Problem Chosen	2024 ShuWei Cup Summary Sheet	Team Control Number
B		2024111130871

Spatial Variable Prediction Problem Based on Co-Kriging and Its Derived Algorithms

Summary

The synergistic kriging algorithm is an algorithm that can use covariates to effectively estimate the main spatial variables when the sampling volume of the main spatial variables is insufficient. This paper mainly focuses on the use of the synergistic kriging algorithm in combination with other algorithms to reduce its computational difficulty and improve its estimation accuracy.

For question 1, using **Kriging Interpolation**, the spatial variables of the F1-target are randomly sampled, and the Kriging Algorithm is applied to estimate the value of the unsampled points by using the value of the sampled points, and on the basis of which, several times of sampling with different sampling rates (1%-50%, with a step size of 1%) are carried out, to explore the relationship between the amount of sampling and the estimation error. It is found that when the sampling rate reaches 20%, the error begins to tend to remain at a low and stable value.

For question 2, the correlation between the covariates and the F1-target was investigated using the data in Attachment 1. By combining the linear and non-linear correlations through a preliminary analysis using the **Spearman** correlation coefficient, it was found that **F1-collaborative-variable1** had a relatively high correlation with the F1target, and this was then This view was further corroborated by **Random Forest Regression Analysis**.

For Problem 3, based on the analysis of Problem 2, F1-collaborative-variable1 is mainly used for the analysis, and **Collaborative Kriging Algorithm** is used to estimate for the sampling points, and the same method of multiple random sampling is adopted, which is used to explore the relationship between the estimation error and the sampling rate. To further analyse the algorithm properties, the Co-Kriging algorithm is found to have significant accuracy by comparing with **RFK model**, **Smooth Bivariate Spline Algorithm**, **InearNDInterpolator Algorithm**, while the RFK model reduces the computational difficulty with better accuracy.

For Problem 4, in the face of a very small effective sampling size (1.24%) and the problem of low correlation between covariates and the main variable, a combination of Random Forest and Kriging algorithms is used for estimation (**RFK model**), an algorithm that effectively solves the problem of the difficulty of calculating covariates for covariates that have low correlation, and at the same time ensures the accuracy of the estimation.

Key word: Kriging Interpolation , Spearman , Random Forest Regression , Co-Kriging , Smooth Bivariate Spline , InearNDInterpolator , RFK model

Content

1. Introduction

1.1 Background

In spatial statistics, spatial variables are correlated between sampling points and their values show a certain trend, therefore, Kriging interpolation is commonly used to estimate the value of the unsampled unknown. In practical engineering the same spatial location can be sampled in multiple dimensions, and these variables may have different physical meanings but show a high degree of correlation. Therefore, in practical applications, spatial variables that are easy to sample can be used to infer information about spatial variables that are not easy to measure in practical engineering, but have important values. This paper focuses on the Co-Kriging algorithm and combines several algorithms to estimate the main variables using covariates.

1.2 Work

Based on the background of the above topic, the following questions are required to be addressed:

- **Q1**: Using the data from Attachment 1to study the variation patterns of one of the spatial variables (F1-target variable). And complete the following tasks.
  (1) Randomly and uniformly resample the target variable, use the resampled values to estimate the values at the unsampled points, and present the results in the form of a contour plot.
  (2) Vary the sampling size and explore the relationship between sampling size and estimation error.
- **Q2**: Using the data from Attachment 1, explore the correlation between the target variable and the covariates and select two covariates to estimate the target variable
- **Q3**: The requirement is to investigate the pattern of change in the spatial variable (F1-target variable) based on the covariates chosen in Q2. The requirements are as follows:
  (1) Randomly and uniformly resample the selected covariates and the target variable, use the resampled values to estimate the values of the unsampled spatial variables, and present the results in the form of contours.
  (2) Vary the size of the sample size and explore the relationship between sample size and estimation error.
  (3) Select at least two methods for comparison.
- **Q4**: Based on Q3, choose the best estimator to estimate the trend of the target variable (F2-target variable) and present the results in the form of a contour plot.

2. Problem Analysis

Analysis of question one For question 1, the Kriging interpolation method is used to randomly sample the spatial variables of the F1-target, and the Kriging algorithm is applied to estimate the value of the unsampled points by using the value of the sampled points, and on the basis of which a number of sampling with different sampling rates are carried out to explore the relationship between the sampling volume and the estimation error.

Analysis of question two Problem 2 requires that two of the four covariates be selected as the estimated covariates for the F1-target. The data should first be pre-processed to standardise the data and eliminate the effect of the scale of the different covariates. Spearman's correlation coefficient should be used to initially analyse the correlation between the covariates and the F1-target, followed by Random Forest regression analysis to determine the selected covariates.

Analysis of question three Based on Problem 2, one or two covariates were selected to estimate the F1-target information by the Co-kriging algorithm and several algorithms such as SmoothBivariateSpline, LinearNDInterpolator ,RFK model were used to compare and analyse the results. The relationship between sample size and estimation error is explored through multiple random sampling to analyse the sensitivity of Co-kriging to error.

Analysis of question four When analysing the information in Annex 2, it was found that the sampling rate of F2-target was only 1.24%, which could be estimated based on the data of the third question with a large degree of error, and the Spearman correlation coefficients of the four covariates were not high enough, which would lead to a large degree of error when using the synergistic kriging estimation directly. Therefore, in order to improve the accuracy of the estimation, the Random Forest algorithm is introduced to predict the preliminary spatial values, and the Kriging algorithm is combined to estimate all the spatial vectors. This can effectively solve the problem of difficulty in calculating the covariance of the kriging algorithm, and ensure a certain degree of accuracy.

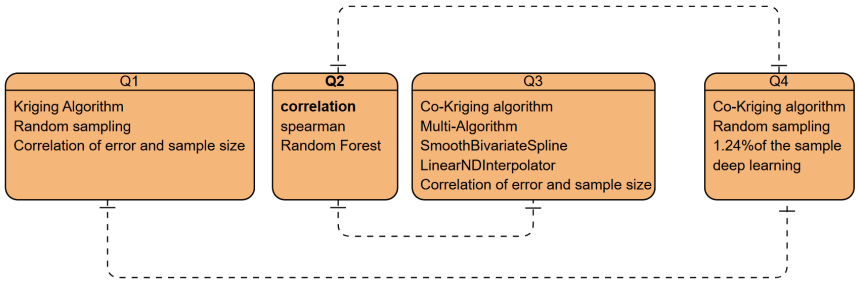


Figure 1 Flowchart of all issues

3. Symbol and Assumptions

3.1 Symbol Description

Symbol	Description
D	Observed dataset containing locations $\mathbf{s}_i$ and observed values $z(\mathbf{s}_i)$.
$\mathbf{s}_i$	Location vector of the i-th sampling point.
$z(\mathbf{s}_i)$	Observed value at sampling point $\mathbf{s}_i$.
$\mathbf{s}_0$	Location where the interpolation prediction is to be made.
$\gamma(h)$	Variogram function representing the variability of observed values at distance h.
$N(h)$	Number of point pairs separated by distance h.
K	Kriging system matrix containing covariance or variogram values.
$\mathbf{b}$	Right-hand vector of the Kriging system representing covariance or variogram values between the target and sampling points.
$\boldsymbol{\lambda}$	Kriging weight vector used for interpolation.
μ	Lagrange multiplier ensuring the weights sum to 1.
$z^*(\mathbf{s}_0)$	Interpolated value at the prediction location $\mathbf{s}_0$.
n_{trees}	Number of decision trees in the random forest.
$d_{\max}$	Maximum depth of each decision tree.
I_{feature}	Feature importance score, measuring each feature's contribution to the model prediction.
$Z_i(s)$	Observed value of the covariate Z_i at location s.
$\gamma_{ij}(h)$	Cross-variogram between covariates Z_i and Z_j.
$\Gamma(h)$	Variogram matrix containing all auto-variogram and cross-variogram values.
C_{ij}	Covariance matrix element between covariates Z_i and Z_j.
c_{i0}	Covariance value of covariate Z_i at the interpolation point $\mathbf{s}_0$.
$\sigma^2(s_0)$	Variance of the interpolation prediction, quantifying prediction uncertainty.

3.2 Fundamental assumptions

1. Assuming covariates exist in space and are easily measured relative to the target variable

2. It is assumed that information about points in space is continuous and present, and that there is no missing spatial information in a given dimension.
3. It is assumed that in kriging interpolation, all observed data point errors are independent. Each observation is considered a random variable with a known covariance and there is no correlation between these errors.
4. Kriging interpolation assumes that the interpolation results are unbiased, i.e., the expectation of the predicted value is equal to the true value. Specifically, given known data points, the mean of the predicted values generated by the Kriging method should equal the mean of the true values.

4. Establishment and solution of the model

4.1 The model of Problem 1

4.1.1 Introduce of Algorithm 1

Algorithm 1:Kriging Algorithm

Input: Observed dataset $D = \{(\mathbf{s}_i, z(\mathbf{s}_i)) \mid i = 1, 2, \ldots, N\}$, Location of interest $\mathbf{s}_0$, Variogram model parameters.

Output: Predicted value $z^*(\mathbf{s}_0)$.

Step1:Compute Experimental Variogram:

i For all pairwise distances $h = \|\mathbf{s}_i - \mathbf{s}_j\|$, compute:

$$\gamma(h) = \frac{1}{2N(h)} \sum_{N(h)} [z(\mathbf{s}_i) - z(\mathbf{s}_j)]^2 \tag{1}$$

ii Aggregate results into distance bins to create the experimental variogram.

Step2:Fit Theoretical Variogram Model:

i Choose a variogram model (e.g., spherical, exponential, Gaussian).

ii Fit the model parameters (e.g., nugget, sill, range) to the experimental variogram using least squares or other optimization techniques.

Step3:Solve Kriging System:

i Construct the Kriging matrix K:

$$K = \begin{bmatrix} \gamma(\mathbf{s}_1, \mathbf{s}_1) & \cdots & \gamma(\mathbf{s}_1, \mathbf{s}_N) & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \gamma(\mathbf{s}_N, \mathbf{s}_1) & \cdots & \gamma(\mathbf{s}_N, \mathbf{s}_N) & 1 \\ 1 & \cdots & 1 & 0 \end{bmatrix} \tag{2}$$

ii Construct the right-hand side vector $\mathbf{b}$:

$$\mathbf{b} = \begin{bmatrix} \gamma(\mathbf{s}_1, \mathbf{s}_0) \\ \vdots \\ \gamma(\mathbf{s}_N, \mathbf{s}_0) \\ 1 \end{bmatrix} \tag{3}$$

iii Solve for weights $\boldsymbol{\lambda}$ and Lagrange multiplier μ:

$$[K] \cdot \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_N \\ \mu \end{bmatrix} = \mathbf{b} \tag{4}$$

Step4:Predict Value at $\mathbf{s}_0$:

i Compute the predicted value $z^*(\mathbf{s}_0)$:

$$z^*(\mathbf{s}_0) = \sum_{i=1}^{N} \lambda_i z(\mathbf{s}_i) \tag{5}$$

ii Optionally compute the Kriging variance for uncertainty estimation:

$$\sigma_K^2 = \sum_{i=1}^{N} \lambda_i \gamma(\mathbf{s}_i, \mathbf{s}_0) + \mu \tag{6}$$

End.

4.1.2 solution of the Problem 1

An initial visualization of the F1-target data was first performed to observe the three-dimensional distribution of the data (Figure 2)

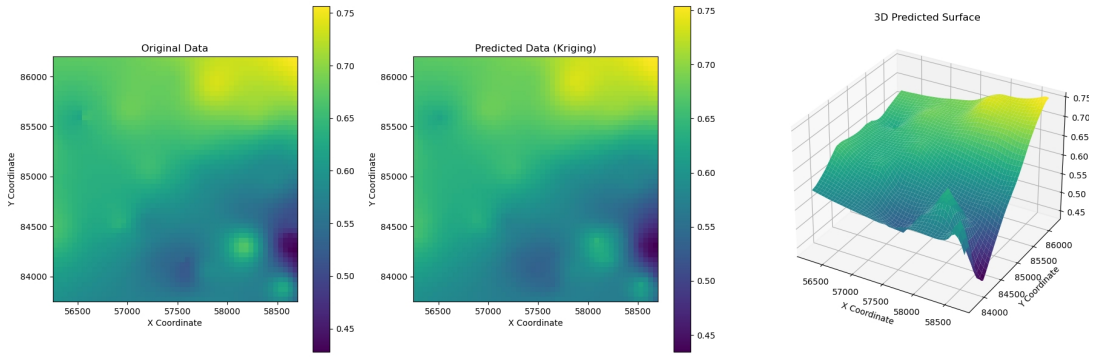


Figure 2 Line graph of sampling rate vs. error

The F1-target data is a 266*266 grid data, where the length is 51250-64500, the width is 78750- 92000, and 50 is a measurement spacing,which means the observation dataset is

$$D = \{(\mathbf{s}_i, z(\mathbf{s}_i)) \mid i = 1, 2, \ldots, 266\} \tag{7}$$

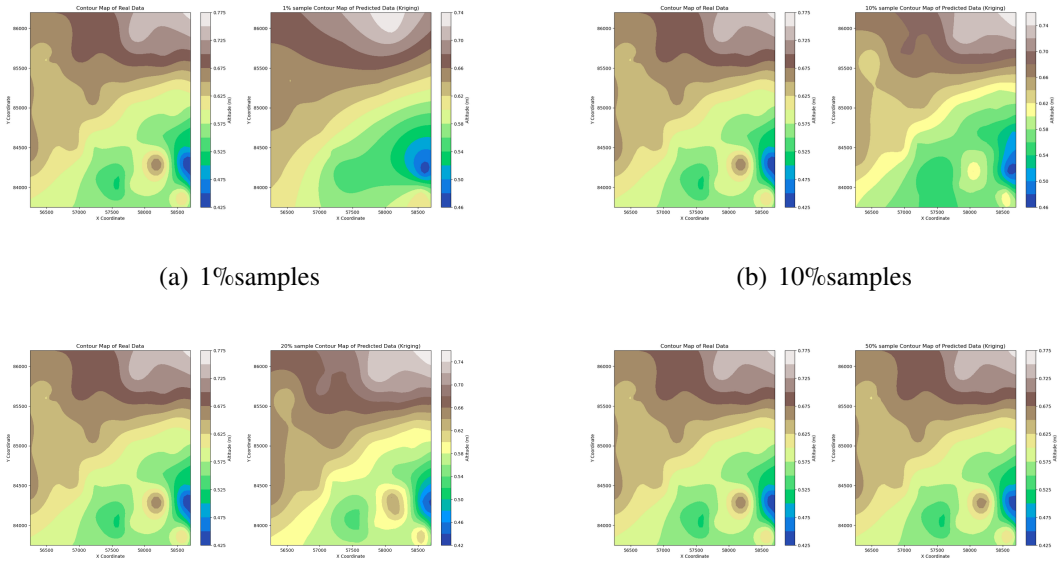
where $\mathbf{s}_i = (x, y)$ is the coordinate position information. and $(x, y), x \in [51250, 64500], y \in [78750, 92000]$.

Calculate all distances h as the Euclidean distance between $\mathbf{s}_i$ and $\mathbf{s}_j$.

Substitute $N = 266$ into Eq. (1), construct the variance function, choose the fitted theoretical variance model that best meets the question as (spherical model), construct the Kriging matrix K, the right end vector $\mathbf{b}$,Eqs. (2)(3), solve for the weights $\boldsymbol{\lambda}$ and μ(4), and finally compute the predicted values $\mathbf{s}_0$ (5).

Subsequently vary the sample size of the random sample to explore the relationship between sample size and error.

As Figure 3 shows the isogram of the prediction using the kriging algorithm at a sampling rate of 10%, with the difference between the original figure, it can be seen from the figure that the estimated information is already very comprehensive at a sampling rate of 20%.



(a) 1%samples (b) 10%samples

(c) 20%samples (d) 50%samples

Figure 3 Contour map of 1%,10%,20%,50% samples

After varying different sampling rates for sampling and evaluating the accuracy, the results are shown in Fig. 4,It can be found that the estimation bias starts to stabilise after the sampling rate reaches 10% and maintains a very low error.

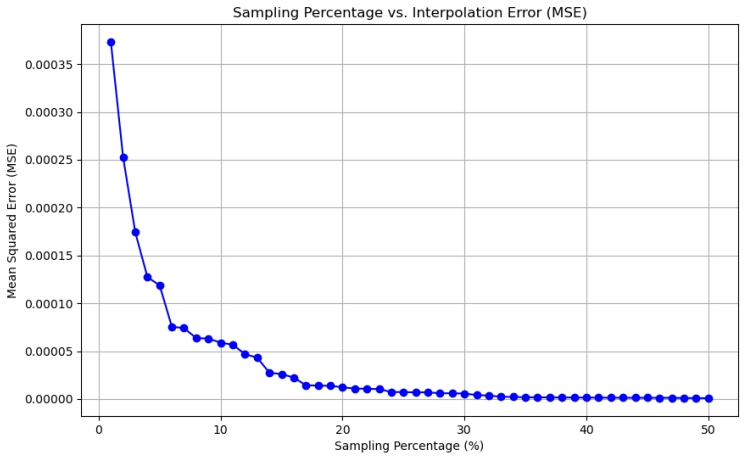


Figure 4 Line graph of sampling rate vs. error

4.2 The model of Problem 2

4.2.1 Introduce of Algorithm 2

Algorithm 2:**Random Forest Regression**

Input

- Dataset: (X, y), where:
  - $X \in \mathbb{R}^{n \times m}$: Feature matrix with n samples and m features.
  - $y \in \mathbb{R}^n$: Target vector representing the true value for each sample.
- Hyperparameters:
  - n_{trees}: Number of decision trees in the random forest.
  - $d_{\max}$: Maximum depth of each decision tree.

Output

- Model Prediction Results: Target values $\hat{y}$.
- Feature Importance: Importance scores for each feature I_{feature}.

Step 1: Data Preparation

- Split the dataset into training and testing sets:

$$X_{\text{train}}, X_{\text{test}}, y_{\text{train}}, y_{\text{test}} = \text{train\_test\_split}(X, y, \text{test\_size}). \tag{8}$$

Step 2: Random Forest Training

- Train n_{trees} decision trees:
  - Use the Bagging method to randomly sample with replacement from the training set to generate multiple subsets $\{X_{\text{train}}^{(i)}, y_{\text{train}}^{(i)}\}$.

– Train a decision tree T_i on each subset to minimize node impurity (e.g., MSE):

$$T_i = \text{train\_decision\_tree}(X_{\text{train}}^{(i)}, y_{\text{train}}^{(i)}). \tag{9}$$

Step 3: Prediction

- For test data X_{test}, compute the average prediction from all decision trees:

$$\hat{y}_{\text{test}} = \frac{1}{n_{\text{trees}}} \sum_{i=1}^{n_{\text{trees}}} T_i(X_{\text{test}}). \tag{10}$$

Step 4: Feature Importance Calculation

- Compute the total reduction in node impurity contributed by each feature across all trees:

$$I_j = \frac{1}{n_{\text{trees}}} \sum_{i=1}^{n_{\text{trees}}} \Delta I_{T_i}(j), \tag{11}$$

  where $\Delta I_{T_i}(j)$ is the reduction in node impurity for feature j in tree T_i.
- Normalize the importance scores for all features:

$$I_{\text{feature}}(j) = \frac{I_j}{\sum_{k=1}^{m} I_k}. \tag{12}$$

Step 5: Output

- Return the predicted values for the test set $\hat{y}_{\text{test}}$.
- Return feature importance scores $\{I_{\text{feature}}(1), I_{\text{feature}}(2), \ldots, I_{\text{feature}}(m)\}$.

4.2.2 solution of the Problem 2

As shown in Figure 5, the first preliminary analysis of the different covariates, the use of Spearman correlation coefficients, different from Person correlation, Spearman correlation coefficients reflect not only reflect the degree of linear correlation but also to a certain extent, can reflect the correlation of non-linear correlations

The Spearman rank correlation coefficient begins by transforming the raw data into a rank, i.e., each data point is sorted by size and assigned a ranking. Features of the rank transformation include:

- rank reflects only the order between data points, independent of their specific values.
- The use of the rank eliminates the linear scale relationship between the variables and retains only their ordering information.

For the samples $(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)$, calculate the order of X and Y:

$$R(X_i), \quad R(Y_i), \quad i = 1, 2, \ldots, n.$$

Correlation calculation by Eq:

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)}, \tag{13}$$

Among them:

- $d_i = R(X_i) - R(Y_i)$: the rank difference of the i th sample.
- n: total number of samples.

The Spearman correlation coefficient essentially measures the monotonic relationship between two variables without assuming a linear relationship between the variables. It therefore has a high robustness to noise and outliers

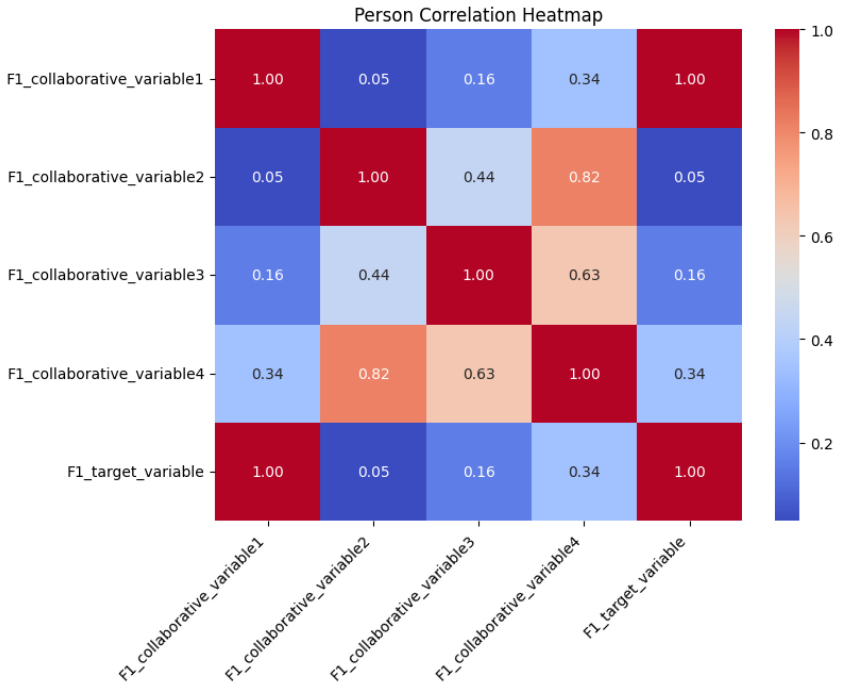


Figure 5 Heatmap of correlation between F1-target and F1-covariates

Then, using Random Forest for feature importance assessment, based on the Attachment 1 information, the F1-collaborative-variable1, F1-collaborative-variable2, F1-collaborative-variable3, and F1- collaborative-variable4 information to form the feature matrix $X \in \mathbb{R}^{n\times m}$, and integrate the information of F1-target to form the target vector matrix $y \in \mathbb{R}^n$, and then divide the dataset into the training set and the test set to perform the random forest Training.

Calculate the sum of reduced node impurity for each feature across all decision trees(11):

$$I_j = \frac{1}{n_{\text{trees}}} \sum_{i=1}^{n_{\text{trees}}} \Delta I_{T_i}(j),$$

Where $\Delta I_{T_i}(j)$ is the node impurity reduction of feature j in decision tree T_i.

Normalise the importance scores of all features (12):

$$I_{\text{feature}}(j) = \frac{I_j}{\sum_{k=1}^{m} I_k}.$$

Figure 6 is a plot of the results of the characteristic importance analysis using Random Forest regression, presenting the characteristic importance of the four covariates, and the data show that F1-collaborative-variable1 is a set of variables that can reflect the F1-target well, and that the other variables may or may not serve as a complement to the F1-target.

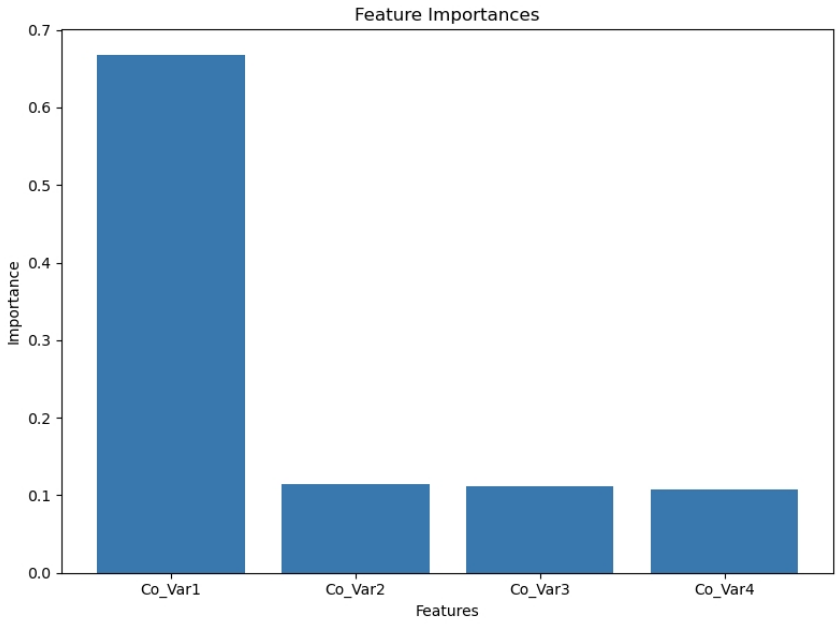


Figure 6 Importance of covariate characteristics

4.3 The model of Problem 3

4.3.1 Introduce of Algorithm 3

Algorithm 3:**Co-kriging Algorithm**

Step1: Problem Definition and Data Preparation - Input Data:

- Sampling point coordinates and observed values for the primary variable $Z_1(s)$.
- Sampling point coordinates and observed values for covariates $Z_2(s), Z_3(s), \ldots, Z_p(s)$.

- Objective: Estimate the value of the primary variable $Z_1(s_0)$ at an unobserved location s_0.

Step2: Constructing the Variogram Model

i Calculate the auto-variogram for each variable:

$$\gamma_{ii}(h) = \frac{1}{2N(h)} \sum_{N(h)} [Z_i(s) - Z_i(s+h)]^2 \tag{14}$$

where $N(h)$ is the set of point pairs separated by distance h.

ii Calculate the cross-variogram between variables:

$$\gamma_{ij}(h) = \frac{1}{2N(h)} \sum_{N(h)} [Z_i(s) - Z_i(s+h)] \left[Z_j(s) - Z_j(s+h)\right] \tag{15}$$

iii Joint variogram matrix:

$$\Gamma(h) = \begin{bmatrix} \gamma_{11}(h) & \gamma_{12}(h) & \cdots & \gamma_{1p}(h) \\ \gamma_{21}(h) & \gamma_{22}(h) & \cdots & \gamma_{2p}(h) \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{p1}(h) & \gamma_{p2}(h) & \cdots & \gamma_{pp}(h) \end{bmatrix} \tag{16}$$

Step3: Constructing the Kriging System

i Construct the weight matrix λ to estimate $Z_1(s_0)$.

ii Co-Kriging system of equations:

$$\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1p} & 1 \\ C_{21} & C_{22} & \cdots & C_{2p} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ C_{p1} & C_{p2} & \cdots & C_{pp} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_p \\ \mu \end{bmatrix} = \begin{bmatrix} c_{10} \\ c_{20} \\ \vdots \\ c_{p0} \\ 1 \end{bmatrix} \tag{17}$$

where:

- C_{ij}: Covariance matrix for variables Z_i and Z_j.
- c_{i0}: Covariance of variable Z_i at the interpolation point s_0.
- λ_i: Interpolation weights.
- μ: Lagrange multiplier ensuring the weights sum to 1.

iii Solve the system of equations to obtain weights λ_i and the Lagrange multiplier μ.

Step4: Interpolation Estimation Use the weights λ_i to estimate the primary variable $Z_1(s_0)$:

$$Z_1(s_0) = \sum_{i=1}^{n} \lambda_i Z_i(s) \tag{18}$$

The uncertainty of the interpolation can be quantified using the covariance matrix, yielding the prediction variance:

$$\sigma^2(s_0) = \gamma_{11}(0) - \sum_{i=1}^{n} \lambda_i c_{i0} \tag{19}$$

4.3.2 Introduce of Algorithm 4

Algorithm 4:**Smooth Bivariate Spline Algorithm**

Step1:Input Data

Given data points (x_i, y_i, z_i) $(i = 1, 2, \ldots, n)$:

- Independent variables: x_i, y_i
- Dependent variable: z_i, the function value at (x_i, y_i)
- Smoothing factor: s, controls the degree of smoothing
- Spline order: k_x, k_y, default is 3 (cubic spline)

Step2:Objective Function

Construct the bivariate spline function $B(x, y)$ by fitting data points using the least squares method:

$$Q = \sum_{i=1}^{n} (z_i - B(x_i, y_i))^2 + s \cdot \text{Penalty},$$

where:

- The first term represents the residual sum of squares.
- The second term enforces the smoothness constraint.

Step3:Bivariate Spline

The bivariate spline $B(x, y)$ is expressed as:

$$B(x, y) = \sum_{j=1}^{m} \sum_{k=1}^{m} c_{jk} N_{j,k}(x, y),$$

where:

- c_{jk} are the spline coefficients.
- $N_{j,k}(x, y)$ are the B-spline basis functions.

Step4:Output Results

After optimizing the spline coefficients c_{jk}, the resulting smooth bivariate spline function $B(x, y)$ can be used for:

- Interpolation: Compute values at arbitrary (x, y) points.
- Visualization: Generate spline surfaces.

4.3.3 Introduce of Algorithm 5

Algorithm 5:InearNDInterpolator Algorithm

Step1:Input Data

- Independent variables: $\mathbf{x} = \{(x_1, x_2, \ldots, x_n)\}$, representing coordinates in n-dimensional space.
- Dependent variable: $\mathbf{z} = \{z_1, z_2, \ldots, z_n\}$, values at the respective coordinates.
- Query points: $\mathbf{q} = \{(q_1, q_2, \ldots, q_m)\}$, where interpolation is performed.

Step2:Algorithm Steps

i Delaunay Triangulation

- Construct a Delaunay triangulation T over the input points $\mathbf{x}$.
- The triangulation divides the n-dimensional space into simplices (e.g., triangles in 2D, tetrahedra in 3D).

ii Linear Interpolation

- For a query point $\mathbf{q}$, find the simplex S in the triangulation T that contains $\mathbf{q}$.
- Represent $\mathbf{q}$ using barycentric coordinates relative to S:

$$\mathbf{q} = \sum_{i=1}^{n+1} \lambda_i \mathbf{x}_i,$$

where λ_i are the barycentric coordinates of $\mathbf{q}$.

- Compute the interpolated value z_q:

$$z_q = \sum_{i=1}^{n+1} \lambda_i z_i,$$

where z_i are the values at the vertices of S.

iii Handling Points Outside Convex Hull

If **q** lies outside the convex hull of **x**, interpolation is not defined, and the result defaults to 'NaN' (configurable).

Step3:Output

The algorithm produces interpolated values $\mathbf{z_q}$ at the query points **q**.

Step4:Complexity

- Delaunay triangulation: $O(n \log n)$ for n-dimensional space with m data points.
- Query complexity: $O(\log m)$ per point using the triangulation structure.

4.3.4 solution of the Problem 3

1 **Co-kriging**

Unlike the ordinary kriging algorithm, the collaborative kriging algorithm, in addition to constructing the variance function for the main variable $Z_1(s)$ (F1-target), also needs to compute its self-variance function for each variable including the covariates (14), compute the covariance function between the variables $\gamma_{ij}(h)$(15), and construct the matrix of the joint variance function $\Gamma(h)$ (16). By constructing the Kriging system equation (17), solve for the weights of each covariate λ_i, and use the weights λ_i to interpolate the main variable $Z_1(s)$ for estimation (18).

2 **Error Analysis**

For the collaborative variable F1-collaborative-variable1 is subjected to error analysis under different sampling rates, and the results are shown in Fig. 7,which shows that the error rate decreases rapidly with the sampling rate and basically stabilizes at a lower value at 20% out.

3 **Cross comparison of different algorithms**

SmoothBivariateSpline is based on spline function and least squares fitting. This method can be understood as interpolating in two dimensions using a spline function and introducing a smoothing parameter to achieve a least squares fit. In order to ensure the smoothness of the fitted surface, a smoothing factor sss is introduced, which balances the fit of the data and the smoothness of the surface by minimising the sum of squares of the residuals between the interpolated surface and the data points.

The **interpolation model** and algorithm used by LinearNDInterpolator is based on multidimensional Delaunay triangulation, where linear interpolation is performed on these

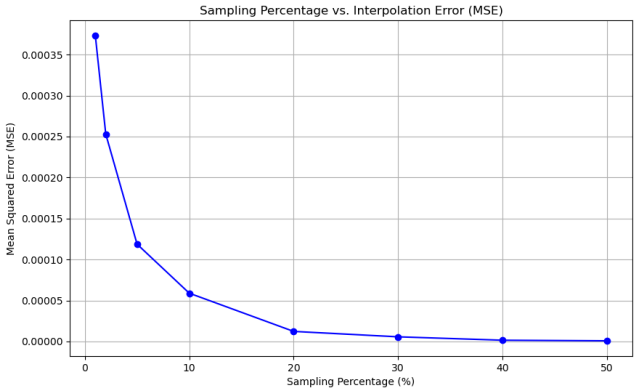


Figure 7 Error analysis with different sampling rates for covariates

triangulation cells to obtain values at arbitrary target positions LinearNDInterpolator first performs a Delaunay triangulation on the input interpolation points. Delaunay triangulation. This is a process that divides the set of points in a multidimensional space into a number of non-overlapping simplexes. This process divides the interpolated points into a number of triangles (or higher dimensional simplexes) such that the vertices of each simplex polygon are points in the input data. Once a simplex is found, the algorithm calculates the interpolated value of the target point by weighting and summing the values of the vertices of the simplex by a linear equation.

RFK model is a combination of random forest regression and kriging interpolation, the model is divided into two parts: 1. regression analysis of covariates using the random forest algorithm; 2. final estimation using kriging algorithm combined with prediction of regression analysis results. Such a method can effectively solve the problem of the difficulty of covariance calculation.

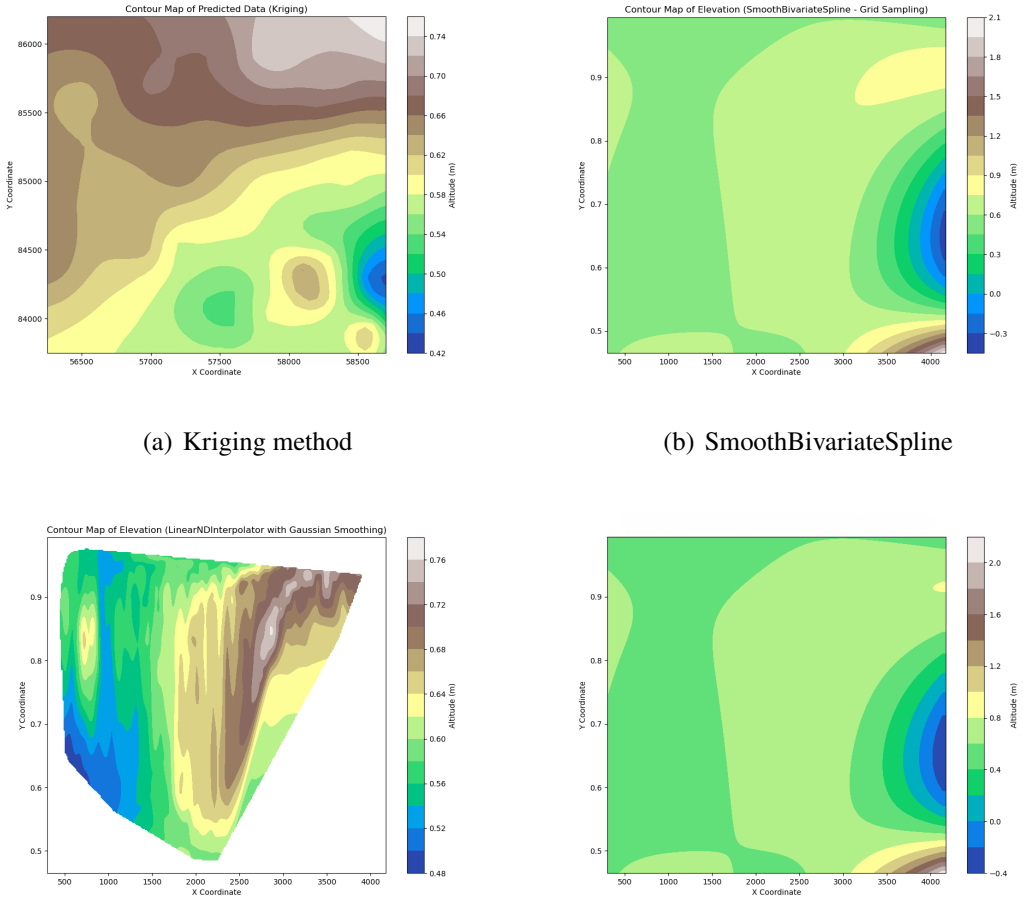
As Figure 8 shows the comparison of the four algorithms (Kriging, SmoothBivariateSpline, LinearNDInterpolator, RFK model), it can be seen that at the same sampling rate (20%), the kriging algorithm is more reflective of the real F1-target situation, followed by the RFK model (with some degree of distortion).

4.4 The model of Problem 4

4.4.1 Introduce of Algorithm 6

Algorithm 6:**RFK Algorithm**

RF is a classification and regression algorithm based on decision trees, where multiple random samples are obtained through bootstrap sampling, and corresponding decision trees are built from these samples to form a random forest. The method is suitable for solving classification and regression problems, and for regression problems, the mean of the predictions



(a) Kriging method

(b) SmoothBivariateSpline

(c) LinearNDInterpolator

(d) RFK model

Figure 8 Comparison of the three methods of Kriging, SmoothBivariateSpline, LinearNDInterpolator,RFK model

of all decision trees is taken as the final prediction.

Random Forest Kriging model (RFK model [1]) refers to the prediction of Kriging algorithm on the basis of random forest, the algorithm is divided into two steps: 1. predicting the value of the target variable with covariates through RF modelling.

2. separating the structured components of the residuals by kriging interpolation and superimposing them on the predicted values of the random forest model (20),The RF model predicted residual values were obtained by Eq.(21) to obtain.

Eq:

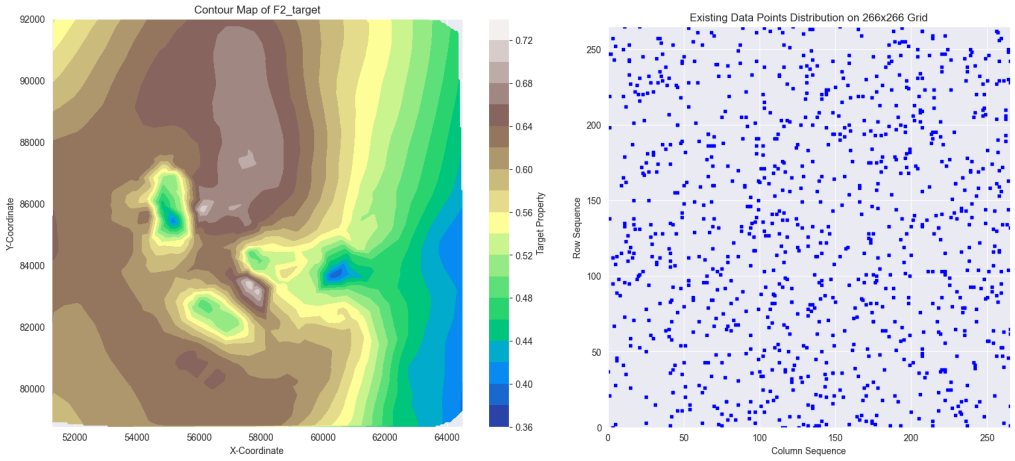
$$R(x_i) = B_{BF}(x_i) - B(x_i). \tag{20}$$

$$B_{RFOK/RFCK}(x_i) = B_{RF}(x_i) - R_{OK/CK}(x_i). \tag{21}$$

Eq: - $R(x_i)$ is the residual value of sample i;
- $B(x_i)$ is the observed value of the sample to be tested for the sample site i;
- OK means the ordinary kriging interpolation method;
- CK refers to the co-operative kriging interpolation method;
- $B_{RF}(x_i)$ is the predicted value of i based on the RF model;
- $B_{RFOK/RFCK}(x_i)$ is the predicted value obtained from the RFOK or RFCK model;
- $R_{OK/CK}(x_i)$ is the residual predicted value of i based on OK or CK.

4.4.2 solution of the Problem 4

Data pre-processing and pre-analysis First of all, a preliminary analysis of the data of F2-target is carried out first, and it is found that the effective sampling volume is only 878, accounting for 1.24% of the total sampling volume, and Fig. 9 demonstrates the contour plots according to the given data and the distribution of its given sampling points. It can be seen that the sampling points are more evenly distributed, which is very helpful for the later analysis.However, the correlation between the covariates and the F2-target is not very high (Figure 10),which poses some challenges for estimation.



(a) F2-target contour plot of known sampling points (b) Distribution of F2-target sampling points

Figure 9 Preliminary analysis of question 4

Solving results using RFK model The results of using the RFK model are shown in Fig. 11, and the distribution of the interpolation points and the results show that the jagged errors at the edges of the contour heat maps are thought to be caused by too little information provided by the boundaries.

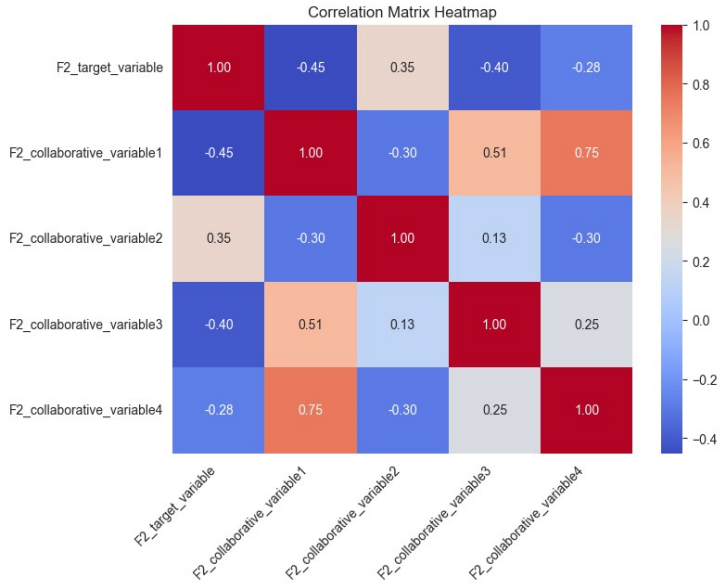


Figure 10 Correlation of F2-target with covariates

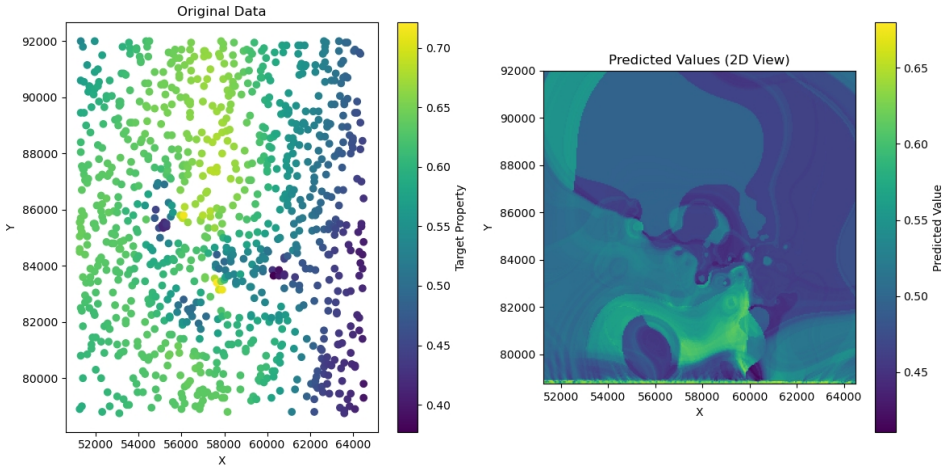


Figure 11 RFk model solving result contour plot

5. Strengths and Weaknesses

5.1 Strengths

1. Kriging:

   Kriging is a spatial autocorrelation-based geostatistical interpolation method that is based on spatial correlation to provide the best linear unbiased estimation (BLUE), and supports a variety of theoretical variational function models (e.g., spherical, exponential, Gaussian). As a result, when sampling the F1-target multiple times to analyse the error, the error decreases rapidly with increasing sample size.

2. Co-kriging:

   Co-kriging extends Kriging to improve prediction by introducing spatially correlated auxil-

iary variables (covariates). The use of auxiliary variables through multivariate integration improves prediction accuracy, and cross-variance function modelling is able to model the interaction between the primary and auxiliary variables.

More accurate results were harvested in the prediction using the covariate F1-collaborative-variable1.

3. RFK model:

   Combines machine learning (Random Forest) with geostatistical methods (Co-Kriging) for spatial prediction. It is able to identify complex non-linear dependencies between variables, quantify and identify the importance of features (covariates), with high prediction accuracy and applicable to large datasets and high-dimensional data.

   In the face of the weak correlation of F2-target and the difficulty of covariance calculation, it effectively reduces the computational complexity.

4. Smooth Bivariate Spline:

   A spline-based interpolation method that fits smooth surfaces using polynomial functions, suitable for smooth surface generation for continuous spatial variables, in addition to eliminating the need for smoothness assumptions. Faster computation on large datasets compared to Kriging.

5. LinearNDInterpolator:

   A simple interpolation method that uses linear functions in N-dimensional space for prediction. Very efficient on low-dimensional datasets. No complex model fitting required. No assumptions about variational functions or spatial processes are required.

5.2 Weaknesses

1. Kriging:

   Solving kriging systems is time-consuming for large-scale datasets. Accurate estimation of the variational function may be more difficult. Assumes that the spatial process is smooth, which does not always hold. But can effectively circumvent these drawbacks in the prediction of spatial variables in general.

2. Co-kriging:

   Requires estimation of the cross-variance function, and the process can be complex and error-prone. Multivariate correlation analyses are computationally demanding. Dependent on the availability and quality of ancillary data.

   Multiple problems with covariance computation occurred when using covariates to compute the F2-target.

3. RFK model:

   Random forests are more difficult to interpret than traditional geostatistical models. The

combination of Random Forests and Co-Kriging adds computational complexity. Fine tuning of the random forest and variational models is required.

The algorithm was found to be poorly interpretable and relies on careful parameter tuning when calculating F2-target.

4. Smooth Bivariate Spline:

   Excessive smoothing: may smooth out important spatial variations. Global Fit: performs poorly in highly heterogeneous or irregular data. No Uncertainty Quantification: does not provide an estimate of the uncertainty in the prediction results.

   In applying the algorithm in the estimation of the F1-target, it is demonstrated that the algorithm's overly smooth fit leads to distortion of information.

5. LinearNDInterpolator:

   Limited accuracy: performs poorly in capturing nonlinear trends or spatial variations. Difficulty with sparse data: weak performance in sparse or unevenly distributed data.

   In applying this algorithm in F1-target prediction, the accuracy is severely lacking in the face of low sampling rates.

References

[1] ZHOU Youfeng,XIE Binglou,LI Mingshi. Mapping regional forest aboveground biomass from random forest Co-Kriging approach: a case study from north Guangdong(in Chinese)[J]. Journal of NanJing Forestry University (Natural Sciences Edition),2024,48(1):169-178. DOI:10.12302/j.issn.1000-2006.202202015.

[2] Rojas-Gonzalez S ,Nieuwenhuyse V I .A survey on kriging-based infill algorithms for multi-objective simulation optimization[J].Computers and Operations Research,2020,116104869-104869.

[3] Algorithms; New Algorithms Findings from Huazhong University of Science and Technology Outlined (An efficient multi-objective PSO algorithm assisted by Kriging metamodel for expensive black-box problems)[J].Journal of Engineering,2017,829-.

[4] Christoph M ,Klaus N ,Mengxi Y .On Cokriging, Neural Networks, and Spatial Blind Source Separation for Multivariate Spatial Prediction[J].IEEE GEOSCIENCE AND REMOTE SENSING LETTERS,2021,18(11):1931-1935.

Appendix

Listing 1: Kriging Algorithm Code for Question 1

```
import numpy as np
import matplotlib.pyplot as plt
from pykrige.ok import OrdinaryKriging
from sklearn.metrics import mean_squared_error
from scipy.spatial.distance import pdist, squareform

# Functions for reading and rearranging data
def read_and_rearrange_data(file_path).
    with open(file_path,  r ) as f.
        # read the first few lines of metadata
        metadata = []
        for i in range(5): # Read the first 4 lines of metadata
            metadata.append(f.readline().strip())

        # Output the metadata
        print(Metadata:  )
        for line in metadata.
            print(line)

        # Read the data and put it into a list
        data = []
        for line in f.
            # Split each line's values and convert them to floats
            values = list(map(float, line.split()))
            data.extend(values) # add all values from the current line to the
                data list

        # Output the length of the data, confirming the total number of values
            read
        print( f \nTotal data points read: {len(data)})

        # Make sure the number of data points is 266 * 266
        expected_length = 266 * 266
        if len(data) ! = expected_length.
            raise ValueError(f Total length of data {len(data)} does not match
                expected matrix size {expected_length})
```

Listing 2: Kriging Algorithm Code for Question 1

```
        # Reshape the list into a 266x266 matrix
        data_matrix = np.array(data).reshape((266, 266))

    return metadata, data_matrix

# Kriging interpolation and plotting logic
file_path = Attachment 1/F1_target_variable.txt
metadata, data_matrix = read_and_rearrange_data(file_path)

# Interpolate using full-size data
x = np.arange(51250, 64500 + 50, 50, dtype=float)
y = np.arange(78750, 92000 + 50, 50, dtype=float)

# Intercept the portion to be interpolated
data_matrix = data_matrix[100:150, 100:150]
x = x[100:150]
y = y[100:150]

# Create the grid
grid_x, grid_y = np.meshgrid(x, y)

# Define different sampling percentages
sampling_percentages = np.arange(0.01, 0.501, 0.01) # Sample percentages in
    steps of 0.001

# Store the error at each sampling percentage
errors = []

for percentage in sampling_percentages.
    # Randomly select sample points
    num_samples = int(percentage * len(x) * len(y)) # select sample points
        based on percentage
    np.random.seed(42) # fix random seed to ensure reproducibility
    sample_indices = np.random.choice(len(x) * len(y), num_samples,
        replace=False)

    # Get the row and column indices of the sample points
    sample_rows = sample_indices // len(x)
```

Listing 3: Kriging Algorithm Code for Question 1

```
    sample_cols = sample_indices % len(x)

    # Get the actual data values for these samples
    sample_values = data_matrix[sample_rows, sample_cols]

    # Kriging interpolation for prediction
    OK = OrdinaryKriging(
        x[sample_cols], y[sample_rows], sample_values.
        variogram_model= linear ,
        verbose=False,
        enable_plotting=False
    )

    # Kriging interpolation to predict values across the grid
    predicted_values, _ = OK.execute(grid, x, y)

    # Calculate the mean square error (MSE)
    mse = mean_squared_error(data_matrix.flatten(), predicted_values.flatten())
    errors.append(mse)

# Plot percentage of samples vs. error
plt.figure(figuresize=(10, 6))
plt.plot([p * 100 for p in sampling_percentages], errors, marker=o,
    linestyle=-, colour= b )
plt.xlabel(Sampling Percentage (%) )
plt.ylabel(Mean Squared Error (MSE))
plt.title(Sampling Percentage vs. Interpolation Error (MSE))
plt.grid(True)
plt.savefig(sampling_vs_error.png)
plt.show()
```

Listing 4: Importance of random forests for finding eigenvectors

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
```

Listing 5: Importance of random forests for finding eigenvectors

```
# 1. Data reading and preparation

# Coordinate range and interval
column_start = 51250
column_end = 64500
column_interval = 50

row_start = 78750
row_end = 92000
row_interval = 50

# Generate x and y axis vectors
x = np.arange(column_start, column_end + column_interval, column_interval,
    dtype=float)
y = np.arange(row_start, row_end + row_interval, row_interval, dtype=float)

# Determine the grid size
grid_size = len(x) # should be 266

# Generate grid coordinates
xx, yy = np.meshgrid(x, y)
coords = np.column_stack((xx.flatten(), yy.flatten()))

# Read target and coords data
target = pd.read_csv(t2 semivariate function analysis/data_matrix.csv,
    header=None).values
co_var1 = pd.read_csv(t2 semivariate function analysis/co_data1.csv,
    header=None).values
co_var2 = pd.read_csv(t2 semi-variable function analysis/co_data2.csv,
    header=None).values
co_var3 = pd.read_csv(t2 semi-variant function analysis/co_data3.csv,
    header=None).values
co_var4 = pd.read_csv(t2 semi-variant function analysis/co_data4.csv,
    header=None).values

# Make sure the data sizes match
```

Listing 6: Importance of random forests for finding eigenvectors

```
# Make sure the data sizes match
target = target[:grid_size, :grid_size]
co_var1 = co_var1[:grid_size, :grid_size]
co_var2 = co_var2[:grid_size, :grid_size]
co_var3 = co_var3[:grid_size, :grid_size]
co_var4 = co_var4[:grid_size, :grid_size]

# Expand a two-dimensional array into a one-dimensional array
target_values = target.flatten()
co_var1_values = co_var1.flatten()
co_var2_values = co_var2.flatten()
co_var3_values = co_var3.flatten()
co_var4_values = co_var4.flatten()

# 2. Construct the feature matrix

# Combine the coordinates and covariates into a feature matrix
X = np.column_stack(((
    #coords, # x and y coordinates
    co_var1_values, # co_var2_values
    co_var2_values, # co_var3_values
    co_var3_values, co_var4_values
    co_var4_values
))

# Target variables
y = target_values

# Print the shape of the feature matrix and target variables
print(Feature matrix shape:, x.shape)
print(Target vector shape:, y.shape)

# 3. Divide the training and test sets.

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
```

Listing 7: Importance of random forests for finding eigenvectors

```
# 4. Train the random forest regression model

rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(x_train, y_train)

# 5. model evaluation

y_pred = rf_model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f Mean Squared Error: {mse:.4f})

# 6. Analyze feature importance

#feature_names = ['X_coord', 'Y_coord', 'Co_Var1', 'Co_Var2', 'Co_Var3',
    'Co_Var4']
feature_names = [ 'Co_Var1', 'Co_Var2', 'Co_Var3', 'Co_Var4' ]
importances = rf_model.feature_importances_

# Sort features by importance
indices = np.argsort(importances)[::-1]

print( Feature importances: )
for idx in indices.
    print( f {feature_names[idx]}: {importances[idx]:.4f})

# Visualize feature importance
plt.figure(figuresize=(8, 6))
plt.title(Feature Importances )
plt.bar(range(len(importances)), importances[indices], align='center')
plt.xticks(range(len(importances)), [feature_names[i] for i in indices])
plt.ylabel( Importance )
plt.xlabel( Features )
plt.tight_layout()
plt.tight_layout()
```

Listing 8: Code of question 3

```
import numpy as np
import pandas as pd
```

Listing 9: Code of question 3

```
from scipy.spatial.distance import cdist
from scipy.sparse import csr_matrix, hstack, vstack, eye, csc_matrix
from scipy.sparse.linalg import spsolve
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from joblib import Parallel, delayed
import warnings

# Ignore specific runtime warnings (e.g. binary incompatibility warnings)
warnings.filterwarnings( ignore , category=RuntimeWarning)

# Data input
target_variable = pd.read_csv(data_matrix.csv, header=None).values
co_variable1 = pd.read_csv(co_data1.csv, header=None).values
co_variable2 = pd.read_csv(co_data4.csv, header=None).values

# Decrease grid_size to reduce training data size
grid_size = 200 # Reduce from 266 to 200, you can further adjust according to
    the actual memory situation.
target_variable = target_variable[:grid_size, :grid_size].flatten()
co_variable1 = co_variable1[:grid_size, :grid_size].flatten()
co_variable2 = co_variable2[:grid_size, :grid_size].flatten()

# Generate coordinates
x = np.linspace(51250, 64500, grid_size)
y = np.linspace(78750, 92000, grid_size)
xx, yy = np.meshgrid(x, y)
coords = np.column_stack((xx.flatten(), yy.flatten()))

# Delineate the training and test sets
indices = np.arange(len(target_variable))
train_indices, test_indices = train_test_split(indices, test_size=0.8,
    random_state=42)

train_coords = coords[train_indices]
```

Listing 10: Code of question 3

```
test_coords = coords[test_indices]
train_target = target_variable[train_indices]
train_co1 = co_variable1[train_indices]
train_co2 = co_variable2[train_indices]
test_target = target_variable[test_indices]

# Covariance function
def spherical_covariance(h, sill=1.0, range_=10.0, nugget=0.0):
    h = np.abs(h)
    cov = np.where(
        h <= range_,
        sill * (1 - 1.5 * (h / range_) + 0.5 * (h / range_) ** 3),
        0.0,
    )
    cov += nugget * (h == 0)
    return cov

# Sparse covariance matrix calculation
def compute_sparse_covariance_matrix(coords1, coords2, covariance_func,
    threshold, **kwargs).
    # compute the distance matrix
    dists = cdist(coords1, coords2)
    # Calculate the covariance matrix
    cov = covariance_func(dists, **kwargs)
    # Apply threshold and ignore small covariance values
    cov[cov < threshold] = 0
    return csr_matrix(cov)

# Parameterize
var_t = np.var(train_target)
var_c1 = np.var(train_co1)
var_c2 = np.var(train_co2)
corr_tc1 = np.corrcoef(train_target, train_co1)[0, 1]
corr_tc2 = np.corrcoef(train_target, train_co2)[0, 1]
corr_c1c2 = np.corrcoef(train_co1, train_co2)[0, 1]

sill_t = var_t
sill_c1 = var_c1
```

Listing 11: Code of question 3

```
sill_c2 = var_c2
range_ = 5000 # Decrease the range to increase the sparsity, you can adjust it
    according to the actual data.
threshold = 1e-3 # Sparse matrix threshold.

# Compute sparse covariance matrix
C_tt = compute_sparse_covariance_matrix(train_coords, train_coords,
    spherical_covariance, threshold, sill=sill_t, range_=range_)
C_c1c1 = compute_sparse_covariance_matrix(train_coords, train_coords,
    spherical_covariance, threshold, sill=sill_c1, range_=range_)
C_c2c2 = compute_sparse_covariance_matrix(train_coords, train_coords,
    spherical_covariance, threshold, sill=sill_c2, range_=range_)
C_tc1 = corr_tc1 * compute_sparse_covariance_matrix(train_coords,
    train_coords, spherical_covariance, threshold, sill=np.sqrt(sill_t * sill_
    c1), range_=range_)
C_tc2 = corr_tc2 * compute_sparse_covariance_matrix(train_coords,
    train_coords, spherical_covariance, threshold, sill=np.sqrt(sill_t * sill_
    c2), range_=range_)
C_c1c2 = corr_c1c2 * compute_sparse_covariance_matrix(train_coords,
    train_coords, spherical_covariance, threshold, sill=np.sqrt(sill_c1 *
    sill_c2), range_=range_)

# Construct a block of covariance matrices
n_train = len(train_coords)
one_vector = csr_matrix(np.ones((n_train, 1)))
zero_vector = csr_matrix(np.zeros((n_train, 1)))

# Construct K_sparse using sparse block matrices
K_top = hstack([C_tt, C_tc1, C_tc2, one_vector])
K_middle1 = hstack([C_tc1.transpose(), C_c1c1, C_c1c2, zero_vector])
K_middle2 = hstack([C_tc2.transpose(), C_c1c2.transpose(), C_c2c2,
    zero_vector])
K_bottom = hstack([one_vector.transpose(), zero_vector.transpose(),
    zero_vector.transpose(), csr_matrix((1,1))])
K_sparse = vstack([K_top, K_middle1, K_middle2, K_bottom])

# Add regularization terms to avoid matrix singularities
epsilon = 1e-10
```

Listing 12: Code of question 3

```
K_sparse_reg = K_sparse + epsilon * eye(K_sparse.shape[0], format='csr')

# Convert to CSC format to speed up solving
K_sparse_reg = K_sparse_reg.tocsc()

# Compute the covariance of the sparse covariance matrix with the test points
C_t0t = compute_sparse_covariance_matrix(test_coords, train_coords,
    spherical_covariance, threshold, sill=sill_t, range_=range_)
C_t0c1 = corr_tc1 * compute_sparse_covariance_matrix(test_coords,
    train_coords, spherical_covariance, threshold, sill=np.sqrt(sill_t * sill_
    c1), range_=range_)
C_t0c2 = corr_tc2 * compute_sparse_covariance_matrix(test_coords,
    train_coords, spherical_covariance, threshold, sill=np.sqrt(sill_t * sill_
    c2), range_=range_)

# Build the vector to be predicted
def build_k_vector(i).
    k_tt = C_t0t.getrow(i)
    k_tc1 = C_t0c1.getrow(i)
    k_tc2 = C_t0c2.getrow(i)
    k = hstack([k_tt, k_tc1, k_tc2, csr_matrix([[1.0]])])
    return k.toarray().flatten()
# Predict function
def predict(i).
    k = build_k_vector(i)
    try.
        weights = spsolve(K_sparse_reg, k)
    except Exception as e.
        print(f Error solving for test point {i}: {e})
        return np.nan
    # Split the weight vector
    lambda_t = weights[:n_train]
    lambda_c1 = weights[n_train:2*n_train]
    lambda_c2 = weights[2*n_train:3*n_train]
    # Calculate the prediction
    pred = (
        np.dot(lambda_t, train_target) +
        np.dot(lambda_c1, train_co1) +
```

Listing 13: Code of question 3

```
        np.dot(lambda_c2, train_co2)
    )
    return pred

# Predict using parallel processing
num_cores = -1 # use all available cores
predictions = Parallel(n_jobs=num_cores, backend=loky)(delayed(predict)(i) for
    i in range(len(test_coords)))

# Convert the predictions into a NumPy array
predictions = np.array(predictions)

# Handle possible NaNs in predictions
valid_indices = ~np.isnan(predictions)
if not np.all(valid_indices): # Handle possible NaN in predictions.
    print(f There are {np.sum(~valid_indices)} points in the prediction that
        were not successfully predicted.)

# Evaluate the results
mse = mean_squared_error(test_target[valid_indices],
    predictions[valid_indices])
print(f Mean Squared Error: {mse:.4f})

# Plot the comparison
plt.figure(figuresize=(12, 6))

# 1. real values
plt.subplot(1, 2, 1)
plt.title(True  Values )
true_grid = np.full(len(target_variable), np.nan)
true_grid[test_indices] = test_target
true_grid[train_indices] = train_target
true_grid = true_grid.reshape((grid_size, grid_size))
plt.imshow(true_grid, extent=(x.min(), x.max(), y.min(), y.max()),
    origin=lower, cmap=viridis)
plt.colorbar(label= Value )
plt.xlabel(  X  )
plt.ylabel(  Y  )
```

Listing 14: Code of question 3

```
# 2. predicted values
plt.subplot(1, 2, 2)
plt.title(Predicted  Values )
pred_grid = np.full(len(target_variable), np.nan)
pred_grid[test_indices] = predictions
pred_grid[train_indices] = train_target
pred_grid = pred_grid.reshape((grid_size, grid_size))
plt.imshow(pred_grid, extent=(x.min(), x.max(), y.min(), y.max()),
    origin=lower, cmap=viridis)
plt.colorbar(label= Value )
plt.xlabel(  X  )
plt.ylabel(  Y  )

plt.tight_layout()
plt.savefig(optimized_predictions.png)
plt.show()
```

Listing 15: Code of question 4

```
import numpy as np
import pandas as pd
from scipy.spatial.distance import cdist, pdist
from scipy.interpolate import griddata
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from scipy.linalg import cho_factor, cho_solve
from joblib import Parallel, delayed
import gstools as gs
import warnings

# Ignore specific runtime warnings (optional)
warnings.filterwarnings( ignore , category=RuntimeWarning)

# 1. Data input and preparation
def load_and_prepare_data().
    # Load the sample point data
    samples_df = pd.read_excel(Attachment 2/F2_target_variable.xlsx)

    # Extract coordinates and target variables
```

Listing 16: Code of question 4

```
sample_coords = samples_df[['X-Coordinate',
    'Y-Coordinate']].values.astype(np.float32)
target_property = samples_df['Target Property'].values.asype(np.float32)

# Load covariate data (assuming each sample point corresponds to a
    covariate value)
co_variable1 = pd.read_csv(co_data2_1.csv,
    header=None).values.flatten().asype(np.float32)
co_variable2 = pd.read_csv(co_data2_2.csv,
    header=None).values.flatten().asype(np.float32)

# Ensure that the sample point and covariate data lengths are the same
assert len(sample_coords) == len(co_variable1) == len(co_variable2), \
     Mismatch  between the number of sample points and the number of
        covariables.

# For data points with the same coordinates, calculate the mean of the
    target variable and keep the first covariate value
grouped_df = samples_df.groupby(['X-Coordinate', 'Y-Coordinate'],
    as_index=False).agg({
    'Target Property': 'mean',
    'Co-variable1': 'first', # Assume that the co-variables are identical
        at the same coordinates
    'Co-variable2': 'first'
})

# Generate grid point coordinates (266 x 266)
grid_size = 266
column_start = 51250
column_end = 64500
column_interval = 50
row_start = 78750
row_end = 92000
row_interval = 50

x = np.arange(column_start, column_end + column_interval, column_interval,
```

```
    dtype=np.float32)
```

Listing 17: Code of question 4

```
y = np.arange(row_start, row_end + row_interval, row_interval,
    dtype=np.float32)
xx, yy = np.meshgrid(x, y)
grid_coords = np.column_stack((xx.flatten(),
    yy.flatten())).astype(np.float32)

# Use interpolation to extend the covariates to the whole grid
print(Interpolating covariate 1...)
grid_co1 = griddata(grouped_df[['X-Coordinate', 'Y-Coordinate']].values,
                    grouped_df['Co-variable1'].values,
                    grid_coords.
                    method='linear')

print(Interpolating values for Co-variable2...)
grid_co2 = griddata(grouped_df[['X-Coordinate', 'Y-Coordinate']].values,
                    grouped_df['Co-variable2'].values,
                    grid_coords.
                    method='linear')

# Fill in the NaN in the interpolation result (e.g., using nearest
    neighbor interpolation)
nan_mask1 = np.isnan(grid_co1)
if np.any(nan_mask1).
    print(f Populating {np.sum(nan_mask1)} NaN values in covariate 1...)
    grid_co1[nan_mask1] = griddata(grouped_df[['X-Coordinate',
        'Y-Coordinate']].values,
                                   grouped_df['Co-variable1'].values,
                                   grid_coords[nan_mask1],
                                   method='nearest')

nan_mask2 = np.isnan(grid_co2)
if np.any(nan_mask2).
    print(f Populating {np.sum(nan_mask2)} NaN values in covariate 2...)
    grid_co2[nan_mask2] = griddata(grouped_df[['X-Coordinate',
        'Y-Coordinate']].values,
                                   grouped_df['Co-variable2'].values,
```

```
                                    grid_coords[nan_mask2],
                                    method='nearest')
```

Listing 18: Code of question 4

```
    # Interpolate the target variables
    print(Interpolating the target variable...)
    grid_target = griddata(grouped_df[['X-Coordinate', 'Y-Coordinate']].values,
                        grouped_df[['Target Property'].values,
                        grid_coords.
                        method='linear')

    # Fill in the NaN values
    nan_mask_t = np.isnan(grid_target)
    if np.any(nan_mask_t).
        print(f Filling {np.sum(nan_mask_t)} NaN values in target variable...)
        grid_target[nan_mask_t] = griddata(grouped_df[['X-Coordinate',
            'Y-Coordinate']].values,
                                    grouped_df['Target Property'].values,
                                    grid_coords[nan_mask_t],
                                    method='nearest')

    # Convert the interpolated data to a 2D matrix
    grid_target_matrix = grid_target.reshape((grid_size, grid_size))
    grid_co1_matrix = grid_co1.reshape((grid_size, grid_size))
    grid_co2_matrix = grid_co2.reshape((grid_size, grid_size))

    return {
          grid_coords  : grid_coords, grid_target_matrix
         grid_target_matrix  : grid_target_matrix, grid_co1_matrix:
           grid_co2.reshape((grid_size, grid_size), grid_size)
         grid_co1_matrix  : grid_co1_matrix, grid_co2_matrix: grid_co2_matrix
         grid_co2_matrix  : grid_co2_matrix,
          x   : x,   y   : y,  grid_co2_matrix : grid_co2_matrix
          y   : y,   grid_size  : grid_co2_matrix
         grid_size  : grid_size
    }
# 2. define the covariance function (spherical model)
def spherical_covariance(h, sill=1.0, range_=10.0, nugget=0.0).
    h = np.abs(h)
```

```
    cov = np.where(
        h <= range_,
        sill * (1 - 1.5 * (h / range_) + 0.5 * (h / range_) ** 3),
        0.0,
    )
    cov += nugget * (h == 0)
    return cov

# 3. Compute the covariance matrix
def compute_covariance_matrix(coords1, coords2, covariance_func, **kwargs):
    dists = cdist(coords1, coords2)
    return covariance_func(dists, **kwargs)

# 4. Build a system of covariance kriging equations
def build_covariance_matrices(grid_data):
    grid_coords = grid_data['grid_coords']
    grid_target = grid_data['grid_target_matrix'].flatten()
    grid_co1 = grid_data['grid_co1_matrix'].flatten()
    grid_co2 = grid_data['grid_co2_matrix'].flatten()

    # Calculate the variance of each variable
    var_t = np.var(grid_target)
    var_c1 = np.var(grid_co1)
    var_c2 = np.var(grid_co2)

    # Calculate the correlation coefficient between the variables
    corr_tc1 = np.corrcoef(grid_target, grid_co1)[0, 1]
    corr_tc2 = np.corrcoef(grid_target, grid_co2)[0, 1]
    corr_c1c2 = np.corrcoef(grid_co1, grid_co2)[0, 1]

    # Adjust the parameters of the covariance function
    sill_t = var_t
    sill_c1 = var_c1
    sill_c2 = var_c2

    # Set the range of influence (adjust as needed)
    range_ = 400000 # Adjust for data units

    # Calculate the covariance matrix
```

Listing 19: Code of question 4

```
    print(Compute C_tt... )
    C_tt = compute_covariance_matrix(grid_coords, grid_coords,
        spherical_covariance, sill=sill_t, range_=range_)
    print(Compute C_c1c1... )
    C_c1c1 = compute_covariance_matrix(grid_coords, grid_coords,
        spherical_covariance, sill=sill_c1, range_=range_)
    print(Compute C_c2c2... )
    C_c2c2 = compute_covariance_matrix(grid_coords, grid_coords,
        spherical_covariance, sill=sill_c2, range_=range_)

    # Calculate the cross-covariance matrix
    print(Calculating C_tc1... )
    C_tc1 = corr_tc1 * compute_covariance_matrix(grid_coords, grid_coords,
        spherical_covariance, sill=np.sqrt(sill_t * sill_c1), range_=range_)
    print(Calculating C_tc2... )
    C_tc2 = corr_tc2 * compute_covariance_matrix(grid_coords, grid_coords,
        spherical_covariance, sill=np.sqrt(sill_t * sill_c2), range_=range_)
    print(Calculating C_c1c2... )
    C_c1c2 = corr_c1c2 * compute_covariance_matrix(grid_coords, grid_coords,
        spherical_covariance, sill=np.sqrt(sill_c1 * sill_c2), range_=range _)

    # Construct the left-hand side covariance matrix K
    n_train = len(grid_coords)
    one_vector = np.ones((n_train, 1))
    zero_vector = np.zeros((n_train, 1))

    print(Constructing the system of synergistic kriging equations K...)
    K = np.block([
        [C_tt, C_tc1, C_tc2, one_vector], [C_tc1.
        [C_tc1.T, C_c1c1, C_c1c2, zero_vector], [C_tc2.
        [C_tc2.T, C_c1c2.T, C_c2c2, zero_vector], [one_vector.
        [one_vector.T, zero_vector.T, zero_vector.T, np.zeros((1, 1))]]
    ])

    return K, sill_t, sill_c1, sill_c2, corr_tc1, corr_tc2, corr_c1c2

# 5. Prediction Functions
def predict_single(i, C_t0t, C_t0c1, C_t0c2, cho_K, grid_target, grid_co1,
```

```
    grid_co2, n_train).
    k = np.concatenate([
        C_t0t[i],
        C_t0c1[i],
        C_t0c2[i],
        [1.0]
    ])
    weights = cho_solve(cho_K, k)
    lambda_t = weights[:n_train]
    lambda_c1 = weights[n_train:2*n_train]
    lambda_c2 = weights[2*n_train:3*n_train]
    pred = (
        np.dot(lambda_t, grid_target) +
        np.dot(lambda_c1, grid_co1) +
        np.dot(lambda_c2, grid_co2)
    )
    return pred
```

Listing 20: Code of question 4

```
# 6. main function
def main().
    # Load and prepare data
    print(Loading and preparing data...)
    grid_data = load_and_prepare_data()

    # Construct the covariance matrix
    print(Constructing the covariance matrix...)
    K, sill_t, sill_c1, sill_c2, corr_tc1, corr_tc2, corr_c1c2 =
        build_covariance_matrices(grid_data)

    # Calculate the covariance between the points to be estimated and the
        training points
    print(Compute C_t0t... )
    C_t0t = compute_covariance_matrix(grid_data['grid_coords'],
        grid_data['grid_coords'], spherical_covariance, sill=sill_t,
        range_=400000)
    print(Calculating C_t0c1... )
    C_t0c1 = corr_tc1 * compute_covariance_matrix(grid_data['grid_coords'],
```

```
    grid_data['grid_coords'], spherical_covariance, sill=np.sqrt(sill _t *
    sill_c1), range_=400000)
print(Calculating C_t0c2... )
C_t0c2 = corr_tc2 * compute_covariance_matrix(grid_data['grid_coords'],
    grid_data['grid_coords'], spherical_covariance, sill=np.sqrt(sill _t *
    sill_c2), range_=400000)
```

Listing 21: Code of question 4

```
# Construct the left-hand side covariance matrix of the system of
    synergistic kriging equations K
epsilon = 1e-10
K_reg = K + epsilon * np.eye(K.shape[0])

    # Pre-decompose the covariance matrix
print(Performing Cholesky decomposition...)
cho_K = cho_factor(K_reg)

# Parallel computation of predictions
print(Starting parallel computation of predictions...)
n_test = len(grid_data['grid_coords'])
predictions = Parallel(n_jobs=-1)(
    delayed(predict_single)(
        i, C_t0t, C_t0c1, C_t0c2, cho_K.
        grid_data['grid_target_matrix'].flatten(),
        grid_data['grid_co1_matrix'].flatten(),
        grid_data['grid_co2_matrix'].flatten(),
        len(grid_data['grid_co2_matrix'].flatten(),
            len(grid_data['grid_co2_matrix'].
    ) for i in range(n_test)
)
predictions = np.array(predictions)

# Map the predictions back to the grid
print(Mapping predictions... )
grid_size = grid_data['grid_size']
x = grid_data['x']
y = grid_data['y']

predictions_grid = predictions.reshape((grid_size, grid_size))
```

Listing 22: Code of question 4

```
plt.figure(figsize=(12, 6))
plt.imshow(predictions_grid, extent=(x.min(), x.max(), y.min(), y.max()),
    origin=lower, cmap=viridis)
plt.colorbar(label=Predicted Value)
plt.title(Synergistic Kriging Predicted Value Distribution)
plt.xlabel( X )
plt.ylabel( Y )
plt.tight_layout()
plt.savefig(new_t4_res1. png )
plt.show()

# Read and visualize the raw data matrix
print(Reading and visualizing the raw data matrix...)
target_variable = pd.read_csv(data_matrix.csv, header=None).values #
    266x266
co_variable1 = pd.read_csv(co_data1.csv, header=None).values # 266x266
co_variable2 = pd.read_csv(co_data4.csv, header=None).values # 266x266

# Check the data shape
assert target_variable.shape == (266, 266), Target variable shape does not
    match
assert co_variable1.shape == (266, 266), co_variable1 shape does not match
assert co_variable2.shape == (266, 266), Co-variable2 shape mismatch

# Visualize the target and covariable
plt.figure(figsize=(18, 5))

plt.subplot(1, 3, 1)
plt.imshow(target_variable, origin='lower', extent=(x.min(), x.max(),
    y.min(), y.max()), cmap='viridis')
plt.colorbar(label='target variable')
plt.title('Target variable distribution')
plt.xlabel('X')
plt.ylabel('Y')

plt.subplot(1, 3, 2)
plt.imshow(co_variable1, origin='lower', extent=(x.min(), x.max(),
    y.min(), y.max()), cmap='plasma')
```

Listing 23: Code of question 4

```
plt.colorbar(label='covariate 1')
plt.title('Synergistic variable 1 distribution')
plt.xlabel('X')
plt.ylabel('Y')

plt.subplot(1, 3, 3)
plt.imshow(co_variable2, origin='lower', extent=(x.min(), x.max(),
    y.min(), y.max()), cmap='plasma')
plt.colorbar(label='covariate 2')
plt.title('Synergistic variable 2 distribution')
plt.xlabel('X')
plt.ylabel('Y')

plt.tight_layout()
plt.savefig(new_t4_res2. png )
plt.show()

# Calculate and visualize the variance function
print(Calculating and visualizing the variational function...)
bin_width = 500 # Adjust for the span of the data
max_distance = 50000 # Maximum distance, covers most spatial correlations
bin_edges = np.arange(0, max_distance + bin_width, bin_width)

coords_flat = grid_data['grid_coords'].T # gstools expects coordinates in
    format (dim, N)
target = grid_data['grid_target_matrix'].flatten().astype(np.float32)
cov1 = grid_data['grid_co1_matrix'].flatten().astype(np.float32)
cov2 = grid_data['grid_co2_matrix'].flatten().astype(np.float32)

# Define the compute variogram function
def compute_variogram(data, coords, bin_edges).
        "
    Compute the experimental variogram function
        "
    vgm = gs.vario_estimate(coords, data, bin_edges=bin_edges)
    return vgm

# Compute the univariate vario function
```

Listing 24: Code of question 4

```
vgm_target = compute_variogram(target, coords_flat, bin_edges)
vgm_cov1 = compute_variogram(cov1, coords_flat, bin_edges)
vgm_cov2 = compute_variogram(cov2, coords_flat, bin_edges)

# Compute the cross variance function
def compute_cross_variogram(var1, var2, coords, bin_edges, bin_width).
        "
    Compute the cross variate function
        "
    distances = pdist(coords) # Shape (N*(N-1)/2, )
    cov = pdist(np.column_stack((var1, var2)), lambda u, v: (u[0] -
        np.mean(var1)) * (v[1] - np.mean(var2)))
    variogram = np.zeros(len(bin_edges) - 1)
    for i in range(len(bin_edges) - 1): lower = bin_edges[i.e.
        lower = bin_edges[i]
        upper = bin_edges[i + 1]: mask = (distances >= lower)
        mask = (distances >= lower) & (distances < upper)
        if np.sum(mask) > 0.
            variogram[i] = 0.5 * np.mean(cov[mask])
        else: variogram[i] = np.mean(cov[mask])
            variogram[i] = np.nan
    bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2
    return bin_centers, variogram

vgm_cross1 = compute_cross_variogram(target, cov1, coords_flat, bin_edges,
    bin_width)
vgm_cross2 = compute_cross_variogram(target, cov2, coords_flat, bin_edges,
    bin_width)

# Plot the variogram
plt.figure(figuresize=(15, 5))

plt.subplot(1, 3, 1)
plt.plot(vgm_target[0], vgm_target[1], 'o-', label='target variable
    variance function')
plt.legend()
plt.xlabel('Distance (meters)')
plt.ylabel('Half variance')
```

Listing 25: Code of question 4

```
plt.title('Experimental variance function of the target variable')

plt.subplot(1, 3, 2)
plt.plot(vgm_cov1[0], vgm_cov1[1], 's-', label='covariate 1 variance
    function')
plt.legend()
plt.xlabel('Distance (meters)')
plt.ylabel('Half variance')
plt.title('Co-variate 1 experimental variance function')

plt.subplot(1, 3, 3)
plt.plot(vgm_cross1[0], vgm_cross1[1], 'x-', label='Target-Synergy 1
    Crossover Variance Function')
plt.plot(vgm_cross2[0], vgm_cross2[1], 'd-', label='target-synergy 2 cross
    variance function')
plt.legend()
plt.xlabel('Distance (meters)')
plt.ylabel('Crossover semivariance')
plt.title('Target and covariate cross-variance function')

plt.savefig(new_t4_res3. png )
plt.show()

# Determine the range of the variational function
def determine_range(vgm, bin_centers, threshold=0.95):
        "
    Determine the range of the variational function
    :param vgm: tuple of (bin_centers, variogram)
    :param bin_centers: array of bin centers
    :param threshold: float, proportion of sill to determine range
    :return: float, range distance
        "
    sill = np.nanmax(vgm[1])
    threshold_value = threshold * sill
    for distance, semivariance in zip(vgm[0], vgm[1]): if semivariance >=
        threshold
        if semivariance >= threshold_value.
```

Listing 26: Code of question 4

```
            if semivariance >= threshold_value: return distance
        return bin_centers[-1]: if semivariance >= threshold_value: return
            distance

    range_target = determine_range(vgm_target, vgm_target[0])
    range_cov1 = determine_range(vgm_cov1, vgm_cov1[0])
    range_cov2 = determine_range(vgm_cov2, vgm_cov2[0])

    print(f Target variable range: {range_target} meters)
    print(f  Co -variable 1 range: {range_cov1} meters)
    print(f Covariate 2 range: {range_cov2} meters)

    # Select a uniform influence range
    influence_range = max(range_target, range_cov1, range_cov2)
    print(f Uniform influence range selected: {influence_range} meters)

if __name__ ==  __main__ .
    main()
```