Contents

1. Introduction

- Some words about FSI and UQ
- Example applications
- Challenges in FSI and UQ; motivation for HPC and sophisticated mathematics
- ...

1.1. Problem Statement

- Project Context
- Perspective from different points of views: practical, HPC, multi-challenge?
- Aims and challenges
- Desired outcomes
- ...

1.2. The Multi-Challenge

- What is the multi-challenge
- Multi-physics and multi-domain FSI + multi-dimensional uncertainty
- Challenges within the multi-challenge
- ...

1.3. Related Work

- Relevant work that was done for FSI + UQ
- ...

And then motivate the structure and to what each chapter focuses on

2. Fluid-Structure Interaction

- Brief chapter description

2.1. Theoretical Background

- A few words about fluid mechanics and solid mechanics
- Multi-physics
- Lagrangian and Eulerian perspectives ; motivation for ALE
- Some FSI literature review
- ...

2.2. Modelling

- State Reynold Transport theorem
- State incompressible Navier-Stokes equations + constitutive equations
- State solid modelling + constitutive equations
- partitioned vs. monolithic approach
- ...

2.3. General Description of the Multi-Physics Code

- Overview of Alya: what is it, what was designed for, etc
- Important features
- ...

2.4. General Description of the Coupling Library

- Overview of preCICE: what is it, what was designed for, etc
- Important features
- ...

3. Forward Propagation of Uncertainty

- Brief summary of what the chapter will contain

3.1. Theoretical Background

- Explain what is forward propagation of uncertainty
- What is assumed to be known
- ..

3.2. Probabilistic Modelling

- Introduce a generic probabilistic space, where the stochastic modelling is made
- Reminder of what are univariate and multivariate random variables
- Important: we work only with iid random vars; plus what this implies
- Formulas for expectation and variance for one dim and multi dim rand vars
- ...

3.3. Overview of Sampling Methods

- Brief reminder of what sampling methods are and specify some examples
- Explain polynomial and generalized polynomial chaos methodology (from Wiener until Xiu and collaborators)
- Why to use gPC (e.g. because of the advancements in HPC, etc)
- Literature review
- ...

3.4. Monte Carlo Sampling

- Overview of Monte Carlo methodology in the UQ context
- ...

3.4.1. One-Dimensional Case

- Describe the algorithm in a one-dimensional stochastic setting
- ...

3.4.2. Multi-Dimensional Case

- Describe the algorithm in a multi-dimensional stochastic setting
- ...

3.4.3. Convergence Analysis and Properties

- Properties: e.g. Convergence for generic Monte Carlo sampling + ways to improve it (e.g. quasi Monte Carlo), independence of dimension, distribution, parallelization, numerical errors, etc
- Pros and Cons; Motivate Collocations

3.5. Stochastic Collocations

- Briefly explain what collocations are
- Specify that MCS is just a particular case of collocations, using the delta function as stochastic basis
- ...

3.5.1. Pseudo-spectral Approach

- Describe what does the pseudo-spectral approach look like and that is based on quadrature
- Explain Gauss based quadrature: Gauss-Hermite and Gauss-Legendre
- Explain how to compute the statistics with this approach
- Explain the restrictions imposed to the mean and standard deviation of the random variable ($\sqrt{2} * node * stddev + mean > 0 => mean/stddev > -\sqrt{2} * node!$)
- ...

3.5.2. One-Dimensional Case

- Describe the algorithm in a one-dimensional stochastic setting
- ..

3.5.3. Multi-Dimensional Case

- Describe the algorithm in a multi-dimensional stochastic setting
- Curse of dimensionality with brute force approach
- ..

3.5.4. Convergence Analysis and Properties

- Properties: convergence (exponential also outside the Askey scheme!!), parallelization, non-intrusive, dependence on distribution and dim, numerical errors, etc
- Pros and Cons of the method
- ...

3.6. Sparse Grids Stochastic Collocations

- Brief overview of SG-SCS
- A way to (partially) break the curse of dimensionality
- ...

3.6.1. Overview of Sparse Grids

- Overview of sparse grid methodology
- Quadrature on sparse grids
- ...

3.6.2. Sparse Grids Toolbox SG++

- Brief overview of SG++ and its capabilities
- Explain how quadrature works within SG++
- Improvements gained from using SG++ (no of collocation points, memory, convergence, etc)
- ...

3.7. Probability Density Function Estimation

3.7.1. General Description

- Overview of what density estimation is and why we use it here
- Some methods: brute force, KDE, etc
- ...

3.7.2. Kernel Density Estimation

- Explain what KDE is
- Examples of Kernels
- Implementation using Python scikit toolbox
- ...

4. Implementation Details

The purpose of this chapter is to present the implemented UQ software and it is divided into four sections. A general description is given in the first section, providing a general overview of the implementation, its functionality and properties, as well as how its interaction with the application code. The next three sections present o more detailed description, focusing on the implementation's constitutive modules: *pre-processing* phase, described in section 4.2 , UQ *simulation* phase, described in section 4.3 and at the end, the *post-processing* step, described in section 4.4 .

4.1. General Description

The UQ software consists of a main *C++ implementation* together with a set of external *tools* used for interacting with the application code. One of its important features is its *non-intrusive interaction* with the FSI software, by only needing access to the *configuration* files and the *executable* file of the multi-physics software, hence, treating the FSI solver as a *black box*. Furthermore, its *configuration* is done via a text file, at *runtime*, facilitating e.g. the switch between UQ methods or probability distributions, without the need for recompiling the code. The configuration step is further explained in the following paragraph. Another key feature is comprised by the division into *three main modules*, i.e. *pre-processing*, *simulation* and *post-processing*, which provides an easier understanding of the code, facilitates its easier extension and/or improvement and also permits its usage as a library[1]. Last but not least, although the code was tested for only two specific applications, it can be easily altered to work for any application similar to the ones considered in this work, whose configuration is done via *external files*, with the alteration being done for the tools used to interact with the application code.

It should be noted that the location of the UQ code is assumed to be *test case/uq code*, thus, all tools that interact with the application code require a relative path to the it. Moreover, the configuration file is located inside *uq code/*. It is a *text file* named *configuration.uq* that obeys the following editing rules:

- lines that start with # are ignored
- configuration lines are of the form: *name* = *value*
- no line is allowed to be empty

This implies that a line of this file has the form e.g. *nsamples* = *100*. It is comprised of three main parts, each responsible for a certain type of configuration.

[1] for the moment, this only works for applications codes with a structure similar to the ones used in this work

1. communication with the application code configuration
   In this part, the *relative path* to the external tools and specific application code files is given. As an example, the set-up for the relative path to the tool used to modify the *viscosity* value for the fluid is: *insert_nastin_exec = ../tools/util/insert_params_nastin_vis*.

2. UQ problem configuration
   In this segment, the user can set-up UQ specific data, such as which *algorithm* to employ, which *pdf* to use in order to model the input uncertainty, the number of samples for Monte Carlo Sampling, the *quadrature degree* for Stochastic Collocations, etc. Although is not a hard restriction, it is preferred that the *algorithm* and *pdf* are specified as *boolean* variables, with the following convention: 0 for Monte Carlo, 1 for Collocations and 0 for normal distribution and 1 for uniform distribution, respectively. As an example, the set-up for normal distribution is *pdf = 0*. If this rule is not met, an error message is output on the console.

3. stochastic problem set-up
   In the final part, the numerical values for the *two parameters* characterizing the uncertain input(s) are specified; for *normal* distribution, the parameters are the *mean* and the *standard deviation*, whereas for *uniform* distribution, the *left* and *right* interval boundaries, respectively. As an example, if the fluid density is modelled as a normal random variable, its configuration is *rho_f_p1 = 0.01*, followed by *rho_f_p2 = 0.001*.

The C++ implementation consists of a sequence of *classes* having the functionality outlined in the *sampling algorithms*, described in chapter 3 , together with their additional required capabilities. This means that for *Monte Carlo Sampling*, classes used to elicit a *user defined* number of *normal* and *uniform random variables* are implemented, while for *Stochastic Collocations*, the additional required functionality consists of classes that generate *nodes* and *weights* used for numerical quadrature. Moreover, additional functions are implemented in a separate helper library, used to e.g. parse the configuration file, create strings with user defined names used to call external tools, etc. An important remark is that the external tools, as well as the multi-physics software's executable file are called via *system* calls[2].

The end of the implementation process resulted in *six projects*, with the following capabilities: the first three *serial* UQ implementations consist in one for a *one-dimensional* problem, one for a scenario with *two stochastic dimensions* and one for a *multi-dimensional* stochastic problem. These codes are built as *one stand-alone applications*, with integrated constitutive modules. Although they encompass the entire UQ simulation pipeline, they proved to be of little use for the testing phase because of the *complexity* of the simulated scenarios. Even if the application codes are already parallel, together with the reduced combined cost of the pre-processing and post-processing phase compared to the simulation cost, a *serial*[3] is not practically feasible. Moreover, it also contradicts one of the goals of this work, the use of *HPC*. Given these, the next logical step seemed to be the *transformation* of the serial implementations into parallel ones, more specifically, *process (MPI-based) parallel* and hence, obtaining an HPC suited application. This was not possible, given the

[2] the *int system (const char* command)* function is available in the *cstdlib* C++ library

[3] in this context, *serial* means either a completely serial application or one that is serial, except for the application code

reasons mentioned in section 2.3 and section 2.4 : the used application code already uses MPI, thus, it is not possible to use MPI in the UQ code as well. This restriction gave rise to two possibilities: either *compile* the application code so that it does not use MPI, i.e. compile preCICE with *mpi=off* flag and Alya with *make*, or *decouple* the UQ code so that the three constitutive modules are separated, opening the possibility to run the *simulation* module in a *pseudo-parallel fashion*, by taking advantage of the *MAC Cluster's batch system* capabilities. Taking once again into account the complexity and the high compute and memory requirements of the tested scenarios, the *second* option was adopted in this work. The decoupled approach can be summarized as follows: after the pre-processing is performed for a user defined number of UQ points[4], the simulation part is done via launching *simultaneously* an equal number of *batch scripts*, letting the cluster's batch system to administrate their run, depending on the degree of cluster's occupancy. Finally, after all simulations are performed, the post-processing phase is realized. This is further explained in the next three sections. In the following paragraph, an overview of all presented implementations is provided.

The motivation behind designing a one-dimensional stochastic implementation stems from the fact that before any multi-dimensional simulation is performed, the one-dimensional ones are to be performed in order to understand *how* does the uncertainty in the input physical parameters affect the underlying simulation scenario and based on the obtained results, to formulate *relevant multi-dimensional* scenarios. The two-dimensional code differs from a *generic multi-dimensioanl* one by the fact that the implementation for Stochastic Collocations is realized using *brute-force*, i.e. the two-dimensional quadrature is done using a *tensor product* basis of the one-dimensional case. This means that for, e.g. Gauss-Hermite quadrature, one has (add reference!):

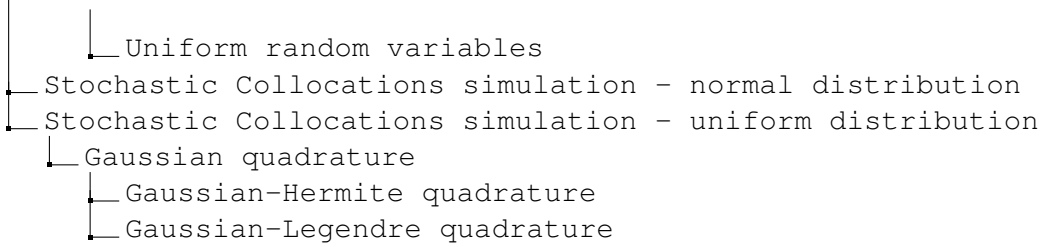
$$\int_{\mathbb{R}}\int_{\mathbb{R}} f(x,y)dxdy = \sum_{i=0}^{n}\sum_{j=0}^{n} f(\zeta_i,\zeta_j)\omega_i\omega_j, \tag{4.1}$$

where $\zeta_i, \omega_i, \zeta_j, \omega_j, i = 0 \dots n, j = 0 \dots n$ are the pair of *nodes* and respectively the *weights* for each dimension. This implementation serves as a comparison tool against the SG++ based implementation because of the relatively similar required computational resources by both approaches, which is be further presented in chapter 5 . Finally, the multi-dimensional[5] design aims to tackle an *arbitrary dimensional* stochastic problem, featuring a multi-dimensional Sparse Grids Stochastic Collocations implementation and a multi-dimensional Monte Carlo sampling algorithm.

The use of C++ as the development language offered multiple advantages, including the possibility to create an implementation that is not only suited for *HPC*, but also permits the usage of modern programming paradigms, such as *object-oriented* programming.

```
main source file
└─UQ simulation
   ├─Monte Carlo simulation - normal distribution
   ├─Monte Carlo simulation - uniform distribution
   │  └─Random number generator
   │     └─Normal random variables
```

[4] either samples, for Monte Carlo sampling, or quadrature nodes, for Stochastic Collocations
[5] in this context, multi-dimensional implies dimension greater or equal to two



At a conceptual level, the above diagram describes the UQ code structure as *tree*, having the *main source file* as root. The additional library containing the *helper* functions was omitted, because it is not illustrative for the object oriented structure. As it can be inferred from the diagram, three *parent* classes were used, each outlining a *generic behaviour*: bottom-up, *Gaussian quadrature* specifies the necessary states and behaviours for specific instances of Gaussian-based quadrature, needed by Stochastic Collocations. In the same fashion, the *Random number generator* characterizes the functionality utilized by the specific random number generators used by Monte Carlo Sampling. Finally, a generic *UQ simulation* class was defined, which outlines the generic, modular behaviour of the form pre-processing, simulation, post-processing.

As it was already stated, the UQ implementation was designed to interact with the multi-physics code in a non-intrusive way. It needs access only to the *configuration* files for the *fluid* and *structure* solvers. As an example, for the *Vertical Flap* (c.f. section 5.1), the configuration files are *Flap.nsi.dat*- for the fluid solver and *Flap.sld.dat*- for the structure solver. For the deterministic problem, the part where the physical parameters are set is:

...
PROPERTIES
DENSITY= 0.01
VISCOSITY= 0.03
END_PROPERTIES
...

for the fluid solver and

...
PROPERTIES
DENSITY= 0.01
CONSTITUTIVE_MODEL ISOLIN 0.5e5 0.3
END_PROPERTIES
...

for the structure problem, where, the numbers from the CONSTITUTIVE_MODEL line represent the *Young's modulus* and *Poisson's ratio*, respectively. It is important to emphasize that every Alya-based test case has two configuration files for the fluid and structure problem, respectively, of the form *Name.nsi.dat* and *Name.sld.dat*, where *Name* is the conventional name of the test case.

In the following sections, the pre-processing, simulation and post-processing steps are explained in more detail. Because of the identical functionality provided by the *serial* and *de-serialized* implementations, each phase is similar for both types implementations. However, as the *de-serialized* version was mainly used throughout simulations, in the following the emphasis is on solely on it.

4.2. Pre-processing

In the *pre-processing* phase consists in set-up of the UQ problem. It differs for Monte Carlo Sampling and Stochastic Collocations, hence it is presented separately. However, given the *robustness* of the Monte Carlo algorithm, the pre-processing step for *one* or *multi-dimensions* is *almost identical*, as the only change from one to many dimensions is reflected by *how many* random variables are to be generated. On the other hand, given the *classical* and *sparse grids* based implementations for Stochastic Collocations, the pre-processing phase differs and thus, it is presented distinctively, for both possibilities. As an observation, it is important to outline that before any simulation is done, it is assumed that the configuration is already set-up in *configuration.uq*.

Monte Carlo Sampling

For Monte Carlo Sampling, the first step is to generate a *user-defined number of random variables*, depending on *dimensionality*. This is realized using one of C++ standard library's pseudo-random number generators[6], based on the *Mersenne Twister 19937 generator* (add reference!) and the *random device*, used as a seed, which provides a highly accurate way of generating random variables (add reference). This functionality is provided by classes *NormalRandomVariable* and *UniformRandomVariable*. Afterwards, via calling an external bash script tool called *create_data_point*, a folder called simulation_$i, i = 0...nsamples - 1$ is created, corresponding to *each* sample that contains *two folders and one file*: *nastin_*i and *solidz_*i, $i = 0...nsamples - 1$ and the *preCICE configuration file* (c.f. section 2.4), respectively. In the *nastin* and *solidz* folders, the data that is needed by the multi-physics application code for one simulation is copied by the tool from *a generic, deterministic set-up*. Succeeding the creation of the simulation folder is the *replacement* of the deterministic physical parameter that was considered to be uncertain by the corresponding random variable. This is realized by calling an external Perl tool, with a name of the form *insert_params_nastin_* -for a fluid specific parameter, or *solidz_* - for a structure specific parameter, followed by the name of the parameter to be replaced. This tool edits the data file *in-place*. All steps can be summarized in the following algorithm:

Algorithm 1 Pre-processing for arbitrary dimension Monte Carlo Sampling

Require: $dimension$, $nsamples$ values ▷ defined in the configuration file

1: **for** $i \leftarrow 0$ to $dimension - 1$ **do**
2: call function that creates $nsamples$ random variables
3: **end for**
4: **for** $i \leftarrow 0$ to $nsample - 1$ **do**
5: call create_data_point ▷ create simulation_i folder
6: call insert_new_data ▷ replace the underlying physical parameter in simulation_i
7: **end for**

Stochastic Collocations

As a general outline, the pre-processing phase for Stochastic Collocations can be summarized as following: depending on dimension, *generate collocations points* - nodes and weight for classical approach, nodes for sparse grids approach - then *create simulation* folders,

[6] all specific functions are available in *random* library

based on the number of collocation points and in the end, modify the underlying parameters that are considered to be uncertain in the solver's data files. In the following, these steps are put into perspective for each approach.

- Classical approach

  Classical approach means *one* or *two* dimensional Stochastic Collocations, where the pre-processing step is performed as following: in the beginning, a *user-defined number of nodes and weights is generated*, where the number is given by the *quadrature degree*, defined in the configuration file. For the one-dimensional case, this functionality is implemented in the classes *GaussHermiteQuadrature* and *GaussLegendreQuadrature*. This is used for the two-dimensional case too, as described in equation 4.1 . The nodes and weights are saved statically using *STL vectors*[7] for every input *degree* less or equal to ten. Afterwards, for the one-dimensional scenario, the steps are similar to the ones for Monte Carlo Sampling: calling *create_data_point*, the folder simulation_$i, i = 0...nsamples - 1$ is created for *each* collocation point, containing *nastin_*i and *solidz_*i, $i = 0...quadraturedegree - 1$ folders, each initially containing the *same* deterministic data, and the *preCICE configuration file*. Afterwards, the *replacement* of the deterministic physical parameter that was considered to be uncertain, as presented in (add formula that will be described in introUQ!!!), by the corresponding node, follows. This is realized by calling *insert_params_nastin_* or *solidz_*, followed by the name of the parameter to be replaced. As the two-dimensional case in based on a *tensor-product* of the one-dimensional case, it follows that in the two-dimensional setting the degree is n^2, where n is the one-dimensional degree. Instead of using *unimodal* vectors for storing the nodes and weights, the two-dimensional implementation uses two *matrices*, each with two rows, one for nodes, the other for weights, each containing *n* times the n corresponding one-dimensional nodes and weights, respectively. After the collocation points are generated, the following steps are the same as presented above. All steps are summarized in algorithm 2.

Algorithm 2 Pre-processing for classical Stochastic Collocations

Require: $dimension$, $quadrature\ degree$ values ▷ defined in the configuration file
1: **if** $dimension$ is 1 **then**
2: call function that generates $quadrature\ degree$ nodes and weights
3: **else if** $dimension$ is 2 **then**
4: **for** $i \leftarrow 0$ to $quadrature\ degree - 1$ **do**
5: call function that generates $quadrature\ degree$ nodes and weights
6: **end for**
7: **end if**
8: **for** $i \leftarrow 0$ to $quadrature\ degree - 1$ **do**
9: call create_data_point ▷ create simulation_i folder
10: call insert_new_data ▷ replace the underlying physical parameter in simulation_i
11: **end for**

- Sparse grids based approach

[7] defined in the *vector* library

One difference between the *classical approach* and the *sparse grids approach* stems in the *generation* of the collocation points. After this step is complete, the remaining is identical to the previous case, hence, it is not repeated here. In this scenario, the generation of collocation points is done by SG++ and because it is more complex than the classical approach, the code providing this functionality is presented next, followed afterwards by an explanation.

```
virtual vec2d_float_t pre_processing() const
{
        SGPP::base::GridIndex* gp;
        double grid_point = 0.0;
        vec2d_float_t result;

        for(size_t i = 0; i < grid_storage_size; ++i)
        {
                std::vector<SGPP::float_t> p;
                gp = grid_storage->get(i);

                for(int j = 0 ; j < i_dim ; ++j)
                {
                        grid_point = l_limits[j] +
                                (r_limits[j] - l_limits[j]) * gp->getCoord(j);
                        p.push_back(grid_point);
                }

                result.push_back(p);
        }

        return result;
}
```

The first particularity of the above implementation is that, as presented in section 3.6.2 , SG++ has the built-in functionality for performing classical numerical quadrature on $[0,1]^d$, where d is the *dimension*. Thus, it generates *only* a set of *nodes*. Moreover, as stated in formula (add formula from section 3.6.2 to show how to do arbitrary quadrature with SG++), for both *normal* and *uniform* random variables, $[0,1]^d$ needs to be *shifted* to an arbitrary $[a,b]^d, a, b \in \mathbb{R}, a \leq b$. This being stated, the above defined method generates a matrix of *shifted* nodes, from $[0,1]^d$ to $[l\_limit_0, r\_limit_0] \times [l\_limit_1, r\_limit_1] \times \ldots \times [l\_limit_{dim-1}, r\_limit_{dim-1}]$ of dimension $grid\_storage\_size \times dim$, where $l\_limit_j, r\_limit_j, j = 0 \ldots dim-1$ represent the new *integration interval along direction j*, while $grid\_storage\_size$ is the size of the underlying generated sparse grid.

In order to use the described code, a *makefile* was created to handle its compilation, for which, the user has to be located inside *test case/uq code/src_preprocessing* and type *make*. After the code is compiled, the next step is to simply type *./executable name* in the command line. In the following, the *simulation* step is described into more details.

4.3. Simulation

to be completed; after a simulation is done on MAC cluster

4.4. Post-processing

Among the three constitutive modules, *post-processing* is the most complex. This is mainly due to the fact that it is a *subjective process*; after all simulations are performed, the resulted data can be assessed from multiple perspectives, e.g. from a probabilistic, statistics or sensitivity analysis points of view. In the underlying implementation, this is visible via a wide palette of post-processing C++ methods and external *Python*-based tools. Before going further, it is important to mention that, as presented in section 2.4 , preCICE has the built-in capability to provide the physical description of the output of the carried out simulation at each time-step, in a user defined watch-point, having the coordinates defined in *precice-config.xml*. This file is saved either for the fluid or for the structure, depending on the specified configuration in *precice-config.xml*. For example, for the *Vertical Flap* scenario, the watch-point is a text file named *Ritghtcorner.watchpoint.txt*, located in *solidz* that stores the *x displacement*, *y displacement*, Δx *displacement*, Δy *displacement* and *forces* on both x and y axis, for the *right corner* corner of the structure, i.e. coordinates *(4, 2)*. Using this feature, the first considered *quantities of interest* were chosen among the parameters available in the watch-point. This offers a good starting point for understanding the uncertainty in the considered scenario, whereas for a wider range of possibilities, specific tools were designed so that the quantities of interest are chosen from the *entire set of time-steps of the simulated data*.

As in the case of *pre-processing* step, post-processing has both common and different parts, with respect to the two considered UQ algorithms. Furthermore, the widest range of functionality is provided for the *one-dimensional* implementation, for reasons which are presented in the following. Next, the C++ based post-processing step is presented for Monte Carlo Sampling and then for Stochastic Collocations, followed by the description of the extra external tools.

Monte Carlo Sampling

In the C++ implementation, the first part of post-processing consist in *gathering* data from all simulations in a common directory, so that it is easier to be processed. Via an external bash script tool named *postprocessing_alya*, called as *system(postprocessing_alya i)*, where *i* is the simulation number, the *binary* files generated at the end of each simulation are gathered in a directory located at the same level as *simulation_i*, $i = 0 \dots nsamples - 1$, named *alya_output*, containing *nsamples* nastin and solidz folders. *Nastin* folders contain the binary data for *pressure*, *velocity*, *mesh displacement* and *mesh velocity* for each time step, whereas *solidz* folders contain structure's *displacement* data. From memory considerations, this tool *moves* all files, which extension contain *.post.*, from *simulation_i/nastin_i* and *simulation_i/solidz_i* to *alya_output/nastin_i* and *alya_output/solidz_i*, respectively. The processing of this files is done externally and it is be presented when the post-processing *external tools* are described. The next step is to gather all *watch point* files in a common folder, named *data_results/MCS/*, located in *test case/uq code/* directory. This is realized via an external Python tool, called *gather_data_mc*. The common location of the data files facilitates the next step of the post-processing, as described in the following.

After all data files are available, the *statistical evaluation* of the chosen quantities of interest, or *observables*, is realized according to the description presented in section 3.4 : the *mean* and *variance* are computed for each time step, as described in (add MCS algorithm reference from section 3.4). After the statistics are computed, they are saved in file called

Statistics.mc.txt, located in a specifically created post-processing folder, *postprocessing_results/SCS/*, found at the same level as *data_results*. This ends the C++ specific implementation description, with the important remark that the described steps hold for *all* considered dimensions. The above description is summarized in algorithm 3.

Algorithm 3 C++ Post-processing for Monte Carlo Sampling

Require: $dimension$, $nsamples$ values ▷ defined in the configuration file
1: **for** $i \leftarrow 0$ to $nsamples - 1$ **do**
2: call get_alya_output ▷ move data from simulation_i/ to alya_output/
3: call gather_data_mc ▷ copy data from simulation_i/ to data_results/MCS/
4: compute statistics for the chosen quantities of interest from the watch point
5: **end for**

Stochastic Collocations

In the case of Stochastic Collocations, the C++ implementation of the post-processing step is more sophisticated that the one for Monte Carlo Sampling. Likewise, from a *functional* point of view, there is no difference between the three dimensionality-based designs. The difference occurs at a *non-functional* level, between classical and sparse grids approaches, as it was explained in chapter 3 . The first two steps are the *gathering* of the binary files and data files for the underlying *watch point* for each time step, respectively, realized by get_alya_output and gather_data_sc tools, the latter being designed in a similar way as the corresponding one for Monte Carlo Sampling, which copies the corresponding data files in *data_results/SCS/*. After these steps are realized, the next part consist in *accessing* all data saved in the *watch point* and saved it locally, using STL vectors. This is done via a tandem consisting of a Python tool, which reads the data from each datafile and outputs it in the command line, and a C++ function, which reads the output by using functionality provided by the *POSIX* library[8], followed by its local storage. The need for this step is motivated by the fact that the heart of Stochastic Collocations based on *spectral approach* is the gPC *expansion coefficients*. Hence, the next operation consist in *computing* the coefficients, as presented in (add formulas for computing coefficients!!!), followed by the computation of the *statistics*, described in (add formulas for SCS statistics!!!). These are steps are put together in the next algorithm:

Algorithm 4 C++ Post-processing for Stochastic Collocations

Require: $dimension$, $quadrature\ degree$ values ▷ defined in the configuration file
1: **for** $i \leftarrow 0$ to $nsamples - 1$ **do**
2: call get_alya_output ▷ move data from simulation_i/ to alya_output/
3: call gather_data_sc ▷ copy data from simulation_i/ to data_results/SCS/
4: call get_output_sc ▷ read and store data from all files in data_results/SCS/
5: compute gPC expansion coefficients
6: compute statistics for the chosen quantities of interest from the watch point
7: **end for**

Besides what does the C++ implementation offer, there are several external tools that aid

[8] all specific functions are found in *cstdio* library

the post-processing step, for both Monte Carlo Sampling and Stochastic Collocations algorithms. The first one that is described is related to the *Alya* package and is called *alya2pos*. It is a Fortran based implementation, whose role is the *transformation* of the *binary* output of each simulation to *ensight*[9] format, which can be visualized using, e.g. Kitware's visualization package Paraview[10]. It is executed as *./alya2pos test_case_name*, where test_case_name is the chosen conventional name of the tested scenario (as an example, for the *vertical flap* scenario, it is executed as *./alya2pos Flap*). Having the entire output data in ensight format, offers many benefits because in this way, the data files contain numerical data which can be easily processed. Whereas the C++ implementation provides only the functionality for the computation of statistics of the underlying *watch point*, several external tools were designed to compute the statistics of the *entire simulation*, i.e. the *mean* and *variance* for each time step. For Monte Carlo Sampling, this process is straightforward, as the only needed quantities are the sum and the sum of squares, respectively, of all quantities. This is realized by two Python tools called *MCS_postprocessing_mean* and *MCS_postprocessing_var*, respectively. For Stochastic Collocations, before the statistics are computed, the first step is to *compute* the gPC expansion coefficients, for each simulated time step. This is done with the aid of two tools, named *SCS_postprocessing_NRV* and *SCS_postprocessing_URV*. The first is used in case of normal random variables, whereas the second is employed in case of uniform random variables. After the coefficients are computed, via two other tools, called *SCS_postprocessing_mean* and *SCS_postprocessing_var*, the mean and the variance are being computed, respectively. A key feature of these tools is that they are *dimension-independent* and hence, adding stochastic dimensions to the problem does not result in an increased cost for their usage. Statistical moments offer a good statistical description of the considered quantity(ies) of interest. But often, this is enough. Considering that the *forward propagation* of uncertain inputs result in a *stochastic output*, besides the moments computation, a deeper description is provided by the estimation of the *probability density function*(pdf) of the underlying quantity of interest. This is realized by employing the *kernel density estimation* procedure, described in section 3.7.2 . Using a *Gaussian* kernel, two tools named *MCS_distr_estimation* and *SCS_distr_estimation*, respectively, were created to estimate the pdf at each time step. While in the case of Monte Carlo Sampling, the process of estimating the pdf is based on the set of outputs corresponding to the propagation of set of input samples, for Stochastic Collocations, this process differs. As previously described, the output of this algorithm consist in the of gPC expansion *coefficients*. Hence, by using formula (add formula for gPC expansion from introUQ section), the stochastic output can be approximated. This is realized in a Monte Carlo-like fashion, by *generating* a set of *i.i.d.* random variables, serving as arguments for the expansion's corresponding hypergeometric polynomial (in this work, *Legendre* or *Hermite* polynomials). Hence, by applying the kernel density estimation methodology on the obtained set of stochastic outputs, the pdf estimation is performed. It is important to remark that this functionality is implemented only for one-dimensional simulations.

[9] see http://www-vis.lbl.gov/NERSC/Software/ensight/doc/OnlineHelp/UM-C11.pdf for more details

[10] see http://www.paraview.org/ for more details

5. Test Scenarios and Results

This chapter presents the numerical results of the undertaken simulations. The testing part of this work was done using *two FSI scenarios*, namely Vertical Flap and add second name. The description of these scenarios, together with the corresponding results is made in section 5.1 and section **??** , respectively. The section describing the first scenario begins with its physical description, with the purpose of familiarizing the reader with its characteristics. This is followed by the description of a *deterministic* simulation, which is intended to provide a mean for comparison with the stochastic simulations. And finally, the uncertainty quantification results are presented in section 5.1.2 , section 5.1.3 and lastly, section **??** , for all considered dimensions. The second scenario - t.b.d.

An important observation is that due to the complexity of both applications, the first challenging task was to find a suitable number of time steps so that one simulation is realistic from a compute and memory point of views. Furthermore, as mentioned in section 3.2 , the uncertain parameters were modelled as *independent and identically distributed (i.i.d.) continuous random variables*.

5.1. Vertical Flap

The first simulated scenario was a *Vertical Flap*, presented in figure 5.1 . In the following, a general overview is given. It consists of a *channel flow* of a fluid and a vertical *elastic structure*, which deforms while the fluid passes through. The spacial discretization is realized via *finite elements* (FEM), resulting in an *unstructured two-dimensional mesh*, consisting of *triangles*, as following: for the *interior* of the fluid domain, there were used 2584 elements and 160 for its *boundaries* , while for the structure domain, 144 elements were used for the *interior* and 32 elements for the *boundaries*.

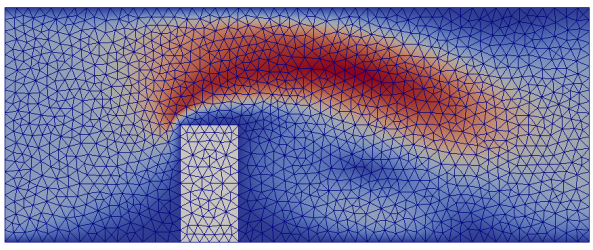


Figure 5.1.: Vertical Flap

The *boundary conditions* for the fluid solver are *inflow* for the left wall, *no-slip* for top and

bottom walls and finally, *outflow* for the right wall. The inflow consists of an initial velocity with value 100. The configuration part which is of interest for this work is the one of the input *physical parameters*, whose initial values are presented below.

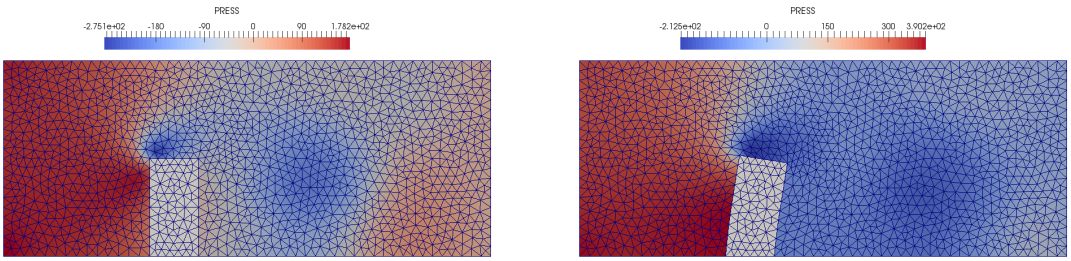
- Fluid density ρ_f = 1e-2 $[kg/m^3]$
- Fluid viscosity η = 3e-2 $[m^2/s]$
- Structure density ρ_s = 1e-2 $[kg/m^3]$
- Young's modulus E = 0.5e5 $[N/m^2]$
- Poisson's ratio ν = 3e-1

The first two quantities describe the fluid flow (*density* and *viscosity*), while the other three characterize the structure (*density*, *Young's modulus* and *Poisson's ratio*). These parameters were at the heart of all simulated UQ scenarios, by being considered, either individually, or in certain combinations, as *uncertain*.

5.1.1. Deterministic Simulation

Before any UQ simulations were carried out, the first important step was to perform *deterministic* simulations. Besides the task to find a deterministic scenario with a suitable number of time steps to make the simulation realistic from a compute and memory point of views, another cornerstone was related to the *amount of information* got from one simulation. This implies that the settled number of simulation steps should be high enough so that e.g. the structure oscillates, the flow develops, etc. Moreover, the resulting deterministic data represents a mean for *comparison* with the data yielded by stochastic simulations. Upon its visualization, the potential *hot spots* can be identified, i.e. spacial locations which exhibit a distinguishable behaviour. Examples of hot spots can be, e.g. the locations close to the structure, where, depending on the physical configuration, small vortices can form, or the location where the fluid flow reaches steady state, etc. For the structure, potential interesting locations can be, e.g. the corners of the structure, etc.

Several numbers of time steps were simulated (100, 200, 500) and the number that was chosen such that all the above criteria are met was *200*. The first and the last time step



(a) Deterministic simulation at time-step 1

(b) Deterministic simulation at time-step 200

Figure 5.2.: Deterministic simulation of the vertical flap

of the deterministic simulation are presented in figure 5.2 , where the visualized fluid's physical parameter was its *pressure*.

In the following, the results obtained after the UQ simulations were performed are presented. The next section describes the *one-dimensional* stochastic cases, while the following ones treat *multi-dimensional* scenarios.

5.1.2. One Dimensional UQ Simulation

As the deterministic case was the first important step in the entire set of simulations, the first milestone for the UQ simulations were the *one dimensional* cases. This means that throughout a simulation, *four* out of the *five* input physical parameters are taken to be *sure variables*[1] (see [1] for details), with the fifth being considered to exhibit *uncertainty*. Given the number of input parameters, all five one-dimensional cases were simulated. This means that for the one-dimensional case, an *exhaustive* treatment of uncertainty was made, paving the way for multi-dimensional simulations, as it is explained later in this section. Moreover, for a comprehensive testing and assessment of the employed methods, multiple *configurations* were accounted. First, the variables taken to exhibit uncertainty were modelled as *normal* random variables, and were propagating in the underlying system using both Monte Carlo Sampling and Stochastic Collocations. Afterwards, they were modelled as *uniform* random variables and propagated using Stochastic Collocations. A more detailed presentation is done in the following.

As a starting point, uncertainty is modelled as a *normal random variable* Ω of the form:

$$\Omega \sim \mathcal{N}(\mu, \sigma^2), \text{with } \sigma = 0.1\mu \tag{5.1}$$

The first used method was Stochastic Collocations and in order to carry out simulations, the *quadrature degree* (denoted here by q) and the *number of coefficients* (denoted here by n) have to be established. Based on the maximal implemented q, i.e. ten (c.f. section 4.2) and the suggested *rule of thumb* from [2], which states that $q \leq 2n$, the established combination for this work is $q = 8$ and $n = 5$. The first processed data sets were the ones

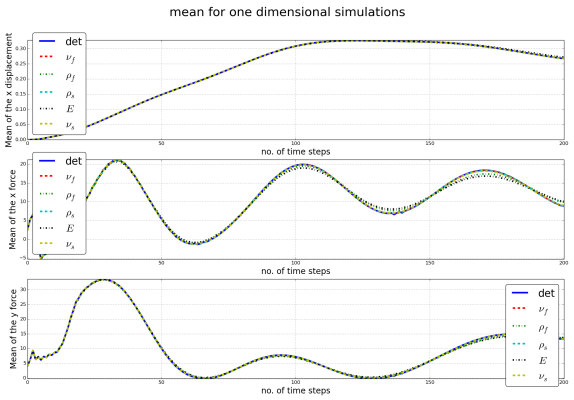


Figure 5.3.: Mean for one dimensional scenarios

[1] a sure variable is defined to be *exactly determined by given conditions*, i.e. *deterministic*

corresponding to the *watch point*, which in this case is the upper right corner. Moreover, the quantities of interest were chosen to be the *x axis displacement*, *x axis force* and *y axis force*. In figure 5.3 and figure 5.4 , the *means* and *variances* of the gathered data are

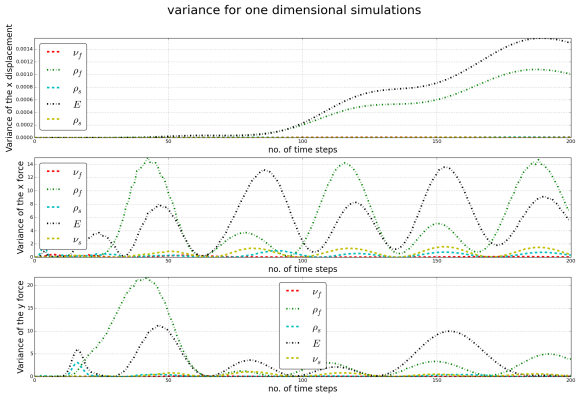


Figure 5.4.: Variance for one dimensional scenarios

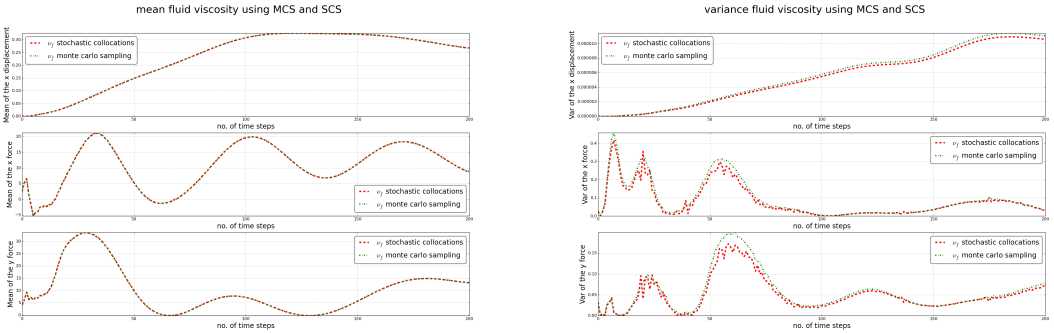
presented, with the means plot including the *deterministic* data as well. From these two plots, several inferences can be made. As a starter, from the first plot, it can be seen that the means of the simulated scenarios almost coincide with each other and even more, are almost identical to the deterministic data, except for data corresponding to the *Young's modulus*, which, after about time step 80, behaves a little different for the force on the x axis. This outlines two important inferences: first, *on average*, the behaviour is *similar* to the deterministic case and second, *Young's modulus* seems to be the parameter whose uncertainty influences the most the outcomes. Although the plot of the expected values offered the first insights, the *variances* plot is even more interesting, because, among others, it indicates which parameters are important from an uncertain point of view. First of all, it can be seen that, as the variances of the forces exhibit *oscillatory* behaviour, with the one for the y axis force more attenuated, the variance of the x axis displacement increases over time, reaching the peak around time step 190, in the underlying scenario of 200 simulated steps. Moreover, among the five physical parameters, two of them stand out: *fluid's density* and *structure's Young modulus*. From the last observation, it can be pointed out that for the two dimensional case it is expected that combinations including these parameters should have a big impact on the outcomes.

The results obtained using Stochastic Collocations and normal random variables were compared with the ones output by Monte Carlo Sampling. Considering the high computational and storage demand of this method, a number of *100 samples* were simulated and to quantify the resulting difference, the *mean squared error* (m.s.e.) of the means and variances was computed. The comparison was made when the stochastic parameter was *fluid's viscosity*. The statistics plot is outlined in figure 5.5 , whereas the corresponding mean squared errors are presented in table 5.1 . Based on these results, it results that Stochastic Collocations with a quadrature degree of order 8 and 5 expansion coefficients converges. Moreover, similar results were obtained when testing for the other four parameters and hence, ascertaining the previous assertion.

The next step was to get a better understanding of how the normal distribution affects

statistic	x axis displacement m.s.e.	x axis force m.s.e.	y axis force m.s.e.
mean	6.238e-10	4.974e-4	2.737e-4
variance	9.603e-14	4.359e-4	1.436e-4

Table 5.1.: Mean squared error for fluid's viscosity



(a) Mean comparisson

(b) Variance comparisson

Figure 5.5.: Comparison of results obtained with both algorithms for fluid's viscosity

the outcomes. To this extent, based on equation 5.1 , the uncertain parameters were modelled as *uniform random variables* Γ of the form:

$$\Gamma \sim \mathcal{U}(a, b), \text{with } a = 0.9\mu \text{ and } b = 1.1\mu, \tag{5.2}$$

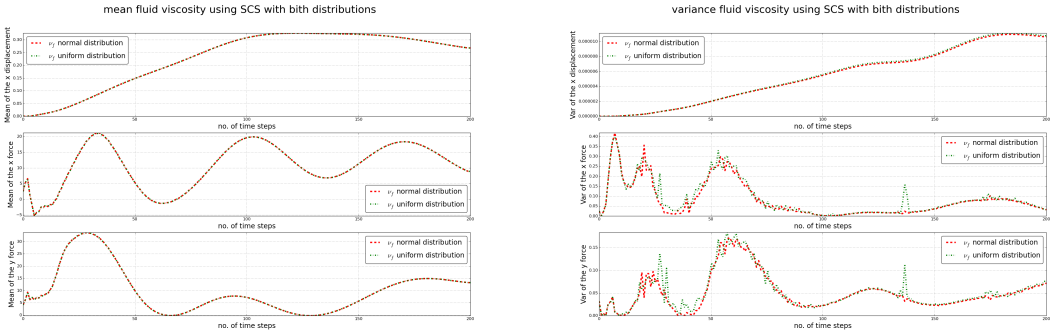
where μ is the mean used for normal random variables. As with Monte Carlo Sampling, the comparison was made when the stochastic parameter was *fluid's viscosity*. The statistics plot is outlined in figure 5.6 and the corresponding mean squared errors are presented in table 5.2 . Likewise Monte Carlo Sampling, the results obtained by using uniform random

statistic	x axis displacement m.s.e.	x axis force m.s.e.	y axis force m.s.e.
mean	1.049e-8	9.987-4	5.072e-4
variance	1.244e-14	4.737e-4	1.568e-4

Table 5.2.: Mean squared error for fluid's viscosity

variables were very similar to the case when using normal random variables. Even more, similar results were obtained for the other four parameters as well.

Based on these results, it follows that Stochastic Collocations with normal random variables as in equation 5.1 , is enough for performing uncertainty propagation in the underlying scenario. That is why, in the following, only results outlined by this strategy are presented.



(a) Mean comparisson

(b) Variance comparisson

Figure 5.6.: Comparison of results obtained with normal and uniform for fluid's viscosity

5.1.3. Two Dimensional UQ Simulations

5.1.4. Five Dimensional UQ Simulations

5.2. Scenario II - rename

6. Conclusions

6.1. Summary

-

6.2. Outlook

-

Appendix

A. To be written

Bibliography

[1] D.S. Lemons. *An Introduction to Stochastic Process in Physics*. The Johns Hopkins University Press, 2002.

[2] B. Sudret. Global sensitivity analysis using polynomial chaos expansions. *Reliability Engineering and Systems Safety*, 93:964–979, 2008.