATCH --partition=fast

cd /home/hamid/FabSim3/results/cannon_app_eagle_vecma_1
module unload python
module load python/3.7.3
module load java8/jdk1.8.0_40

OUTPUT_DIR="output_files"
INPUT_DIR="input_files"

# run c program
gcc cannonsim.cpp -o cannonsim -lm
./cannonsim

# run python program
python cannonsim.py

# run java program
export CLASSPATH='java_libs/commons-cli-1.3.1.jar:.'
javac cannonsim.java
java cannonsim

# show output results
echo "Output results for python program :"
cat $OUTPUT_DIR/py_output.txt

echo "Output results for C program :"
cat $OUTPUT_DIR/c_output.txt

echo "Output results for Java program :"
cat $OUTPUT_DIR/java_output.txt
```

(c) The generated script for executing the `cannon_app` application on the sample SLURM based remote machine

Fig. A.11. Overview of how the script template (a) is used to generate job scripts for local execution (b) and remote execution on a HPC resource using SLURM (c).

`find_config_file_path(app)` will return the PATH to the application, here, the return value will be: `FabSim3/plugins/FabCannonsim/config_files/cannon_app`

- `run_ensemble(app, sweep_dir, **args)`
  is an internal low level function to map and execute ensemble jobs. Two mandatory input arguments are: (1) the config/application directory name, and (2) the PATH to SWEEP directory which contains inputs that will vary per ensemble simulation instance.

As you can see in Fig. A.13, unlike the single job execution, there is no need to call `execute(put_configs, app)`; The `execute` function will be called automatically by the `run_ensemble` API.

To submit and execute an ensemble `cannon_app` job,

```
# to execute on localhost
>_ fabsim localhost Cannonsim_ensemble:cannon_app
# to execute on remote machine
>_ fabsim eagle_vecma Cannonsim_ensemble:cannon_app
```

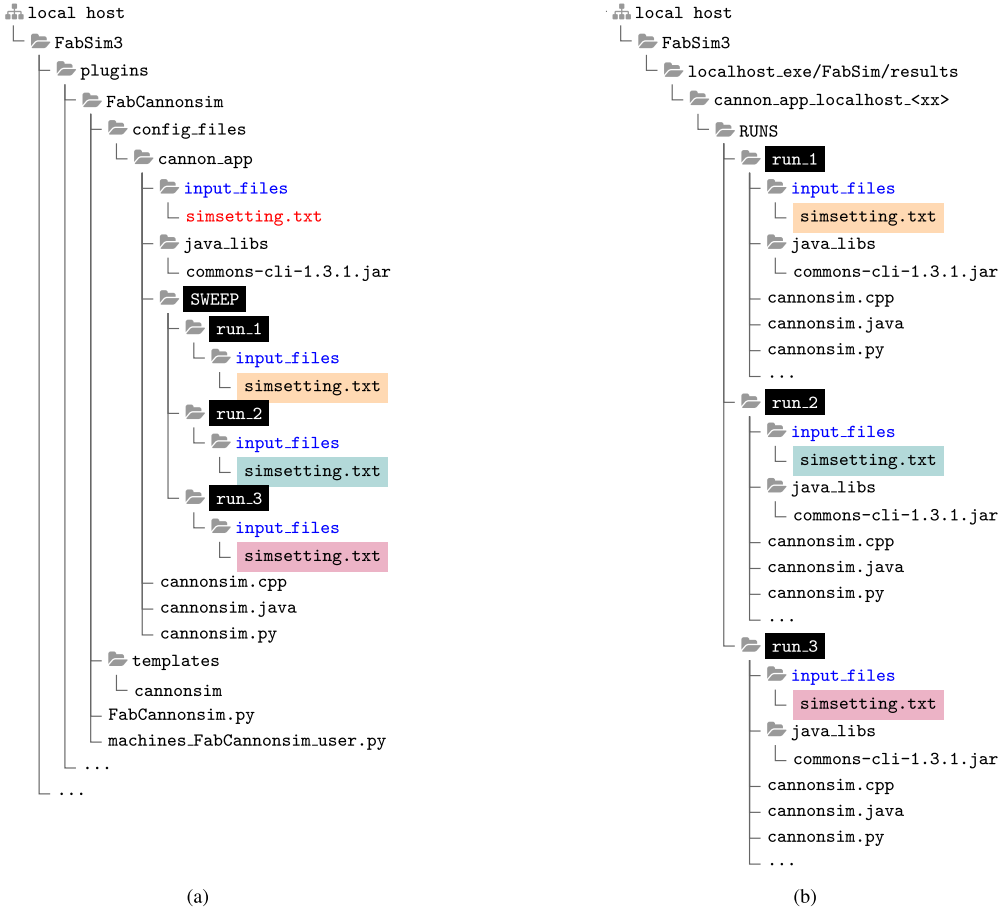


Fig. A.12. (a) A sample files and folders structure for `cannon_app` application with 3 ensemble runs. Please not that, the target file, here is `simsetting.txt`, should follow the same path as the original version. (b) the generated files and folder structure for execution side of the ensemble execution.

```python
from base.fab import *
# Add local script, blackbox and template path.
add_local_paths("FabCannonsim")

@task
def Cannonsim(app, **args):
    ...

@task
def Cannonsim_ensemble(app, **args):
    # Submit an ensemble of canon_app jobs
    #       >_ fabsim <remote_machine>
    ↪  Cannonsim_ensemble:cannon_app
    # e.g., >_ fabsim localhost Cannonsim_ensemble:cannon_app
    update_environment(args)
    with_config(app)
    sweep_dir = find_config_file_path(app) + "/SWEEP"
    env.script = "cannonsim"
    run_ensemble(app, sweep_dir, **args)
```

Fig. A.13. Ensemble job submission function.

Here, the target machine, e.g., `localhost` or `eagle_vecma` should be defined and available in both in `machines.yml` and `machines_user.yml` files. To see the machine configuration attribute, you can run: `fabsim <target_host> machine_config_info`. Additionally, you can overwrite or add new attributes to the target machine, tailored to your plugin, in `machines_<plugin_name>_user.yml` file.

References

[1] D. Groen, J. Knap, P. Neumann, D. Suleimenova, L. Veen, K. Leiter, Philos. Trans. R. Soc. Lond. Ser. A 377 (2142) (2019) 20180147.

[2] G. Pizzi, A. Cepellotti, R. Sabatini, N. Marzari, B. Kozinsky, Comput. Mater. Sci. 111 (2016) 218–230, https://doi.org/10.1016/j.commatsci.2015.09.013.
[3] F. Di Natale, Maestro workflow conductor, Tech. Rep., Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), 2017.
[4] S. Portegies Zwart, S. McMillan, Astrophysical Recipes; the Art of Amuse, Astrophysical Recipes; the Art of AMUSE, by Portegies Zwart, Simon; McMillan, Steve, IOP Publishing, Bristol, UK, ISBN 978-0-7503-1321-6, 2018, IOP ebooks.
[5] S.F.P. Zwart, S.L. McMillan, A. van Elteren, F.I. Pelupessy, N. de Vries, Comput. Phys. Commun. 184 (3) (2013) 456–468.
[6] J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M.B. Belgacem, B. Chopard, D. Groen, P.V. Coveney, A.G. Hoekstra, J. Comput. Sci. 5 (5) (2014) 719–731.
[7] A. Craig, S. Valcke, L. Coquart, Geosci. Model Dev. 10 (9) (2017).
[8] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, in: Advanced Web Services, Springer, 2014, pp. 527–549.
[9] T. Piontek, B. Bosak, M. Ciżnicki, P. Grabowski, P. Kopta, M. Kulczewski, D. Szejnfeld, K. Kurowski, J. Grid Comput. 14 (4) (2016) 559–573.
[10] S. Alowayyed, T. Piontek, J.L. Suter, O. Hoenen, D. Groen, O. Luk, B. Bosak, P. Kopta, K. Kurowski, O. Perks, et al., Future Gener. Comput. Syst. 91 (2019) 335–346.
[11] T.J. Sullivan, Introduction to Uncertainty Quantification, vol. 63, Springer, 2015.
[12] D. Ye, L. Veen, A. Nikishova, J. Lakhlili, W. Edeling, O. Luk, V. Krzhizhanovskaya, A. Hoekstra, Philos. Trans. R. Soc. Lond. Ser. A 379 (2197) (2021) 20200072.
[13] C.J. Roy, J. Comput. Phys. 205 (1) (2005) 131–156.
[14] I. Babuska, J.T. Oden, Comput. Methods Appl. Mech. Eng. 193 (36) (2004) 4057–4066.
[15] Y. Xue, M. Li, H. Arabnejad, D. Suleimenova, A. Jahani, B.C. Geiger, Z. Wang, X. Liu, D. Groen, in: International Conference on Computational Science, Springer, 2022, pp. 497–504.
[16] D. Groen, A.P. Bhati, J. Suter, J. Hetherington, S.J. Zasada, P.V. Coveney, Comput. Phys. Commun. 207 (2016) 375–385.
[17] I. Mahmood, H. Arabnejad, D. Suleimenova, I. Sassoon, A. Marshan, A. Serrano-Rico, P. Louvieris, A. Anagnostou, S.J.E. Taylor, D. Bell, et al., J. Syst. Simul. 16 (4) (2022) 355–373.
[18] A. Potterton, F.S. Husseini, M.W. Southey, M.J. Bodkin, A. Heifetz, P.V. Coveney, A. Townsend-Nicholson, J. Chem. Theory Comput. 15 (5) (2019) 3316–3330.