DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUMKolleg Forschungsarbeit

Parallel Evaluation of Adaptive Sparse Grids with Application to Uncertainty Quantification of Hydrology Simulations

Jonas Treplin

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUMKolleg Forschungsarbeit

Parallel Evaluation of Adaptive Sparse Grids with Application to Uncertainty Quantification of Hydrology Simulations

Parallele Evaluierung von adaptiven dünnen Gittern mit Anwendung für Unsicherheitsquantifikation von Hydrologie Simulationen

Author:	Jonas Treplin
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz, Markus Stöckle
Advisor:	Tobias Neckel, Ivana Jovanovic, Michael Obersteiner
Submission Date:	1.12.2020

Abstract

In this work we implement parallel evaluation of an arbitrary problem function for integration-based operations using the sparseSpACE framework. We apply this implementation to Uncertainty Quantification using the LARSIM hydrology model as problem function and compare the sparseSpACE solution to the result of a Monte-Carlo simulation.
Testing the limitations and scalability of our work on Sparse Grid and Adaptive Refinement techniques shows that on small to medium problems a parallelisation can be profitable. Our parallel evaluation is best suited for functions with a medium execution time of seconds to minutes. We suspect a number of difficulties could arise when applying our technique to problems with longer execution time or a high amount of required evaluation points. Other possible architectures of parallelisation and their advantages and disadvantages over this work are discussed in the end.

Kurzfassung

In dieser Arbeit wird eine parallele Evaluierung einer willkürlichen Funktion für integrationsbasierte Operationen des sparseSpACE Framework implementiert. Wir nutzen diese neue Implementation in der Uncertainty Quantification und wenden sie auf das Hydrologiesimulationsmodell LARSIM an. Die Ergebnisse von sparseSpACE werden mit denen einer Monte-Carlo Simulation verglichen. Tests der Limitationen und Skalierbarkeit unserer Arbeit auf dünnen und adaptiven Gitter-Methoden zeigen, dass kleine bis mittelgroße Anwendungen von einer Parallelisierung profitieren können. Unsere Parallelisierungsmethode ist für Funktionen mit einer mittleren Ausführungszeit in der Größenordnung von Sekunden bis Minuten geeignet. Wir vermuten, dass mehrere Schwierigkeiten auftreten könnten, wenn unsere Methode auf Probleme mit längerer Ausführungszeit oder einer hohen Anzahl an benötigten Evaluationspunkten angewandt wird. Andere mögliche parallele Architekturen und deren Vor- und Nachteile gegenüber dieser Arbeit werden am Ende diskutiert.

Contents

1. Introduction

Conventional scientific models take parameters they assume are correct and calculate a deterministic output which also is presumably exact. In reality however we can't have this kind of infinite fidelity into the parameters and model output. Aside from a numerical error the input values often are not deterministic but have to be regarded as a random variable. A measurement for example may be a sample from a random variable with normal probability distribution. Uncertainty introduced in the parameters is propagated through the model into the output.
In Uncertainty Quantification one wants to determine how uncertain this output is. Multiple techniques exist to achieve this. Monte-Carlo methods use the law of large numbers to build an output distribution with a high amount of samples. Another common technique is to determine statistical moments directly through weighted integration. A third option is to use a Polynomial Chaos Expansion (gPCE) approach. The latter two both require numerical integration of a multidimensional function whereas Monte-Carlo relies on a large amount of samples. To perform the quadrature efficiently Sparse and Adaptive Grids can be used to reduce the number of function evaluations.
The sparseSpACE framework [1] provides implementations of different adaptive refinement techniques for Sparse Grids and supports Uncertainty Quantification using gPCE as well as direct methods.
The model used for this work is LARSIM [2], a hydrology model. Since it has a considerable execution time of around half a minute for a single run, parallel evaluation of integration nodes could drastically improve the computation time. The sparseSpACE framework had limited capabilities for parallel evaluation. Since Uncertainty Quantification relies on the quadrature facilities our application can profit from parallel evaluation.
Similar work was done in [3] where the sparseSpACE framework has been used to quantify uncertainty. Also a pedestrian simulation model was executed in parallel. However it focused on the comparison of different integration techniques and its parallel evaluation scheme was not generally applicable.
Our goal is to provide a function independent approach to parallel evaluation in sparseSpACE and apply its Uncertainty Quantification facilities to another model. Additionally we specifically benchmark and test the limitations of our parallel evaluation engine and compare the three different Uncertainty Quantification techniques described earlier with regard to their parallelizability using the LARSIM model as test function.

2. Forward Uncertainty Quantification

A scientific model usually operates on presumably exact inputs and computes a deterministic output. However, the input values are often uncertain and are more accurately described by a probability distribution. Forward Uncertainty Quantification is concerned with describing the output of the model in a probabilistic sense. There are multiple techniques to perform an Uncertainty Quantification. Here we only focus on the non-intrusive ones which regard the model as a black box and do not require code modification in the model. Additionally we also limit ourselves to determine only expectation and variance of the output distribution.
For a random variable X with probability density function (PDF) $\rho(x)$ expectation and variance of the output of a function f are defined as.

$$E[f(x)] = \int_X f(x)\rho(x)dx \tag{2.1}$$

$$Var[f(x)] = E[(f(x) - E[f(x)])^2] \tag{2.2}$$

A straightforward approach to determining them would be to directly evaluate these weighted integrals. In this case the error convergence depends on the used integration technique. However other techniques exist which have different effects on error convergence rates.

2.1. Monte-Carlo Methods

Another way to perform forward Uncertainty Quantification is through the Monte-Carlo Method. This section is based on a summary in section 2.8 of [4].
Suppose we want to calculate the expectation of the output of an arbitrary function f which depends on an uncertain parameter ω with a known distribution. Using the Monte-Carlo approach we sample N samples $\{\theta_1, \theta_2, \ldots, \theta_N\}$ from the distribution of ω. The expectation of $f(\omega)$ is then estimated by:

$$\mathbb{E}[f(\omega)] \approx \frac{1}{N}\sum_{i=0}^{N} f(\theta_i) \tag{2.3}$$

This can also be generalized to the variance:

$$Var[f(\omega)] = \mathbb{E}[(f(\omega) - \mathbb{E}[f(\omega)])^2] \approx \frac{1}{N}\sum_{j=0}^{N}(f(\theta_j) - \frac{1}{N}\sum_{i=0}^{N} f(\theta_i))^2 \tag{2.4}$$

It can be shown that the error on the estimation of expectation for this technique is in $\mathcal{O}(\frac{1}{\sqrt{N}})$. Although it converges relatively slow its convergence rate is independent of the dimensionality of the problem function. This is an important advantage over the integration based techniques whose convergence rates depend on the number of dimensions.

2.2. Uncertainty Quantification using Polynomial Chaos Expansion

Many techniques of Uncertainty Quantification rely on generalized Polynomial Chaos Expansion (gPCE). This section provides a quick overview of gPCE and is based on chapter 10 of [5] and chapter 5 of [6].
A gPCE approximates a function $f(\omega)$ that depends on the uncertain parameter ω as a weighted sum of orthogonal polynomials similar to a Fourier Transform.

$$f = \sum_{i}^{\infty} \hat{f}_i \Phi_i \tag{2.5}$$

Orthogonal polynomials Φ are families of polynomials whose inner product is 0. The weighted integral of two functions is used as the inner product.

$$\langle f, g \rangle_\rho = \int f(x) g(x) \rho(x) dx \tag{2.6}$$

We also note that

$$\mathbb{E}[f(\omega) g(\omega)] = \langle f, g \rangle_\rho \tag{2.7}$$

where ρ is the probability density function (PDF) of the random variable ω.
A family of polynomials Φ is orthogonal if the following holds:

$$\left\langle \Phi_i, \Phi_j \right\rangle_\rho = \gamma_i \delta_{ij} \tag{2.8}$$

$$\gamma_i = \langle \Phi_i, \Phi_i \rangle_\rho \tag{2.9}$$

The polynomials Φ are only orthogonal with respect to a certain weighting function ρ. Some often used families for common PDFs are:

- The Hermite polynomials H_i for Normal distributions ($X \sim N(0,1)$):

  $$\rho(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

  The first four polynomials:

  $$\begin{aligned} H_0(X) &= 1 \\ H_1(X) &= X \\ H_2(X) &= X^2 - 1 \\ H_3(X) &= X^3 - 3X \end{aligned}$$

- The Legendre polynomials L_i for Uniform distributions ($X \sim U(-1,1)$)

  $$\rho(x) = \frac{1}{2}$$

The first three polynomials:

$$L_0(X) = 1$$
$$L_1(X) = X$$
$$L_2(X) = \frac{2}{3}X^2 - \frac{1}{2}$$

For other weighting functions the polynomials can be numerically computed. Orthogonal polynomials are used as a basis to approximate a function $f(\omega)$. Where ω is a random variable. The polynomials have to be chosen with respect to the PDF of ω. Similar to a Fourier series the function is then approximated up to a level N by:

$$f \approx \sum_{i}^{N} \hat{f}_i \Phi_i \tag{2.10}$$

$$\hat{f}_i = \frac{1}{\gamma_i} \langle f, \Phi_i \rangle_\rho \tag{2.11}$$

The coefficients can be used to gain information about the distribution of $f(\omega)$. Using the following derivation for the calculation of the expectation:

$$\begin{aligned}\mathbb{E}[f(\omega)] &= \mathbb{E}[\sum_n \hat{f}_n \Phi_n(\omega)] \\ &= \sum_n \hat{f}_n \mathbb{E}[1 * \Phi_n(\omega)] \\ &= \sum_n \hat{f}_n \mathbb{E}[\Phi_0(\omega) * \Phi_n(\omega)]\end{aligned}$$

Because of the orthogonality $\mathbb{E}[\Phi_0 * \Phi_n]$ is always 0 except for $n = 0$. So it follows that:

$$\mathbb{E}[f(\omega)] = \hat{f}_0 \tag{2.12}$$

To compute the variance we use a similar approach:

$$Var[f(\omega)] = \mathbb{E}[(f(\omega) - \mathbb{E}[f(\omega)])^2] \tag{2.13}$$
$$= \mathbb{E}[(\sum_n \hat{f}_n \Phi_n(\omega) - \hat{f}_0)^2] \tag{2.14}$$
$$= \mathbb{E}[(\sum_{n=1} \hat{f}_n \Phi_n(\omega))^2] \tag{2.15}$$

Because of orthogonality we can simplify the squared sum, terms with mixed polynomials are 0. Then the following can be used to determine the Variance:

$$Var[f(\omega)] = \sum_{n=1} \hat{f}_n^2 \mathbb{E}[\Phi_n(\omega)^2] = \sum_{n=1} \hat{f}_n^2 \gamma_n \tag{2.16}$$

For proof and further explanation see chapter 10 of [5].
To use this framework for a multivariate function $f(\vec{\omega})$ for the vector $\vec{\omega}$ of mutually independent variables we use multidimensional polynomials.

$$\mathbf{\Phi}_{\vec{i}} = \prod_{j=1}^{d} \Phi_{\vec{i}_j} \tag{2.17}$$

$$\left\langle \mathbf{\Phi}_{\vec{i}}, \mathbf{\Phi}_{\vec{j}} \right\rangle = \gamma_{\vec{i}} \delta_{\vec{i}\vec{j}} \tag{2.18}$$

$$\gamma_{\vec{i}} = \gamma_{\vec{i}_1} \gamma_{\vec{i}_2} \dots \gamma_{\vec{i}_d} \tag{2.19}$$

$$\delta_{\vec{i}\vec{j}} = \delta_{\vec{i}_1\vec{j}_1} \dots \delta_{\vec{i}_d\vec{j}_d} \tag{2.20}$$

To truncate at some user-given level N we use all polynomials with 1-norm less than N ($|\vec{i}|_1 \leq N$; $|\vec{i}|_1 = i_1 + i_2 + \dots$). The following expansion then approximates f.

$$f \approx \sum_{|\vec{i}|_1 \leq N} \hat{f}_{\vec{i}} \mathbf{\Phi}_{\vec{i}} \tag{2.21}$$

$$\hat{f}_{\vec{i}} = \frac{1}{\gamma_{\vec{i}}} \langle f, \mathbf{\Phi}_{\vec{i}} \rangle \tag{2.22}$$

The expectation and variance are then determined by:

$$\mathbb{E}[f(\vec{\omega})] = \hat{f}_{\vec{0}} \tag{2.23}$$

$$Var[f(\vec{\omega})] \approx \sum_{1 \leq |\vec{i}|_1 \leq N} \hat{f}^2_{\vec{i}} \gamma_{\vec{i}} \tag{2.24}$$

In the following section we describe methods of numerical integration to obtain the coefficients needed for the PCE.

2.3. Integration Techniques for Uncertainty Quantification

To perform the integration required to determine the coefficients of the PCE described in equation 2.11 efficiently, an understanding of numerical integration schemes is required. The first subsection introduces important one-dimensional numerical integration techniques. The latter subsections provide a quick overview over Sparse Grids and Adaptive Grids. They are loosely based on chapter 11 of [5] and Section 2 of [7].
In general a quadrature rule Q is a weighted sum which approximates the Integral I:

$$If \approx Qf = \sum_{i<n} w_i^Q f(p_i^Q) \tag{2.25}$$

Where p^Q are the points where the function f is evaluated, also called nodes, w^Q are their corresponding weights and n is the number of points.

2.3.1. Examples of Single-Dimensional Quadrature Rules

This section provides a quick explanation of some of the single dimensional quadrature rules used in section 5.2.

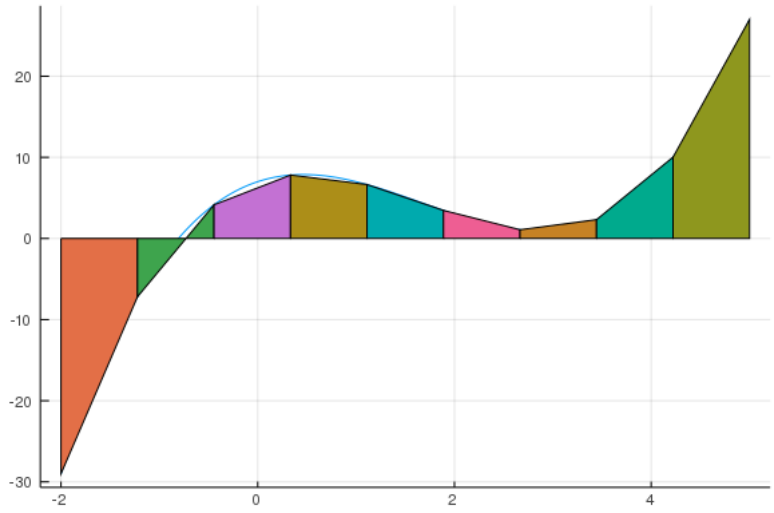

Figure 2.1.: This figure shows the linear interpolation of a polynomial as well as the resulting trapezes used for integration.

Trapezoidal rule A simple approach to integrate a one-dimensional function is to linearly interpolate it and integrate the surrogate. This ultimately leads to evaluating multiple trapezes. Figure 2.1 illustrates the resulting area. The integral can then be estimated by:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-2} h\frac{f(ih+a)+f((i+1)h+a)}{2} \tag{2.26}$$

Because every point except the ones on the boundary is evaluated twice we can simplify this expression a bit. Rephrasing into nodes and weights we have the following for N points:

$$p = \{i * \frac{|b-a|}{N} | i \in 1,2,\ldots,N\} \tag{2.27}$$

$$w = \{\frac{1}{2},1,1,\ldots,1,\frac{1}{2}\} \tag{2.28}$$

This approach is simple and flexible because of its equispaced points. It is often used for more complicated refinement strategies. However an accurate result with this technique requires that relatively many points have to be evaluated. Furthermore it is highly similar to the definition of the Riemann integral being the average of the lower and upper sums.

Gaussian Quadrature Named after Gauss and invented in its current form by Jacobi this technique is especially suited to evaluate weighted integrals of the form $\int_a^b f(x)w(x)dx$. They can correctly integrate polynomials up to degree $2n-1$ with n points. Multiple variants specific to different a, b and w exist but all follow the same scheme. For further explanation we refer to [8].
The general rule is to take a family of polynomials $p_0, p_1, p_2 \ldots p_n$ which are orthogonal with

respect to w in the interval between a and b. Take the roots $x_0, x_1, \dots x_n$ of the n-th polynomial as points to get n nodes. The i-th weights are then the integral of the Lagrange polynomial for the i-th point.

$$w_i = \int_a^b \prod_{j=0 j \neq i}^{n} \frac{x - x_j}{x_i - x_j} dx \tag{2.29}$$

This integrates the polynomial interpolation of $f(x)w(x)$. Important variants of Gauss Quadrature include the Gauss-Hermite Quadrature which uses the Hermite polynomials, integrates from $-\infty$ to ∞ and uses the PDF of a standard distribution as weighting function. Another example is Gauss-Legendre using the before mentioned Legendre polynomials with a Uniform distribution on the interval between -1 and 1. In Uncertainty Quantification we often have to evaluate weighted integrals of the form $\int_a^b f(x)w(x)dx$. Gaussian Quadrature is an efficient way to numerically estimate these.

2.3.2. Full Grid Quadrature

We can generalize an arbitrary one-dimensional quadrature rule to a multidimensional problem function $f(x_1, x_2, \dots, x_n)$. The points of the single dimension quadrature rules have to be combined with each other to form multidimensional coordinates. Equation 2.30 shows how to construct a quadrature rule $\boldsymbol{Q}$ for higher dimensions from one-dimensional rules.

$$\boldsymbol{Q} = Q \otimes Q \otimes \cdots \otimes Q \tag{2.30}$$

$$(Q \otimes Q)f = \sum_i \sum_j w_i^Q w_j^Q f(p_i^Q, p_j^Q) \tag{2.31}$$

This tensor product formulation leads to them often being called full tensor grids. Standard full grids use $\mathcal{O}(n^d)$ grid points where n is the number of grid points for a single dimension and d the dimensionality of the function. This means to obtain the same error in a $d+1$ dimensions as in d dimensions one has to multiply the number of points evaluated by N. This exponential growth is also called the curse of dimensionality.

2.3.3. Sparse Grids

Because of this exponential growth, Full Grids are not viable for problems with a higher number of dimensions, therefor more efficient methods have been invented. The goal of Sparse Grids is to remove a large portion of nodes while maintaining a reasonable error convergence rate. To integrate functions of medium dimensionality ($d < 20$) they provide a more optimal solution.
The first step for a Sparse Grid is to split the full grid into hierarchical subsets of points. The following is an example of a hierarchization of a Trapezoidal Grid.
Consider this slightly different formulation for the points of the trapezoidal quadrature rule:

$$\Phi_N = \{i * (N+1)^{-1} \mid i \in 1, 2, \dots N+1\} \tag{2.32}$$

A hierarchical subset ϕ_l of level l is defined by

$$\phi_l = \{\frac{i}{2^l} \mid i \in 1, 2, \dots 2^l, i \; odd \; \} \tag{2.33}$$

It contains 2^{l-1} points. Figure 2.4 shows subspaces of the levels 1 to 4.

In general this hierarchization can be performed via a telescopic sum.

$$\Delta_l f = (Q_l - Q_{l-1}) f \tag{2.34}$$

$$Q_l f = \sum_{i<l} \Delta_i f \tag{2.35}$$

To combine points for a number of multiple dimensions d into a multidimensional grid $\boldsymbol{Q}_{\vec{l}}$ the tensor product is used.

$$\boldsymbol{Q}_{\vec{l}} = Q_{\vec{l}_1} \otimes Q_{\vec{l}_2} \otimes \cdots \otimes Q_{\vec{l}_d} \tag{2.36}$$

A multidimensional Full Grid quadrature $\boldsymbol{Q_N}$ can be obtained by adding all multidimensional subsets together whose maximum norm of their level is less than or equal to a user defined level N. This combines a hypercube of subsets.

$$\boldsymbol{Q_N^{full}} = \bigoplus_{|\vec{l}\|_\infty \leq N} \boldsymbol{Q_{\vec{l}}} \tag{2.37}$$

One can observe that higher level spaces have drastically more points and therefore a higher computation cost than the lower level spaces. They also contribute less to the accuracy in comparison to their cost. The number of evaluated points can be severely reduced by eliminating the higher level subspaces. We restrict the sum of the single dimension levels instead of their maximum to be below a user defined value N. This can be interpreted as cutting the hypercube on a diagonal hyperplane.

$$\boldsymbol{Q_N^{sparse}} = \bigoplus_{|\vec{l}|_1 \leq N+d-1} \Delta_{\vec{l}} \tag{2.38}$$

An often used variant of Sparse Grids is the combination technique. Instead of a hierarchical set of subspaces one uses anisotropic Full Grids. The combination technique Quadrature $\boldsymbol{Q_N^{combi}}$ then follows:

$$\boldsymbol{Q_N^{combi}} f = \sum_{l \leq |\vec{l}|_1 \leq N+d-1} (-1)^{N+d-|\vec{l}|_1-1} \binom{d-1}{|\vec{l}|_1 - N} \boldsymbol{Q_{\vec{l}}} f \tag{2.39}$$

Figure 2.2 shows a tableau of full grids used and Figure 2.3 the resulting Sparse Grid for $N = 4$ and $d = 2$.

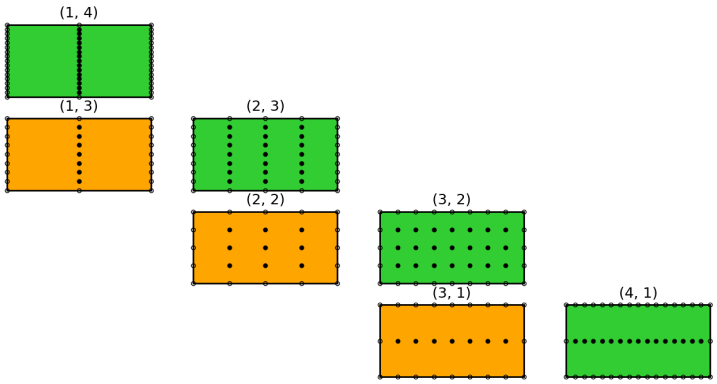

Figure 2.2.: The anisotropic full grids used for the standard combination technique with $N = 4$. The green grids have a positive weight while the orange ones are subtracted. The plot has been produced by the sparseSpACE framework [1].

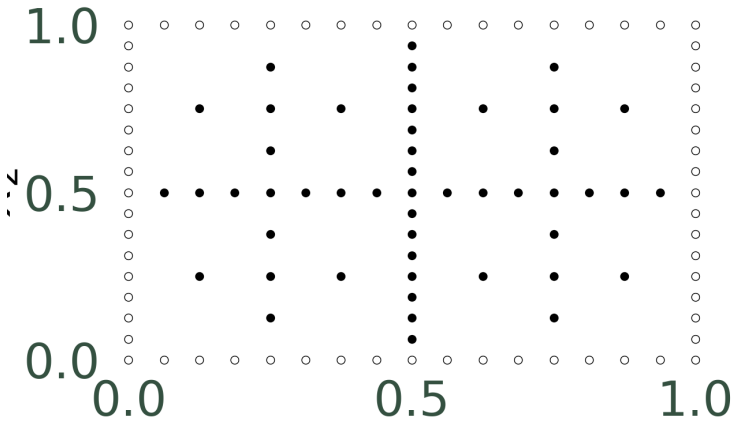

Figure 2.3.: The resulting grid points for the combination technique with the subspaces used in Figure 2.2. The plot has been produced by the sparseSpACE framework [1].

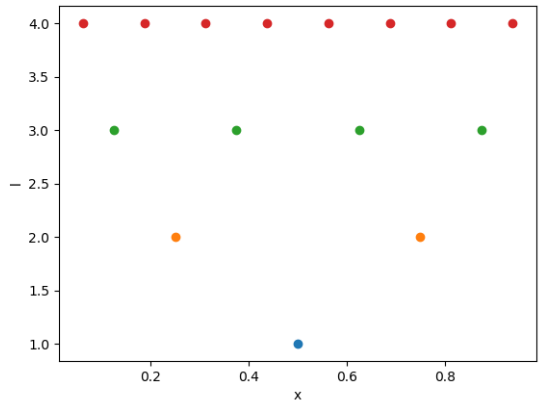

Figure 2.4.: The hierarchical subsets of a equispaced grid from level 1 to 4.

2.3.4. Spatially Adaptive Grids

Static Sparse Grids perform well on smooth functions but come at a higher error cost if the function is not smooth. To overcome this limitation Sparse Grids can be adaptively refined at runtime. There are multiple strategies to refine a Sparse Grid. This section discusses simple spatially Adaptive Grids.
Spatially adaptive refinement schemes are able to improve accuracy in certain areas without adding more points in other areas of the grid. They allow the user to initially choose a relatively low level and only focus the computation cost on non smooth parts of the function such as discontinuities.
To motivate a primitive refinement strategy we notice that the hierarchical nodes each have $2d$ child nodes which are adjacent to them and contained in the subsets that are one level higher than their parents. Refining a point adds his child nodes to the grid. The added nodes can have other parent nodes which have to be added to the grid in a recursive way. Figure 2.5 shows an example of two possible refinement steps including adding additional parent nodes.
Multiple refinement criteria exist to decide at which points more nodes have to be added. A straightforward one is to take the points with the highest surpluses, meaning the points which currently contribute the most to the result. This criterion has the advantage that it only relies on already available information and does not require additional function evaluations. This is especially important in scenarios where a function evaluation is costly and/or time-consuming.

2.3.5. Boundary Points

When selecting a grid on which to integrate, it is an important decision if one includes nodes on the boundary of the integrated interval. This choice usually depends on the evaluated function. Including them results in drastically more grid points that are particularly

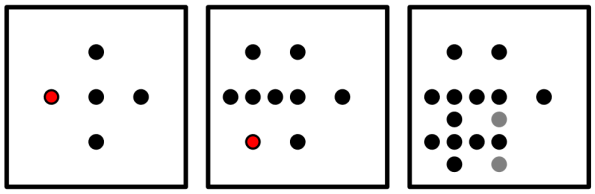

Figure 2.5.: The Figure shows two steps of refinement. The black points are already evaluated, the red are the points that are to be refined and the gray ones are additional parent points that have to be added. The figure was taken from [7].

concentrated on the boundary which leads to a higher integration cost. To demonstrate this consider the case of a d-dimensional space. When excluding the boundary, the level 1 grid has only a single point to evaluate, whereas including them there are already 3^d nodes. Furthermore this decision does not change the asymptotic convergence rate. Also Figure 2.3 shows the boundary points. It can be observed that in two dimensions these already make up for a very significant portion of nodes.

3. Simulation Software

In this chapter we describe the software used to perform the Uncertainty Quantification as well as the hydrology simulation model LARSIM (Large Area Runoff Simulation Model).

3.1. sparseSpACE

The Sparse Grid Spatially Adaptive Combination Environment (sparseSpACE) [1] is a python framework to perform different operations optimized with Sparse and Adaptive Grid schemes. These operations include Integration, Interpolation and Uncertainty Quantification. It also supports different grids and refinement techniques together with multiple error estimators.

3.1.1. General Architecture

This section is a summary of the authors own understanding that has been acquired with the tutorials provided by [1]. SparseSpACE uses a Function object that the operation is applied to. The `Function` class defines an abstract `eval` method which has to be implemented in custom applications. The object includes a cache of previously evaluated nodes and their results. This enables sparseSpACE to evaluate nodes multiple times at the cost of only one evaluation.
A `Grid` object defines a Full Grid that is later used by the Sparse or Adaptive Grid Scheme. It also defines the interval on which the function is evaluated. Implemented grids include Gauss-Hermite and Gauss-Legendre-Grids as well as Trapezoidal Grids. The `Operation` object specifies which operation should be performed on the Function object. It needs at least the `Function` and the `Grid` instances as parameters. The operation object also handles the evaluation of the function. Some of the possible operations are Integration, Interpolation and Uncertainty Quantification. The final piece is the Sparse Grid Scheme which can also be adaptive. It needs the `Operation` object to initialize. It is responsible for creating the nodes at which the instance of `Function` is evaluated. Some Schemes only work with specific kinds of grids. For example the adaptive schemes do not accept the Gaussian Grids.

3.1.2. Uncertainty Quantification with sparseSpACE

In [9] Uncertainty Quantification for sparseSpACE was implemented. Its corresponding `Operation` is the `UncertaintyQuantification`.
This operation only works with weighted trapezoidal, HighOrder and Langrage Grids, which is important because using another grid may not cause an error but will produce a wrong result. After an adaptive refinement has been performed the Operation can use the method `calculate_expectation_and_variance` to calculate the expectation and

variance directly through the weighted integration approach. Alternatively we can use `get_expectation_and_variance_PCE` after calling `calculate_PCE` to use PCE method described in 2.2. The operation was originally described for the single dimension refinement technique introduced in [10].
We can decide which method we want to use to determine the expectation and variance. Either we choose to calculate the expectation and variance directly using the weighted integration or we decide to make use of the gPCE technique. The `UncertaintyQuantification` operation itself is based on the `Integration` class which means, it uses its integration facilities to calculate the statistical moments and from there the expectation and variance. It relies on the grid-specific `Integrator` to perform the actual evaluation and weighting of the nodes.

3.2. LARSIM

For this work we use the Large Area Runoff Simulation Model (LARSIM) as the application model for the Uncertainty Quantification. This section is not an accurate description of the mathematical model but focuses on the input and output data. It is based on the English documentation [2].
LARSIM is a conceptional hydrology simulation model. It can be used to describe the current state of the system, to simulate a hypothetical system or to make forecasts. Larsim models multiple processes including storage in soil as well as lakes, snow accumulation and melt, evapotranspiration and runoff of stored water. These are applied to a discrete approximation of a landscape which is divided into cells. The landscape data includes for example specifications for land use, runoff connections to other parcels and vegetation. LARSIM has two different kinds of parameters, first meteorological data like precipitation and temperature and secondly terrain specific data like area type. Furthermore constants used for modeling can be modified as well. The Leaf Area Index (LAI) models the amount of precipitation that can be intercepted by plants. Some of these parameters have constraints originating from their physical background. For example The EQI parameter which models the retention of interflow, has to be always smaller than EQB which models the retention for groundwater storage. For this work we are only interested in the former kind of parameters which are passed to the model in a CSV format. The output we focus on is the time series of the runoff at a single station.
To use the model which is written in FORTRAN, in Python we use the Larsim_Utility_Set [11]. It wraps calling of the LARSIM executable and writes parameters into the correct input files. It also transforms the given parameters to fit the previously described constraints. To do this only an offset from the dependent value is passed to the utility function which is then scaled and added/subtracted to its dependency. Therefor, we can only perform an Uncertainty Quantification with respect to offsets for these constrained parameters.
The Larsim_Utility_Set framework does not support meteorological and terrain-specific parameters at the time of writing. Therefor we can only perform experiments with the modelling constants.

4. Implementation of parallel evaluation of LARSIM in Python

Since each evaluation of LARSIM takes about half a minute of time, a parallel evaluation could save a tremendous amount of time. When we started this project sparseSpACE had no general way to evaluate a function in parallel. A parallelisation done by the user of the framework is almost impossible because its top-level interface does not leave much control to him. Implementing it in the framework itself also turned out to be a difficult task. Function evaluations can be triggered at a multitude of points. One would have to rewrite a large portion of the framework to execute them in parallel.
A small exception exists: a grid evaluates its points for integration using an `Integrator` object which receives an area it has to integrate, generates the necessary evaluation nodes and executes the function with the corresponding parameters. This integrator can be turned into a parallel execution engine in a straightforward way. However this engine is limited to operations which use the `Integrator` to evaluate the function.

4.1. Parallelisation of sparseSpACE Integration

We parallelised integration on the `Integrator` level to be as low level as possible and therefore keep the implementation general. A new variant of the `IntegratorArbitraryGrid`, `IntegratorParallelArbitraryGrid`, has been introduced. To use it one has to set the `integrator` attribute of the used grid to an instance of `IntegratorParallelArbitraryGrid` which requires the grid itself as an argument. However it requires that the function can run in parallel. This may mean that one has to setup separate working directories for different processes.
We use a multiprocessing approach which is more flexible and will work on different architectures; it also circumvents the global interpreter lock of CPython. To communicate between the processes we used the Message Passing Interface (MPI) standard for which bindings where provided by the `mpi4py` library [12].
The `IntegratorParallelArbitraryGrid` works by collecting all nodes on the master process and use the MPI gather/scatter idiom to distribute them onto each process. After the evaluation is done the results are gathered on the master process which then performs a weighted summation to calculate the integration result.
To prevent that each process works with a different fraction of the cache, they have to be updated. This is done by gathering them on the master process, merging and broadcasting them back to all processes which then update the cache of their respective function with the new global one.

Pseudocode of the whole procedure is shown in Algorithm 1. However this naive version

Algorithm 1 Pseudocode of the parallel integration function executed by the `IntegratorParallelArbitraryGrid` for N processes

```
function INTEGRATE(...)
    if currentprocess == masterprocess then
        nodes ← collect all points and weights to be evaluated
        packets ← split them into N evenly large packets
    end if

    packet ← MPI_Scatter(packets)
    result ← weighted sum of evaluated nodes in packet
    results ← MPI_Gather(result)

    if currentprocess == masterprocess then
        total ← sum results
    end if

    caches ← MPI_Gather(cache)
    if currentprocess == masterprocess then
        globalcache ← merge caches
    end if

    cache ← MPI_Bcast(globalcache)
    return MPI_Bcast(total)
end function
```

does not necessarily distribute an even amount of work to each process. Some processes could get more cached nodes whereas others have to do more evaluation work. This leads to the latter processes taking substantially more time while the former just wait for them which means that resources are not used efficiently. Especially in steps with a large amount of nodes these imbalances could cause a significant waiting time. To prevent this, one has to distribute packets with the same ratio of cached and uncached nodes. We implemented `IntegratorParallelArbitraryGridOptimized` which solves this problem by distinguishing between cached and uncached nodes, builds packets of evaluated and unevaluated nodes for each process and concatenates them so that each have a roughly equal amount of uncached nodes.

4.2. Embedding Larsim

The Larsim_Utility_Set [11] provides the necessary functions to call the LARSIM executable from the python code. To work with the sparseSpACE framework we have to wrap the

`LarsimModel` object of the Utility Set into a `Function` class. Therefore we have to implement a `call` method. The array of parameters has to be converted to a named dictionary for the `LarsimModel`, which then returns a pandas dataframe. Another utility function is used to pick out values for only one station. These are then converted to a numpy `ndarray` so that sparseSpACE can work with them.
Since the sparseSpACE framework may want to evaluate the function on relatively precise points the pandas library [13] may export them in a csv using the exponent format (e.g. "0.123e-9"). This format is not understood by the LARSIM model. We had to modify the Larsim_Utility_Set library to specifically instruct pandas to format floating point numbers using ten decimal digits and never use the exponent alternative.

5. Tests

We performed all of our experiments on a LCX Linux container hosted on a Proxmox Virtual Environment providing 32 CPUs (Xeon E5-2630 v4 @ 2.20GHz), 32 GByte of RAM and 100 GByte disk space. We used the `configuration_larsim_updated_lai.json` configuration file which is provided with [11]. It assumes the uniform distribution over different Leaf Area Index (LAI) parameters. The LAI is specific to different area types and months. Our setup uses four area type, month pairs as uncertain parameters. The Quantity of Interest was the "Abfluss Messung".

5.1. Testing the parallelisation

One of the first weaknesses we noticed was a rapidly growing memory usage, which eventually lead to out-of-memory faults. However we also observed a small but considerable growth when using a single-processed unmodified application. This effect was scaled by the multi-process environment and it caused a lack of resources in a shorter amount of time. Through experimentation we could isolate the effect. It was only present when using the gPCE approach for Uncertainty Quantification. When calculating the expectation and variance directly memory usage was stable. We also observed that more disk IO actions were performed when we disabled the gPCE interpolation. Since LARSIM relies heavily on external configuration files we suspect that the gPCE method also produced a considerable overhead. All following experiments used the direct way of calculation.
It is possible that the `Integrator` only receives a smaller portion of the total amount of grid points to be evaluated. This occurs for example when applying the combination technique: For each of the anisotropic subgrids integration is performed once, which results in only a fraction of the total workload being known to the integrator at a single call. On lower levels this means that the `IntegratorParallelAbitraryGrid` may not be able to make full use of system resources due to there being less nodes than processes. Also adaptive refinement techniques may add only a small number of points at a time. Therefore it is important to measure how many points can be evaluated in parallel to determine the value gained by parallel evaluation for a specific method.
Therefor, we log how many points are gathered at each call of the Integrator and how many of them are yet uncached, meaning the true work that has to be done. We compare a static Sparse Grid approach with an increasing maximum level and an Adaptive Grid technique. For the Sparse Grid case we can also regard the average amount of points evaluated per call at different levels. We expect to observe more points per call at a higher level since the subgrids

will contain more nodes at higher levels.
For a static Sparse Grid measurement we use the Gauss-Legendre Grid as basis and count the work packets for each increasing level. We finished computations up to level 5 and included partial results for level 6. In Figure 5.1 we show histograms which count work packets grouped by their length at different levels. One can observe a very large growth of work packet size with increasing level and at smaller levels the packets are already considerably large. The adaptive technique only uses the combination scheme to set a basis for later refinement and a comparison by level is not really sensible since most of the work should be done during refinement. In Figure 5.2 we can see a count work packets consisting of nodes an integrator had to evaluate at once grouped by their length. We also determined how many of them were already cached in order to determine the real work done. We can clearly observe that large parts of the calculation have already been cached and usually only less than 30 points have to be evaluated. This is however not universally applicable to every function and dimensionality. We strongly advice to make own measurements to determine the optimal assignment of physical resources.
Measuring the communication overhead on the adaptive technique we found that the longest a process has to wait until all others have finished evaluating is on average $6.893747679794891s$. This considerable overhead is probably caused by some threads receiving less workload than others. This is confirmed by Figure 5.3 which shows how the waiting times were distributed. One can clearly observe small groups of outliers around 30 and 50 seconds which roughly correspond to multiples of a single runtime.

5.2. Error Calculation

The theoretical error convergence rates of different integration techniques presume certain properties of the function f that is evaluated. Some of these are for example smoothness or Riemann integrability. Therefor, we have to evaluate and compare different integration techniques to be able to make any meaningful statement about their effectiveness.
This is usually done by comparing the result of a simulation to a measured reference solution. We would compare the errors of multiple techniques at different numbers of evaluated nodes. This would allow us to compare their real convergence rates.
Although single measured values can be obtained to calculate an error estimate for a deterministic forward pass, no such reference solution exists for the probabilistic results of a Forward Uncertainty Quantification. The variance for example cannot be observed from a single data point.
Another approach has to be chosen. To verify our solution is correct we compare it to the result of a simulation using the Monte-Carlo method (N=2025). Figure 5.5 shows the 2-norm of expectation and variance with respect to the number of samples calculated by the Monte-Carlo simulation. We can see the Monte-Carlo approach converges quite cleanly. We also plot the expectation and variance result of a Sparse Grid simulation that used a Gauss-Legendre Grid as basis (Figure 5.4). This approach also converges but to slightly different values. Figure 5.6 shows the error in dependence of the number of points evaluated using a Sparse Grid with

a Gauss-Legendre Grid as basis. Using this visualization we cannot confirm a convergence towards the Monte-Carlo result. The deviation however is relatively small.
Since we compute a time series of means and variances the error is normalized using the 2-norm. This is a measure of multidimensional distance from the reference to the current solution.
We also tested an Adaptive Grid approach using a Trapezoidal Grid as basis. In this run we did not take the result at different numbers of evaluated points. In the end we obtained an 2-norm-reduced error of 1.6998841230065735 in the expectation and of 1.4722099789459928 at 573 evaluated points. If we divide the error by the mean of the respective value before applying the norm we get 0.005958529231525151 for expectation and 23.571891082425328 for the error in variance. We note that this approach converges to the same values as the standard combination technique. When comparing the results of the Adaptive Grid to the high-level Sparse Grid estimates the 2-norm reduced error is considerably smaller at only 0.3907605946223124 in expectation and 0.24071648764009526 in variance.
Therefor, we suspect a small numerical error introduced by the sparseSpACE framework or our custom implementation of the Monte-Carlo approach which used the chaospy library [14] and did not rely on sparseSpACE.
The results we obtained from each of the different techniques is shown in Figure 5.7.

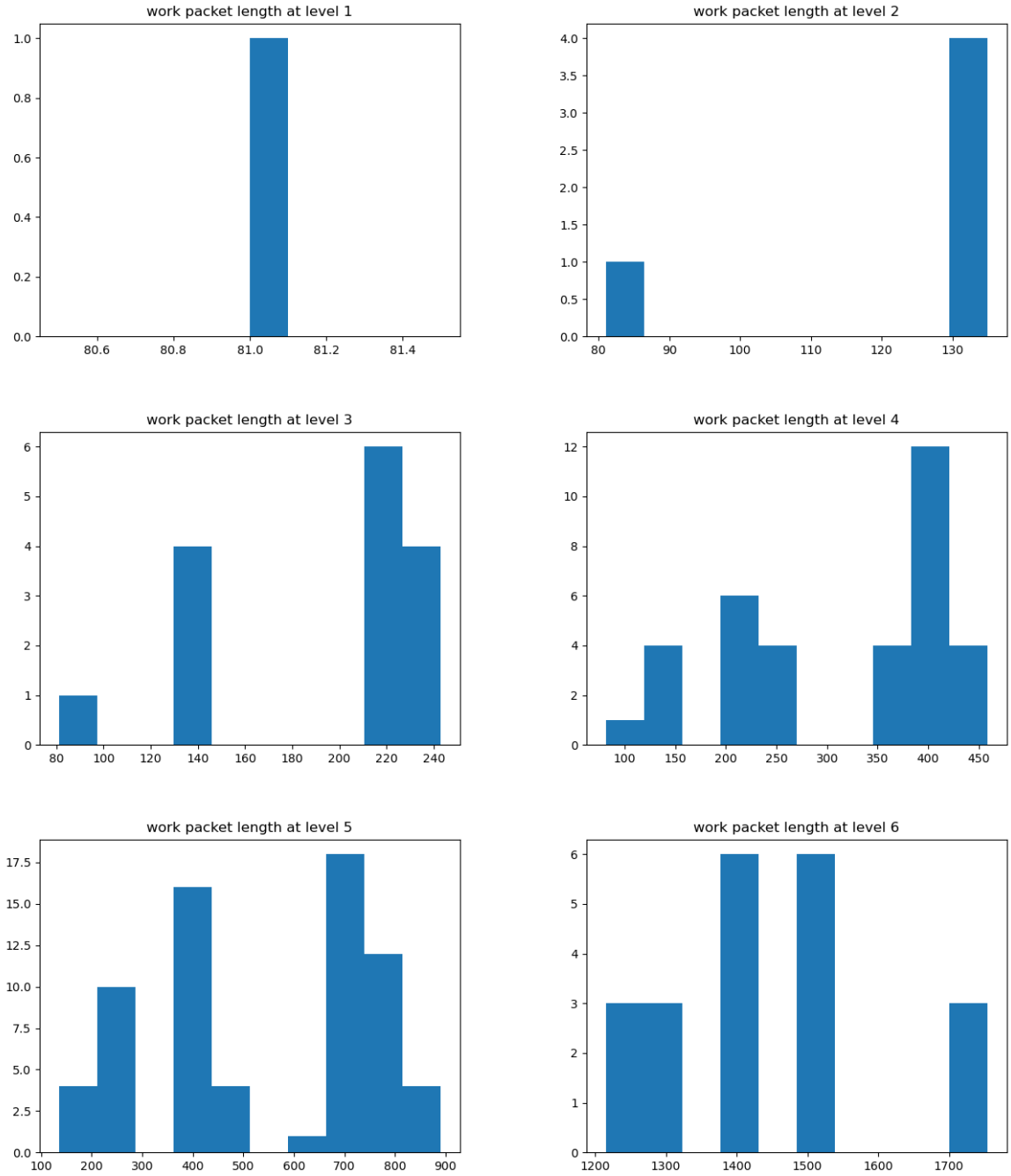

Figure 5.1.: The top plot shows work packet count grouped by their total length at different levels. We note that level 6 is not a finished computation but has been included for additional insight.

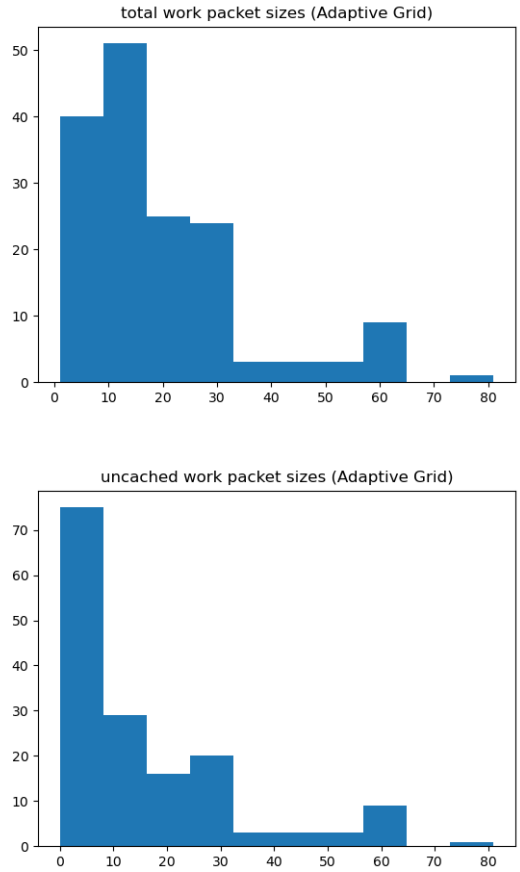

Figure 5.2.: The top plot shows work packet count grouped by their total length for the adaptive technique. The bottom plot shows work packet count group by the number of uncached nodes.

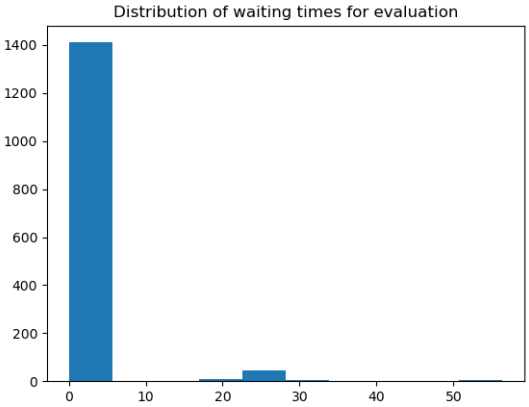

Figure 5.3.: A count of waiting times grouped by their lengths.

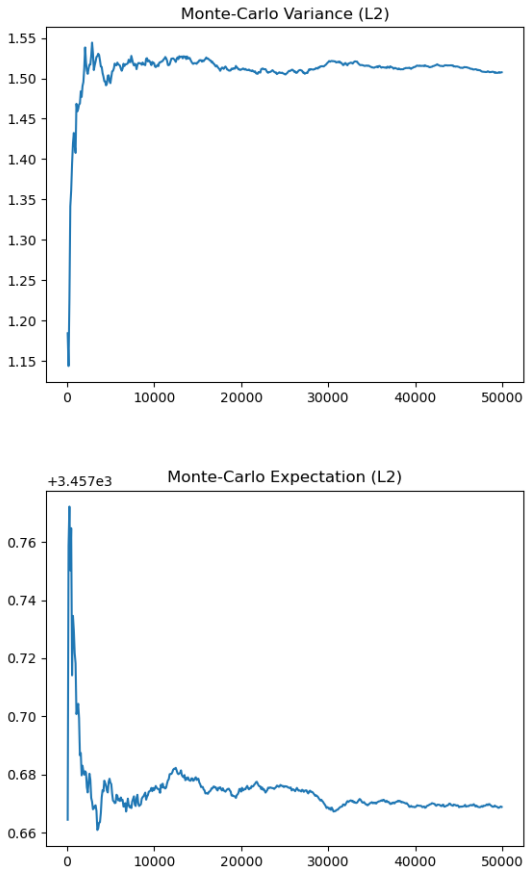

Figure 5.4.: This figure shows the expectation and variance results produced by the Monte-Carlo simulation in dependence of the number of evaluated points. Both have been reduced using the 2-norm.

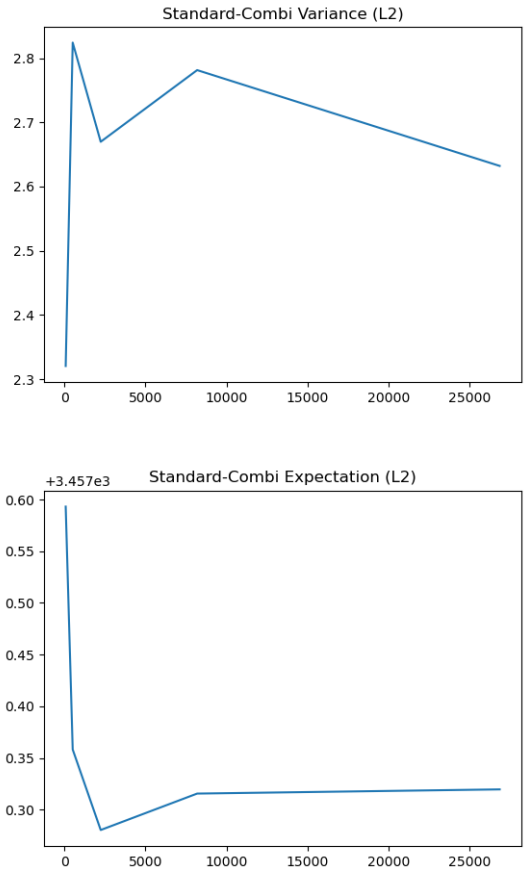

Figure 5.5.: This figure shows the expectation and variance results produced by a Sparse Grid approach with a Gauss-Legendre basis in dependence of the number of evaluated points. Both have been reduced using the 2-norm.

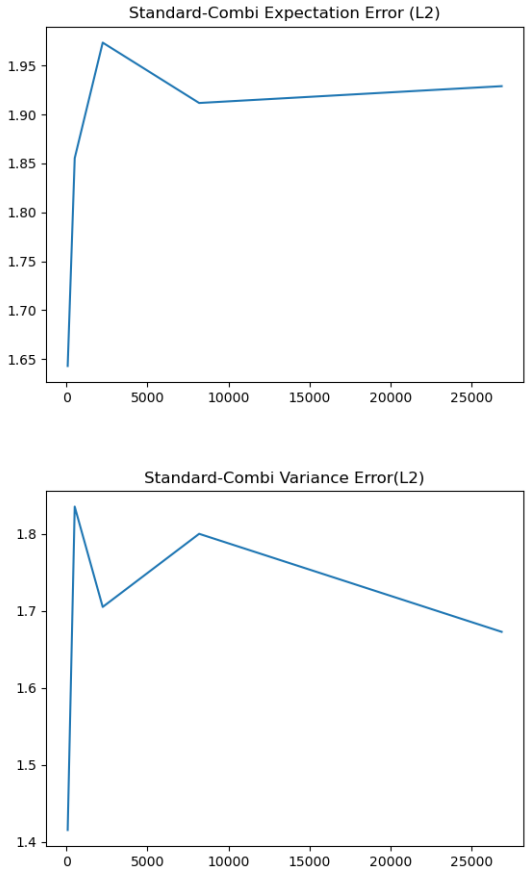

Figure 5.6.: A visualization of the convergence of the combination techniques using a Gauss-Legendre grid as basis. It show the error reduced by the 2-norm using the Monte-Carlo result at $N = 50000$ as reference solution.

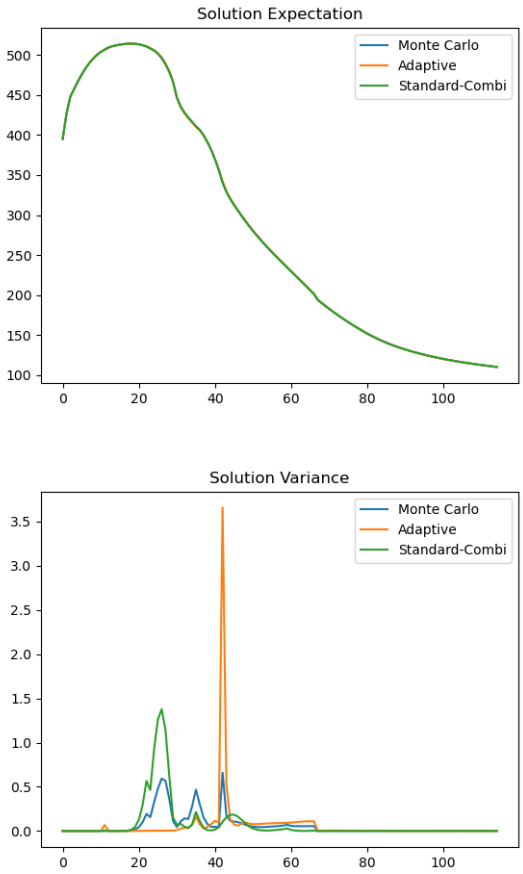

Figure 5.7.: We show the result of our three different simulations which where time series with an interval of 1-hour increments. The top one shows the simulated expectation. Only a single line is visible since the deviations are so small they overlap. The bottom figure shows the estimation of variance where some divergence exists.

6. Discussion and Future Work

We have shown a promising way to parallelize the execution of a single model within a Sparse Grid framework and apply it to Uncertainty Quantification. The results of small test calculations are within reasonable error margins which confirms that we did not introduce any considerable error. However a small numerical difference in results between the sparseSpACE and custom Monte-Carlo implementation exists. Although it is almost negligible for this application it could scale and introduce a more considerable error for others.
Although a speed-up is definitely recognizable no quantitative measurements have been made regarding the total required time. Theoretically we can imagine that the only limit of speed is the execution time of the model if the system resources can match the number of points in each work packet. In this scenario however we neglect overhead that might be worth measuring by logging the amount of idling time or more specifically the time that processes generally spent waiting for blocking MPI calls.
The total overhead should also have been measured since the current architecture is very symmetrical. Almost all processes have to execute the same steps. An asymmetrical master-slave architecture could be implemented where the master process determines grid points and computes results while worker processes are only responsible for function evaluations. This method has the potential to reduce redundant computation and therefore RAM and energy usage. We do not think it would lead to a large reduction of computation time because worker processes would have to wait until the master gives them new instructions.
However there are still ways to optimize the current implementation. One of its flaws is for example its limitation to integration based operations. Other operations like density estimation and interpolation are still only available in serial execution models.
A possible solution to this would be a centralized evaluator which receives all points that have to be evaluated. Another advantage could be a buffering mechanism that implements a lazy evaluation system which only triggers execution of the model once a result is specifically requested. This would allow the evaluator to store as many points as possible before executing them in parallel and maximize the use of all system resources. An implementation in the sparseSpACE framework would be difficult because it requires refactoring of large portions of the codebase to fully make use of a buffering mechanism.
Another possible way to solve this problem could be a parallelisation on the subgrid level. Each subspace of nodes could be evaluated separately and in parallel. This method is not limited to any specific operation. However when the maximum level and/or the number of dimensions is low only a small amount of subgrids will exist which is possibly lower than the number of processes available.
For our tests we only used the LARSIM model with a relatively short execution time. In this case we did not have to worry too much about waiting times which arise from parameter

dependent evaluation time. On lower levels the overall time is sufficiently small to make slight imbalances negligible and on on larger levels the imbalances should cancel out because of a higher number of nodes per work packet. When using a model with considerably larger execution time, significant waiting times could arise.
A potential drawback of using sparseSpACE is the growing memory usage when using the gPCE approach with sparseSpACE, which added about 1 MB for every evaluation of the problem function.
Specific to our implementation there might also be a problem when scaling the size of the cache. Since the entire cache has to be transmitted to the master process for merging a significant overhead could be introduced when evaluating a large number of nodes especially with a long time series as output.
Considering the amount of complications when dealing with parallel evaluation in sparseSpACE one might consider using the Monte-Carlo approach that we originally intended as reference solution. We noticed it is in comparison to sparseSpACE easy to parallelise and introduces less overhead. Furthermore its only restriction for scaling up the parallel evaluation is the overall number of points which we want to evaluate. Additionally it has also shown a significantly faster and cleaner convergence specifically for the LARSIM application.

A. General Addenda

The digital appendix contains the Python scripts which where used to perform the experiments:

- **larsim_chaospy.py** contains the code used for the Monte-Carlo simulation.
- **larsim_adaptive.py** uses sparseSpACE to perform an Adaptive Grid Integration.
- **larsim_combi.py** uses sparseSpACE to perform a Standard-Combination Sparse Grid Integration.

To execute the scripts one has to install the Larsim_Utility_Set as well as sparseSpACE. Also the Data required for LARSIM and its exectuable have to be present.
The file `parallel_integrator.py` contains both versions of our new parallel integrator. Our changes are also implemented in the `jonas-treplin-experimental` branch of the sparseSpACE Github project.
The configuration file `configuration_larsim_updated_lai.json` describes the configuration we used for our experiments
We also provide a git repository with all the files described here. It is accessible under `https://github.com/Xnartharax/forschungsarbeit-appendix.git`.

List of Figures

Bibliography

[1] M. Obersteiner. *sparseSpACE*. URL: https://github.com/obersteiner/sparseSpACE.

[2] K. Ludwig and M. Bremicker. "The Water Balance Model LARSIM – Design, Content and Applications". In: *Freiburger Schriften zur Hydrologie* 22 (2006).

[3] A. Liatsetskaya. "Adaptive Quadrature with the Combination Technique for UQ Applications". Bachelorarbeit. Technical University of Munich, June 2020.

[4] I.-G. Farcas. "Context-aware Model Hierarchies for Higher-dimensional Uncertainty Quantification". Dissertation. München: Technische Universität München, 2020.

[5] R. C. Smith. *Uncertainty Quantification: Theory, Implementation, and Applications*. USA: Society for Industrial and Applied Mathematics, 2013. ISBN: 161197321X.

[6] D. Xiu. *Numerical Methods for Stochastic Computations: A Spectral Method Approach*. USA: Princeton University Press, 2010. ISBN: 0691142122.

[7] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. address: München URL: http://www5.in.tum.de/pub/pflueger10spatially.pdf. München: Verlag Dr. Hut, Aug. 2010.

[8] T. Neckel. *Algorithms for Uncertainty Quantification Lecture 5: Aspects of Interpolation and Quadrature*. Lecture Slides. 2019.

[9] F. Hofmeier. "Applying the Spatially Adaptive Combination Technique to Uncertainty Quantification". Bachelorarbeit. Technical University of Munich, Sept. 2019.

[10] H. Möller. "Dimension-wise Spatial-adaptive Refinement with the Sparse Grid Combination Technique". Bachelorarbeit. Department of Informatics, Technische Universität München, Oct. 2018.

[11] I. Jovanovic Buha. *Larsim_Utility_Set*. 2020. URL: https://github.com/ivanajovanovic/Larsim_Utility_Set.

[12] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo. "Parallel distributed computing using Python". In: *Advances in Water Resources* 34.9 (2011). New Computational Methods and Software Tools, pp. 1124–1139. ISSN: 0309-1708. DOI: https://doi.org/10.1016/j.advwatres.2011.04.013. URL: http://www.sciencedirect.com/science/article/pii/S0309170811000777.

[13] T. pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[14] J. Feinberg and H. P. Langtangen. "Chaospy: An open source tool for designing methods of uncertainty quantification". In: *Journal of Computational Science* 11 (2015), pp. 46–57. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2015.08.008. URL: http://www.sciencedirect.com/science/article/pii/S1877750315300119.

Ich erkläre hiermit, dass ich die Forschungsarbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

Munich, 1.12.2020 Jonas Treplin