Technische Universität Dortmund
Fakultät Statistik

Fallstudien II / Case Studies II
Winter Semester 2025/26

Computer Experiments: Maximum Cauchy Stress Analysis

Group 4
Jiahui Fan, Ömercan Misirlioğlu, Yordan Saputra

Supervisors:
Prof. Dr. Katja Ickstadt, Dr. Henrike Weinert,
Yassine Talleb, Carmen van Meegen

Date: January 12, 2026

Contents

1 Introduction **1**

1.1 Objective . . . 1
1.2 Research Questions . . . 2
1.3 Approach to Problem Solving . . . 2
1.4 Key Results . . . 2
1.5 Report Structure . . . 2

2 The Physical System and Problem Description **3**

3 Statistical Methods **4**

3.1 Experimental Design: Latin Hypercube Sampling . . . 5
3.2 Surrogate Models . . . 6
3.2.1 Support Vector Regression (SVR) . . . 6
3.2.2 Random Forest (RF) . . . 7
3.2.3 Multi-layer Perceptron (MLP) . . . 8
3.2.4 Gaussian Process (GP) . . . 9
3.3 Model Selection and Evaluation . . . 10
3.4 Sensitivity Analysis: Permutation Feature Importance . . . 11
3.5 Distribution Analysis . . . 12
3.6 Resources and the Use of Artificial Intelligence . . . 12

4 Statistical Analysis **13**

4.1 Evaluation of Surrogate Model Accuracy . . . 13
4.2 Sensitivity Analysis . . . 13
4.3 Distribution Analysis of Maximum Stress . . . 15

5 Summary **17**

5.1 Final Remarks . . . 18

A Program Code **22**

A.1 Data Generation and LHS Implementation (R) . . . 22
A.2 Python Program Code . . . 25
A.2.1 Library Imports and Configuration . . . 25
A.2.2 Nested Cross-Validation and Hyperparameter Search . . . 25
A.2.3 Sensitivity Analysis: Permutation Importance Comparison . . . 28
A.2.4 Distribution Analysis . . . 30

1 Introduction

In modern engineering, analyzing the structural integrity of systems such as buildings, vehicles, and similar structures often requires complex numerical simulations. As noted by Gramacy (Gramacy, 2020), computer simulation experiments have become essential to modern scientific discovery because the "low-hanging fruit" of simple, closed-form mathematicalsolutions, which served as crude approximations in the past, has largely been exhausted (Gramacy, 2020). Today, solving interesting engineering problems with high fidelity requires intricate numerics that account for the finebalance, disequilibrium, and chaotic behavior inherent in the real world.

However, a fundamental challenge remains: while computing capacity has grown vastly over the last decades, it is not infinite. High-fidelity simulations, such as Finite Element Analysis (FEA) for a sun sail membrane, despite their precision, remain computationally expensive and time-consuming. This makes it impractical to perform large-scale sensitivity analyses, explore "what-if" scenarios, or conduct extensive Monte Carlo uncertainty quantifications directly on the simulator. In the context of this project, the physical simulator acts as a "black box", which should be modeled as precisely as possible, utilizing the finite computational resource of 200 trials properly.

Accordingly, to bridge the gap between the need for precise results and the finite computing time certain surrogate models are employed. These are essentially "meta-models", in other words models of the computer simulation itself, which are employed to solve mathematical systems that are too intricate to be worked by hand (Gramacy, 2020). By training a proper surrogate model on a limited set of observations, the behavior of the underlying physical system can be represented and emulated, thereby reducing the corresponding computational costs significantly while maintaining the potential for precise discovery and stress-testing.

1.1 Objective

The primary objective of this study is to predict the maximum Cauchy stress ($\sigma_{mem,max}$) in a sun sail membrane structure based on six variable input parameters and a surrogate model. By utilizing a space-filling design of 200 observations, it is aimed to build a robust and precise mathematical approximation model. Therefore it will be enabled to move beyond Newton's principle of parsimony, which assumes the simplest explanation is always best, to a model that can handle the inherent complexity of structural membrane responses (Gramacy, 2020).

1.2 Research Questions

To evaluate the reliability of the emulators and analyze the mechanical sensitivity of the sun sail, this report addresses three central research questions within a unified investigative framework.

First, a comparative assessment of model accuracy is conducted to determine whether Support Vector Regression, Random Forest, Multi-layer Perceptron, or Gaussian Processes provides the most precise prediction of the maximum Cauchy stress ($\sigma_{mem,max}$).

Second, a sensitivity analysis is performed to identify which of the six input parameters, which are categorized into pre-stress factors, material properties, and external loads, exerts the most significant influence on the resulting membrane stress.

Finally, the study focuses on Distribution Characterization to define the underlying statistical distribution of the maximum Cauchy stress under stochastic input variations and to ascertain which theoretical distribution provides the best fit for the observed data.

1.3 Approach to Problem Solving

The methodology follows a structured computational pipeline. First, a space-filling Latin Hypercube Sampling (LHS) design is used to generate 200 initial observations from the physical "black box" simulator. Second, four distinct surrogate model architectures (MLP, GP, SVR, and RF) are trained and hyperparameter-tuned using nested 10-fold cross-validation. Finally, the model with the best performance is utilized to conduct a large Monte Carlo simulation with $100,000$ samples to perform sensitivity analysis and determine the probabilistic distribution of the structural response.

1.4 Key Results

The evaluation demonstrates that the Multi-layer Perceptron (MLP) provides the highest predictive fidelity for the sun sail membrane, achieving an R^2 score of 0.9865. Sensitivity analysis identifies the load factor (f_{mem}) and membrane pre-stress (σ_{mem}) as the dominant drivers of structural stress. Furthermore, distribution fitting reveals that the maximum Cauchy stress is best characterized by a Weibull distribution, providing a robust basis for future designs.

1.5 Report Structure

The remainder of this report is organized as follows: Section 2 describes the physical system and the problem context. Section 3 describes the data generation process via Latin Hypercube Sampling and the theoretical background of the surrogate models. Section 4 presents the comparative performance of the models, the results of the

permutation-based sensitivity analysis, and the statistical distribution fitting. Finally, Section 5 summarizes the findings and discusses the implications for the computational design of membrane structures.

2 The Physical System and Problem Description

The focus of this investigation is a tensioned sun sail, a structural membrane system whose geometry is derived from a complex form-finding process based on the equilibrium of internal forces. Unlike traditional rigid structures, the stability and load capacity of a sun sail are not inherent but are instead achieved through a proper balance of pre-tensioning. This mechanical requirement requires a precise operational window: insufficient pre-stress can lead to structural instabilities or ponding while excessive tensioning risks exceeding the material's tearing limit. Consequently, understanding the structural response under varying conditions is vital for ensuring long-term structural integrity.

The central objective of this study is to develop a high-precision computational framework capable of emulating the structural response of the sun sail. To achieve this, the investigation focuses on three primary operational goals. First, the project aims to identify an optimal surrogate model by benchmarking the predictive accuracy of Support Vector Regression, Random Forest, Multi-layer Perceptron, or Gaussian Processes architectures against deterministic FEA results. Second, the study seeks to quantify the importance of each input variable through global sensitivity analysis, thereby identifying the dominant drivers of membrane stress. Finally, the objective includes a statistical characterization of the output space, determining the theoretical distribution that best represents the maximum Cauchy stress under stochastic conditions. Collectively, these objectives provide the analytical basis for assessing structural reliability and answering the previously established research questions.

The data material utilized for this analysis originates from a planned computer experiment, a methodology chosen because the underlying physical response is governed by non-linear partial differential equations typically solved via Finite Element Analysis. In this context, the FEA solver acts as a deterministic black-box function. To ensure a robust and space-filling exploration of the six-dimensional parameter space, a Latin Hypercube Sampling strategy was implemented to generate observations. This stratified sampling approach is significantly more efficient than simple random sampling for computer experiments, as it ensures that the marginal distributions of all input variables are sampled uniformly, thereby capturing complex interactions with a reduced number of simulations.

The structural model is defined by a comprehensive set of parameters categorized into stochastic variables, which drive the uncertainty analysis, and deterministic vari-

ables, which establish the fixed mechanical baseline for the simulations.

The stochastic space is defined by six primary input variables: the membrane surface pre-stress (σ_{mem}), the edge and support cable pre-stresses ($\sigma_{edg}, \sigma_{sup}$), the membrane's Young's modulus (E_{mem}) and Poisson's ratio (ν_{mem}), and the external surface loading (f_{mem}). While these stochastic inputs vary to capture system uncertainty, the structural integrity is also dependent on deterministic parameters, such as the membrane thickness (t_{mem}), the mechanical properties of the truss (E_{tru}, A_{tru}), and the geometric properties of the cables (d_{edg}, d_{sup}). All variables, their units, and their statistical types are summarized in Table 1.

Table 1: Specification of Model Input and Output Variables.

Part	**Quantity (Symbol)**	**Unit**	**Type**
Membrane	Young's modulus (E_{mem})	GPa	Stochastic
	Poisson's ratio (ν_{mem})	[-]	Stochastic
	Thickness (t_{mem})	mm	Deterministic
	Pre-stress (σ_{mem})	MPa	Stochastic
	Surface loading (f_{mem})	kPa	Stochastic
Truss	Young's modulus (E_{tru})	GPa	Deterministic
	Cross-sectional area (A_{tru})	cm^2	Deterministic
Edge cable	Young's modulus (E_{edg})	GPa	Deterministic
	Diameter (d_{edg})	mm	Deterministic
	Pre-stress (σ_{edg})	MPa	Stochastic
Support cable	Young's modulus (E_{sup})	GPa	Deterministic
	Diameter (d_{sup})	mm	Deterministic
	Pre-stress (σ_{sup})	MPa	Stochastic
Target	Max. Cauchy Stress ($\sigma_{mem,max}$)	MPa	Output

Data quality and numerical integrity are crucial in computer experiments, where traditional measurement noise is absent due to the deterministic nature of the FEA solver. This preprocessing ensures that the surrogate models are trained on a high-precision dataset that is free from numerical artifacts and accurately represents the physical limits of the sun sail system.

3 Statistical Methods

This section details the statistical framework used to generate the dataset, build predictive surrogate models, and analyze the importance of input parameters and the distribution of the output.

3.1 Experimental Design: Latin Hypercube Sampling

To efficiently explore the six-dimensional parameter space with limited computational resources, Latin Hypercube Sampling (LHS) was employed (Gramacy, 2020). Since the true physical model is a "black box," the choice of experimental design is crucial to ensure that the surrogate model captures the global behavior of the system without requiring an excessive number of expensive simulations.

Unlike Simple Random Sampling (SRS), which can leave large gaps in the input space or create clusters of points, LHS is a "stratified" sampling strategy. As described by Gramacy (2020), LHS provides a middle ground between a full factorial design (which suffers from the "curse of dimensionality") and random sampling. The primary advantage of LHS is that it ensures *univariate stratification*: if you project the d-dimensional samples onto any single dimension, the points are spread uniformly across the entire range. In addition, it allowed us to fully utilize all 200 simulation runs we have in our simulator.

The generation of the experimental design involved a structured four-stage approach to produce a sample size of $n = 200$ across the $d = 6$ input dimensions. Initially, the process commenced with marginal distribution scaling, where each input parameter was truncated at its 0.01% and 99.99% quantiles. This pre-processing step was essential to eliminate extreme outliers that might otherwise destabilize the training phase of the surrogate models. Following this scaling, the range of each marginalized variable underwent partitioning into $n = 200$ intervals of equal probability, ensuring that the entire range of each variable was accounted for in the sampling logic.

The core of the design was established through permutation and stratification, where a Latin Hypercube was constructed by generating d independent permutations of the sequence $\{1, 2, \ldots, n\}$. These permutations are mathematically designed to ensure that for any given dimension, each of the n bins contains exactly one sample point, thereby preventing the "clustering" common in simple random sampling. To further enhance the exploratory power of the design, a jittering technique was applied within each selected bin. By drawing sample points using a uniform distribution $U(0, 1)$ rather than selecting the bin centers, the points are prevented from falling onto a regular grid. This stochastic displacement is particularly valuable for identifying complex non-linearities in the response surface that a rigid grid might miss.

Ultimately, this procedure resulted in a 200×6 design matrix characterized by high-density space-filling properties. This ensures that the surrogate models receive sufficient and well-distributed information to interpolate accurately across the entire six-dimensional parameter domain.

3.2 Surrogate Models

Because the underlying physical simulation of the sun sail is computationally intensive, the simulator is replaced by a mathematical emulator $\hat{f}(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^6$ represents the vector of stochastic input parameters. This section provides a detailed theoretical overview of the four surrogate model architectures implemented to approximate the unknown simulator function $f(\mathbf{x})$ using the 200 observations generated via Latin Hypercube Sampling.

3.2.1 Support Vector Regression (SVR)

Support Vector Regression extends the principles of Support Vector Machines to regression problems by seeking a function that deviates from the training targets by no more than a predefined value ϵ, while simultaneously remaining as "flat" as possible to ensure generalization. In the context of the sun sail's response, the SVR maps input data into a high-dimensional feature space using a mapping $\Phi(\mathbf{x})$. The optimization objective is defined by the primal problem (Smola and Schölkopf, 2004, pp. 200):

$$\min_{\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\xi}^*} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) \tag{1}$$

subject to the constraints:

$$y_i - \langle \mathbf{w}, \Phi(\mathbf{x}_i)\rangle - b \leq \epsilon + \xi_i \tag{2}$$

$$\langle \mathbf{w}, \Phi(\mathbf{x}_i)\rangle + b - y_i \leq \epsilon + \xi_i^* \tag{3}$$

$$\xi_i, \xi_i^* \geq 0 \tag{4}$$

Here, $\mathbf{w}$ represents the weight vector, b the bias, and C a regularization parameter that determines the trade-off between the flatness of the model and the degree to which deviations larger than ϵ are tolerated. The slack variables ξ_i and ξ_i^* represent the distance of points falling outside the ϵ-insensitive "tube."

To identify the most suitable projection for the membrane stress data, both Linear and Radial Basis Function (RBF) kernels were evaluated. The Linear kernel (Smola and Schölkopf, 2004, pp. 201-202), defined as:

$$K(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}'\rangle \tag{5}$$

was utilized to test for direct proportional relationships between the input parameters and the maximum stress. Conversely, the RBF kernel (Smola and Schölkopf, 2004, p. 213), defined as:

$$K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma\|\mathbf{x} - \mathbf{x}'\|^2) \tag{6}$$

was employed to capture potential high-order non-linear mechanical interactions. This dual-kernel approach allows the model to compute dot products in high-dimensional feature spaces without explicitly calculating the coordinates of Φ, a technique known as the "kernel trick" (Smola and Schölkopf, 2004).

3.2.2 Random Forest (RF)

The Random Forest regressor is a sophisticated ensemble learning method that constructs a large collection of de-correlated decision trees during the training phase to improve predictive stability. For a regression task, the final model output $\hat{f}_{rf}(\mathbf{x})$ is obtained by calculating the arithmetic mean of the individual predictions from B decision trees $\{T_1(\mathbf{x}), T_2(\mathbf{x}), \ldots, T_B(\mathbf{x})\}$ (Breiman, 2001, p. 6):

$$\hat{f}_{rf}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} T_b(\mathbf{x}) \tag{7}$$

The robustness of this architecture stems from two primary sources of randomness. First, each individual tree is grown on a separate bootstrap sample drawn with replacement from the original 200 observations, a process formally known as bootstrap aggregating or "bagging." Second, during the construction of each tree, the algorithm employs feature subsampling: at every node split, it considers only a random subset of the six input parameters. This ensures that the resulting trees are decorrelated, preventing a single dominant feature from making every tree identical and thereby reducing the ensemble's overall variance (Breiman, 2001).

The internal logic of each tree follows a recursive partitioning approach. At any given node m, the algorithm identifies the optimal split $\theta = (j, t_m)$, which consists of a feature j and a threshold t_m, partitions the data into two child nodes, Q_{left} and Q_{right}. This split is determined by minimizing the weighted Mean Squared Error (MSE), which serves as the impurity criterion for regression:

$$H(Q_m, \theta) = \frac{n_{left}}{n_m}\text{MSE}(Q_{left}) + \frac{n_{right}}{n_m}\text{MSE}(Q_{right}) \tag{8}$$

where the MSE for a node Q is defined as:

$$\text{MSE}(Q) = \frac{1}{n}\sum_{i \in Q}(y_i - \bar{y})^2 \tag{9}$$

By iteratively minimizing this objective function, the trees create axis-aligned decision boundaries that effectively segment the six-dimensional parameter space into regions of similar Cauchy stress. In our implementation, hyperparameters such as the number of estimators ($B \in \{100, 200\}$) and the maximum tree depth were systematically varied to identify the optimal balance between bias and variance. By averaging

these high-variance, unbiased trees, the Random Forest produces a smooth response surface that is particularly resilient to potential noise in the simulator's output.

3.2.3 Multi-layer Perceptron (MLP)

The Multi-layer Perceptron is a powerful class of feedforward artificial neural networks capable of acting as a universal function approximator. For this study, the MLP was designed to map the six-dimensional input space $\mathbf{x} \in \mathbb{R}^6$ to the scalar response of the maximum Cauchy stress. The network's strength lies in its hierarchical structure, consisting of an input layer, multiple hidden layers, and an output layer, which allows it to extract and combine features of increasing complexity.

The fundamental operation of each neuron in a hidden layer l involves an affine transformation of the activations from the previous layer $\mathbf{a}^{(l-1)}$, followed by the application of a non-linear activation function σ. For a specific neuron j, this is mathematically expressed as (Goodfellow et al., 2016, p. 164-165):

$$a_j^{(l)} = \sigma\left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right) \tag{10}$$

In this equation, $w_{jk}^{(l)}$ represents the weight connecting the k-th neuron in layer $l-1$ to the j-th neuron in layer l, while $b_j^{(l)}$ denotes the bias term. To facilitate the learning of the highly non-linear structural response of the sun sail, the Rectified Linear Unit (ReLU) activation function is utilized, defined as $\sigma(z) = \max(0, z)$. ReLU is particularly effective in deep learning because it mitigates the vanishing gradient problem and introduces sparsity, enabling the model to represent the piecewise non-linear behavior of membrane stresses.

The network is trained through the minimization of a regularized cost function $J(\mathbf{w})$ that combines the mean squared error with L_2 regularization (weight decay) to prevent overfitting (Goodfellow et al., 2016, p. 227):

$$J(\mathbf{w}, \mathbf{b}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \frac{\alpha}{2}\|\mathbf{w}\|_2^2 \tag{11}$$

The optimization is performed via backpropagation using a constant learning rate η. By maintaining a fixed step size throughout the training process, the model avoids the complexities of decaying schedules, which is appropriate for the scale of the current experimental design. The parameter update rule is defined as:

$$w_{t+1} = w_t - \eta \frac{\partial J}{\partial w_t} \tag{12}$$

This gradient information allows the optimizer to iteratively update the parameters, ultimately converging on a configuration that provides the highest predictive fidelity

for the structural response (Goodfellow et al., 2016).

3.2.4 Gaussian Process (GP)

Gaussian Process regression, often referred to as Kriging in engineering design, is a non-parametric Bayesian framework that treats the simulator's response surface as a realization of a stochastic process. A GP is uniquely defined by a mean function $m(\mathbf{x})$, typically assumed to be zero, and a covariance function $k(\mathbf{x}, \mathbf{x}')$, which encodes the spatial correlation and prior assumptions regarding the underlying function's behavior (Gramacy, 2020, p. 144):

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{13}$$

The fundamental strength of the GP lies in its ability to perform formal Bayesian inference. Given a set of training observations $\mathbf{y}$ at locations X, the joint distribution of the training data and a prediction f^* at a previously unseen location $\mathbf{x}^*$ is modeled as a multivariate normal distribution (Gramacy, 2020, p. 144-146):

$$\begin{bmatrix} \mathbf{y} \\ f^* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X, \mathbf{x}^*) \\ K(\mathbf{x}^*, X) & k(\mathbf{x}^*, \mathbf{x}^*) \end{bmatrix}\right) \tag{14}$$

where K represents the covariance matrix, I is the identity matrix, and σ_n^2 accounts for potential observation noise. By conditioning the prior on the observed data, the posterior predictive distribution is derived, where the mean prediction μ^* and the associated uncertainty Σ^* are calculated as (Gramacy, 2020, p. 147-149):

$$\mu^* = K(\mathbf{x}^*, X)[K(X,X) + \sigma_n^2 I]^{-1}\mathbf{y} \tag{15}$$

$$\Sigma^* = k(\mathbf{x}^*, \mathbf{x}^*) - K(\mathbf{x}^*, X)[K(X,X) + \sigma_n^2 I]^{-1}K(X, \mathbf{x}^*) \tag{16}$$

To accurately model the mechanical response of the sun sail, the Matérn kernel is utilized. This kernel is widely preferred in physical sciences over the Squared Exponential kernel because it does not assume infinite differentiability, which can be unrealistically smooth for complex engineering surfaces. The general form of the Matérn kernel is (Gramacy, 2020, p. 192-195):

$$k_{Matern}(\mathbf{x}, \mathbf{x}') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\frac{d}{\ell}\right)^{\nu} K_\nu \left(\sqrt{2\nu}\frac{d}{\ell}\right) \tag{17}$$

where $d = \|\mathbf{x} - \mathbf{x}'\|$ is the Euclidean distance, ℓ is the characteristic length scale, and K_ν is the modified Bessel function of the second kind. Specifically, $\nu = 2.5$ is set, which results in a twice-differentiable process that captures the expected physical

smoothness of stress distributions while remaining robust to local fluctuations. The closed-form expression for this specific kernel is defined as:

$$k_{\nu=2.5}(d) = \sigma^2 \left(1 + \frac{\sqrt{5}d}{\ell} + \frac{5d^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}d}{\ell}\right) \tag{18}$$

This probabilistic approach ensures that the surrogate model not only yields high predictive accuracy but also provides a quantifiable measure of confidence in its estimations across the six-dimensional parameter domain (Gramacy, 2020).

3.3 Model Selection and Evaluation

To ensure a robust evaluation of model performance and prevent the risk of data leakage during hyperparameter tuning, we utilized a nested cross-validation framework, frequently referred to as double cross-validation, employing 10 outer and 10 inner folds (Blancas, 2022). This hierarchical approach is critical in scenarios involving limited datasets where using the same data for both parameter optimization and final performance assessment would lead to overoptimistic results.

The process is governed by an inner loop and an outer loop. In the inner loop, the data from the current training fold is further subdivided into 10 folds to perform hyperparameter optimization, such as identifying the optimal regularization parameter (α) for the MLP or the most effective kernel for the SVR. Once the optimal configuration is identified, the outer loop evaluates the model on the entirely "unseen" data of the outer fold. This separation ensures that the resulting performance metrics provide an unbiased estimate of the model's generalization error on future, independent samples from the physical simulator.

The primary performance metric utilized throughout this evaluation was the Coefficient of Determination (R^2), which quantifies the predictive accuracy of the surrogate model relative to a simple mean-based baseline. Mathematically, it is defined as:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{19}$$

where y_i represents the actual observed maximum Cauchy stress from the simulator, $\hat{y}_i$ is the corresponding prediction from the surrogate emulator, and $\bar{y}$ is the arithmetic mean of the observed stress values. Within this framework, the Residual Sum of Squares (SS_{res}) accounts for the unexplained variance, while the Total Sum of Squares (SS_{tot}) represents the total variance inherent in the dataset.

The interpretation of this metric is fundamental to model validation: an R^2 value approaching 1 indicates that the surrogate model's behavior closely emulates the physical "black box" simulator, suggesting that the emulator can effectively substitute for

the expensive numerical simulation. Conversely, an R^2 near 0 implies that the model fails to capture the underlying structural patterns, rendering it unable to describe the system more effectively than a simple average (Molnar, 2025).

3.4 Sensitivity Analysis: Permutation Feature Importance

To interpret the surrogate models and quantify the influence of each input parameter on the maximum Cauchy stress, we employed permutation feature importance. As described by Molnar (2025), this is a model-agnostic technique formally developed by Fisher et al. (2019).

The importance of an input variable X_j is determined by measuring the increase in the model's prediction error after its values are randomly shuffled. This breaks the relationship between X_j and the target $\sigma_{mem,max}$, effectively removing the feature's information from the model. The procedure follows these steps:

Algorithm 1 Permutation Feature Importance (Fisher et al., 2019)

Require: Trained surrogate model $\hat{f}$, test dataset $\mathcal{D}_{test}$ with n samples, and loss function L (e.g., MSE).

Ensure: Importance scores I_j for each feature $j \in \{1, \ldots, 6\}$.

1: Calculate the baseline error on the original dataset:

$$e_{orig} = L(\mathcal{D}_{test}, \hat{f})$$

2: **for** each feature $j \in \{1, \ldots, 6\}$ **do**

3: Create a permuted dataset $\mathcal{D}_{perm}$ by randomly shuffling the values of column j in $\mathcal{D}_{test}$.

4: Calculate the error on the permuted data:

$$e_{perm} = L(\mathcal{D}_{perm}, \hat{f})$$

5: Compute the ratio-based importance score:

$$I_j = \frac{e_{perm}}{e_{orig}}$$

6: **end for**

7: Rank features X_j by descending order of I_j.

A high importance score indicates that the model relies heavily on that specific feature for its predictions. This method is preferred over standard correlation measures because it captures non-linear interactions and is not biased by the scale of input variables (Molnar, 2025).

3.5 Distribution Analysis

Following the sensitivity analysis, we focus on characterizing the resulting distribution of $\sigma_{mem,max}$. Understanding the probability density function (PDF) of the stress is crucial for reliability engineering.

In this stage, the most accurate surrogate model (MLP) is used to perform a large-scale Monte Carlo simulation ($N = 100,000$). We employ the Kolmogorov-Smirnov test to compare the empirical distribution against theoretical distributions, such as the Lognormal or Weibull distributions, which are commonly used for material stress analysis.

For the final characterization of the system's stochastic behavior, we performed a Monte Carlo simulation. Using the best-performing surrogate model (MLP), we generated 100,000 synthetic samples by drawing from the input distributions. These results were then used to estimate the probability density function (PDF) of the maximum Cauchy stress.

We fitted the resulting data to several candidate distributions using `SciPy` library (Virtanen et al., 2020): Normal, Lognormal, Gumbel, Gamma, and Weibull. The parameters for these distributions were estimated using Maximum Likelihood Estimation (MLE), which identifies the parameter vector $\hat{\theta}$ that maximizes the likelihood function $\mathcal{L}(\theta)$. To rigorously determine the "best fit," we utilized the Kolmogorov-Smirnov (KS) test. The KS statistic D_n is defined as:

$$D_n = \sup_x |F_n(x) - F(x)| \tag{20}$$

where $F_n(x)$ is the empirical distribution function and $F(x)$ is the cumulative distribution function (CDF) of the candidate theoretical distribution. A lower D_n value indicates a better fit.

3.6 Resources and the Use of Artificial Intelligence

The methodological framework and statistical analyses presented in this report rely on a synthesis of foundational literature, specialized software, and computational tools. Theoretical grounding for surrogate modeling and Gaussian processes is derived from Gramacy (2020), while the principles of deep learning and neural architectures are based on Goodfellow et al. (2016). Specific algorithmic implementations for Support Vector Regression and Random Forests follow the methodologies established by Smola and Schölkopf (2004) and Breiman (2001), respectively. The interpretive framework for sensitivity analysis is informed by the work of Molnar (2025) and Fisher et al. (2019), with model selection strategies guided by the nested cross-validation principles described by Blancas (2022).

Computational implementations were executed using Python 3.12.2 (Van Rossum and Drake, 2009), utilizing Scikit-Learn 1.5.1 (Pedregosa et al., 2011) for surrogate modeling and SciPy 1.13.1 (Virtanen et al., 2020) for distribution fitting and numerical optimization. Data preprocessing, exploratory analysis, and the Latin Hypercube Sampling (LHS) design were conducted using R 4.4.2 (R Core Team, 2024). All graphical representations were generated with Matplotlib 3.9.2 (Hunter, 2007). Additionally, the AI model Gemini 1.5 (Google Gemini Team, 2024) was utilized as a supportive tool for structural organization, LaTeX formatting, and linguistic refinement. All statistical interpretations, code implementations, and final results remain the original work of the authors.

4 Statistical Analysis

This section presents the empirical findings of our study, structured around the three primary research questions. We analyze the predictive performance of the surrogate models, the sensitivity of the system to input parameters, and the stochastic behavior of the maximum stress.

4.1 Evaluation of Surrogate Model Accuracy

The predictive performance of the four candidate surrogate models, including Gaussian Process (GP), Multi-layer Perceptron (MLP), Support Vector Regression (SVR), and Random Forest (RF), was evaluated using a nested 10-fold cross-validation approach.

Table 2 summarizes the results for all hyperparameter configurations. The models are ranked by their mean coefficient of determination (R^2), representing the proportion of variance in the maximum Cauchy stress ($\sigma_{mem,max}$) captured by the surrogate.

Based on these results, the MLP with a learning rate η of 0.01 and $\alpha = 0.0001$ was selected as the final surrogate model. While the Gaussian Process with a Matern kernel performed similarly, the MLP exhibited the highest overall stability (lowest standard deviation) and accuracy, making it the most suitable tool for the subsequent sensitivity and distribution analyses. One could wonder why some SVR outputs has negative R^2 values, which happens when a model predicts worse than a simple mean prediction. The problem can either be explained by a poorly fitting model or model misspecifications such as ignoring non-linear relationships.

4.2 Sensitivity Analysis

To characterize the influence of the input variables on the maximum Cauchy stress ($\sigma_{mem,max}$), a sensitivity analysis was performed using the permutation importance method. This approach quantifies the dependency of the surrogate model on each

Table 2: Performance comparison of surrogate models (sorted by Mean R^2).

Model	**Parameters**	**Mean R^2**	**Std R^2**
MLP	$\alpha = 0.0001, \eta = 0.01$	**0.9865**	0.0026
MLP	$\alpha = 0.001, \eta = 0.01$	0.9864	0.0027
GP	Kernel: Matern ($\nu = 2.5$)	0.9849	0.0044
MLP	$\alpha = 0.001, \eta = 0.1$	0.9668	0.0604
MLP	$\alpha = 0.0001, \eta = 0.1$	0.9456	0.0841
GP	Kernel: RBF	0.9332	0.0157
SVR	$C = 10$, Kernel: Linear	0.9244	0.0070
RF	Depth=10, Split=2, Est=200	0.7919	0.0110
RF	Depth=10, Split=2, Est=100	0.7912	0.0129
RF	Depth=10, Split=10, Est=200	0.7608	0.0117
RF	Depth=10, Split=10, Est=100	0.7596	0.0126
RF	Depth=5, Split=2, Est=200	0.7406	0.0132
RF	Depth=5, Split=2, Est=100	0.7398	0.0143
RF	Depth=5, Split=10, Est=200	0.7265	0.0131
RF	Depth=5, Split=10, Est=100	0.7262	0.0132
SVR	$C = 1$, Kernel: Linear	0.2574	0.0277
SVR	$C = 10$, Kernel: RBF	0.1231	0.0341
SVR	$C = 0.1$, Kernel: Linear	-0.0503	0.0473
SVR	$C = 1$, Kernel: RBF	-0.0669	0.0480
SVR	$C = 0.1$, Kernel: RBF	-0.0870	0.0496

feature by calculating the reduction in the coefficient of determination (R^2) when the feature's values are randomly permuted. A higher importance score signifies that the parameter has a significant effect on the structural response.

That being the case, four candidate models have been created and trained using the best hyperparameter settings found in the former section. Accordingly, the sensitivity analysis was applied to the four candidate models that have been created to verify the consistency of the findings across different learning architectures.

Figure 1 displays the permutation importance scores for each input parameter based on 100 iterations. Although absolute values fluctuate between the SVR, Random Forest, and MLP models, the consistency in feature ranking suggests that the surrogate models have converged on a physically meaningful representation of the system. The distribution of these scores, visualized via boxplots, confirms the reliability of the identified parameter sensitivities.

The results clearly indicate that the membrane pre-stress (σ_{mem}) is the most influential parameter followed by surface loading (f_{mem}) and edge cable pre-stress (σ_{edg}), accounting for the majority of the variance in the output stress. Conversely, the material constants, including Young's modulus (E_{mem}) and Poisson's ratio (ν_{mem}), exhibit significantly lower importance scores. This suggests that the structural design of the sun sail is primarily governed by its tensioning and loading environment rather than small variations in its mechanical properties.

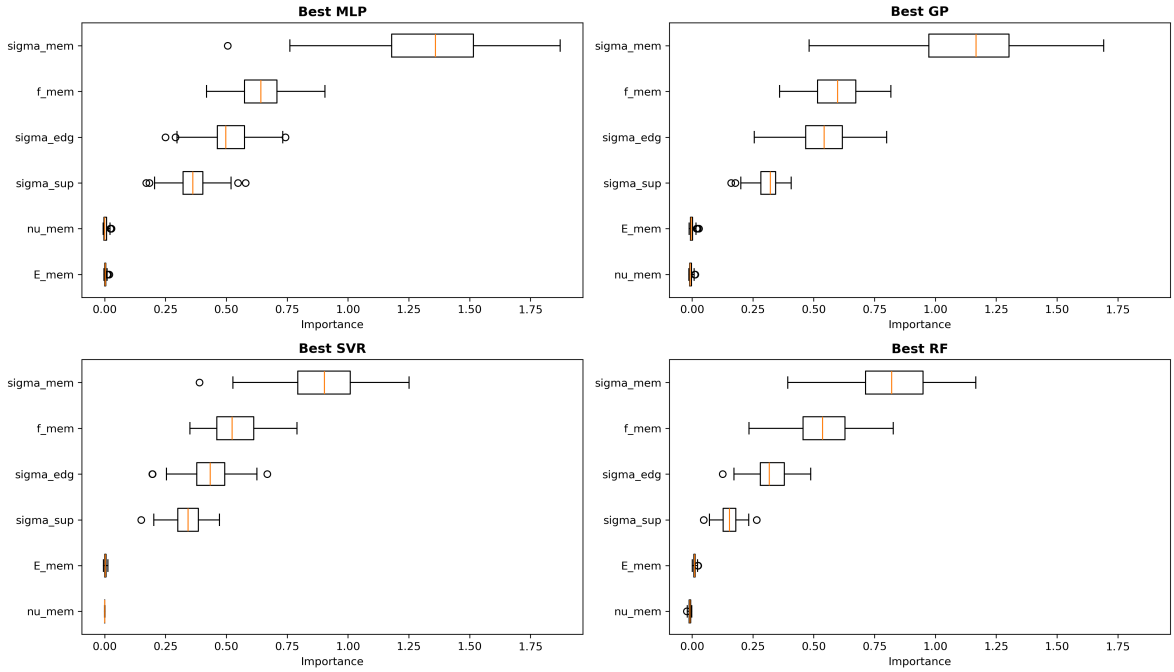

Figure 1: Permutation importance comparison across the surrogate models.

4.3 Distribution Analysis of Maximum Stress

Firstly, to characterize the probabilistic nature of the structural response, the maximum Cauchy stress results obtained from $100,000$ samples were analyzed against several theoretical probability distributions. This step is crucial for reliability assessment, as it identifies which statistical model best represents the stochastic behavior of the membrane under the defined uncertainties. The candidate distributions evaluated are namely Normal, Lognormal, Gumbel, Gamma, and Weibull.

Accordingly, a goodness-of-fit was quantified using the Kolmogorov-Smirnov (KS) test, where a lower test statistic indicates a closer match between the empirical data and the theoretical distribution. The descriptive statistics and the KS test results for each distribution are summarized in Table 3.

The analysis indicates that the Weibull distribution provides the better fit for the maximum membrane stress, yielding the lowest KS statistic of 0.0248.

Table 3: Descriptive statistics and Kolmogorov-Smirnov (KS) test results

Distribution	**Mean**	**Std. Dev.**	**Skewness**	**Median**	**KS Stat**
Empirical Samples	4311.45	375.29	1.14	4278.85	—
Weibull	**4311.86**	**373.09**	**0.82**	**4255.32**	**0.0248**
Gumbel	4311.32	379.09	1.14	4249.04	0.0389
Lognormal	4312.22	383.08	1.22	4245.52	0.0411
Gamma	4311.45	387.20	1.17	4237.65	0.0447
Normal	4311.45	375.29	0.00	4311.45	0.0578

The visual comparison between the Empirical Cumulative Distribution Function (ECDF) of the generated samples and the theoretical Cumulative Distribution Functions (CDFs) of the candidate models is illustrated in Figure 2, alongside the corresponding Q-Q (Quantile-Quantile) plots. The top row of the figure displays the global alignment of the distributions, where the ECDF represents the step-wise cumulative probability of the 100,000 Monte Carlo samples. By comparing this to the continuous theoretical curves, one can observe how closely each model tracks the mean and the variance of the structural response.

The bottom row provides the Q-Q plots, which are essential for evaluating the goodness-of-fit in the distribution tails. In these plots, the quantiles of the empirical data are plotted against the quantiles of the theoretical distributions; a perfect fit would follow the 45-degree reference line exactly. The deviations observed in the extreme quantiles of the Normal and Gamma distributions further highlight their inability to accurately capture the peak stress events. Conversely, the Weibull distribution exhibits the most consistent alignment across the entire range, particularly in the upper tail, which is the most critical region for structural safety and failure probability estimation.

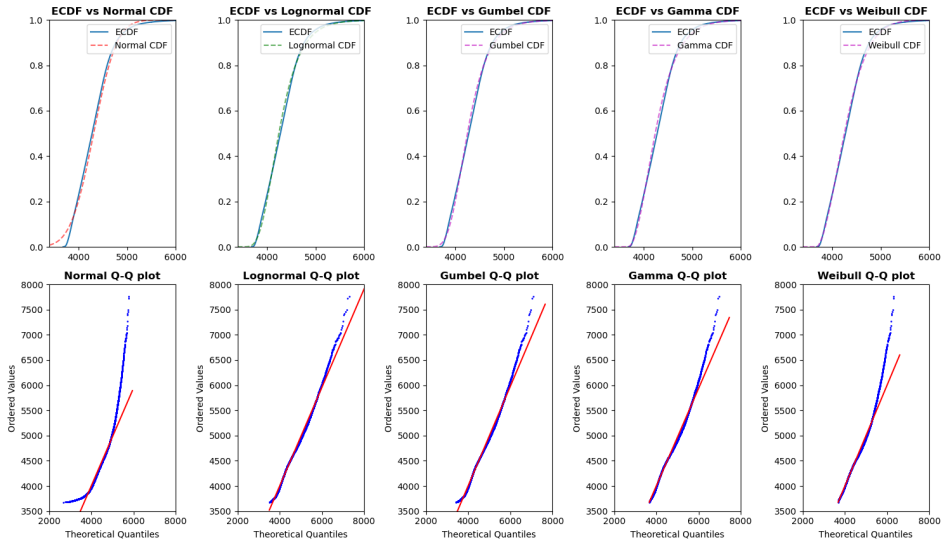

Figure 2: Distribution analysis of the maximum Cauchy stress.

Further insight into the frequency of stress occurrences is provided by the histogram in Figure 3. The empirical distribution of the maximum Cauchy stress is visibly right-skewed, characterized by a long tail extending toward higher stress values. This inherent asymmetry immediately disqualifies the Normal distribution as a viable candidate, as its symmetric nature fails to account for the increased probability of extreme stress events in a tensioned membrane system.

One can see that the Lognormal, Gumbel, and Gamma distributions all incorporate some degree of right-skewness, providing the visual evidence for the fit. This visual evidence, when coupled with the previously discussed Kolmogorov-Smirnov test results,

confirms that the Weibull distribution is the most appropriate statistical model for characterizing the stochastic response of the sun sail under environmental and pre-stress uncertainties.

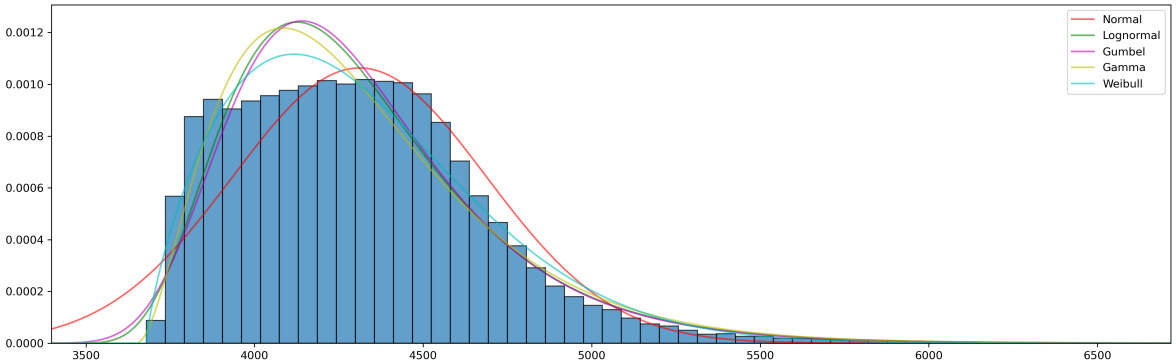

Figure 3: Histogram of the distribution of maximum stres.

5 Summary

This report investigated the application of surrogate modeling to predict the maximum Cauchy stress ($\sigma_{mem,max}$) in a sun sail structure, utilizing a 200-sample design generated through Latin Hypercube Sampling. This way it was aimed to analyze the research questions presented at the beginning by treating the complex mechanical simulation as a "black box,". Various statistical approaches were evaluated to maximize predictive accuracy while keeping the computational efficiency in an acceptable range.

Consequently, the investigation has yielded several critical insights regarding the efficacy of surrogate modeling and the stochastic nature of tensioned structures. Regarding the first research objective, the comparative performance analysis of the candidate emulators demonstrated that the Multi-layer Perceptron (MLP) and Gaussian Process (GP) regression models possess superior predictive accuracy for this specific high-dimensional system. Both architectures achieved a coefficient of determination (R^2) exceeding 0.98, successfully capturing the complex non-linearities inherent in the membrane's mechanical behavior. While the MLP's success is attributed to its ability to model deep hierarchical relationships within the input space, the Gaussian Process effectively utilized spatial interpolation via the Matérn kernel to provide highly precise local approximations. In contrast, the Support Vector Regression (SVR) and Random Forest (RF) models proved less robust, suggesting that axis-aligned decision boundaries and standard kernel-based regression may be insufficient to fully emulate the response surfaces generated by high-fidelity simulators.

Beyond model selection, the permutation feature importance analysis established a definitive hierarchy of parameter influence, providing a clearer understanding of the

system's sensitivity. The findings indicate that the membrane pre-stress (σ_{mem}) is the paramount driver of the maximum Cauchy stress, followed closely by the external snow load factor (f_{mem}). This dominance suggests that the structural safety of the sail is fundamentally governed by its initial tensioning state and external environmental forces rather than its inherent material constants. Interestingly, material properties such as the Young's modulus (E_{mem}) and Poisson's ratio (ν_{mem}) exhibited a negligible impact on the output stress within the defined stochastic ranges. This implies that for the design of such tensioned systems, precision in installation and load prediction outweighs the marginal variations in material stiffness.

Finally, the stochastic characterization of the system through a large-scale Monte Carlo simulation provided a robust probabilistic basis for reliability assessment. By utilizing the optimized MLP surrogate, it was determined that the resulting distribution of the maximum Cauchy stress is best represented by a Weibull distribution. The goodness-of-fit was statistically validated using the Kolmogorov-Smirnov test, which confirmed the Weibull model's superiority over standard Normal or Gumbel candidates in capturing the observed right-skewed behavior. While the overall fit is highly significant, the characterization of the extreme upper tails remains a significant consideration for structural safety. The identification of a Weibull-distributed response allows for a more rigorous calculation of failure probabilities, ensuring that the sun sail design maintains a sufficient margin of safety against material tearing under extreme loading scenarios.

5.1 Final Remarks

The results of this study underscore the transformative potential of high-precision emulators in the structural analysis of tensioned membranes. By successfully bridging the gap between the computational intensity of Finite Element Analysis and the requirements of large-scale stochastic exploration, this research demonstrates that surrogate models are not merely approximations, but essential tools for modern uncertainty quantification. The ability to perform $10,000$ evaluations in seconds, which is a task that would be physically and financially prohibitive using traditional solvers, allows engineers to shift from deterministic safety factors to a reliability-based design philosophy.

Moving forward, the integration of active learning, which is often referred to as sequential experimental design, epresents a promising frontier for refining these emulators. While the current Latin Hypercube Sampling provided an acceptable global overview of the response surface, an active learning approach could iteratively identify and sample localized regions of the design space associated with high stress or potential failure. By densifying the training data specifically where the membrane reaches its physical limits, future models could achieve even greater precision in the stochastic

tails of the distribution. This evolution toward adaptive surrogates will be useful in the development of the next generation of efficient and resilient architectural structures, where the margin between optimal performance and structural collapse is increasingly narrow.

References

Eduardo Blancas. Model selection done right: A gentle introduction to nested cross-validation, Apr 2022. URL https://ploomber.io/blog/nested-cv/.

Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously. *Journal of Machine Learning Research*, 20(177): 1–81, 2019. URL http://jmlr.org/papers/v20/18-760.html.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Google Gemini Team. *Gemini 1.5: A Multimodal Model for Long Context and Reasoning*. Google, 2024. URL https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/. Accessed: 2026-01-11.

Robert B. Gramacy. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Chapman Hall/CRC, Boca Raton, Florida, 2020. http://bobby.gramacy.com/surrogates/.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. Version 3.9.2.

Christoph Molnar. *Interpretable Machine Learning*. 3 edition, 2025. ISBN 978-3-911578-03-5. URL https://christophm.github.io/interpretable-ml-book.

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.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2024. URL https://www.R-project.org/. Version 4.4.2.

Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14:199–222, 2004.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. Version 3.12.2.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2. Version 1.13.1.

A Program Code

A.1 Data Generation and LHS Implementation (R)

The following code details the parameterization of the input variables and the construction of the Latin Hypercube design matrix.

Listing 1: Latin Hypercube Sampling and Distribution Scaling in R

```r
library(EnvStats)
library(lhs)   # for Latin Hypercube sampling matrix

# ---------------------------------------------------------
# Distribution parameters for the Sun Sail System
# ---------------------------------------------------------
log_params <- list(
  E_mem     = list(mu = -0.5219509282334006,
  sigma = 0.14916638004195087),
  sigma_mem = list(mu =  1.36668400454325,
    sigma = 0.1980422004353651),
  sigma_edg = list(mu =  5.84877686433685,
     sigma = 0.19804055309923496),
  sigma_sup = list(mu =  5.973937403748808,
    sigma = 0.19804026238761846)
)

uniform_params <- list(
  nu_mem = list(a = 0.38008141571295795, b = 0.4199185842870421)
)

gumbel_params <- list(
  f_mem = list(loc = 0.34599361509451665, scale = 0.09356361614804114)
)

variables <- c("E_mem", "nu_mem", "sigma_mem", "sigma_edg",
 "sigma_sup", "f_mem")
N <- 200 # Number of samples

# ---------------------------------------------------------
# Generation of Quantile Boundaries (Truncated at 0.01% and 99.99%)
# ---------------------------------------------------------
get_bin_bounds <- function(name, num_bins) {
  qs <- seq(1e-4, 1 - 1e-4, length.out = num_bins + 1)
```

```r

  if (name %in% names(log_params)) {
    p <- log_params[[name]]
    bounds <- qlnorm(qs, meanlog = p$mu, sdlog = p$sigma)
  } else if (name == "nu_mem") {
    p <- uniform_params$nu_mem
    bounds <- qunif(qs, min = p$a, max = p$b)
  } else if (name == "f_mem") {
    p <- gumbel_params$f_mem
    bounds <- qgumbel(qs, location = p$loc, scale = p$scale)
  } else stop("Unknown variable")

  return(bounds)
}

# ------------------------------------------------------------
# Stratified Sampling Logic
# ------------------------------------------------------------
set.seed(2025)
lhs_indices <- randomLHS(N, length(variables))
lhs_bins <- apply(lhs_indices, 2, function(col) ceiling(col * N))
lhs_bins[lhs_bins == 0] <- 1

sample_from_bins <- function(name, bins, bounds) {
  samples <- numeric(length(bins))
  for (i in seq_along(bins)) {
    low <- bounds[bins[i]]
    high <- bounds[bins[i] + 1]

    # Jittering: Uniform draw within the quantile bin
    if (abs(high - low) < 1e-12) {
      samples[i] <- low
    } else {
      samples[i] <- runif(1, min = low, max = high)
    }
  }
  return(samples)
}

# Build Final Design Matrix
X <- matrix(nrow = N, ncol = length(variables))
```

```r
colnames(X) <- variables

for (j in seq_along(variables)) {
  name <- variables[j]
  bounds <- get_bin_bounds(name, N)
  X[, j] <- sample_from_bins(name, lhs_bins[, j], bounds)
}

X <- as.data.frame(X)
```

A.2 Python Program Code

The following sections contain the complete Python implementation for the surrogate modeling, sensitivity analysis, and uncertainty propagation.

A.2.1 Library Imports and Configuration

```
import itertools
import matplotlib.pyplot as plt
import math
import numpy as np
import pandas as pd
import seaborn as sns

from scipy import stats
from scipy.stats import anderson, kstest, kurtosis, skew
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor,
    ↪ VotingRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, Matern, ConstantKernel
from sklearn.inspection import permutation_importance
from sklearn.model_selection import cross_val_score, GridSearchCV, KFold,
    ↪ LeaveOneOut, train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR

pd.set_option('display.max_colwidth', 500)
pd.set_option('display.float_format', '{:.6f}'.format)

# Setting global seeds for reproducibility
RANDOM_STATE = 202512
np.random.seed(RANDOM_STATE)
```

A.2.2 Nested Cross-Validation and Hyperparameter Search

This implementation follows a nested 10-fold CV approach to ensure unbiased performance estimation across SVR, Random Forest, MLP, and GP models.

```
RANDOM_STATE = 202512
np.random.seed(RANDOM_STATE)

def get_fold_results(grid_search, model_name, outer_fold):
    fold_results = pd.DataFrame(grid_search.cv_results_)
```

```
    fold_results = fold_results[['params', 'mean_test_score', 'std_test_score'
    ↪ ]]
    fold_results['params'] = fold_results['params'].apply(str)
    fold_results.insert(0, 'model_name', model_name)
    fold_results.insert(0, 'outer_fold', outer_fold + 1)
    return fold_results

def get_summary(row):
    fold_columns = [f'fold{fold}_mean_r2' for fold in range(1, 11)]
    fold_scores = row[fold_columns].values
    return pd.Series({
        'mean_of_mean': np.mean(fold_scores),
        'std_of_mean': np.std(fold_scores),
        'min_of_mean': np.min(fold_scores),
        '2.5%_of_mean': np.percentile(fold_scores, 2.5),
        'median_of_mean': np.median(fold_scores),
        '97.5%_of_mean': np.percentile(fold_scores, 97.5),
        'max_of_mean': np.max(fold_scores)
    })


X = df[['E_mem', 'nu_mem', 'sigma_mem', 'sigma_edg', 'sigma_sup', 'f_mem']].
    ↪ values
y = df['sigma_mem_max'].values

outer_cv = KFold(n_splits=10, shuffle=True, random_state=RANDOM_STATE)
inner_cv = KFold(n_splits=10, shuffle=True, random_state=RANDOM_STATE)

results = []

for outer_fold, (train_idx, test_idx) in enumerate(outer_cv.split(X, y)):
    print(f'OUTER FOLD: {outer_fold + 1}')

    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # SUPPORT VECTOR REGRESSION
    print(f'Fitting SVR model')
    params = {'kernel': ['linear', 'rbf'], 'C': [0.1, 1, 10]}
    svr_grid_search = GridSearchCV(SVR(), params, cv=inner_cv, scoring='r2',
    ↪ verbose=1, n_jobs=-1)
    svr_grid_search.fit(X_train_scaled, y_train)
    fold_results = get_fold_results(svr_grid_search, 'SVR', outer_fold)
```

```
    results.append(fold_results)

    # RANDOM FOREST REGRESSION
    print('Fitting RF model')
    params = {'n_estimators': [100, 200], 'max_depth': [5, 10], '
    ↪ min_samples_split': [2, 10]}
    rf_grid_search = GridSearchCV(RandomForestRegressor(random_state=
    ↪ RANDOM_STATE), params, cv=inner_cv, scoring='r2', verbose=1, n_jobs=-1)
    rf_grid_search.fit(X_train_scaled, y_train)
    fold_results = get_fold_results(rf_grid_search, 'RF', outer_fold)
    results.append(fold_results)

    # DEEP NEURAL NETWORK REGRESSION
    print('Fitting MLP model')
    params = {'alpha': [0.0001, 0.001], 'learning_rate_init': [0.01, 0.1]}
    mlp_grid_search = GridSearchCV(MLPRegressor(hidden_layer_sizes=(64, 32),
    ↪ max_iter=10000, random_state=RANDOM_STATE), params, cv=inner_cv, scoring
    ↪ ='r2', verbose=1, n_jobs=-1)
    mlp_grid_search.fit(X_train_scaled, y_train)
    fold_results = get_fold_results(mlp_grid_search, 'MLP', outer_fold)
    results.append(fold_results)

    # GAUSSIAN PROCESS REGRESSION
    print('Fitting GP model')
    params = {'kernel': [RBF(), Matern(nu=2.5)]}
    gp_grid_search = GridSearchCV(GaussianProcessRegressor(random_state=
    ↪ RANDOM_STATE), params, cv=inner_cv, scoring='r2', verbose=1, n_jobs=-1)
    gp_grid_search.fit(X_train_scaled, y_train)
    fold_results = get_fold_results(gp_grid_search, 'GP', outer_fold)
    results.append(fold_results)

    print('\n')

results = pd.concat(results, ignore_index=True)
results = pd.pivot_table(results, index=['model_name', 'params'], columns=['
    ↪ outer_fold'], values=['mean_test_score', 'std_test_score'])
results.columns = [f'fold{fold}_{metric.replace('test_score', 'r2')}' for
    ↪ metric, fold in results.columns]
results = results.reset_index()

fold_columns = [f'fold{fold}_mean_r2' for fold in range(1, 11)]
sorted_results = results.sort_values('fold1_mean_r2', ascending=False).
    ↪ reset_index(drop=True)[['model_name', 'params'] + fold_columns]
sorted_results.to_csv(path + 'sorted_results.csv', index=False)
sorted_results

summary_df = results.apply(get_summary, axis=1)
```

```
summary_df = pd.concat([results, summary_df], axis=1)
summary_df = summary_df.drop(columns=[f'fold{fold}_mean_r2' for fold in range
    ↪ (1, 11)] + [f'fold{fold}_std_r2' for fold in range(1, 11)])
summary_df = summary_df.sort_values(['mean_of_mean', 'std_of_mean'], ascending
    ↪ =[False, True]).reset_index(drop=True)
summary_df.to_csv(path + 'summary_df.csv', index=False)
summary_df
```

A.2.3 Sensitivity Analysis: Permutation Importance Comparison

The following code block visualizes the feature importance across all four surrogate models (MLP, GP, SVR, and Random Forest) using a unified scale for direct comparison.

```
# PARAMETERS OF THE MODEL ARE SELECTED BASED ON NESTED CROSS VALIDATION
feature_names = ['E_mem', 'nu_mem', 'sigma_mem', 'sigma_edg', 'sigma_sup', '
    ↪ f_mem']
X = df[feature_names].values
y = df['sigma_mem_max'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    ↪ random_state=RANDOM_STATE)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

best_model1 = MLPRegressor(hidden_layer_sizes=(64, 32), alpha=0.0001,
    ↪ learning_rate_init=0.01, max_iter=10000, random_state=RANDOM_STATE)
best_model1.fit(X_train_scaled, y_train)
perm_result1 = permutation_importance(best_model1, X_test_scaled, y_test,
    ↪ n_repeats=100, random_state=RANDOM_STATE, n_jobs=-1)
sorted_idx1 = perm_result1.importances_mean.argsort()

best_model2 = GaussianProcessRegressor(Matern(nu=2.5), random_state=
    ↪ RANDOM_STATE)
best_model2.fit(X_train_scaled, y_train)
perm_result2 = permutation_importance(best_model2, X_test_scaled, y_test,
    ↪ n_repeats=100, random_state=RANDOM_STATE, n_jobs=-1)
sorted_idx2 = perm_result2.importances_mean.argsort()

best_model3 = SVR(kernel='linear', C=10)
best_model3.fit(X_train_scaled, y_train)
perm_result3 = permutation_importance(best_model3, X_test_scaled, y_test,
    ↪ n_repeats=100, random_state=RANDOM_STATE, n_jobs=-1)
sorted_idx3 = perm_result3.importances_mean.argsort()

```

```
best_model4 = RandomForestRegressor(n_estimators=200, max_depth=10,
    ↪ min_samples_split=2, random_state=RANDOM_STATE)
best_model4.fit(X_train_scaled, y_train)
perm_result4 = permutation_importance(best_model4, X_test_scaled, y_test,
    ↪ n_repeats=100, random_state=RANDOM_STATE, n_jobs=-1)
sorted_idx4 = perm_result4.importances_mean.argsort()

stacked_dataset = np.stack([perm_result1.importances, perm_result2.importances,
    ↪  perm_result3.importances, perm_result4.importances])
x_min, x_max = 0, np.max(stacked_dataset)
x_padding = 0.05 * (x_max - x_min)
x_min, x_max = x_min - x_padding, x_max + x_padding

fig, axes = plt.subplots(2, 2, figsize=(15, 9))

axes[0, 0].boxplot(perm_result1.importances[sorted_idx1].T, vert=False,
    ↪ tick_labels=[feature_names[i] for i in sorted_idx1])
axes[0, 0].set_title('Best MLP', fontsize=12, fontweight='bold')
axes[0, 0].set_xlabel('Importance')
axes[0, 0].set_xlim([x_min, x_max])

axes[0, 1].boxplot(perm_result2.importances[sorted_idx2].T, vert=False,
    ↪ tick_labels=[feature_names[i] for i in sorted_idx2])
axes[0, 1].set_title('Best GP', fontsize=12, fontweight='bold')
axes[0, 1].set_xlabel('Importance')
axes[0, 1].set_xlim([x_min, x_max])

axes[1, 0].boxplot(perm_result3.importances[sorted_idx3].T, vert=False,
    ↪ tick_labels=[feature_names[i] for i in sorted_idx3])
axes[1, 0].set_title('Best SVR', fontsize=12, fontweight='bold')
axes[1, 0].set_xlabel('Importance')
axes[1, 0].set_xlim([x_min, x_max])

axes[1, 1].boxplot(perm_result4.importances[sorted_idx4].T, vert=False,
    ↪ tick_labels=[feature_names[i] for i in sorted_idx4])
axes[1, 1].set_title('Best RF', fontsize=12, fontweight='bold')
axes[1, 1].set_xlabel('Importance')
axes[1, 1].set_xlim([x_min, x_max])

plt.suptitle('Permutation Importance Comparison Across Models', fontsize=14,
    ↪ fontweight='bold', y=1)
plt.tight_layout()
plt.savefig(path + 'permutation_importance.png', dpi=300, bbox_inches='tight')
plt.show()
```

A.2.4 Distribution Analysis

This section details the Monte Carlo sampling from parametric distributions and the subsequent fitting of the surrogate model output to various statistical distributions.

```
def sample_lognorm(mean, std, size):
    s = np.sqrt(np.log(1 + (std**2) / (mean**2)))
    scale = mean**2 / math.sqrt(mean**2 + std**2)
    return stats.lognorm.rvs(s=s, scale=scale, size=size)

def sample_uniform(mean, std, size):
    loc = mean - np.sqrt(3) * std
    scale = 2 * np.sqrt(3) * std
    return stats.uniform.rvs(loc=loc, scale=scale, size=size)

def sample_gumble(mean, std, size):
    em_constant = round((1. - math.gamma(1 + 1.e-8)) * 1.e14) * 1.e-6
    scale = np.sqrt(6) * std / np.pi
    loc = mean - em_constant * scale
    return stats.gumbel_r.rvs(loc=loc, scale=scale, size=size)
```

The implementation below calculates descriptive statistics and generates the Histogram/PDF, ECDF, and Q-Q plots for the maximum stress (σ_{max}).

```
# Monte Carlo Simulation (100,000 samples)
np.random.seed(RANDOM_STATE)
size = 100000

E_mem_samples = sample_lognorm(0.6, 0.09, size)
nu_mem_samples = sample_uniform(0.4, 0.0115, size)
sigma_mem_samples = sample_lognorm(4, 0.8, size)
sigma_edg_samples = sample_lognorm(353.678, 70.735, size)
sigma_sup_samples = sample_lognorm(400.834, 80.166, size)
f_mem_samples = sample_gumble(0.4, 0.12, size)

# Predict outcomes using the Best MLP Surrogate Model
# GET THE MODEL
X = df[['E_mem', 'nu_mem', 'sigma_mem', 'sigma_edg', 'sigma_sup', 'f_mem']].
    ↪ values
y = df['sigma_mem_max'].values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

model = MLPRegressor(hidden_layer_sizes=(64, 32), alpha=0.0001,
    ↪ learning_rate_init=0.01, max_iter=10000, random_state=RANDOM_STATE)
model.fit(X_scaled, y)
```

```
22
23 X_samples = np.column_stack((E_mem_samples, nu_mem_samples, sigma_mem_samples,
       ↪ sigma_edg_samples, sigma_sup_samples, f_mem_samples))
24 X_samples_scaled = scaler.transform(X_samples)
25 sigma_mem_max_samples = model.predict(X_samples_scaled)
26 # Distribution Fitting and Descriptive Statistics (Normal, Lognormal, Gumbel,
       ↪ desc_stats = []
27
28 # samples
29 data = sigma_mem_max_samples
30 mean_data = np.mean(data)
31 std_data = np.std(data, ddof=1)
32 skew_data = skew(data)
33 kurtosis_data = kurtosis(data)
34 q1_data = np.percentile(data, 25)
35 median_data = np.median(data)
36 q3_data = np.percentile(data, 75)
37 desc_stats.append(['samples (sigma_mem_max)', mean_data, std_data, skew_data,
       ↪ kurtosis_data, q1_data, median_data, q3_data])
38
39 # normal distribution
40 loc_norm, scale_norm = stats.norm.fit(data)
41 normal = stats.norm(loc=loc_norm, scale=scale_norm)
42 mean_norm = normal.mean()
43 std_norm = normal.std()
44 skew_norm = normal.stats(moments='s')
45 kurtosis_norm = normal.stats(moments='k')
46 q1_norm = normal.ppf(0.25)
47 median_norm = normal.median()
48 q3_norm = normal.ppf(0.75)
49 test_norm = kstest(data, stats.norm.cdf, args=(loc_norm, scale_norm))
50 desc_stats.append(['normal', mean_norm, std_norm, skew_norm, kurtosis_norm,
       ↪ q1_norm, median_norm, q3_norm, test_norm.statistic])
51
52 # lognormal distribution
53 sigma_lognorm, loc_lognorm, scale_lognorm = stats.lognorm.fit(data)
54 lognormal = stats.lognorm(s=sigma_lognorm, loc=loc_lognorm, scale=scale_lognorm
       ↪ )
55 mean_lognorm = lognormal.mean()
56 std_lognorm = lognormal.std()
57 skew_lognorm = lognormal.stats(moments='s')
58 kurtosis_lognorm = lognormal.stats(moments='k')
59 q1_lognorm = lognormal.ppf(0.25)
60 median_lognorm = lognormal.median()
61 q3_lognorm = lognormal.ppf(0.75)
62 test_lognorm = kstest(data, stats.lognorm.cdf, args=(sigma_lognorm, loc_lognorm
       ↪ , scale_lognorm))
```

```
63 desc_stats.append(['lognormal', mean_lognorm, std_lognorm, skew_lognorm,
       ↪ kurtosis_lognorm, q1_lognorm, median_lognorm, q3_lognorm, test_lognorm.
       ↪ statistic])
64
65 # gumbel distribution
66 loc_gumbel, scale_gumbel = stats.gumbel_r.fit(data)
67 gumbel = stats.gumbel_r(loc=loc_gumbel, scale=scale_gumbel)
68 mean_gumbel = gumbel.mean()
69 std_gumbel = gumbel.std()
70 skew_gumbel = gumbel.stats(moments='s')
71 kurtosis_gumbel = gumbel.stats(moments='k')
72 q1_gumbel = gumbel.ppf(0.25)
73 median_gumbel = gumbel.median()
74 q3_gumbel = gumbel.ppf(0.75)
75 test_gumbel = kstest(data, stats.gumbel_r.cdf, args=(loc_gumbel, scale_gumbel))
76 desc_stats.append(['gumbel', mean_gumbel, std_gumbel, skew_gumbel,
       ↪ kurtosis_gumbel, q1_gumbel, median_gumbel, q3_gumbel, test_gumbel.
       ↪ statistic])
77
78 # gamma distribution
79 a_gamma, loc_gamma, scale_gamma = stats.gamma.fit(data)
80 gamma = stats.gamma(a_gamma, loc=loc_gamma, scale=scale_gamma)
81 mean_gamma = gamma.mean()
82 std_gamma = gamma.std()
83 skew_gamma = gamma.stats(moments='s')
84 kurtosis_gamma = gamma.stats(moments='k')
85 q1_gamma = gamma.ppf(0.25)
86 median_gamma = gamma.median()
87 q3_gamma = gamma.ppf(0.75)
88 test_gamma = kstest(data, stats.gamma.cdf, args=(a_gamma, loc_gamma,
       ↪ scale_gamma))
89 desc_stats.append(['gamma', mean_gamma, std_gamma, skew_gamma, kurtosis_gamma,
       ↪ q1_gamma, median_gamma, q3_gamma, test_gamma.statistic])
90
91 # weibull distribution
92 c_weibull, loc_weibull, scale_weibull = stats.weibull_min.fit(data)
93 weibull = stats.weibull_min(c_weibull, loc=loc_weibull, scale=scale_weibull)
94 mean_weibull = weibull.mean()
95 std_weibull = weibull.std()
96 skew_weibull = weibull.stats(moments='s')
97 kurtosis_weibull = weibull.stats(moments='k')
98 q1_weibull = weibull.ppf(0.25)
99 median_weibull = weibull.median()
100 q3_weibull = weibull.ppf(0.75)
101 test_weibull = kstest(data, stats.weibull_min.cdf, args=(c_weibull, loc_weibull
       ↪ , scale_weibull))
102 desc_stats.append(['weibull', mean_weibull, std_weibull, skew_weibull,
```

```
    ↪ kurtosis_weibull, q1_weibull, median_weibull, q3_weibull, test_weibull.
    ↪ statistic])

desc_stats = pd.DataFrame(desc_stats, columns=['name', 'mean', 'std', 'skew', '
    ↪ excess_kurtosis', 'q1', 'median', 'q3', 'ks_test'])
desc_stats

# Visualization: Histogram and PDF Overlays
x_min, x_max = np.min(data), np.max(data)
x_padding = 0.05 * (x_max - x_min)
x_min, x_max = x_min - x_padding, x_max + x_padding
x = np.linspace(x_min, x_max, size)

plt.figure(figsize=(15, 5))
plt.hist(data, bins=100, density=True, alpha=0.7, edgecolor='black')
plt.plot(x, stats.norm.pdf(x, loc=loc_norm, scale=scale_norm), 'r-', alpha=0.6,
    ↪  label='Normal')
plt.plot(x, stats.lognorm.pdf(x, s=sigma_lognorm, loc=loc_lognorm, scale=
    ↪ scale_lognorm), 'g-', alpha=0.6, label='Lognormal')
plt.plot(x, stats.gumbel_r.pdf(x, loc=loc_gumbel, scale=scale_gumbel), 'm-',
    ↪ alpha=0.6, label='Gumbel')
plt.plot(x, stats.gamma.pdf(x, a_gamma, loc=loc_gamma, scale=scale_gamma), 'y-'
    ↪ , alpha=0.6, label='Gamma')
plt.plot(x, stats.weibull_min.pdf(x, c_weibull, loc=loc_weibull, scale=
    ↪ scale_weibull), 'c-', alpha=0.6, label='Weibull')
plt.title('Histogram of maximum stress', fontsize=12, fontweight='bold')
plt.xlim([x_min, 0.7*x_max])
plt.legend(loc=1)

plt.tight_layout()
plt.savefig(path + 'histogram_max_stress.png', dpi=300, bbox_inches='tight')
plt.show()
```

```
# Visualization: ECDF and Q-Q Plots
fig = plt.figure(figsize=(15, 9))
spec = fig.add_gridspec(2, 5)

ax00 = fig.add_subplot(spec[0, 0])
ax00.ecdf(data, label='ECDF')
ax00.plot(x, stats.norm.cdf(x, loc=loc_norm, scale=scale_norm), 'r--', alpha
    ↪ =0.6, label='Normal CDF')
ax00.set_title('ECDF vs Normal CDF', fontsize=12, fontweight='bold')
ax00.set_xlim([x_min, 6000])
ax00.legend(loc=1)

ax01 = fig.add_subplot(spec[0, 1])
ax01.ecdf(data, label='ECDF')
```

```
ax01.plot(x, stats.lognorm.cdf(x, s=sigma_lognorm, loc=loc_lognorm, scale=
    ↪ scale_lognorm), 'g--', alpha=0.6, label='Lognormal CDF')
ax01.set_title('ECDF vs Lognormal CDF', fontsize=12, fontweight='bold')
ax01.set_xlim([x_min, 6000])
ax01.legend(loc=1)

ax02 = fig.add_subplot(spec[0, 2])
ax02.ecdf(data, label='ECDF')
ax02.plot(x, stats.gumbel_r.cdf(x, loc=loc_gumbel, scale=scale_gumbel), 'm--',
    ↪ alpha=0.6, label='Gumbel CDF')
ax02.set_title('ECDF vs Gumbel CDF', fontsize=12, fontweight='bold')
ax02.set_xlim([x_min, 6000])
ax02.legend(loc=1)

ax03 = fig.add_subplot(spec[0, 3])
ax03.ecdf(data, label='ECDF')
ax03.plot(x, stats.gamma.cdf(x, a=a_gamma, loc=loc_gamma, scale=scale_gamma), '
    ↪ m--', alpha=0.6, label='Gamma CDF')
ax03.set_title('ECDF vs Gamma CDF', fontsize=12, fontweight='bold')
ax03.set_xlim([x_min, 6000])
ax03.legend(loc=1)

ax04 = fig.add_subplot(spec[0, 4])
ax04.ecdf(data, label='ECDF')
ax04.plot(x, stats.weibull_min.cdf(x, c=c_weibull, loc=loc_weibull, scale=
    ↪ scale_weibull), 'm--', alpha=0.6, label='Weibull CDF')
ax04.set_title('ECDF vs Weibull CDF', fontsize=12, fontweight='bold')
ax04.set_xlim([x_min, 6000])
ax04.legend(loc=1)

ax10 = fig.add_subplot(spec[1, 0])
res = stats.probplot(data, dist=stats.norm, sparams=(loc_norm, scale_norm),
    ↪ plot=ax10)
ax10.get_lines()[0].set_markersize(1)
ax10.set_xlabel('')
ax10.set_ylabel('')
ax10.set_title('Normal Q-Q plot', fontsize=12, fontweight='bold')

ax11 = fig.add_subplot(spec[1, 1])
res = stats.probplot(data, dist=stats.lognorm, sparams=(sigma_lognorm,
    ↪ loc_lognorm, scale_lognorm), plot=ax11)
ax11.get_lines()[0].set_markersize(1)
ax11.set_xlabel('')
ax11.set_ylabel('')
ax11.set_title('Lognormal Q-Q plot', fontsize=12, fontweight='bold')

ax12 = fig.add_subplot(spec[1, 2])
```

```
res = stats.probplot(data, dist=stats.gumbel_r, sparams=(loc_gumbel,
    ↪ scale_gumbel), plot=ax12)
ax12.get_lines()[0].set_markersize(1)
ax12.set_xlabel('')
ax12.set_ylabel('')
ax12.set_title('Gumbel Q-Q plot', fontsize=12, fontweight='bold')

ax13 = fig.add_subplot(spec[1, 3])
res = stats.probplot(data, dist=stats.gamma, sparams=(a_gamma, loc_gamma,
    ↪ scale_gamma), plot=ax13)
ax13.get_lines()[0].set_markersize(1)
ax13.set_xlabel('')
ax13.set_ylabel('')
ax13.set_title('Gamma Q-Q plot', fontsize=12, fontweight='bold')

ax14 = fig.add_subplot(spec[1, 4])
res = stats.probplot(data, dist=stats.weibull_min, sparams=(c_weibull,
    ↪ loc_weibull, scale_weibull), plot=ax14)
ax14.get_lines()[0].set_markersize(1)
ax14.set_xlabel('')
ax14.set_ylabel('')
ax14.set_title('Weibull Q-Q plot', fontsize=12, fontweight='bold')

plt.suptitle('Distribution Analysis', fontsize=14, fontweight='bold', y=1)
plt.tight_layout()
plt.savefig(path + 'distribution_analysis.png', dpi=300, bbox_inches='tight')
plt.show()
```