

Vignette for geospaNN

1 Summary

geospaNN stands for **Geospatial Neural Networks** and is a python package that implements NN-GLS, a geographically-informed Graph Neural Network (GNN) architecture (Zhan and Datta 2024). This file provides a brief introduction on how to use **geospaNN** for geospatial data analysis. The vignette is divided into four parts:

- **Data preparation:** This section covers features that transform raw geospatial data into the input format for the Neural Network trainer. It also introduces a geospatial simulation function for conducting general statistical experiments.
- **Model training:** This section explains the main functions used to train the model. To streamline the common training procedure, a user-friendly trainer is introduced for NN-GLS.
- **Model evaluation:** This section visualizes the training process and spatial parameter estimation. It also provides functions for mean-function estimation and prediction at new location, which are the primary goals of the package and can support various geospatial applications.
- **Additional examples:** This section is independent with the main pipeline in the sections above and provides additional examples where **geospaNN** can be applied in different scenarios.

In addition to this vignette, **geospaNN**'s official website (<https://wentaozhan1998.github.io/geospaNN-doc/>) offers an installation guide, comprehensive documentation, and several statistical experiments. The demo codes for this vignette is available at (<https://wentaozhan1998.github.io/geospaNN-doc/Examples/>).

2 Main Content

We will start by loading the **geospaNN** and other dependencies for this vignette:

```
import torch
import geospaNN
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'
```

2.1 Spatial Mixed Effect Model (SPMM) and Goal

In this vignette, we focus on the spatial mixed effect model, formulated as:

$$Y(s) = m(X(s)) + w(s) + \epsilon(s),$$

where:

- $X(s)$ are the observed covariates at location s ,

- $Y(s)$ is the observed outcome,
- $\epsilon(s)$ is the i.i.d. random Gaussian noise following $N(0, \lambda\sigma^2)$, also known as the nugget in spatial literature, where σ^2 denotes the variance of the spatial effect, and λ quantifies the proportion of non-spatial variance relative to the spatial variance.
- $w(s)$ represents a stochastic process accounting for spatial correlation, modeled as a mean-zero exponential Gaussian Process (GP) with the following covariance structure:

$$\text{cov}(w(s_1), w(s_2)) = C(s_1, s_2 | \theta) = \sigma^2 \exp(-\phi \|s_1 - s_2\|).$$

Here, θ is introduced as a general notation for $(\sigma^2, \phi, \lambda)$. Under this setting, **geospaNN** aims to estimate the non-linear mean function $m(\cdot)$ by using the Neural Network family (estimation task), as well as predict the response $y(s_{new})$ at new locations s_{new} by combining the mean estimate $\hat{m}(\cdot)$ and the kriging prediction $\hat{w}(s_{new})$. Nearest Neighbor Gaussian Process (NNGP) (Datta et al. 2016) is used as an approximate process to the full GP, ensuring the scalability of **geospaNN**. This enables analysis on geospatial datasets with up to 1 million observations.

2.2 Simulation and Data Preprocessing

2.2.1 Simulation

Simulation of geospatial data is a key feature of **geospaNN**, enabling scalable generation of geospatial datasets for various experiments and facilitating methodology development. Simulating spatially correlated data typically requires $O(n^3)$ running time due to the need to compute the Cholesky factor of an $n \times n$ Gaussian process covariance matrix. However, NNGP provides efficient precision matrix computation and Cholesky factorization, reducing the time complexity to $O(n)$ for large sample sizes.

In this section, we simulate data from the model described earlier. Specifically, we use the function **funXY**:

$$m(x) = 10 \sin(2\pi x).$$

The simulation setup includes:

- The non-spatial covariates X , each feature sampled uniformly from an $[0,1]$ interval.
- The spatial effect $w(s)$, generated as an exponential Gaussian process with spatial variance $\sigma^2 = 1$ (**sigma**) and spatial correlation decay $\phi = 3\sqrt{2}/20$ (**phi**).
- The i.i.d. random noise $\epsilon(s)$, with variance $\tau^2 = \lambda \cdot \sigma^2$, where $\lambda = 0.01$ (**Lambda**).

Additional required arguments include:

- Sample size (**n**): The desired number of locations.
- Number of covariates (**p**): The number of predictors.
- Number of neighbors (**nn**): Used for NNGP approximation.
- Range of spatial locations (**range**): Specifies the 2D domain $[\text{range}]^2$ from which spatial locations will be sampled.

The function returns the following in tensor formats: the coordinates **X**, response **Y**, coordinates **coord**, covariance matrix **cov** in sparse format, and random effect $w(s) + \epsilon(s)$ **corerr**.

```
def f1(X): return 10 * np.sin(2*np.pi * X)

p = 1;
funXY = f1
```

```

n = 1000
nn = 20

sigma = 1
phi = 0.3
Lambda = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), Lambda])

X, Y, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])

```

2.2.2 Practical Simulation Approach

In practice, the covariate X itself is often spatially correlated (denoted as $X(s)$). A practical shortcut for simulating in this setting involves:

1. Using `geospaNN.Simulation` to generate one correlated spatial process `corerr` as $X(s)$.
2. Generating another `corerr` as $w(s) + \epsilon(s)$.
3. Assembling Y as $m(X(s)) + w(s) + \epsilon(s)$.

The process is illustrated in the following code chunks, and figure 1 visualizes the simulated data.

```

#1
torch.manual_seed(2025)
_, _, _, X = geospaNN.Simulation(n, p, nn, funXY, torch.tensor([1, 5, 0.01]), range=[0, 1])
X = X.reshape(-1, p)
X = (X - X.min(dim=0).values) / (X.max(dim=0).values - X.min(dim=0).values)
#2
torch.manual_seed(2025)
_, _, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])
#3
Y = funXY(X).reshape(-1) + corerr

```

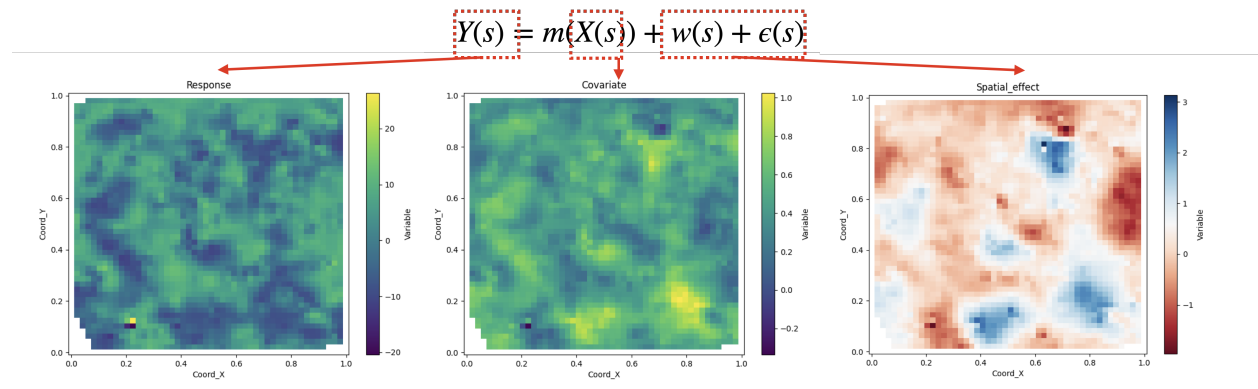


Figure 1: Simulated Data Visualization

2.2.3 Data Preprocessing

Covariates X ($n \times p$), response Y ($n \times 1$), and coordinates s ($n \times 2$) form the fundamental elements of a geospatial dataset. To prepare the dataset for a GNN module, a directed graph must be constructed,

where the direction depends on the ordering of data. However, since the default order for real data may be confounded with the spatial location, we recommend user to reorder the data randomly (`method="random"`) or by maxmin strategy proposed by Katzfuss and Guinness (2021) (`method="max-min"`) before constructing graph. (Since our data is simulated with random ordering, this line is not run to generate the results in this vignette).

```
X, Y, coord, _ = geospaNN.spatial_order(X, Y, coord, method='max-min')
```

After data reordering, the `geospaNN.make_graph` function creates a `DataLoader` object for efficient data loading and management. In addition to `X`, `Y`, and `coord`, it requires:

- The number of nearest neighbors (`nn`), since the function constructs a k -nearest-neighbor graph by default, with k specified by `nn`.
- An optional $n \times k$ index tensor `Ind_list`, which allows using a custom neighbor list. The i -th row contains the indices of nodes connected to s_i , with -1 indicating no connection. If no index tensor is specified, a k -nearest-neighbor graph mentioned above will be constructed by default.

```
data = geospaNN.make_graph(X, Y, coord, nn, Ind_list = None)
```

Additionally, splitting of the dataset into training, testing, and validation sets is simplified using the `geospaNN.split_data` function. Like `geospaNN.make_graph`, this function takes covariates X , response Y , coordinates s , and neighbor size as inputs. Users can customize the split proportions with `val_proportion` and `test_proportion`, both defaulting to 0.2. The output includes three `DataLoader` objects required for the training process.

```
np.random.seed(0)
data_train, data_val, data_test = geospaNN.split_data(X, Y, coord, neighbor_size=nn,
test_proportion=0.2, val_proportion=0.2)
```

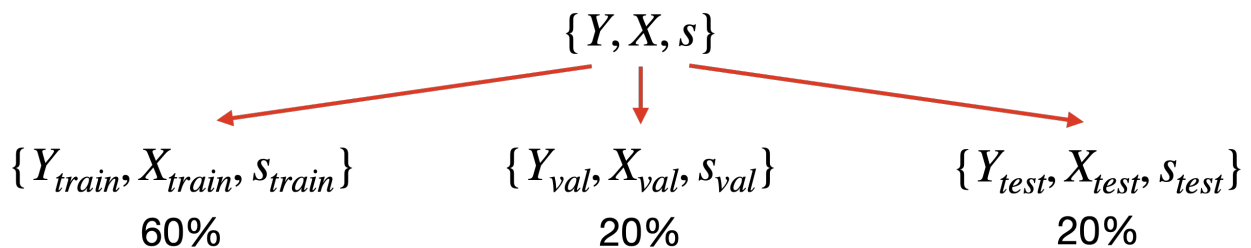


Figure 2: Data Splitting Visualization

2.3 Model Training

After preprocessing the data, `geospaNN` fits the model in two steps:

1. **Initialize the spatial parameters.**
2. **Fit NN-GLS.**

2.3.1 Parameter Initialization

Initializing the spatial parameters θ is necessary for training NN-GLS since L-BFGS-based likelihood maximum-likelihood estimation is employed to iteratively update θ . A reasonable initial guess is crucial to a successful training because the GLS-style loss function is sensitive to these spatial parameters.

Unless in special cases where initial value is available, users are encouraged to follow our default pipeline, where θ is initialized as the maximum likelihood estimator from NN-based residual. In this example, we use a 3-layer multi-layer perceptron as the NN's architecture, with 100, 50, and 20 nodes for each layer and a scalar as the output.

```
torch.manual_seed(2024)
mlp_nn = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
trainer_nn = geospaNN.nn_train(mlp_nn, lr=0.01, min_delta=0.001)
training_log = trainer_nn.train(data_train, data_val, data_test, epoch_num= 200,
                                batch_size = 60, seed = 2025)
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                                data_train.pos, neighbor_size=20)
```

2.3.2 Fit NN-GLS

With `theta0` being the initial value, the model is fitted as following:

```
torch.manual_seed(2024)
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
model = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nngls,
                      theta=torch.tensor(theta0))
trainer_nngls = geospaNN.nngls_train(model, lr=0.1, min_delta=0.001)
training_log = trainer_nngls.train(data_train, data_val, data_test, epoch_num= 200,
                                   Update_init=10, Update_step=2,
                                   batch_size = 60, seed = 2025)
```

The code chunk consists of 4 steps:

1. **Step 1:** Define the non-spatial architecture. Here we defined a 3-layer multi-layer perceptron (MLP) through `torch.nn.Sequential`).

2. **Step 2:** Define the NN-GLS model with the MLP defined in step 1 and an initialized exponential Gaussian covariance structure specified by `theta0` using `geospaNN.ngls`, which requires:
 - Number of covariates (`p`).
 - Number of nearest neighbors (`neighbor_size`).
 - Dimension of spatial coordinates (`coord_dimensions`).
 - Nonspatial architecture (`mlp_ngls`).
 - Spatial parameters (`theta`).
3. **Step 3:** Define a trainer for NNGLS through `geospaNN.ngls_train`, which requires:
 - Learning rate (`lr`): we recommend values in the range of $[0.005, 0.5]$ for our default ADMM optimizer.
 - Minimum value of validation loss drop (`min_delta`): any decrease of validation loss from last epoch larger than this value will be recorded as “significant gain.” If no significant gain happens in the last few epochs, early stopping will be triggered.
4. **Step 4:** Run the trainer with the dataloaders, which requires:
 - Dataloaders for training, validation, and testing (`data_train`, `data_val`, `data_test`).
 - Maximum number of epochs (`epoch_num`).
 - Initial epoch for spatial parameter estimation (`Update_init`).
 - Number of epochs between two spatial parameter updates (`Update_step`).
 - Size of each mini-batch (`batch_size`) and seed for batch splitting (`seed`).

2.3.3 Iterative Parameter Update

We note that in training NN-GLS, we isolate the updating of spatial parameters θ from that of the other weights parameters in the neural network architecture and update them iteratively as is demonstrated in figure 3. We control the frequency and starting point of updating θ to improve the efficiency and stability of the training. If θ is updated early when the non-spatial estimates remain unstable and biased, the newly estimated θ will be substantially influenced by these inaccuracies. This can lead to a feedback loop in which θ progressively deviates from its true value, ultimately resulting in non-convergence.

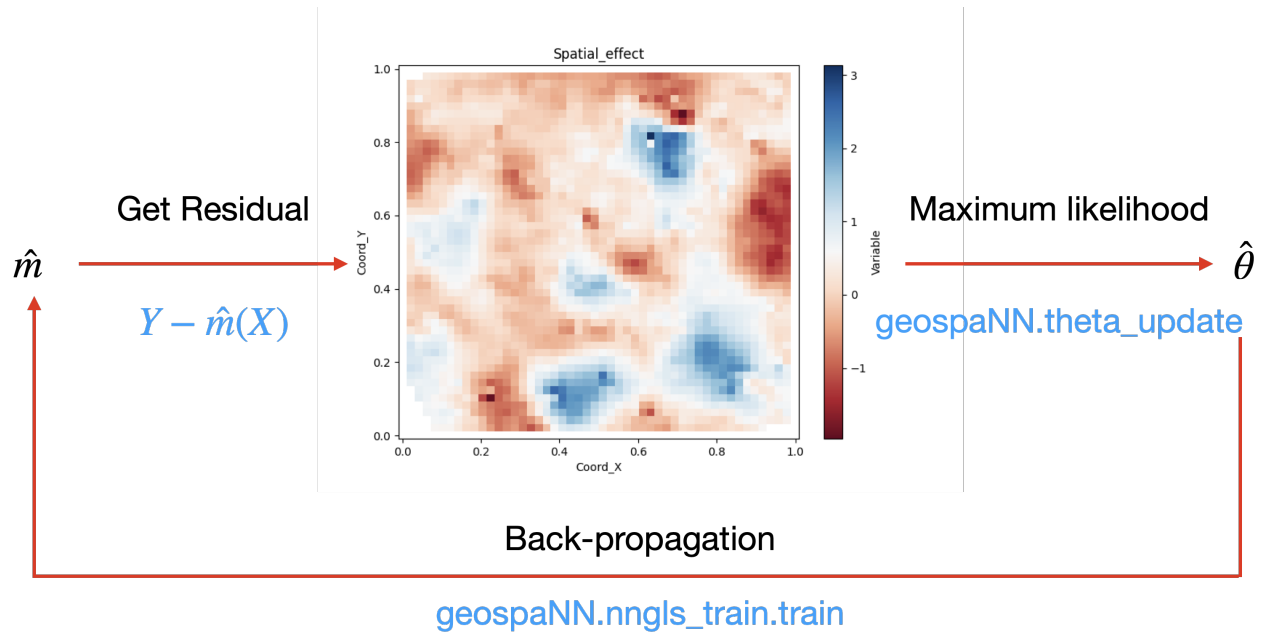


Figure 3: Iterative Parameter Update

2.4 Result Output

Our pipeline returns the outputs in the following aspects:

- The performance of training.
- The model-related performance, including prediction at new locations and mean effect estimation.
- Visualization tools for the functions mentioned above.

2.4.1 Performance of Training

To evaluate how `geospaNN` performed in the training process, use `geospaNN.plot_log` which requires:

- Optional true spatial parameters (`theta`).
- Path to save the figure (`path`).

```
geospaNN.plot_log(training_log, theta, path, save = True)
```

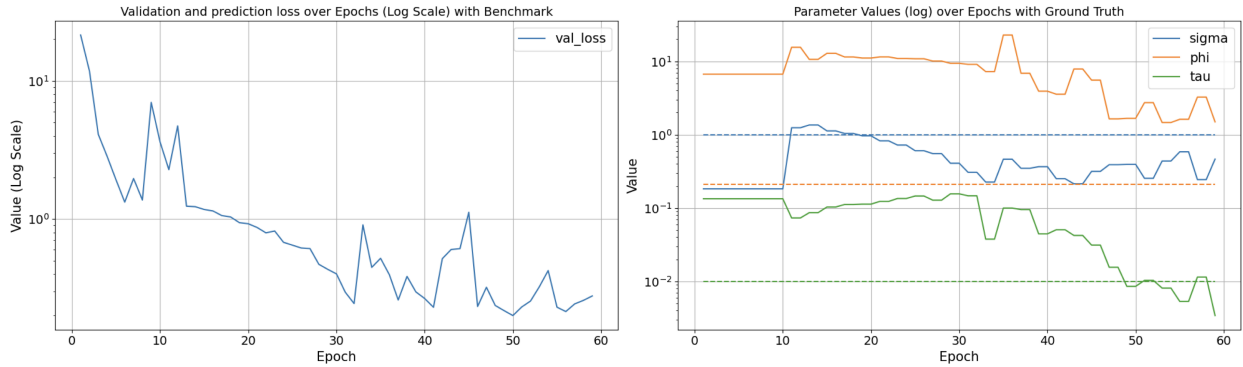


Figure 4: Training Log

2.4.2 Mean Function Estimation

Mean function estimation corresponds to estimating the $m(x)$ term in SPMM model, which represents the non-spatial relationship between Y and covariates X . To obtain the mean function estimation for covariates X , use the following method:

```
estimate = model.estimate(X)
```

Note that for the NN-GLS model, the mean function is estimated by the non-spatial architecture `mlp_nngls`, or can be equivalently called by `model.mlp`. Thus, `model.estimate(X)` is equivalent to `model.mlp(X)` and `mlp_nngls(X)`. The runnable code to produce figures 1-4 is attached in Appendix chunk 1.

2.4.3 Prediction and Intervals

To predict, use `geospaNN.predict()` as follows:

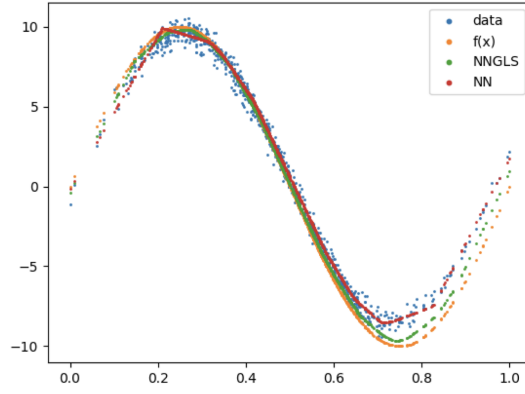


Figure 5: Mean Function Estimation

```
[test_predict, test_PI_U, test_PI_L] = model.predict(data_train, data_test, PI = True)
```

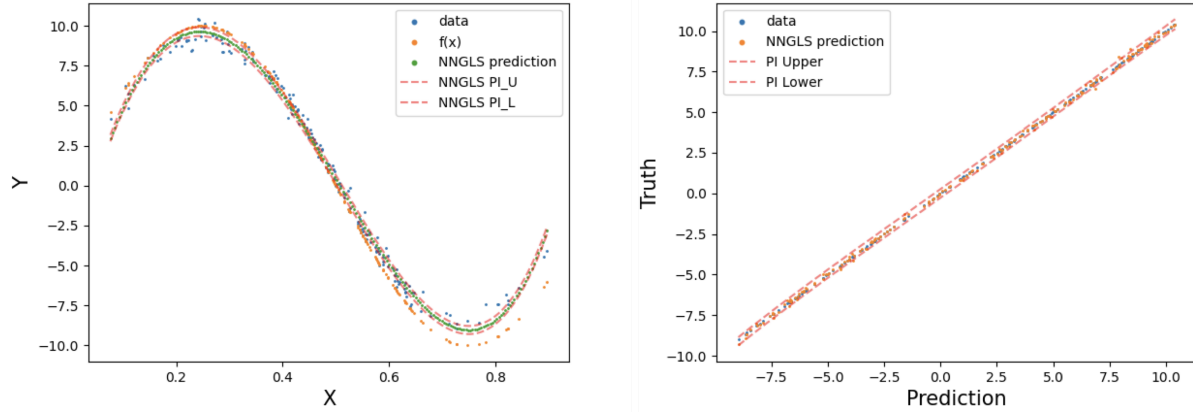


Figure 6: Prediction

NNGLS provides prediction and 95% prediction intervals by kriging. For new location(s) s_{new} and covariates $X(s_{new})$, prediction is obtained by evaluating the mean function estimate at the new covariate value:

$$\hat{Y}(s_{new}) = \hat{m}(X(s_{new})) + \widehat{w(s_{new})} + \epsilon(s_{new}).$$

The first term is computed by evaluating the mean function estimate at the new covariate value $X(s_{new})$. The second term is assumed a spatial process and incorporates uncertainty. In our model, conditioning on the training set, the spatial effect on new locations follows a Gaussian distribution whose mean and variance are obtained through kriging. Nearest neighbor kriging (NNGP approximation) is utilized here to guarantee the scalability of prediction. The runnable code to produce figures 5-6 is also attached in Appendix chunk 1 and a full version is available on https://wentaozhan1998.github.io/geospaNN-doc/Example_utils/Example_utils.

2.4.4 Partial Dependency Plot

When the covariates are multi-dimensional, **geospaNN** offers plots of partial dependence functions summarizing the marginal contribution of each covariate to the mean function. To illustrate this, we consider a simulation scenario where the mean is specified as a 5-dimensional Friedman’s function and run the same pipeline, the PDP can be generated by using `geospaNN.visualize.plot_PDP_list` (figure 7).

```
geospaNN.visualize.plot_PDP_list([funXY, mlp_nngls, mlp_nn],
                                ['Friedmans function', 'NNGLS', 'NN'],
                                X, split = True)
```

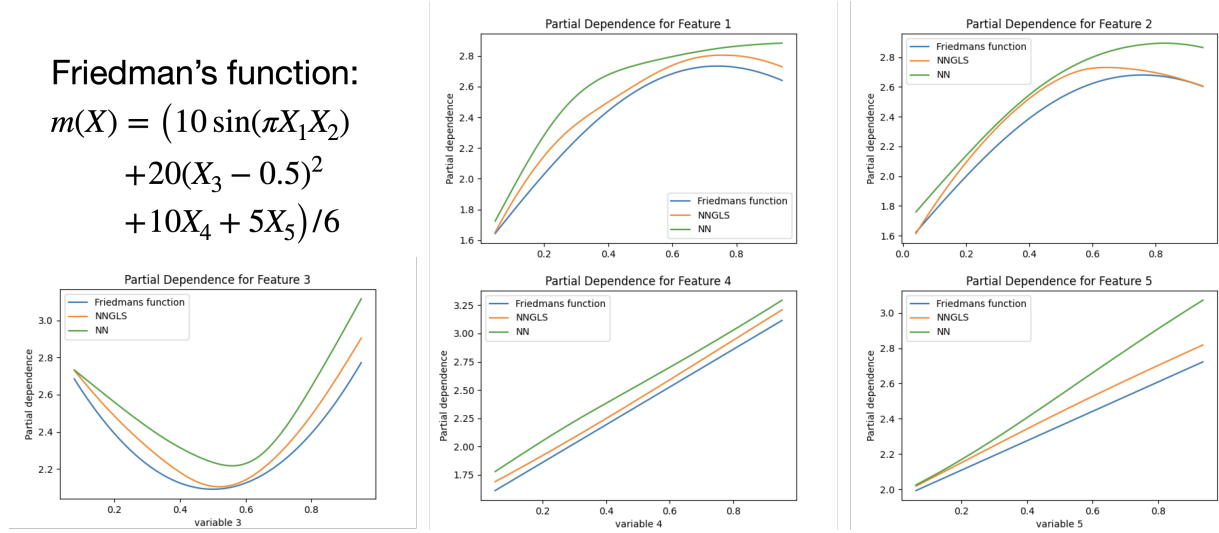


Figure 7: PDP

Partial Dependency Plot (PDP) is a commonly used visualization tool in the machine learning community to illustrate the marginal effect of a single covariate on the response for a high-dimensional model. Given a multi-dimensional function $m(X)$ and one covariate X_i , its PDP is generated by numerically integrating out the other covariates via:

$$PD(m, X_i) = \int m(X_1, \dots, X_p) P(X_{-i}) dX_{-i}.$$

Note that PDPs are only needed when the covariate dimension is greater than one. So the PDP function in **geospaNN** only works for multi-dimensional covariates and will not work for the 1-D example in this vignette. A full PDP script is attached in Appendix chunk 2 and is also available at https://wentaozhan1998.github.io/geospaNN-doc/Example_PDP/PDP/.

2.5 Additional Examples

2.5.1 Spatial Linear Mixed Model **geospaNN** also provides efficient solution to the spatial linear mixed model (SPLMM), which is a special case of SPMM, where the mean function is assumed to be linear:

$$Y(s) = X(s)\beta + w(s) + \epsilon(s).$$

In **geospaNN**, similar to that of NN-GLS, NNGP approximation is also used to simplify the computation and keep the linear complexity ($O(n)$) of solving SPLMM. To apply SPLMM, use the function

`geospaNN.linear_gls` on the dataset constructed by `geospaNN.make_graph` with the same arguments introduced in section 2.2.3:

```
model_linear = geospaNN.linear_gls(data_train)

estimate = model_linear.estimate(X)
[test_predict, test_PI_U, test_PI_L] = model_linear.predict(data_train, data_test, PI = True)
#### To get estimated covariance of linear coefficients, call:
model_linear.var
```

`model_linear` is in the same class as the NN-GLS model, both estimation and prediction (in figure 8) can be obtained using `model_linear.estimate()` and `model_linear.predict()`. Additionally, the function estimates the covariance matrix of linear coefficients β , which is contained in the additional attribute `model_linear.var`. SPLMM naturally extends the linear mixed model and is the most widely used model for data with spatial structure. The implementation of SPLMM here is equivalent to the R-packages BRISC (Saha and Datta 2018) and offers similar functionality for the Python users if efficient solution is wanted for large geospatial dataset. Code to produce figure 8 is provided in Appendix chunk 3 and also available on https://wentaozhan1998.github.io/geospaNN-doc/Example_linear/Example_linear/.

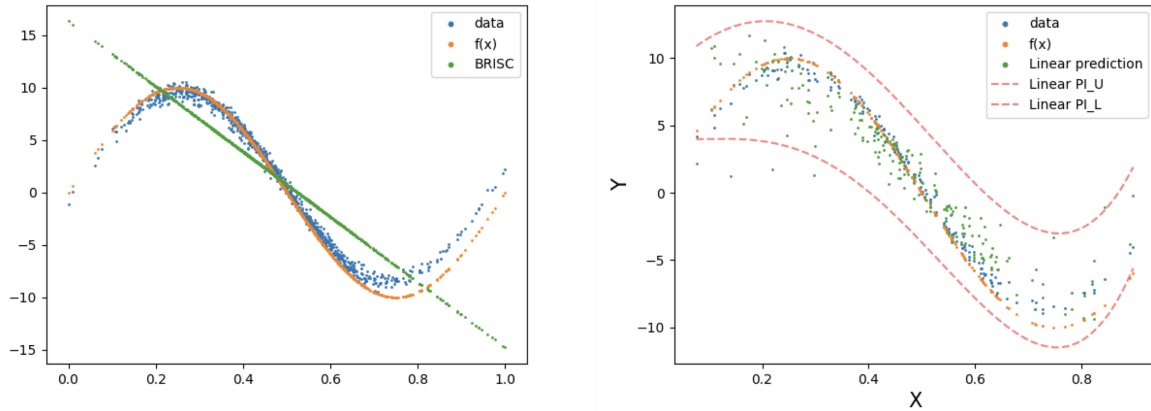


Figure 8: Estimation and Prediction of SPLMM

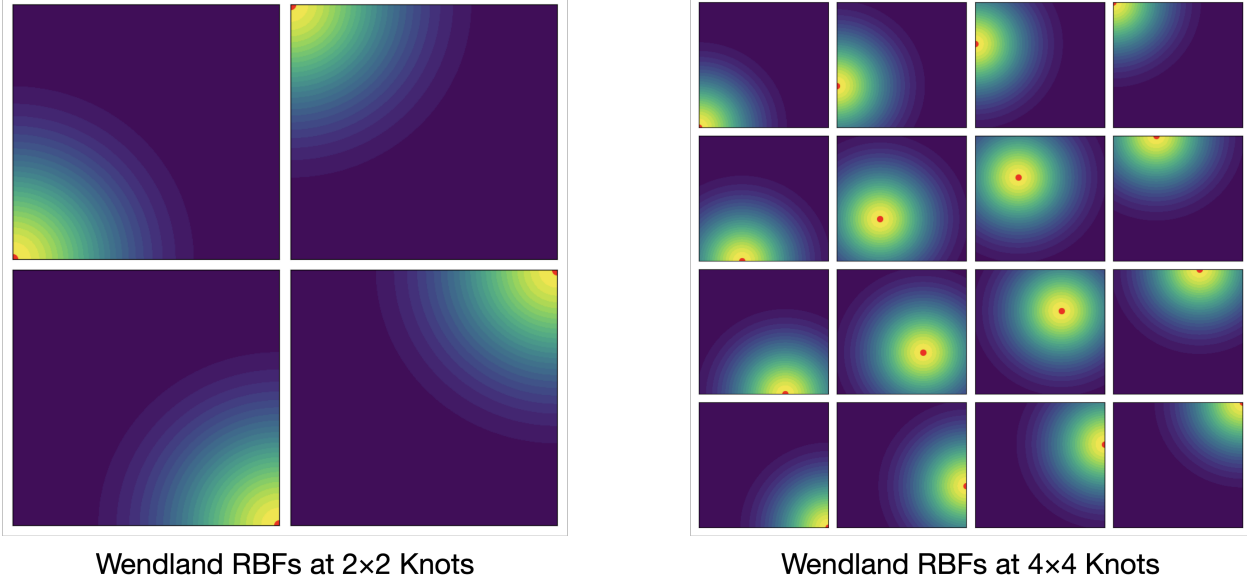
2.5.2 NN-GLS with added spatial features A common strategy for geospatial analysis of neural networks is to add spatial co-ordinates or some transformations (such as distances or spatial basis functions) as additional covariates. These added-spatial-features models incorporate all spatial information into the mean function. As a result, they are not able to separate non-spatial and spatial effect and can only be used for prediction. In `geospaNN`, we provide easy method for the added-spatial-features style neural networks, some of which have been proposed in literature but to our knowledge has not been formally implemented in any software.

To simply add spatial coordinates to the covariates X , use concatenation:

```
K = 2

data_add_train, data_add_val, data_add_test =
    geospaNN.split_data(torch.concat([X, coord], axis = 1),
        Y, coord, neighbor_size=nn, test_proportion=0.2)
```

Here K is the additional dimension brought by the spatial features. We also implemented 2-D radial basis functions using Wendland kernel as a more complex type of spatial feature. This spline-based approach was taken in a recent work DeepKriging (Chen et al. 2020) and is closely related to kriging and its associated variants such as fixed rank kriging. `geospaNN.coord_basis` can be used to generate the basis functions, where the user need to specify the resolutions for each set of knots by a list of integers. For example, `4 ** 2` in `num_basis` means the knots for the second set of basis functions are on the 4×4 grid within a unit square (see the figure below for an illustration of 2×2 and 4×4 knots).



```
K, coord_basis = geospaNN.coord_basis(coord, num_basis = [2 ** 2, 4 ** 2, 6 ** 2])

data_add_train, data_add_val, data_add_test =
    geospaNN.split_data(torch.concat([X, coord_basis], axis = 1),
        Y, coord, neighbor_size=nn,test_proportion=0.2)
```

Regardless of the added-spatial-features approach being used, the same code for training the non-spatial neural networks (section 2.3.1) can be used for added-spatial-features neural networks:

```
mlp_nn_add = torch.nn.Sequential(
    torch.nn.Linear(p+K, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
nn_add_model = geospaNN.nn_train(mlp_nn_add, lr=0.01, min_delta=0.001)
training_log = nn_add_model.train(data_add_train, data_add_val, data_add_test, seed = 2025)
```

and the prediction can be obtained by the mean function estimation of the model:

```
predict_nn_add = mlp_nn_add(data_add_test.x).reshape(-1)
```

In example provided for this section (figure 9), the MSE of prediction on test set for the first approach, i.e. adding coordinates as covariates, is 2.84, while for the second approach using basis functions is 1.70. For comparison, the MSE for NN-GLS is 0.45, which illustrates the advantage of Gaussian process-based

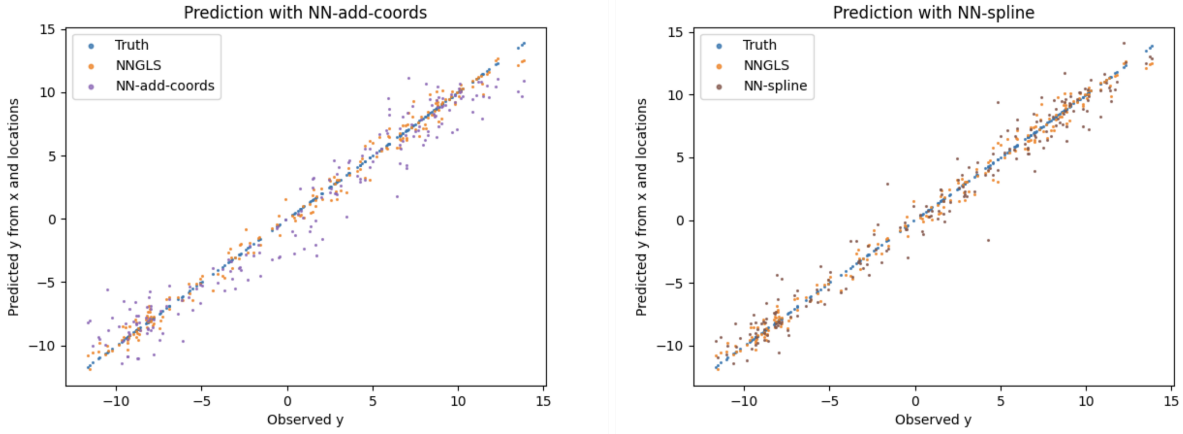


Figure 9: Prediction of added-spatial-features approaches vs NN-GLS

modeling. The full code to produce figure 9 and visualize Wendland RBF is provided in Appendix chunk 4 and also available on https://wentaozhan1998.github.io/geospaNN-doc/Example_addcovariates/Example_addcovariates/.

2.5.3 Application on time series analysis The applications of NN-GLS are not restricted to geospatial data. It was shown in Zhan and Datta (2024) that NN-GLS works well in scenarios where the covariance structure and data generating process are misspecified. To demonstrate how user can easily generate other types of dependent data and apply NN-GLS on it with `geospaNN`, we provide a time series example in this section. In this example, the mean function is still $m(x) = 10 \sin(2\pi x)$ and the noise is generated from an auto regressive (AR1) process, one of the most popular models used to account for temporal effect in time series analysis.

$$w(t) = \rho w(t-1) + \epsilon(t)$$

Given the arguments and data generated in section 2.2.1, the following steps are taken to generate the time series data:

1. Specifying parameters for the AR(1) process, which include:
 - AR(1) coefficient ρ (`rho`).
 - Standard deviation of noise σ_{AR} (`sigma_AR`).
 - Initial value X_0 (`x_0`).
2. Simulate white noise ϵ (`epsilon`).
3. Initialize and generate the AR(1) process `corerrr`. Define the time points by `coord` to fit into the previous training process.
4. Follow the other steps in section 2.3 to train NN-GLS.

```
#1
rho = np.sqrt(0.7)
sigma_AR = 5
x0 = 0
#2
np.random.seed(2024)
epsilon = np.random.normal(0, sigma_AR, n)
#3
corerrr = np.zeros(n)
```

```

corerr[0] = x0
for t in range(1, n):
    corerr[t] = rho * corerr[t-1] + epsilon[t]
coord = torch.zeros((n,2))
coord[:,0] = torch.tensor(range(n))/100

```

This generates an AR(1) process (random effect) with the coefficient $\rho^2 = 0.7$ (figure 10 left). The estimation performance of NN-GLS is illustrated in the right panel of figure 10. The MSE of mean-function estimations are 0.556 and 0.958 for NN-GLS and NN respectively. Appendix chunk 5 contains the code to simulate the time series data and produce figure 10, which is also available online at https://wentaozhan1998.github.io/geospaNN-doc/Example_time/Example_time/.

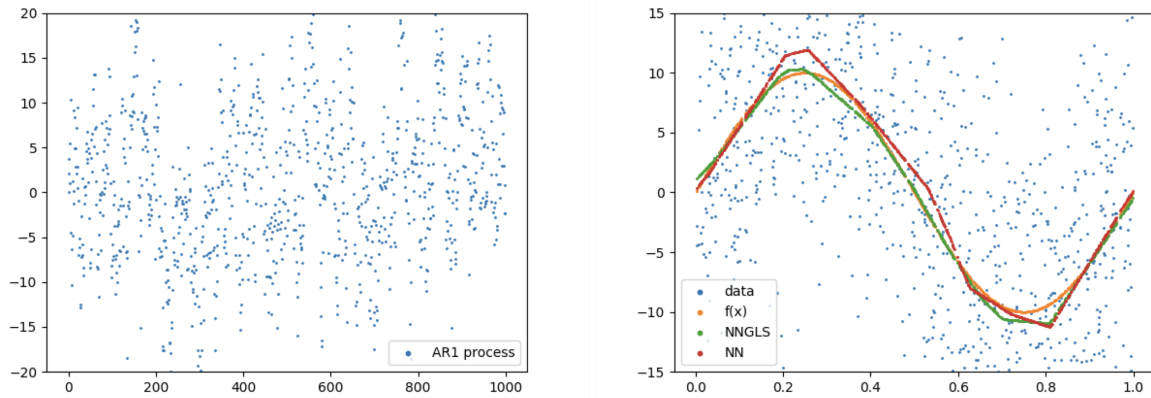


Figure 10: AR(1) process and the estimation performance

Appendix

Chunk 1

Available at https://wentaozhan1998.github.io/geospaNN-doc/Example_utils/Example_utils. Produce figures 1-6.

```

import torch
import geospaNN
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'

def f1(X): return 10 * np.sin(2 * np.pi * X)
p = 1;
funXY = f1

```

```

sigma = 1
phi = 0.3
Lambda = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), Lambda])

n = 1000
nn = 20

# Simulate data
## The case where X is not spatially independent
X, Y, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])

## The case where X is of spatial dependency
torch.manual_seed(2025)
_, _, _, X = geospaNN.Simulation(n, p, nn, funXY, torch.tensor([1, 5, 0.01]), range=[0, 1])
X = X.reshape(-1, p)
X = (X - X.min(dim=0).values) / (X.max(dim=0).values - X.min(dim=0).values)
torch.manual_seed(2025)
_, _, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])
Y = funXY(X).reshape(-1) + corerr

# Spatial surface plots
dict = {"Covariate": X, "Response": Y, "Spatial_effect": corerr}

for index, (name, variable) in enumerate(dict.items()):
    geospaNN.spatial_plot_surface(variable.detach().numpy().reshape(-1), coord.detach().numpy(),
                                  grid_resolution = 50, method = "CloughTocher", cmap = "RdBu",
                                  title = name, save_path = path, file_name = name + "_RdBu.png")

# Preprocess data
data = geospaNN.make_graph(X, Y, coord, nn, Ind_list = None)

np.random.seed(0)
data_train, data_val, data_test = geospaNN.split_data(X, Y, coord, neighbor_size=nn,
                                                         test_proportion=0.2, val_proportion=0.2)

# Fit NN
torch.manual_seed(2024)
mlp_nn = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
trainer_nn = geospaNN.nn_train(mlp_nn, lr=0.01, min_delta=0.001)
training_log = trainer_nn.train(data_train, data_val, data_test, epoch_num= 200,
                                batch_size = 60, seed = 2025)
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                                data_train.pos, neighbor_size=20)

```

```

# Fit NN-GLS
torch.manual_seed(2024)
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
model = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nngls,
                      theta=torch.tensor(theta0))
trainer_nngls = geospaNN.nngls_train(model, lr=0.1, min_delta=0.001)
training_log = trainer_nngls.train(data_train, data_val, data_test, epoch_num= 200,
                                   Update_init=10, Update_step=2,
                                   batch_size = 60, seed = 2025)

# Estimation and save the figure
estimate = model.estimate(X)
plt.clf()
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='NNGLS')
plt.scatter(X.detach().numpy(), mlp_nn(X).detach().numpy(), s=1, label='NN')
lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
plt.savefig(path + 'Estimation.png')

# Prediction and save the figure
[test_predict, test_PI_U, test_PI_L] = model.predict(data_train, data_test, PI = True)
x_np = data_test.x.detach().numpy().reshape(-1)
x_smooth = np.linspace(x_np.min(), x_np.max(), 200) # Create finer x-points
degree = 4
U_fit = np.polyfit(x_np, test_PI_U, degree)
L_fit = np.polyfit(x_np, test_PI_L, degree)
Pred_fit = np.polyfit(x_np, test_predict, degree)

y_smooth_U = np.polyval(U_fit, x_smooth)
y_smooth_L = np.polyval(L_fit, x_smooth)
y_smooth = np.polyval(Pred_fit, x_smooth)

plt.clf()
plt.scatter(data_test.x.detach().numpy(), data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(data_test.x.detach().numpy(), funXY(data_test.x.detach().numpy()), s=1, label='f(x)')
plt.scatter(data_test.x.detach().numpy(), test_predict.detach().numpy(), s=1, label='NNGLS prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='NNGLS PI_U', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='NNGLS PI_L', color = 'red', alpha = 0.5)
plt.xlabel("X", fontsize=15)
plt.ylabel("Y", fontsize=15)
lgnd = plt.legend()
for handle in lgnd.legend_handles[:3]:

```

```

        handle.set_sizes([10.0])
plt.savefig(path + "Prediction_PI.png")

plt.clf()
plt.scatter(data_test.y.detach().numpy(), data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(data_test.y.detach().numpy(), test_predict.detach().numpy(), s=1, label='NNGLS prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='PI Upper', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='PI Lower', color = 'red', alpha = 0.5)
plt.xlabel("Prediction", fontsize=15)
plt.ylabel("Truth", fontsize=15)
lgnd = plt.legend()
for handle in lgnd.legend_handles[:2]:
    handle.set_sizes([10.0])
plt.savefig(path + "Prediction_vs_PI.png")

# Training log
geospaNN.plot_log(training_log, theta, path, save = True)

```

Chunk 2

Available at https://wentaozhan1998.github.io/geospaNN-doc/Example_PDP/PDP/. Produces figure 7.

```

import torch
import geospaNN
import numpy as np
import pandas as pd
import random
import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'

def f5(X): return (10 * np.sin(np.pi * X[:, 0] * X[:, 1]) + 20 * (X[:, 2] - 0.5) ** 2
                  + 10 * X[:, 3] + 5 * X[:, 4]) / 6

p = 5;
funXY = f5

n = 1000
nn = 20
b = 10

sigma = 1
phi = 1
Lambda = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), Lambda])

# Simulate data
torch.manual_seed(2024)
X, Y, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, b])

# Preprocess data
random.seed(2024)
X, Y, coord, _ = geospaNN.spatial_order(X, Y, coord, method='max-min')

```



```

data = geospaNN.make_graph(X, Y, coord, nn)

torch.manual_seed(2024)
np.random.seed(0)
data_train, data_val, data_test =
    geospaNN.split_data(X, Y, coord, neighbor_size=nn, test_proportion=0.2)

# Plot PDP of truth function
PDP_truth = geospaNN.visualize.plot_PDP(f5, X, names = ["PDP"],
                                         save_path = path, save = True)

# Fit NN
torch.manual_seed(2024)
mlp_nn = torch.nn.Sequential(
    torch.nn.Linear(p, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
trainer_nn = geospaNN.nn_train(mlp_nn, lr=0.01, min_delta=0.001)
training_log = trainer_nn.train(data_train, data_val, data_test, seed = 2)
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                                data_train.pos, neighbor_size=20)

# Plot PDP of NN fit
PDP_NN = geospaNN.visualize.plot_PDP(mlp_nn, X, names = ["PDP"],
                                       save_path = path, save = True)

# Fit NN-GLS
torch.manual_seed(2024)
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
nngls = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nngls, theta=torch.tensor(theta0))
trainer_nngls = geospaNN.nngls_train(nngls, lr=0.01, min_delta=0.001)
training_log = trainer_nngls.train(data_train, data_val, data_test,
                                   Update_init=20, Update_step=10, seed = 2)

# Plot PDP of NN-GLS fit
geospaNN.visualize.plot_PDP(mlp_nngls, X, names = ["PDP"],
                           save_path = path, save = True)

# Plot PDPs
geospaNN.visualize.plot_PDP_list([funXY, mlp_nngls, mlp_nn],
                                 ['Friedmans function', 'NNGLS', 'NN'], X,
                                 split = True, save_path = path, save = True)

```

Chunk 3

Available at https://wentaozhan1998.github.io/geospaNN-doc/Example_linear/Example_linear/. Produces figure 8.

```
import torch
import geospaNN
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'

def f1(X): return 10 * np.sin(2*np.pi * X)
p = 1;
funXY = f1

n = 1000
nn = 20

sigma = 1
phi = 0.3
Lambda = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), Lambda])

# Simulate data
torch.manual_seed(2025)
_, _, _, X = geospaNN.Simulation(n, p, nn, funXY, torch.tensor([1, 5, 0.01]),
                                range=[0, 1])

X = X.reshape(-1,p)
X = (X - X.min())/(X.max() - X.min())
torch.manual_seed(2025)
_, _, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])
Y = funXY(X).reshape(-1) + corerr

# Preprocess data
data = geospaNN.make_graph(X, Y, coord, nn, Ind_list = None)

torch.manual_seed(2024)
np.random.seed(0)
data_train, data_val, data_test =
    geospaNN.split_data(X, Y, coord, neighbor_size=nn,
                        test_proportion=0.2, val_proportion=0.2)

# Fit SPLMM
model_linear = geospaNN.model.linear_gls(data_train)

# Estimation and save the figure
estimate = model_linear.estimate(X)
plt.clf()
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='Linear estimation')
```

```

lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
plt.savefig(path + 'Estimation_linear.png')

# Confidence interval and save the figure
[CI_U, CI_L] = geospaNN.confidence_interval(model_linear, X, rep = 200,
                                           quantiles = [97.5, 2.5])
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='Linear estimation')
plt.scatter(X.detach().numpy(), CI_U, s=1, label='CI_U')
plt.scatter(X.detach().numpy(), CI_L, s=1, label='CI_L')
lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
plt.savefig(path + "Prediction_linear_CI.png")

# Prediction interval and save the figure
[test_predict, test_PI_U, test_PI_L] =
    model_linear.predict(data_train, data_test, PI = True)

x_np = data_test.x.detach().numpy().reshape(-1)
x_smooth = np.linspace(x_np.min(), x_np.max(), 200) # Create finer x-points
degree = 4
U_fit = np.polyfit(x_np, test_PI_U, degree)
L_fit = np.polyfit(x_np, test_PI_L, degree)
Pred_fit = np.polyfit(x_np, test_predict, degree)
y_smooth_U = np.polyval(U_fit, x_smooth)
y_smooth_L = np.polyval(L_fit, x_smooth)
y_smooth = np.polyval(Pred_fit, x_smooth)

x_test = data_test.x.detach().numpy()
plt.clf()
plt.scatter(x_test, data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(x_test, funXY(data_test.x.detach().numpy()), s=1, label='f(x)')
plt.scatter(x_test, test_predict.detach().numpy(), s=1, label='Linear prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='Linear PI_U', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='Linear PI_L', color = 'red', alpha = 0.5)
plt.xlabel("X", fontsize=15)
plt.ylabel("Y", fontsize=15)
lgnd = plt.legend()
for handle in lgnd.legend_handles[:3]:
    handle.set_sizes([10.0])
plt.savefig(path + "Prediction_linear.png")

```

Chunk 4

Available at https://wentaozhan1998.github.io/geospaNN-doc/Example_addcovariates/Example_addcovariates/. Produces figure 9.

```

import torch
import geospaNN
import numpy as np
import pandas as pd
import random
import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'

def f1(X): return 10 * np.sin(2 * np.pi * X)
p = 1;
funXY = f1

n = 1000
nn = 20
b = 10

sigma = 5
phi = 0.3
Lambda = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), Lambda])

# Simulate data
torch.manual_seed(2025)
X, Y, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, b])

# Preprocess data
random.seed(2024)
X, Y, coord, _ = geospaNN.spatial_order(X, Y, coord, method='max-min')
data = geospaNN.make_graph(X, Y, coord, nn)

np.random.seed(0)
data_train, data_val, data_test = geospaNN.split_data(X, Y, coord, neighbor_size=nn,
                                                         test_proportion=0.2)

# Fit NN
torch.manual_seed(2024)
mlp_nn = torch.nn.Sequential(
    torch.nn.Linear(p, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
nn_model = geospaNN.nn_train(mlp_nn, lr=0.01, min_delta=0.001)
training_log = nn_model.train(data_train, data_val, data_test, seed = 2024)
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                                data_train.pos, neighbor_size=20)
model = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nn,
                       theta=torch.tensor(theta0))
predict_nn = model.predict(data_train, data_test)

```

```

# Fit NN-GLS
torch.manual_seed(2024)
mlp_nn_gls = torch.nn.Sequential(
    torch.nn.Linear(p, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
model_nn_gls = geospaNN.ngls(p=p, neighbor_size=nn, coord_dimensions=2,
                             mlp=mlp_nn_gls, theta=torch.tensor(theta0))
ngls_model = geospaNN.ngls_train(model_nn_gls, lr=0.1, min_delta=0.001)
training_log = ngl_model.train(data_train, data_val, data_test,
                               Update_init=20, Update_step=10, seed = 2024)
theta_hat = geospaNN.theta_update(mlp_nn_gls(data_train.x).squeeze() - data_train.y,
                                   data_train.pos, neighbor_size = 20)
model = geospaNN.ngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nn_gls,
                      theta=torch.tensor(theta_hat))
predict_nn_gls = model.predict(data_train, data_test)

# Similar fitting for added-spatial-feature
np.random.seed(0)
data_add_train, data_add_val, data_add_test =
    geospaNN.split_data(torch.concat([X, coord], axis = 1), Y, coord,
                        neighbor_size=nn, test_proportion=0.2)
torch.manual_seed(2025)
mlp_nn_add = torch.nn.Sequential(
    torch.nn.Linear(p+2, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
nn_add_model = geospaNN.nn_train(mlp_nn_add, lr=0.01, min_delta=0.001)
training_log = nn_add_model.train(data_add_train, data_add_val, data_add_test, seed = 2024)
predict_nn_add = mlp_nn_add(data_add_test.x).detach().numpy().reshape(-1)

# Similar fitting for added-spatial-feature
## Spline generation
K, phi_temp = geospaNN.coord_basis(coord, num_basis = [2 ** 2, 4 ** 2, 6 ** 2])
np.random.seed(0)
data_DK_train, data_DK_val, data_DK_test =
    geospaNN.split_data(torch.concat([X, phi_temp], axis = 1).float(), Y, coord,
                        neighbor_size=nn, test_proportion=0.2)

torch.manual_seed(2024)
mlp_nn_DK = torch.nn.Sequential(
    torch.nn.Linear(p+K, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1))
nn_DK_model = geospaNN.nn_train(mlp_nn_DK, lr=0.01, min_delta=0.001)

```

```

training_log = nn_DK_model.train(data_DK_train, data_DK_val, data_DK_test, seed = 2024)
predict_DK = mlp_nn_DK(data_DK_test.x).detach().numpy().reshape(-1)

estimate_nn = mlp_nn(data_test.x).detach().numpy().reshape(-1)
y_test = data_test.y.detach().numpy()
plt.clf()
plt.scatter(y_test, data_test.y.detach().numpy(), s=1, alpha = 0.5, label='Truth')
plt.scatter(y_test, predict_nngls, s=1, alpha = 0.5, label='NNGLS')
plt.scatter(y_test, estimate_nn, s=1, alpha = 0.5, label='NN estimate')
plt.scatter(y_test, predict_nn, s=1, alpha = 0.5, label='NN + kriging')
plt.scatter(y_test, predict_nn_add, s=1, alpha = 0.5, label='NN-add-coords')
plt.scatter(y_test, predict_DK, s=1, alpha = 0.5, label='NN-spline')
lgnd = plt.legend(fontsize=10)
plt.xlabel('Observed y', fontsize=10)
plt.ylabel('Predicted y from x and locations', fontsize=10)
plt.title('Prediction')

for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
plt.savefig(path + 'Prediction.png')

print(f"RMSE nn-estimate: {torch.mean((data_test.y - estimate_nn)**2):.2f}")
print(f"RMSE nngls: {torch.mean((data_test.y - predict_nngls)**2):.2f}")
print(f"RMSE nn+kriging: {torch.mean((data_test.y - predict_nn)**2):.2f}")
print(f"RMSE nn-add-coordinates: {torch.mean((data_test.y - predict_nn_add)**2):.2f}")
print(f"RMSE nn-spline: {torch.mean((data_test.y - predict_DK)**2):.2f}")

# Visualize Wendland RBF
## Define the Wendland C2 RBF function
def wendland_c2(r):
    return ((1 - r)**4 * (4 * r + 1)) * (r < 1)

## Define the grid for visualization
x = np.linspace(0, 1, 100)
y = np.linspace(0, 1, 100)
X, Y = np.meshgrid(x, y)

## Define a 2x2 grid of knots within the unit square
knots = [(0, 0), (0, 1), (1, 0), (1, 1)]

## Create figure with 2x2 subplots
fig, axes = plt.subplots(2, 2, figsize=(8, 8))

## Plot Wendland RBF centered at each knot
for ax, (kx, ky) in zip(axes.flat, knots):
    R = np.sqrt((X - kx)**2 + (Y - ky)**2) # Compute radial distance
    Z = wendland_c2(R) # Apply Wendland function

    c = ax.contourf(X, Y, Z, levels=30, cmap='viridis')
    ax.scatter([kx], [ky], color='red', marker='o', s=100, label="Knot")
    ax.set_xticks([])
    ax.set_yticks([])

```

```

fig.suptitle("Wendland RBF at 2x2 knots of a Unit Square", fontsize=14)
plt.tight_layout()
plt.savefig(path + 'Knots_2x2.png')

## Define a 4x4 grid of knots within the unit square
num_knots = 4
knot_positions = np.linspace(0, 1, num_knots)
knots = [(kx, ky) for kx in knot_positions for ky in knot_positions]

## Create figure with 4x4 subplots
fig, axes = plt.subplots(num_knots, num_knots, figsize=(10, 10))

## Plot Wendland RBF centered at each knot
for ax, (kx, ky) in zip(axes.flat, knots):
    R = np.sqrt((X - kx)**2 + (Y - ky)**2) # Compute radial distance
    Z = wendland_c2(R) # Apply Wendland function

    c = ax.contourf(X, Y, Z, levels=30, cmap='viridis')
    ax.scatter([kx], [ky], color='red', marker='o', s=50, label="Knot")
    ax.set_xticks([])
    ax.set_yticks([])

fig.suptitle("Wendland RBFs at 4x4 Knots in Unit Square", fontsize=14)
plt.tight_layout()
plt.savefig(path + 'Knots_4x4.png')

```

Chunk 5

Available at https://wentaozhan1998.github.io/geospaNN-doc/Example_time/Example_time/. Produces figure 10.

```

import random
import torch
import geospaNN
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'

def f1(X): return 10 * np.sin(2 * np.pi * X)
p = 1;
funXY = f1

sigma = 5
phi = 0.3
Lambda = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), Lambda])

n = 1000
nn = 20
b = 10

```

```

# Simulate data
rho = np.sqrt(0.7) # AR(1) coefficient #0.7
sigma_AR = 5 # Standard deviation of noise
x0 = 0 # Initial value

## Simulate white noise
np.random.seed(2024) #2025
epsilon = np.random.normal(0, sigma_AR, n)

## Initialize the process
corerr = np.zeros(n)
corerr[0] = x0

## Generate AR(1) process
for t in range(1, n):
    corerr[t] = rho * corerr[t-1] + epsilon[t]

coord = torch.zeros((n,2))
coord[:,0] = torch.tensor(range(n))/100

torch.manual_seed(2024)
X, _, _, _ = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, b])
Y = funXY(X) + corerr.reshape((-1,1))
Y = Y.reshape(-1)

# Preprocess data
random.seed(2024)
X, Y, coord, _ = geospaNN.spatial_order(X.float(), Y.float(), coord, method='max-min')
data = geospaNN.make_graph(X, Y, coord, nn)

np.random.seed(0)
data_train, data_val, data_test =
    geospaNN.split_data(X, Y, coord, neighbor_size=nn, test_proportion=0.2)

# Fit NN
torch.manual_seed(2024)
mlp_nn = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
trainer_nn = geospaNN.nn_train(mlp_nn, lr=0.01, min_delta=0.001)
training_log = trainer_nn.train(data_train, data_val, data_test, seed = 2025)
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                                data_train.pos, neighbor_size=20)

# Fit NN-GLS
torch.manual_seed(2024)
mlp_nngls = torch.nn.Sequential(

```



```

    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
model = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nngls,
                      theta=torch.tensor(theta0))
trainer_nngls = geospaNN.nngls_train(model, lr=0.1, min_delta=0.001)
training_log = trainer_nngls.train(data_train, data_val, data_test, epoch_num= 200,
                                   Update_init=20, Update_step=5, seed = 2025)

# Estimation and save the figure
estimate = model.estimate(X)
plt.clf()
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='NNGLS')
plt.scatter(X.detach().numpy(), mlp_nn(X).detach().numpy(), s=1, label='NN')
lgnd = plt.legend()
plt.ylim([-15, 15])
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
plt.savefig(path + 'Estimation_AR.png')

# Plot AR(1) process
plt.clf()
plt.scatter(range(n), corerr, s=1, label='AR1 process')
plt.ylim([-20, 20])
lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
plt.savefig(path + 'AR.png')

# Compute loss
def RMSE(x,y):
    x = x.reshape(-1)
    y = y.reshape(-1)
    n = x.shape[0]
    return(np.sqrt(np.sum(np.square(x-y))/n))
RMSE(estimate.detach().numpy(), funXY(X).reshape(-1).detach().numpy())
RMSE(mlp_nn(X).detach().numpy(), funXY(X).reshape(-1).detach().numpy())

```

- Chen, Wanfang, Yuxiao Li, Brian J Reich, and Ying Sun. 2020. “Deepkriging: Spatially Dependent Deep Neural Networks for Spatial Prediction.” *arXiv Preprint arXiv:2007.11972*.
- Datta, Abhirup, Sudipto Banerjee, Andrew O Finley, and Alan E Gelfand. 2016. “On Nearest-Neighbor Gaussian Process Models for Massive Spatial Data.” *Wiley Interdisciplinary Reviews: Computational Statistics* 8 (5): 162–71.
- Katzfuss, Matthias, and Joseph Guinness. 2021. “A General Framework for Vecchia Approximations of Gaussian Processes.” *Statistical Science* 36 (1): 124–41.
- Saha, Arkajyoti, and Abhirup Datta. 2018. “BRISC: Bootstrap for Rapid Inference on Spatial Covariances.” *Stat* 7 (1): e184.

Zhan, Wentao, and Abhirup Datta. 2024. “Neural Networks for Geospatial Data.” *Journal of the American Statistical Association*, no. just-accepted: 1–21.