Excalibur-Neptune report 2047356-TN-08-1

Task 2.4 Non-intrusive UQ with BOUT++ 1D fluid solver

Ben Dudson, Peter Hill, Ed Higgins, David Dickinson, and Steven Wright

University of York

David Moxey

KCL

March 29, 2022

Contents

1 Executive summary

In this report we summarise experience gained in the application of non-intrusive uncertainty quantifiaction (UQ) to a sample test case from the SD1D [1] test cases set out in task 83-2.1 [2]. This provides a test ground for the practical aspects of UQ for somewhat realistic plasma simulations and showcases the *easyvvuq* package to faciliate the construction of UQ workflows.

2 Introduction

Simulations are a vital tool for developing understanding of systems and for extrapolating beyond existing facilities. This means that they can be used as

evidence towards significant decisions, such as substantial investment in product/facility development (e.g. STEP [3]). The output of such simulations will be sensitive to its inputs to varying degrees. Understanding how sensitive outputs are to inputs is known as sensitivity analysis and this can give an insight into how important it is to pin down the inputs to simulations. A related, but different, area is that of uncertainty quantification (UQ). Here, we are interested in the uncertainty on the output of simulations given some probability distribution function for the input parameters. This can play an important role in the verification and validation (VV) of computational models, improving our confidence in these models whilst also allowing more nuanced interpretation of the predictions the model makes. There are several approaches to uncertainty quantification. One split is between intrusive and non-intrusive. Applying intrusive approaches to an existing simulation code can require substantial modification of the code in order to effectively project the system state onto a "random trial basis" of size N and substitute this into the system of equations. This gives an increase in the effective number of equations to evolve in a single run by a factor related to N. Such approaches can be challenging to implement into an existing code due to the significant modifications potentially required. Non-intrusive UQ, on the other hand, does not require modification to the existing code but rather requires the user to effectively run multiple copies of the simulation with inputs varied according to their probability distribution.

One of the exciting opportunities offered by exascale computing is the increased throughput which will greatly enable our ability to run large ensembles of simulations. This will start to enable more routine study of sensitivity and uncertainty in code outputs due to inputs, by directly simulating a range of cases within the probable inputs. One challenge introduced by this is the need to analyse a large number of simulations in order to extract information about the expected results and associated uncertainty. Fortunately such analysis exhibits regular patterns across different domains and several generic toolkits exist to assist with implementing a UQ workflow with existing codes. Here we explore the *easyvvuq* toolkit [4, 5] developed through the VECMA project [6].

In essence, the UQ work flow can be summarised as being comprised of several distinct generic stages, with the breakdown used by *easyvvuq* outlined in figure 1. Discussion of these stages is provided in the *easyvvuq* documentation [7], so here we only give a brief summary of the key aspects. Firstly it is useful to

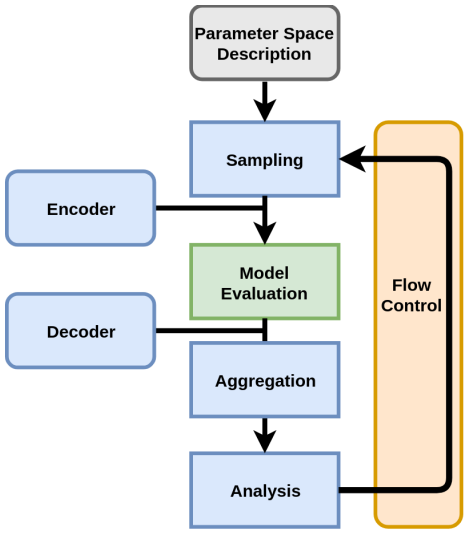


Figure 1: Breakdown of stages in a generic VVUQ workflow as considered by *easyvvuq*. Figure reproduced from [7].

note that *easyvvuq* introduces the concept of a Campaign. This is a class designed to coordinate a given UQ workflow instance and interfaces to a database stored on the filesystem recording information about the model system, the runs performed and other aspects of the workflow. To construct a campaign, the first step, "Parameter Space Description", is to describe the relevant model parameters along with their properties. This takes the form of a python dictionary with parameter names as the keys and then other relevant properties such as data type, default values and expected ranges forming a secondary dictionay as the value. This does not need to be a complete listing of all model parameters, but rather any parameters which one may wish the UQ tools to be aware of. This is most likely to contain the uncertain input parameters for which we wish to perform a UQ analysis. Alongside this information, the campaign must also be able to generate relevant simulation inputs, execute simulations using these inputs and extract the appropriate output of the simulations. This corresponds to the "Encoder", "Model Evaluation" and "Decoder" stages respectively and these are stored together within a *easyvvuq* "Actions" class . Whilst *easyvvuq* provides some example decoders useful for output stored in a few simple file types[1] it will often be necessary to write a specific decoder for the code at hand. Indeed, it is possible for this decoder to go beyond simply returning the raw output of the simulation and instead to perform any required post-processing,

analysis and data reduction stages before returning the relevant data of interest. Similarly, it will often be necessary to write a custom decoder for the particular input format of the code being used. This will be given a set of parameters with values and must be able to create the relevant input file(s) in a provided directory alongside any further setup (e.g. providing other files to be accessed by the code). It is also necessary to describe how to run the simulation. This can take several forms but here we will focus on the "local_execute" approach, in which we simply provide a list of strings with the command to be run and any options. More sophisticated approaches, including the submission of runs to a slurm queue-managed system are provided by *easyvvuq*. Given this information about how to construct, execute and extract the output of runs all that remains is to decide which runs should be performed and the collation and analysis of the series of runs generated. The decision of which simulation cases should be performed is controlled by a "Sampler" object. This can be a key choice in the construction of the UQ workflow and it can determine how many and which simulations are required as well as how the analysis is performed. We defer further discussion of this stage to section 5. Finally, analysis of the collection of simulations is automated by *easyvvuq* and leverages other packages such as ChaosPy [8], with the user simply needing to choose what properties to explore such as the mean and standard deviation of the result and the Sobol indices representing the relative importance of the different input uncertainities on the output quantity of interest.

3 Using easyvvuq with BOUT++ models

Here we briefly describe the approach taken within this work in order to use *easyvvuq* with BOUT++ [9] (and specifically SD1D [1]). Encoders and decoders for use with BOUT++ have been created [2] and have been made publicly available [10][3]. We use these encoders and decoders in this work. The input format for BOUT++ is a simple text file in "config" format and we note that the test cases here do not require any further inputs (i.e. no grid files). As

[1] Specifically json, csv and yaml.

[2] This work was supported through VECMA hackathons run through the financial year 2021/22 and the access to the VECMA team that this provided.

[3] We also note that there are severals examples of using *easyvvuq* with various models implemented with BOUT++ also provided in this repository. The SD1D examples in branch "sd1d" of [10] are closely related to the approach used here.

such, the encoder is relatively straightforward. Several decoders are provided to demonstrate a number of different scenarios, such as

1. Reading a quantity at the final time in the simulation. This is a relatively simple, but common use case.
2. Sampling a quantity onto a fixed grid. This can be helpful to allow comparison across cases in which the simulation grid may change, for example.
3. A decoder specific to the *blob2d* model, calculating and returning derived quantities rather than the raw output.

For the results shown later we make use of the first, simple, option. Finally, it is helpful to discuss the approach taken to executing simulations. As the test cases used here, based on *case-04* of SD1D, are relatively cheap to run on a single core we opt to use the "local_execute" approach and we will describe this in more detail in section 4. Clearly, for more expensive simulations one must look to use a distributed approach. Branch "slurm" of the repository [10] demonstrates one approach to the use of a slurm managed system.

4 SD1D configuration and general approach

We will use *case-04* of SD1D as an example case with which we can demonstrate an *easyvvuq* facilitated UQ workflow. This test case consists of coupled single plasma and neutral fluids in 1D with sheath boundary conditions and is described in more detail in the report for 83-2.2 [11]. We note that we use branch *bout-next* of SD1D at commit *7bd6bc91* and then build this using BOUT++ commit *080f3b27*[4]. We configure BOUT++ to enable its interface to PETSc [12] and link to a version of PETSc configured with HYPRE [13] support. This enables access to the "beuler" time solver, which is very effective when seeking steady state solutions, and the efficient preconditioners provided by HYPRE.

To demonstrate how this is implemented, we show example python code for running a small UQ analysis workflow for *case-02* of SD1D. In listing 1 we

[4]This is not the BOUT++ commit which will be used by default if SD1D is not supplied with an existing BOUT++ build.

show the creation of the encoder, decoder and local exectute action and how these are used to create a campaign. For completeness, listing 2 shows how one then creates a sampler, adds this to the campaign and launches the simulations, whilst listing 3 shows how this is used to create some simple plots showing the uncertainty on the plasma density, N_e, output. Figure 2 shows example outputs from these listings for second and sixth order PCE samplers. One can see that whilst the mean density agrees fairly well between the two orders, there is more significant disagreement between the 1^{st} and 99^{th} percentiles and the Sobol indices. Whilst the second order sampler required 9 separate simulations and the sixth order sampler required 49 simulations, the time to solution was broadly comparable here as we ran on a single node of Archer2. This provides 128 cores and the "local_execute" action provided by *easyvvuq* will create a thread pool which uses these cores to run the planned simulations in parallel. This demonstrates the useful ability to run a number of the identified simulations in parallel, dependent on available CPU resource. However, it is perhaps worth noting that this can lead to inefficient resource utilisation. For example, when running a large analysis the run time across the parameter space being sampled may vary somewhat. The analysis cannot be completed until all simulations are complete. As such, the total resource usage is dependent on the slowest simulation encountered and this is likely to become a more significant concern as the number of samples increases. In such scenarios a more suitable approach may be to use the slurm action instead.

```python
#!/usr/bin/env python3
import boutvecma
import easyvvuq as uq
import chaospy
import os
import numpy as np
import time
import matplotlib.pyplot as plt

# Path to the executable
EXE_PATH="./sd1d"

# Create an encoder to produce input files
encoder = boutvecma.BOUTEncoder(template_input="BOUT.inp")
# Create a decoder to extract data of interest (here just Ne)
decoder = boutvecma.SimpleBOUTDecoder(variables=["Ne"])
# Specify the parameters of interest
params = {
    "P:powerflux": {"type": "float", "min": 1e7, "max": 1e8, "default": 2e7},
    "Ne:flux": {"type": "float", "min": 1e23, "max": 1e24, "default": 4e23},
}

# Now we can create a local_execute action describing how to create, run and analyse
# a single run
actions = uq.actions.local_execute(
    encoder,
    os.path.abspath(
        EXE_PATH + " -d . solver:type=beuler input:error_on_unused_options=false -q -q -q"
    ),
    decoder,
)

# Create the campaign
campaign = uq.Campaign(name="SD1D_Case02_order3.", actions=actions, params=params)
```

Listing 1: Example python code to create a BOUT++ UQ campaign.

```python
# Describe the distribution of the uncertain inputs
vary = {
        "P:powerflux": chaospy.Uniform(1e7, 1e8),
        "Ne:flux": chaospy.Uniform(1e23, 1e24),
}

# Create a PCE sampler for this system of uncertain inputs
sampler = uq.sampling.PCESampler(vary=vary, polynomial_order=3)
# Attach sampler to campaign
campaign.set_sampler(sampler)

print(f"Code will be evaluated {sampler.n_samples} times")

# Actually run the simulations, wait for the runs to finish and collect outputs
time_start = time.time()
campaign.execute().collate(progress_bar=True)
time_end = time.time()
print(f"Finished, took {time_end - time_start}")

results_df = campaign.get_collation_result()
print(results_df)
results = campaign.analyse(qoi_cols=["Ne"])
```

Listing 2: Example python code to create a polynomial chaos expansion sampler, add this to an existing campaign and run and analyse the simulations.

```python
# Helper method to make standard plots
def make_plots(campaign, variable_name, xlabel = r'$\rho$', ylabel = None):

    # Plot results of interest
    moment_plot_filename = os.path.join(f"{campaign.campaign_dir}",
                                        variable_name + "_moments.png")
    sobols_plot_filename = os.path.join(f"{campaign.campaign_dir}",
                                        variable_name + "_sobols_first.png")

    fig, ax = plt.subplots()
    xvalues = np.arange(len(results.describe(variable_name, "mean")))
    ax.fill_between(
        xvalues,
        results.describe(variable_name, "mean") - results.describe(variable_name, "std"),
        results.describe(variable_name, "mean") + results.describe(variable_name, "std"),
        label="std",
        alpha=0.2,
    )
    ax.plot(xvalues, results.describe(variable_name, "mean"), label="mean")
    try:
        ax.plot(xvalues, results.describe(variable_name, "1%"),
                "--", label="1%", color="black")
        ax.plot(xvalues, results.describe(variable_name, "99%"),
                "--", label="99%", color="black")
    except RuntimeError:
        pass

    ax.grid(True)
    if ylabel is None: ylabel = variable_name
    ax.set_ylabel(ylabel)
    ax.set_xlabel(xlabel)
    ax.legend()
    fig.savefig(moment_plot_filename)

    plt.figure()
    results.plot_sobols_first(variable_name,
                              xlabel=xlabel, filename=sobols_plot_filename)
    print(f"Results are in:\n\t{moment_plot_filename}\n\t{sobols_plot_filename}")

make_plots(campaign, "Ne", ylabel = r'$N_e$')
```

Listing 3: Example python code using the analysed results to produce a summary of the simulation output with uncertainties.

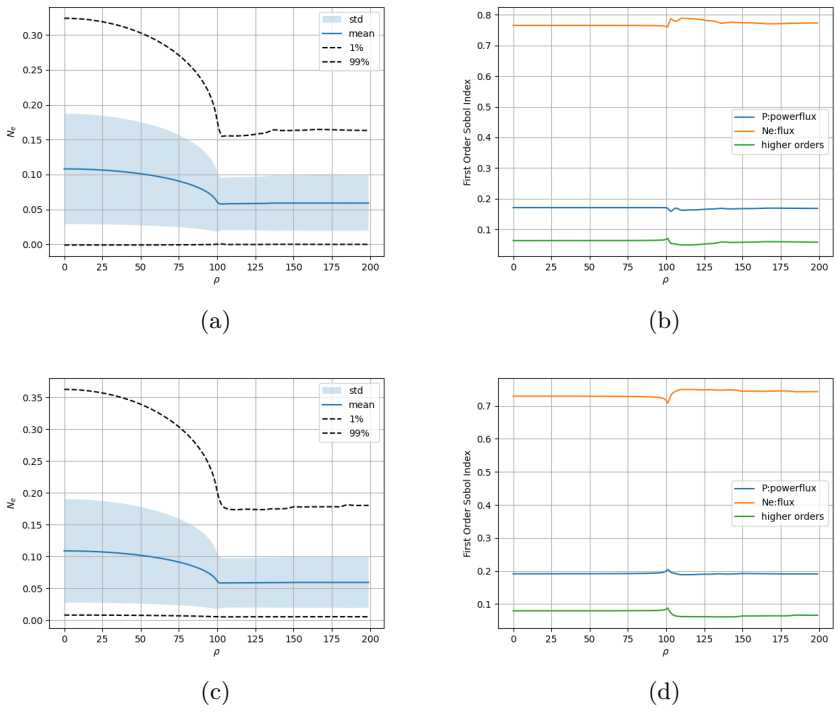


Figure 2: Plots of the mean normalised plasma density, N_e, as a function of parallel arc length along with the mean $\pm$ the standard deviation and 1^{st} and 99^{th} percentiles for a 2^{nd} (a) and 6^{th} (c) order PCE sampler and the corresponding first order Sobol indices (b/d).

5 Uncertainty quantification for SD1D case-04

We now turn to our more realistic case, including a neutral fluid. As discussed in the report for 83-2.2 [11], the performance of this case can be quite sensitive to the preconditioning approach adopted. Here we choose to use HYPRE's euclid preconditioner. This implements a parallel ILU preconditioner. Whilst we continue to run each simulation on a single core, it has been observed that the euclid preconditioner gives runs with the default input file which are approximately twice as fast as PETSC's serial ILU, resulting in run times of the order of 30 seconds per simulation. We will comment further on performance considerations shortly.

We will explore two approaches to sampling; the "polynomial chaos expansion" (PCE) sampler and stochastic collocation (SC) sampler. By default, these result in the same set of simulations (i.e. the same input samples). However, they enable different analysis chains. In particular, SC is compatible with an adaptive approach in which the UQ study can be refined by adding points in the "most useful" quantity (i.e. the one in which we have identified having the most influence on the uncertainty of the result). This is important as the number of uncertain inputs increases as unfortunately, as this increases the number of samples increases rapidly. Specifically the number of samples (i.e. simulations) in a non-adaptive approach is $(p+1)^d$ where p is the order of polynomial and d is the number of uncertain inputs.

5.1 Polynomial Chaos Expanison

We begin by exploring the uncertainty associated with the input plasma density and pressure sources, as done in section 4 and no change is required to the earlier code listings aside from changing the template input file. Figures 3 and 4 show the uncertainty in plasma density and pressure for order 1, 2, 3 and 4 polynomials (corresponding to 4, 9, 16 and 25 simulations respectively) from a study with uncertain density and pressure sources. The distributions are taken to be uniform ranging from 2×10^{23} to 1×10^{24} and 1×10^{7} to 5×10^{7} respectively. One can see that the first order case gives somewhat different results for the density than the other orders. By fourth order, the results have mostly converged to a good degree. Figure 5 shows the Sobol indices

for the density and pressure from the fourth order simulation. This demonstrates that the uncertainty on the density is primarily due to the uncertainty in the density source. However, the pressure is sensitve to both sources and is in fact most sensitive to the density source near the target. In performing such studies one can note that the run time of the individual samples can be quite variable. Here we can identify the slowest simulations as those with the largest pressure source and the smallest density source, suggesting effectively a large upstream temperature source, potentially increasing the significance of the heat conduction term.

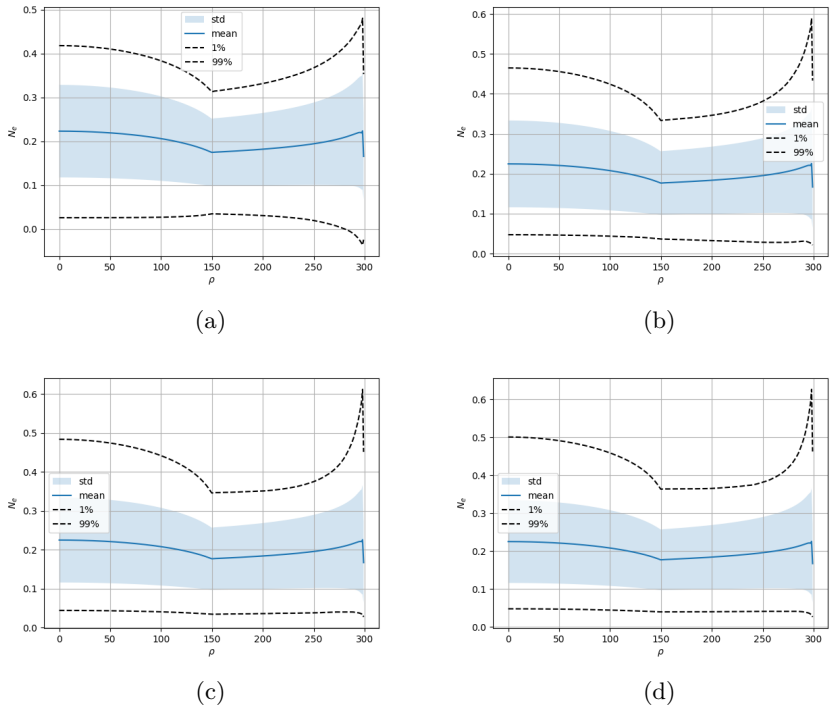


Figure 3: Plots of the mean normalised plasma density, N_e, as a function of parallel arc length along with the mean $\pm$ the standard deviation and 1^{st} and 99^{th} percentiles for first (a), second (b), third (c) and fourth (d) order PCE samplers with uncertain density and pressure sources.

In addition to the density and pressure sources one may wish to explore the uncertainty in the recyling fraction, the redistribution of neutrals, the sheath heat transmission coefficient etc. Here we include uncertainty on the sheath heat transmission coefficient with uniform distribution from 6 to 7 and uncertainty

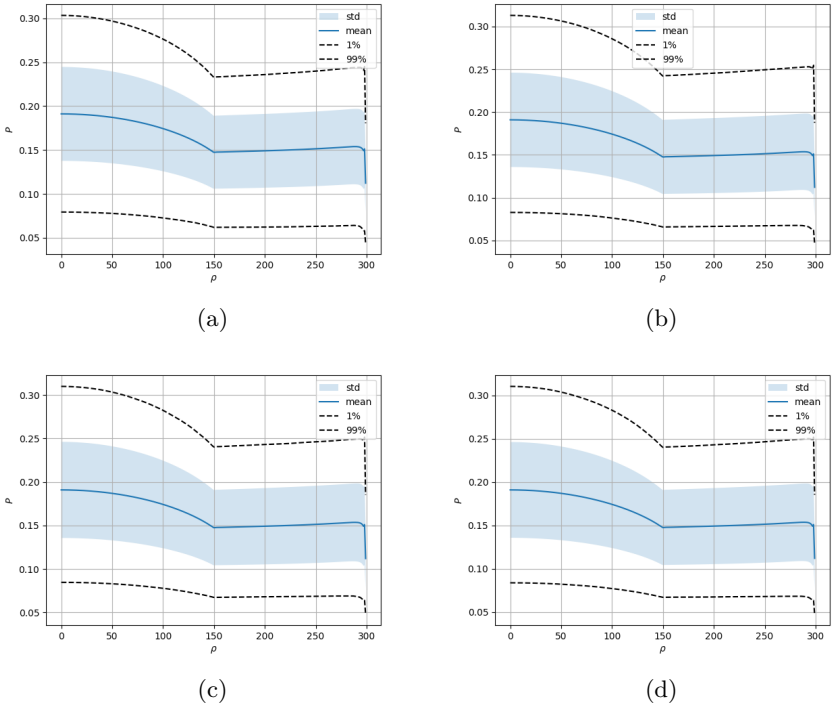
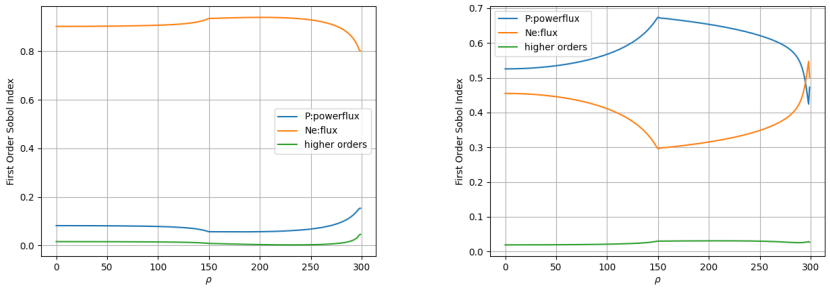


Figure 4: Plots of the mean normalised plasma pressure, P, as a function of parallel arc length along with the mean $\pm$ the standard deviation and 1^{st} and 99^{th} percentiles for first (a), second (b), third (c) and fourth (d) order PCE samplers with uncertain density and pressure sources.

on the recycling fraction with uniform distribution from 0 to 0.95. We show results from a second order case in figure 6 and it can be seen that introducing these additional terms has had a significant impact on the density and pressure. We note that the Sobol indices indicate that both the density and pressure are most sensitive to the recycling fraction, whilst both have very little dependence on the sheath heat transmission coefficient. Despite the lack of dependence on sheath heat we have had to treat this as all other parameters. If we had excluded this parameter from the sampling the number of simulations would have reduced from 81 to 27. This motivates an adaptive scheme which can iteratively refine along the most significant directions.



(a) (b)

Figure 5: Plots of the Sobol indices for the normalised plasma density, N_e, (a) and pressure, P, (b) for a fourth order PCE sampler with uncertain density and pressure sources.

5.2 Stochastic Collocation

We now turn our attention to developing an adaptive scheme more appropriate for systems with a large number of uncertain inputs. The general strategy for an adaptive scheme is as follows

1. Draw initial samples and run simulations
2. Analyse current results.
3. If stopping condition has been met go to step 8.
4. Determine all allowable sample points at the next refinement level. This is handled by `look_ahead` of the SC sampler.
5. Run simulations at all new sample points.
6. Determine which of the new sample points should be included in the analysis stage. This is handled by `adapt_dimension` of the SC analysis instance and one must choose which quantity of interest (output) to use in determing the optimal direction.
7. Go to step 2.
8. Peform final analysis and save results.

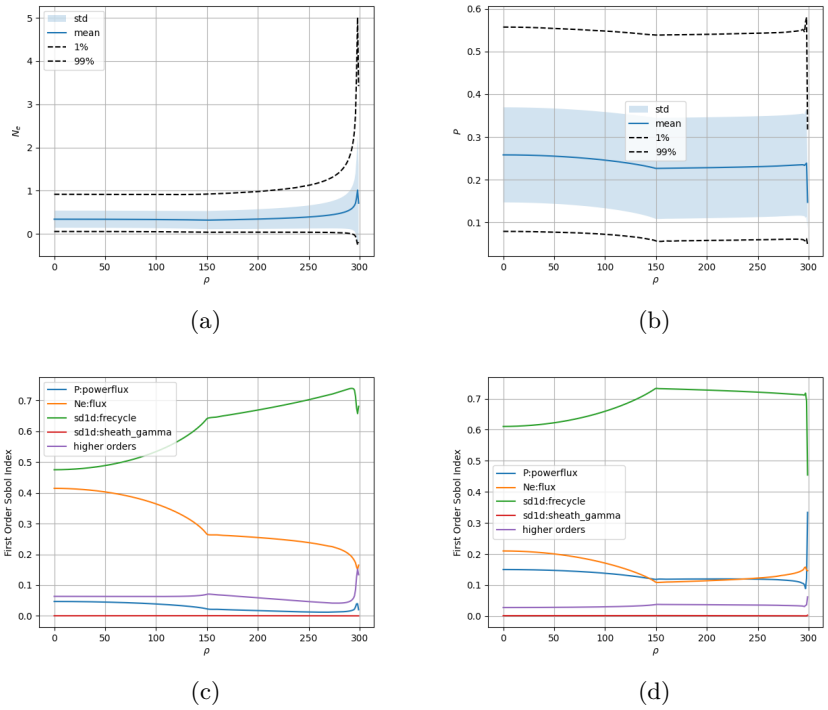


Figure 6: Plots of the mean normalised plasma density (a) and pressure (b) as a function of parallel arc length along with the mean $\pm$ the standard deviation and 1^{st} and 99^{th} percentiles for second order PCE samplers with uncertain density and pressure sources, recycling fraction and sheath heat transmission coefficient. The first order Sobol indices are also shown for the density (c) and pressure (d).

In order to achieve this practically, we switch our sampler type from PCE to SC. Whilst this can be used in the same way as the PCE solver, some additional settings are required in order make this suitable for an adaptive scheme. The full code listing is given in appendix A and the other steps to construct the campaign are unchanged from the earlier examples. Once the campaign has been created we must launch our simulations. We then continue to refine the study by executing steps 2-7 of the general strategy. One must choose an appropriate stopping condition. This could be something simple like halting after a fixed number of refinements or something more sophisticated. Here, we track the mean change in the first order Sobol indices between iterations and halt when this drops below some tolerance[5]. Of course each simulation output of interest will have different Sobol indices, so one must consider how to deal with this.

Common options will be to average the Sobol indices over all outputs or to focus on the "main" output of interest used in `adapt_dimension`.

We start by repeating the original case of section 5.1 in which there are just two uncertain inputs, the density and pressure source fluxes. We choose a Sobol convergence tolerance of 0.005 and use just the density Sobol indices in the convergence test. For this setup the analysis finishes after 4 iterations, having run 13 simulations. Figure 7 shows the total sampling points used at each iteration of the refinement. It can be seen that we start by refining the density flux, and it was shown in the equivalent PCE study that this has the most influence on the density (i.e. largest Sobol index). Following this, the direction of refinement alternates, indicating that both inputs have a significant impact on the density output. Figure 8 shows the mean density and the Sobol indices at the second and fourth (final) iteration. There is very little discernable difference in the density output, but it can be seen that the Sobol metrics have gained a higher order contribution by the final iteration. This study was repeated with the plasma pressure as the output of interest. This used the same number of simulations but completed in just three iterations. The refinement began in the pressure flux direction, but was qualitatively similar to that seen in the density focused study.

We now turn to the system with four uncertain inputs; the density and pressure sources, the sheath heat transmission coefficient and the recycling fraction. To achieve this one simply needs to add the relevant distributions to the `vary` object passed to the sampler. This study completed after seven iterations and used 41 simulations. Figure 9 shows the sampling points during the start, middle and end of the refinement phase. It can be seen that the first input refined is the recycling fraction, `sd1d:frecycle`, and that this input is the most heavily refined by the end of the study. Furthermore, it can be seen that no refinement has occured in the sheath heat transmission coefficient, `sd1d:sheath_gamma`. These observations are consistent with the results of the PCE results in figure 6 which showed that the density output was most sensitive to the recycling fraction and insensitive to the sheath coefficient. The adaptive scheme here has therefore correctly avoided refining inputs which do not impact the final results. The

[5] Whilst often effective, it is important to note that there is no guarantee that the Sobol indices have a monotonic dependence and one can sometimes find promising convergence followed by signifcant jumps. Such jumps can indicate a physics regime change or may be associated with refinement along a new input.

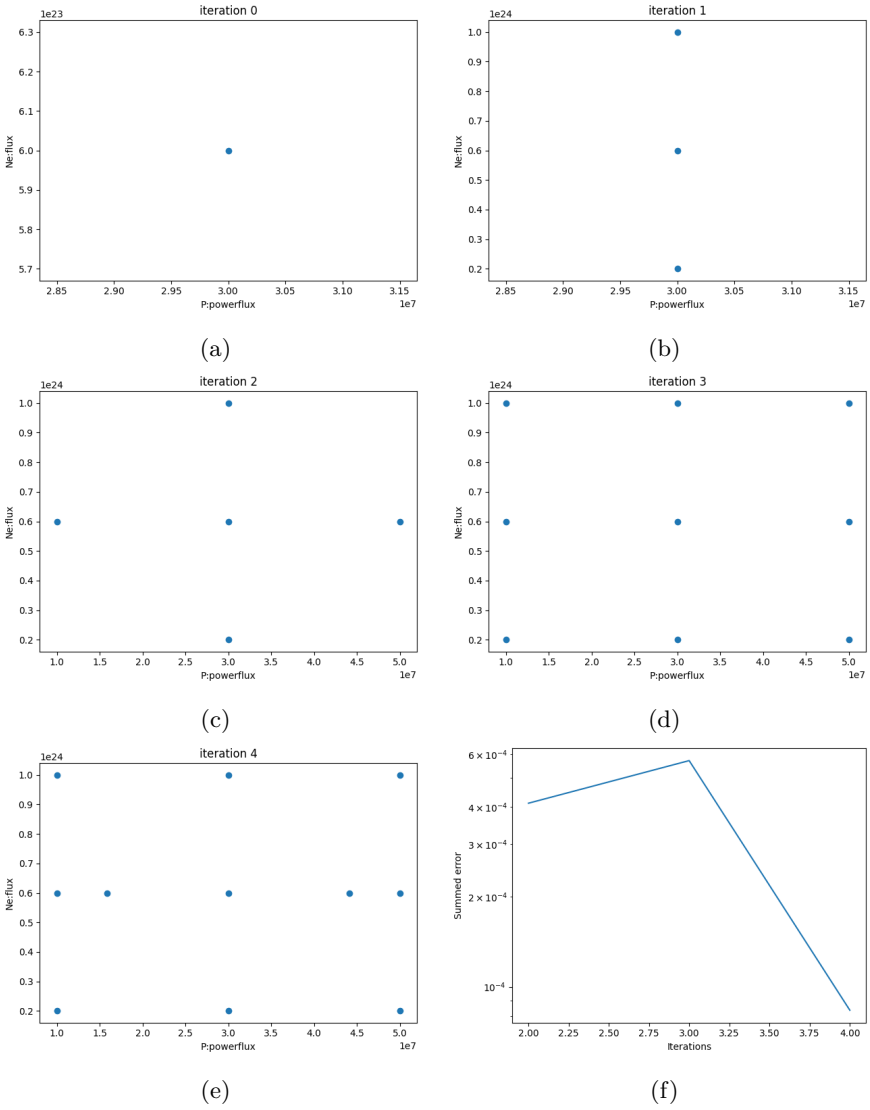
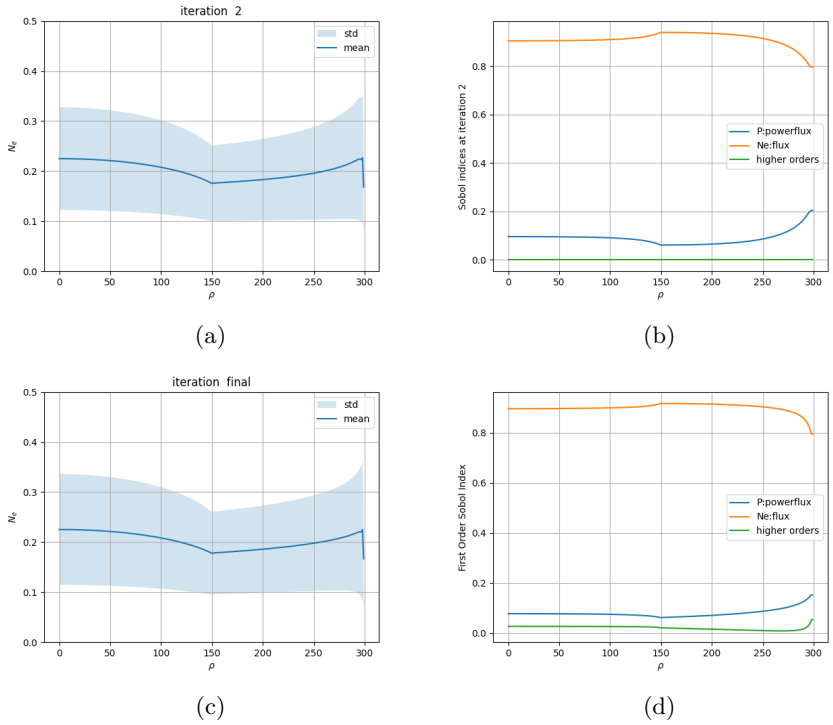


Figure 7: Sample points for each iteration of the adaptive SC scheme for the case with uncertain density and pressure sources (a-e) along with the error vs iteration count (f).

change in output from the second iteration to the final (seventh) interation is shown in figure 10

The 41 simulations used here sits between that required for the equivalent PCE study of second order (81) and first order (16). Despite involving around half the



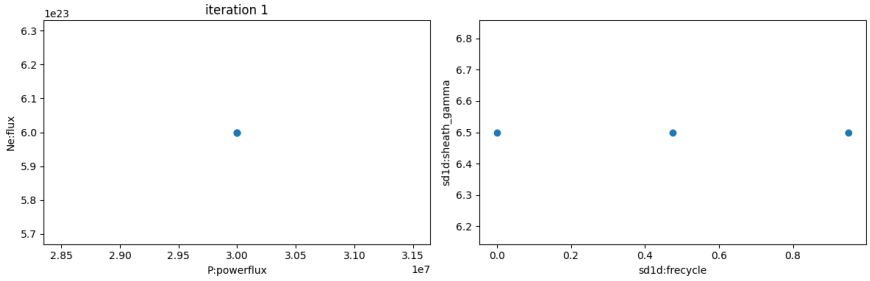
(a) (b)

(c) (d)

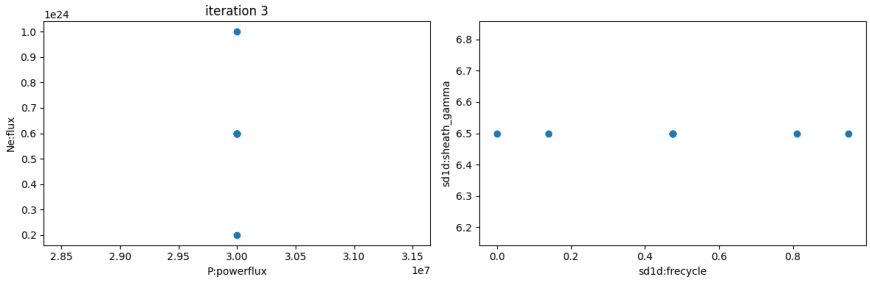
Figure 8: Plots showing mean density (a/c) and first order Sobol indices (b/d) at the second/final iteration for the case with two uncertain inputs.

simulations of the second order PCE study, the time to solution was very similar in this instance. Whilst the PCE sampler is able to generate and run all cases at once, the adaptive SC sampler can only generate a small number of cases at each refinement interation. This effecitvely limits the opportunity for simulation level parallelisation, forcing more synchronisation points within the UQ workflow. This is likely to become particularly accute in systems using a large number of refinement levels with jobs being submitted to a queue managed system. It is currently not possible to generate the simulations for the next refinement iteration until the current iteration has completed in full. This means that the simulations for each level cannot be generated and submitted to the queue until the previous stage has completed. For facilities and problem sizes where the queuing time is substantial this could lead to a large increase in the time to solution when compared to an approach which can more readily pipeline the simulations. Despite this, the adaptive approach outlined here is a powerful tool

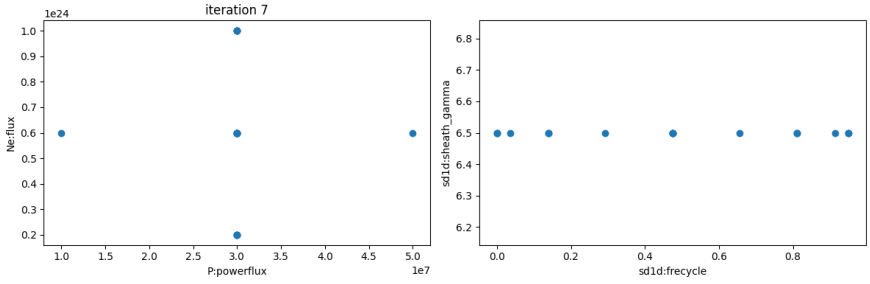
enabling the study of systems with a large number of uncertain inputs which would not be possible with a direct approach such as the PCE sampler.



(a)

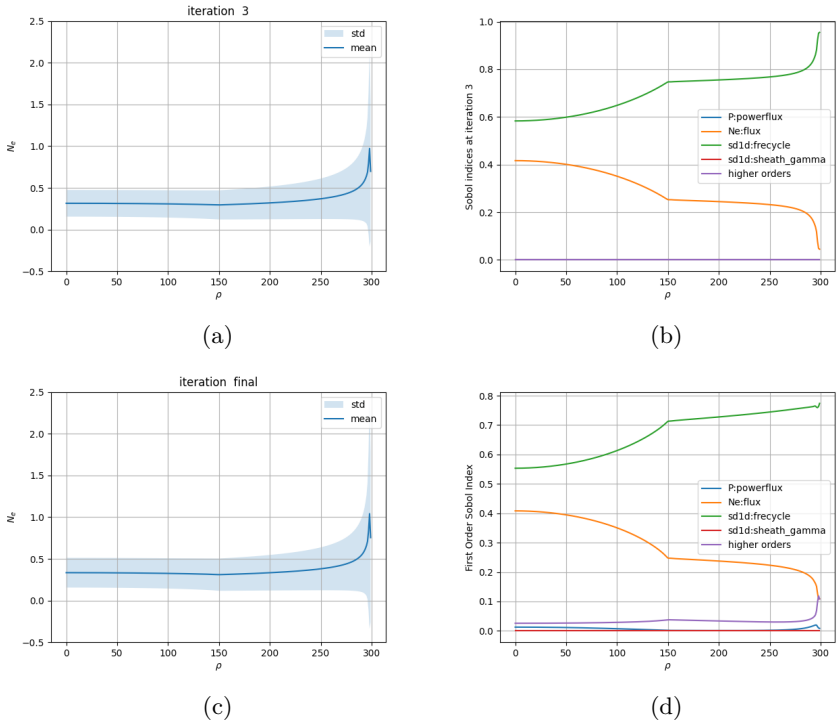


(b)



(c)

Figure 9: Sample points for the first (a), third (b) and seventh (c) iteration of the adaptive SC scheme for the case with uncertain density and pressure sources, sheath heat transmission coefficient and recycling fraction.



(a) (b)

(c) (d)

Figure 10: Plots showing mean density (a/c) and first order Sobol indices (b/d) at the third/final iteration for the case with four uncertain inputs.

6 Conclusions

In this report we have introduced some of the practical aspects of using the *easyvvuq* toolkit to construct a non-intrusive UQ workflow for an existing realistic physics code. In producing such a workflow the user both needs to supply a means to generate, run and analyse simulations for passed settings and to decide which approaches to sampling and analysis will be taken. We explored non-adaptive PCE and adaptive SC based approaches and applied this to an SD1D realistic test case. Whilst the PCE based approach offered some performance related benefits (e.g. a single trivially parallelisable group of simulations) the adaptive SC approach is more flexible and can avoid sampling along inputs which do not impact the final result. This can potentially save a lot of cpu time and can enable the study of problems with a large number of uncertain inputs. It also enables campaigns to be easily restarted and expanded, although this

has not been shown here[6].

Future work should move towards yet more realistic problems. In particular it will be useful to explore changes to the approach required when studying problems for which the simulations require a substantial number of processors. This will require a change to the execution approach to use the slurm executor. It will also be important to explore appropriate convergence criteria to halt adaptive studies. As the problem becomes yet more complex it may be necessary to consider developing cheaper surrogate models. Fortunately, projects such as VECMA are developing additional packages alongside *easyvvuq* to facilitate the construction of such problems and as these are developed further the entire community can benefit from improvements.

7 Acknowledgements

We gratefully acknowledge compute time provided on CIRRUS and Archer2 through the Excalibur SEAVEA project [14].

A Stochastic collocation code listing

The code used in section 5.2 is shown below. This is based on workflows available in reference [10] and [15].

```
#!/usr/bin/env python3

import boutvecma
import easyvvuq as uq
import chaospy
import os
import numpy as np
import time
import matplotlib
```

[6] See the `sc_adaptive_restartable` workflow of [10] for an example.

```
# Do not open figures:
matplotlib.use("Agg")
import matplotlib.pyplot as plt

# Path to the executable
EXE_PATH="./sd1d"
MINIMUM_NUMBER_OF_REFINEMENTS = 3
MAXIMUM_NUMBER_OF_REFINEMENTS = 30
ERROR_TOLERANCE = 5.0e-4
QOI="Ne"
QOI_YLABEL=r'$N_e$'

# Determine which points are allowable, run simulations here and then accept
# must relevant points into campaign.
def refine_sampling_plan(number_of_refinements, campaign, analysis, sampler):
    for i in range(number_of_refinements):
        # compute the admissible indices
        sampler.look_ahead(analysis.l_norm)

        # run the ensemble
        campaign.execute().collate(progress_bar = True)

        # accept one of the multi indices of the new admissible set
        data_frame = campaign.get_collation_result()
        analysis.adapt_dimension(QOI, data_frame)

# Plot the current set of sample points. Here we assume four
# parameters in vary, powerflux, flux, frecycle and sheath_gamma
def plot_grid_2D(i, sample, analysis, filename="out.pdf"):
    fig = plt.figure(figsize=[12, 4])
    ax1 = fig.add_subplot(
        121
    )
    ax2 = fig.add_subplot(
        122
    )
    accepted_grid = sampler.generate_grid(analysis.l_norm)
```

```
    ax1.plot(accepted_grid[:, 0], accepted_grid[:, 1], "o")
    ax1.set_xlabel('P:powerflux') ; ax1.set_ylabel('Ne:flux')
    ax2.plot(accepted_grid[:, 2], accepted_grid[:, 3], "o")
    ax2.set_xlabel('sd1d:frecycle') ; ax2.set_ylabel('sd1d:sheath_gamma')
    ax1.set_title("iteration " + str(i))
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()

# Produce plot of output with uncertainty
def custom_moments_plot(results, filename, i, quantity, ylabel = None):
    fig, ax = plt.subplots()
    xvalues = np.arange(len(results.describe(quantity, "mean")))
    ax.fill_between(
        xvalues,
        results.describe(quantity, "mean") - results.describe(quantity, "std"),
        results.describe(quantity, "mean") + results.describe(quantity, "std"),
        label="std",
        alpha=0.2,
    )
    ax.plot(xvalues, results.describe(quantity, "mean"), label="mean")
    ax.grid(True)
    if ylabel is None: ylabel = quantity
    ax.set_ylabel(ylabel)
    ax.set_xlabel(r"$\rho$")
    ax.set_title("iteration  " + str(i))
    ax.set_ylim(-0.5,2.5)
    ax.legend()
    fig.savefig(filename)
    plt.close()

def make_error_vs_iterations_plot(error_vs_its):
    plt.figure()
    plt.plot(error_vs_its)
    plt.xlabel("Iterations")
    plt.ylabel("Summed error")
    plt.tight_layout()
```

```
    plt.savefig("error_vs_iterations.png")
    plt.close()

    plt.figure()
    plt.semilogy(error_vs_its)
    plt.xlabel("Iterations")
    plt.ylabel("Summed error")
    plt.tight_layout()
    plt.savefig("error_vs_iterations_log.png")
    plt.close()

def make_samples_vs_iterations_plot(samples_vs_its):
    plt.figure()
    plt.plot(samples_vs_its)
    plt.xlabel("Iterations")
    plt.ylabel("Samples")
    plt.tight_layout()
    plt.savefig("samples_vs_iterations.png")
    plt.close()

    plt.figure()
    plt.semilogy(samples_vs_its)
    plt.xlabel("Iterations")
    plt.ylabel("Samples")
    plt.tight_layout()
    plt.savefig("samples_vs_iterations_log.png")
    plt.close()

if __name__ == "__main__":
    encoder = boutvecma.BOUTEncoder(template_input="./BOUT.inp")
    decoder = boutvecma.SimpleBOUTDecoder(variables=[QOI])
    params = {
        "P:powerflux": {"type": "float", "min": 1e7, "max": 5e7, "default": 2e7},
        "Ne:flux": {"type": "float", "min": 2e23, "max": 1e24, "default": 5e23},
        "sd1d:frecycle": {"type": "float", "min": 0.0, "max": 1.0, "default": 0.2},
        "sd1d:sheath_gamma": {"type": "float", "min": 6.0, "max": 7.0, "default": 6.5},
    }
```

```
actions = uq.actions.local_execute(
    encoder,
    os.path.abspath(
        EXE_PATH + " -d . -q -q -q solver:type=beuler" +
        "input:error_on_unused_options=false " +
        "solver:petsc:ksp_initial_guess_nonzero=yes " +
        "solver:kspsetinitialguessnonzero=true solver:petsc:pc_type=ilu"
    ),
    decoder,
)
campaign = uq.Campaign(name="adaptive_sc.", actions=actions, params=params)

vary = {
    "P:powerflux": chaospy.Uniform(1e7, 5e7),
    "Ne:flux": chaospy.Uniform(2e23, 1e24),
    "sd1d:frecycle": chaospy.Uniform(0.0, 0.95),
    "sd1d:sheath_gamma": chaospy.Uniform(6.0,7.0),
}

sampler = uq.sampling.SCSampler(
    vary=vary,
    polynomial_order=1,
    quadrature_rule="C",
    sparse=True,
    growth=True,
    midpoint_level1=True,
    dimension_adaptive=True,
)
campaign.set_sampler(sampler)

print(f"Computing {sampler.n_samples} samples")

time_start = time.time()
campaign.execute().collate(progress_bar = True)

# Create an analysis class and run the analysis.
analysis = uq.analysis.SCAnalysis(sampler=sampler, qoi_cols=[QOI])
```

```
campaign.apply_analysis(analysis)
plot_grid_2D(0, sampler, analysis, "grid0.png")

i = 0
sobols_error = 1e6
error_vs_its = [ np.nan]
samples_vs_its = [ 1 ]
while sobols_error > ERROR_TOLERANCE or i < MINIMUM_NUMBER_OF_REFINEMENTS:
    print("Iteration "+str(i))
    i += 1
    refine_sampling_plan(1, campaign, analysis, sampler)
    campaign.apply_analysis(analysis)
    results = campaign.last_analysis
    samples_vs_its.append(sampler.n_samples)

    plot_grid_2D(i, sampler, analysis, "grid" + str(i) + ".png")
    moment_plot_filename = os.path.join(
        f"{campaign.campaign_dir}", "moments" + str(i) + ".png"
    )
    sobols_plot_filename = os.path.join(
        f"{campaign.campaign_dir}", "sobols_first" + str(i) + ".png"
    )
    plt.figure()
    results.plot_sobols_first(
        QOI,
        ylabel=r"Sobol indices at iteration " + str(i),
        xlabel=r"$\rho$",
        filename=sobols_plot_filename,
    )
    plt.ylim(-0.05, 1.05)
    plt.savefig("sobols" + str(i) + ".png")
    plt.close()

    plt.figure()
    custom_moments_plot(results, moment_plot_filename, i,
                        QOI, ylabel = QOI_YLABEL)
    plt.close()
```

```
    # Prevent overwrite of old fig
    plt.figure("stat_conv").clear()
    analysis.plot_stat_convergence()
    plt.savefig("stat_convergence.png")
    plt.close()

    sobols = analysis.get_pce_sobol_indices(QOI)[2]

    if i > 1:
        sobols_error = 0
        count = 0
        for j in sobols:
            count += 1
            sobols_error += np.mean(abs(sobols[j] - sobols_last[j]))
        sobols_error = sobols_error / count
        error_vs_its.append(sobols_error)
        print("Iteration "+str(i) + " : Error " + str(sobols_error))
    else:
        # Not possible to compute error here so store Nan
        error_vs_its.append(np.nan)
        sobols_last = sobols

    # Make plot of error vs iteration now to allow user
    # to monitor progress of the job
    make_error_vs_iterations_plot(error_vs_its)
    make_samples_vs_iterations_plot(samples_vs_its)

    if i > MAXIMUM_NUMBER_OF_REFINEMENTS:
        break

time_end = time.time()

print(f"Finished, took {time_end - time_start}")

make_error_vs_iterations_plot(error_vs_its)
make_samples_vs_iterations_plot(samples_vs_its)
```

```
sobols_plot_filename = os.path.join(
    f"{campaign.campaign_dir}", "sobols_first_final.png"
)
moment_plot_filename = os.path.join(
    f"{campaign.campaign_dir}", "moments_final.png"
)

results.plot_sobols_first(QOI, xlabel=r"$\rho$", filename=sobols_plot_filename)

custom_moments_plot(results, moment_plot_filename, 'final', QOI, ylabel = QOI_YLABEL)

analysis.merge_accepted_and_admissible()
df = campaign.get_collation_result()
results = analysis.analyse(df)

sobols_plot_filename = os.path.join(
    f"{campaign.campaign_dir}", "sobols_first_final_all.png"
)
moment_plot_filename = os.path.join(
    f"{campaign.campaign_dir}", "moments_final_all.png"
)

results.plot_sobols_first(QOI, xlabel=r"$\rho$", filename=sobols_plot_filename)
custom_moments_plot(results, moment_plot_filename, 'final (merged)',
                    QOI, ylabel = QOI_YLABEL)

print(f"Results are in:\n\t{moment_plot_filename}\n\t{sobols_plot_filename}")
```

B References

[1] Benjamin Dudson. SD1D: Sol and Divertor in 1D. `https://github.com/boutproject/SD1D`.

[2] Benjamin Dudson, Peter Hill, Ed Higgins, David Dickinson, Steven Wright and David Moxey. 1D fluid model tests. `https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/2047356/TN-04.pdf`.

[3] UKAEA. Spherical Tokamak for Energy Production. `https://step.ukaea.uk`.

[4] Wright, David W. and Richardson, Robin A. and Edeling, Wouter and Lakhlili, Jalal and Sinclair, Robert C. and Jancauskas, Vytautas and Suleimenova, Diana and Bosak, Bartosz and Kulczewski, Michal and Piontek, Tomasz and Kopta, Piotr and Chirca, Irina and Arabnejad, Hamid and Luk, Onnie O. and Hoenen, Olivier and Weglarz, Jan and Crommelin, Daan and Groen, Derek and Coveney, Peter V. Building confidence in simulation: Applications of easyvvuq. *Advanced Theory and Simulations*, 3(8):1900246, 2020, doi:https://doi.org/10.1002/adts.201900246.

[5] Richardson, Robin A. and Wright, David W. and Edeling, Wouter and Jancauskas, Vytautas and Lakhlili, Jalal and Coveney, Peter V. EasyVVUQ: A library for verification, validation and uncertainty quantification in high performance computing. *Journal of Open Research Software*, 8(1):1–8, 2020, doi:10.5334/JORS.303.

[6] VECMA Team. Verified Exascale Computing for Multiscale Applications. `https://www.vecma.eu`.

[7] EasyVVUQ team. EasyVVUQ Manual : Concepts. `https://easyvvuq.readthedocs.io/en/dev/concepts.html`.

[8] Jonathan Feinberg and Hans Petter Langtangen. Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal of Computational Science*, 11:46–57, 2015, doi:https://doi.org/10.1016/j.jocs.2015.08.008.

[9] Benjamin Daniel Dudson, Peter Alec Hill, David Dickinson, Joseph Parker, Adam Dempsey, Andrew Allen, Arka Bokshi, Brendan Shanahan, Brett

Friedman, Chenhao Ma, David Bold, Dmitry Meyerson, Eric Grinaker, George Breyiannis, Hasan Muhammed, Haruki Seto, Hong Zhang, Ilon Joseph, Jarrod Leddy, Jed Brown, Jens Madsen, John Omotani, Joshua Sauppe, Kevin Savage, Licheng Wang, Luke Easy, Marta Estarellas, Matt Thomas, Maxim Umansky, Michael Løiten, Minwoo Kim, M Leconte, Nicholas Walkden, Olivier Izacard, Pengwei Xi, Peter Naylor, Fabio Riva, Sanat Tiwari, Sean Farley, Simon Myers, Tianyang Xia, Tongnyeol Rhee, Xiang Liu, Xueqiao Xu, Zhanhui Wang, Sajidah Ahmed, and Toby James. BOUT++, 3 2022.

[10] Joseph Parker, Peter Hill, David Dickinson and Benjamin Dudson. BOUT++ Vecma Hackathon. `https://github.com/boutproject/VECMA-hackathon`.

[11] Benjamin Dudson, Peter Hill, Ed Higgins, David Dickinson, Steven Wright and David Moxey. Implementation of 1D fluid model with realistic boundary conditions. `https://github.com/ExCALIBUR-NEPTUNE/Documents/blob/main/reports/2047356/`.

[12] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc Web page. `https://petsc.org/`, 2021.

[13] Chow, E and Cleary, A and Falgout, R. Design of the HYPRE preconditioner library. 9 1998.

[14] SEAVEA Team. Software Environment for Actionable and VVUQ-evaluated Exascale Applications. `https://excalibur.ac.uk/projects/seavea/`.

[15] EasyVVUQ Team. Dimension adaptive sampling tutorial. `https://easyvvuq.readthedocs.io/en/nbsphinx/notebooks/dimension_adaptive_tutorial.html`.