Tutorial applications for Verification, Validation and Uncertainty Quantification using VECMA toolkit


DOI
10.1016/j.jocs.2021.101402

Publication date
2021

Document Version
Final published version

Published in
Journal of Computational Science

License
CC BY

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Tutorial applications for Verification, Validation and Uncertainty Quantification using VECMA toolkit

Diana Suleimenova a, *, Hamid Arabnejad a, Wouter N. Edeling b, David Coster c, Onnie O. Luk c, Jalal Lakhli c, Vytautas Jancauskas d, Michal Kulczewski e, Lourens Veen f, Dongwei Ye g, Pavel Zun g, Valeria Krzhizhanovskaya g, Alfons Hoekstra g, Daan Crommelin b, g, Peter V. Coveney h, Derek Groen a, h

a Department of Computer Science, Brunel University London, London, UK
b Scientific Computing Group, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
c Max-Planck Institute for Plasma Physics, Garching, Munich, Germany
d Leibniz Supercomputing Centre, Garching, Germany
f Netherlands eScience Center, Amsterdam, The Netherlands
g University of Amsterdam, Amsterdam, The Netherlands
h Centre for Computational Science, University College London, London, UK

ARTICLE INFO

Keywords:
Validation
Verification
Sensitivity analysis
Uncertainty quantification

ABSTRACT

The VECMA toolkit enables automated Verification, Validation and Uncertainty Quantification (VVUQ) for complex applications that can be deployed on emerging exascale platforms and provides support for software applications for any domain of interest. The toolkit has four main components including EasyVVUQ for VVUQ workflows, FabSim3 for automation and tool integration, MUSCLE3 for coupling multiscale models and QCG tools to execute application workflows on high performance computing (HPC). A more recent addition to the VECMA toolkit is EasySurrogate for various types of surrogate methods. In this paper, we present five tutorials from different application domains that apply these VECMA toolkit components to perform uncertainty quantification analysis, use surrogate models, couple multiscale models and execute sensitivity analysis on HPC. This paper aims to provide hands-on experience for practitioners aiming to test and contrast with their own applications.

1. Introduction

The 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. However, the computation of the model and reality is subject to the uncertainty that emerges from various sources. We use Verification, Validation and Uncertainty Quantification (VVUQ) analysis to determine and estimate uncertainty and their sources in the computational models.

VVUQ analysis is crucial as verification determines how accurately the model solves the mathematical equations applied in the simulation, validation defines the degree to which the models accurately represent the real world, and uncertainty quantification (UQ) identifies how variations in input parameters affect simulation results. Overall, VVUQ process provides the level of accuracy and reliability in any given model and obtained simulation results [1].

There are several tools available in the research area of VVUQ, which provide algorithms for parameter investigations, model calibration, optimisation and UQ analysis. In this paper, we solely focus on the VECMA toolkit (VECMAtk) that facilitates VVUQ techniques and patterns for verification and validation (V&V), sensitivity analysis (SA) and UQ in application to single and multiscale simulations [1].

VECMAtk has four main components, namely EasyVVUQ [2] that is used for simplifying the implementation and use of VVUQ workflows, in...
that takes input data in EasyVVUQ internal format and outputs an input
language is supported. The Decoder is a parser that takes the output of
provides support for large scale execution of jobs. There is an optimised
EasyVVUQ divides VVUQ workflows into several distinct stages –
2. Application of FabSim3 and EasyVVUQ: forced human
Forecasting forced human migration is crucial since global forced
parameters, we recommend reducing the number of
Manual routine tasks in simulations, such as construction, execution, analysis, and validation of various models, can be simplified using automation tools. For Flee application, we use the FabSim3 toolkit to simplify and accelerate activities [17], as well as automate several phases of Flee-based simulations. Specifically, we use the FabSim3-based plugin FabFlee to instantiate and execute multiple runs for different policy decisions, and to validate and visualize the obtained results against the existing data [18].

There are four different ways to execute multiscale migration simulations in FabFlee: (1) Single-model execution, (2) Ensemble execution, (3) Replica execution, and (4) Coupled execution. Each method has its unique purpose. The single-model execution can be easily performed on a laptop and instantly provide an overview to users. The ensemble execution could be useful for those who run multiple simulation instances simultaneously with different inputs or configuration settings of a target simulation run. While the replica execution could be an interesting option for those who run simulations multiple times at once with identical inputs due to the uncertain nature of a code. The coupled executions allow to couple macroscale and microscale (multiscale) models and conflict scenarios with the weather, telecommunication and other data sources.

All FabFlee simulation tasks are callable from the terminal, adhere to the following structure shown in Fig. 1. Moreover, we present the list of available FabFlee tasks and their description in Table 1.

2.1. Sensitivity analysis on input parameters of Flee

Sensitivity analysis (SA) is a well-established approach to analyse the influence of changes in assumptions used in modelling and simulation research [19]. It helps to identify which input parameters or assumptions have a higher impact or influence on the simulation output. SA results can be used to provide reliable parameters/assumption estimates for validation and model improvement. The SA process may involve investigation of the influence of changes in (a) model structure, or (b) input parameters. For this tutorial, we apply SA to the Flee algorithm and investigate which input parameters are pivotal in the simulation output.

The Flee code is based on the algorithm assumptions for forced migration including several parameters defining the movement logic of forcibly displaced people (see Suleimenova et al. [20] for a more detailed description of the algorithm and input parameters). The list of input parameters defining forced migration simulation algorithm is described below and parameter ranges are illustrated in Table 2:

- **max_move_speed**: Agents’ maximum movement speed in the simulation while traversing between locations with vehicles.
- **max_walk_speed**: Agents’ maximum movement speed in the simulation while travelling on foot between locations.
- **camp_move_chance**: Probability of an agent moving from a camp location where an agent resides to another location.
- **conflict_move_chance**: Probability of an agent moving from a conflict location where an agent resides to another location.
- **default_move_chance**: Probability of an agent moving from other (default) location where an agent resides to another location.
- **camp_weight**: The attractiveness value for camp locations making them twice as likely to be chosen as destination.
- **conflict_weight**: The attractiveness value for conflict locations making them four times less likely to be chosen as destination.

For forced migration sensitivity analysis, we use FabSim3 and EasyVVUQ components of VECMAtk, which provide an automated execution environment to achieve highly transparent and customised simulations by simplifying and accelerating key task activities.

**Step 1: Installation**

To perform this tutorial, the following software packages are required: (i) Flee code [21], (ii) FabSim3 toolkit [3], (iii) FabFlee plugin [22], and (iv) EasyVVUQ [2]. To install these application, simply follow the instruction below:

**Flee**

To clone the Flee code into your working directory, simply type:

```
git clone https://github.com/djgroen/flee.git
```

**FabSim3**

To clone the FabSim3 toolkit, simply type:

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

To install all required python packages automatically and configure YML files, simply go to your FabSim3 directory and type:

```
python3 configure_fabsim.py
```

If you encounter an error or issue during the installation process, please see the Section known issues in the FabSim3 documentation.

After installation and configuration process, the main FabSim3 directory is added in your $PYTHONPATH and $PATH environment variable. You can find these changes on your bash profile (for Linux check ~/.bashrc, and for MacOS check ~/.bash_profile).

Then, to make the fabsim command available in your system, restart the shell by opening a new terminal or just re-load your bash profile using the source command.

```
source ~/.bashrc
```

To make sure that installation is done correctly and the fabsim command available in your system, simply execute the following command:

```
which fabsim
```

To use the Flee code library in FabFlee, we need to add the Flee location to the system PYTHONPATH. To add Flee, simply go to <FabSim3_dir>/plugins/FabFlee directory, and update the machines_FabFlee_user.yml file by adding the variable flee_-location under localhost section as shown below:

```
localhost:
  # location of flee in your local PC
  flee_location: "<PATH_TO_FLEE>"
```

**FabFlee**

To install the FabFlee plugin, simply go to <FabSim3_dir> and type:

```
fabsim localhost install_plugin:FabFlee
```

The FabFlee plugin will appear in <FabSim3_dir>/plugins/FabFlee.

To use the Flee code library in FabFlee, we need to add the Flee location to the system PYTHONPATH. To add Flee, simply go to <FabSim3_dir>/plugins/FabFlee directory, and update the machines_FabFlee_user.yml file by adding the variable flee_-location under localhost section as shown below:

```
localhost:
  # location of flee in your local PC
  flee_location: "<PATH_TO_FLEE>"
```
EasyVVUQ

EasyVVUQ is a Python library and build upon existing libraries, such as Chaospy, for statistical functionalities. To install EasyVVUQ, simply type:

```bash
git clone https://github.com/UCL-CCS/EasyVVUQ
```

There are several sampling methods for UQ analysis in EasyVVUQ, such as Stochastic Collocation, Polynomial Chaos Expansion, Monte Carlo and Markov-Chain Monte Carlo techniques. The easiest way to examine these methods is to follow Jupyter Notebooks provided in http://mybinder.org/v2/gh/UCL-CCS/EasyVVUQ/dev?filepath=tutorials.

Step 2: Parameter exploration

To perform sensitivity analysis on input parameters of Flee, we mainly focus on two sampler examples, namely (a) SCSampler (Stochastic Collocation sampler) and (b) PCESampler (Polynomial Chaos Expansion sampler), that are available in EasyVVUQ. The configuration for SA can be set in `flee_SA_config.yml` located in `<FabSim3_dir>/plugin/FabFlee/SA` directory. All required configurations for FabFlee SA, such as sampler name, varying input parameters, and the number of polynomial order, are loaded from `flee_SA_config.yml` file. To illustrate, we present an example of two config parameters below, namely (`max_move_speed` and `max_walk_speed`):

```yaml
# Some example config parameters:
# vary_parameters.range:
#   # parameter name:
#   # range: [lower value, upper value]
# max_move_speed:
#   range: [80, 100]
# max_walk_speed:
#   range: [10, 100]
```

The results will then be saved in `<FabSim3_dir>/plugins/FabFlee/SA/flee_SA_SCSampler` directory.

Table 1
List of tasks commonly used in FabFlee.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Brief description &amp; Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>flee / pflee</td>
<td>Executes a conflict scenario for the input simulation period.</td>
</tr>
<tr>
<td>pflee ensemble</td>
<td>Executes the ensemble of simulation runs for a conflict scenario with the input simulation period.</td>
</tr>
</tbody>
</table>

Table 2
Defining an input parameter space for the uncertain parameters of the Flee simulation.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Default value</th>
<th>Uniform range</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_move_speed</td>
<td>float</td>
<td>420 km/day</td>
<td>(100, 500)</td>
</tr>
<tr>
<td>max_walk_speed</td>
<td>float</td>
<td>35 km/day</td>
<td>(10, 100)</td>
</tr>
<tr>
<td>camp_move_chance</td>
<td>float</td>
<td>0.001</td>
<td>(0.01, 0.1)</td>
</tr>
<tr>
<td>conflict_move_chance</td>
<td>float</td>
<td>1.0</td>
<td>(0.1, 1.0)</td>
</tr>
<tr>
<td>default_move_chance</td>
<td>float</td>
<td>0.3</td>
<td>(0.1, 1.0)</td>
</tr>
<tr>
<td>camp_weight</td>
<td>float</td>
<td>2.0</td>
<td>(1.0, 10.0)</td>
</tr>
<tr>
<td>conflict_weight</td>
<td>float</td>
<td>0.25</td>
<td>(0.1, 1.0)</td>
</tr>
</tbody>
</table>

Table 3
List of available conflict scenarios for sensitivity analysis.

<table>
<thead>
<tr>
<th>Conflict country</th>
<th>Conflict name</th>
<th>Simulation duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mali</td>
<td>mali</td>
<td>300 days</td>
</tr>
<tr>
<td>Burundi</td>
<td>burundi</td>
<td>396 days</td>
</tr>
<tr>
<td>South Sudan</td>
<td>sudan</td>
<td>604 days</td>
</tr>
<tr>
<td>Central African Republic</td>
<td>car</td>
<td>820 days</td>
</tr>
</tbody>
</table>

Table 4
List of tasks commonly used in FabFlee.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Brief description &amp; Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>flee / pflee</td>
<td>Executes a conflict scenario for the input simulation period.</td>
</tr>
<tr>
<td>pflee ensemble</td>
<td>Executes the ensemble of simulation runs for a conflict scenario with the input simulation period.</td>
</tr>
</tbody>
</table>

Step 3: Execution

To execute sensitivity analysis on your local PC, using FabFlee, simply run:

```bash
fbsim localhost flee_init_SA:<conflict_name>, simulation_period=<number>
```

In Table 5, we present several conflict scenarios available in forced migration application. Simply replace `conflict_name` and `<number>` in `simulation_period` to execute and perform sensitivity analysis. To illustrate, simply run the following:

```bash
fbsim localhost fleeing_SA:mali, simulation_period=300
```

After the job has finished, the terminal becomes available again, and a message is printed indicating where the output data resides. Run the following command to copy back results from the localhost results directory (or remote machine):

```bash
fbsim localhost fetch_results
```

The results will then be in `<FabSim3_dir>/results` directory.

Step 4: Results and analysis

To analyse and plot the obtained results, simply type:

```bash
fbsim localhost flee_analyse_SA:<conflict_name>
```

If you set `sampler_name: SCSample` in `flee_SA_config.yml`, the target folder name will be `flee_SA_SCSampler`. All output results will be saved in `<FabSim3_dir>/plugins/FabFlee/SA/flee_SA_SCSampler`. We will also find two figures automatically created from the obtained results. To illustrate, Fig. 2 is the first-order Sobol sensitivity indices for the selected parameter set in `flee_SA_config.yml` file and Fig. 3 is the mean and the standard deviation of total error over the simulation period. We observe that `max_move_speed` is highly sensitive input parameter and influential to the simulation output of Mali conflict compare to `max_walk_speed` parameter.

2.2. The required resolution of certain model parameters

Increasing the resolution (or polynomial order) results in a larger number of simulation runs, which may give us better estimation of sensitivity analysis on target parameters. However, in turn, it increases the final computational cost of executing the model. In case of the
migration application, we tested a set of certain parameters with different polynomial orders to evaluate the asymptotic behaviour in the quantities of interest (QoIs) upon increasing the resolution. The execution time for runs varied due to the increasing number of polynomial order. Fig. 4 compares the Sobol indices per each resolution size of the uncertain parameters. As it can be observed, after a certain polynomial order, such as polynomial order of 7, the sensitivity of input parameters did not change significantly. This will be helpful for future analysis to reduce the computational cost and total execution of the analysis.

3. Fusion tutorial

Nuclear fusion powers the sun and the main goal of fusion research is to bring this down to earth. One of the approaches taken is to use magnetic fields to confine a sufficiently large plasma for long enough so that more energy is produced from the fusion of deuterium and tritium isotopes of hydrogen than is required to heat and confine the plasma. The main process determining the confinement time is the turbulent transport of particles and energy in the plasma. To gain a better understanding of this, a fusion workflow (described in more detail in [23–26]) has been developed. To understand the role played by various sources of uncertainty, a number of workflows based on the above fusion workflow have been developed:

1. a workflow without UQ involving 4 codes (equilibrium code, turbulence code, code for converting fluxes to transport coefficients and transport code) coupled via MUSCLE.
2. workflows applying EasyVVUQ to particular components (equilibrium code, turbulence code, transport code).
3. a workflow with UQ using EasyVVUQ involving 3 codes (equilibrium code, code for calculating transport coefficients (not using turbulence) and a transport code) directly coupled.

While the ultimate goal is to apply the knowledge based in 3 above to doing UQ on 1, this would currently be too expensive (simple extrapolation would require approximately 35 million node hours) if the technique used for 3 were to be directly applied to 1. Under current investigation is to see if information gained from 2 can be used to speed up the UQ for the 1 workflow.

Building a tutorial around any of the above workflows is difficult because of code licensing issues and so a simpler model was created to explore some of the ideas underlying these workflows. In this, the toroidal plasma is replaced by a cylindrical model which simplifies the calculation of the equilibrium and associated metric coefficients. The turbulence code is replaced by a single uncertain number specifying the transport coefficient, and rather than solve for densities and electron and ion temperature equation, the density is fixed and only a single temperature equation is solved.

3.1. Uncertainty quantification on the fusion research

In this tutorial, we will use EasyVVUQ [7] to perform UQ [6] on an example taken from fusion research, which consists of

- easyvvuq_fusion_tutorial.ipynb: Jupyter notebook containing the EasyVVUQ workflow.
- fusion.template: template used by the EasyVVUQ to prepare the input files for the fusion program.
- fusion_model.py: a python program that reads the input file prepared by EasyVVUQ (based on fusion.template) and then calls the actual fusion function.
- fusion.py: a python program containing the function that performs the actual calculation using the fipy python package [27].

(While not really necessary to separate fusion_model.py and fusion.py, the latter has a life outside of this project and is therefore
The simplified fusion workflow maps the tokamak torus to a circular plasma (with a correction for ellipticity), see Fig. 5.

The model solves for the temperature, $T(\rho,t)$, across the cross-section of the cylinder, $\rho$, in the presence of a specified thermal diffusivity and sources:

$$\frac{3}{2} \frac{\partial}{\partial t} [n(\rho,t)T(\rho,t)] = \nabla \cdot [n(\rho,t) \chi(\rho,t) \nabla T(\rho,t)] + S(\rho,t)$$

with a boundary condition given by $T_{bc}$ and an initial uniform temperature of 1000 eV; the quantities are $n(\rho,t)$, the plasma density; $\chi(\rho,t)$, the thermal conductivity and $S(\rho,t)$ the source.

The geometry of the simulation is parameterised by the minor radius $a_0$, major radius $R_0$ and elongation $E_0$ (while the geometry is solved in the cylindrical approximation, the actual radius used, $a$, is adjusted on the basis of $a_0$ and $E_0$).

The density $n(\rho_{norm})$ is given by

$$\frac{b_{\text{height}} - b_{\text{tot}}}{2} \left( \tanh \left( \frac{b_{\text{max}} - \rho_{\text{norm}}}{2 b_{\text{width}}} b_{\text{slope}} \right) + 1 \right) + b_{\text{tot}}$$

where $b_{\text{height}}$ is the density at the top of the pedestal; $b_{\text{tot}}$ is the density at the base of the pedestal; $b_{\text{max}}$ is the position of the pedestal; $b_{\text{width}}$ is the pedestal width and the modified tanh function ([28] which cites [29]):

$$\tanh(x, b_{\text{slope}}) = \frac{(1 + x b_{\text{slope}}) \exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

A typical density profile used in these simulations is shown in Fig. 6. The source is given by

$$S(\rho,t) = a \exp \left( - \left( \frac{\rho/a - H_{\chi}}{H_{\omega}} \right)^2 \right)$$

where $a$ is chosen so that $\int S(\rho,t) dV = Q_{\text{tot}}$, the total heating power. In this application of the model we will be looking for the steady-state solution.

The parameters that can be varied are given in Table 4; though we will restrict the variation to that shown in Table 5 (corresponding to the vary_5 case mentioned later, or the first and last entries in that table for the vary_2 case).

**Step 1: Installation**

The starting point for the fusion tutorial is the following Binder link: https://mybinder.org/v2/gh/UCL-CCS/EasyVVUQ/dev?filepath=tutorials.

Once the Jupyter Notebook has started, click on easyvvuq_fusion_dask_tutorial.ipynb which should start the notebook.

![Fig. 5. The actual tokamak geometry (here JET [https://www.euro-fusion.org/devices/jet/] on the left) is mapped to a cylinder (on the right) in the simple fusion workflow.](image)

**Step 2: Parameter exploration**

For this model, 13 parameters are available to be set. In the notebook a few selections have been made consisting of 2, 5 and 10 in addition to the complete set. These are identified as

- **vary_2**, a minimal test case varying the heating power and the temperature boundary condition;

![Fig. 6. From top to bottom, a typical density profile and the profile of the electron temperature profile predicted by the simple fusion model indicating the range of possible values arising from a variation in the heating and boundary condition.](image)

### Table 4

Quantities that can be varied in the fusion example.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Min</th>
<th>Max</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_{\text{tot}}$</td>
<td>1.0e6</td>
<td>50.0e6</td>
<td>2e6</td>
</tr>
<tr>
<td>$H_0$</td>
<td>0.00</td>
<td>1.0</td>
<td>0</td>
</tr>
<tr>
<td>$H_{\chi}$</td>
<td>0.01</td>
<td>100.0</td>
<td>0.1</td>
</tr>
<tr>
<td>$T_{bc}$</td>
<td>10.0</td>
<td>1000.0</td>
<td>100</td>
</tr>
<tr>
<td>$\chi$</td>
<td>0.01</td>
<td>100.0</td>
<td>1</td>
</tr>
<tr>
<td>$a_0$</td>
<td>0.2</td>
<td>10.0</td>
<td>1</td>
</tr>
<tr>
<td>$R_0$</td>
<td>0.5</td>
<td>20.0</td>
<td>3</td>
</tr>
<tr>
<td>$E_0$</td>
<td>1.0</td>
<td>10.0</td>
<td>1.5</td>
</tr>
<tr>
<td>$b_{\text{tot}}$</td>
<td>0.95</td>
<td>0.99</td>
<td>0.98</td>
</tr>
<tr>
<td>$b_{\text{height}}$</td>
<td>3e19</td>
<td>10e19</td>
<td>6e19</td>
</tr>
<tr>
<td>$b_{\text{tot}}$</td>
<td>2e18</td>
<td>3e19</td>
<td>2e19</td>
</tr>
<tr>
<td>$b_{\text{width}}$</td>
<td>0.005</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>$b_{\text{slope}}$</td>
<td>0.0</td>
<td>0.05</td>
<td>0.01</td>
</tr>
</tbody>
</table>
• vary_5, a more interesting case varying the heating power source function (3 parameters), the transport coefficient and the temperature boundary condition;
• vary_10, adding in 5 quantities related to the density profile;
• vary_all, the complete set.

The number of cases required to be run for PCE scales as $\text{number of cores, memory, etc.}$

Step 3: Execution

For the initial tests, in the cell with the header

```python
# define varying quantities
return vary_2
```

which selects a minimal case to get results quickly and then move to vary_5. We will also scan over a range of PCE orders and look at the convergence of the statistical quantities.

Other parameters that can be changed are:

• whether DASK [12] is used, and if so whether locally or using SLURM [30];
• how many jobs to run in parallel

Step 4: Results and analysis

Typical output from the sensitivity analysis is shown in Fig. 6 where profiles of the electron density, $n_e$, and electron temperature, $T_e$, are plotted. Since no variations affected $n_e$ for this vary_5 case, only one line can be seen. The $T_e$ plot shows the mean, plus and minus one standard deviation, and the 10 and 90 percentiles. The range of $T_e$ between 1 and 99 % is also shown. The percentiles are calculated using the chaospy [31] Perc routine which samples from a distribution built on the basis of a fit by PCE to the local $T_e$ as a function of the uncertain, varying parameters, performed independently for each position across the $T_e$ profile.

The Sobol indices indicate (Fig. 7) that in the core ($\rho$ close to zero) the most important parameters are the width of the heating profile ($H_w$) followed by the transport coefficient ($\chi$); at the mid-radius of the plasma, the transport coefficient ($\chi$) is the most important parameter; and at the edge the boundary condition ($T_{e\text{bc}}$) dominates.

The convergence of the mean, standard deviation and Sobol first indices (Fig. 8) show a rapid convergence with PCE order indicating that for most purposes a PCE order of 3 should be sufficient for this problem.

This tutorial has just touched on a few issues but other options including changing from Polynomial Chaos Expansion to Stochastic Collocation, or the use of sparse grids, have not been covered.

4. Application of EasySurrogate: Lorenz 96

Multiscale systems are comprised of processes which span over a wide range of spatial and/or temporal scales. A direct numerical simulation of these systems, which resolves all relevant scales, is typically not possible due to computational constraints. A common engineering option is to decompose the solution into macroscopic and microscopic variables, after which a reduced model for the macroscopic variables is derived. The corresponding governing equations will be unclosed, meaning that they contain a so-called subgrid-scale term dependent upon microscopic variables. To close the system, the subgrid-scale term must be parameterized using macroscopic variables, in effect creating a surrogate model for the exact subgrid scale term. Classical approaches use deterministic parameterizations, see e.g. [32]. Data-driven surrogate models have more recently also become popular ([33], [34]), as well as a variety of machine-learning models ([35–37]).

Let the multiscale dynamical system be represented by a set of coupled nonlinear ordinary differential equations (ODEs) for the time-

![Fig. 7. Sobol first index describing the source of the variance in the profile of the electron temperature profile predicted by the simple fusion model.](image-url)
dependent macroscopic variables $x(t)$ and microscopic variables $y(t)$:

$$\frac{d}{dt}x = f(x,r), \quad \frac{d}{dt}y = g(x,y), \quad r = r(y). \quad (1)$$

Here, $r(y)$ is the subgrid scale term. Note that if we have a suitable surrogate $\tilde{r}$, we do not have to solve the equations of the expensive microscopic component:

$$\frac{d}{dt}\tilde{x} = f(\tilde{x}, \tilde{r}), \quad \tilde{r} = \tilde{r}(\tilde{x}). \quad (2)$$

In this tutorial, we will focus on neural-network surrogates $\tilde{r}$ which are (i) stochastic, and (ii) have memory. Specifically, we will use the EasySurrogate toolkit to build a model for the time evolution of $r$ by resampling training data from the distribution of $r_{i-1}$ ($r$ at time $t_{i-1}$), conditional on the past states of $x$. That is, we sample from the conditional distribution

$$\tilde{r} \sim r_{i-1} | x_i, x_{i-1}, \ldots, x_{i-\mathcal{L}},$$

where $\mathcal{L} \in \mathbb{N}$ is the maximum considered lag. We note that we do not need to have an explicit expression for the conditional distribution of $r_{i-1}$, we merely need to be able to sample from it. There are a variety of ways to do so, and we will focus here on so-called quantized softmax networks (QSNs). Essentially, we divide the domain of the $r_{i-1}$ training data into $B$ non-overlapping intervals, called ‘bins’. For each data point $r_{i-1}$ we can find the unique bin with index $k_{i-1} \in [1, \ldots, K]$ in which it falls, and we can create a corresponding time-lagged feature vector $X_{i-1} := [x_i, x_{i-1}, \ldots, x_{i-J}]^T$. Now, all $(X_{i-1}, k_{i-1})$ pairs form a classification data set, on which we train a feed-forward neural network, see Fig. 9.

The network has softmax output layers, which predict discrete probability mass functions (pmf) over the $K$ bins. During prediction, we sample a bin index from this pmf, conditional on the time-lagged feature vector $X_{t_i}$. The prediction $\tilde{r}_{i+1}$ is obtained by randomly sampling from the $r_{i+1}$ data inside the selected bin, see again Fig. 9. For more information behind this approach we refer to [38]. The authors discuss sampling from a more general conditional distribution than in Eq. (3), where also the past states of $r$ are included.

### 4.1. Lorenz 96

Before giving instructions on how to create a QSN surrogate, let us briefly introduce the model on which we will test the approach. Specifically, we will use the the well-known two-layer Lorenz 96 (L96) system, originally proposed by [39] as a toy model for the atmosphere. It consists of a set of $K$ ODEs describing the evolution of the macroscopic variables $x_k$, of which each ODE is coupled to $J$ microscopic variables $y_j$:

$$\frac{dy_j}{dt} = x^{(k-1)}(y^{(k+1)} - x^{(k)} - F + x^{(j)})$$

$$r^{(k)} := \frac{h}{\mathcal{J}} \sum_{j=1}^{\mathcal{J}} y^{(j)}, \quad (4)$$

The macroscopic and microscopic variables $x^{(k)}$ and $y^{(j)}$ are considered variables on a circle of constant latitude, where the indices $k = 1, \ldots, K$ and $j = 1, \ldots, J$ denote the spatial location. Periodic boundary conditions are imposed, and we will use the following parameter settings: $\{J, K, F, h, \mathcal{L}, \mathcal{J}, 18, 20, 10. \ldots 2, 1, 0.5\}$. Note that the full system (equivalent to (1)), consists of $K \times J = 360$ coupled ODEs. Once we have a surrogate for $r^{(k)}$, the reduced system (corresponding to (2)), consists of $K = 18$ ODEs.

There are five main steps for running this tutorial, namely (i) installation of EasySurrogate, (ii) running the full model to generate training data, (iii) training the QSN surrogate, (iv) running the macroscopic model with the microscopic surrogate, and (v) post processing the results.

**Step 1: Installation**

As in the preceding sections, one option is to install via a Binder link, found in the README of [https://github.com/wedeling/](https://github.com/wedeling/)
EasySurrogate. However, training the QSN network can be very slow in the Binder environment. We therefore recommend to install locally via:

```
git clone https://github.com/wedeling/EasySurrogate.git
cd EasySurrogate
python3 setup.py install
```

For both options, the Jupyter notebook containing the tutorial is found in `tutorials/tutorial_paper_96_tutorial.ipynb`. Finally, in case the install step fails, `(pip install easysurrogate`) found in the Binder environment. We therefore recommend to install locally via:

```
git clone https://github.com/wedeling/EasySurrogate
```

**Step 2: Generate training data**

We start by creating an EasySurrogate campaign object:

```python
import easysurrogate as es
create an EasySurrogate campaign
campaign = es.Campaign()
```

EasySurrogate has a similar design structure as EasyVVUQ, in the sense that we start with creating an overarching Campaign object as shown above. This object will handle the data frame (in HDF5 format), and we will assign a particular surrogate method to the campaign later on. For now, we will instantiate a L96 solver object via:

```python
import L96 as solver
# Time step
dt = 0.01
# Create a solver for the two-layer lorentz 96 model
L96 = solver.L96(dt)
# Get the initial condition of X and the right-hand sides
# of the ODEs
X_0, f_t = L96.initial_conditions()
```

The main time loop, which simulates the full system (4), is given by:

```python
@simulation time
T_end = 1000.0
N = np.arange(0, N_end, dt)
# Start time integration
for t in enumerate(N):
    # Integrate the two-layer L96 model in time
    X, f_t = L96.timestep(X, f_t)
    # Update variables
    X_m, f_m = f_t,
    # Snapshot of macro state \( X \) and SOS term \( r \) at time \( t \)
    snapshot = (X_m, X, r, f_m, t)
    # Accumulate the training data while running
    campaign.accumulate_data(snapshot)
```

By passing the dict `accumulate_data()` to the `campaign`, we are accumulating data of the macroscopic states and the corresponding subgrid scale term inside the Campaign object. Once the time integration has finished, we can store all accumulated data to an HDF5 data frame via:

```python
campaign.store_accumulated_data()
```

This will open a filedialog window to specify a storage location. Alternatively, by passing `file_path` as a keyword argument, the HDF5 file is written directly to the specified file path.

**Step 3: Train a QSN surrogate**

The HDF5 data frame generated in step 2 is used as training data for a QSN surrogate \( r \), which we load the via:

```python
# Load HDF5 data frame
data_frame = campaign.load_hdf5_dataset()
```

Next, we will create a QSN surrogate object:

```python
r = np.zeros((196, K))
# Load a pre-trained surrogate
campaign = cs.Campaign(load_state=True)
# Start time integration
for t in enumerate(N):
    # Do not compute \( r \) the first \( n_{\text{max, lag}} \) steps
    if t > n_{\text{max, lag}}:
        # Predict the sgs term with the QSN surrogate
        r_t = campaign.predict(X_m)
        # Integrate the one-layer L96 model in time
        X, f_t = L96.timestep(X, f_t, r, n_{\text{max, lag}})
        # Update variables
        X_m, f_m = f_t,
        # Snapshot of macro state \( X \) and SOS term \( r \) at time \( t \)
        snapshot = (X_m, X, r, f_m, t)
        # Accumulate the training data while running
        campaign.accumulate_data(snapshot)
```

The function call `predict(X_m)` returns a random sample from \( r \sim r_{n_{\text{lag}}} | \hat{x}_1, \hat{x}_{1:n} \). Internally, the current macroscopic state \( X_n \) is appended to the feature vector, and the time-lagged history \( \hat{x}_1, \hat{x}_{1:n} \) is automatically updated. To couple the surrogate to the macroscopic solver, the L96 solver module is programmed such that when the argument \( x_{n} \) is passed, this vector is directly used as the subgrid-scale term, and therefore the microscopic ODEs are not solved. This solution

```
# create Quantized Softnn Network surrogate
surrogate = cs.methods.QSN_Surrogate()
```

Training the surrogate is done via:

```
# Create time-lagged features
lags = range(1, 10)
# Train the surrogate on the data
n_iter = 40000
surrogate.train(features, target, n_iter, lags=lags, n_bins=10, n_layers=5, n_neurons=256, batch_size=256, test_frac=0.5)
```

When we specify a `lags` keyword, time-lagged features vectors as displayed in Fig. 9 will be created. Since we have specified `range(1, 10)` for the \( X_n \) feature array, we are creating a surrogate with 9 lagged \( X \) vectors: \( r \sim r_{n_{\text{lag}}} | \hat{x}_1, \hat{x}_{1:n} \). We are creating a ‘non-local’ QSN surrogate here, which takes entire \( X \) vectors as input. Since each \( X \) vector consists of \( K = 18 \) entries, we will have an input layer of \( 18 \times 9 = 162 \) neurons. Through \( n_{\text{bins}}=10 \), we are dividing the domain of each \( r_{n_{\text{lag}}} \) entry up into 10 non-overlapping, equidistant bins. As the \( r_{n}\) vectors in the target array also contains \( K \) entries, a QSN surrogate is created with \( K \) softmax layers, i.e. every spatial point \( k \), has its own pmf with \( n_{\text{bins}}=10 \) discrete probabilities. The output layer therefore has \( 18 \times 10 = 180 \) neurons. For more detail on the QSN structure, and a discussion on local vs non-local surrogates, we refer to [38]. The remaining keywords, \( n\_layers \) and \( n\_neurons \) regulates the number of (hidden) layers and the number of neurons per hidden layer. The mini batch size used in the stochastic gradient descent (see e.g. [40]) is specified through `batch_size`. Finally, by setting `test_frac=0.5` train only on the first 50% of the training data, thus keeping the latter half separate as a test set.

Just as a sampler is added to an EasyVVUQ campaign, a (trained) surrogate is added to an EasySurrogate campaign via the `add_app` subroutine:

```
campaign.add_app(name='L96_campaign', surrogate=surrogate)
campaign.save_state()
```

The `save_state` saves both the campaign and the surrogate object to disk. Similar to `store_accumulated_data`, this is done via a file dialog window or a `file_path` argument.

**Step 4: Predict with a QSN surrogate**

Here, we will use the trained QSN surrogate as a source term in the macroscopic ODEs. This results in a small change in the main time loop:

```
r_n = np.zeros((196, K))
# Load a pre-trained surrogate
campaign = cs.Campaign(load_state=True)
# Start time integration
for t in enumerate(N):
    # Do not compute \( r \) the first \( n_{\text{max, lag}} \) steps
    if t > n_{\text{max, lag}}:
        # Predict the sgs term with the QSN surrogate
        r_n = campaign.surrogate.predict(X_m)
        # Integrate the one-layer L96 model in time
        X, f_t = L96.timestep(X, f_t, r_n, n_{\text{max, lag}})
        # Update variables
        X_m, f_m = f_t,
        # Snapshot of macro state \( X \) and SOS term \( r \) at time \( t \)
        snapshot = (X_m, X, r, f_m, t)
        # Accumulate the training data while running
        campaign.accumulate_data(snapshot)
```

The function call `predict(X_m)` returns a random sample from \( r \sim r_{n_{\text{lag}}} | \hat{x}_1, \hat{x}_{1:n} \). Internally, the current macroscopic state \( X_n \) is appended to the feature vector, and the time-lagged history \( \hat{x}_1, \hat{x}_{1:n} \) is automatically updated. To couple the surrogate to the macroscopic solver, the L96 solver module is programmed such that when the argument \( x_{n} \) is passed, this vector is directly used as the subgrid-scale term, and therefore the microscopic ODEs are not solved. This solution
works well in the case of the L96 model. For more complex coupling situations we can use the Multiscale Coupling Library and Environment, see Section 5 or go to github.com/wedeling/EasySurrogate for a tutorial on coupling a micro-scale surrogate to a macroscopic reaction diffusion model.

When the training completed, an initial time-lagged feature vector was created from the training data, and stored in the QSN surrogate object. In this example this is \( x_0, x_2, \ldots, x_0 \), which is consistent with the prediction of \( r_i \). For this reason we do not compute the QSN prediction \( r_n \) until we get to the 9-th time step.

Finally, we note that there are other surrogate methods available in EasySurrogate, e.g. standard feed-forward neural networks or kernel-mixture networks [41]. These have the same design, so these will work without modification to the code above, which allows for easy comparison of surrogate method performance.

**Step 5: Post processing**

Each surrogate method has its own analysis class, in analogy to the different EasyVVUQ samplers. We create a QSN analysis object via:

```python
# Create a QSN analysis object
analysis = es.analysis.QSN_analysis(campaign='surrogate')
```

Due to chaos, and the accumulation of error over time, we cannot expect to have path-wise exact trajectories, i.e. \( x(t) \neq \tilde{x}(t) \) in general. Instead, we wish to create a surrogate such that the time-averaged statistics of \( \tilde{x} \) are a good approximation of the statistics of \( x \). To estimate the probability density functions of \( x \) and \( \tilde{x} \), we use:

```python
# Load the training data
data_frame = data_frames.data_frame
x_ref = data_frame['ref/\tilde{x}']
# Load the prediction data
data_frame, qsn = campaign.load_hdf5.data
x_qsn = data_frame['qsn/\tilde{x}']
# Compute two kernel density estimates
x_dens_post, x_pdf_post = analysis.get_pdf(x_qsn, x_ref)
x_dens_init, x_pdf_init = analysis.get_pdf(x_ref, x_ref)
```

The corresponding figure can be found in Fig. 10, which displays a good overlap between the reference (two-layer) model pdf from (4), and the pdf of the one-layer model with QSN surrogate \( \tilde{r} \). In a similar fashion we can use `auto_correlation_function` and `cross_correlation_function` from the analysis object, to compute auto correlation functions, or the cross-corrleation function between neighbouring points. Fig. 11 compares the auto correlations functions of \( x \) and \( \tilde{x} \), as well as those from \( r \) and \( \tilde{r} \).

Finally, let us note that we trained the surrogate completely offline, i.e. on data alone, whereas we predict in an online fashion, where the surrogate is coupled to a system of ODEs. Note that these two situations are not the same, as the online case is subject to two-way interaction between the surrogate and the physical (macroscopic) system. For the L96 case we obtained very good results with a surrogate that was trained between the surrogate and the physical (macroscopic) system. For more complex problems it might well be true that the introduction of such a surrogate eventually causes a bias in the statistics, or perhaps even results in an unstable system. We are currently investigating the use of a second, online learning phase (developed by [42]), where we retrain a neural network while it is part of the physical system. The idea here is that the surrogate should be trained to perform well within the modelling environment, rather than training it to just represent the data well.

5. Application of MUSCLE3: the 3D in-stent restenosis (ISR3D)

Another application that we consider in this article is the 3D in-stent restenosis model (ISR3D) [43,44]. ISR3D is a multiscale simulation that mimics the process of post-stenting growth of the neointima in the artery. It consists of two submodels: an agent-based smooth muscle cell model (SMC) and a blood flow model using the Lattice Boltzmann method. The smooth muscle cell model takes in the computational geometry of the vessel and corresponding stent after deployment, and starts the dynamics of restenosis. This submodel models cell growth, proliferation and death on the scale of an hour. The wall shear stress is one of the crucial factor influencing the proliferation of smooth muscle cells as it decides the turn off of nitric oxide inhibition from endothelial cells. Therefore at each time step, the current computational geometry of the blood vessel is passed to the blood flow solver, Palabos [45] and wall shear stresses are fed back to the SMC model after the flow computation. There are three helper modules (referred as `mapper` in MUSCLE3) in between the SMC model and blood flow model assisting the transmission of data. They are voxelizer, distributor and collector. Fig. 12 shows the communication diagram between submodels and helper modules.

MUSCLE3 [46] is a multiscale coupling library which can be used to connect multiple submodels together. A MUSCLE3 multiscale simulation consists of several programs which run simultaneously, passing

---

**Fig. 10.** The pdfs of \( x_k \) (left) and \( r_k \) (right), for both the two-layer model (4) and the one-layer model with QSN surrogate.

**Fig. 11.** The auto correlation function of \( x_k \) (left) and \( r_k \) (right), for both the two-layer model (4) and the one-layer model with QSN surrogate.

**Fig. 12.** Communication scheme of the ISR3D model.

---

messages to each other either at the beginning or the end of a simulation, or on every iteration. Besides the submodels, a MUSCLE3 simulation contains helper modules which perform e.g. data conversion or load balancing. The structure and the parameters of a MUSCLE3 model are all described in one or more ymmsl files. These describe the structure of the multiscale model by listing the programs and the communication lines between them, as well as global and submodel-specific settings. We refer the reader to Appendix A for more details about how ISR3D uses MUSCLE3 to communicate. In this tutorial, we demonstrate how to perform an uncertainty quantification analysis on ISR3D with MUSCLE3.

5.1. Uncertainty quantification analysis on ISR3D with MUSCLE3

With the MUSCLE3 communication set up, we can perform a non-intrusive uncertainty quantification analysis of the ISR3D application. We present an uncertainty quantification analysis with a quasi Monte Carlo (qMC) method, and the details about generating the UQ instances and launching a SLURM job array with MUSCLE3 are demonstrated. We first start with the installation of ISR3D and MUSCLE3. Since the uncertainty quantification campaign described in this tutorial is computationally expensive, it is recommended to use HPC resources for its execution.

Step 1: Installation

Installing ISR3D entails building MUSCLE3 including its C++ support, and then ISR3D. Python 3.5 or later, CMake ver. 3.6.3 or later, a C++ compiler supporting C++14 (e.g. GCC 6 or later), and a compatible MPI library with C++ support (e.g. OpenMPI) must be available. The first step is to install the Python module for MUSCLE3:

```
pip3 install muscle3
```

Next, MUSCLE3’s C++ library needs to be downloaded, wget https://github.com/multiscale/muscle3/archive/0.4.0/muscle3-0.4.0.tar.gz unpacked,

tar -xf muscle3-0.4.0.tar.gz

and built:

cd muscle3-0.4.0

MUSCLE ENABLE_MPI=1 make

MUSCLE3 can then be installed, for example in ~/musc le3:

```
PREFIX=~/musc le3 make install
```

ISR3D can be obtained from GitHub:

git clone https://github.com/ISR3D/ISR3D.git

A build script for the current machine must then be made. For building locally, editing the existing build.linux.sh is easiest. In this file MUSCLE3_HOME must be set to the location where MUSCLE3 has been installed:

```
cd ISR3D
nano ./build.linux.sh
```

ISR3D can then be built by running the script:

```
./build.linux.sh
```

After the installation, one can find a sample generation script UQtutorial.py, a SLURM script job_array.sl and a postprocessing script postprocessing.py under the directory ISR3D/UQ/ in which all the codes of this UQ campaign are contained. To execute these scripts, we recommend you to create a new directory under ISR3D root directory with the name of the UQ campaign and copy the three scripts in, for instance:

```
mkdir -p ISR3D/Result/UQcampaign
cd ISR3D/Result/UQcampaign
cp ./.../UQ/UQtutorial.py.
```

Step 2: Sample generation

The ISR3D multiscale model simulates the tissue growth after stenting. We are interested in the influence of the four biological uncertain parameters on the neointimal growth of the restenosis process. The four parameters are the endothelium regeneration time, the threshold strain for smooth muscle cells bond breaking, the balloon extension area and the fenestration percentage in the internal elastic lamina. As mentioned before, these four uncertain parameters, like all the other inputs for the model are included in the ymmsl. To change the value of these uncertain parameters for each instance, we first read in the template ymmsl file with:

```
with open(input_path+input_ymmsl_filename, ’r’) as f:
    ymmsl_data = ymmsl.load(f)
model = ymmsl_data.model
settings = ymmsl_data.settings
```

The ranges of uncertainty are given in Table 6 and are assumed to be uniformly distributed. We apply the quasi Monte Carlo method with Sobol sequence [47–49] to sample the instances for uncertainty quantification. In this tutorial, we demonstrate a UQ campaign with 128 instances which requires approximately 500 core-hours for computation in total. The quasi Monte Carlo method and several other sampling strategies can be found in EasyVVUQ. Alternatively, one can generate the Sobol sequence via other existing libraries, i.e sobol-seq. SciPy [50], SobolEngine function in PyTorch [51]. To show a more flexible combination of the toolkits, we demonstrate the usage of an external library sobol-seq with MUSCLE3 in this uncertainty quantification campaign. A sample matrix \( A \in \mathbb{R}^{N \times D} \) is formed with elements \( a_{i,j} \) where \( i \) ranges from 1 to the number of samples \( N \), and \( j \) ranges from 1 to the dimension of uncertain inputs \( D \). This can be achieved by the code:

```
# Generate Sobol sequence range (0,1), save the file and transform to (min,max)
A = sobol_seq.i4_sobol_generate(num_uncer_para,N_sample)
A = dim_transform(A,ymmsl_uncertain_parameters)
```

Note that the sample matrices generated by these toolkits are generally in a normalised range from 0 to 1, hence, an additional step is needed to adapt the normalised sample matrix to the application-specific one. This can be achieved by creating a python dictionary with a set of keys, ‘parameter_name’, ‘max’ and ‘min’ with corresponding ranges of the uncertain parameters. The details can be found in the UQtutorial.py.

ISR3D simulation can be denoted as a function mapping the uncertain inputs to the QoD, which in this case is the lumen volume of the stented blood vessel \( y = f(A_i) \), where \( A_i = (a_1, a_2, \ldots, a_D) \) are the uncertain input vector (one row of the sample matrix \( A \)) of a UQ instance. To analyse the uncertainty of the model, the probability density distribution, as well as the mean \( \mathbb{E}[y] \approx \frac{1}{N} \sum_{i=1}^{N} f(A_i) \) and the variance

### Table 6

<table>
<thead>
<tr>
<th>Uncertain parameters</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>endothelium regeneration</td>
<td>10</td>
<td>20</td>
<td>day</td>
</tr>
<tr>
<td>balloon extension</td>
<td>0.5</td>
<td>1.5</td>
<td>mm</td>
</tr>
<tr>
<td>threshold strain</td>
<td>1.2</td>
<td>1.8</td>
<td>/</td>
</tr>
<tr>
<td>percentage of fenestration</td>
<td>0</td>
<td>10</td>
<td>%</td>
</tr>
</tbody>
</table>


3 https://pypi.org/project/sobol-seq/.
Var(y) ≈ \frac{1}{N} \sum_{i=1}^{N} f(A_i)^2 - \left( \frac{1}{N} \sum_{i=1}^{N} f(A_i) \right)^2 \text{ will be estimated.}

Once the samples of the qMC method have been generated, we can easily replace the value of the uncertain parameters with: settings ["Para_Name"] = Value and create a ymmml input file for each UQ instance in a loop:

```python
# Replace the corresponding value within the dict and output the file
os.mkdir(output_path+experiment_name+"/'"
checklist = ['x']

for x in range(N_samples):
    sample_path = output_path+experiment_name+"/'"/x",
    IC = IC(x)
    os.mkdir(sample_path)

# Generate file for ymmml
num_param = 0
for para in ymmml_uncertain_parameters:
    settings[para.get('name')] = float(a[n,num_para])
    num_param += num_param + 1

config = ymmml.configuration(model, settings)
with open(sample_path + "/'input stiffness_ymmml", 'w') as f:
    ymmml.save(config, f)
```

We generate an input file for each instance and save it in a directory named '_x' where 'x' notes the numbering of the instance.

Except for the ymmml input file, there are several other input files required by ISR3D. They are hosted on Zenodo. The tutorial script UQtutorial.py includes the code to download these input files and to broadcast them to each UQ instance directory.

The sample generation described above is also included in the UQtutorial.py. The number of samples can be adjusted by editing the parameter, NumSample. To execute the sample generation, simply type:

```bash
gedit UQtutorial.py
```

Note that ISR3D is a computationally expensive application, especially when we simulate the restenosis process at a realistic scale. The showcase we offer in this UQ tutorial is based on a tiny vessel. However, it still takes approximately 500 core hours to run 128 instances. To reduce the load, you can reduce the number of samples NumSample in UQtutorial.py and job_array.sl, but at a cost of the accuracy of uncertainty estimation.

**Step 3: Execution**

To execute the UQ campaign with a large number of instances, the SLURM job-array script job_array.sl is written. Within each job, the MUSCLE3 manager is launched via muscle_manager and followed by the submodel execution:

```bash
muscle_manager./input.ymmm1
ISR3D/build/smc-muscle-instance=smc &
ISR3D/build/voxelerizer-muscle-instance=voxelerizer &
ISR3D/build/distributor-muscle-instance=distributor &
ISR3D/build/collector-muscle-instance=collector &
mpirun -n 16 ISR3D/build/flow-muscle-instance=flow
```

where muscle_instance informs the MUSCLE manager which submodel defined in the ymmml input file corresponds to this executable. The & means that all submodels are ran and communicate simultaneously.

Before the submission of the SLURM script, some modifications are required to adapt the setting to your running machine/cluster. First, check your cluster’s configuration and set the correct partition name of your cluster. Second, adapt your own path to the MUSCLE3 C++ library and the way to activate the Python environment with the MUSCLE3 Python library. Third, load the MPI module that was used for the installation. Fourth, set the directory for each submodel. Lastly, adapt the number of threads and the number of processes for OpenMP and MPI. Note that the ISR3D submodels from each sample are preferably executed on one node exclusively to avoid communication problems. We recommend you to set both the number of threads for OpenMP and the number of processes for MPI to be the number of cores in this node, since the execution of the SMC and the flow models alternates. To launch the SLURM job array, simply type:

```bash
sbatch job_array.sl
```

**Step 4: Results and analysis**

After the computation, the data of QoI is recorded in a CSV file in the directory of each instance. The postprocessing.py script can help you collect the data and plot the probability density function as well as the mean and standard deviation of QoI over time:

```bash
python3 postprocessing.py
```

The figures are saved under the current directory. The probability density function of the vessel lumen volume at day 3, 6, 9, 12 and 15 after stent deployment is demonstrated in Fig. 13. The mean and the standard deviation of ISR3D output on the vessel lumen volume over time is shown in Fig. 14.

**6. Execution of QCG tools: urban air pollution**

Constantly growing society condensing already dense, large cities, results in an increase of the contamination emission. And with the poorer air quality, citizens become more prone to hazardous pollutants, which in turn causes health problems including premature deaths. This is why studying air quality by the means of scientific simulations is of importance to understand how hazardous contamination can be lowered if not eliminated.

Predicting air quality in urban areas is a challenging topic that requires a trade-off between the accuracy of results and acceptable time-to-solution. There are numerous models for predicting contamination transport and dispersion, ranging from fast, computationally cheap but not necessarily accurate, e.g. simple Gaussian models, to quite accurate simulations resolving difficulty of the flows around buildings, but computationally expensive, e.g. computational fluid dynamics simulations. UrbanAir [6] aims at the latter in terms of quality of the results, and at the former with respect to the computational expense.

The quality of the results depends on the proper formulation of the model and the quality of input data. Modelling air quality requires an accurate emission database that contains emission rates for different pollutants and different types of sources, including line (attributed to road transportation) and area (attributed mainly to house heat appliances). UrbanAir is able to predict NO2/NOx, SO2 and two types of

---


---

Fig. 13. The probability density function of the vessel lumen volume at day 3, 6, 9, 12 and 15 after stent deployment.
Defining an input parameter space for the uncertain parameters of the UrbanAir application is crucial. The UrbanAir application uses uncertainty quantification analysis to understand which input parameters have a higher impact on simulation results, thus are required to be analysed. To solve these shortcomings, the UrbanAir application uses uncertainty analysis, e.g. ratio between engine types or hot/cold engine start. While some of them can be estimated quite well, e.g. the number of cars or fuel density, some are like a puzzle, e.g. ratio between engine types or hot/cold engine start. To solve these shortcomings, the UrbanAir application uses uncertainty quantification analysis.

In this section, we will demonstrate how to perform a demonstration assessment of air quality over one of the largest cities in Poland, Poznan, and perform a sensitivity analysis of the input parameters. We will also demonstrate how to use EasyVVUQ with QCG-PilotJob [1] in a working station/laptop environment, as well as on an HPC machine.

### 6.1. Sensitivity analysis on input parameters of UrbanAir

In the view of missing or incomplete emission data, sensitivity analysis plays a crucial role in deciding which input parameters have a higher impact on the simulation results, thus are required to be analysed for each run. In the assessment of air quality over complex urban areas, there are a lot of input parameters which are not known or not well recognised, and they differ with respect to analysed contamination. To understand which input parameters have a higher impact on simulation results, the sensitivity analysis (SA) approach is used. It allows for better estimation of the results, but also for a significant decrease in computation power required to perform necessary calculations.

In this tutorial NO\textsubscript{2} concentration attributed to road transportation is considered. The uncertainty comes from unknown number of vehicles, ratio between gasoline and diesel engines, fuel usage, NO\textsubscript{2} index related to type of engine, hot vs. cold engine start, etc. The input parameters required to predict air quality are described in the list below, while ranges are illustrated in Table 7:

- **no_of_cars**: Number of cars passing within 1 hour.
- **gas_cars_ratio**: Ratio between gasoline to diesel engines.
- **gas_usage**: Gasoline usage per 100 km.
- **gas_density**: Density of the gasoline fuel.
- **gas_no2_index**: NO\textsubscript{2} index related to gasoline engines.
- **oil_usage**: Oil usage per 100 km.
- **oil_density**: Density of the oil fuel.
- **oil_no2_index**: NO\textsubscript{2} index related to diesel engines.

#### Table 7

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Type</th>
<th>Default value</th>
<th>Uniform range</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_of_cars</td>
<td>integer</td>
<td>600</td>
<td>(200, 1200)</td>
</tr>
<tr>
<td>gas_cars_ratio</td>
<td>float</td>
<td>0.72</td>
<td>(0.1, 0.9)</td>
</tr>
<tr>
<td>gas_usage</td>
<td>float</td>
<td>9.0 l/100 km</td>
<td>(5.0, 12.0)</td>
</tr>
<tr>
<td>gas_density</td>
<td>float</td>
<td>0.75</td>
<td>(0.1, 0.9)</td>
</tr>
<tr>
<td>gas_no2_index</td>
<td>float</td>
<td>0.00855</td>
<td>(0.001, 0.012)</td>
</tr>
<tr>
<td>oil_usage</td>
<td>float</td>
<td>7.0 l/100 km</td>
<td>(4.0, 12.0)</td>
</tr>
<tr>
<td>oil_density</td>
<td>float</td>
<td>0.07</td>
<td>(0.15, 0.95)</td>
</tr>
<tr>
<td>oil_no2_index</td>
<td>float</td>
<td>0.008</td>
<td>(0.001, 0.015)</td>
</tr>
</tbody>
</table>
The command will place tutorial files in \$HOME/urban-air_tutorial directory.

**Step 2: Parameters exploration**

For this tutorial up to 8 input parameters can be sampled. In order to change their default, min or max values please navigate to the directory with downloaded tutorial and edit `urbanair_pj_executor_SC.py` accordingly. For example, to change the values of `gas_usage` parameter edit the following lines:

```python
params.update(
    "gas_usage": {
        "type": "float",
        "min": 8.0,
        "max": 13.0,
        "default": 8.0
    }
)
```

**Step 3: Execution**

For the efficient execution of highly demanding and large-scale calculations on HPC machines, VECMAtk proposes the QCG-PilotJob tool. In order to enable easy usage of QCG-PilotJob from EasyVVUQ a dedicated EasyVVUQ-QCGPJ (EQI) library has been also provided [1]. Within this tutorial we make use of both EQI and QCG-PilotJob.

The UrbanAir tutorial can be run on laptops, workstation or HPC cluster. In either case we assume that the Singularity image is used in a shell mode. Required input data for the application is located in the \$HOME/urbanair_tutorial directory. Please navigate to that directory by typing:

```
cd $HOME/urbanair_tutorial
```

If you changed the directory of virtual Python environment or location of UrbanAir tutorial files, please edit `easypj_config.sh` to reflect modified paths:

```
#!/bin/bash
# Change this to point to Python virtual environment
# $HOME/urbanair_env/bin/activate
this_dir="$( cd \$ (dirname "$ (BASH_SOURCE[0])") \>/dev/null 2> / dev/null && pwd )"
this_file="$ (BASH_SOURCE[0])"
PYTHONPATH="$ (PYTHONPATH):$ (this_dir)"
ENCODER_MODULES="cell_encoder"
export PYTHONPATH
export ENCODER_MODULES
```

If you changed the directory of virtual Python environment or location of UrbanAir tutorial files, please edit `easypj_config.sh` to reflect modified paths:

```
# Change this to point to easypj_config.sh script
export EASYPJ_CONFIG=$ (HOME)/urbanair_tutorial/
easypj_config.sh
```

**Step 4: Results and analysis**

Each ensemble (sample) execution is proceeded with the simulation results post-processing before passing them to the VECMAtk analysis phase. The post-processing is done by `prepare_hdf5.py` Python script, which aims at extracting emission output data at 2 m height for the sensitivity analysis. If other height is preferable, please update the aforementioned script accordingly:

```
#height
# 1 = 2m
# 2 = 4m
# 3 = 6m
# etc
h = 1
```

The UrbanAir application coupled to VECMAtk delivers sensitivity analysis of input parameters (`sobols_analysis.csv`), as well as emission concentration mean and standard deviation values for the whole domain at a given height (`stats.csv`).

The example mean and standard deviation of the NO\textsubscript{2} concentration for a given point in 2D space and for different heights are presented in Fig. 15.

There is an exemplary Python script to visualise NO\textsubscript{2} concentrations at 2 m height. It iterates over all generated results and creates plot in 3D in `2m_no2_mean.png`. If you want to analyse results from 24 runs, located e.g. under `tmp/urbanair_no2/runs` just type:

```
python3.6 analyze_hdf5.py 24/tmp/urbanair_no2/runs
```

In Table 8, we present the sensitivity analysis of 8 input parameters in descending order.

Fig. 16 presents mean NO\textsubscript{2} concentration at 2 m height.

UrbanAir is coupled to VECMAtk which allows not only for sensitivity analysis, but first of all for automatic creations of required ensembles (samples), their execution and results collation. Assessing air quality in urban areas is computational expensive, that is why a user is able to select how many input parameters are to be sampled:

```
# 1 = no of cars
# 2 = gas_cars
# 3 = gas_usage
# 4 = gas_density
# 5 = gas_no2_index
# 6 = oil_usage
# 7 = oil_density
# 8 = oil_no2_index
```

By default, just one input parameter – `no_of_cars` is sampled. To run the UrbanAir application with more parameters being sampled, e.g. first five, just type (in `$HOME/urbanair_tutorial` directory):

```
python3.6 urbanair_pj_executor_SC.py 5
```

The UrbanAir example uses 4 CPU cores by default, but more can be used to allow running more samples in parallel. In case more CPU cores are available, please type:

```
python3.6 urbanair_pj_executor_SC.py 8 24
```

In this case, all 8 input parameters will be sampled on 24 CPU cores, which means 6 samples will be analysed in parallel. In case less than four CPU cores are available, still you can run the UrbanAir demo. However please mind the UrbanAir is suited for larger runs – on a modern CPU, equipped with 4 cores, a single sample run would take several minutes up to half an hour. Thus expect that a very basic demo, sampling just one parameter, would last for less than an hour.

Fig. 15. Emissions of NO\textsubscript{2} at different heights above a street-level from road transportation, with the mean (red line) and standard deviation (blue region) calculated using EasyVVUQ. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)
VECMAtk can facilitate VVUQ for users, irrespective of their scientific domain, and how it can be re-used in a variety of ways.

The VECMA toolkit is under continuous evolution, as several dozen of alpha users provide us with feedback on what to improve in terms of robustness, scalability and ease of use. The tutorials in this paper therefore serve as a snapshot of the current VECMA development, and as useful examples which can be adapted to suit different purposes.

Verification, validation, sensitivity analysis and uncertainty quantification are essential for simulation results to become relevant outside their base field of research, and eventually suitable for practical decision-taking. With these tutorials we show that VVUQ techniques can be efficiently repurposed from one domain into another, and quickly adopted with clear benefits without the need to modify underlying source codes. In addition, the techniques provided here scale to larger problems: though an analysis on the local laptop is in many cases possible, most of the examples scale just as well to petascale and emerging exascale supercomputers.

Declaration of interests

None.

Declaration of Competing Interest

The authors report no declarations of interest.

Acknowledgements

This work was supported by the VECMA project, which has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No. 800925. The development of MUSCLE3 and its respective description was supported by the Netherlands eScience Center and NWO under the e-MUSC project. The development of IR3D was supported by the InSilic project and the In Silico World (ISW) project (European Union Horizon 2020 research and innovation programme grant agreements #777119 and #101016503 respectively). The calculations were performed in the Poznan Supercomputing and Networking Center.

Appendix A. Structure of the ISR3D model and communication with MUSCLE3

The structure and the parameters of a MUSCLE3 model are described in one or more ymmsl (multiscale modeling and simulation language) files. ISR3D uses a single file which is located at ISR3D/cxa/input_stage4.ymmsl. This file contains a header describing its version, and two main sections: model and settings. The first section describes the structure of the multiscale model by listing the executables and the communications between them, while the second contains the parameters that are either shared by the whole simulation or passed to individual modules.

We will focus on the model section first. The first line of this section gives the name to be used for the multiscale simulation by MUSCLE3, in this case name: isr3d. Next comes the subsection components: which lists all the executables involved. For ISR3D, they are:

- smc, the submodel containing smooth muscle cells and other components of the vessel wall;
- voxelizer, which takes the point cloud data from the smc and produces a voxel vessel wall where each tissue type is marked separately;
- distributor that adapts the voxel geometry to be used for flow calculation and distributes it to the flow solver module and also to the collector module;
- flow module that resolves the steady-state flow for the changing vessel geometry and passes the solution back to collector;
- collector, which collects the data from the flow model and the other helper modules, and maps the flow solver output back onto the cells in the smc model.

The section after that, conduits: lists the connections between the single-scale modules. The general syntax is

```
source.sending_name: target.receiving_name
```

This means the data is sent from source as sending name, and received by target as receiving name. The specifics such as the frequency of sending and receiving, and also the data format have to be described in the executables’ code itself. The communication scheme of ISR3D is shown in Fig. 12 and reflected in this section of the ymmsl file.

---

5 https://github.com/multiscale/ymmsl-python.

Table 8

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Sobol first index</th>
</tr>
</thead>
<tbody>
<tr>
<td>gas_no2_index</td>
<td>0.14294</td>
</tr>
<tr>
<td>gas_density</td>
<td>0.13251</td>
</tr>
<tr>
<td>oil_no2_index</td>
<td>0.11094</td>
</tr>
<tr>
<td>oil_density</td>
<td>0.10523</td>
</tr>
<tr>
<td>no_of_cars</td>
<td>0.0831</td>
</tr>
<tr>
<td>gas_usage</td>
<td>0.0592</td>
</tr>
<tr>
<td>oil_usage</td>
<td>0.0591</td>
</tr>
<tr>
<td>gas_cars_ratio</td>
<td>0.00184</td>
</tr>
</tbody>
</table>

Fig. 16. Mean emission of NO₂ at 2 m height from road transportation.

7. Conclusion

We have presented a set of five tutorials that showcase how VECMA can help users to create and execute complex HPC workflows and to simplify verification, validation and uncertainty quantification activities for their applications. Each tutorial has been tested by users from other domains, and has been presented such that it can be performed using any HPC infrastructure with sufficient capacity.

The tutorials cover five different application domains, and five different combinations of VECMA components, to showcase how
For example, the following conduit handles the sending of the voxel domain (a rectangular 3D grid of tissue type values) from voxelizer to distributor:

voxelizer.domainOut1: distributor.domainIn

On the side of voxelizer, the following code is called:

```cpp
auto ad_result = Data::grid(aggregateDomain.data(),
    {sphereVox.sizeX, sphereVox.sizeY, sphereVox.sizeZ},
    {"x", "y", "z"});
instance.send("domainOut1", Message(t_cur, ad_result));
```

Initially, the results we want to send are stored in the aggregateDomain variable. To pass them to MUSCLE3, we construct an object of the MUSCLE3 type `Data::grid`, which we call `ad_result`. The arguments are: the raw data to be stored (`aggregateDomain.data()`), a list of grid sizes along each axis, which we receive here from the sphereVox object, and an optional list of axis labels. Naturally, `sizeX-sizeY-sizeZ` has to match the size of the data in the first argument. When the data object is formed, all that is left to do is to form a `Message` by combining the data with a timestamp `t_cur` and to send it, specifying the same sending name as in the configuration file, domainOut1.

For each send operation, there has to be a receive operation in the other executable. The receiving code in the distributor is shown below:

```cpp
auto domainObs = instance.receive("domainIn");
t_cur = domainObs.timestamp();
std::vector<size_t> boxSize = domainObs.data().shpux();
size_t dOsize = domainObs.data().size();
auto domOutStr = domainObs.data().elements<int32_t>();
std::vector<int32_t> aggregateDomain;
aggregateDomain.assign(domObsStr, domObsStr + dOsize);
```

The message is received by calling the MUSCLE3 function `instance.receive` with the receiving name specified in the configuration file. This produces a MUSCLE3 `Message` on the receiving side, from which it’s possible to obtain the timestamp, the shape (three sizes for the three axes specified on the sending side), and the size (the total number of elements in the grid). Since C++ does not have a standard multidimensional grid type, we convert the received data into a 1D array.

To convert the MUSCLE3 grid to an `std::vector`, we have to assign the data directly by manipulating memory pointers (obtained by calling `elements<int32_t>();`).

For most non-grid message types direct memory manipulation is not required, and C++ types can be directly obtained. MUSCLE3 also has support for Python and Fortran languages. For more details on this, we refer the reader to the online MUSCLE3 documentation.

The final section of the configuration file is the `settings: section`. The settings without a prefix are accessible to all modules, and the settings with a prefix are only accessible from the matching executable. The `settings: section` also has the following syntax:

```cpp
const std::string inputFile = "test_vessel.dat";
instance.get_setting_as<std::string>("run_input_file");
```

Here `instance` is the MUSCLE3-specific object assigned to the `smc` submodel.

References


Wouter Ededing is a researcher at CWI Amsterdam, and has a background in Uncertainty Quantification applied to turbulence models. He received a joint PhD in 2015 between Arts et Métiers ParisTech and Delft University of Technology, and held a post-doc position at Stanford University. His current interests lie at the intersection of Uncertainty Quantification, Machine Learning and Scientific Computing, and his email address is Wouter.Edeling@CWI.nl.

Dr. Vytautas Jancauskas got his PhD from Vilnius University in 2016. The subject of the thesis was evaluating the performance of multi-objective optimization methods. He worked as a lecturer and assistant lecturer for more than 3 years. Supervised undergraduate computer networks course at the Faculty of Mathematics and Informatics at Vilnius University. Taught computer architecture, operating systems design and C programming. He has worked in EU funded projects, such as ComPat and VECMA. He was one of the lead developers of the EasyVVUQ framework. Main research interests lie in the intersection of computer science, software engineering and various statistical and optimization methods. His email address is jancauskas@lrz.de.

Onnie L. Luk is a postdoctoral researcher at the Numerical Methods for Plasma Physics division of the Max-Planck-Institute for Plasma Physics at Garching. She is a member of the Verified Exascale Computing for Multiscale Applications (VECMA) project, and past member of the Computing Patterns for High Performance Multiscale Computing (ComPat) project, both of which are funded by the European Union’s Horizon 2020 research and innovation programme. Her current research topics include exploration of time bridging methods between turbulence and transport models in a component-based multiscale fusion plasma workflow and validation of the simulation results. She received her Ph.D. from the University of California at Irvine, where she conducted her research on the role of convective cell in nonlinear interaction of kinetic Alfvén waves. Her email address is onnie.luk@ipp.mpg.de.

Jalal Lakhlili is a computational scientist in Post-doc at Max-Planck Institute for Plasma Physics in Munich, Germany. Has a PhD in Applied Mathematics from Toulon University, France, and an engineer degree in Computer Science from Grenoble INP, France. Has worked in EU research projects EcoColl and VECMA. Main research areas are numerical simulations with applications in HPC, including algorithmics, numerical analysis, uncertainty quantification and statistics. His email address is jalal.lakhlili@ipp.mpg.de.

Dongwei Ye is a PhD candidate in the Computational Science Lab at the University of Amsterdam. He is working on surrogate modelling, uncertainty quantification and computational biophysics as part of Verified Exascale Computing for Multiscale Applications (VECMA) project funded by the European Union Horizon 2020 research and innovation programme. His email address is: d.ye@uva.nl.

Lourens Veen is a Senior eScience Research Engineer at the Netherlands eScience Center. As an engineer, his background is originally in databases and information system architecture, including geographical information systems, and more recently he has worked in High Performance Computing and networking. As a scientist, his skills are in modelling, multiscale models, model coupling, parameter optimization and Uncertainty Quantification, with applications in biogeography, molecular simulation and computational biophysics. His email address is l.veen@esciencecenter.nl.

Pavel Zun is a postdoctoral researcher at the University of Amsterdam (UvA). He received his PhD in 2019, and has since then worked on the iBase (In-silico trials for drug-eluting BVS design, development and evaluation) project at Erasmus Medical Center, and on the VECMA project at UvA. His research interests are in multiscale modelling of the cardiovascular system, and in agent-based tissue models. His email is p.zun@uva.nl.

Dr. Valeria Krzhizhanovskaya is an editor-in-chief of the Journal of Computational Science, working in the University of Amsterdam and leading the development of algorithms and formalisms in the VECMA project. Valeria is an expert in computational science with 25 years of experience in simulation of multiscale complex systems, data-driven modelling, high-performance parallel and distributed computing. Valeria has published over 150 scientific papers and edited many volumes of journal special issues and conference proceedings. She is a co-organizer of the annual International Conference on Computational Science and thematic workshops on Multiscale Modelling and Simulation. Her email address is v.krzhizhanovskaya@uva.nl.

Michal Kulczewski is a senior specialist at Poznan Supercomputing and Networking Center. His research interests include mainly numerical weather prediction, modeling air quality in urban environments, code optimization, programming multi- and many-core computing systems, advanced in-situ data analysis and visualization. He has actively contributed to many EU and national projects for the past 16 years. His email address is michal.kulczewski@man.poznan.pl.

David Coster has been active in the field of magnetized plasma physics for over 35 years with over 200 published papers as author or co-author. His PhD from Princeton University dates from 1993, and since then he has been employed by the Max Planck Institute for Plasma Physics. His email address is david.coster@ipp.mpg.de.

University of California at Irvine, where she conducted her research on the role of convective cell in nonlinear interaction of kinetic Alfvén waves. Her email address is onnie.luk@ipp.mpg.de.

Dr. Valeria Krzhizhanovskaya is an editor-in-chief of the Journal of Computational Science, working in the University of Amsterdam and leading the development of algorithms and formalisms in the VECMA project. Valeria is an expert in computational science with 25 years of experience in simulation of multiscale complex systems, data-driven modelling, high-performance parallel and distributed computing. Valeria has published over 150 scientific papers and edited many volumes of journal special issues and conference proceedings. She is a co-organizer of the annual International Conference on Computational Science and thematic workshops on Multiscale Modelling and Simulation. Her email address is v.krzhizhanovskaya@uva.nl.
Prof. dr. ir. Alfons G. Hoekstra is full professor of Computational Science and Engineering at the University of Amsterdam. He is also the director of the Informatics Institute at that university. His research focuses on multiscale modelling, actionable simulations, and high performance computing. His research is driven forward by and applied in the biomedical domain. His main interests are currently in multiscale modelling of hemodynamics with applications in cardiovascular diseases, and in the development and accreditation of in-silico trials. His email address is a.g.hoekstra@uva.nl.

Daan Crommelin leads the Scientific Computing group at CWI, Amsterdam, and has a part-time position as full professor in numerical analysis and dynamical systems at the University of Amsterdam. He received his PhD from Utrecht University and was a postdoc at the Courant Institute of Mathematical Sciences of New York University. His research is focused on complex dynamical systems and their uncertainties, and includes topics such as stochastic modeling for multiscale systems, uncertainty quantification, rare event simulation, data-driven modeling, and applications in climate science and energy systems. His email address is Daan.Crommelin@cwi.nl.

Prof Peter V. Coveney holds a chair in Physical Chemistry and is an Honorary Professor in Computer Science at University College London (UCL). He is a Professor in Applied High Performance Computing at the University of Amsterdam (UvA), and Professor Adjunct at Yale University School of Medicine (USA). He is Director of the Centre for Computational Science (CCS) at UCL. Coveney is active in a broad area of interdisciplinary research including condensed matter physics and chemistry, materials science, as well as life and medical sciences in all of which high performance computing plays a major role. His email address is p.v.coveney@ucl.ac.uk.

Derek Groen is a Senior Lecturer in Computer Science at Brunel. He received his PhD in 2010 from the University of Amsterdam, and was a PDRA at UCL for five years prior to joining Brunel as Lecturer. He has published 35 journal articles in diverse journals, worked on collaborative software and HPC projects for 16 years and currently leads the development of tools, such as VECMAtk, FabSim3, the Flee migration modelling code and the Flu And Coronavirus Simulator. He has direct application experience in turbulence, blood flow, materials and migration modelling and simulation, among other areas. His email address is derek.groen@brunel.ac.uk.