The present work was submitted to the Institute of Software and Tools for Computational Engineering

Randomized Subspace Quasi Taylor-Order Analysis

Master Thesis

Presented by Simon Lukas Märtens
Matr.-Nr. 345674

Supervised by Univ.-Prof. Dr. rer. nat. Uwe Naumann

Aachen, March 8, 2022

Contents

I Acknowledgements

II Abstract

III List of Figures

IV List of Definitions

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Thesis Structure . . . 1

2 Theory **3**
- 2.1 Regression . . . 3
  - 2.1.1 Taylor Model . . . 3
  - 2.1.2 Exponential Model . . . 4
  - 2.1.3 Periodical Model . . . 5
- 2.2 Separability . . . 5
- 2.3 Model Reduction . . . 8
- 2.4 Training and Model Reassembling . . . 8
- 2.5 Subspaces . . . 9
- 2.6 Sampling . . . 9

3 Implementation **13**
- 3.1 Sampling . . . 13
- 3.2 Model Reduction via Separability Analysis . . . 14
- 3.3 Subdomain Splitting . . . 16
- 3.4 TensorFlow Integration . . . 21
- 3.5 API . . . 21
  - 3.5.1 Python . . . 22
  - 3.5.2 `C++` . . . 22
- 3.6 Configuration Options . . . 22

4 Showcase **25**

5 Case Study (*Black-Scholes*) **31**

6 Conclusion **41**
- 6.1 Summary & Discussion . . . 41
- 6.2 Further Research . . . 41

V References

A Appendix
- A.1 YAML Configuration File . . .
- A.2 Source Code . . .

I Acknowledgements

I want to express my sincere gratitude to Professor Uwe Naumann, who accepted this Master thesis, helped me formulate an outline, and gave me valuable ideas for my thesis.

I also want to thank Uwe Naumann and my colleagues at the STCE for the collaboration and the lovely working environment in the last five years. Furthermore, I am thankful that Sebastian Christodoulou helped me find my way into the framework he wrote for his Bachelor's thesis, which this thesis uses.

Last but not least, I thank my family and friends, who supported me in every way possible during my Bachelor's and Master's studies in Aachen.

II Abstract

We present a new technique to reduce the dimensionality of a high-dimensional multivariate scalar function. Expanding the concept of *quasi-linearity* to arbitrary regression models, we build a framework that can assess the separability of functions and especially the *Quasi-Separability*. Combining this technique with other approximation methods, we can build accurate surrogate models at a lower cost. We use machine learning in order to approximate non-reducible functions. Furthermore, we introduce domain decomposition for the investigation of *Quasi-Separability* properties on subdomains.

III List of Figures

1 Subspace regression tree for Equation (37). 9
2 Random sampling in $\mathcal{X} \subset \mathbb{R}^2$ $(N = 9)$. 10
3 Full-factorial sampling in $\mathcal{X} \subset \mathbb{R}^2$ $(N = 3^2 = 9)$. 11
4 Latin-Hypercube sampling in $\mathcal{X} \subset \mathbb{R}^2$ $(N = 9)$. 11
5 Class diagram of the *RSQTOA* framework. 13
6 Class diagram of the sampling subpackage. 14
7 Class diagram of the *Quasi-Separability* regression analysis subpackage. . . . 14
8 Class diagram of the subspace splitting subpackage. 17
9 Domain **reduction** via further bisection. $f(x)$ is separable, $g(x)$ is not. . . . 19
10 Domain **expansion** via further bisection. $f(x)$ is separable, $g(x)$ is not. . . . 20
11 Surface of objective function Equation (42). 25
12 The subspace regression tree, calculated by our framework. 26
13 Reduced model of Equation (42) plotted as surface. 26
14 Regression errors of the reassembled function Equation (42). 27
15 Residuals from the pure machine learning approach (100000 samples). 28
16 Residuals from the RSQTOA guided approach (100000 samples). 28
17 Residuals from the pure machine learning approach (10000 samples). 29
18 Residuals from the pure RSQTOA guided approach (10000 samples). 29
19 Full regression analysis with ten subdomains. 30
20 *Black-Scholes* 2-D surface slices at the anchor point $(53, 58, 0.045, 0.015)$. . . 32
21 Loss (2-norm) history, learning the *Black-Scholes* model. Training set: 10000 samples. Test set: 2000 samples. Batch size: 256. 33
22 Loss (1-norm) surfaces, learning the *Black-Scholes* model. Training set: 10000 samples. Test set: 2000 samples. Batch size: 256. 34
23 Loss (2-norm) history, learning the *Black-Scholes* model. Guided by RSQTOA **without** domain decomposition. 36
24 Loss (1-norm) surfaces, learning the *Black-Scholes* model. Guided by RSQTOA **without** domain decomposition. 37
25 Loss (2-norm) history, learning the *Black-Scholes* model. RSQTOA **with** domain decomposition. 38
26 Loss (1-norm) surfaces, learning the *Black-Scholes* model. Guided by RSQTOA **with** domain decomposition. 39

IV List of Definitions

1 Definition (Partial Separability) 5
1 Example 5
2 Definition (Dimensional Separability) 6
3 Definition (Full Separability) 6
2 Example 6
4 Definition (Quasi-Separability) 7
1 Corollary (Quasi-Taylor) 7
2 Corollary (Quasi-Exponential) 7
3 Corollary (Quasi-Periodic) 7
3 Example 7
5 Definition (Quasi-Reducibility) 8
4 Example 9

1 Introduction

1.1 Motivation

This thesis is devoted to finding a new approach to mitigate the effects of the *Curse of Dimensionality* in regression methods. High-dimensional multivariate functions are very prominent in large-scale simulations. Finding approximations for computationally expensive models becomes increasingly difficult with the number of dimensions. Many techniques are trying to reduce the complexity of high-dimensional problems. For years, researchers and practitioners in Computational Science, Engineering, and Finance have aimed to build cheap surrogate models for computationally expensive simulation programs.

Surrogate models are an essential tool to avoid expensive simulations when the exact result is less important than the time and energy used for the simulation. Imagine a design process tightly coupled with a market simulation that will predict the impact of specific design choices on the business. For the sake of argument, let us say the simulation takes twenty minutes to produce a result. The work would be very tedious and expensive - an ideal use case for a surrogate model. Instead of waiting twenty minutes, even for small changes, the surrogate model could potentially produce a result in seconds. Even if that result is not entirely accurate, it will help speed up the design process, and in the end, one can validate the results with the accurate simulation.

Our approach tries to reduce the approximation cost of those large problems by changing the one thing that makes the problem difficult - the number of dimensions. Knowledge about unique mathematical properties of the function to be approximated can help decrease dimensionality. One example for such a property is *quasi-linearity* which is the research topic in Kauerauf [2021]. We expand this concept to arbitrary ansatz functions focusing on Taylor, exponential and periodic approximations. We introduce *Quasi-Separability* as a more extensive concept that includes *quasi-linearity*.

Quasi-Separability allows us to reduce the expensive model in its dimensionality. We can then apply other techniques to approximate the reduced model. We chose approximation by machine learning as the backend technique based on the work of Christodoulou [2020].

Not only are we considering *Quasi-Separability* on the entire domain but also on subdomains. Although this will make the task exponentially more complex, the prospects are visible. Understanding the inner dynamics of complex models is per se a difficult task, let alone on subdomains. Our framework can detect such properties and use them in a meaningful way.

We take notes from the works of Owen [2000] and Chui and Diamond [1987] which also consider higher-dimensional regressions. An and Owen [2001] introduce *Quasi-Regressions* for function approximations on high-dimensional unit cubes. Müller and Zhao [1999] and Chaudhuri et al. [1994] also considered domain decompositions and their findings influence our approach.

1.2 Thesis Structure

First, we introduce the concepts of scalar regression as a way to reduce the dimensionality of models in Section 2. We show how to reduce a model, approximate the remaining dimensions and then reassemble everything. Sampling techniques and domain decomposition are discussed as well in the theoretical section.

Following the theory, in Section 3 we present our implementation. We discuss our algorithms for regression analysis and domain decomposition in detail.
The implementation is applied to an artificial example in Section 4 which demonstrates the improvements our framework can provide. We investigate a real-world example in Section 5. We finish with a discussion of the results and an outlook for future considerations.

2 Theory

This chapter concerns the mathematical background and theoretical considerations for our framework. First, we recapitulate scalar regression analysis, the primary tool used in our framework. *Separability of dimensions* and *Reducibility of the model* are the other concepts that we introduce and combine with the regression analysis. This combination leads to a new approach for higher-dimensional approximation.

2.1 Regression

In regression analysis, we usually have a certain amount data consisting of samples $\boldsymbol{x} \in \mathbb{R}^m$ with their corresponding observations $\boldsymbol{y} \in \mathbb{R}^m$. Observations are obtainable via experiments, simulations or simply evaluations of an objective function. For our purposes we consider only scalar regression models $\tilde{f} : \mathbb{R} \to \mathbb{R}$ with $x \mapsto \tilde{f}(x; \boldsymbol{\theta})$ and parameters $\boldsymbol{\theta} \in \mathbb{R}^n$. Applying the model to the data leads to

$$y_i = \tilde{f}(x_i; \boldsymbol{\theta}) + \epsilon_i \quad i = 1 \dots m \tag{1}$$

with ϵ_i the error residuals between the observed data and the model. The regression analysis tries to fit the model to the given data by tuning the parameters $\boldsymbol{\theta}$ such that the sum of squared residuals is minimized:

$$\min_{\boldsymbol{\theta}} \|\boldsymbol{\epsilon}\|_2^2 = \min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \epsilon_i^2 = \min_{\boldsymbol{\theta}} \sum_{i=1}^{m} \left(y_i - \tilde{f}(x_i; \boldsymbol{\theta}) \right)^2 . \tag{2}$$

We do not need to use the Euclidian norm, but it is one of the most preferred choices due to its simplicity. It gives the problem its usual name '*least-squares problem*'.
The biggest distinction between different regression models is whether they are linear or non-linear with respect to the parameters $\boldsymbol{\theta}$. Linear models have an easy closed-form solution in form of a linear system of equations where non-linear models need some iterative optimization technique. We implemented three different regression models: A linear taylor model which fits polynomials of arbitrary order, a non-linear exponential model as well as a non-linear periodic model.

2.1.1 Taylor Model

The taylor regression model of order n is defined as $\tilde{f} : \mathbb{R} \to \mathbb{R}$ with

$$\tilde{f}(x) = \sum_{k=0}^{n} \theta_k * x^k \tag{3}$$

with parameters $\boldsymbol{\theta} \in \mathbb{R}^{n+1}$. The regression analysis leads to the following linear system of equations:

$$\boldsymbol{X\theta} = \mathcal{Y} \quad \Longleftrightarrow \quad \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} * \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \tag{4}$$

with $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ containing the coefficients of the parameters $\boldsymbol{\theta} \in \mathbb{R}^n$ in the Taylor model, evaluated at the samples and $\boldsymbol{y} \in \mathbb{R}^m$ the observations. Our framework will ensure that

$m >= n$, thus the least-squares problem is always over determined. There are various methods to solve this least-squares problem, one of the easiest being the *normal equation*:

$$\boldsymbol{X}^T \boldsymbol{X} \boldsymbol{\theta} = \boldsymbol{X}^T \mathcal{Y} \tag{5}$$

If $\boldsymbol{X}$ has a full column rank, which is the case using the Taylor model with $m > n$ unique samples, the matrix $\boldsymbol{X}^T \boldsymbol{X}$ is symmetric positive definite (SPD). That allows us to use the *Cholesky-Decomposition* described in Dahmen and Reusken [2008]. One problem that arises using the *normal equation* is the poor handling of ill-conditioned problems. Since the condition number of $\boldsymbol{X}$ gets effectively squared, we cannot solve ill-conditioned problems accurately. Other methods like *Gram-Schmidt orthogonalization* described in Björck [1967] can be used instead, but for our purposes the *normal equation* is sufficient.

2.1.2 Exponential Model

Our exponential regression model is defined as $\tilde{f} : \mathcal{D} \subset \mathbb{R} \to \mathbb{R}$ with

$$\tilde{f}(x) = a * e^{b*x} + c \tag{6}$$

with parameters $a, b, c \in \mathbb{R}$. Unlike the taylor regression model in Section 2.1.1, the exponential model is non-linear with respect to the parameters. Therefore, we need other methods like *Gradient-Descent*, *Gauss-Newton* or *Levenberg-Marquart*. The latter being a combination of the previous ones as described in Gavin [2019].
In order to achieve a fast convergence, a good initial guess for the parameters is required. Let us consider the derivative of the exponential regression model, defined as $\tilde{f}' : \mathcal{D} \subset \mathbb{R} \to \mathbb{R}$ with

$$\tilde{f}'(x) = a * b * e^{b*x} \tag{7}$$

on a closed domain $\mathcal{D} = [x_{lb}, x_{ub}] \subset \mathbb{R}$. We can get an estimate for the parameter b with the following equation:

$$\begin{aligned} & \frac{\tilde{f}'(x_{ub})}{\tilde{f}'(x_{lb})} = \frac{a * b * e^{b*x_{ub}}}{a * b * e^{b*x_{lb}}} = e^{b*(x_{ub}-x_{lb})} \\ \iff \quad & \log\left(\frac{\tilde{f}'(x_{ub})}{\tilde{f}'(x_{lb})}\right) = b * (x_{ub} - x_{lb}) \\ \iff \quad & \frac{1}{x_{ub} - x_{lb}} * \log\left(\frac{\tilde{f}'(x_{ub})}{\tilde{f}'(x_{lb})}\right) = b \end{aligned} \tag{8}$$

This approximation only works if $\tilde{f}'(x_{lb})$ and $\tilde{f}'(x_{ub})$ have the same sign. Using the midpoint of the domain $x_m = \frac{1}{2} * (x_{ub} - x_{lb})$ and the guess for parameter b, we can derive the parameters a and c as

$$\tilde{f}'(x_m) = a * b * e^{b*x_m} \quad \iff \quad \frac{\tilde{f}'(x_m)}{b * e^{b*x_m}} = a \tag{9}$$

and

$$\tilde{f}(x_m) = a * b * e^{b*x_m} + c \quad \iff \quad \tilde{f}(x_m) - a * b * e^{b*x_m} = c. \tag{10}$$

If the approximation for the initial guess is not possible, we can use $a = 1$, $b = 1$ and $c = 0$.

2.1.3 Periodical Model

The periodic regression model is defined as $\tilde{f} : \mathbb{R} \to \mathbb{R}$ with

$$\tilde{f}(x) = a * \sin(b * x + c) \tag{11}$$

with parameters $a, b, c \in \mathbb{R}$. Similar to the exponential model in Section 2.1.2, the periodic regression model is non-linear and needs to be solved with a non-linear solver.
The initial guess uses the periodicity of the model. Assuming the data that we want to fit is actually periodic, we try to find a minimum and a maximum within the domain:

$$x_{min} = \arg\min_{x \in \mathcal{D}} f(x) \quad \text{and} \quad x_{max} = \arg\max_{x \in \mathcal{D}} f(x). \tag{12}$$

Since the parameter a is the amplitude of the periodic model, our initial guess is

$$a = \frac{f(x_{max}) - f(x_{min})}{2}. \tag{13}$$

For the frequency, represented by the parameter b, we can use the fact, that the distance between the maximum and the minimum is directly correlated to the frequency, resulting in

$$|f(x_{max}) - f(x_{min})| = \frac{\pi}{b} \quad \Longleftrightarrow \quad b = \frac{\pi}{|f(x_{max}) - f(x_{min})|}. \tag{14}$$

Lastly, we can deduce the phase shift parameter c using the guessed frequency with

$$c = \left(x_{max} - \frac{\pi}{2 * b}\right) \mod \frac{2\pi}{b} \tag{15}$$

and if the calculated c is outside $\left[-\frac{\pi}{b}, \frac{\pi}{b}\right]$, we can add or subtract $\frac{\pi}{2*b}$. That guarantees that we use the smallest phase shift possible.
Notice that the periodic regression model does not have a constant term, unlike the exponential and Taylor models. Instead, we shift the observations $\tilde{y}_i = y_i - (f(x_{max}) - a)$. Assuming the data is periodic, any constant term in the data should vanish.

2.2 Separability

In order to mitigate the effects of the *Curse of Dimensionality*, we need to reduce the number of dimensions. We define the concepts of *Separability* and *Quasi-Separability* as tools for dimensionality reduction.

Definition 1 (Partial Separability) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$. We call the function* ***partially separable*** *if and only if the function can be written as*

$$f(\boldsymbol{x}) = \tilde{f}_0\left((x_i)_{i \in \mathcal{I}_0}\right) + \tilde{f}_1\left((x_i)_{i \in \mathcal{I}_1}\right) \tag{16}$$

with functions $\tilde{f}_0 : \mathbb{R}^{|\mathcal{I}_0|} \to \mathbb{R}$ and $\tilde{f}_1 : \mathbb{R}^{|\mathcal{I}_1|} \to \mathbb{R}$ and two disjoint subsets of indices of input variables $\mathcal{I}_0$ and $\mathcal{I}_1$ (Deussen [2021]).

Example 1 *Consider a function $f : \mathbb{R}^4 \to \mathbb{R}$ with*

$$f(\boldsymbol{x}) = e^{x_1 + x_2} - x_3 * x_4. \tag{17}$$

The function is partially separable with

$$\tilde{f}_0(x_0, x_1) = e^{x_1 + x_2} \quad \textit{and} \quad \tilde{f}_1(x_2, x_3) = -x_3 * x_4 \tag{18}$$

since we can write it as

$$f(\boldsymbol{x}) = \tilde{f}_0(x_1, x_2) + \tilde{f}_1(x_3, x_4). \tag{19}$$

For our purposes, we will only consider separations where at least one of the two index subsets only contains a single element. With this in mind, we introduce our definitions of *Dimensional Separability* and *Full Separability.*

Definition 2 (Dimensional Separability) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$. We call a dimension* ***separable*** *if and only if the function is additive with respect to the corresponding input variable:*

$$\textit{dimension } i \textit{ is } \textbf{\textit{separable}} \iff f(\boldsymbol{x}) = \tilde{f}(x_1, \ldots, x_{i-1}, x_{i+1}, x_n) + \phi(x_i) \tag{20}$$

with $\tilde{f} : \mathbb{R}^{n-1} \to \mathbb{R}$ and $\phi : \mathbb{R} \to \mathbb{R}$

Definition 3 (Full Separability) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$. We call the function* ***fully separable*** *if and only if all dimensions are separable. That means the function can be written as*

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \phi_i(x_i) + C \tag{21}$$

with functions $\phi_i : \mathbb{R} \to \mathbb{R} \quad \forall i = 1 \ldots n$ and a constant $C \in \mathbb{R}$.

Example 2 *Consider a function $f : \mathbb{R}^3 \to \mathbb{R}$ with*

$$f(\boldsymbol{x}) = e^{x_1 + x_2} - sin(x_3) + 3. \tag{22}$$

The dimension 3 is separable with

$$\tilde{f}(x_0, x_1) = e^{x_1 + x_2} + 3 \quad \textit{and} \quad \phi(x_3) = -sin(x_3) \tag{23}$$

since we can write it as

$$f(\boldsymbol{x}) = \tilde{f}(x_1, x_2) + \phi(x_3). \tag{24}$$

The function is not fully separable, because $\tilde{f}$ cannot be separated further. In contrast, consider the function $g : \mathbb{R}^3 \to \mathbb{R}$ with

$$g(\boldsymbol{x}) = e^{x_1} + tan(x_2) - \frac{x_2}{10} - sin(x_3) + 3. \tag{25}$$

The function is fully separable with

$$\phi_1(x_1) = e^{x_1}, \quad \phi_2(x_2) = tan(x_2) - \frac{x_2}{10}, \quad \phi_3(x_3) = -sin(x_3) \quad \textit{and} \quad C = 3 \tag{26}$$

such that

$$f(\boldsymbol{x}) = \sum_{i=1}^{3} \phi_i(x_i) + C. \tag{27}$$

Unfortunately, this property is rarely applicable to real-world models. There are often dependencies between the dimensions, which invalidate the *Separability*. It is also impossible to consider all scalar ansatz functions as a choice for ϕ. We want to consider only a small subset of ansatz function and chose the one with the best accuracy. Here we can use a weaker definition of *Separability* i.e. *Quasi-Separability* which introduces a trade-off between accuracy and separability.

Definition 4 (Quasi-Separability) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$ and an error tolerance $\epsilon \geq 0$. We call a dimension **quasi-separable** if and only if the error of the additive approximation, caused by separating the dimension i, is smaller than ϵ:*

$$\textit{dimension } i \textit{ is } \textbf{\textit{quasi-separable}} \iff \left| f(\boldsymbol{x}) - \left(\tilde{f}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) + \phi(x_i) \right) \right| \leq \epsilon \tag{28}$$

with $\tilde{f} : \mathbb{R}^{n-1} \to \mathbb{R}$ and $\phi : \mathbb{R} \to \mathbb{R}$.

With this definition, *Separability* is a special case of *Quasi-Separability* with $\epsilon = 0$.
The ansatz function ϕ is an arbitrary scalar regression. We use a subset of regressions for which we introduce some corresponding notations.

Corollary 1 (Quasi-Taylor) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$. We call a dimension **quasi-taylor of order n** if and only if the dimension is quasi-separable and*

$$\phi(x) = \sum_{k=0}^{n} a_k * x^k \quad \textit{with} \quad a_k \in \mathbb{R} \quad \forall k = 0 \ldots n. \tag{29}$$

Corollary 2 (Quasi-Exponential) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$. We call a dimension **quasi-exponential** if and only if the dimension is quasi-separable and*

$$\phi(x) = a * e^{b*x} + c \quad \textit{with} \quad a, b, c \in \mathbb{R}. \tag{30}$$

Corollary 3 (Quasi-Periodic) *Considering the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$. We call a dimension **quasi-periodic** if and only if the dimension is quasi-separable and*

$$\phi(x) = a * sin(b * x + c) \quad \textit{with} \quad a, b, c \in \mathbb{R}. \tag{31}$$

Example 3 *Consider a function $f : \mathbb{R}^2 \to \mathbb{R}$ with*

$$f(\boldsymbol{x}) = 2 * x_1 - cos(x_2) + 0.01 * g(\boldsymbol{x}) + 3 \tag{32}$$

and $g : \mathbb{R}^n \to [-1, 1]$ which represents an arbitrary non-separable function that depends on all input variables. Following the definitions, the function is not separable because separability is a transitive property. Here the concept of Quasi-Separability can provide an approximation. If we allow an error of $\epsilon \geq 0.01$ we can deduce that f is quasi-taylor of order $n = 1$ because the image of g is bounded:

$$\left| f(\boldsymbol{x}) - \left(\tilde{f}(x_2) + \phi(x_1) \right) \right| = |0.01 * g(\boldsymbol{x})| \leq \max_{\boldsymbol{x}} |0.01 * g(\boldsymbol{x})| = 0.01 \leq \epsilon \tag{33}$$

where $\tilde{f} : \mathbb{R} \to \mathbb{R}$ with $\tilde{f}(x_2) = -cos(x_2) + 3$ and $\phi : \mathbb{R} \to \mathbb{R}$ with $\phi(x_1) = 2 * x_1$.

The non-separable function g in the previous example could represent a complex dependency in the original model. Another option is that the function represents an error term which stems from noisy measurements if we are investigating samples from real-world processes.

2.3 Model Reduction

The main objective is to separate a given function into one part that we can approximate via a given set of scalar regression models and one where this is impossible. From now on, we will consider only finite domains for all dimensions.
Using the concept of *Quasi-Separability* we can define what it means to be able to reduce a function.

Definition 5 (Quasi-Reducibility) *Considering the function $f : \mathcal{X} \to \mathcal{Y} \subset \mathbb{R}$ defined by $\boldsymbol{x} \mapsto f(\boldsymbol{x})$ with a finite domain $\mathcal{X} \subset \mathbb{R}^n$ and a given error tolerance $\epsilon > 0$. We call the function* ***quasi-reducible*** *in $\mathcal{X}$ if and only if at least on dimension is quasi-separable with an accuracy of ϵ.*

Detection of *Quasi-Separability* In order to detect quasi-separable dimensions, we investigate a randomized approach. Let $f : \mathcal{X} \subset \mathbb{R}^n$ be the domain of a function we want to analyze. We want to assess the *Quasi-Separability* of dimension i, considering an error threshold of ϵ. We create m random samples in the entire domain. For each sample all dimension $j \neq i$ are fixed, resulting in m different scalar sub-functions. Using one of the introduced regression models (Section 2.1) we fit all scalar sub-functions. The coefficients of the m regressions are averaged, resulting in one final approximation $\phi_i(x_i)$ for the separated dimension i.
Using the approximation and the definition of the *Quasi-Separability* we can construct the reduced model $\tilde{f} : \mathbb{R}^{n-1} \to \mathbb{R}$ defined as

$$\tilde{f}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f\left(\boldsymbol{x}|_{x_i=\hat{x}_i}\right) - \phi_i(\hat{x}_i). \tag{34}$$

with $\hat{x}_i \in \mathcal{X}|_i$ a random point from the domain of dimension i.

Validation The reduced model plugged back into the *Quasi-Separability* condition gives us the following test equation:

$$\left|f(\boldsymbol{x}) - \left(f\left(\boldsymbol{x}|_{x_i=\hat{x}_i}\right) - \phi_i(\hat{x}_i) + \phi_i(x_i)\right)\right| \overset{!}{\leq} \epsilon \tag{35}$$

For validation of the reduced model we sample a certain amount of test pairs of the form $(\boldsymbol{x}, \hat{x}_i) \in \mathcal{X} \times \mathcal{X}|_i$. If the test equation holds for all test samples, we accept the dimension as separable, and the reduced model is considered valid.

2.4 Training and Model Reassembling

After the model reduction, we need to approximate the non-separable part of the function via different means. We chose to use neural networks based on the work of Christodoulou [2020]. Let us consider two disjoint subsets of indices for the separable and non-separable dimensions, $\mathcal{I}_0$ and $\mathcal{I}_1$. Subsequently, the reduced model is $\tilde{f} : \mathbb{R}^{|\mathcal{I}_0|} \to \mathbb{R}$ and the trained approximation is $\tilde{f}_{trained} : \mathbb{R}^{|\mathcal{I}_0|} \to \mathbb{R}$. Based on the *Quasi-Separability* definition, we can reassemble the model by adding all regressions to the trained approximation. We arrive at the final approximation $f_{trained} : \mathcal{X} \subset \mathbb{R}^n \to \mathbb{R}$ with

$$f_{trained}(\boldsymbol{x}) = \tilde{f}_{trained}\left((x_i)_{i \in \mathcal{I}_0}\right) + \sum_{i \in \mathcal{I}_1} \phi_i\left(x_i\right) \tag{36}$$

2.5 Subspaces

Quasi-Reducibility is rarely achievable if we consider the entire domain of the original function, at least not with reasonable accuracy. Therefore, we will explore *Quasi-Reducibility* on subdomains of the original function. Here bisection within a tree-structure like in Chaudhuri et al. [1994] is a reasonable approach. Unfortunately, this will introduce a new exponential complexity if we consider recursive bisection.

Example 4 *Consider a function* $f : [0,1]^3 \to \mathbb{R}$ *with*

$$f(\boldsymbol{x}) = x_0 + (sign(x_0 - 0.3) - 1) * (sin(x_0 - 0.3)) + 3.2 * x_1^2 - tan(x_2). \tag{37}$$

With an error tolerance of $\epsilon = 0$ *the function is of quasi-taylor order* 2 *in dimension* x_1 *on the entire domain. If we consider the subdomain* $[0, 0.3] \times [0,1]^2$ *we can also say that the function is of quasi-taylor order* 1 *in dimension* x_0*. The corresponding subspace tree structure is presented in Fig. 1.*

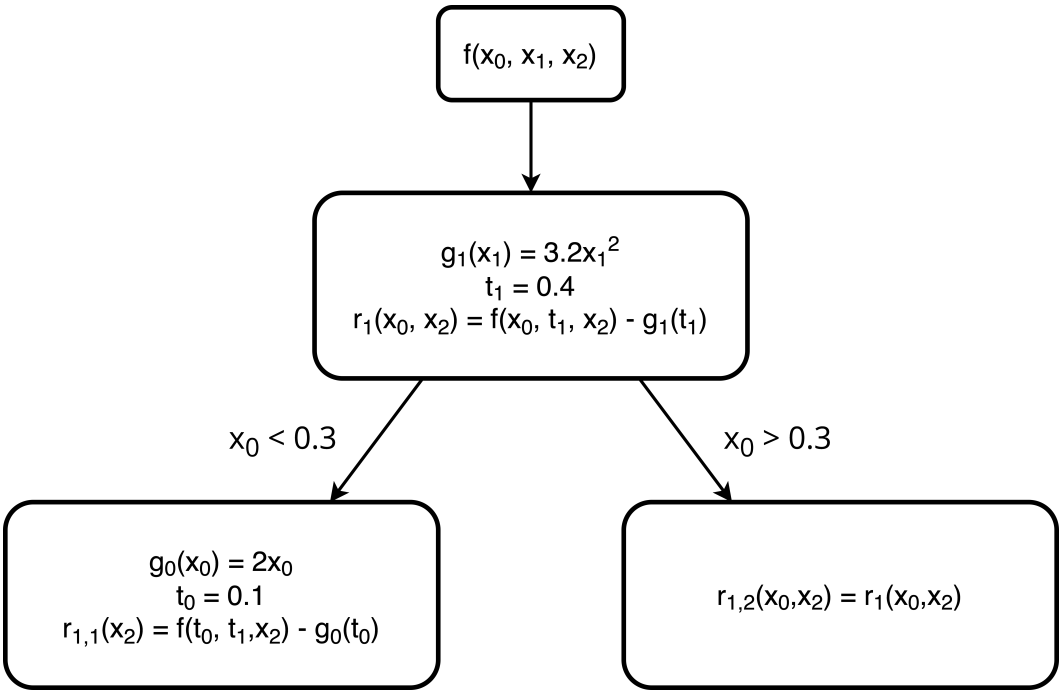


Figure 1: Subspace regression tree for Equation (37).

In the subspace tree each node corresponds to a certain subdomain. If *Quasi-Separability* is detected in the subdomain we create a reduced model by the method presented in Section 2.3. Using bisection techniques we can then split the model further and investigate the dimensions which haven't been separated yet.

2.6 Sampling

Sampling in high-dimensional domains is cumbersome since here we feel the *Curse of Dimensionality* the most. We considered the following sampling methods, all with uniform probability distributions (Wil [2008a]).

Random sampling Pure random sampling is the easiest to implement, but at the same time, it provides no guarantees about the final distribution of samples, especially in high-dimensional space. For example, the validation of the *Quasi-Separability* (Section 2.3) would need an enormous amount of samples to produce a confident result. One advantage of random sampling is that we can easily add samples without resampling everything. See Figure 2 for a random sampling example with 9 samples.

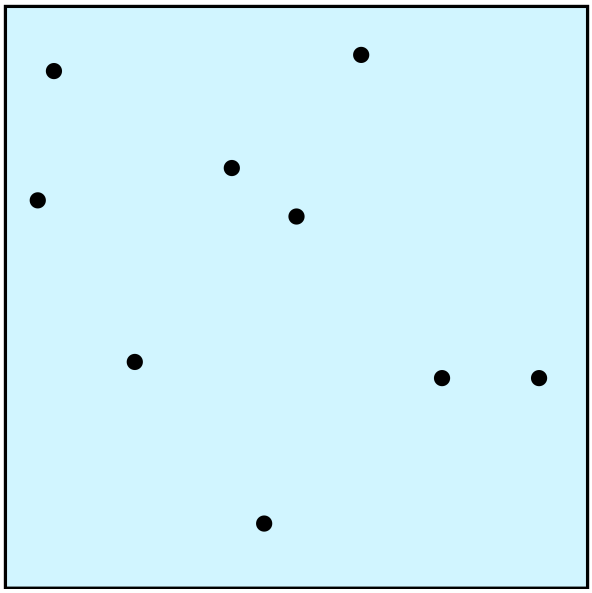


Figure 2: Random sampling in $\mathcal{X} \subset \mathbb{R}^2$ ($N = 9$).

Full-factorial sampling Full-factorial sampling cuts each dimension into N subspaces. Considering a domain $\mathcal{X} \subset \mathbb{R}^n$ this will result in N^n subdomains. In each subdomain, we sample one random point. This method results in a very uniform distribution. Ideally, this is the sampling method we want to use. Unfortunately, the number of samples will grow too fast in high-dimensional domains. We are limited to sample sets of size $k * N^n$, making this method infeasible. See Figure 3 for a full-factorial sampling example with 9 samples.

Latin-Hypercube sampling This method is a compromise between random sampling and full-factorial sampling. Like the full-factorial sampling, we divide each dimension into N subspaces. We have N^n subdomains again, but now we only choose N of these subdomains. The N subdomains are arranged in a so-called *Latin-Hypercube*. Considering a two-dimensional case, each grid row and column can only contain one sample. See Figure 3 for a Latin-Hypercube sampling example with 9 samples. Some techniques improve the choice of the *Latin-Hypercube*. For example, maximizing the minimal distance between each pair of samples leads to a good representation of the real variability. We use *Latin-Hypercube* sampling as the default method in our framework.

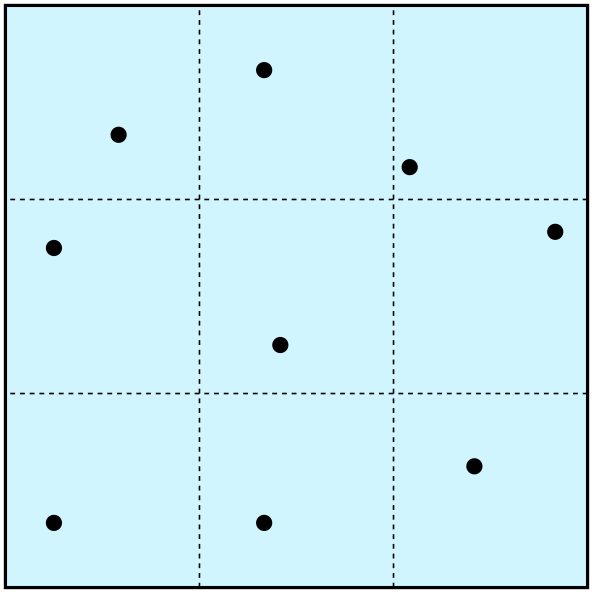


Figure 3: Full-factorial sampling in $\mathcal{X} \subset \mathbb{R}^2$ ($N = 3^2 = 9$).

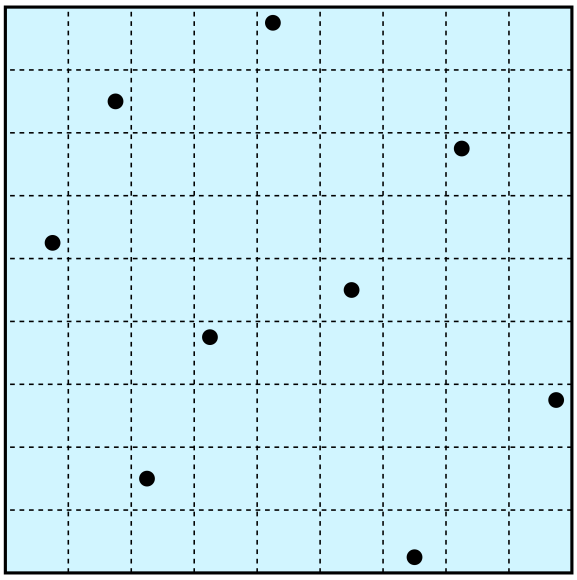


Figure 4: Latin-Hypercube sampling in $\mathcal{X} \subset \mathbb{R}^2$ ($N = 9$).

3 Implementation

This chapter concerns itself with the actual implementation of our framework. It covers the ideas and applications of the *Separability* concepts introduced in Section 2 as well as some technical details of the implementation.
The framework is written entirely in *Python 3*. We pondered about whether to use `C++` or Python for the implementation. From a development perspective, Python is easier to handle and has more numerical libraries which we can utilize. `C++` is strongly typed, which would have made it quite complicated for some implementations, especially the subdomain regression tree. We settled for a `C++` support via *pybind11* (pyb [2022]) if the user wants to analyze `C++` models.
Figure 5 presents the general class structure. The framework is split into three main parts: the sampling, the *Quasi-Separability* detection, and the subdomain splitting. We will introduce each in the next sections.

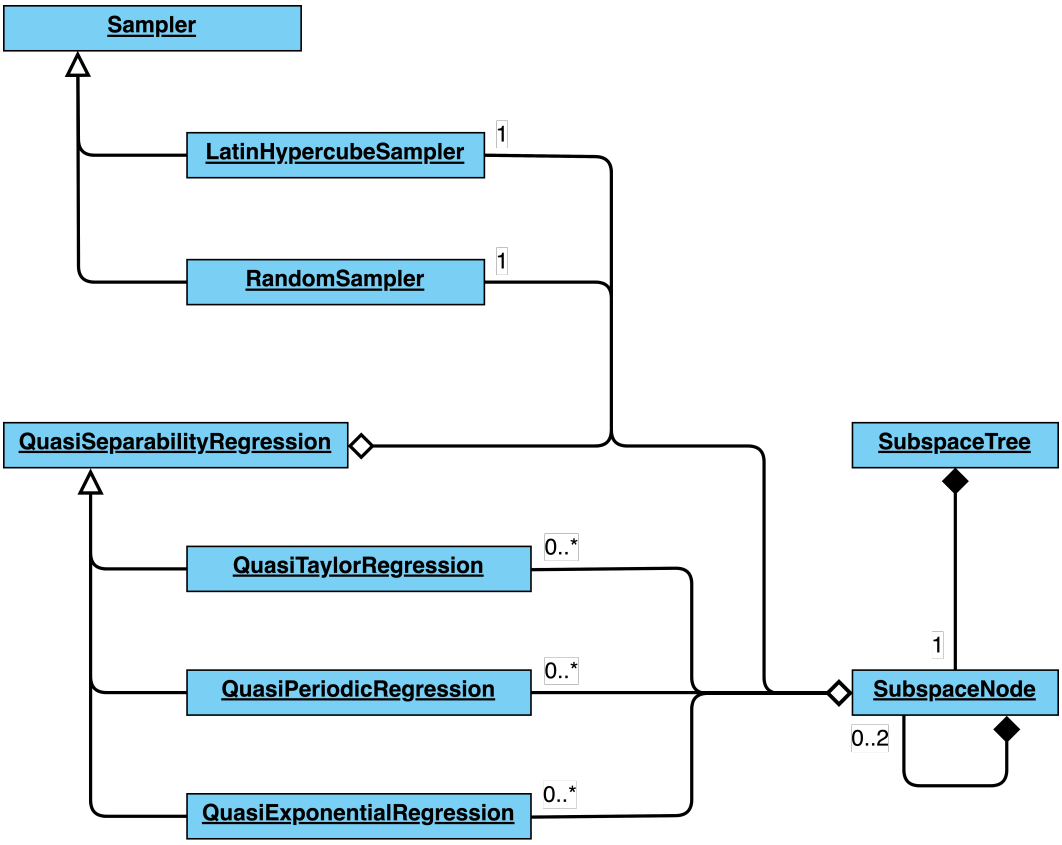


Figure 5: Class diagram of the *RSQTOA* framework.

3.1 Sampling

The sampling framework implements random sampling and *Latin-Hypercube* sampling as presented in Section 2.6. As shown in Fig. 6, the base class provides the subspace sampling as well a way to evaluate the samples. The subclasses implement the high-dimensional sampling techniques. We use the *pyDOE* (pyD [2022]) module as a backend for the *Latin-Hypercube* sampling. See Code 6 for the source code.

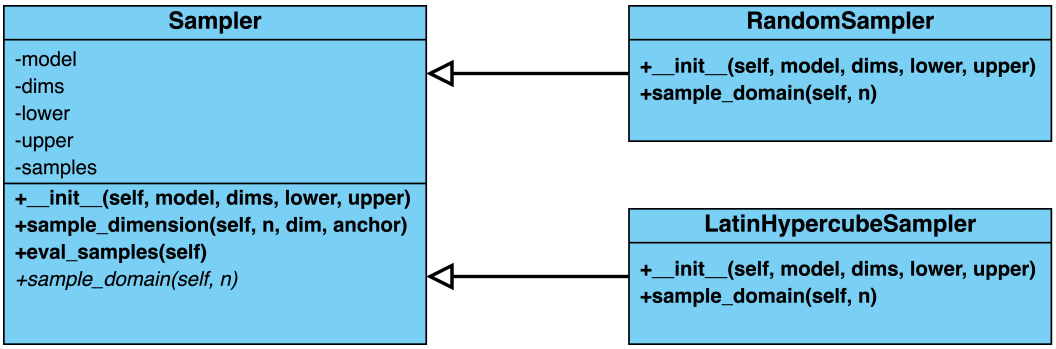


Figure 6: Class diagram of the sampling subpackage.

3.2 Model Reduction via Separability Analysis

The *Quasi-Separability* detection is the core of our framework. Figure 7 provides an overview of the regression classes. The base class `QuasiSeparabilityRegression` provides an implementation for the detection strategy, which was introduced in Section 2.3. The subclasses implement the actual regression model, namely the Taylor model, the exponential model and the periodic model. See Code 7, Code 8, Code 9 and Code 10 for the source code.

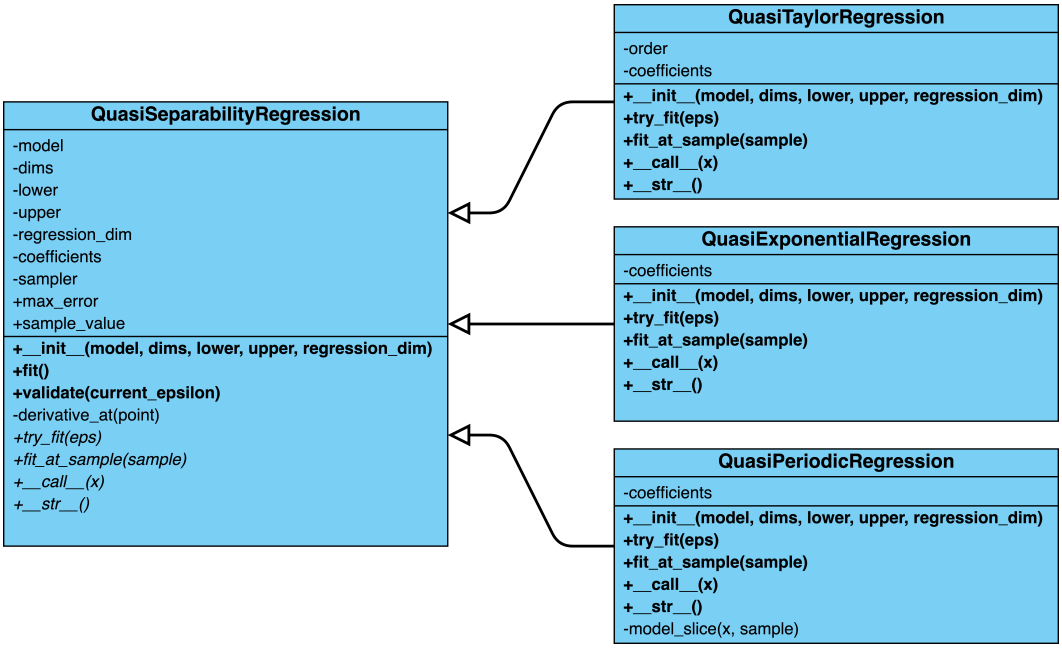


Figure 7: Class diagram of the *Quasi-Separability* regression analysis subpackage.

Detection Algorithm Code 1 presents the implementation of the *Quasi-Separability* detection algorithm. The general algorithm steps are the following:

- **Sampling**

  A sampling of the current domain. These samples provide the anchors in the domain for the scalar regressions (line 5).

- **Regression Analysis**

  For each anchor, we perform the scalar regression analysis. The coefficients of the regression models are averaged, resulting in one regression model for the current domain (lines 10-12).

- **Sampling**

  We sample the current domain again as well as the regression dimension. These sample-value pairs are our test candidates (lines 25-39).

- **Validation**

  For each of those sample-value pairs, we calculate the accuracy with Equation (35) (line 46). In the end we calculate the largest error and store it.

```
1  def fit(self):
     """ Tries to fit the regression model over a subspace of the domain """

     # Get random samples in the entire domain
     X = self.sampler.sample_domain(config.regression_samples)
6
     # Perform subspace regression at samples and take average of coefficients
     try:
       self.coefficients = np.zeros_like(self.coefficients)
       for X_sample in X:
11       self.coefficients += self.fit_at_sample(X_sample)
       self.coefficients /= config.regression_samples
     except KeyboardInterrupt:
       sys.exit()
     except:
16     self.max_error = np.Inf
       return

     # Quick return in case of infinite or NaN coefficeints
     if any(np.isnan(self.coefficients)) or any(np.isinf(self.coefficients)):
21     self.max_error = np.Inf
       return

     # Get random test points in the entire domain (N=config.test_samples)
     X_test = self.sampler.sample_domain(config.test_samples)
26   y_test = self.sampler.eval_samples()

     # Errors
     err_matrix = np.zeros((config.test_samples, config.test_values))

31   for i in range(config.test_samples):
       # Evaluate regression model with x_i from the test point
       y_test_reg = self(X_test[i][self.regression_dim])

       # For each test point, sample a certain number of points in dimension x_i
36     # with the test point as anchor
       X_tilde = self.sampler.sample_dimension(
         config.test_values, self.regression_dim, X_test[i])
       y_tilde = self.sampler.eval_samples()

41     for j in range(config.test_values):
         # Evaluate regression model with the additionally sampled x_i
         y_tilde_reg = self(X_tilde[j][self.regression_dim])

         # Calculate error (see Quasi-Separability definition)
46       err_matrix[i,j] = abs(y_test_reg + y_tilde[j] - y_tilde_reg - y_test[i])

     # Caluclate maximal error and choose a sample value for the model reduction
     self.max_error = err_matrix.max()
     self.sample_value = (
51     X_tilde[err_matrix.max(0).argmin()][self.regression_dim])
```

Code Listing 1: Separability detection.

Regression Algorithm Each regression model implements the `fit_at_sample` function which takes a sample from the current domain, the anchor. From that anchor point it performs scalar regression into a single dimension. Code 2 presents the Taylor regression model. Here we can see that the taylor model uses a normal equation to solve the least-squares problem with *Numpy* (num [2022]) as the solver. The exponential and periodical regression models are using *Scipy* (sci [2022]) to solve their non-linear least-squares problems. *Scipy* uses the *Levenberg-Marquart* algorithm in its curve-fitting toolbox which we already mentioned in Section 2.1.2.

```
def fit_at_sample(self, sample):
  """ Performs the regression over the subspace defined by the sample """

  # Number of samples is at least as large as the number of coefficents
  n_coefficients = self.order + 1
  n_samples = n_coefficients + config.taylor_regression_samples
  sample = np.full(1, sample) if self.dims == 1 else sample

  # Sample random points in the subspace
  X = self.sampler.sample_dimension(n_samples, self.regression_dim, sample)
  y = self.sampler.eval_samples()

  # Compute regression matrix X (linear system)
  X_reg = np.empty((n_samples, n_coefficients))
  for i in range(n_samples):
    X_i = np.full(1, X[i]) if self.dims == 1 else X[i]
    for j in range(n_coefficients):
      X_reg[i][j] = X_i[self.regression_dim] ** j

  # Perform regression via normal equation
  coeffs = np.linalg.solve(
    np.matmul(X_reg.transpose(), X_reg),
    np.matmul(X_reg.transpose(), y))

  # Return all coefficients apart from the constant term
  return coeffs[1:]
```

Code Listing 2: Taylor regression model.

3.3 Subdomain Splitting

As discussed in Section 2.5 we cannot rely on *Quasi-Separability* alone. Complex models will rarely contain a quasi-separable dimension on their entire domain, at least not within reasonable accuracy. One could limit the domain manually by considering a smaller input space for the variables, but that requires deep knowledge about the model itself. Part of this thesis was the implementation of an automatic approach that splits the domain into smaller subdomains and searches for subspaces where *Quasi-Separability* is achievable. The code for the algorithm presented in this section is listed in Code 11.

Figure 8 presents the class structure of this subpackage. We chose to use a tree structure to traverse the domain. This approach is similar to a *Kd-Tree*, but we are not using it in, for example, nearest neighbor searches like Ram and Sinha [2019]. We use it to store the subspace regressions, allowing us to easily generate the reduced models at each stage of the domain decomposition. It also helps us to reassemble it later.

The `SubsapceTree` class acts as the root. It holds only one `SubspaceNode` which is the node for the entire domain. All other nodes are stored recursively inside each other, representing the recursive domain decomposition. We will now present the general algorithm used to decompose the domain. Many parameters control the algorithm which are listed in Section 3.6.

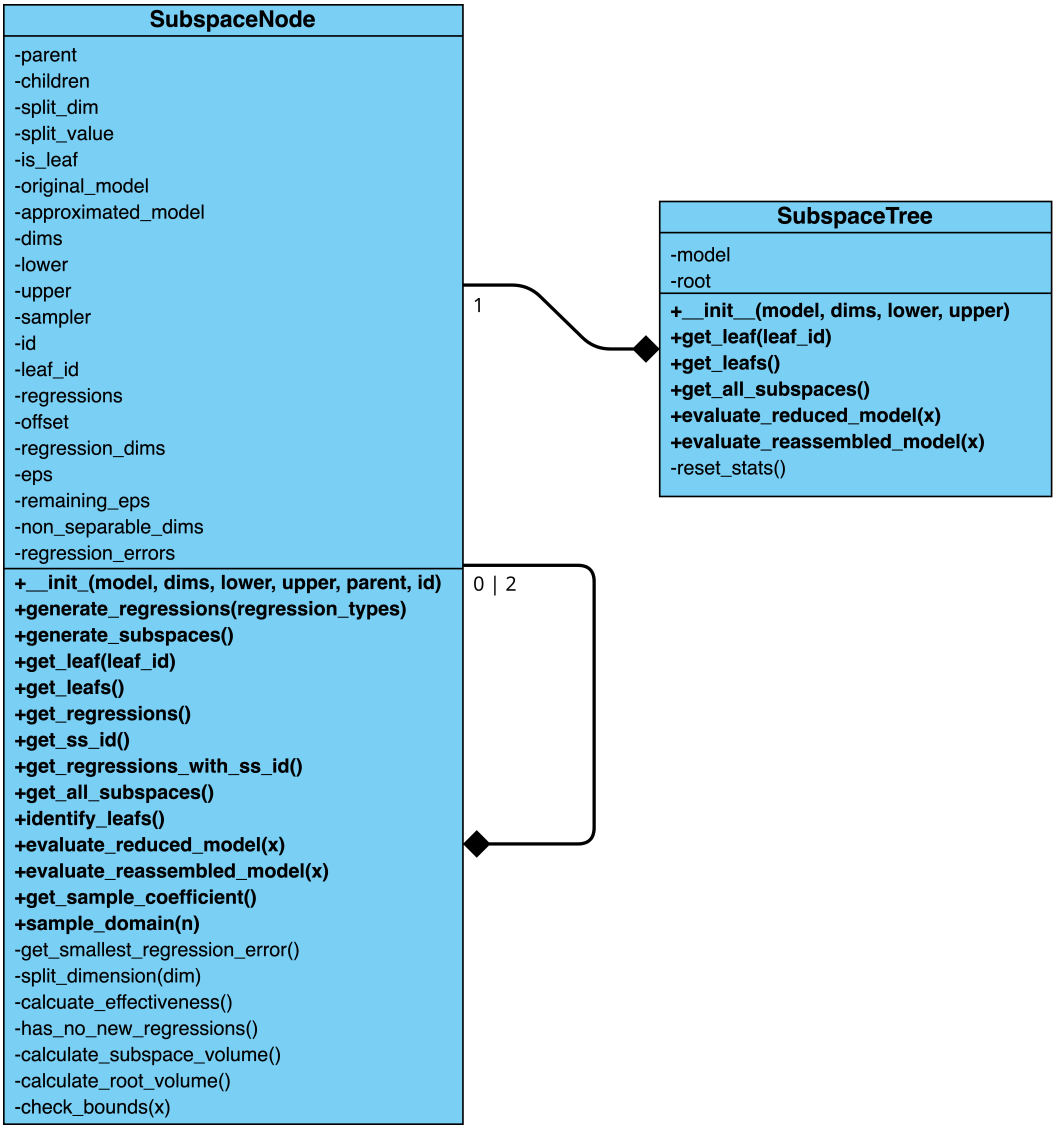


Figure 8: Class diagram of the subspace splitting subpackage.

Algorithm for the Domain Decomposition The algorithm has two main stages: splitting the domain and performing the *Quasi-Separability* analysis. The main steps for *Quasi-Separability* analysis with a particular regression model are already listed in Section 3.2. Here we have an abstraction layer above that. The steps are roughly the following:

- **Iterate over dimensions and regression models**

  We iterate over the dimensions that are not separated yet, and for each one, we perform the analysis with all regression models. Depending on the configuration, we might stop as soon as we find a regression that works. Otherwise, we perform all regressions and choose the best.

- **Regression combinatorics**

  If we find more than one quasi-separable dimension given our current error threshold, we need to ensure that their accuracies combined are within the threshold. We accomplish that by considering all possible combinations and choosing a valid combination that separates the most dimensions while minimizing the error.

- **Calculating offset**

  A function may be fully separable on a subdomain. That means the reduced model consists of only a constant term. We calculate that offset and store it in the subspace node.

We perform this part on the first node at the beginning. For all subsequent nodes, the domain splitting and the *Quasi-Separability* analysis are in a combined step. The combined algorithm:

- **Find a leaf node**

  The first step is to go through the tree depth-first and choose a leaf node with non-separated dimensions that we have not visited yet.

- **Sort dimensions**

  Depending on the configuration, we sort the remaining dimension according to the given criterion. We could, for example, split the dimension that had the lowest separability error. Another logical option is to split the largest dimension.

- **Perform domain decomposition in each dimension**

  When using a greedy algorithm, we stop the analysis when finding a subspace with a successful regression analysis. Otherwise, we are performing the next steps for each dimension:

  - **Bisect the dimension**

    First, we bisect the current dimension in the center, creating two new subdomains.

  - **Perform regression**

    In each new subdomain we perform the *Quasi-Separability* analysis as described above.

  - **Subdomain reduction**

    If the analysis yields no result in either subdomain, we try to bisect between the current split and the boundaries. Fig. 9 shows an example of a subdomain reduction. There is an option that controls how many times we bisect, but keep in mind that we need to perform the *Quasi-Separability* analysis again each time.

- **Subdomain expansion**

  If a viable subdomain decomposition is found we have a few options. If both subdomains contain newly separated dimension we stop immediately and add the new subdomain pair to the tree. If only one of the subdomains has a successful regression, we can expand the subspace via further bisection as depicted in Fig. 10. If one expansion invalidates the regression, we bisect in the other direction. The amount of expansions is also controllable.

- **Add new subspaces**

  If we found more than one subspace pair, we choose the best one. If there are no subdomains with successful regressions, we can allow adding the subdomain that contains the regression with the smallest error. Otherwise, the node is marked as a final leaf node, and we continue from step one.

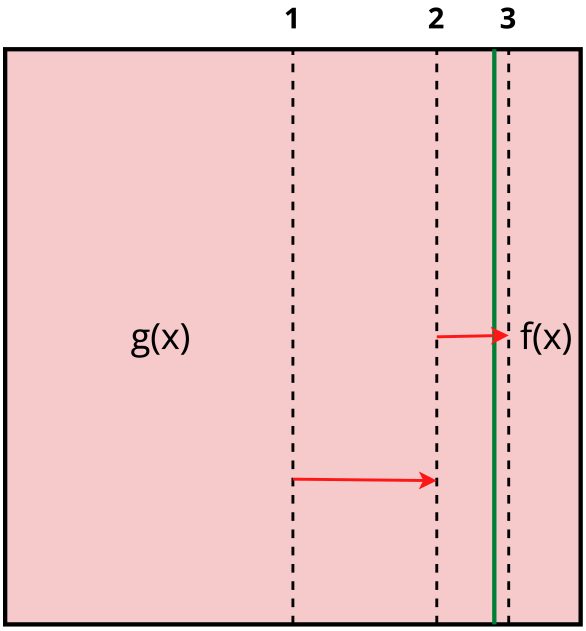


Figure 9: Domain **reduction** via further bisection. $f(x)$ is separable, $g(x)$ is not.

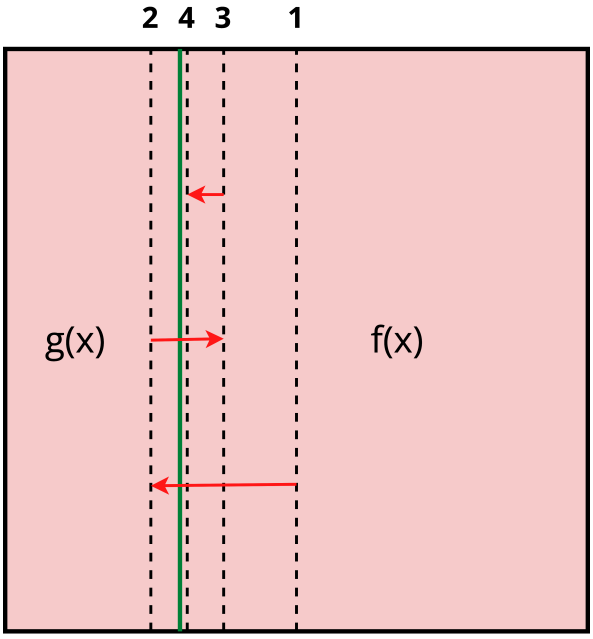


Figure 10: Domain **expansion** via further bisection. $f(x)$ is separable, $g(x)$ is not.

Effectiveness In the algorithm we occasionally need to assess whether one subdomain is better than another when both contain a successful regression. We create our own measure which we call the *Effectiveness* of a subdomain. The *Effectiveness* contain three main components.

- **The relative amount of separated dimension**

  Considering d_{before} is the amount of non-separated dimensions before the analysis and d_{after} the amount after the analysis completed. The value we are using is

$$d_{rel} = \frac{d_{before} - d_{after}}{d_{before}} \tag{38}$$

  and the smaller that quantity is the better.

- **The relative volume of the subdomain**

  We compare the volume of the subdomain to the volume of the entire domain:

$$V_{rel} = \frac{V_{subdomain}}{V_{root}}. \tag{39}$$

  Here we prefer large subdomains over smaller ones.

- **The relative accuracy**

  We start with a global error threshold ϵ. If a regression in a subdomain is successful that will reduce the error threshold in that subdomain for all coming recursive splits. We therefore prefer subdomains with regressions that do not introduce relatively large errors. Similar to the dimension factor, we define the accuracy factor as

$$\epsilon_{rel} = \frac{\epsilon_{before} - \epsilon_{after}}{\epsilon_{before}}. \tag{40}$$

We combine these factors into one equation:

$$\mathcal{E} = 2 + \lfloor \frac{d_{rel} * V_{rel}}{\epsilon_{rel}} \rfloor. \tag{41}$$

We offset the *Effectiveness* by two because we assign subdomains with no regression an *Effectiveness* $\mathcal{E} = 1$. That allows us to get a combined *Effectiveness* for both new subdomains by multiplying their scores. This will favor two mediocre subdomains over one very good combined with a bad one. Code 3 presents the actual implementation.

```
def calculate_effectiveness(self):
  """ Calculate effectiveness of the model reduction """

  # Relative amount of separable dimensions
  reduced_dims_factor = len(self.regression_dims)
  reduced_dims_factor -= len(self.non_separable_dims)
  reduced_dims_factor /= len(self.regression_dims)

  # Quick return if non of the dimension was separable
  if reduced_dims_factor == 0:
    return 1

  # Relative volume of the current subdomain compared to the whole domain
  area_factor = self.calculate_subspace_volume()
  area_factor /= self.calculate_root_volume()

  # Factor which considers the remaining error threshold compared to the
  # original threshold.
  reduced_eps_factor = (self.eps - self.remaining_eps) / self.eps
  if reduced_eps_factor == 0 or len(self.non_separable_dims) == 0:
    reduced_eps_factor = np.finfo(np.float32).eps

  # Calculate effectiveness of the model reduction in the current subdomain
  return 2 + int(area_factor * reduced_dims_factor / reduced_eps_factor)
```

Code Listing 3: Effectiveness measure of subdomain model reduction.

3.4 TensorFlow Integration

After the *Quasi-Separability* analysis is complete, the reduced models need to be approximated through other means. We decided to build upon the work of Christodoulou [2020] and use TensorFlow to learn the reduced models. The training of the objective function itself needs only one neural network. Since we created possibly many subspaces with different regressions and dimensions, we need to train the neural network for each subdomain.
The subspace regression tree provides an interface to access the subdomain nodes, which are so-called *leaves*. A leaf node contains a subdomain and a possibly reduced model that needs to be learned by TensorFlow. All dimensions in a specific subdomain may be separable, meaning that the reduced function is only a constant. In that case, we do not need a TensorFlow model for that particular subdomain.
We need to sample and evaluate each subdomain, which we achieve through the interface provided by the subspace node. With this data, we train a neural network. In the end, we can provide the subspace node with the learned model, and the subspace node allows us to evaluate the combined reassembled model.

3.5 API

This section gives a brief overview over the public API of our framework. We support Python and partly `C++`. Code 4 gives a short overview over the public API functions. The interface functions are fully listed in Code 14 and Code 15.

```python
def create_config(cfg=None, argv=None):
  """ Functional interface to load a new config """
  ...

def create_subspace_tree(yaml_config, model=None):
  """ Functional interface to create a subspace tree for a given model """
  ...

def generate_reduced_model(ss_tree, yaml_config):
  """ Functional interface to generate C++ code with the reduced models """
  ...
```

Code Listing 4: Public API functions.

3.5.1 Python

The Python API consists mainly of two functions: `create_config` and `create_subspace_tree`. The first creates a configuration object from a configuration file or command-line arguments. The configuration object can also be edited after it is created. The second API function takes the configuration object and optionally the objective function as a callable object and performs the *Quasi-Separability* analysis. The returned object is the subspace regression tree which can evaluate the reduced models.

3.5.2 C++

The API for the *Quasi-Separability* analysis is the same for C++. The user needs to create an interface for their C++ code with *pybind* (pyb [2022]) which is then used by our framework. Part of the C++ code is based on the framework built by Christodoulou [2020].

Code Generation Additionally, to the two API functions explained in Section 3.5.1 there is a third one for the C++ interface. The function `generate_reduced_model` takes the subspace regression tree and generates C++ code for all reduced models. The code generation is handled completely in Python with a simple search and replace that uses template files and replaces some defined keywords with generated code.

Test Framework There is a test framework in the repository which uses the C++ interface. The build process is handled via CMake, which builds the *pybind* interfaces, invokes the *Quasi-Separability* analysis, and tests if the framework behaves correctly.

3.6 Configuration Options

The framework offers various ways to configure regression analysis and domain decomposition. The user has the option to use a *YAML* configuration file, pass options via command-line arguments, or set options from within Python. As an example, Appendix A.1 shows a *YAML* file that configures the most important options. The next paragraphs explain the configuration options the file provides.

General options

- **output_dir**

  The output directory for generated files.

- **debug**

  Whether to print debug messages or not.

Model definition

- **model_function**

  The name of the objective function. This option is ignored if the model is passed to the framework directly.

- **dimensions**

  Number of dimensions of the objective function.

- **lower_bounds**

  Lower bounds of all dimensions.

- **upper_bounds**

  Upper bounds of all dimensions.

RSQTOA hyperparameter

- **epsilon**

  The overall error threshold. The framework will keep the error of the regressions below this threshold.

- **sampling_strategy**

  The sampling method, used for sampling the regression anchors as well as the validation samples. Possible values: 'random', 'latin', 'latin_maximin', 'latin_corr'.

- **regression_samples**

  Number of regression anchors in the domain. At each anchor the subspace regression is performed.

- **test_samples**

  Number of validation samples in the domain.

- **test_values**

  Number of values sampled in the scalar subspace. These are used together with the validation samples.

- **max_taylor_order**

  The maximum order we will consider for Taylor regression.

- **taylor_regression_samples**

  The number of samples that is used to over determine the least squares problem. If this number is zero, the number of samples is equal the number of regression parameters.

- **periodic_regression_samples**

  The number of samples used for periodic regression. If this number is zero, periodic regression is disabled.

- **exponential_regression_samples**

  The number of samples used for exponential regression. If this number is zero, periodic regression is disabled.

- **regression_selection**

  Possible values: 'greedy', 'best'. If the regression selection is greedy, we stop analyzing as soon we have found a regression model that is valid. Otherwise, we use all possible regression models and choose the best one.

Domain decomposition

- **subspace_depth**

  The maximum recursive domain splitting depth.

- **subspace_threshold**

  The minimum domain size. We do not split a dimension with a domain smaller than this size.

- **subspace_needs_regression**

  Whether at least one of the two subdomains needs a successful model reduction.

- **subspace_selection**

  Possible values: 'greedy', 'best'. With a greedy subspace selection, we will use the first domain split that yields model reduction. Otherwise, we analyze domain splits in all possible dimensions and choose the most effective one.

- **subspace_ordering**

  The ordering strategy for the remaining dimensions. Possible values: 'default', 'max_error', 'min_error', 'area'.

- **subspace_expansion_steps**

  If a subdomain with a successful model reduction is found, this option controls how many times we try to expand this subdomain.

- **subspace_reduction_steps**

  If a split did not result in a subdomain with a successful model reduction, this option controls how many times we reduce this subdomain.

4 Showcase

In this chapter we will demonstrate the superiority of our approach. We compare a function approximation via machine learning to the same approximation guided by our framework. Let us consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined as

$$f(\boldsymbol{x}) = \max(0, 10 * (x_1 - 0.3)^2 + x_0 * \sin(5 * x_0) + 2 \tag{42}$$

with a surface plot shown in Figure 11.
Analyzing the function, it is obvious that we can split the domain at $x_1 = 0.3$. On the two subdomains, the dimension x_1 is of Taylor order 0 and 2 respectively. Figure 12 shows the subspace tree that our framework came up with. With five subdomain expansions, we came close enough to the split value of $x_1 = 0.3$. Furthermore, both subdomains contain the correct regressions for dimension x_1. We plot the reduced model in Fig. 13. One can see that the plot is only dependent on x_0. Finally, the regression error is pictured in Fig. 14, which shows the largest errors at the subdomain boundaries.

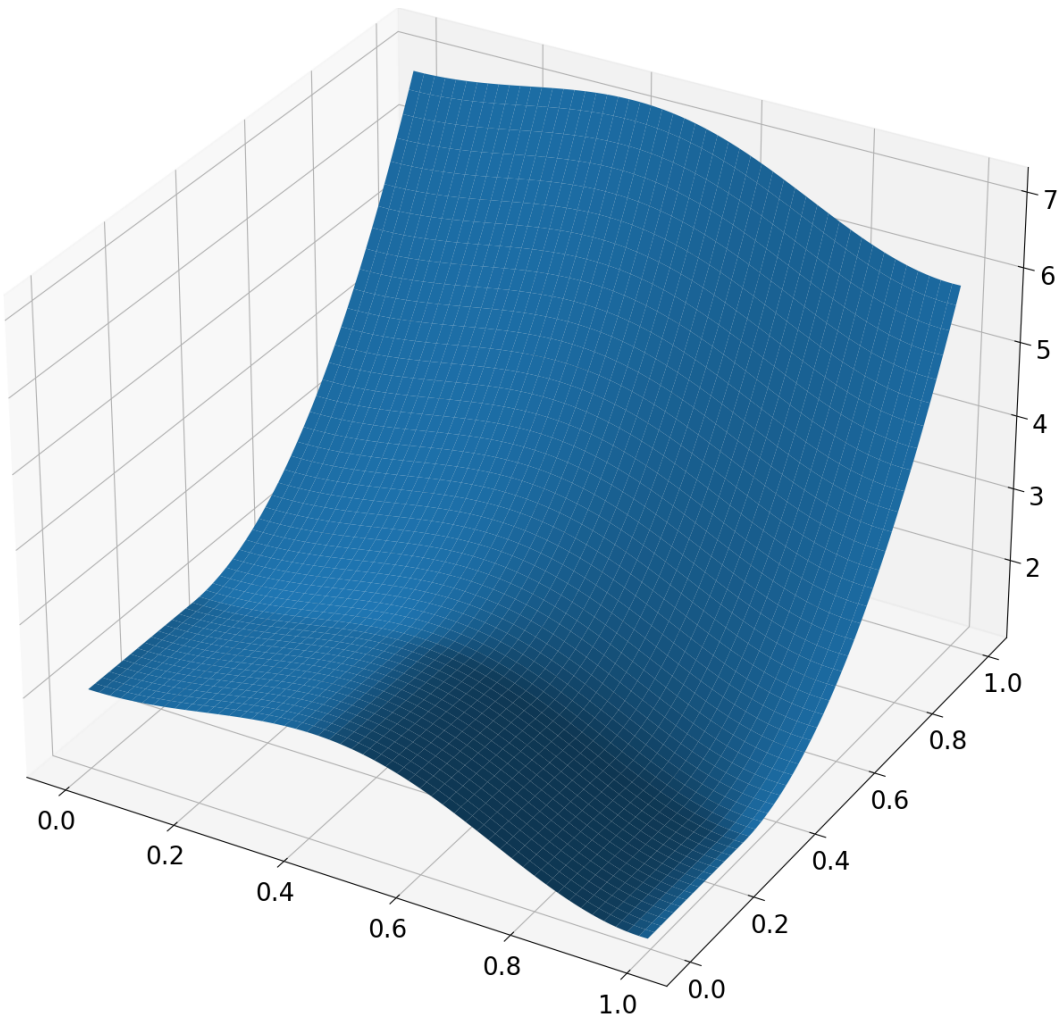


Figure 11: Surface of objective function Equation (42).

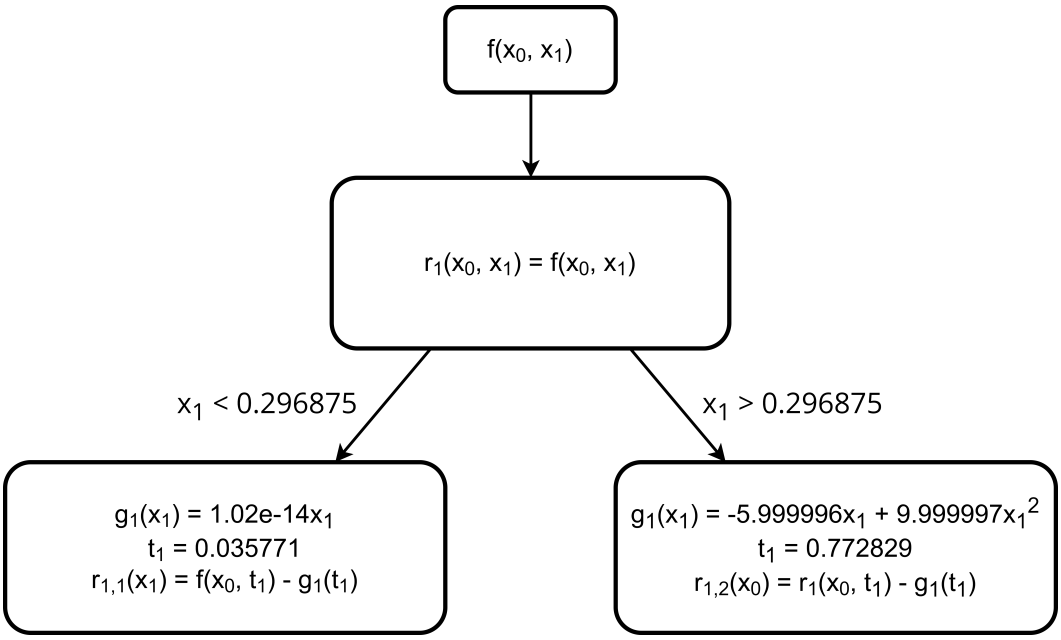


Figure 12: The subspace regression tree, calculated by our framework.

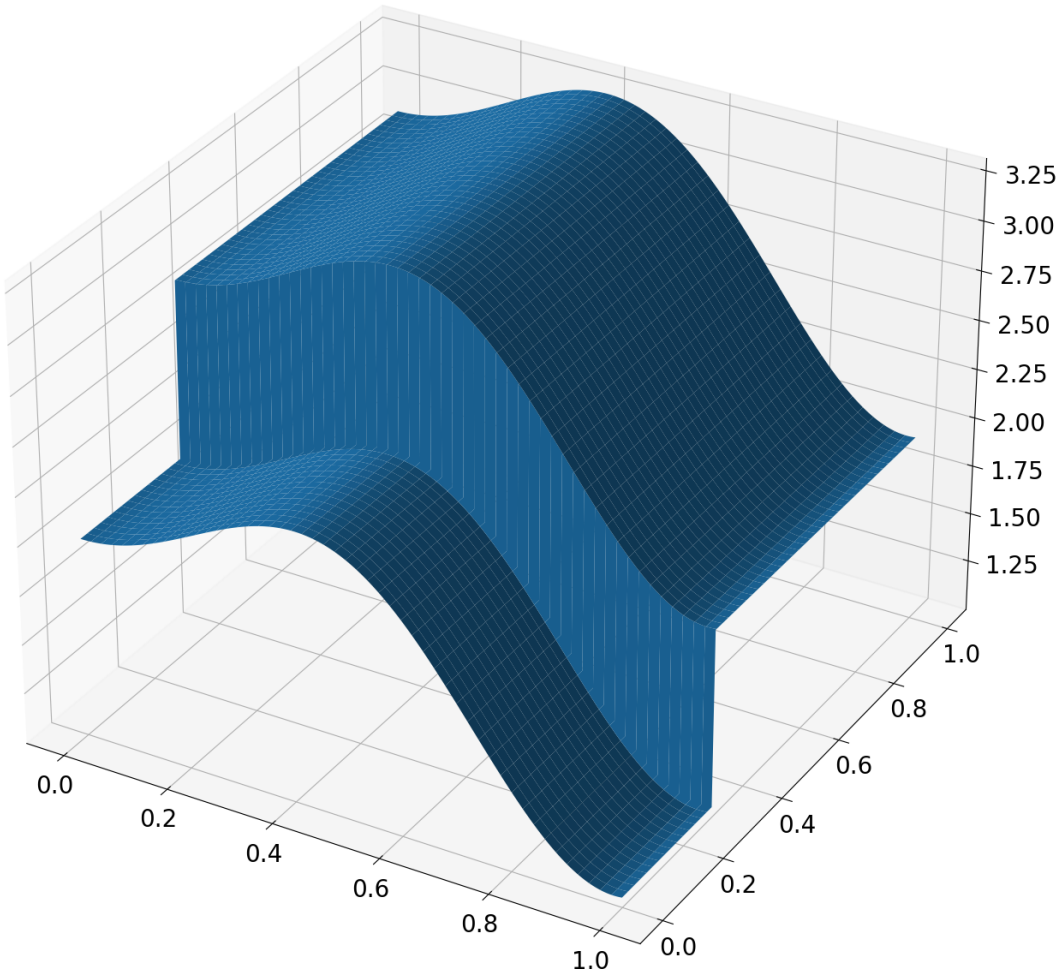


Figure 13: Reduced model of Equation (42) plotted as surface.

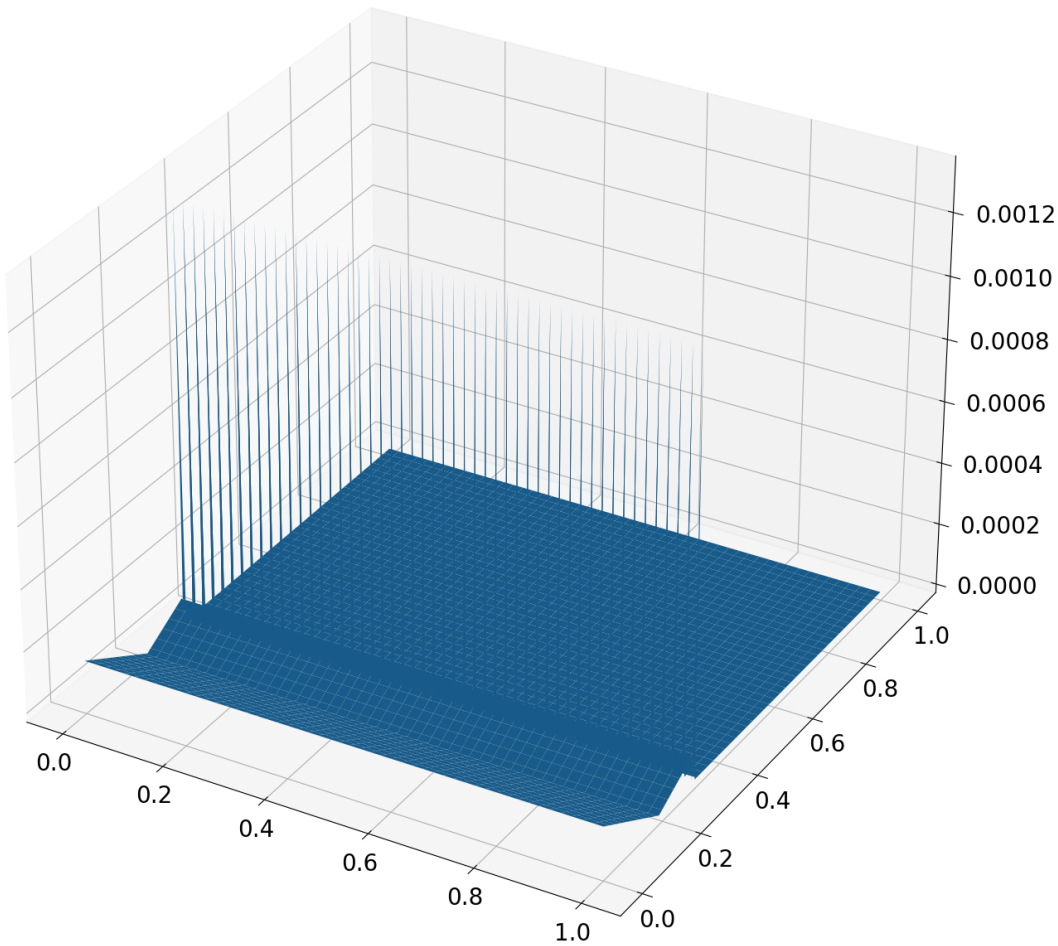


Figure 14: Regression errors of the reassembled function Equation (42).

Pure machine learning approach The neural network that we used to learn the objective function consists of five dense hidden layers with 100, 1000, 1000, 1000, and 10 nodes and exponential linear units as transition functions (Clevert et al. [2016]). With 100000 training and 20000 test samples, we trained the network 200 epochs with a batch size of 256. Figure 15 shows the loss history, as well as the final loss surface, plotted over the domain.

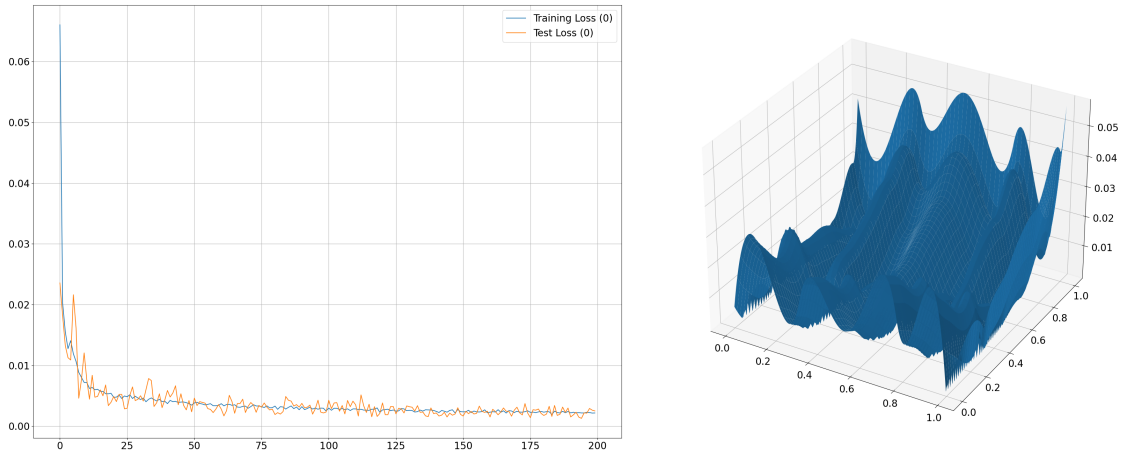


Figure 15: Residuals from the pure machine learning approach (100000 samples).

RSQTOA guided approach After our framework constructed the subspace regression tree, we used the same network to train both subdomains. The samples were equally split, i.e., each network is trained with 50000 samples plus 10000 test samples. The results are in Fig. 16. We see an improvement in accuracy by a factor of 4. The loss history shows that the neural networks are converging faster, and we can see that the loss surface plot is only dependent on x_0.

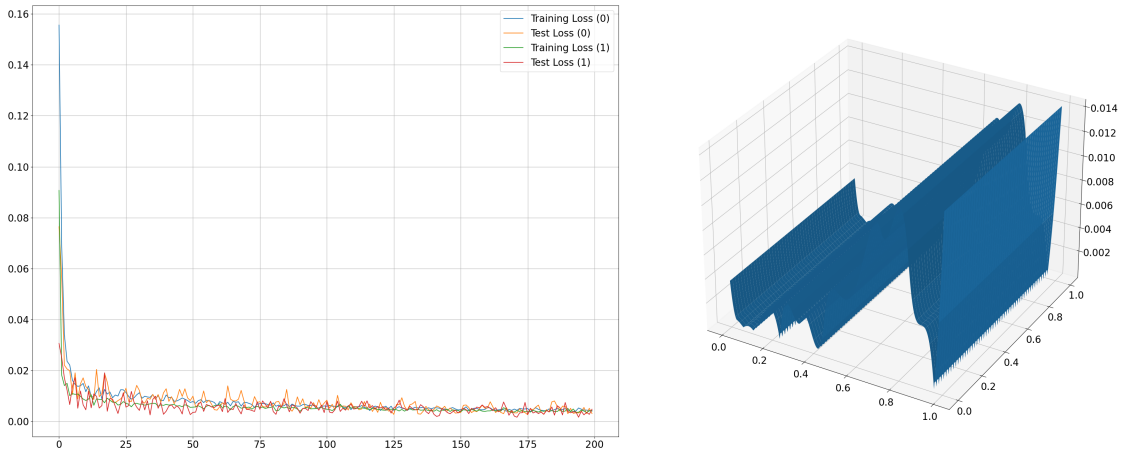


Figure 16: Residuals from the RSQTOA guided approach (100000 samples).

Smaller training sets We repeated the experiment with a smaller number of samples. With 10000 training samples and 2000 test samples, the results are shown in Figure 17 and Fig. 18. The improvements are worse, but we are still better by a factor of 2.5. Furthermore, the loss histories show a slightly slower convergence rate.

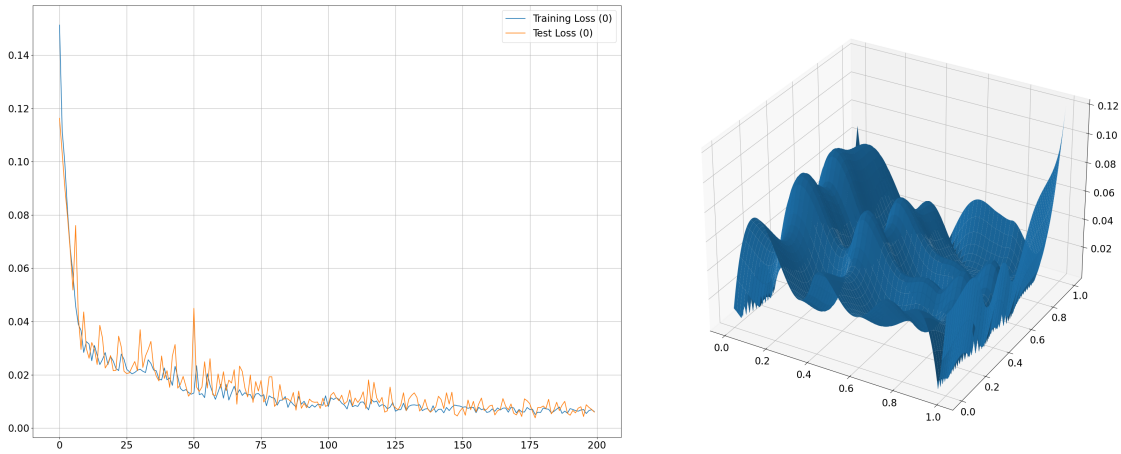


Figure 17: Residuals from the pure machine learning approach (10000 samples).

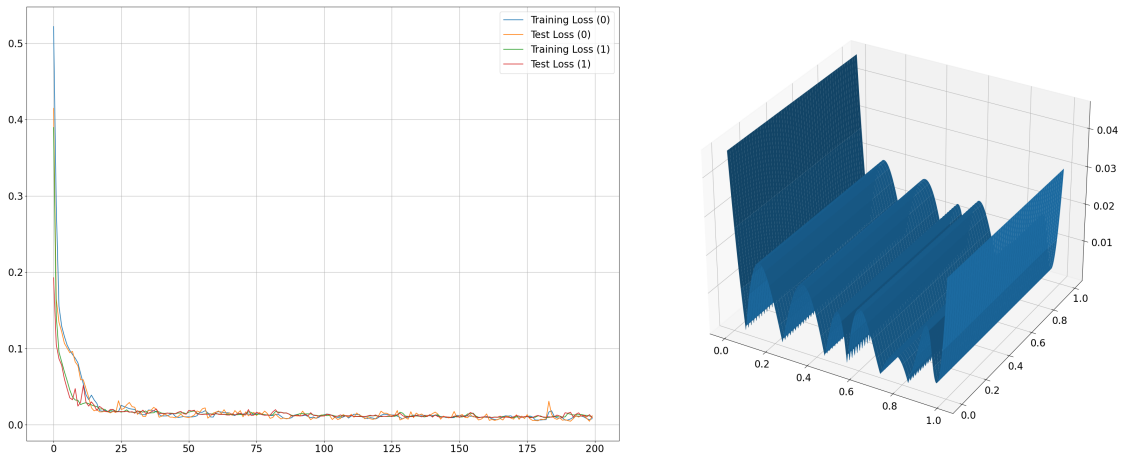


Figure 18: Residuals from the pure RSQTOA guided approach (10000 samples).

Complete regression analysis By increasing the amount of subdomains, one could theoretically divide the domain until the function is fully separable on all subdomains. We reached this case with ten subspaces. Figure 19 shows the regression errors after reassembling the model.

Discussion We showed that our framework could significantly improve accuracy even for a relatively simple function. However, accuracy is not necessarily what we want. In the context of computationally expensive functions, we want to reduce the number of samples needed. So instead of the additional accuracy, we can use fewer samples to get to the same accuracy as the pure machine learning approach. Of course, we need a certain amount of samples for the *Quasi-Linearity* analysis itself, exactly 4740 in this showcase. Fortunately, we can reuse those samples in the learning process.

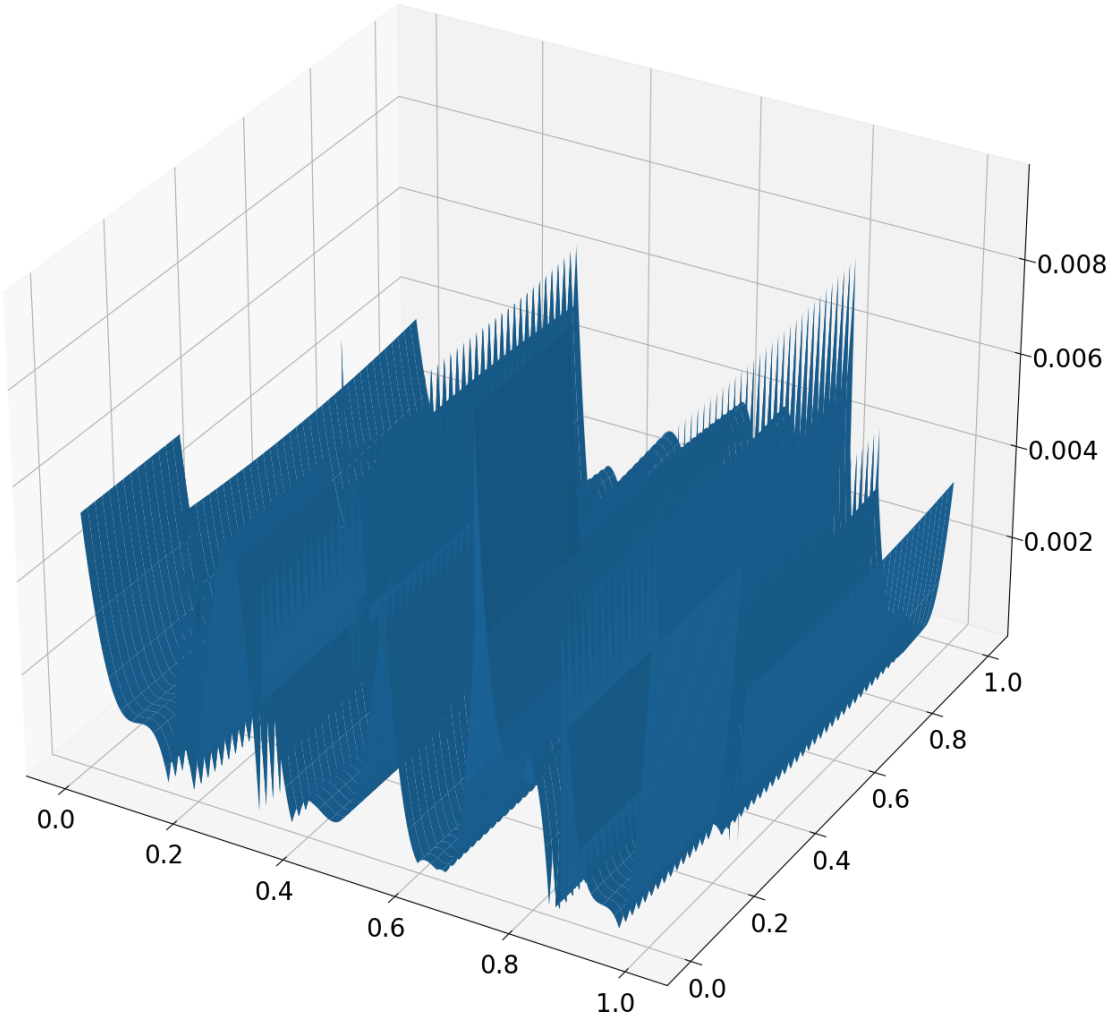


Figure 19: Full regression analysis with ten subdomains.

5 Case Study (*Black-Scholes*)

The model is a partial differential equation that models the dynamics of financial markets (Black and Scholes [1973]). The model is given by

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0. \tag{43}$$

From this model, a simpler formula for European-style options can be derived, given by

$$C\left(S_t, t\right) = N\left(d_1\right) S_t - N\left(d_2\right) K e^{-r(T-t)} \tag{44}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\left(T-t\right)\right)\right] \tag{45}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{46}$$

with $N(x)$ being the cumulative normal distribution. The parameters are the following:

$S(t)$ - price of the underlying asset at time t

K - strike price of the option

σ - standard deviation of the stock's return (a measure of volatility)

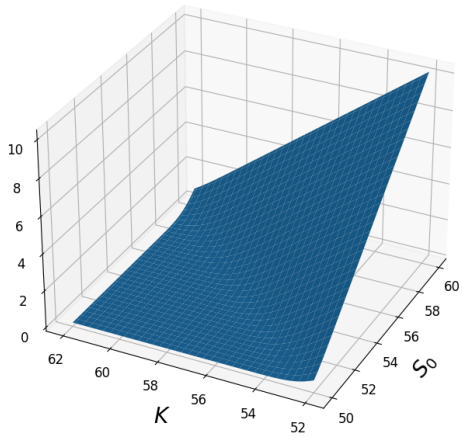
r - annualized risk-free interest rate

t - time in years

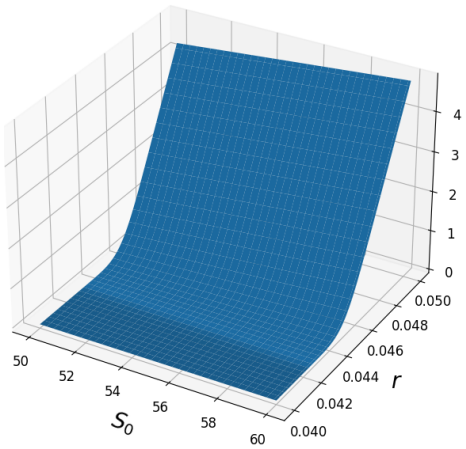
T - time of option expiration

The model we are analyzing is Equation (46) with a fixed expiration time $T = 1$ at time $t = 0$. The domain that we are considering is given by $C : [50, 60] \times [52, 62] \times [0.04, 0.05] \times [0.01, 0.02] \to \mathbb{R}$ with $(S_0, K, r, \sigma) \mapsto C(S_0, K, r, \sigma)$.
Since the model has four dimensions, it is not easy to visualize it. However, we plotted all 2-D surfaces at a given anchor point in Fig. 20. Although, in general, we can not use these surface plots to deduce *Quasi-Separability*, we can see that some plots show subdomains with different behaviors. Our program might find these subdomains.

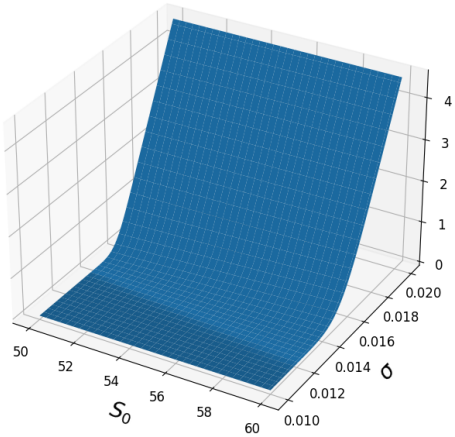
Pure machine learning approach The network we use for the training contains six dense hidden layers with 100, 1000, 1000, 1000, 1000, 10 neurons. We used 10000 training samples and 2000 test samples with a batch size of 256. After 1000 epochs we stopped the training since the returns after that were diminishing. Figure 21 shows the loss history. To visualize the loss we chose a similar approach to the 2-D surface slices. However, instead of an anchor point we averaged to loss over two dimension at a time. See Fig. 22 for the 2-D loss surfaces.



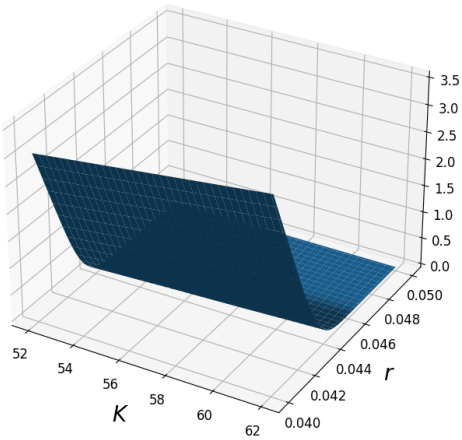
(a) $r = 0.045$ and $\sigma = 0.015$



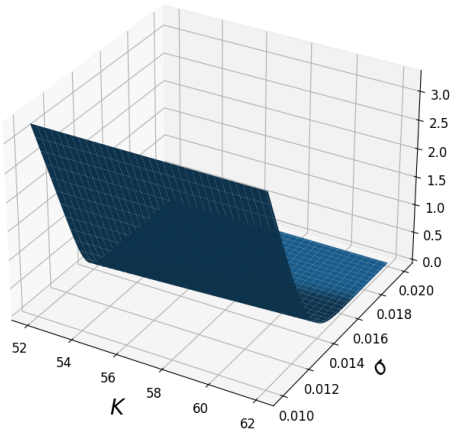
(b) $K = 58$ and $\sigma = 0.015$



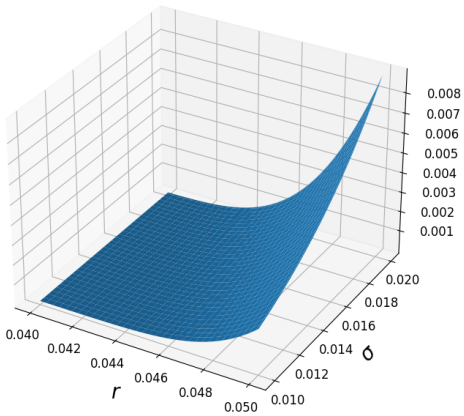
(c) $K = 58$ and $r = 0.045$



(d) $S_0 = 53$ and $\sigma = 0.015$



(e) $K = 58$ and $\sigma = 0.015$



(f) $S_0 = 53$ and $K = 58$

Figure 20: *Black-Scholes* 2-D surface slices at the anchor point $(53, 58, 0.045, 0.015)$.

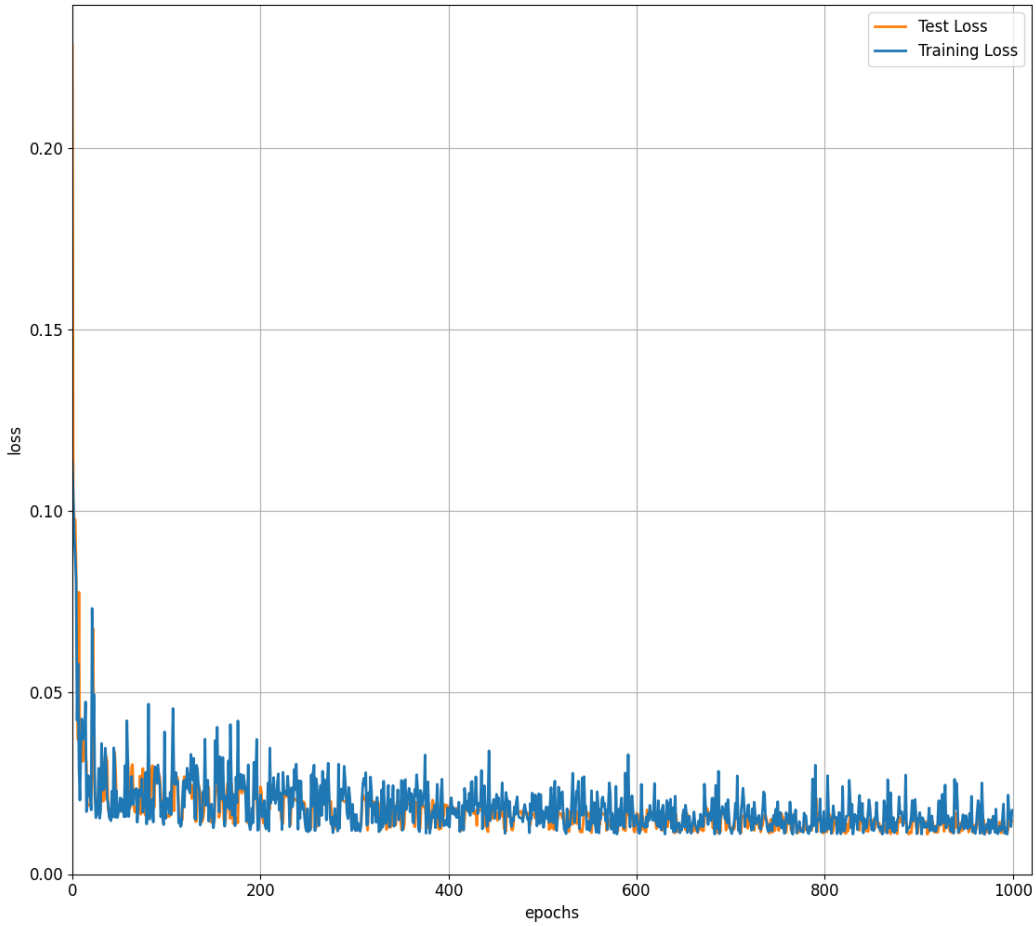
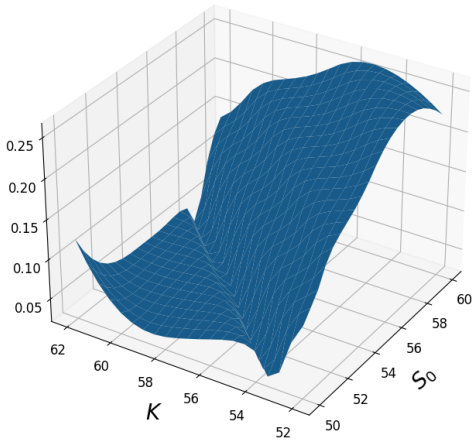
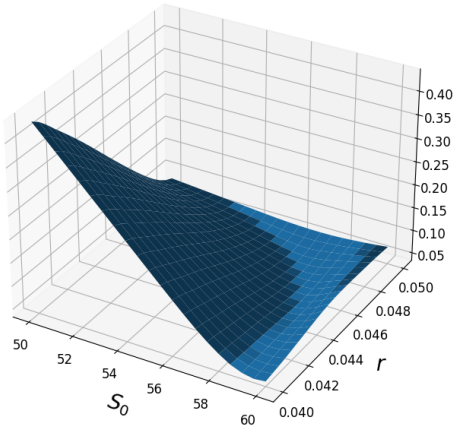


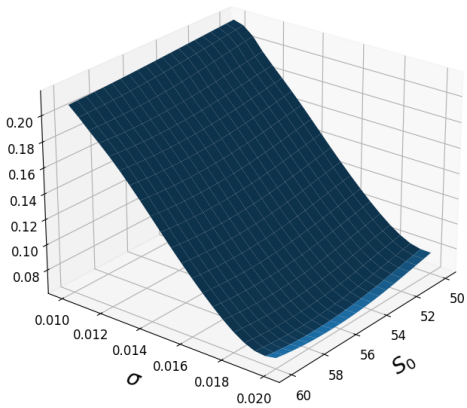
Figure 21: Loss (2-norm) history, learning the *Black-Scholes* model. Training set: 10000 samples. Test set: 2000 samples. Batch size: 256.



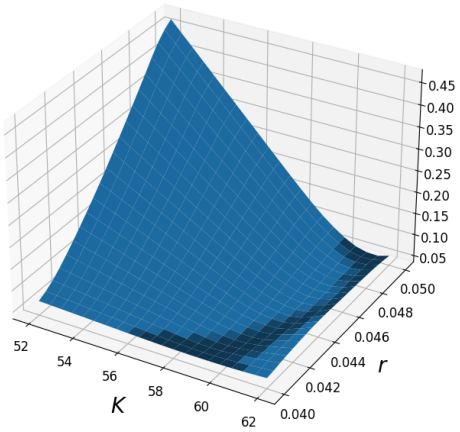
(a) loss averaged over S_0 and σ



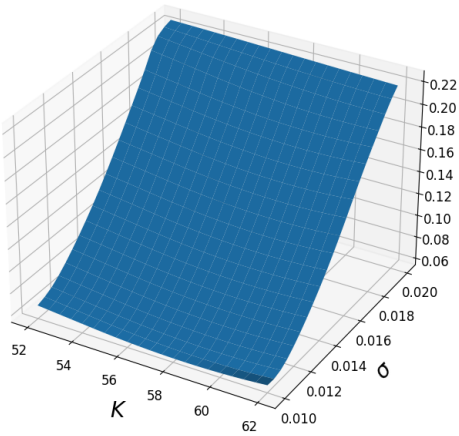
(b) loss averaged over K and σ



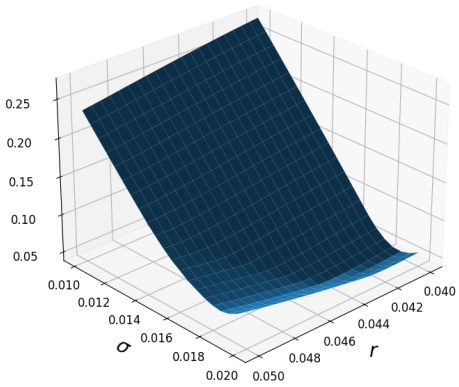
(c) loss averaged over K and r



(d) loss averaged over S_0 and σ



(e) loss averaged over S_0 and r



(f) loss averaged over S_0 and K

Figure 22: Loss (1-norm) surfaces, learning the *Black-Scholes* model. Training set: 10000 samples. Test set: 2000 samples. Batch size: 256.

RSQTOA guided approach Since the maximum error from the machine learning approach is around 0.45, our framework needs to use an error threshold below this value. We used an error threshold of 0.15 for the *Quasi-Separability* analysis.
Our framework found that dimension 3 (σ) is of taylor order 2 on the entire domain. The regression function is given by $g_3 : [0.01, 0.02] \to \mathbb{R}$ with

$$g_3(\sigma) = -0.605998 * \sigma + 139.701574 * \sigma^2. \tag{47}$$

For the first study we disabled the domain decomposition and used only the regression in Eq. (47). The results are plotted in Fig. 23 (loss history) and Fig. 24 (loss surfaces). The loss history shows a training which is a bit slower in the beginning but after the 1000 epochs we end up around the same accuracy using a Euclidean norm. However, we have some improvements in the loss surface plots which use a 1-norm. The maximum error is reduced from 0.451 to 0.291 when we compare to the machine learning approach.

RSQTOA guided approach (domain decomposition) Our framework found six subdomains, two of which are fully separable. The regression models for the quasi-separable dimensions are listed below with their respective subdomains.

- $g_0 : [50, 60] \times [52, 62] \times [0.04, 0.05] \times [0.01, 0.02] \to \mathbb{R}$ with

$$g_0(\sigma) = -2.09254 * \sigma + 227.87801 * \sigma^2 - 1633.27454 * \sigma^3 \tag{48}$$

- $g_1 : [50, 60] \times [52, 62] \times [0.04, 0.040625] \times [0.01, 0.02] \to \mathbb{R}$ with

$$g_1(r) = 28.13073 * r \tag{49}$$

- $g_2 : [58.828125, 60] \times [52, 58.250] \times [0.04, 0.040625] \times [0.01, 0.02] \to \mathbb{R}$ with

$$g_2(S_0) = 0.99999 * S_0 \tag{50}$$

- $g_3 : [58.828125, 60] \times [52, 58.250] \times [0.04, 0.040625] \times [0.01, 0.02] \to \mathbb{R}$ with

$$g_3(K) = -0.960409 * K \tag{51}$$

- $g_4 : [50, 60] \times [52, 58.250] \times [0.040625, 0.041211] \times [0.01, 0.02] \to \mathbb{R}$ with

$$g_4(r) = 26.42151 * r \tag{52}$$

- $g_5 : [50, 60] \times [52, 58.250] \times [0.041211, 0.04176] \times [0.01, 0.02] \to \mathbb{R}$ with

$$g_5(r) = 30.44232 * r \tag{53}$$

The results for this study are plotted in Fig. 25 (loss history) and Fig. 26 (loss surfaces). We see the same maximum error as the previous study without domain decomposition. However, some loss surface plots show worse errors than the previous study. The loss history also shows slightly worse converged accuracies for some subdomains. The most likely reason is that we also divided the number of samples through domain decomposition. Those subdomains are trained with only 1666 samples each.

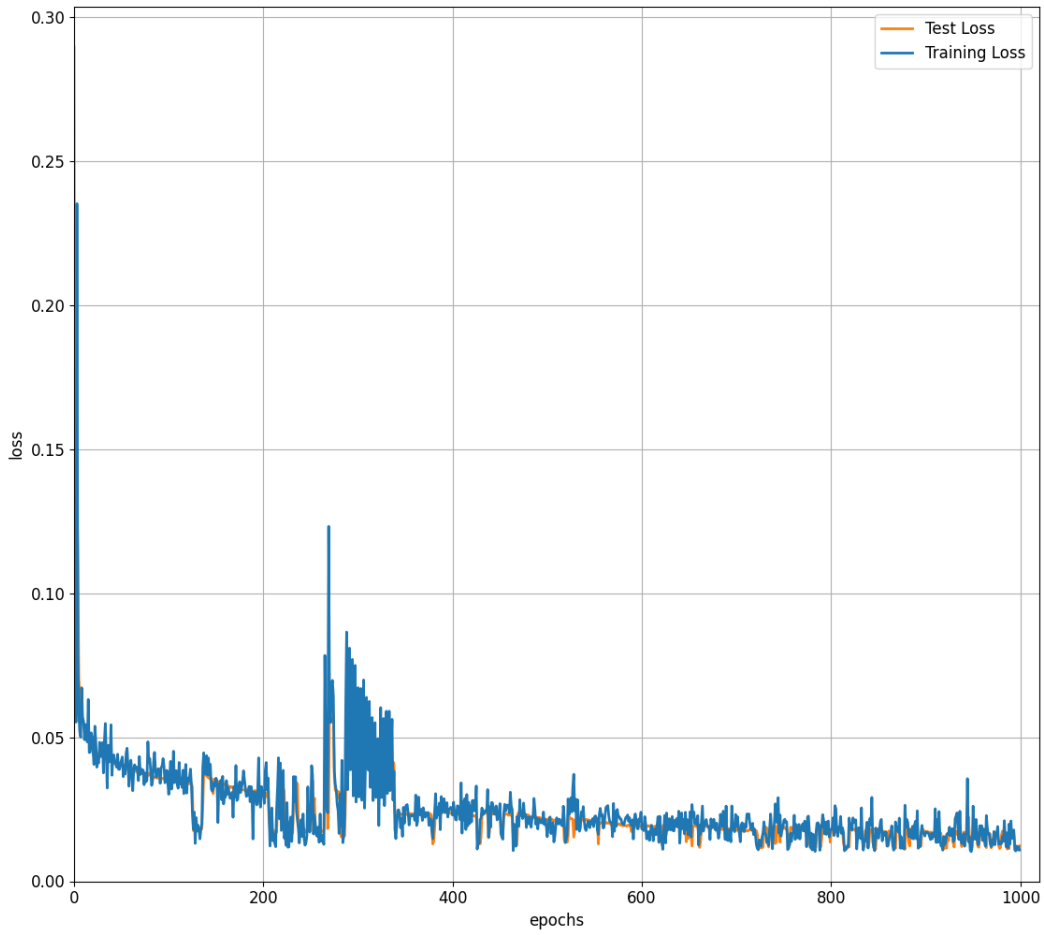
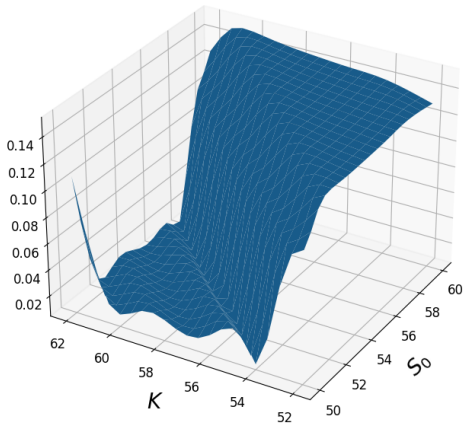
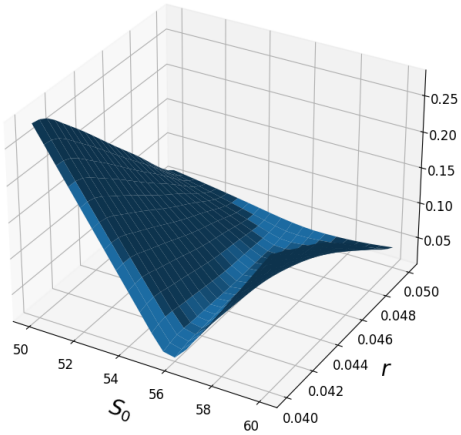


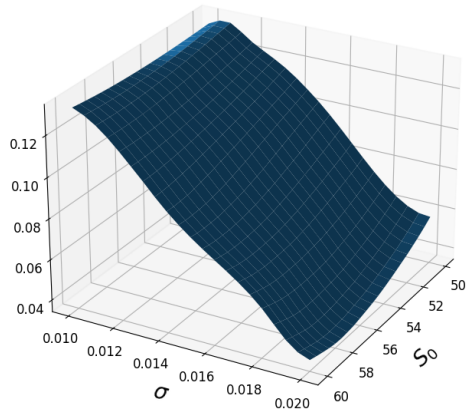
Figure 23: Loss (2-norm) history, learning the *Black-Scholes* model. Guided by RSQ-TOA **without** domain decomposition.



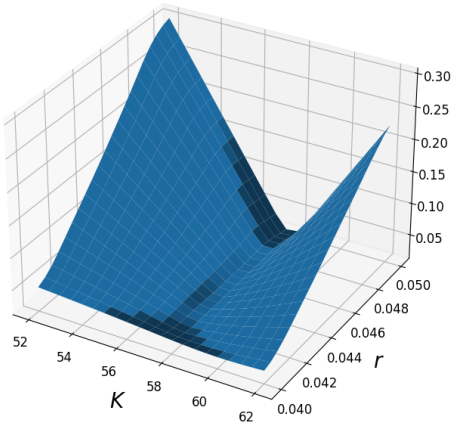
(a) loss averaged over S_0 and σ



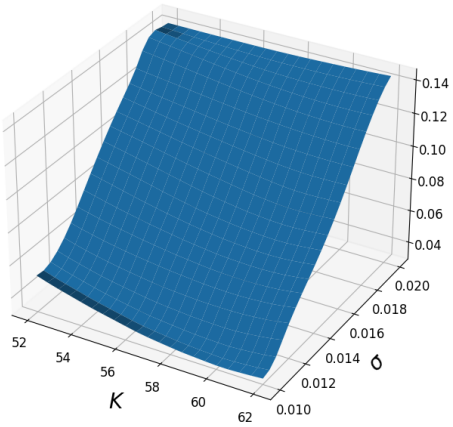
(b) loss averaged over K and σ



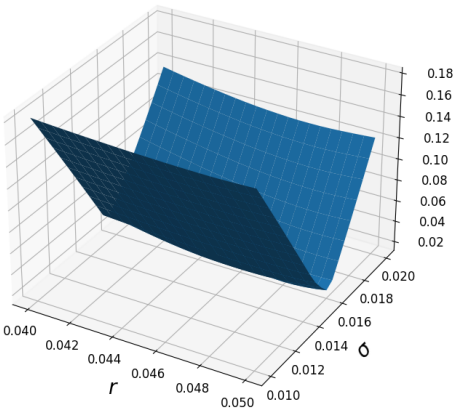
(c) loss averaged over K and r



(d) loss averaged over S_0 and σ



(e) loss averaged over S_0 and r



(f) loss averaged over S_0 and K

Figure 24: Loss (1-norm) surfaces, learning the *Black-Scholes* model. Guided by RSQ-TOA **without** domain decomposition.

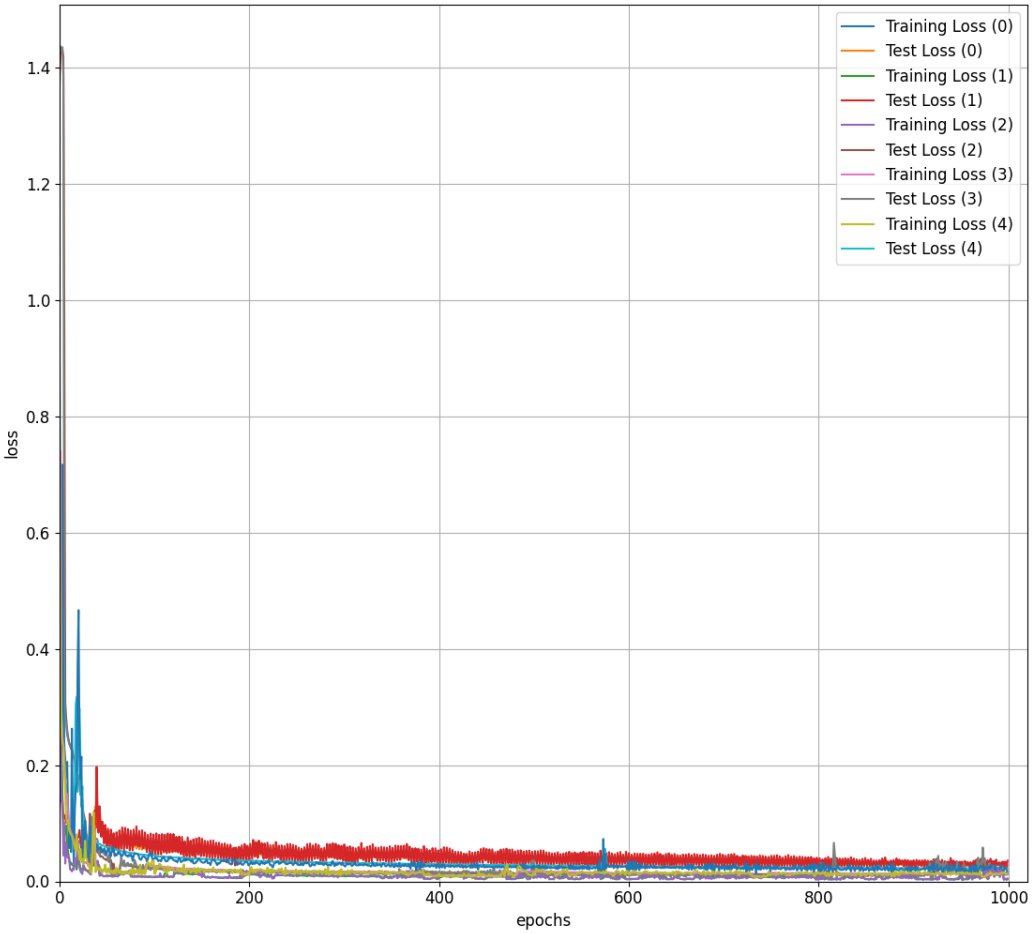
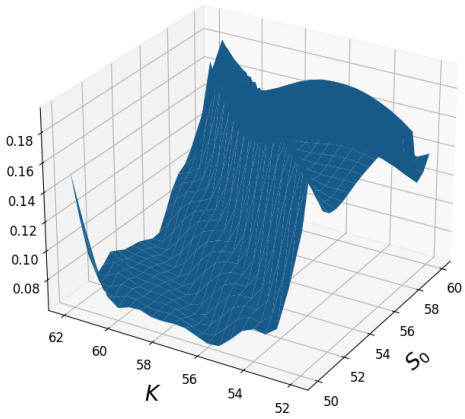
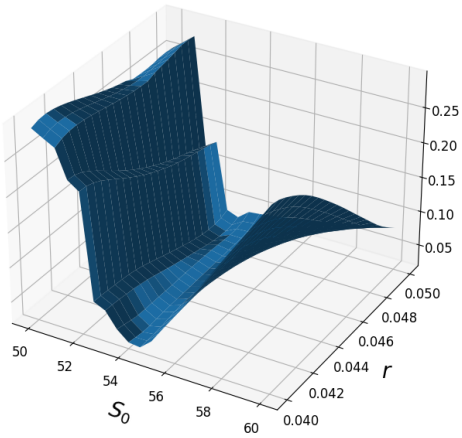


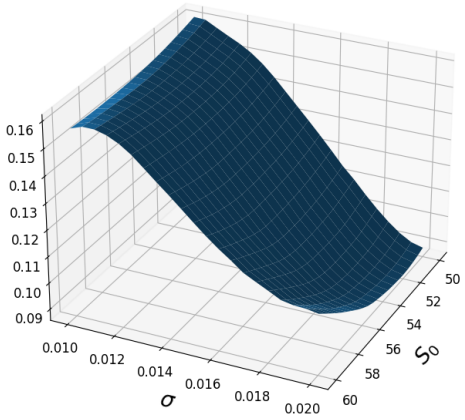
Figure 25: Loss (2-norm) history, learning the *Black-Scholes* model. RSQTOA **with** domain decomposition.



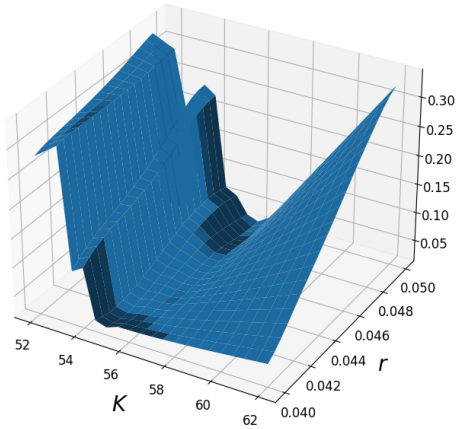
(a) loss averaged over S_0 and σ



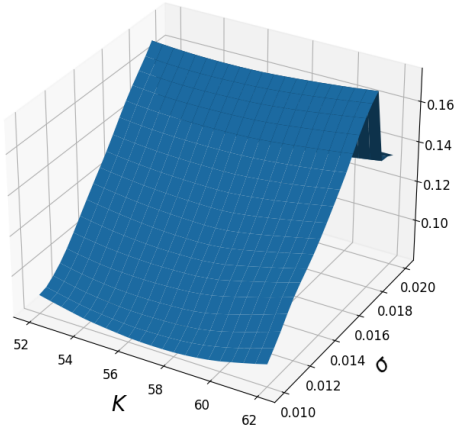
(b) loss averaged over K and σ



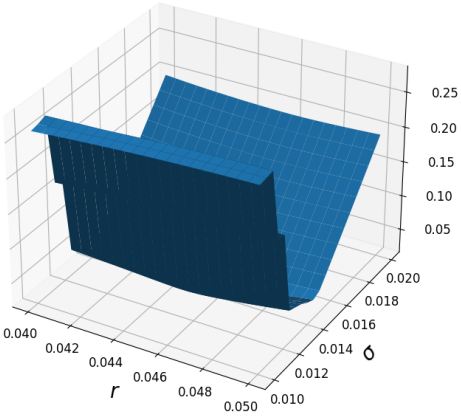
(c) loss averaged over K and r



(d) loss averaged over S_0 and σ



(e) loss averaged over S_0 and r



(f) loss averaged over S_0 and K

Figure 26: Loss (1-norm) surfaces, learning the *Black-Scholes* model. Guided by RSQ-TOA **with** domain decomposition.

6 Conclusion

6.1 Summary & Discussion

We introduced *Quasi-Separability* as a concept to reduce the dimensionality of a model. Our framework can successfully identify dimensions that are *quasi-taylor* of arbitrary order, quasi-periodic or *quasi-exponential*. Furthermore, we implemented an automatic domain decomposition that identifies *Quasi-Separability* on subdomains. In theory, our software can decompose the entire domain until the model is fully-separable everywhere. In practice, that is rather difficult since the *Curse of Dimensionality* will haunt us here. As a stand-alone method, our approach is not superior to machine learning or other surrogate model techniques (Wil [2008b]). The combination of our method with another surrogate model is the key. Our software is used as a preprocessing step that reduces the function, and the other method creates a surrogate model for that reduced function. As we have shown, the reassembled model of the combined approach can be more accurate than each method on its own.

We tested our framework on an artificial showcase and a real-world example. The showcase validated the combined approach's increased accuracy and showed that our software could indeed be used as a stand-alone tool. The real-world examples showed some improvements as well. However, the domain decomposition proved to be only somewhat helpful. Even though we found multiple subdomains, some even *fully-separable*, the result did not improve. The explanation is most likely the reduced number of samples that each subdomain can use. The machine learning we used as a backend did not perform well with the reduced number of samples. That means there is a trade-off between the number of subdomains and final accuracy. Additionally, the amount of samples we need for our analysis grows exponentially with the subdomain depth. The conclusion is that one should rather fit a more complex regression model, e.g., higher-order Taylor models, than decompose the domain.

Out of the three regression models that we considered, the Taylor model was by far the best. The non-linear regression models are rarely applicable while being more expensive and unstable than the linear model. One could instead fit linear models with non-linear ansatz functions.

The improvements in accuracy we showed could also be exploited to reduce the number of samples. This idea is exciting in the context of computationally expensive functions.

6.2 Further Research

At the moment, we are only considering surrogate models for functions that we can evaluate. Suppose we do not have the function but only a large set of samples. Our framework would not work since we need to sample in specific dimensions during the *Quasi-Separability* analysis. A possible approach is to first use some rudimentary interpolation technique on the data set and then feed the interpolation into our software. Otherwise, we need to refactor the *Quasi-Separability* analysis and find an algorithm that works on already sampled data.

As mentioned before, the non-linear models were not as successful as we hoped, at least on the model we examined. New regression models, like linear models with non-Taylor ansatz functions, are a possible topic for future implementations. We also only considered regression models in one dimension. We could think about fitting n-dimensional linear models like hyperplanes. This method would increase the complexity of the regression analysis, though, since we would need to consider all possible combinations of dimensions.

Another fact worth mentioning is that our framework's reassembled model is likely not smooth if we use domain decomposition. We could adapt smoothing techniques like the ones used in Chaudhuri et al. [1994].

Furthermore, we could consider using derivative information in the domain decomposition algorithm. It could be possible to utilize the derivatives to make better choices in the subdomain search.

V References

Sampling Plans, chapter 1, pages 1–31. John Wiley & Sons, Ltd, 2008a. ISBN 9780470770801. doi: https://doi.org/10.1002/9780470770801.ch1. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470770801.ch1.

Constructing a Surrogate, chapter 2, pages 33–76. John Wiley & Sons, Ltd, 2008b. ISBN 9780470770801. doi: https://doi.org/10.1002/9780470770801.ch2. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470770801.ch2.

03 2022. URL https://numpy.org/.

03 2022. URL https://github.com/clicumu/pyDOE2/.

03 2022. URL https://github.com/pybind/pybind11/.

03 2022. URL https://scipy.org/.

Jian An and Art Owen. Quasi-regression. *Journal of Complexity*, 17(4):588–607, 2001. ISSN 0885-064X. doi: https://doi.org/10.1006/jcom.2001.0588. URL https://www.sciencedirect.com/science/article/pii/S0885064X01905886.

Åke Björck. Solving linear least squares problems by gram-schmidt orthogonalization. *BIT Numerical Mathematics*, 7:1–21, 1967.

Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973. ISSN 00223808, 1537534X. URL http://www.jstor.org/stable/1831029.

Mohamed Amine Bouhlel, John T. Hwang, Nathalie Bartoli, Rémi Lafage, Joseph Morlier, and Joaquim R.R.A. Martins. A python surrogate modeling framework with derivatives. *Advances in Engineering Software*, 135:102662, 2019. ISSN 0965-9978. doi: https://doi.org/10.1016/j.advengsoft.2019.03.005. URL https://www.sciencedirect.com/science/article/pii/S0965997818309360.

Probal Chaudhuri, M. Huang, Wei-Yin Loh, and R. Yao. Piecewise-polynomial regression trees. *Statistica Sinica*, 4:143–167, 01 1994.

Sebastian F. Christodoulou. Approximartion of expensive functions through neural networks. Bachelor's thesis, RWTH Aachen University, 2020.

Charles Chui and Harvey Diamond. A natural formulation of quasi-interpolation by multivariate splines. *Proceedings of The American Mathematical Society - PROC AMER MATH SOC*, 99:643–643, 04 1987. doi: 10.1090/S0002-9939-1987-0877032-6.

Djork-ArnÃ© Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2016.

Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. 01 2008. ISBN 978-3-540-76492-2. doi: 10.1007/978-3-540-76493-9.

Jens Deussen. *Global derivatives*. Dissertation, RWTH Aachen University, Aachen, 2021. URL https://publications.rwth-aachen.de/record/837052. Veröffentlicht auf dem Publikationsserver der RWTH Aachen University 2022; Dissertation, RWTH Aachen University, 2021.

Henri P. Gavin. The levenberg-marquardt algorithm for nonlinear least squares curve-fitting problems. 2019.

Gero F. Kauerauf. Randomized subspace quasi-linearity analysis. Bachelor's thesis, RWTH Aachen University, 2021.

Hans-Georg Müller and Peng-Liang Zhao. Domain splitting in regression analysis. *Statistica Sinica*, 9(2):571–594, 1999. ISSN 10170405, 19968507. URL http://www.jstor.org/stable/24306599.

Art B. Owen. Assessing linearity in high dimensions. *The Annals of Statistics*, 28(1):1 – 19, 2000. doi: 10.1214/aos/1016120362. URL https://doi.org/10.1214/aos/1016120362.

Parikshit Ram and Kaushik Sinha. Revisiting kd-tree for nearest neighbor search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 2019, pages 1378–1388, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330875. URL https://doi.org/10.1145/3292500.3330875.

A Appendix

A.1 YAML Configuration File

```yaml
# RSQTOA configuration file

##############################################################
#                       General options                      #
##############################################################
output_dir: ~
debug: True

##############################################################
#                       Model definiton                      #
##############################################################
model_function: black_scholes
dimensions: 4
lower_bounds: [50, 52, 0.04, 0.01]
upper_bounds: [60, 62, 0.05, 0.02]

##############################################################
#                   RSQTOA hyper-parameter                   #
##############################################################
epsilon: 1e-1
sampling_strategy: "latin"
regression_samples: 10
test_samples: 25
test_values: 4
max_taylor_order: 3
taylor_regression_samples: 1
periodic_regression_samples: 10
exponential_regression_samples: 10
regression_selection: "greedy"

##############################################################
#                   Domain decomposition                     #
##############################################################
subspace_ordering: "default"
subspace_depth: 0
subspace_threshold: 1e-3
subspace_needs_regression: True
subspace_selection: "best"
subspace_expansion_steps: 3
subspace_reduction_steps: 3
```

Code Listing 5: YAML configuration file.

A.2 Source Code

```
import numpy as np
from pyDOE2 import lhs

# Import abstract parent class
from abc import ABC, abstractmethod

# Import global config and statistics
from rsqtoa import rsqtoa_config as config
from rsqtoa import rsqtoa_statistics as stats

# ------------------------------------------------------------------------- #

class Sampler(ABC):
  def __init__(self, model, dims, lower, upper):
    """ Constructor of the sampler base """
    self.model = model
    self.dims = dims
    self.lower = lower.astype(float)
    self.upper = upper.astype(float)
    self.samples = None

  def sample_dimension(self, n, dim, anchor):
    """ Sample in one dimension from an anchor in the domain"""
    X = np.empty((n, self.dims))

    # Broadcast anchor point and sample in the given dimension
    X[:] = np.copy(anchor)
    X[:, dim] = self.lower[dim]
    X[:, dim] += np.random.rand(n) * (self.upper[dim] - self.lower[dim])

    self.samples = np.copy(X)
    return X

  def eval_samples(self):
    """ Evaluate the current samples set """
    if self.model is not None and self.samples is not None:
      n = np.shape(self.samples)[0]
      Y = np.empty(n)
      for i in range(n):
        Y[i] = self.model(self.samples[i])
    else:
      raise RuntimeError("Either model or samples are not defined.")

    # Increase amount of samples
    stats.amount_of_samples += n

    return Y

  @abstractmethod
  def sample_domain(self, n):
    pass

# ------------------------------------------------------------------------- #

class RandomSampler(Sampler):
  def __init__(self, model, dims, lower, upper):
    """ Constructor of the random sampler """
    Sampler.__init__(self, model, dims, lower, upper)

  def sample_domain(self, n):
    """ Return n random samples. """
    X = np.random.rand(n, self.dims)
    X = self.lower + np.multiply(X, self.upper - self.lower)

    self.samples = np.copy(X)
    return X

# ------------------------------------------------------------------------- #

class LatinHypercubeSampler(Sampler):
  def __init__(self, model, dims, lower, upper):
    """ Constructor of the latin hypercube sampler """
    Sampler.__init__(self, model, dims, lower, upper)

  def sample_domain(self, n):
    """ Return n latin-hypercube samples. """
    if config.sampling_strategy == 'latin':
      criterion=None
    elif config.sampling_strategy == 'latin_maximin':
      criterion='maximin'
    elif config.sampling_strategy == 'latin_corr':
      criterion='correlation'
    else:
      raise ValueError(
        'Invalid sampling strategy: {}'.format(config.sampling_strategy))

    X = lhs(self.dims, samples=n, criterion=criterion)
    X = self.lower + np.multiply(X, self.upper - self.lower)

    self.samples = np.copy(X)
    return X
```

Code Listing 6: Sampling classes.

```python
import sys
import numpy as np

# Import abstract parent class
from abc import ABC, abstractmethod, abstractclassmethod

# Import global config
from rsqtoa import rsqtoa_config as config
from rsqtoa.rsqtoa_sampling import RandomSampler, LatinHypercubeSampler
from rsqtoa.rsqtoa_fd import _fd_derivative

# ---------------------------------------------------------------------------- #

class QuasiSeparabilityRegression(ABC):
  def __init__(self, model, dims, lower, upper, regression_dim):
    """ Constructor of the quasi-separability regression base class  """
    self.model = model
    self.dims = dims
    self.lower = lower.astype(float)
    self.upper = upper.astype(float)
    self.regression_dim = regression_dim
    self.coefficients = None
    self.max_error = 0.0
    self.sample_value = 0.0

    if config.sampling_strategy == 'random':
      self.sampler = RandomSampler(model, dims, lower, upper)
    else:
      self.sampler = LatinHypercubeSampler(model, dims, lower, upper)

  def derivative_at(self, point):
    """ Return the partial derivative at a given sample """
    return _fd_derivative(
      self.model, self.lower, self.upper, self.regression_dim, point)

  def fit(self):
    """ Tries to fit the regression model over a subspace of the domain """

    # Get random samples in the entire domain
    X = self.sampler.sample_domain(config.regression_samples)

    # Perform subspace regression at samples and take average of coefficients
    try:
      self.coefficients = np.zeros_like(self.coefficients)
      for X_sample in X:
        self.coefficients += self.fit_at_sample(X_sample)
      self.coefficients /= config.regression_samples
    except KeyboardInterrupt:
      sys.exit()
    except:
      self.max_error = np.Inf
      return

    # Quick return in case of infinite or NaN coefficeints
    if any(np.isnan(self.coefficients)) or any(np.isinf(self.coefficients)):
      self.max_error = np.Inf
      return

    # Get random test points in the entire domain (N=config.test_samples)
    X_test = self.sampler.sample_domain(config.test_samples)
    y_test = self.sampler.eval_samples()

    # Errors
    err_matrix = np.zeros((config.test_samples, config.test_values))

    for i in range(config.test_samples):
      # Evaluate regression model with x_i from the test point
      y_test_reg = self(X_test[i][self.regression_dim])

      # For each test point, sample a certain number of points in dimension x_i
      # with the test point as anchor
      X_tilde = self.sampler.sample_dimension(
        config.test_values, self.regression_dim, X_test[i])
      y_tilde = self.sampler.eval_samples()

      for j in range(config.test_values):
        # Evaluate regression model with the additionally sampled x_i
        y_tilde_reg = self(X_tilde[j][self.regression_dim])

        # Calculate error (see Quasi-Separability definition)
        err_matrix[i,j] = abs(y_test_reg + y_tilde[j] - y_tilde_reg - y_test[i])

    # Caluclate maximal error and choose a sample value for the model reduction
    self.max_error = err_matrix.max()
    self.sample_value = (
      X_tilde[err_matrix.max(0).argmin()][self.regression_dim])

  def validate(self, current_epsilon):
    """ Validated the maximal error against the given error threshold """
    success = self.max_error < current_epsilon
    if success:
      if config.debug:
        print(
          "success: max error (" + str(self.max_error) + ") < epsilon ("
          + str(current_epsilon) + ")")
    else:
```

```
      if config.debug:
        print(
          "fail: max error (" + str(self.max_error) + ") > epsilon ("
          + str(current_epsilon) + ")")

    return success

  @abstractclassmethod
  def try_fit(self, eps):
    pass

  @abstractclassmethod
  def fit_at_sample(self, sample):
    pass

  @abstractmethod
  def __call__(self, x):
    pass

  @abstractmethod
  def __str__(self):
    pass
```

Code Listing 7: Base class for *Quasi-Separability* detection with subspace regression.

```python
import numpy as np

# Import abstract parent class
from rsqtoa.rsqtoa_subspace_regression import QuasiSeparabilityRegression

# Import global config
from rsqtoa import rsqtoa_config as config

# ---------------------------------------------------------------------------- #

class QuasiTaylorRegression(QuasiSeparabilityRegression):
  def __init__(self, model, dims, lower, upper, regression_dim):
    """ Constructor of the Taylor subspace regression model """
    QuasiSeparabilityRegression.__init__(
      self, model, dims, lower, upper, regression_dim)
    self.order = 1
    self.coefficients = np.empty(self.order)

  def try_fit(self, eps):
    """ Applies the taylor regression in one dimension """

    # Try up to the maximal taylor order. Return immediately on success.
    for p in range(1, config.max_taylor_order + 1):
      self.order = p
      self.coefficients = np.empty(self.order)

      if config.debug:
        print(
          "Dimension " + str(self.regression_dim) + " taylor (order = " + str(p)
          + ") regression ... ", flush=True)

      # Tries to fit the current regression config
      self.fit()
      if self.validate(eps):
        return True

    return False

  def fit_at_sample(self, sample):
    """ Performs the regression over the subspace defined by the sample """

    # Number of samples is at least as large as the number of coefficents
    n_coefficients = self.order + 1
    n_samples = n_coefficients + config.taylor_regression_samples
    sample = np.full(1, sample) if self.dims == 1 else sample

    # Sample random points in the subspace
    X = self.sampler.sample_dimension(n_samples, self.regression_dim, sample)
    y = self.sampler.eval_samples()

    # Compute regression matrix X (linear system)
    X_reg = np.empty((n_samples, n_coefficients))
    for i in range(n_samples):
      X_i = np.full(1, X[i]) if self.dims == 1 else X[i]
      for j in range(n_coefficients):
        X_reg[i][j] = X_i[self.regression_dim] ** j

    # Perform regression via normal equation
    coeffs = np.linalg.solve(
      np.matmul(X_reg.transpose(), X_reg),
      np.matmul(X_reg.transpose(), y))

    # Return all coefficients apart from the constant term
    return coeffs[1:]

  def __call__(self, x):
    """ Predicts the value at point x """
    y = 0
    for i in range(len(self.coefficients)):
      y += self.coefficients[i] * (x ** (i+1))

    return y

  def __str__(self):
    """ Creates a string representation with possible line breaks """
    code = ""
    for i in range(len(self.coefficients)):
      code += str(self.coefficients[i])
      if i==0:
        code += " * x"
      else:
        code += " * std::pow(x, " + str(i+1) + ")"
      code += " +$lb$"

    return code[:-6]
```

Code Listing 8: Quasi-Taylor regression model.

```python
import warnings
import numpy as np
from scipy.optimize import curve_fit, OptimizeWarning

# Import abstract parent class
from rsqtoa.rsqtoa_subspace_regression import QuasiSeparabilityRegression

# Import global config
from rsqtoa import rsqtoa_config as config

# ------------------------------------------------------------------------- #

def exp_regressor(x, a, b, c):
  """ Exponential regressor """
  return a * np.exp(b * x) + c

def exp_regressor_jac(x, a, b, c):
  """ Jacobian of exponential regressor """
  jac = np.zeros((3, len(x)))
  jac[0,:] = np.exp(b * x)
  jac[1,:] = x * a * np.exp(b * x)
  jac[2,:] = 1

  return jac

# ------------------------------------------------------------------------- #

class QuasiExponentialRegression(QuasiSeparabilityRegression):
  def __init__(self, model, dims, lower, upper, regression_dim):
    """ Constructor of the exponential subspace regression model """
    QuasiSeparabilityRegression.__init__(
      self, model, dims, lower, upper, regression_dim)
    self.coefficients = np.empty(2)

  def try_fit(self, eps):
    """ Applies the exponential regression in one dimension """

    # Quick return if we don't want exponential regressions
    if config.exponential_regression_samples == 0:
      self.max_error = np.Inf
      return False

    if config.debug:
      print(
        "Dimension " + str(self.regression_dim) +
        " exponential regression ... ", flush=True)

    # Tries to fit the current regression config
    self.fit()
    return self.validate(eps)

  def fit_at_sample(self, sample):
    """ Performs the regression over the subspace defined by the sample """
    n_samples = config.exponential_regression_samples
    sample = np.full(1, sample) if self.dims == 1 else sample

    # Sample random points in the subspace
    X = self.sampler.sample_dimension(n_samples, self.regression_dim, sample)
    y = self.sampler.eval_samples()

    # Compute regression X
    X_reg = np.empty(n_samples)
    for i in range(n_samples):
      X_i = np.full(1, X[i]) if self.dims == 1 else X[i]
      X_reg[i] = X_i[self.regression_dim]

    # Lower bound, upper bound and mid point
    distance = self.upper[self.regression_dim] - self.lower[self.regression_dim]
    lb = np.copy(sample)
    lb[self.regression_dim] = self.lower[self.regression_dim]
    mid = np.copy(sample)
    mid[self.regression_dim] = distance / 2.0
    ub = np.copy(sample)
    ub[self.regression_dim] = self.upper[self.regression_dim]

    # Disable scipy warnings
    warnings.simplefilter("ignore", OptimizeWarning)

    # Estimate for the coefficients
    coeffs = np.full(3, np.Inf)
    b = self.derivative_at(lb)
    if b == 0:
      return coeffs[:-1]
    b = self.derivative_at(ub) / b
    if b <= 0:
      return coeffs[:-1]

    b = np.log(b) / distance
    tmp = np.exp(b * mid[self.regression_dim])
    if b * tmp == 0:
      return coeffs[:-1]
    a = self.derivative_at(mid) / (b * tmp)
    c = self.model(mid) - a * tmp

    coeffs[0] = a
    coeffs[1] = b
```

```
    coeffs[2] = c

    # Perform non-linear regression
    coeffs, _ = curve_fit(exp_regressor, X_reg, y, p0=coeffs)

    # Reenable scipy warnings
    warnings.simplefilter("default", OptimizeWarning)

    return coeffs[:-1]

  def __call__(self, x):
    """ Predicts the value at point x """
    return exp_regressor(x, *self.coefficients, 0)

  def __str__(self):
    """ Creates a string representation with possible line breaks """
    code = str(self.coefficients[0]) + " * std::exp($lb$"
    code += str(self.coefficients[1]) + " * x$lb$)"

    return code
```

Code Listing 9: Quasi-Exponential regression model.

```python
import warnings
import numpy as np
from scipy.optimize import curve_fit, fmin, OptimizeWarning

# Import abstract parent class
from rsqtoa.rsqtoa_subspace_regression import QuasiSeparabilityRegression

# Import global config
from rsqtoa import rsqtoa_config as config

# ------------------------------------------------------------------------- #

def sin_regressor(x, a, b, c):
  """ Periodic regressor """
  return a * np.sin(b * x + c)

# ------------------------------------------------------------------------- #

class QuasiPeriodicRegression(QuasiSeparabilityRegression):
  def __init__(self, model, dims, lower, upper, regression_dim):
    """ Constructor of the periodic subspace regression model """
    QuasiSeparabilityRegression.__init__(
      self, model, dims, lower, upper, regression_dim)
    self.coefficients = np.empty(3)

  def try_fit(self, eps):
    """ Applies the periodic regression in one dimension """

    # Quick return if we don't want periodic regressions
    if config.periodic_regression_samples == 0:
      self.max_error = np.Inf
      return False

    if config.debug:
      print(
        "Dimension " + str(self.regression_dim) +
        " periodic regression ... ", flush=True)

    # Tries to fit the current regression config
    self.fit()
    return self.validate(eps)

  def _model_slice(self, x, sample):
    """ Scalar model slice in the regression dimension. """
    x_eval = sample
    x_eval[self.regression_dim] = x
    return self.model(x_eval)

  def fit_at_sample(self, sample):
    """ Performs the regression over the subspace defined by the sample """
    n_samples = config.periodic_regression_samples
    sample = np.full(1, sample) if self.dims == 1 else sample

    # Sample random points in the subspace
    X = self.sampler.sample_dimension(n_samples, self.regression_dim, sample)
    y = self.sampler.eval_samples()

    # Compute regression X
    X_reg = np.empty(n_samples)
    for i in range(n_samples):
      X_i = np.full(1, X[i]) if self.dims == 1 else X[i]
      X_reg[i] = X_i[self.regression_dim]

    # Midpoint of the scalar domain
    mid_point = self.lower[self.regression_dim]
    mid_point += (
      self.upper[self.regression_dim] - self.lower[self.regression_dim]
    ) / 2.0

    # Disable scipy warnings
    warnings.simplefilter("ignore", RuntimeWarning)
    warnings.simplefilter("ignore", OptimizeWarning)

    # Estimates for the amplitude, frequency and phase
    min_x = fmin(lambda x: self._model_slice(x, sample), mid_point, disp=False)
    max_x = fmin(lambda x: -self._model_slice(x, sample), mid_point, disp=False)
    min_y = self._model_slice(min_x, sample)
    max_y = self._model_slice(max_x, sample)

    # Get the amplitude
    amp = (max_y - min_y) / 2.0

    # Get the frequency
    freq = np.pi / np.abs(min_x - max_x)

    # Get the phase
    phase = np.fmod(max_x - np.pi / (2*freq), (2*np.pi / freq))
    if phase > np.pi / freq:
      phase -= 2*np.pi / freq
    elif phase < -np.pi / freq:
      phase += 2*np.pi / freq

    # Perform non-linearregression in order to polish the results
    y -= (max_y - amp)
    coeffs = np.empty(3)
    coeffs[0] = amp
```

```
    coeffs[1] = freq
    coeffs[2] = phase

    coeffs, _ = curve_fit(sin_regressor, X_reg, y, p0=coeffs)

    # Reenable scipy warnings
    warnings.simplefilter("default", RuntimeWarning)
    warnings.simplefilter("default", OptimizeWarning)

    return coeffs

  def __call__(self, x):
    """ Predicts the value at point x """
    return sin_regressor(x, *self.coefficients)

  def __str__(self):
    """ Creates a string representation with possible line breaks """
    code = str(self.coefficients[0]) + " * std::sin($lb$"
    code += str(self.coefficients[1]) + " * x + "
    code += str(self.coefficients[2]) + "$lb$)"

    return code
```

Code Listing 10: Quasi-Periodic regression model.

```python
import copy
import numpy as np
import itertools
from datetime import datetime

# Import regression classes
from rsqtoa.rsqtoa_taylor_regression import QuasiTaylorRegression
from rsqtoa.rsqtoa_periodic_regression import QuasiPeriodicRegression
from rsqtoa.rsqtoa_exponential_regression import QuasiExponentialRegression
from rsqtoa.rsqtoa_sampling import RandomSampler, LatinHypercubeSampler

# Import global config and statistics
from rsqtoa import rsqtoa_config as config
from rsqtoa import rsqtoa_statistics as stats

# All regression types we want to use in our analysis
REGRESSION_TYPES = [
  QuasiTaylorRegression, QuasiPeriodicRegression,
  QuasiExponentialRegression]

# ------------------------------------------------------------------------- #

class SubspaceNode:
  def __init__(self, model, dims, lower, upper, parent=None, id=0):
    """ Constructor of the subspace node """
    self.parent = parent
    self.children = []
    self.split_dim = -1
    self.split_value = np.NaN
    self.is_leaf = False

    # Models
    self.original_model = model
    self.approximated_model = None

    # Bounds of the subspace
    self.dims = dims
    self.lower = lower.astype(float)
    self.upper = upper.astype(float)

    if config.sampling_strategy == 'random':
      self.sampler = RandomSampler(model, dims, lower, upper)
    else:
      self.sampler = LatinHypercubeSampler(model, dims, lower, upper)

    # Set subspace id
    self.id = id
    self.leaf_id = -1

    # Regression configurations
    self.regressions = []
    self.offset = None
    if parent:
      self.regression_dims = copy.deepcopy(parent.non_separable_dims)
      self.eps = parent.remaining_eps
      self.remaining_eps = parent.remaining_eps
    else:
      self.regression_dims = list(range(dims))
      self.eps = config.epsilon
      self.remaining_eps = config.epsilon

    self.non_separable_dims = copy.deepcopy(self.regression_dims)
    self.regression_errors = {
      d: {r: np.inf for r in REGRESSION_TYPES} for d in self.regression_dims}

    if config.debug:
      print(
        "\nNew Subspace Node: lower={0} upper={1}".format(
          self.lower, self.upper
        ), flush=True)

  def generate_regressions(self, regression_types=REGRESSION_TYPES):
    """ Tries to generate subspace regressions for the remaining dimensions """

    # Generate regressions
    tmp_regressions = [[] for _ in range(self.dims)]
    for reg_dim in self.regression_dims:
      # Iterate over regression models
      for Regression in regression_types:
        reg = Regression(
          self.original_model, self.dims, self.lower, self.upper, reg_dim)
        if reg.try_fit(self.eps):
          tmp_regressions[reg_dim].append(reg)
          if config.regression_selection == "greedy":
            break

        self.regression_errors[reg_dim][Regression] = reg.max_error

    # Remove non succesful dimensions
    successful_dims = self.dims - tmp_regressions.count([])
    tmp_regressions = [reg for dim in tmp_regressions for reg in dim]

    # Chose best combination of regressions
    for k in reversed(range(successful_dims)):
      regression_combinations = itertools.combinations(tmp_regressions, k+1)
      min_eps = np.Inf
```

```
      best_combination = None
      for comb in regression_combinations:
        # Filter out permutations with duplicate dimensions
        if len(set(map(lambda reg: reg.regression_dim, comb))) != len(comb):
          continue

        tmp_eps = np.sum(list(map(lambda reg: reg.max_error, comb)))
        if tmp_eps < min_eps:
          min_eps = tmp_eps
          best_combination = comb

      if min_eps <= self.eps:
        self.regressions = list(best_combination)
        self.remaining_eps = self.eps - min_eps
        break

    for reg in self.regressions:
      self.regression_errors.pop(reg.regression_dim)
      self.non_separable_dims.remove(reg.regression_dim)

    self.is_leaf = len(self.non_separable_dims) == 0

    # Calculate offset for subspaces with full regression
    if self.is_leaf:
      X = self.sampler.sample_domain(config.regression_samples)
      Y = self.evaluate_reduced_model(X.transpose())
      self.offset = np.sum(Y) / config.regression_samples
      if config.debug:
        print("Calcualting offset for subspace:  " + str(self.offset))

  def generate_subspaces(self):
    """ Tries to find subspaces with successful regressions """
    found_new_subspaces = False

    # If this node is already divided, pass the call onto its children
    if self.children or self.is_leaf:
      for child in self.children:
        found_new_subspaces |= child.generate_subspaces()
      return found_new_subspaces

    subspaces1 = []
    subspaces2 = []
    new_subspace1 = None
    new_subspace2 = None

    # Sort the dimensions for the subspace division (only with greedy selection)
    split_dims = range(self.dims)
    if config.subspace_selection == "greedy":
      if config.subspace_ordering == "min_error":
        split_dims = sorted(
          split_dims, key=lambda d: np.min(self.regression_errors[d].values()))
      elif config.subspace_ordering == "max_error":
        split_dims = sorted(
          split_dims, key=lambda d: np.min(self.regression_errors[d].values()),
          reverse=True)
      elif config.subspace_ordering == "area":
        split_dims = sorted(
          split_dims, key=lambda d: self.upper[d] - self.lower[d],
          reverse=True)

    # Perform subspace division
    for dim in range(self.dims):
      node1, node2, eff1, eff2 = self.split_dimension(dim)

      if config.subspace_selection == "greedy" and (eff1 > 1 or eff2 > 1):
        new_subspace1 = node1
        new_subspace2 = node2
        break
      else:
        subspaces1.append(node1)
        subspaces2.append(node2)

    # Choose best subspaces
    if config.subspace_selection == "best":
      max_eff = 1
      for ss1, ss2 in zip(subspaces1, subspaces2):
        eff = ss1.calculate_effectiveness() * ss2.calculate_effectiveness()
        if eff > max_eff:
          new_subspace1 = ss1
          new_subspace2 = ss2
          eff1 = ss1.calculate_effectiveness()
          eff2 = ss2.calculate_effectiveness()
          max_eff = eff

    # Choose best subspaces with no regressions
    if not (new_subspace1 or new_subspace2):
      if not config.subspace_needs_regression:
        min_error = np.Inf
        for ss1, ss2 in zip(subspaces1, subspaces2):
          err1 = ss1.get_smallest_regression_error()
          err2 = ss2.get_smallest_regression_error()
          err = err1 * err2
          if err < min_error:
            new_subspace1 = ss1
            new_subspace2 = ss2
            min_error = err
```

```
    # Add new subspaces
    if new_subspace1 and new_subspace2:
      if config.debug:
        print("Adding new subspaces ...")
        print("... Effectiveness: " + str(eff1) + " " + str(eff2))
      self.children.append(new_subspace1)
      self.children.append(new_subspace2)
      found_new_subspaces = True
    else:
      if config.debug:
        print("Couldn't find new subspaces.")

    self.is_leaf = not found_new_subspaces

    return found_new_subspaces

  def get_smallest_regression_error(self):
    """ Return the smallest regression error. """
    min_err = np.Inf
    for dim in self.non_separable_dims.values():
      for error in dim.values():
        min_error = np.min(min_err, error)
    return min_error

  def split_dimension(self, dim):
    """ Search for new subspaces """
    # Subspace boundaries
    self.split_dim = dim
    lb1 = np.copy(self.lower)
    lb2 = np.copy(self.lower)
    ub1 = np.copy(self.upper)
    ub2 = np.copy(self.upper)

    # Generate the subspace node from the current config
    def generate_nodes():
      node1 = SubspaceNode(self.original_model, self.dims, lb1, ub1, self, 1)
      node1.generate_regressions()
      eff1 = node1.calculate_effectiveness()
      node2 = SubspaceNode(self.original_model, self.dims, lb2, ub2, self, 2)
      node2.generate_regressions()
      eff2 = node2.calculate_effectiveness()
      return node1, node2, eff1, eff2

    # Initial split
    self.split_value = (self.upper[dim] + self.lower[dim]) / 2.0
    lb2[dim] = self.split_value
    ub1[dim] = self.split_value

    if config.debug:
      print("\nSplitting dim " + str(dim) + " at " + str(self.split_value))

    node1, node2, eff1, eff2 = generate_nodes()
    print("Effectiveness: " + str(eff1) + " " + str(eff2))

    # Search for at least one effective subspace (reduce first subspace)
    for i in range(config.subspace_reduction_steps):
      if eff1 > 1 or eff2 > 1:
        break

      # Shift split value to 'the left'
      self.split_value = (self.split_value + self.lower[dim]) / 2.0
      if self.split_value - self.lower[dim] < config.subspace_threshold:
        break

      lb2[dim] = self.split_value
      ub1[dim] = self.split_value

      if config.debug:
        print("\nLeft reduction step " + str(i+1) + ":")
        print("\nSplitting dim " + str(dim) + " at " + str(self.split_value))

      node1, node2, eff1, eff2 = generate_nodes()
      print("Effectiveness: " + str(eff1) + " " + str(eff2))

    # Search for at least one effective subspace (reduce second subspace)
    for i in range(config.subspace_reduction_steps):
      if eff1 > 1 or eff2 > 1:
        break

      # Shift split value to 'the right'
      self.split_value = (self.upper[dim] + self.split_value) / 2.0
      if self.upper[dim] - self.split_value < config.subspace_threshold:
        break

      lb2[dim] = self.split_value
      ub1[dim] = self.split_value

      if config.debug:
        print("\nRight reduction step " + str(i+1) + ":")
        print("\nSplitting dim " + str(dim) + " at " + str(self.split_value))

      node1, node2, eff1, eff2 = generate_nodes()
      print("Effectiveness: " + str(eff1) + " " + str(eff2))

    # Expand left subspace if it is effective
    if eff1 > 1:
      left_search_bound = self.split_value
```

```
      right_search_bound = self.upper[dim]

      for i in range(config.subspace_expansion_steps):
        # Cancel expansion if both subspaces are effective
        if eff1 > 1 and eff2 > 1:
          break

        # Shift split value
        self.split_value = (left_search_bound + right_search_bound) / 2.0
        lb2[dim] = self.split_value
        ub1[dim] = self.split_value
        if config.debug:
          print("\nLeft expansion step " + str(i+1) + ":")
          print("\nSplitting dim " + str(dim) + " at " + str(self.split_value))

        tmp_node1, tmp_node2, tmp_eff1, tmp_eff2 = generate_nodes()

        # If the efficiency didn't regress, save new subspaces
        if tmp_eff1 >= eff1:
          left_search_bound = self.split_value
          node1, node2, eff1, eff2 = tmp_node1, tmp_node2, tmp_eff1, tmp_eff2
          if config.debug:
            print("Success - new effectiveness: " + str(eff1))
        else:
          right_search_bound = self.split_value
          if config.debug:
            print("Failure - keep old effectiveness: " + str(eff1))

    # Expand right subspace if it is effective
    if eff2 > 2:
      left_search_bound = self.lower[dim]
      right_search_bound = self.split_value

      for i in range(config.subspace_expansion_steps):
        # Cancel expansion if both subspaces are effective
        if eff1 > 1 and eff2 > 1:
          break

        # Shift split value
        self.split_value = (left_search_bound + right_search_bound) / 2.0
        lb2[dim] = self.split_value
        ub1[dim] = self.split_value
        if config.debug:
          print("\nRight expansion step " + str(i+1) + ":")
          print("\nSplitting dim " + str(dim) + " at " + str(self.split_value))

        tmp_node1, tmp_node2, tmp_eff1, tmp_eff2 = generate_nodes()

        # If the efficiency didn't regress, save new subspaces
        if tmp_eff2 >= eff2:
          right_search_bound = self.split_value
          node1, node2, eff1, eff2 = tmp_node1, tmp_node2, tmp_eff1, tmp_eff2
          if config.debug:
            print("Success - new effectiveness: " + str(eff2))
        else:
          left_search_bound = self.split_value
          if config.debug:
            print("Failure - keep old effectiveness: " + str(eff2))

    return node1, node2, eff1, eff2

  def calculate_effectiveness(self):
    """ Calculate effectiveness of the model reduction """

    # Relative amount of separable dimensions
    reduced_dims_factor = len(self.regression_dims)
    reduced_dims_factor -= len(self.non_separable_dims)
    reduced_dims_factor /= len(self.regression_dims)

    # Quick return if non of the dimension was separable
    if reduced_dims_factor == 0:
      return 1

    # Relative volume of the current subdomain compared to the whole domain
    area_factor = self.calculate_subspace_volume()
    area_factor /= self.calculate_root_volume()

    # Factor which considers the remaining error threshold compared to the
    # original threshold.
    reduced_eps_factor = (self.eps - self.remaining_eps) / self.eps
    if reduced_eps_factor == 0 or len(self.non_separable_dims) == 0:
      reduced_eps_factor = np.finfo(np.float32).eps

    # Calculate effectiveness of the model reduction in the current subdomain
    return 2 + int(area_factor * reduced_dims_factor / reduced_eps_factor)

  def get_leaf(self, leaf_id):
    """ Returns the leaf with the given leaf id """
    if self.leaf_id == leaf_id:
      return self

    for child in self.children:
      tmp = child.get_leaf(leaf_id)
      if tmp:
        return tmp

    return None
```

```python
  def get_leafs(self):
    """ Returns all leaf nodes """
    if self.is_leaf:
      return [self]

    leafs = []
    for child in self.children:
      leafs += child.get_leafs()

    return leafs

  def get_ss_id(self):
    """ Returns the unique subspace id """
    if self.parent:
      return self.parent.get_ss_id() + [str(self.id)]
    return []

  def get_regressions(self):
    """ Returns a list of regressions for the current subspace """
    reg = []
    if self.parent:
      reg += self.parent.get_regressions()
    reg += self.regressions
    return reg

  def get_regressions_with_ss_id(self):
    """ Returns a list of regressions with their subspace ids """
    reg = []
    ss_id = []
    if self.parent:
      parent_reg, parent_ss_id = self.parent.get_regressions_with_ss_id()
      reg += parent_reg
      ss_id += parent_ss_id
    reg += self.regressions
    ss_id += [self.get_ss_id()] * len(self.regressions)
    return reg, ss_id

  def get_all_subspaces(self):
    """ Returns a list of all subspaces """
    ss = [self]
    for child in self.children:
      ss += child.get_all_subspaces()
    return ss

  def has_no_new_regressions(self):
    has_no_regressions = len(self.regressions) == 0
    for child in self.children:
      has_no_regressions &= child.has_no_new_regressions()
    return has_no_regressions

  def identify_leafs(self):
    """ Assign leaf ids and remove unnecessary subspaces """
    children_are_unnecessary = True
    for child in self.children:
      children_are_unnecessary &= child.has_no_new_regressions()

    # Remove childeren if they don't add any new regressions
    if children_are_unnecessary:
      self.children.clear()
      self.is_leaf = True

    if self.is_leaf:
      self.leaf_id = stats.leafs
      stats.leafs += 1
      stats.average_dim_reduction += self.dims - len(self.non_separable_dims)
      if self.remaining_eps < stats.remaining_epsilon:
        stats.remaining_epsilon = self.remaining_eps
    else:
      for child in self.children:
        child.identify_leafs()

  def calculate_subspace_volume(self):
    """ Calculate the volume of the current subdomain """
    volume = 1
    for dim in range(self.dims):
      volume *= (self.upper[dim] - self.lower[dim])
    return volume

  def calculate_root_volume(self):
    """ Calculate the volume of the entire domain """
    if self.parent:
      return self.parent.calculate_root_volume()
    return self.calculate_subspace_volume()

  def _check_bounds(self, x):
    """ Check if the point is inside the bounds. """
    output_dims = list(np.shape(x))
    output_dims = output_dims[1:]
    inside = np.full(tuple(output_dims), True, dtype=bool)
    for dim in range(self.dims):
      inside = np.logical_and(inside, np.logical_and(
        x[dim, :] >= self.lower[dim], x[dim, :] <= self.upper[dim]))

    return inside

  def evaluate_reduced_model(self, x):
```

```
    """ Evaluates the reduced model """
    output_dims = list(np.shape(x))
    output_dims = output_dims[1:]
    y = np.zeros(shape=tuple(output_dims))

    inside_points = self._check_bounds(x)
    if self.is_leaf:
      if np.ndim(x) == 1:
        x_tmp = np.copy(x).reshape((1, len(x)))
      else:
        x_tmp = np.copy(x)

      for reg in self.get_regressions():
        x_tmp[reg.regression_dim, :][inside_points] = reg.sample_value
        y[inside_points] -= reg(reg.sample_value)

      try:
        y[inside_points] += self.original_model(x_tmp[:, inside_points])
      except TypeError:
        it = np.nditer(inside_points, flags=['multi_index'])
        for x in it:
          print(it.multi_index)
          if inside_points[it.multi_index]:
            y[it.multi_index] = self.original_model(x_tmp[:, it.multi_index])

      if self.offset is not None:
        y[inside_points] -= self.offset
    else:
      overlap = np.zeros_like(self._check_bounds(x), dtype=int)
      for child in self.children:
        child_check = child._check_bounds(x)
        if child_check.any():
          overlap += child_check
          y += child.evaluate_reduced_model(x)

      y[overlap > 0] /= overlap[overlap > 0]

    return y

  def evaluate_reassembled_model(self, x):
    """ Evaluates the reassembled model """
    output_dims = list(np.shape(x))
    output_dims = output_dims[1:]
    y = np.zeros(shape=tuple(output_dims))

    inside_points = self._check_bounds(x)
    if self.is_leaf:
      if np.ndim(x) == 1:
        x_tmp = np.copy(x).reshape((1, len(x)))
      else:
        x_tmp = np.copy(x)

      for reg in self.get_regressions():
        y[inside_points] += reg(x_tmp[reg.regression_dim, :][inside_points])

      if self.offset is not None:
        y[inside_points] += self.offset
      else:
        # Evaluate the approximated model. Use the reduced model if there is
        # no approximation yet
        if self.approximated_model:
          y[inside_points] += (
            self.approximated_model(x).reshape(np.shape(y))[inside_points])
        else:
          y[inside_points] += self.evaluate_reduced_model(x[:, inside_points])
    else:
      overlap = np.zeros_like(self._check_bounds(x), dtype=int)
      for child in self.children:
        child_check = child._check_bounds(x)
        if child_check.any():
          overlap += child_check
          y += child.evaluate_reassembled_model(x)

      y[overlap > 0] /= overlap[overlap > 0]

    return y

  def get_sample_coefficient(self):
    """ Calculate the percentual amount of samples for this domain """
    reduced_dims_factor = len(self.non_separable_dims)
    reduced_dims_factor /= self.dims

    area_factor = self.calculate_subspace_volume()
    area_factor /= self.calculate_root_volume()

    return reduced_dims_factor * area_factor

  def sample_domain(self, n):
    """ Sample the current domain """
    return self.sampler.sample_domain(n)

class SubspaceTree:
  def __init__(self, model, dims, lower, upper):
    """ Constructor of the subspace tree """
    self.reset_stats()
    stats.start_time = datetime.now()
    self.model = model
```

```
    self.root = SubspaceNode(model, dims, lower, upper)
    self.root.generate_regressions()

    for i in range(config.subspace_depth):
      print("\nSubspace division depth: " + str(i+1))
      if not self.root.generate_subspaces():
        print("Couldn't find any new subspaces.")
        break

    # Assign leaf ids and simplify subspace tree if possible
    self.root.identify_leafs()
    stats.end_time = datetime.now()
    stats.average_dim_reduction /= stats.leafs

    if config.debug:
      print("\n\nRSQTOA Statistics:")
      print("Elapsed time: " + str(stats.end_time - stats.start_time))
      print("Samples: " + str(stats.amount_of_samples))
      print("Remaining epsilon: " + str(stats.remaining_epsilon))
      print("Amount of subspaces: " + str(stats.leafs))
      print("Average reduced dimensions: " + str(stats.average_dim_reduction))

  def get_leaf(self, leaf_id):
    """ Returns the leaf with the given id """
    return self.root.get_leaf(leaf_id)

  def get_leafs(self):
    """ Returns all leafs """
    return self.root.get_leafs()

  def get_all_subspaces(self):
    """ Returns a list of all subspaces """
    return self.root.get_all_subspaces()

  def evaluate_reduced_model(self, x):
    """ Evaluates the reduced model """
    if self.root._check_bounds(x).all():
      return self.root.evaluate_reduced_model(x)

    raise RuntimeError("Input x={0} out of bounds!".format(x))

  def evaluate_reassembled_model(self, x):
    """ Evaluates the reduced model """
    if self.root._check_bounds(x).all():
      return self.root.evaluate_reassembled_model(x)

    raise RuntimeError("Input x={0} out of bounds!".format(x))

  def reset_stats(self):
    """ Reset statistics """
    stats.leafs = 0
    stats.remaining_epsilon = np.Inf
    stats.start_time = None
    stats.end_time = None
    stats.amount_of_samples = 0
    stats.average_dim_reduction = 0.0
```

Code Listing 11: The subspace regression tree.

```python
epsilon = 10e-2
""" The overall error threshold """

sampling_strategy = "latin"
""" Sampling strategy. Possible values:
    random, latin, latin_maximin, latin_corr """

regression_samples = 10
""" Amount of points at which a subspace regression is performed """

test_samples = 10
""" Amount of points at which the regression is tested in the entire domain """

test_values = 10
""" Amount of values which are tested with the test points """

debug = False
""" Debug mode flag """

# ---------------------------------------------------------------------------- #

max_taylor_order = 1
""" Maximum taylor order """

taylor_regression_samples = 0
""" Amount of samples used to improve the taylor regression.
    Taylor regression is always enabled and uses at least the minimum amount
    of samples such that the least squares problem is never underdetermined. """

periodic_regression_samples = 50
""" Amount of samples used to improve the periodic regression.
    Set to zero to disable periodic regression. """

exponential_regression_samples = 50
""" Amount of samples used to do the exponential regression.
    Set to zero to disable exponential regression. """

regression_selection = "greedy"
""" Regression selection strategy. Possible values:
    greedy, best """

# ---------------------------------------------------------------------------- #

subspace_ordering = "default"
""" The ordering strategy for the remaining dimensions. Possible values:
    default, max_error, min_error, area """

subspace_depth = 2
""" Maximum recursive subspace divisions """

subspace_threshold = 1e-1
""" Minimum subspace domain size """

subspace_needs_regression = True
""" Whether we allow subspaces with no regression or not """

subspace_selection = "greedy"
""" Subspace selection strategy. Possible values:
    greedy, best """

subspace_expansion_steps = 0
""" Amount of steps in the adaptive subspace expansion """

subspace_reduction_steps = 0
""" Amount of steps in the adaptive subspace reduction """

# ---------------------------------------------------------------------------- #

regressions = []
""" List which stores the current regressions """

reduced_model = None
""" Reduced model """

reduced_dims = 0
""" Reduced amount of dimensions """

reduced_lb = 0
""" Lower boundary of the reduced model domain """

reduced_ub = 0
""" Upper boundary of the reduced model domain """
```

Code Listing 12: Global configurations used during the analysis.

```python
import numpy as np

remaining_epsilon = np.Inf
""" The remaining epsilon in the whole domain """

start_time = None
""" The time when the analysis started """

end_time = None
""" The time when the analysis ended """

amount_of_samples = 0
""" The amount of samples that were used """

average_dim_reduction = 0.0
""" Average amount of reduced dimensions """

leafs = 0
""" Amount of subspaces """
```

Code Listing 13: Global statistics which are collected during the analysis.

```python
import sys
import importlib
import numpy as np

# Import global config
from rsqtoa import rsqtoa_config as config
import rsqtoa.codegen.rsqtoa_codegen_config as codegen_config
from rsqtoa import rsqtoa_statistics as stats

# Import codegen and subspace regression modules
from rsqtoa.rsqtoa_args import _create_config
from rsqtoa.rsqtoa_subspace_tree import SubspaceTree
from rsqtoa.codegen import _generate_reduced_model

# ------------------------------------------------------------------------- #

def create_config(cfg=None, argv=None):
  """ Functional interface to load a new config """

  # Subspace regression
  return _create_config(cfg=cfg, argv=argv)

def create_subspace_tree(yaml_config, model=None):
  """ Functional interface to create a subspace tree for a given model """

  if yaml_config["model_dir"]:
    sys.path.append(yaml_config["model_dir"])

  # Import model
  if not model:
    if yaml_config["model_module"]:
      model_module = importlib.import_module(yaml_config["model_module"])
      model = getattr(model_module, yaml_config["model_function"])
    else:
      model = globals()[yaml_config["model_function"]]

  # Define domain and function
  dims = int(yaml_config["dimensions"])
  lower = np.array(yaml_config["lower_bounds"])
  upper = np.array(yaml_config["upper_bounds"])

  # Subspace regression config
  config.epsilon = float(
    yaml_config["epsilon"])
  config.sampling_strategy = yaml_config["sampling_strategy"]
  config.regression_samples = int(
    yaml_config["regression_samples"])
  config.test_samples = int(
    yaml_config["test_samples"])
  config.test_values = int(
    yaml_config["test_values"])
  config.max_taylor_order = int(
    yaml_config["max_taylor_order"])
  config.periodic_regression_samples = int(
    yaml_config["taylor_regression_samples"])
  config.periodic_regression_samples = int(
    yaml_config["periodic_regression_samples"])
  config.exponential_regression_samples = int(
    yaml_config["exponential_regression_samples"])
  config.debug = bool(
    yaml_config["debug"])

  config.subspace_ordering=bool(
    yaml_config["subspace_ordering"])
  config.subspace_depth = int(
    yaml_config["subspace_depth"])
  config.subspace_threshold = float(
    yaml_config["subspace_threshold"])
  config.subspace_needs_regression = bool(
    yaml_config["subspace_needs_regression"])
  config.subspace_selection = yaml_config["subspace_selection"]
  config.subspace_expansion_steps = int(
    yaml_config["subspace_expansion_steps"])
  config.subspace_reduction_steps = int(
    yaml_config["subspace_reduction_steps"])

  # Subspace regression
  return SubspaceTree(model, dims, lower, upper)

def generate_reduced_model(ss_tree, yaml_config):
  """ Functional interface to generate C++ code with the reduced models """

  # Setup output directories
  codegen_config.output_dir = yaml_config["output_dir"]
  codegen_config.model_file = yaml_config["model_file"]
  codegen_config.model_module = yaml_config["model_module"]
  codegen_config.model_function = yaml_config["model_function"]

  # Setup model
  codegen_config.full_dims = yaml_config["dimensions"]
  codegen_config.lower = yaml_config["lower_bounds"]
  codegen_config.upper = yaml_config["upper_bounds"]

  # Reduced model codegen
  codegen_config.leafs = stats.leafs

```

```
_generate_reduced_model(ss_tree)

# Write epsilon file only if there was an epsilon input file
if yaml_config["epsilon_input_file"]:
  with open(yaml_config["epsilon_output_file"], 'w') as epsilon_file:
    epsilon_file.write(str(config.epsilon))
```

Code Listing 14: The main API to the RSQTOA module.

```python
# Export RSQTOA API
from .rsqtoa import create_config
from .rsqtoa import create_subspace_tree
from .rsqtoa import generate_reduced_model

# Export Tensorflow integration
from .fapprox.rsqtoa_tf_function_approximator import FunctionApproximator
from .fapprox.rsqtoa_fapprox import train_model
from .fapprox.rsqtoa_fapprox import normalize_data
from .fapprox.rsqtoa_fapprox import denormalize_data
from .fapprox.rsqtoa_fapprox import reset_normalization
```

Code Listing 15: RSQTOA module exports.

```python
from argparse import ArgumentError
import os
import yaml

# ---------------------------------------------------------------------------- #

def _get_arg_value(argv, option, default = ""):
  """Extract a command line option from argument list.

  Parameters
  ----------
  argv : list
    List of command line arguments.

  option : str
    The option which the function shall search for.

  default : str, optional
    The default value in case the option is not in the argument list.

  """
  value = ""

  tmp_option = "--" + option + "="
  found = False
  for arg in argv:
    if arg.startswith(tmp_option):
      found = True
      value = arg[len(tmp_option):]
      if value.startswith('"') and value.endswith('"'):
        value = value[1:-1]

  if not found:
    if default != "":
      value = default
    else:
      raise RuntimeError("Expected command line option: '" + tmp_option + "'.")

  return value

def _get_arg_values(argv, option, separator=" ", default=[]):
  """Extract a command line option from argument list and separates it.

  Parameters
  ----------
  argv : list
    List of command line arguments.

  option : str
    The option which the function shall search for.

  default : str, optional
    The default value in case the option is not in the argument list.

  """
  value = _get_arg_value(argv, option, None)

  if value:
    return value.split(separator)
  else:
    return default

def _arg_exists(argv, option):
  """Check if a command line option exists in the argument list.

  Parameters
  ----------
  argv : list
    List of command line arguments.

  option : str
    The option which the function shall search for.

  """
  return ("--" + option) in argv

def _load_config_file(config_file):
  """Loads a yaml config file.

  Parameters
  ----------
  config_file : str
    The path to the config file.

  """
  yaml_config = None
  with open(config_file, 'r') as configuration:
    try:
      yaml_config = yaml.safe_load(configuration)
    except yaml.YAMLError as exc:
      raise SyntaxError(
        "Unable to parse configuration file: '" + config_file + "'.")

  return yaml_config

def _create_config(argv=None, cfg=None):
```

```python
    """Creates a unified configuration from the default config, provided config
       file and options passed through command line arguments.

    Parameters
    ----------
    argv : list
      List of command line arguments.

    """
    if not (argv or cfg):
      raise ArgumentError("Neither argv nor config is given.")

    # Load default config file
    default_config_file = os.path.join(
      os.path.dirname(os.path.realpath(__file__)), "default.yml")
    yaml_config = _load_config_file(default_config_file)

    # Set system dependent defaults
    yaml_config["output_dir"] = os.getcwd()

    # Look for a provided config file and replace the options in the config
    if cfg:
      specialized_config_file = cfg
    else:
      specialized_config_file = _get_arg_value(argv, "config", default=None)
    if specialized_config_file:
      specialized_config = _load_config_file(specialized_config_file)
      for key in yaml_config:
        if key in specialized_config and specialized_config[key] is not None:
          yaml_config[key] = specialized_config[key]

    # Parse remaining command line arguments and replace the options in the config
    if argv:
      for key in yaml_config:
        if isinstance(yaml_config[key], list):
          yaml_config[key] = _get_arg_values(argv, key, default=yaml_config[key])
        else:
          yaml_config[key] = _get_arg_value(argv, key, default=yaml_config[key])

    return yaml_config
```

Code Listing 16: An argument parser which handles the configuration object.

Hiermit versichere ich, diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht zu haben. Das zugehörige Formblatt des ZPA liegt zusätzlich der Abgabe der Arbeit bei.

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such. The relevant form of the ZPA is additionally attached to the submission of this work.

Aachen, March 8, 2022
Simon Märtens