Comparison of Regression Methods for Approximating a Noisy Runge Function

Live L. Storborg and Henrik Haug
Department of Mathematics, University of Oslo,
PO Box 1053, Blindern 0316, Oslo, Norway

AND

Simon S. Thommesen and Adam Falchenberg
Institute of Theoretical Astrophysics, University of Oslo,
PO Box 1029, Blindern 0315, Oslo, Norway

(Dated: October 7, 2025)

When approximating Runge's function OLS overfit after modelled with complexity corresponding to 10 polynomial degrees. Ridge regression introduces stability in the MSE, leading to overfitting after higher complexity than the OLS case. Ridge regression shrinks the polynomial coefficients, where in the most extreme case, some of the parameters are 2 orders of magnitude smaller than in the OLS case. When approximating the Runge function with gradient descent, we need a learning rate lower than $\eta = 0.37$, as the methods diverge for learning rates higher and equal to this. When using Lasso regression, Momentum and Adam converge fastest (70-84 iterations). Stochastic gradient descent does not outperform the other methods, although it could be useful for more complex problems. OLS is the simplest of the methods, but turns out to be generally the best to approximate the Runge function. Through the use of bootstrapping and cross-validation we find areas where Ridge and Lasso outperform OLS. These are areas with low number of sample points, and high noise and complexity.

I. INTRODUCTION

Machine learning and statistical modelling face a fundamental challenge: balancing model complexity with predictive accuracy. Overly simple models fail to capture underlying patterns (high bias) while overly complex models overfit to training data noise (high variance). Regression analysis is key for understanding these relationships, with polynomial fitting representing a classic example.

The Runge function, defined as

$$f(x) = \frac{1}{1 + 25x^2} \quad \text{for } x \in [-1, 1], \tag{1}$$

provides an ideal test case for studying regression methods due to its association with Runge's phenomenon. This phenomenon demonstrates that high-degree polynomial interpolation can experience severe oscillations near the boundaries, making it challenging to achieve stable approximations [1].

To simulate realistic experimental conditions, we add stochastic noise to the Runge function using a normal distribution:

$$y_i = f(x_i) + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2), \tag{2}$$

where ϵ_i represents random noise with zero mean and standard deviation σ. This noise addition mimics real-world measurement uncertainties and allows us to study how the regression methods handle noisy data.

In this project, we implement and compare three regression models: Ordinary Least Squares (OLS), Ridge regression, and Lasso regression. We explore how well these can approximate the Runge function through polynomial fitting. Our analysis includes a systematic investigation of hyperparameters such as regularization strength λ, learning rate η, polynomial degree, and bias-variance analysis using bootstrap resampling. We also implement various gradient descent optimization methods including momentum, AdaGrad, RMSprop, and ADAM to explore their effectiveness in finding optimal parameters.

In this article, section II describes the theoretical foundations of the regression methods and our implementation approach, including gradient descent algorithms and resampling techniques. Section III presents our experimental results, which are discussed in section IV where bias-variance analysis and systematic parameter optimization are studied to explore how well the models perform. Finally, Section V summarizes our key findings regarding regression analysis.

II. METHOD & THEORY

We will assume we have some data $\mathbf{y}$ described by the Runge function (1). To find an approximation $\tilde{\boldsymbol{y}}$ of the Runge function, we will apply polynomial regression. We will analyze the three regression models; OLS, Ridge and Lasso. Each approach has its own cost function to determine the optimal parameters $\boldsymbol{\theta}$.

To study the performance of the different models we use the two evaluation metrics, the Mean squared error (MSE) and the R^2-score function. The MSE measures the squared difference between the predicted and the true

value:

$$MSE(\mathbf{y},\tilde{\mathbf{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2\,. \tag{3}$$

The R^2 score, however, quantifies how well the model performs compared to a model that always predicts the mean value. Values range from 0 (model performs no better than the mean) to 1 (a perfect fit):

$$R^2(\mathbf{y},\tilde{\mathbf{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i-\tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i-\bar{y})^2}, \tag{4}$$

where $\tilde{y}_i$ is the i-th predicted value, y_i is the corresponding true value, and $\bar{y}$ is the mean value of $\mathbf{y}$, defined as

$$\bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i.$$

A. Creating the Design Matrix & Cost Function

For polynomial fitting, we seek to approximate the function using a polynomial of degree p:

$$\tilde{y}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_{p-1}x^{p-1} = \sum_{i=0}^{p-1}\theta_i x^i, \tag{5}$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \ldots, \theta_{p-1}]^T$ are the polynomial coefficients to be determined.

Given n data points, the design matrix $\mathbf{X}$ is constructed as the Vandermonde matrix

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{p-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{p-1} \end{bmatrix}, \tag{6}$$

where $x_1, x_2, \ldots, x_n$ are the input data points sampled uniformly from the interval $[-1, 1]$, and each row represents one data point with its corresponding polynomial features. In essence, the linear system $\mathbf{X}\boldsymbol{\theta} = \mathbf{y}$ relates the design matrix (6) and coefficients to the target values contained in $\mathbf{y}$. When we cannot recreate the data perfectly using our model, the predicted values are given by the design matrix and the optimal parameters $\hat{\boldsymbol{\theta}}$ found by the model, $\mathbf{X}\hat{\boldsymbol{\theta}} = \tilde{\mathbf{y}}$.

The optimal parameters $\hat{\boldsymbol{\theta}}$ are found by minimizing a cost function. The cost functions of the models are usually given by the MSE (3) (there will be some differences, see section II D) which expresses the difference between the predicted values and the data. When discussing the role of the design matrix, we saw that the predicted values $\tilde{\mathbf{y}}$ can be expressed as a function of the parameters $\boldsymbol{\theta}$ and the design matrix $\mathbf{X}$. Using this, as well as writing equation 3 in vector form, we can define the cost function

$$C(\boldsymbol{\theta}) = \frac{1}{n}\Big\{(\mathbf{y}-\mathbf{X}\boldsymbol{\theta})^T(\mathbf{y}-\mathbf{X}\boldsymbol{\theta})\Big\}. \tag{7}$$

Here, $\mathbf{y}$ is the data, while $\tilde{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$ is the predicted values from our model, given by the calculated parameters $\boldsymbol{\theta}$. Since the data and the design matrix are given, the cost function is a function of only the model parameters.

To find the parameters that minimize the cost function, we differentiate the cost function with respect to $\boldsymbol{\theta}$ and set the derivative equal to zero:

$$\frac{\partial C(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0. \tag{8}$$

Solving this equation will result in a stationary solution. To determine which stationary solution we get, we can examine the second derivative of the cost function.

$$\frac{\partial}{\partial \boldsymbol{\theta}}\frac{\partial C(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^T} = H \tag{9}$$

Here, H is the Hessian matrix. The Hessian matrix, given in equation 10, is positive semi-definite. Because of this, we know that the cost function is convex. A convex function will not have maxima or saddle points, only minima.

$$H = \frac{2}{n}\mathbf{X}^T\mathbf{X} \tag{10}$$

High complexity models can lead to an ill-conditioned Hessian matrix. This is especially true in our case with the Runge function, since it is defined in $x \in [-1, 1]$. If we look at the Vandermonde matrix (6), we can see that for high degree polynomials the last columns can become very similar. For instance, in a 15th-degree polynomial, the columns corresponding to x^{14} and x^{15} take on values that are very close to zero for most x-values except near the boundaries. As a result, the columns become nearly linearly dependent.

While this is not a theoretical issue, it poses practical problems: even small numerical errors can cause large changes in the model. In other words, the design matrix has a large condition number, meaning the system is highly sensitive to round-off errors, which causes ill-conditioning. The Hessian matrix amplifies this issue since it effectively squares the condition number. Later, we will see that the methods require the inverse of the Hessian matrix. The ill-conditioned nature of the matrix can lead to unstable and inaccurate solutions due to rounding errors.

B. Data splitting

In machine learning, it is customary to split the data. The data is typically split into training and test data.

The training data is used to create the model $\tilde{\mathbf{y}}$. This is the data that is applied in the regression methods. The essence of data splitting is to train our model on the training data and test its accuracy and generality on the test data. In doing this, we can measure the variation of bias and variance with respect to the complexity of the model. Usually, the training data is chosen to be approximately 2/3 to 4/5 of the data.

An advantage of splitting the data into training and test sets is that it allows us to detect overfitting. Overfitting happens when the model captures the noise of the training data, and deviates from the underlying pattern of our full dataset. The training data is used to determine the optimal parameters $\boldsymbol{\theta}$. These parameters are then applied to the test data to obtain an independent approximation of the underlying function. By comparing the MSE on the training and test sets, we can assess whether the model is overfitting for higher complexity models (like higher degree polynomials).

C. Scaling

In machine learning, different features often have vastly different units and numerical scales. Without proper scaling, algorithms frequently perform poorly since features with large numerical values, regardless of their physical significance, will dominate the optimization process and create numerical instabilities. For polynomial fitting, this issue becomes particularly pronounced with high-order features like x^{15}, which can lead to ill-conditioned matrices, numerical overflow, and poor convergence in gradient descent algorithms.

To address these challenges, we standardize the design matrix $\mathbf{X}$ using z-score normalization and apply mean centering to the target variable $\mathbf{y}$. For the design matrix, each feature is transformed according to

$$\mathbf{X}_{norm} = \frac{\mathbf{X} - \boldsymbol{\mu}_X}{\boldsymbol{\sigma}_X}, \tag{11}$$

where $\boldsymbol{\mu}_X$ and $\boldsymbol{\sigma}_X$ are the column-wise means and standard deviations computed from the training data. To prevent division by zero when encountering near-constant features, any standard deviation below 10^{-8} is set to 1.0, effectively leaving such features unscaled.

The target variable is mean-centered by

$$\mathbf{y}_{centered} = \mathbf{y} - \bar{\mathbf{y}} \tag{12}$$

where $\bar{\mathbf{y}}$ is a vector where all the elements are the mean μ_y of the training targets. This centering ensures that the intercept term θ_0 represents the model's prediction when all standardized features equal zero.

A critical aspect of our scaling procedure is that all scaling parameters ($\boldsymbol{\mu}_X$, $\boldsymbol{\sigma}_X$, and μ_y) are computed exclusively from the training data and then applied to both training and test sets. This approach prevents data leakage, where information from the test set could inadvertently influence the model training process. Using test set statistics for scaling would allow the model to gain knowledge about the test distribution, leading to an overly optimistic and biased evaluation of model performance.

The reason we scale the data using z-score normalization is to avoid numerical instability problems when inverting the Hessian matrix. As discussed previously, we can encounter very small values in the design matrix because of the domain of the Runge function. This is countered by applying z-score normalization. The contribution of the standard deviation in equation 11 increases the column-values so that every column has standard deviation $\sigma = 1$. This is especially helpful for the higher degree values, as they are scaled to avoid numerical rounding errors.

D. Regression Analysis

1. *Ordinary Least Squares Regression*

For the OLS method, we seek to minimize the cost function

$$\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 = \frac{1}{n}\left(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\right)^T\left(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\right). \tag{13}$$

The gradient becomes

$$\nabla_{OLS}(\boldsymbol{\theta}) = \frac{2}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}\right), \tag{14}$$

which set to zero, gives the following expression for the optimal parameters:

$$\hat{\boldsymbol{\theta}}_{OLS} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}. \tag{15}$$

2. *Ridge Regression*

Ridge regression adds a regularization penalty term to the OLS cost function:

$$C_{\text{Ridge}}(\boldsymbol{\theta}) = \frac{1}{n}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_2^2, \tag{16}$$

where $\lambda > 0$ is a hyperparameter that controls the strength of regularization. This modification ensures that the matrix $\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I}$ is always invertible, even when $\mathbf{X}^T\mathbf{X}$ is singular or near singular.

The optimal parameters are obtained by minimizing the Ridge cost function. Taking the gradient with respect to $\boldsymbol{\theta}$ yields:

$$\nabla_{Ridge}(\boldsymbol{\theta}) = \frac{2}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}\right) + 2\lambda\boldsymbol{\theta}, \tag{17}$$

Setting this gradient to zero and rearranging:

$$\begin{aligned}\frac{2}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}\right) + 2\lambda\boldsymbol{\theta} &= \mathbf{0}\\ \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} + n\lambda\boldsymbol{\theta} &= \mathbf{X}^T\mathbf{y}\\ (\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})\boldsymbol{\theta} &= \mathbf{X}^T\mathbf{y}\end{aligned}$$

The optimal Ridge regression parameters are then given by:

$$\hat{\boldsymbol{\theta}}_{\text{Ridge}} = (\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}, \tag{18}$$

where $\mathbf{I}$ is the identity matrix. The addition of $n\lambda\mathbf{I}$ to the diagonal guarantees that all eigenvalues are at least $n\lambda > 0$, ensuring numerical stability and preventing overfitting by shrinking the parameter values.

3. Lasso Regression

Lasso regression changes the L_2 penalty in equation 16 with an L_1 penalty in the cost function, which yields:

$$C_{Lasso}(\boldsymbol{\theta}) = \frac{1}{n}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda\|\boldsymbol{\theta}\|_1. \tag{19}$$

The L_1 penalty provides more shrinkage than the ℓ_2 that Ridge regression uses, shrinking variables exactly to zero. This gives Lasso regression the characteristic of variable selection, as many of the variables completely vanish during the shrinkage process. Unlike Ridge, the L_1 term is not differentiable at 0, meaning we do not have an analytical solution, and we have to approximate it numerically. This leads the use of a subgradient when minimizing the cost function:

$$\nabla_{Lasso}(\boldsymbol{\theta}) = \frac{2}{n}\left(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}\right) + \lambda sgn(\boldsymbol{\theta}). \tag{20}$$

where $sgn(0) \in [-1, 1]$.

E. Gradient Descent

Since the Lasso cost function has no closed-form minimizer, we use Gradient Descent to approximate its minimizer. Consider minimizing an objective $F : \mathbb{R}^p \to \mathbb{R}$. A Gradient Descent method iteratively updates a parameter vector by moving in the direction of the steepest descent of F. Starting from an initial guess $\boldsymbol{\theta}_0$, we continue to compute new values

$$\theta_{k+1} = \theta_k - \eta_k \nabla F(\theta_k), \quad k \geq 0, \tag{21}$$

with $\eta_k > 0$ being the learning rate (step length). This procedure produces a sequence of approximations, where the sequence ideally converges to a minimizer of F.

Convergence is declared when relative MSE change falls below 10^{-6}. Divergence detection uses multiple criteria to distinguish catastrophic failure from temporary oscillations: immediate termination if MSE becomes NaN/infinite or exceeds $10\times$ the initial value, or if MSE increases consecutively over five iterations while exceeding $1.5\times$ the initial value. These criteria ensure that only persistent instability (typically from excessively large learning rates) triggers early termination, while normal oscillatory behavior is allowed to converge.

The method correlating to the update rule in equation 21 is known as Steepest Descent, and is the simplest and most traditional variant of Gradient Descent. While the scheme is conceptually simple, the method has several limitations. In machine learning applications, we usually work with complex functions which are not necessarily convex. The gradient descent scheme might get stuck in local minima "ravines" and fails to approximate the global minima. The method is also inherently limited by its sensitivity to the learning rate η_k.

1. Momentum

To address some limitations of steepest descent, we can introduce momentum-based gradient descent. This is a method that uses the momentum as a memory of the direction we are moving in parameter space, and introduces a velocity vector

$$\mathbf{v}_t = \gamma\mathbf{v}_{t-1} + \nabla_\theta E(\boldsymbol{\theta}_t),$$

where $0 \leq \gamma \leq 1$ is the momentum parameter and $E(\boldsymbol{\theta}_t)$ is the expected value of $\boldsymbol{\theta}_t$. The velocity vector is used for averaging recently encountered gradients. The update rule is then given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t\mathbf{v}_t, \tag{22}$$

where η_t is the learning rate.

This method helps to dampen oscillations and accelerates convergence along directions of the gradient. Instead of relying solely on the current gradient, the algorithm uses a combination of past gradients to build velocity, which can help it move more smoothly through narrow, curved valleys.

2. RMSprop

RMS prop keeps track of a moving average of the squared gradients. This allows the method to adaptively scale the learning rate for each parameter based on the historical magnitude of its gradient. We start by denoting $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$, and we continue to calculate

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}),$$

and

$$\mathbf{s}_t = \beta\mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2.$$

The update rule for RMS prop is then given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t\frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \tag{23}$$

where β is a parameter that controls the averaging time of the second moment, η is the learning rate, and ϵ is

a small regularization constant to prevent divergences. This method effectively reduces the learning rate for parameters with consistently large gradients, and increases it for those with smaller gradients. This helps stabilize training and often leads to faster convergence when working with stochastic and noisy gradients.

3. ADAM

Another momentum-based method for gradient descent is the ADAM optimizer. ADAM stands for Adaptive Moment Estimation and combines ideas from both momentum and RMSprop. It keeps track of an exponentially decaying average of past gradients (first moment) and an exponentially decaying average of past squared gradients (second moment). These are denoted as:

$$\mathbf{m}_t = \mathbb{E}[\mathbf{g}_t], \quad \mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2],$$

where $\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}_t)$ is the gradient of the cost function with respect to the parameters at time step t.

The moment estimates are updated as:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \quad \mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2,$$

where β_1 and β_2 are hyperparameters that control the rate of decay. Since these moment estimates will be initialized as zero vectors, they will have a bias towards zero. Therefore, ADAM performs a bias correction

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}.$$

The update rule for ADAM is then given by:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{24}$$

ADAM combines the advantages of Momentum and RMSprop, adapting learning rates for each parameter while smoothing updates through moving averages. It typically achieves fast, stable convergence with minimal tuning.

4. ADAGrad

Another adaptive method for gradient descent is the AdaGrad optimizer, short for Adaptive Gradient Algorithm. The main idea of AdaGrad is to adapt the learning rate for each parameter individually, based on the history of its gradients. Parameters that receive frequent updates will have their learning rate reduced, while parameters that are updated less often will retain a relatively larger learning rate. This makes AdaGrad particularly effective for problems with sparse data.

Formally, let

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}_t)$$

be the gradient of the cost function with respect to the parameters at time step t. AdaGrad keeps track of the accumulated squared gradients for each parameter:

$$\mathbf{r}_t = \mathbf{r}_{t-1} + \mathbf{g}_t \circ \mathbf{g}_t,$$

where $\circ$ denotes element-wise multiplication.

The update rule for AdaGrad is then given by:

$$\boldsymbol{\theta}_{t+1,i} = \boldsymbol{\theta}_{t,i} - \frac{\eta}{\sqrt{r_{t,i}} + \epsilon} g_{t,i}, \tag{25}$$

where η is the initial learning rate and ϵ is a small constant added for numerical stability.

While AdaGrad works well for sparse problems and eliminates the need to manually tune per-parameter learning rates, a drawback is that the accumulated squared gradients $\mathbf{r}_t$ grow without bound, causing the effective learning rates to shrink monotonically. This can eventually make the algorithm stop progressing.

5. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is one of the most central optimization algorithms in machine learning. Unlike standard gradient descent, which computes the gradient of the cost function using the entire dataset, SGD approximates the gradient by using only a single randomly chosen sample, or a small batch of samples, at each update step. This drastically reduces the computational cost per iteration and makes training feasible on large datasets. The randomness also injects noise into the updates, which can help the algorithm escape shallow local minima or saddle points.

The starting point is that the cost function can be expressed as a sum over n data points:

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} c_i(\mathbf{x}_i, \boldsymbol{\theta}),$$

so that the gradient can be written as

$$\nabla_\theta C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta c_i(\mathbf{x}_i, \boldsymbol{\theta}).$$

Instead of summing over all n terms, SGD estimates this gradient using a mini-batch $B_k \subset \{1, \ldots, n\}$ of size M. The update rule is then given by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{1}{M}\sum_{i \in B_k} \nabla_\theta c_i(\mathbf{x}_i, \boldsymbol{\theta}_t), \tag{26}$$

where η_t is the learning rate at iteration t, and B_k is sampled uniformly at random.

In our implementation, we sample mini-batches **without replacement** within each epoch. The dataset is shuffled once at the start of each epoch, then divided sequentially into batches. This ensures each sample is used exactly once per epoch, reducing variance compared to sampling with replacement.

Convergence and divergence detection follow the same criteria as standard gradient descent (Section II E), but MSE is evaluated after each epoch rather than after each iteration. While SGD greatly improves efficiency, the noisy updates may lead to unstable convergence.

F. Resampling

Resampling methods are an important aspect of machine learning. By resampling, we can retrain our model on shuffled data and gain access to new estimations that can better our model performance and provide more reliable measures of its ability to generalize.

1. Bootstrap

Bootstrapping is a statistical resampling method that provides deeper insight into the variability and reliability of data-driven estimates. The idea of bootstrapping is simply to create new bootstrap datasets by sampling with replacement from our original dataset. These bootstrap samples are then used to recalculate the statistic of interest (such as the mean, median, or regression coefficients). By repeating this procedure many times, we can create a distribution of the calculations and compare it against the statistic of interest. Normally, as in this project, bootstrap is used as an estimator for the prediction error for a fixed training set. This is done by resampling a subset of the training data and evaluating it on the unseen test data.

2. Cross-validation

Cross validation is another resampling technique we use to estimate how good our model fits the data. The idea of cross validation is to split our dataset into a number of folds. In each round, one fold will be used for test data, and the remaining folds will be used for training data. We will record how accurate our model's prediction is against the test data. The model is trained and evaluated on these splits repeatedly, rotating the role of the test fold each time.

By averaging performance over all folds, cross validation gives us a more reliable estimate of how well our model generalizes to new data. By running a k-fold cross validation technique on multiple machine learning methods, we will have obtained a good indication of which model to select, based on their ability to generalize well to unseen data.

3. Bias-Variance Tradeoff

In model selection, we are interested in knowing which model has a better fit to unseen data. Bias-Variance tradeoff is a central concept in how we would select the best model and why it has the best fit. Ideally, we would like a model with both low bias and low variance. In practice, however, this is rarely achievable. On one hand, simpler models are prone to have a high bias, and on the other hand, more complex models are prone to have a higher variance. The key to model selection is to find a good balance between bias and variance, as you would want to have a model that is complex enough to fit your data, but also simple enough to not overfit it.

4. Bias Variance Decomposition

We assume that the expectation values are taken over different realizations of the training data (e.g., through bootstrap sampling). For a fixed x, we study how the model prediction $\tilde{y}$ varies when trained on different datasets. Starting from the mean squared error, the expected prediction error can be decomposed into bias, variance, and an irreducible noise term, where the noise is governed by a Gaussian distribution.

$$E[(y-\tilde{y})^2] = \mathrm{Var}[\tilde{y}] + \mathrm{Bias}_f^2[\tilde{y}] + \sigma^2. \tag{27}$$

Here, $\mathrm{Var}[\tilde{y}]$ quantifies how sensitive the model is to variations in the training data, $\mathrm{Bias}_f^2[\tilde{y}]$ measures the systematic error between the expected model prediction and the true function $f(x)$, and σ^2 represents the irreducible noise inherent in the data. A full step-by-step derivation of this result, including all intermediate algebraic expansions and expectation operations, is provided in Appendix A 1.

G. Use of AI Tools

To enhance the quality of this scientific report, we utilized several AI tools throughout the writing and development process. Claude and ChatGPT were used to assist with formulating precise scientific language and improving sentence structure to meet academic writing standards. For code development, we used Claude, ChatGPT, and GitHub Copilot to support debugging processes, optimize code structure, and ensure consistency across our implementations. Additionally, these tools helped generate figures of good quality that maintain visual consistency throughout the document.

H. Tools

For code collaboration, we have used GitHub, and our code can be found by following this link [2]. Our code is

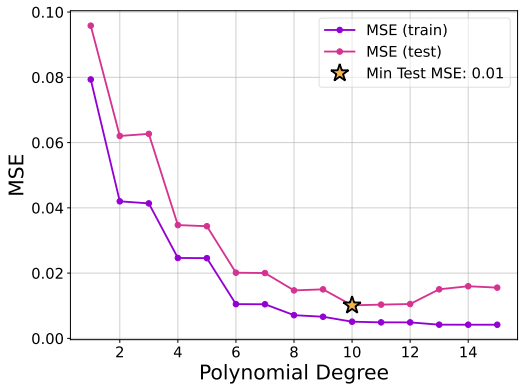

Figure 1. MSE of OLS training and test data for polynomial degree $p = 1, \ldots, 15$ using 50 datapoints.

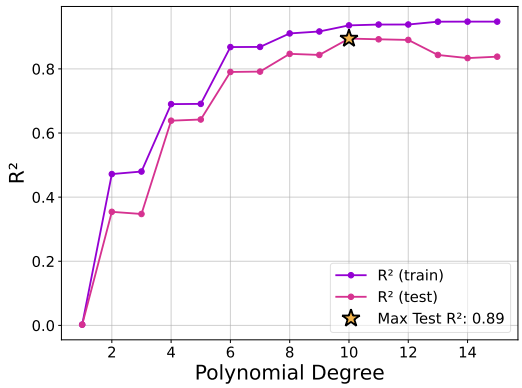

Figure 2. R^2 score of OLS training and test data for polynomial degree $p = 1, \ldots, 15$ using 50 datapoints.

written in Python, with libraries: matplotlib [3], numpy [4], scikit-learn [5], seaborn [6] and pandas [7].

III. RESULTS

In this section, we will present how the regression models OLS, Ridge and Lasso, as well as the gradient descent methods, perform when approximating the Runge function with noise. We observe the results for different ranges of polynomial degrees and sample size. The noise added to the Runge function follows a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.1$. The data has been split into 3/4 training data and 1/4 test data. The scaling is done using z-score.

A. Ordinary Least Squares

We have implemented the OLS regression method for approximating the Runge function for polynomials up to degree 15, with data consisting of 34 to 1000 datapoints. In figure 1 we have plotted the MSE as a function of polynomial degree for 50 datapoints. The red curve shows the calculated MSE for the test data, while the blue curve shows the MSE for the training data. In figure 2 we have plotted the R2 score for the models. The colors represent the same as in figure 1. Figure 3 shows the test MSE of the model as a function of polynomial degree and sample size. The heatmap shows the value of the MSE, where yellow represents high MSE values and blue represents low MSE values. The spread of values of the optimal parameters $\hat{\boldsymbol{\theta}}$ are shown in figure 4 as a function of polynomial degree.

B. Ridge

Alongside the OLS method, we implemented Ridge regression to approximate the Runge function. Since Ridge

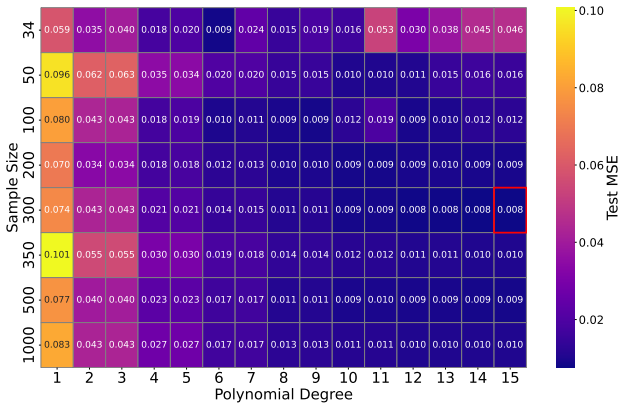

Figure 3. Test MSE of the OLS model as a function of polynomial degree and sample size. The red outline contains the lowest MSE.

regression introduces a penalty term that shrinks the regression coefficients, we extended the analysis to higher polynomial degrees to better illustrate its stability compared to OLS. In figure 5, we present the MSE for polynomial degrees up to 35, using 50 data points and a regularization parameter of $\lambda = 0.01$. This highlights when overfitting begins to appear and demonstrates how Ridge mitigates this effect. Similarly, figure 6 shows the corresponding R^2 scores for the same setup. Figure 7 illustrates the evolution of the regression coefficients $\boldsymbol{\theta}$ as a function of the polynomial degree in the Ridge method. Lastly, figure 8 presents the relationship between polynomial degree, from 1 to 15, regularization parameter λ from $log_{10}(-5)$ to $\log_{10}(2)$, sample size from 0 to 800 and shows the MSE for all the points in a 3D graph.

C. Gradient Descent

We evaluated five gradient descent optimizers (plain GD, Momentum, AdaGrad, RMSprop, and Adam) on OLS and Ridge regression ($\lambda = 0.01$). For each opti-

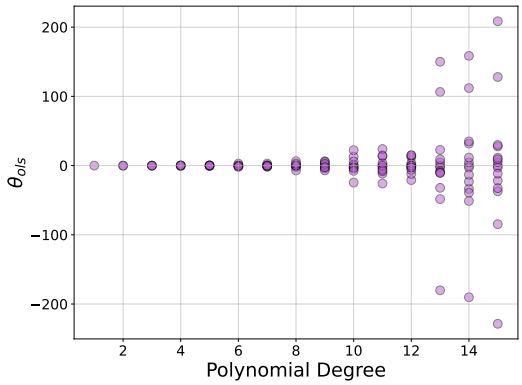

Figure 4. Spread of the values of the optimal parameters $\hat{\boldsymbol{\theta}}$ of the OLS model as a function for polynomial degree $p = 1, ..., 15$.

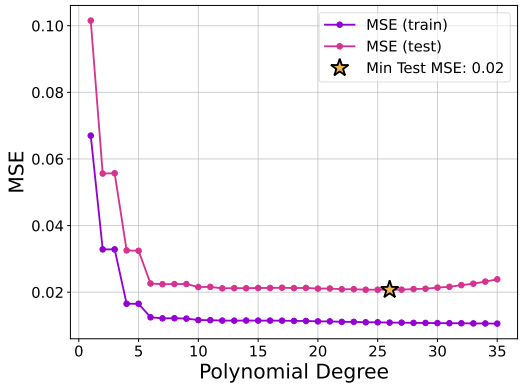

Figure 5. MSE of Ridge training and test data for polynomial degree $p = 1, \ldots, 35$ using 50 datapoints with $\lambda = 0.01$.

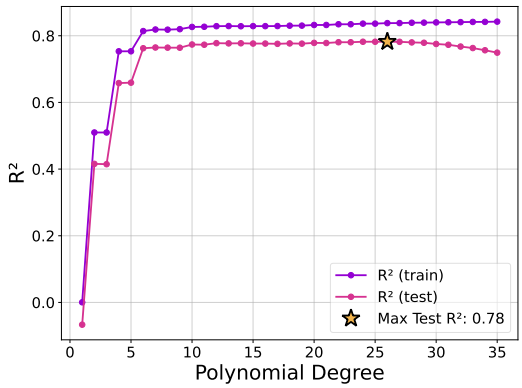

Figure 6. R^2 score of Ridge training and test data for polynomial degree $p = 1, \ldots, 35$ using 50 datapoints with $\lambda = 0.01$.

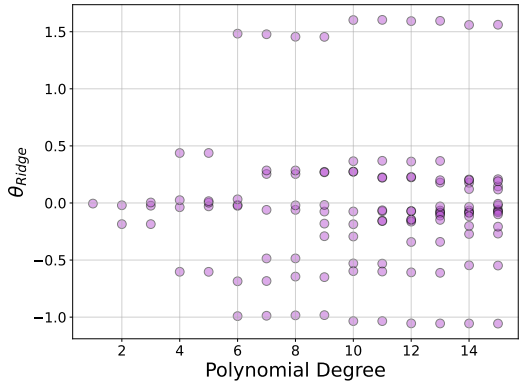

Figure 7. Spread of the values of the optimal parameters $\hat{\boldsymbol{\theta}}$ of the Ridge model as a function for polynomial degree $p = 1, ..., 15$.

mizer, we tested multiple learning rates to identify values achieving convergence within 1500 iterations.

Figure 9 illustrates the learning rate sensitivity of steepest descent. Both OLS and Ridge diverged at $\eta = 0.37$ but converged at $\eta = 0.365$, demonstrating the critical importance of proper learning rate selection. Ridge consistently required more iterations than OLS (88 vs 93 iterations at $\eta = 0.365$), reflecting the additional computational cost of regularization.

Table I presents learning rate sensitivity for OLS across all optimizers. Momentum achieved fastest convergence (71 iterations), while RMSprop required the smallest learning rates and showed highly variable iteration counts. All converged methods achieved similar final MSE values (≈ 0.026).

Table II shows corresponding Ridge regression results. Compared to OLS, Ridge required more iterations across optimizers and exhibited increased sensitivity—notably, RMSprop failed to converge for $\eta \geq 0.01$. Final MSE values (≈ 0.027) were consistently higher than OLS due to the regularization penalty.

Table III presents optimal learning rates for Lasso regression. Momentum converged fastest (70 iterations),

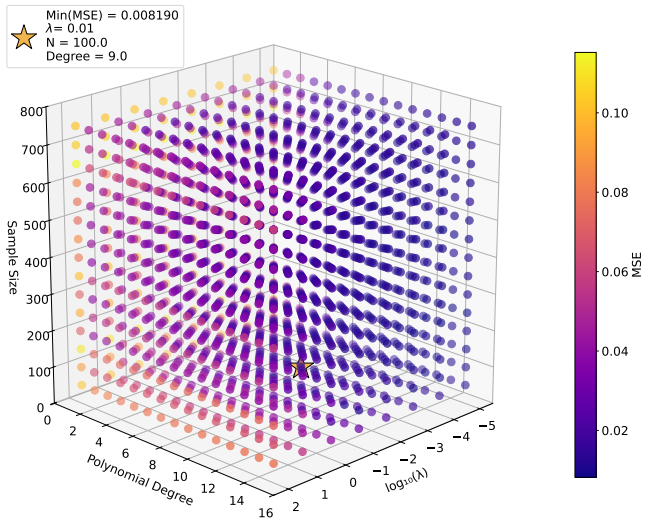

Figure 8. Sample sizes 50 to 750, degrees $1 - 15$, λ from $10^{-5} - 10^{2}$. Showing optimal MSE at degree 9, sample size 100 and $\lambda = 10^{-2}$.

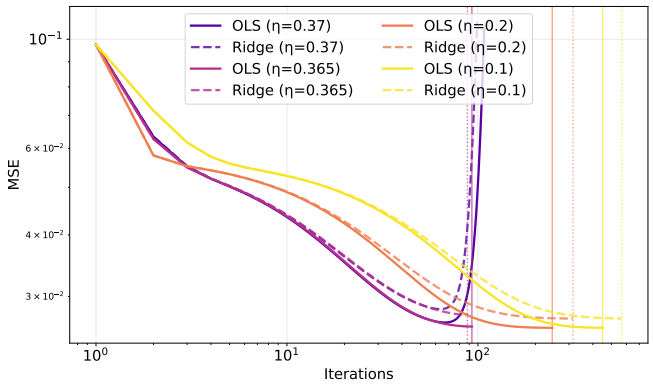

Figure 9. Gradient descent convergence for OLS (solid) and Ridge (dashed, $\lambda = 0.01$) on the Runge function ($N = 500$, $p = 5$) with $\eta \in \{0.1, 0.2, 0.365, 0.37\}$. Vertical lines indicate convergence. Log-log scale.

while Lasso generally tolerated higher learning rates than Ridge (e.g., plain GD: $\eta = 0.5$ vs $\eta = 0.365$). Final MSE values ranged from 0.026 to 0.027.

Table I. OLS regression learning rate sensitivity ($N = 500$, $p = 5$) for five gradient-based optimizers.

Method	η	Iterations	Final MSE	Converged
GD	0.370	111	–	Diverged
GD	0.365	93	0.026051	Yes
GD	0.200	245	0.025893	Yes
GD	0.100	450	0.025901	Yes
Momentum	0.400	115	0.025886	Yes
Momentum	0.300	93	0.025887	Yes
Momentum	0.100	71	0.025890	Yes
Momentum	0.050	95	0.025890	Yes
AdaGrad	0.300	163	0.025890	Yes
AdaGrad	0.250	161	0.025890	Yes
AdaGrad	0.150	170	0.025890	Yes
AdaGrad	0.100	232	0.025893	Yes
RMSProp	0.020	136	0.027134	Yes
RMSProp	0.010	112	0.026289	Yes
RMSProp	0.005	185	0.025995	Yes
RMSProp	0.001	757	0.025892	Yes
Adam	0.200	99	0.025889	Yes
Adam	0.100	92	0.025896	Yes
Adam	0.050	86	0.025900	Yes
Adam	0.020	181	0.025888	Yes

D. Lasso

We first evaluated five gradient descent optimizers for Lasso regression with $\lambda = 0.01$ on a polynomial of degree $d = 5$, testing multiple learning rates to achieve convergence within 1500 iterations. Table III presents these results, demonstrating that gradient descent methods can successfully solve the non-differentiable Lasso optimization problem. Momentum and Adam achieved fastest convergence (70-84 iterations).

Table II. Ridge regression learning rate sensitivity ($N = 500$, $p = 5$, $\lambda = 0.01$) for five gradient-based optimizers.

Method	η	Iterations	Final MSE	Converged
GD	0.370	102	–	Diverged
GD	0.365	88	0.027516	Yes
GD	0.200	315	0.027038	Yes
GD	0.100	566	0.027051	Yes
Momentum	0.220	110	0.026993	Yes
Momentum	0.200	66	0.026804	Yes
Momentum	0.150	131	0.027018	Yes
Momentum	0.050	122	0.027043	Yes
AdaGrad	0.225	208	0.027033	Yes
AdaGrad	0.220	207	0.027033	Yes
AdaGrad	0.215	207	0.027033	Yes
AdaGrad	0.150	218	0.027034	Yes
RMSProp	0.010	1500	0.027355	No
RMSProp	0.005	162	0.027365	Yes
RMSProp	0.002	346	0.027159	Yes
RMSProp	0.001	648	0.027090	Yes
Adam	0.250	112	0.027059	Yes
Adam	0.200	98	0.027021	Yes
Adam	0.100	135	0.027034	Yes
Adam	0.050	119	0.027057	Yes

For the comparison across polynomial degrees shown in Figure 10, we used analytical solutions for OLS and Ridge, and scikit-learn's coordinate descent implementation for Lasso to ensure robust convergence across all complexity levels.

Table III. Optimal learning rates and convergence behavior for Lasso regression ($N = 500$, $p = 5$, $\lambda = 0.01$) using five gradient-based optimizers. All methods successfully converged despite the non-differentiable L_1 penalty.

Method	η	Iterations	Final MSE
GD	0.500	171	0.027131
Momentum	0.050	70	0.025916
AdaGrad	0.200	231	0.026420
RMSprop	0.010	123	0.026228
Adam	0.100	84	0.025908

E. Stochastic Gradient Descent

In table IV we can see differences in calculation time, epochs/iterations, and MSE of Full- and Mini-Batch SGD for all the methods. We can see that the momentum method is both the fastest and the method that requires the least amount of epochs/iterations for both Full- and Mini-Batch. Outside this, there is more variation between the methods and batch type, especially for RMSprop and Adam. The final MSE of every method goes toward a similar value $MSE \simeq 0.027$.

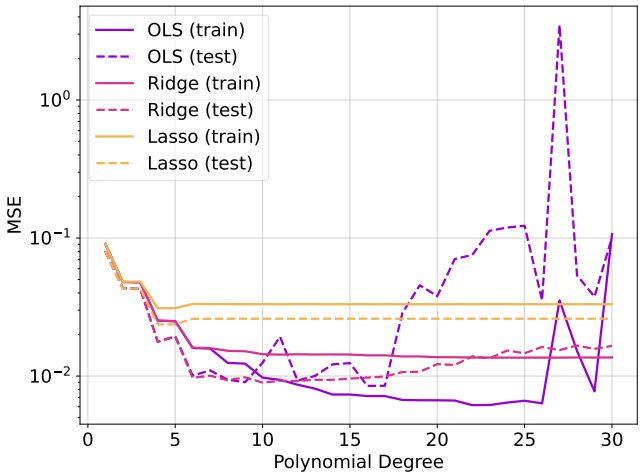

Figure 10. Comparison of training and test MSE for OLS, Ridge, and Lasso across polynomial degrees $p = 1, \ldots, 30$ using 100 datapoints. OLS and Ridge use analytical solutions; Lasso uses scikit-learn's coordinate descent.

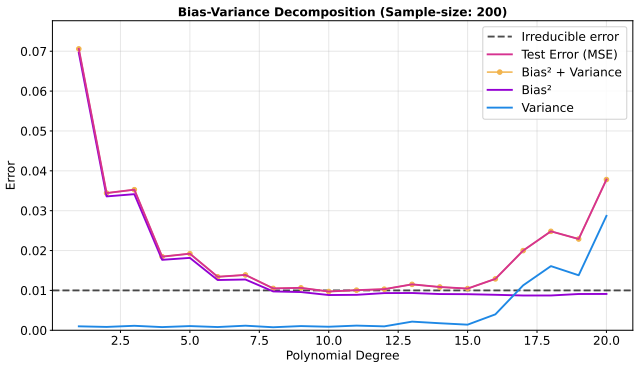

Figure 11. Bias-variance decomposition for OLS regression on Runge's function with $n = 200$ data points and $\sigma = 0.1$. The blue curve shows the averaged variance, purple shows the averaged bias-squared, and yellow shows the sum of variance and bias-squared (the theoretical expected error/MSE). The pink curve is the averaged test error/MSE. The black dashed line indicates the irreducible error ($\sigma^2 = 0.01$).

Table IV. Convergence comparison of Full-Batch and Mini-Batch SGD for OLS (N = 1,000,000, $p = 5$, batch size 256, stochastic methods averaged over 30 runs).

Method	Batch Type	Time (s)	Epochs/Iters	Final MSE
GD	Full-Batch	0.47	129	0.026848
Momentum	Full-Batch	0.25	71	0.026827
AdaGrad	Full-Batch	0.60	163	0.026829
RMSprop	Full-Batch	0.44	111	0.027161
Adam	Full-Batch	0.37	92	0.026835
SGD	Mini-Batch	0.55	15	0.026826
Momentum	Mini-Batch	0.64	16	0.026828
AdaGrad	Mini-Batch	0.49	12	0.026825
RMSprop	Mini-Batch	1.30	30	0.026835
Adam	Mini-Batch	0.79	17	0.026826

F. Bootstrapping

We used the bootstrapping resampling technique described in the theory to analyze the bias-variance decomposition for models with varying complexity; varying sample sizes n and polynomial degrees. In our analysis we used 2000 bootstrap samples. In 11 we present the bias variance decomposition for OLS using bootstrap, while in 12 we present a bootstrap bias-variance analysis for different sample sizes and model complexities. Finally, in 13, we represent different model predictions for different combinations of sample sizes and polynomial degree.

G. Cross-Validation

We performed 5-fold cross-validation resampling to compare the estimated model generalization error with the bootstrap estimated error. In bootstrap we split the data in 0.25/0.75 test/train samples. Then we resampled 2000 times to train our OLS model and evaluate it on the fixed 0.25 test data. We then used 5-fold Cross-Validation as an estimator, where we trained our models on the full data-set and reported the average MSE with standard-deviation across the 5-folds. The comparison between the two estimators is presented in Figure 14

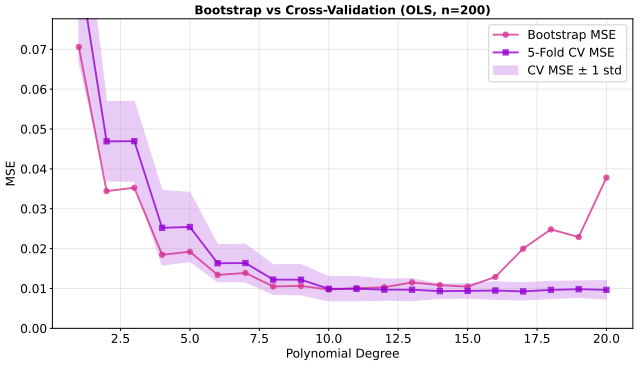

Figure 14. Bootstrap vs. cross-validation MSE comparison for OLS ($n = 200$, $\sigma = 0.1$). Pink: bootstrap test error (2000 resamples), purple: 5-fold CV validation error with ± 1 standard deviation (shaded).

We then leveraged the CV method to compare the effect of regularization in Ridge and Lasso against OLS. We wanted to analyze the effectiveness of regularization

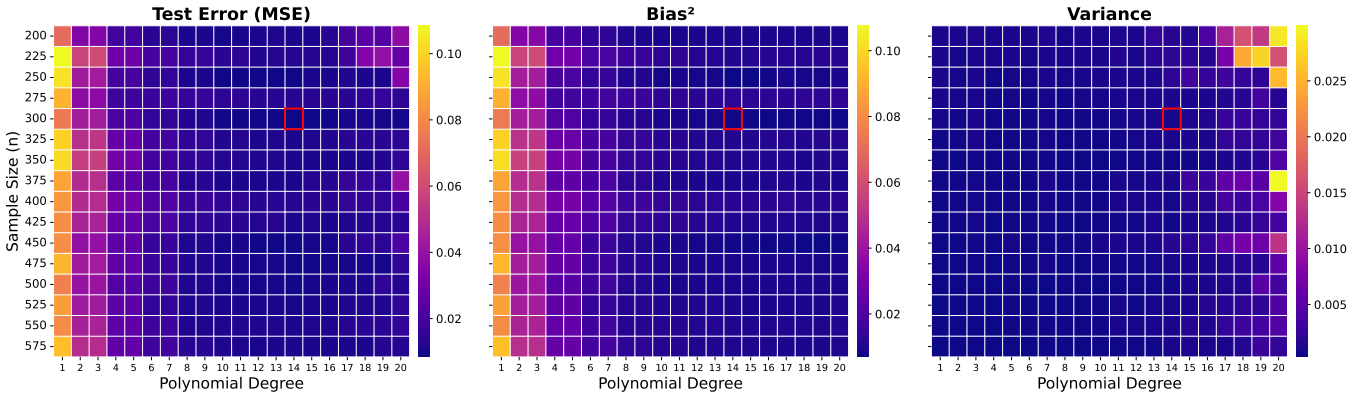

Figure 12. Bootstrap bias-variance analysis across model complexities. Left panel shows MSE, middle panel shows bias^2, and the right panel shows variance as functions of sample size n (y-axis) and polynomial degree (x-axis). The red boxes mark the model complexity ($n = 300$, degree $= 14$) that resulted in the minimum MSE. The coloring of the cells represents the magnitude of the MSE, bias^2, and variance for each combination of model complexity.

models (Ridge and Lasso) in the full parameter space we used for bootstrapping, but extended for thoroughness. We performed 5-fold cross-validation for OLS, Ridge (best λ), and Lasso (best λ) across three noise levels ($\sigma \in \{0.1, 0.2, 0.3\}$). We knew that the models performance could be affected by the random train-test split. Therefore, we averaged the MSE across 10 random seeds. For 20 lambdas in the range $\lambda \in (-5, 2)$ in logspace. After around 2 million models and more than one hour of computation, we selected the best λ's independently for each (n, degree). We then computed the relative MSE difference in percent between OLS and Ridge, and OLS and Lasso, respectively in this parameter space. The results are presented in Figure 15.

IV. DISCUSSION

A. Ordinary Least Squares Performance

We will discuss figures 1 and 2. We can see that as we increase the complexity, the MSE decreases for the training data. This is expected as the higher complexity models will be good at approximating the training data. The MSE of the test data also follows this trend for the lower degree polynomials. The test MSE is generally higher than the training MSE. This is consistent with the idea that the test data provides an independent measure of the model.

For the lowest polynomial degrees, we can see that the MSE is quite high, which corresponds to underfitting (not capturing the detail of the data). We can also see that the test MSE increases again after polynomial degree ten. This is overfitting. As described in section II B, this represents the fact the model, trained on the training data, is getting too detailed. This entails that the model deviates from the underlying pattern because the model starts memorizing the noise. We can see the same trend in figure 2 for the R2 score of the model for 50 datapoints. The test data's R2 score decreases after degree 10, while the test data's R2 score goes towards one.

In figure 3 we can see that the MSE is not necessarily better for higher sample size. This contradicts theory in the sense that higher sample size should "smooth out" the noise, making it less influential. This is, however, caused by the randomness of the noisy data. See section IV G for more detailed explanation. In essence, what happens is that we hit a plateau where the MSE will be around the variance $\sigma^2 = 0.01$ introduced in the Gaussian noise. The reason the MSE is lower than this at some points comes from that the random data gives us a "lucky" dataset for these parameters.

B. Ridge Performance Analysis

Alongside the OLS regression method, we have also implemented Ridge regression to highlight the stabilizing effect of regularization. As shown in Figure 7, the penalty term introduced by Ridge shrinks the regression coefficients compared to the OLS model as seen in figure 4. This shrinkage reduces sensitivity to oscillations, particularly near the boundaries, and effectively mitigates Runge's phenomenon observed in the OLS case. By discouraging large coefficient values, Ridge regression produces a smoother and more stable approximation.

From a performance perspective, this regularization introduces a trade-off. Although Ridge slightly reduces the model's flexibility, it often improves generalization to unseen data. Compared to OLS, Ridge yields lower variance in prediction accuracy and maintains stability across a wider range of polynomial degrees.

This is evident in the comparison between Figures 1 and 5. The OLS model begins to show instability already around polynomial degree 12, whereas Ridge remains stable up to around degree 30—a clear advantage of the

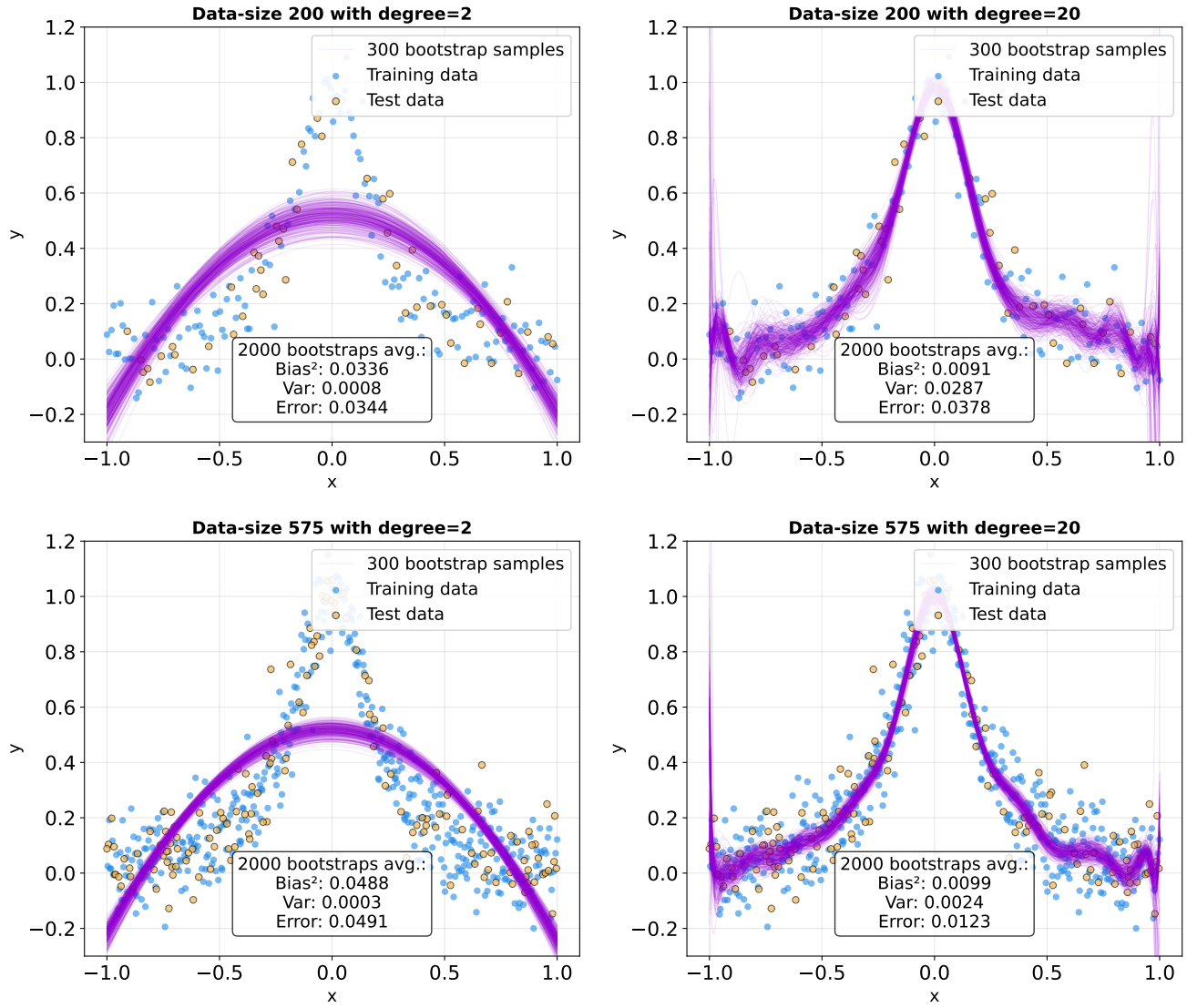

Figure 13. Model variance for individual bootstrap samples. Each subplot shows 300 bootstrap model predictions (purple, semi-transparent) for different combinations of sample sizes and polynomial degree. The bootstrap model used a train-test split of 25% test samples (yellow dots) and 75% training samples (blue dots).

penalty term. Similarly, in Figures 2 and 6, the R^2-score of Ridge demonstrates greater stability at higher polynomial degrees, reinforcing the benefits of regularization.

C. Gradient Descent Performance Analysis

To validate our gradient descent implementation and understand its behavior, we examined convergence patterns for OLS and Ridge regression across different learning rates: $\eta \in \{0.37, 0.365, 0.2, 0.05\}$. With properly tuned learning rates, gradient descent successfully reproduces the analytical solutions from Parts a) and b) within numerical precision. For instance, at $\eta = 0.365$, OLS converged to MSE = 0.026051 in 93 iterations, matching the analytical result (slightly higher) presented in figure 3 for $N = 500$ and degree 5. While the analytical approach provides instantaneous solutions through direct matrix inversion, gradient descent requires iterative optimization but offers crucial flexibility for problems where closed-form solutions don't exist, as presented in II D 3, and will be discussed further in IV E.

Figure 9 demonstrates the critical importance of learning rate selection. At $\eta = 0.37$, both methods diverge, but reducing η by merely 0.005 to 0.365 achieves stable convergence. Ridge converged in 88 iterations (MSE = 0.027516) and OLS converged in 93 iterations (MSE = 0.026051). This sensitivity underscores that gradient

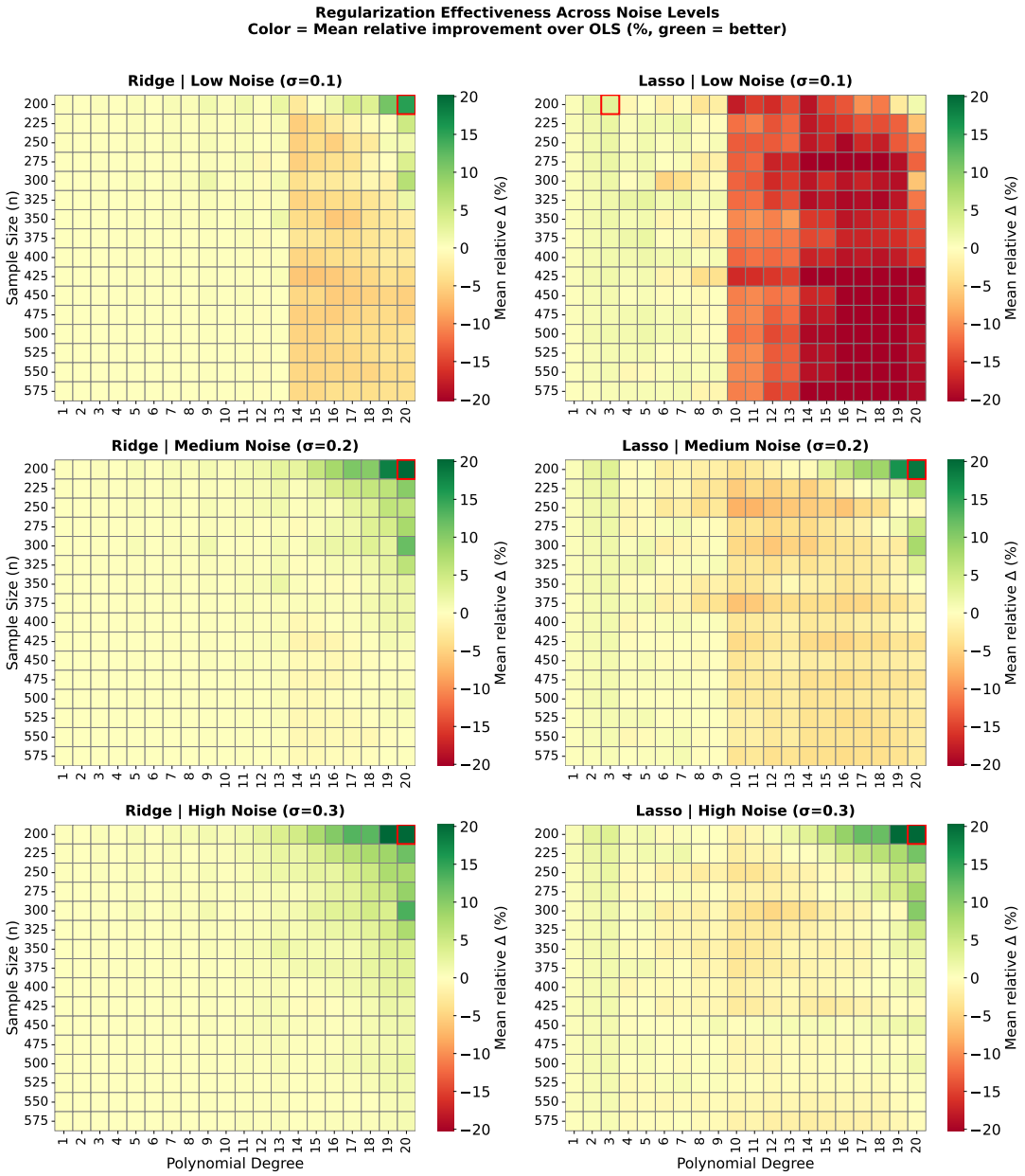

Figure 15. Relative improvement of regularization over OLS across noise levels. Each cell shows the mean relative improvement $\Delta_{\text{rel}} = (\text{MSE}_{\text{OLS}} - \text{MSE}_{\text{model}})/\text{MSE}_{\text{OLS}}$ (in percent) averaged over 10 random seeds. Green indicates regularization improves over OLS, red indicates degradation. Left column shows Ridge (best λ), right column shows Lasso (best λ). Red outlines mark the maximum improvement for each configuration.

descent requires careful tuning: learning rates that are too large cause divergence, while excessively small values lead to prohibitively slow convergence.

Comparing OLS and Ridge, we observe that Ridge consistently requires slightly more iterations to converge. This is expected behavior stemming from the regularization term $\lambda\|\boldsymbol{\theta}\|_2^2$ in equation 16, which creates a "restoring force" that simultaneously fits the data while shrinking parameters (as discussed in IV B). However, this iteration penalty is offset by Ridge's improved numerical stability: the regularized matrix $\mathbf{X}^T\mathbf{X} + n\lambda\mathbf{I}$ (equation 18) is better conditioned than $\mathbf{X}^T\mathbf{X}$ (equation 15), enabling Ridge to use larger learning rates than OLS. This advantage should become more pronounced as model complexity increases and OLS's conditioning deteriorates.

Testing this hypothesis, we repeated the analysis at higher polynomial degrees (10 and 15). As model complexity increased, OLS gradient descent became increasingly unstable, requiring progressively smaller learning rates and more iterations to converge. Ridge regression, by contrast, maintained efficient convergence when λ was scaled appropriately with model complexity—using

learning rates an order of magnitude larger than OLS while achieving convergence in comparable or fewer iterations. This confirms that Ridge's regularization provides both statistical benefits (preventing overfitting) and computational advantages (enabling efficient gradient-based optimization for ill-conditioned problems).

D. Gradient Descent Optimizers

Having established that plain gradient descent requires careful learning rate tuning, we now evaluate four adaptive optimization methods: Momentum, AdaGrad, RMSprop, and Adam. Tables I and II present comprehensive learning rate sweeps for each optimizer on both OLS and Ridge regression.

Momentum achieved the fastest convergence (71 iterations for OLS, 66 for Ridge), validating its theoretical advantage described in II E 1. Adam demonstrated the second-best performance (86–99 iterations for OLS, 98–135 for Ridge) with notable stability across learning rates. AdaGrad was consistently slowest (161–232 iterations for OLS, 207–218 for Ridge), likely due to the density of our problem. RMSprop showed irregular behaviour: although theory predicts fast and stable convergence (II E 2), it converged well for OLS at $\eta = 0.01$ but failed entirely for Ridge. This difference likely arises because Ridge increases gradient magnitudes via its regularization term, and in combination with RMSprop's adaptive scaling in a smooth, deterministic, full-batch setting, the effective step size may become too large, leading to divergence. The small dataset size may further amplify this sensitivity.

E. Performance analysis of Lasso

Table III demonstrates that gradient descent methods can optimize Lasso effectively despite its non-differentiable penalty. Momentum-based methods (Momentum, Adam) converged fastest, while all optimizers achieved similar final MSE values ($0.0259 - 0.0271$), confirming robust convergence to the same solution. Interestingly, the optimal learning rates for Lasso are notably higher than those for Ridge (e.g., plain GD: $\eta = 0.5$ vs. $\eta = 0.365$). This is likely due to Lasso's constant-magnitude penalty gradient $\lambda \cdot \text{sgn}(\theta)$, whereas Ridge's penalty gradient ($2\lambda\theta$) grows unboundedly with parameter magnitude, making it more sensitive to large learning rates.

Figure 10 reveals Lasso's key advantage: stability at high polynomial degrees. While OLS and Ridge severely overfit with test MSE exploding while training MSE remains low, Lasso maintains low MSE for both training and test data throughout. Although Lasso's training MSE is slightly higher than Ridge's, it generalizes far better to unseen data.

This stability stems from Lasso's feature selection capability. By setting irrelevant coefficients to exactly zero rather than shrinking them as Ridge does, Lasso eliminates high-degree terms that overfit to noise. Ridge keeps all terms active with small weights, allowing unnecessary high-degree features to degrade performance at high complexity. For the smooth Runge function, Lasso's automatic variable selection provides more robust regularization than Ridge's uniform shrinkage.

F. Stochastic Gradient Descent Performance Analysis

Table IV compares full-batch and mini-batch gradient descent for OLS regression ($N = 1{,}000{,}000$). Ridge and Lasso exhibit comparable patterns with appropriately adjusted learning rates. Mini-batch SGD converges in significantly fewer epochs (12–30) compared to full-batch methods (71–163 iterations). With a batch size of 256 and 750,000 training samples, each epoch performs approximately 2,930 gradient updates, whereas full-batch methods perform a single update per iteration. SGD, Momentum, AdaGrad, and Adam converge within 15–17 epochs, while RMSprop remains the worst performer, requiring 30 epochs, consistent with observations in IV D. Despite the reduced number of epochs, mini-batch methods do not achieve faster wall-clock times for this dataset. The overhead associated with data shuffling, batch extraction, and per-parameter learning rate updates across thousands of mini-batches dominates the computation. Gradient evaluations for these smooth, convex functions are relatively inexpensive, so the overhead negates the gains from fewer iterations. Comparing results with and without stochastic updates, mini-batch methods achieve similar final MSE values to full-batch methods. RMSprop shows slightly improved final MSE for mini-batch compared to full-batch, whereas SGD, Momentum, AdaGrad, and Adam perform similarly in both settings. Overall, full-batch methods remain highly efficient for smooth, convex problems, such as this, providing reliable convergence with minimal tuning. Mini-batch methods add flexibility and robustness, particularly for very large datasets or non-convex problems, but may introduce unnecessary overhead for moderate-sized datasets. In this experiment, all optimizers achieve comparable convergence speed and accuracy, indicating that the choice of optimizer is less critical for this problem.

G. Bootstrapping

Figure 11 illustrates the Bias-Variance tradeoff for an OLS model with 200 data samples. Using the bootstrap resampling method with 2000 resamples we calculated the averaged Bias^2, variance and MSE across the 2000 resamples. From the figure we see that as the complexity of the model increase (polynomial degree increase)

the bias of the model decreases while the variance increases. Specifically we see that for polynomial degrees in the range $p \in (1, 7)$ the system is dominated by Bias^2, then there is a stable zone for both bias and variance $p \in (7, 15)$, before variance begins to increase rapidly in the region $p \in (15, 20)$. In ML-terms, this translates to: our model is underfitting in $p \in (1, 7)$ and overfitting in $p \in (15, 20)$. Our model is too simple to capture the underlying pattern (Runge's function) in $p \in (1, 7)$, and our model is too flexible in $p \in (15, 20)$, so it starts to capture the random noise in the data. This is teh same effect as we saw in figure 1 and discussed in section III A (this phenomenon is also shown in figure 2.11 in Hastie et al [8]).

Furthermore, we see that the measured MSE curve closely follows the curve of $\text{Var}[\tilde{y}] + \text{Bias}^2[\tilde{y}]$, and the minimum of this curve is approximately the irreducible error. This is expected. In the theory in Appendix A 1, we showed that the expected error could be decomposed as $E[MSE] = \text{Var}[\tilde{y}] + \text{Bias}^2[\tilde{y}]$. We also found that the minimum MSE the system could obtain is the irreducible error σ^2. Our system has noise $\sigma = 0.1$, so the irreducible error $\sigma^2 = 0.01$. Our results seem to agree with theory.

We expanded the parameter space to perform the same bootstrap-analysis for polynomial degrees $p \in (1, 20)$ and sample-sizes $n \in (200, 575)$. In Figure 12 the heatmaps show how MSE, Bias^2 and variance varies across this parameter space. The results reveal that the best OLS model complexity, across the entire space is at $n = 300$ with degree $= 14$ (MSE ≈ 0.008). This is the sweet-spot where bias^2 (0.0076) and variance (0.0008) added together give the lowest error. Since bias dominates this point, we can determine that the model is still slightly underfitting even at degree 14. Now we notice something unexpected: the lowest MSE we found is lower than the irreducible error. The reason we find a result slightly lower than this limit is probably an artifact of the random train-test-splits in the data split. In our bootstrap method we left 25% of the data out for evaluation of the sampled models. This gives $300 \cdot 0.25 = 75$ for testing. This random 75 point subset of the data might not accurately represent the statistical properties of the full Gaussian noise-distribution $\epsilon \sim N(0, \sigma^2)$.

In the figure we also notice another interesting pattern. We see that the MSE, bias and variance follow the same pattern as in Figure 11: bias dominates in regions with low polynomial degree, and variance dominates in regions with high polynomial degree. The second dimension in the parameter space (sample-sizes) only shifts these regions. Lower sample-sizes makes variance dominate at lower polynomial degrees, and large sample-sizes pushes this to higher polynomial degrees. The Bias seems to be quite unaffected by the sample-sizes. This behavior is expected. Variance is an effect of noise: more data averages out this effect. Bias also slightly decrease with larger sample-sizes. The effect is probably less visible, since Bias measures how well the model can capture the underlying pattern (Runge's function). This is determined by both the model parameters (which is better estimated with more data), but most importantly, by the chosen basis for the model itself. Therefore the bias does not change that much with different sample-sizes.

In Figure 13 we visually see the statistical properties we have just discussed. It shows how choosing a too simple model $p = 2$ fundamentally limits how well the model can fit the data. It also illustrates how the variance in the models performance is reduced by choosing a higher sample size $n = 575$.

H. Cross Validation

In the figure 14 we see that CV-estimator follows the Bootstrap-prediction closely (largely within $\pm 1std$) in the region $p \in (1, 15)$. Then, variance increases in the Bootstrapping estimate from $p = 15$, and the two estimators start to diverge. This is expected. In our implementation of Cross-Validation the model is trained on the full dataset, and then within 5-folds a test-point is chosen on rotation of each fold. In our respectively dense grid $x \in (-1, 1)$ with $n \in (200, 575)$ sample points, each validation point is destined to have nearby neighbors within the fold. Therefore, our implementation of Cross-Validation method might be prone to being too optimistic, and report lower variance estimations. This might be the reason for why the two estimators start to diverge at higher degrees where variance kicks in. Another reason the CV MSE maintains a low value, is the fact that in contrast to our bootstrap implementation, our implementation of CV does not reserve some od the data for testing exclusively. Instead, every point in the data will be used for training.

Figure 15 quantifies where Ridge and Lasso improves the models compared to OLS: Ridge shows $> 1\%$ improvement in 4%, 13%, and 23% of configurations at $\sigma \in \{0.1, 0.2, 0.3\}$ respectively, with maximum gains of 15-23%. Lasso exhibits similar patterns with maximum gains of 19-22%. We see that the improvements are concentrated at high complexity and high noise, confirming regularization's primary benefit is controlling overfitting when signal-to-noise ratio is low. From the figure we see that the regularization is most effective at higher degrees, in the same region where we observed variance kick-in in the Bootstrap analysis. We see that higher level of noise expand or shift this region to lower complexity and larger sample sizes.

In the case of $\sigma = 0.1$ we can see a clear divide in MSE for both Ridge and Lasso. This divide is much clearer in the Lasso case, however present in Ridge as well. The effect reduces when we increase the noise, but is still present. As discussed previously, we can see that these models become favorable in regions close to this high MSE "blankets". The reason this blanket disap-

pears with higher noise, could be caused by the methods regularization terms, which is better at dampening the noise' contribution. When little noise is present, Ridge and especially Lasso might over-regularize. Here Lasso is worse than Ridge, because where Ridge only shrinks the collinear features, Lasso push them to zero. This makes Ridge more stable than Lasso. For Ridge the divide happens at $p = 14$, for Lasso it happens at $p = 10$, and for both the divide is constant across all sample sizes. This is similar to the behavior observed in for the Bias in Figure 12. As we discussed in section IV G, Bias has consistent behavior across sample sizes because it is very dependent on the chosen basis for the model we are considering. Maybe the effect we are observing in Figure 15, have a similar explanation. We will leave this for further studies.

V. CONCLUSION

We have explored OLS, Ridge and Lasso regression methods for approximating the Runge function. We have scaled data using z-score normalization to avoid numerical instability. We have seen that the simple OLS method often does a good job approximation the data in comparison to the other methods. We have also detected under- and overfitting related to the complexity of the model and discussed this in light of Ridge regressions, which counteracts overfitting at higher complexities than OLS. This has been discussed in light of the hyperparameter λ, which shrinks the parameters (especially the large ones) leading to stability.

We managed to perform regression with gradient descent for OLS and Ridge with high accuracy compared to the analytical solutions, and saw that the learning rate is important for convergence. There is critical point between divergence and convergence at a learning rate value around $\eta \approx 0.37$. This tells us that too large learning rates will never lead the parameters to small enough model MSE. We also saw that Ridge consistently needed more iterations to converge than OLS. Momentum turned out to be the fastest GD method to converge, while Adam wa second-best. AdaGrad was consistently slowest. We have seen that Lasso maintains low MSE for both training and test data. The train MSE is higher for Lasso then for Ridge, but it generalizes better to unseen data. Stochastic gradient descent turned out to not be especially useful, as we, in our case, deal with nearly smooth cost functions.

Finally, we used bootstrapping on OLS to analyze the Bias-Variance tradeoff. We found that Bootstrap-resampling is an effective technique for identifying regions in the models parameter space where bias and variance dominate, respectively. Subsequently, we were able to identify the region in this space where our model has the highest performance. This illustrates how Bootstrap-resampling is useful for estimating model performance on unseen data. We also found that the techniques is sensitive to the data we use for evaluating performance. In our case, we saw that the test-data did not possess the random distribution properties as the full dataset. Therefore, the error estimator became overly optimistic for the minimum MSE we could obtain in certain regions. When we compared the Bootstrapping resampling technique with Cross-Validation, this was confirmed. Using Cross-Validation we compared Ridge and Lasso regularization properties against the OLS model. We found that in regions with high variance in OLS-models, regularization is an effective tool to increase performance. In further studies it could be interesting exploring the high MSE blankets in figure 15. This figure is encoded with a lot of information, and could be tweaked to show a fuller picture of regression analysis. This could be used to analyze the differences between Ridge and Lasso regression.

[1] Wikipedia contributors. Runge's phenomenon. https://en.wikipedia.org/wiki/Runge%27s_phenomenon, 2025. Accessed: 2025-10-05.

[2] H. Haug, S. S. Thommesen, C. A. Falchenberg, and L. L. Storborg. Project 1. https://github.com/livelstorborg/FYS-STK4155, 2025. GitHub repository.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. https://doi.org/10.1038/s41586-020-2649-2.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. https://doi.org/10.21105/joss.03021.

[7] The pandas development team. pandas-dev/pandas: Pandas, August 2025. https://doi.org/10.5281/zenodo.16918803.

[8] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning. *Springer New York, NY*, page 38, 2009.

Appendix A

1. Bias-Variance Decomposition

The cost function $C(X,\theta)$ can be expressed in terms of the variance-bias decomposition. This can be derived as follows.

We start by defining Variance and Bias^2.

$$\mathrm{Var}[\tilde{y}(x)] = \mathbb{E}\left[(\tilde{y}(x) - \mathbb{E}[\tilde{y}(x)])^2\right], \tag{A1}$$

$$\mathrm{Bias}_f^2[\tilde{y}(x)] = \left(\mathbb{E}[\tilde{y}(x)] - f(x)\right)^2. \tag{A2}$$

We assume that the expectation values are taken over different samples for the training data. For fixed values x, we look at how the prediction $\tilde{y}$ changes when the model is trained on different training samples (e.g. bootstrap). Rewriting the terms inside the parenthesis.

$$\begin{aligned}(y-\tilde{y})^2 &= (y - \tilde{y} + E[\tilde{y}] - E[\tilde{y}])^2 \\ &= (y - E[\tilde{y}])^2 + (\tilde{y} - E[\tilde{y}])^2 \\ &\quad - 2(y\tilde{y} - yE[\tilde{y}] - \tilde{y}E[\tilde{y}] + E[\tilde{y}]^2)\end{aligned}$$

Now distributing the expectation operator linearly:

$$\begin{aligned}E[(y-\tilde{y})^2] &= E[(y - E[\tilde{y}])^2] + E[(\tilde{y} - E[\tilde{y}])^2] \\ &\quad - 2(E[y\tilde{y}] - E[y]E[\tilde{y}] - E[\tilde{y}]E[\tilde{y}] + E[\tilde{y}]^2) \\ &= E[(y - E[\tilde{y}])^2] + E[(\tilde{y} - E[\tilde{y}])^2] \\ &\quad - 2(E[y\tilde{y}] - E[y]E[\tilde{y}])\end{aligned}$$

Now we assume that our sampled values y, can be expressed as $y = f(x) + \epsilon$, where $f(x)$ is some deterministic function and $\epsilon \sim N(0,\sigma^2)$ is assumed to be a Gaussian distribution of random noise that is independent of $f(x)$. The noise in the training data is independent of the noise in the test samples. Since ϵ represents the noise in y and is independent of the training-data used to construct $\tilde{y}$, ϵ is independent of $\tilde{y}$: $E[\epsilon\tilde{y}] = E[\epsilon]E[\tilde{y}]$. ϵ follows a normal distribution with zero mean and variance σ^2, so per definition $E[\epsilon] = 0$, and $\mathrm{Var}(\epsilon) = E[\epsilon^2] = \sigma^2$. We also notice that since $f(x)$ is deterministic, we know $E[f(x)] = f(x)$ and that $E[f(x)\tilde{y}] = f(x)E[\tilde{y}]$. We can now start by expanding the expression within the parenthesis and use the substitution $y = f(x) + \epsilon$ to simplify.

$$\begin{aligned}(E[y\tilde{y}] - E[y]E[\tilde{y}]) &= E[(f+\epsilon)\tilde{y}] - E[f+\epsilon]E[\tilde{y}] \\ &= E[f]E[\tilde{y}] + E[\epsilon]E[\tilde{y}] - E[f]E[\tilde{y}] \\ &\quad - E[\epsilon]E[\tilde{y}] \\ &= 0\end{aligned}$$

So we are left with:

$$\begin{aligned}E[(y-\tilde{y})^2] &= E[(y - E[\tilde{y}])^2] + E[(\tilde{y} - E[\tilde{y}])^2] \\ &\quad - 2(E[y\tilde{y}] - E[y]E[\tilde{y}]) \\ &= E[(y - E[\tilde{y}])^2] + E[(\tilde{y} - E[\tilde{y}])^2]\end{aligned}$$

Here we notice that this is the sum of the variance and the bias squared, where the bias squared is given by:

$$\mathrm{Bias}_y^2[\tilde{y}] = E[(y(x) - E[\tilde{y}])^2]$$

We can expand this as before using the substitution $y(x) = f(x) + \epsilon$ again.

$$\begin{aligned}E[(y-\tilde{y})^2] &= \mathrm{Var}[\tilde{y}] + \mathrm{Bias}_y^2[\tilde{y}] \\ &= E[(y - E[\tilde{y}])^2] + E[(\tilde{y} - E[\tilde{y}])^2] \\ &= E[(\tilde{y} - E[\tilde{y}])^2] + E[y^2] - 2E[yE[\tilde{y}]] + E[\tilde{y}]^2 \\ &= E[(\tilde{y} - E[\tilde{y}])^2] + E[f^2] + E[2f\epsilon] + E[\epsilon^2] \\ &\quad - 2(E[f]E[\tilde{y}] + E[\epsilon]E[\tilde{y}]) + E[\tilde{y}]^2 \\ &= E[(\tilde{y} - E[\tilde{y}])^2] + E[f^2] - 2E[f]E[\tilde{y}] \\ &\quad + E[\tilde{y}]^2 + E[\epsilon^2] \\ &= E[(\tilde{y} - E[\tilde{y}])^2] + E[(f - E[\tilde{y}])^2] + E[\epsilon^2] \\ &= \mathrm{Var}[\tilde{y}] + \mathrm{Bias}_f^2[\tilde{y}] + \sigma^2\end{aligned}$$

Therefore we have shown that the cost-function or the expected error, can be decomposed as:

$$\begin{aligned}C(X,\theta) = E[(y-\tilde{y})^2] &= \mathrm{Var}[\tilde{y}] + \mathrm{Bias}_y^2[\tilde{y}] \\ &= \mathrm{Var}[\tilde{y}] + \mathrm{Bias}_f^2[\tilde{y}] + \sigma^2\end{aligned}$$

We see that we can express the bias-term both using the true function $f(x)$ and the values we sample y. These values differ by the irreducible error σ^2.