NAULearningDataScience

Ryan E Lima

2025-10-24

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

2 Kendall rank correlation coefficient (Kendall's tau)

In statistics, the Kendall rank correlation coefficient, commonly referred to as Kendall's tau ( ), is a statistic used to measure the ordinal association between two measured quantities. A **T test** is a non-parametric hypothesis test for statistical dependence based on the T coefficient. It is a measure of rank correlation: the similarity of the orderings of data when ranked by each of the quantities. It is named after Maurice Kendall, who developed it in 1938, though Gustav Fechner has proposed a similar measure in the context of time series in 1897.

Kendall's motification was to create a rubust, intuitive measure of association between two rankings–one that was: - non parameteric – meaning it made no assumptions about the distribution of the data - intuative in interpretation – meaning it could be easily understood (based on concordant and discordant pairs) and - suitable for ordinal or ranked data (like preferences, ratings, or scores).

Before Kendall's tau, other correlation measures like Pearson's correlation coefficient were commonly used, but they assumed linear relationships and required interval or ratio data. Kendall's tau provided a way to assess relationships in a more flexible manner, especially for non-linear or non-parametric data.

In many real-world problems, especially in decision analysis (also social sciences) data are often ordinal – things we can rank but not measure on a precise numerical scale.

For example: a hydrologist looking to build a groundwater recharge project, might want to rank potential sites based on suitability critiera, and rank them in suitability from 1 (low suitability) to 5 or 10 (high suitability). Kendall's tau would allow the hydrologist to assess the association between different ranking criteria (like soil type, proximity to water sources, land use, etc.) without making assumptions about the underlying data distribution.

How it works

Imagine two people rank the same set of sites independently based on their expert opinion of suitability for groundwater recharge.

site	Rater A Rank	Rater B Rank
S1	1	2
S2	2	1

site	Rater A Rank	Rater B Rank
S3	3	4
S4	4	3
S5	5	5

Kendall's tau looks at **all possible pairs of sites** (S1 vs S2, S1 vs S3, … S4 vs S5) and asks: - Do the two analysts agree on which site should be ranked higher?

For any pair of sites (i, j): - The pair is **concordant** if both analysts put the same site higher. (Example: if A says $S2$ better than $S5$, and B also says $S2$ better than $S5$.) - The pair is **discordant** if the analysts disagree about which one is better.
(Example: A says $S1$ better than $S4$, but B says $S4$ better than $S1$.) - (Ties are possible in general, though not shown in this simple example. We handle those with slight variations of tau.)

Intuition: - If most pairs are concordant $\rightarrow \tau$ is close to $+1$ (the rankings mostly agree). - If most pairs are discordant $\rightarrow \tau$ is close to -1 (the rankings mostly disagree / almost inverted). - If agreement and disagreement are about equal $\rightarrow \tau$ is near 0.

2.1 The "probability" view

One clean way to define Kendall's tau is:

$$\tau = P(\text{concordant}) - P(\text{discordant})$$

Here $P(\text{concordant})$ means:
"Out of all possible pairs of items, what fraction of pairs are concordant?"

In other words, these are not probabilities in the sense of randomness over repeated experiments — they are proportions over all $\frac{n(n-1)}{2}$ pairs in *this* dataset.

So you can read τ as: > "If I pick two items at random, how much more likely is it that the two rankings agree on their order than disagree?"

That's the core interpretation.

2.2 The counting (pairwise) formula

Let: - n = number of items being ranked
- C = number of concordant pairs
- D = number of discordant pairs
- $T = \frac{n(n-1)}{2}$ = total number of distinct pairs

Then Kendall's tau can be written as:

$$\tau = \frac{C - D}{T} = \frac{C - D}{\frac{1}{2}n(n-1)}$$

This is the same as the "probability" version, just written in terms of counts instead of proportions: - $\frac{C}{T}$ is $P(\text{concordant})$ - $\frac{D}{T}$ is $P(\text{discordant})$

So:

$$\tau = \frac{C}{T} - \frac{D}{T}$$

In practice, we'll compute C and D from two ranked lists, calculate τ, and then visualize where disagreements are happening spatially or across alternatives.

Next, we'll implement this calculation in Python, both "by hand" (to see C and D) and using `scipy.stats.kendalltau`.

Now lets explore how to calculate Kendall's tau using Python.

```python
# import pandas to create an manipulate dataframes
import pandas as pd

# create a sample dataframe with rankings from two analysts
data = pd.DataFrame({
    "Site": ["A", "B", "C", "D", "E", "F"],
    "Rank_Analyst1": [1, 2, 3, 4, 5, 6],  # Analyst 1 thinks A>B>C>D>E>F
    "Rank_Analyst2": [1, 3, 2, 4, 6, 5]   # Analyst 2 mostly agrees, but swaps B/C and E/F
})

data
```

	Site	Rank_Analyst1	Rank_Analyst2
0	A	1	1
1	B	2	3
2	C	3	2
3	D	4	4
4	E	5	6
5	F	6	5

Ok, now lets create a function that checks all possible pairings rankings to determine concordant (agreeing) and discordant (disagreeing) pairs.

```
# itertools is a useful library for creating combinations and permutations
import itertools

def kendall_concordance_table(df, col_x, col_y):
    """
    Create a table showing concordant and discordant pairs between two rankings.
    df: DataFrame with rankings
    col_x: column name for first ranking
    col_y: column name for second ranking
    Returns a DataFrame with pairwise comparisons and counts of concordant/discordant pairs.
    """
    pairs_info = [] # to store info about each pair
    C = 0  # concordant = they agree on order
    D = 0  # discordant = they disagree on order
    for (i, j) in itertools.combinations(df.index, 2): # for all unique pairs of indices (i,
        x_i = df.loc[i, col_x] # x_i is the rank of item i in ranking x
        x_j = df.loc[j, col_x] # x_j is the rank of item j in ranking x
        y_i = df.loc[i, col_y] # y_i is the rank of item i in ranking y
        y_j = df.loc[j, col_y] # y_j is the rank of item j in ranking y
        site_i = df.loc[i, "Site"] # get site names for reporting for item i
        site_j = df.loc[j, "Site"] # get site names for reporting for item j

        # Compare pair ordering in each ranking
        diff_x = x_i - x_j # difference in ranking for pair (i, j) in ranking x
        diff_y = y_i - y_j # difference in ranking for pair (i, j) in ranking y

        # If both differences have same sign -> concordant
        # If opposite sign -> discordant
        # If diff_x or diff_y == 0, that's a tie (we'll just mark it)
        if diff_x * diff_y > 0:
            relation = "concordant"
            C += 1
        elif diff_x * diff_y < 0:
            relation = "discordant"
            D += 1
        else:
            relation = "tie"

        pairs_info.append({
            "Pair": f"{site_i}-{site_j}",
            f"Order in {col_x}": "i<j" if diff_x < 0 else "i>j",
            f"Order in {col_y}": "i<j" if diff_y < 0 else "i>j",
            "Relation": relation
        })

    pairs_df = pd.DataFrame(pairs_info)
    return pairs_df, C, D

# Now let's use the function on our sample data
pairs_df, C, D = kendall_concordance_table(data, "Rank_Analyst1", "Rank_Analyst2")
print(f"Concordant pairs (C): {C}, Discordant pairs (D): {D}") # print out the counts of conc
pairs_df # print the output table from the function: kendall_concordance_table()
```

```
Concordant pairs (C): 13, Discordant pairs (D): 2
```

	Pair	Order in Rank_Analyst1	Order in Rank_Analyst2	Relation
0	A-B	i<j	i<j	concordant
1	A-C	i<j	i<j	concordant
2	A-D	i<j	i<j	concordant
3	A-E	i<j	i<j	concordant
4	A-F	i<j	i<j	concordant
5	B-C	i<j	i>j	discordant
6	B-D	i<j	i<j	concordant
7	B-E	i<j	i<j	concordant
8	B-F	i<j	i<j	concordant
9	C-D	i<j	i<j	concordant
10	C-E	i<j	i<j	concordant
11	C-F	i<j	i<j	concordant
12	D-E	i<j	i<j	concordant
13	D-F	i<j	i<j	concordant
14	E-F	i<j	i>j	discordant

2.3 Compute Kendall's Tau from first principles

Next we will compute Kendall's tau manually using Python.

remember that tau is the difference between the probability of concordant and discordant pairs.

For n items, the total number of distince pairs is given by the formula: $T = \frac{n(n-1)}{2}$.

Then, we can use the output of the function above which counted the number of concordant and discordant pairs to compute kendall's tau.

```python
import numpy as np # for numerical operations we use numpy

n = len(data) # number of items being ranked
total_pairs = n * (n - 1) / 2 # total number of distinct pairs T
tau_manual = (C - D) / total_pairs # Kendall's tau formula

print(f" C = {C}, D = {D}, Total pairs (T): {total_pairs}")
print(f"Kendall's tau (manual calculation): {tau_manual:.3f}")
```

```
 C = 13, D = 2, Total pairs (T): 15.0
Kendall's tau (manual calculation): 0.733
```

Interpreting the results

if tau is close to +1, the rankings mostly agree if tau is close to 0.5, mostly agree but with notable flips if tau is close to -1, the rankings mostly disagree if tau is close to -0.5, mostly disagree but with agreements if tau is close to 0, there is little association between the rankings

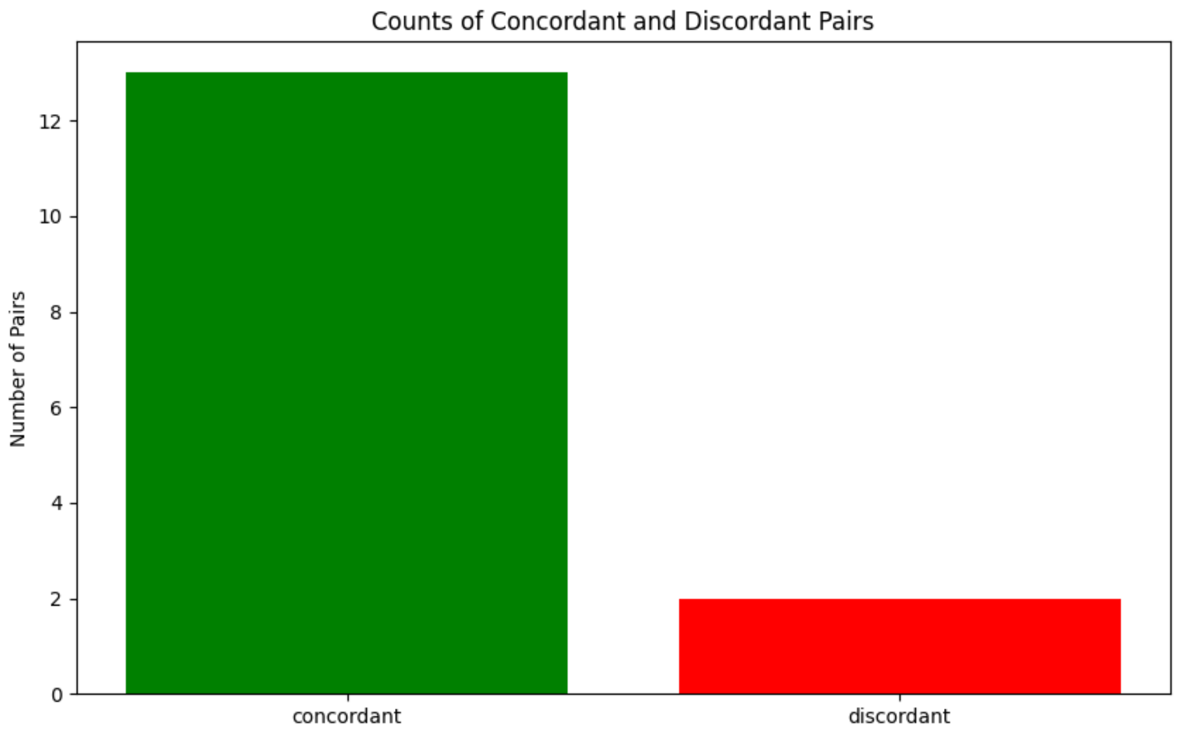
For this synthetic dataset we should see τ as high but not 1 because those B vs C and E vs F swaps create discordance.

Next lets visualize the agreement and disagreement between the two rankings.

To do this we will plot…

```python
import matplotlib.pyplot as plt

color_map = { 'concordant': 'green', 'discordant': 'red', "tie": 'gray' }

plt.figure(figsize=(10, 6))
pair_counts = pairs_df["Relation"].value_counts()
bars = plt.bar(pair_counts.index, pair_counts.values,
               color=[color_map[r] for r in pair_counts.index])
```

```
plt.ylabel("Number of Pairs")
plt.title("Counts of Concordant and Discordant Pairs")
plt.show()
```



So looking at the figure we can see that most of the pairs are concordant, we have a few discordant pairs (in red) and no ties (gray). We got a kendall's tau of 0.733 which indicates a strong positive association between the two rankings.

2.4 Compare manual calculation to scipy.stats

Now that we understand the basic calculation of Kendall's tau, lets try to use the scipy.stats version of kendall's tau to see if we get the same results and how our manual calculation compares to the built-in function in scipy.stats

```
from scipy.stats import kendalltau # import kendalltau from scipy.stats

# recall the structure of our data
print(data.columns) # show column names
data.head(5) # show first 5 rows of data
```

```
Index(['Site', 'Rank_Analyst1', 'Rank_Analyst2'], dtype='object')
```

	Site	Rank_Analyst1	Rank_Analyst2
0	A	1	1
1	B	2	3
2	C	3	2
3	D	4	4
4	E	5	6

```
tau_scipy, p_value = kendalltau(data["Rank_Analyst1"], data["Rank_Analyst2"])
print(f"Kendall's tau (scipy.stats): {tau_scipy:.3f}, p-value: {p_value:.3f}")
```

```
Kendall's tau (scipy.stats): 0.733, p-value: 0.056
```

Notice that both our manual calculation and scipy's kendalltau function give the same result of approximately 0.733, confirming the correctness of our manual implementation. But the scipy function also provides a p-value for testing the hypothesis of no association ($\tau = 0$)

So the scipy version does two things:

1. It computes Kendall's tau using an efficient algorithm measuring the strength of monotonic association between two rankings.
2. It provides a p-value for testing the null hypothesis that there is no association between the two rankings (i.e., $\tau = 0$). A low p-value (typically < 0.05) indicates that we can reject the null hypothesis and conclude that there is a statistically significant association between the rankings.

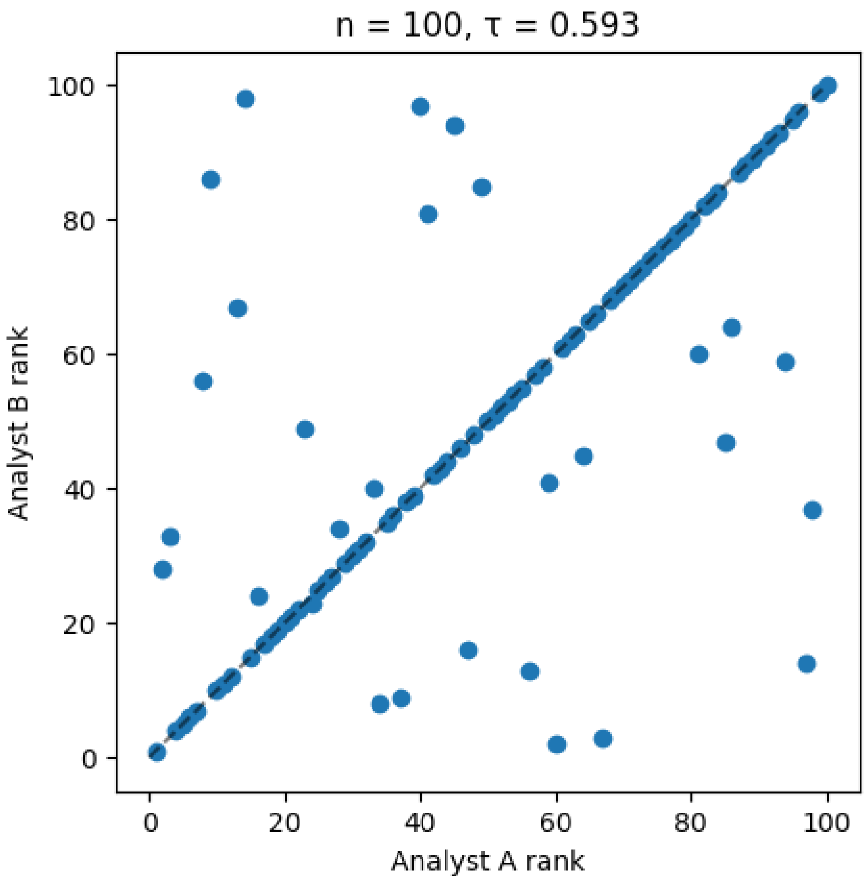
Here we got a p-value of approximately 0.056, which indicates that the association is marginally significant at the 0.05 level. This suggests that while there is a positive association between the rankings, we should be cautious in interpreting it as statistically significant. Why? Because our sample dataset is small (only 5 items), with a such a small number of pairs its more likely that random chance could produce similar levels of concordance. lets see what happens when we increase the size of the dataset.

2.5 Adding complexity

To further explore the behavior of Kendall's tau, we can increase the size of our dataset from 6 sites to 30. We will randomly generate base ranking, then create a slightly "noisy" version to simulate small differences in judgement or weight perturbations.

```python
np.random.seed(32)  # for reproducibility
n = 100 # change this and re-run as well to see the effect of sample size
swap_n = 30 # number of swaps to introduce, change this value and re-run to see different le

# Analyst A: perfect ranking 1 -> n
rank_A = np.arange(1, n + 1) # Analyst A ranks items from 1 to n

# Analyst B: same order bit with some random swaps (simulateing disagreement)
rank_B = rank_A.copy() # start with same ranking as Analyst A
swap_indices = np.random.choice(n, size=swap_n, replace=False) # choose 5 random indices to
np.random.shuffle(swap_indices) # shuffle the selected indices
rank_B[swap_indices] = rank_B[np.random.permutation(swap_indices)]  # perform the swaps

tau, p_value = kendalltau(rank_A, rank_B)
print(f"Kendall's tau between Analyst A and B: {tau:.3f}, p-value: {p_value:.3f}")

# Visualize the rankings in a scatter plot
plt.figure(figsize=(5,5))
plt.scatter(rank_A, rank_B)
plt.plot([0,n],[0,n],'k--',alpha=0.5)
plt.xlabel("Analyst A rank")
plt.ylabel("Analyst B rank")
plt.title(f"n = {n},   = {tau:.3f}")
plt.show()
```

```
Kendall's tau between Analyst A and B: 0.593, p-value: 0.000
```



Next lets add noise a different way.

First we will create a set of data, we will call *base_scores* we will just take n numbers spaced equally from 0 to 1

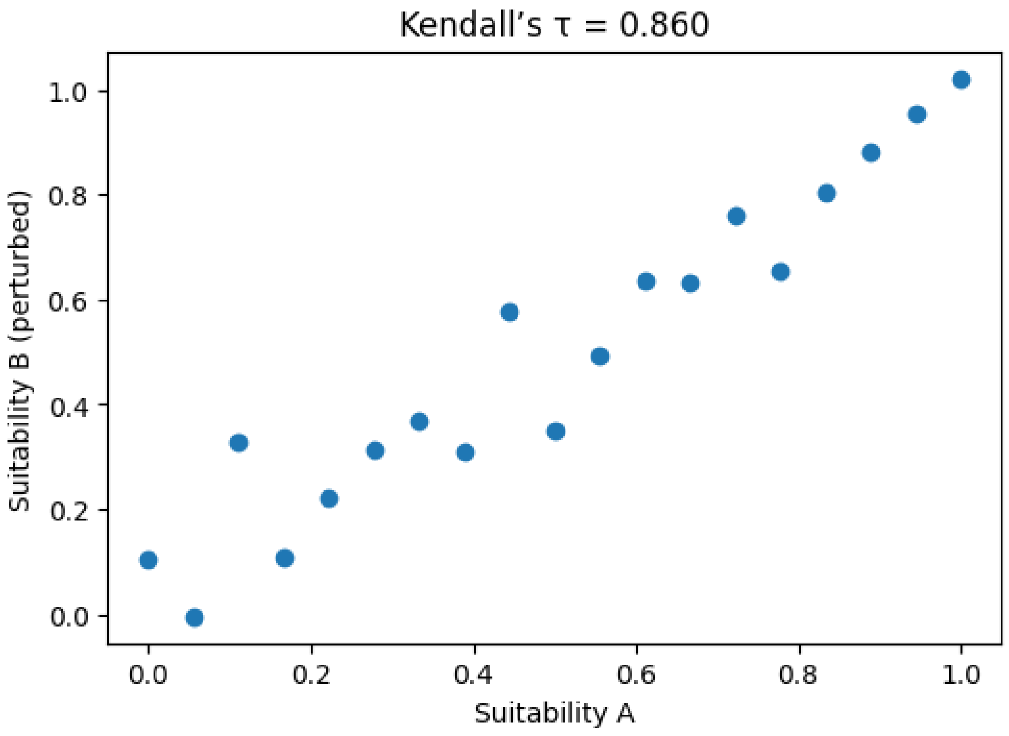
then we will create alternative scores which are the base_scores with some noise added, noise from a random normal distribution.

```
n = 19 # change this depending on the sample size you want
noise_factor = 0.08 # this is the standard devation of the distribution from which the noise
base_scores = np.linspace(0,1,n)
#print(f"base_scores {base_scores}")

noise = np.random.normal (0,noise_factor,n) # create noise by drawing random samples from a r

alt_scores = base_scores + noise # add the noise to the base scores to create alternative sco
#print(f"alt scores (base scores + noise) {alt_scores}")
```

```
rank_base = pd.Series(base_scores).rank() # rank the original scores, (remember we want ranks
rank_alt = pd.Series(alt_scores).rank() # the base scores have been changed a bit randomly sc

# now lets calculate kendall's tau

tau, p_value = kendalltau(rank_base, rank_alt)
print(f"  = {tau:.3f}; pvalue = {p_value:.5f}")
```

```
  = 0.860; pvalue = 0.00000
```

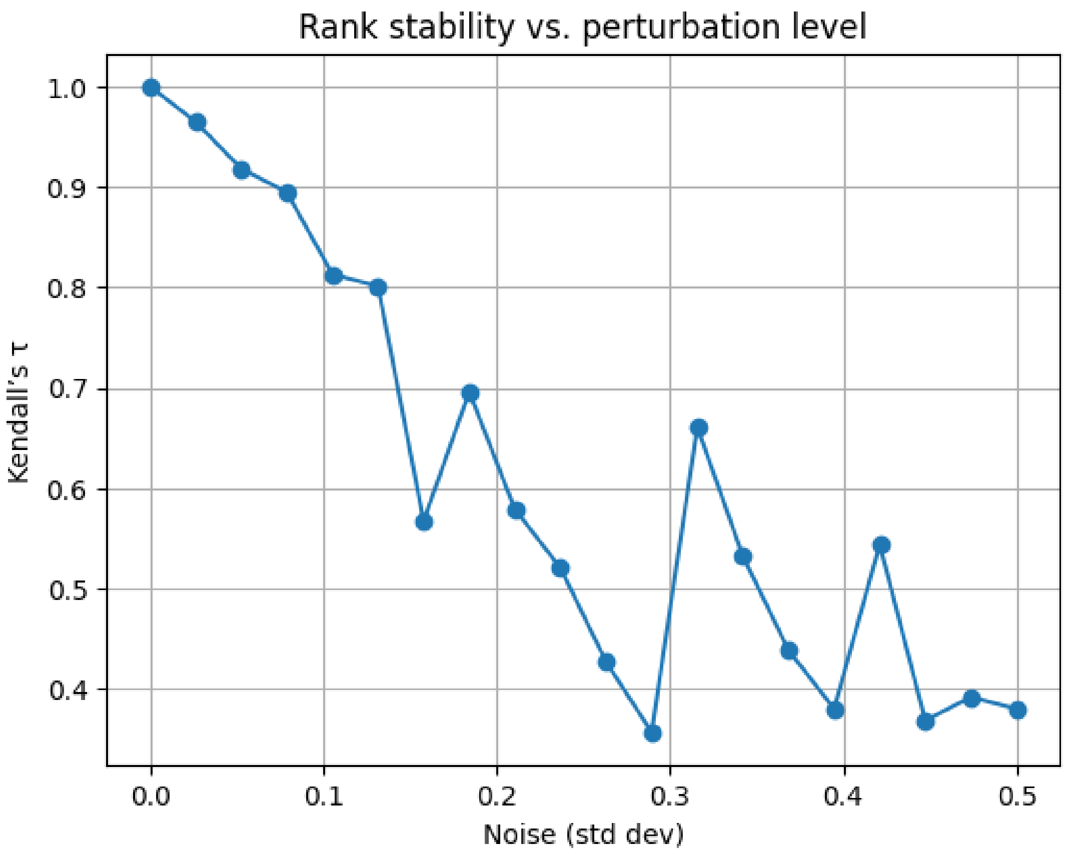
```
# plot the relationship between rank_base, and rank_alt (our two different rankings)

plt.figure(figsize=(6,4))
plt.scatter(base_scores, alt_scores)
plt.xlabel("Suitability A")
plt.ylabel("Suitability B (perturbed)")
plt.title(f"Kendall’s   = {tau:.3f}")
plt.show()
```



> Lets now look at how changing the level of noise in the data affects the kendalls tau

We will generate many random pertubations and compute τ each time. This mimics the way sensitivity analysis samples random weight combinations in a Weighted Linear Combination (WLC). basically we are looking at the kendall's tau through time, and automating the changing of the standard devation within the noise addition to see how adding different levels of noise affects kendall's tau.

```
noise_levels = np.linspace(0,0.5,20)
#print(noise_levels)

taus = []

for s in noise_levels:
    alt = base_scores + np.random.normal(0 , s , n) # recall n is defined above in previous c
    taus.append(kendalltau(pd.Series(base_scores).rank(), pd.Series(alt).rank())[0])


plt.plot(noise_levels, taus, marker='o')
plt.xlabel("Noise (std dev)")
plt.ylabel("Kendall's  ")
plt.title("Rank stability vs. perturbation level")
plt.grid(True)
plt.show()
```



you can see from the above figure that as noise increases kendall's tau a measure of similarity between rankings decreases, rank stability decreases.

2.6 real-world example - Countries Ranked by Life Expectancy and GDP

Ok, enough with fake data sets, lets step away from the hard sciences for a second and look at something more social. Lets look at how life expectancy compares to GDP, we would think life expectancy is higher in rich countries and lower in poor countries. So we can get the data on life expectancy, and we can get the data on GDP, then rank the countries in order of GDP and life expectancy, and compare how these two different ways to rank countries are concordant or discordant using Kendalls Tau

to see how we cleaned and created these datasets see this notebook: `Data\DataWranglingScripts\GDPvLif`

```
df = pd.read_csv("../Data/CLEAN/GDP_LifeExpectancy_2022_Clean.csv")
print(df.head())
print(df.columns)
```

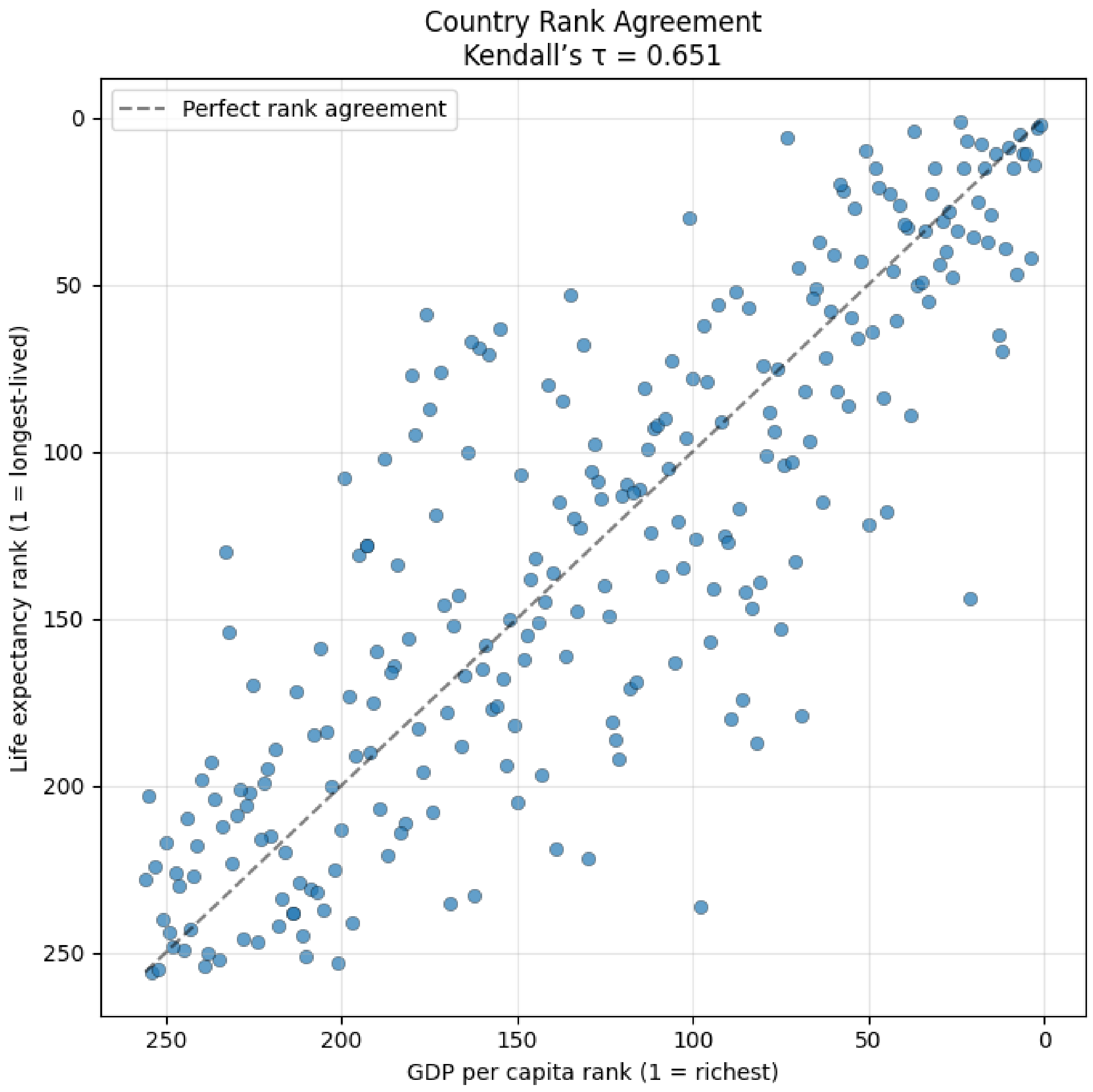
```
                 Country Name Country Code   GDP_PC_2022  LIFE_EX_YRS_2022  \
0                       Aruba          ABW  30559.533535         73.537000
1  Africa Eastern and Southern          AFE   1628.318944         61.765707
2                 Afghanistan          AFG    357.261153         63.941000
3   Africa Western and Central          AFW   1796.668633         56.906135
4                      Angola          AGO   2929.694455         61.748000

   GDP_PC_RANK_2022  LIFE_EX_YRS_RANK_2022
0              56.0                   86.0
1             216.0                  220.0
2             255.0                  203.0
3             210.0                  251.0
4             187.0                  221.0
Index(['Country Name', 'Country Code', 'GDP_PC_2022', 'LIFE_EX_YRS_2022',
       'GDP_PC_RANK_2022', 'LIFE_EX_YRS_RANK_2022'],
      dtype='object')
```

```
tau, pval = kendalltau(df["GDP_PC_RANK_2022"], df["LIFE_EX_YRS_RANK_2022"])

print(f"Kendall's tau ( ) = {tau:.3f}")
print(f"p-value = {pval:.8f}")
```

```
Kendall's tau ( ) = 0.651
p-value = 0.00000000
```

We see a strong positive rank correlation, which is what we would expect. Wealthier countries generally have longer life expectancy, though the relationship isnt perfect. Now lets visualize the data.

```
plt.figure(figsize=(7,7))
plt.scatter(
    df["GDP_PC_RANK_2022"],
    df["LIFE_EX_YRS_RANK_2022"],
    alpha=0.7,
    edgecolor="k",
    linewidth=0.3
)

# Add a diagonal "perfect agreement" line
plt.plot(
    [1, df["GDP_PC_RANK_2022"].max()],
    [1, df["LIFE_EX_YRS_RANK_2022"].max()],
    'k--', alpha=0.5, label="Perfect rank agreement"
)

plt.xlabel("GDP per capita rank (1 = richest)")
plt.ylabel("Life expectancy rank (1 = longest-lived)")
plt.title(f"Country Rank Agreement\nKendall’s   = {tau:.3f}")

# Flip axes so 'better' (rank 1) appears top-right
plt.gca().invert_xaxis()
plt.gca().invert_yaxis()
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



Next we will zoom in on the top 40 countries in terms of GDP and see if the kendalls tau is better or worse when we exclude all but the 40 richest.

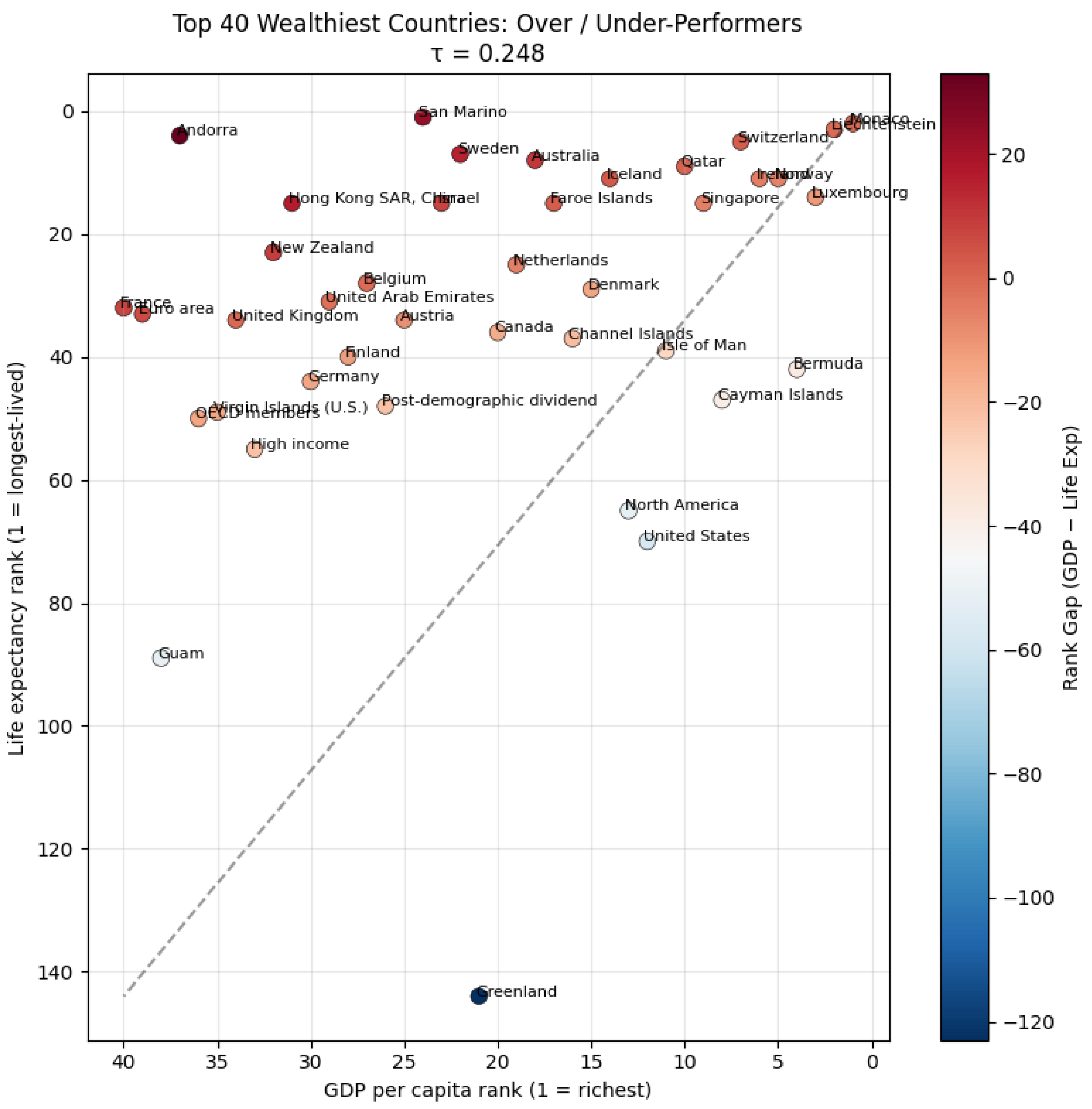
```
# Sort by GDP rank (1 = richest)
df_top40 = (
    df
    .sort_values("GDP_PC_RANK_2022", ascending=True)
    .head(40)
    .copy()
)

print(df_top40[["Country Name", "GDP_PC_RANK_2022", "LIFE_EX_YRS_RANK_2022"]].head())
print(f"Number of countries in subset: {len(df_top40)}")
```

```
      Country Name  GDP_PC_RANK_2022  LIFE_EX_YRS_RANK_2022
144         Monaco               1.0                    2.0
133  Liechtenstein               2.0                    3.0
140     Luxembourg               3.0                   14.0
27         Bermuda               4.0                   42.0
172         Norway               5.0                   11.0
Number of countries in subset: 40
```

```
tau_top, pval_top = kendalltau(
    df_top40["GDP_PC_RANK_2022"],
    df_top40["LIFE_EX_YRS_RANK_2022"]
)

print(f"Kendall’s  (top 40 richest) = {tau_top:.3f}")
print(f"p-value = {pval_top:.8f}")
```

```
Kendall’s  (top 40 richest) = 0.248
p-value = 0.02515424
```

```
df_top40["RankGap"] = (
    df_top40["GDP_PC_RANK_2022"] - df_top40["LIFE_EX_YRS_RANK_2022"]
)

plt.figure(figsize=(8,8))
scatter = plt.scatter(
    df_top40["GDP_PC_RANK_2022"],
    df_top40["LIFE_EX_YRS_RANK_2022"],
    c=df_top40["RankGap"],
    cmap="RdBu_r",
    s=70,
    edgecolor="k",
    linewidth=0.4
)
plt.colorbar(scatter, label="Rank Gap (GDP - Life Exp)")
plt.plot(
    [1, df_top40["GDP_PC_RANK_2022"].max()],
    [1, df_top40["LIFE_EX_YRS_RANK_2022"].max()],
    'k--', alpha=0.4
)

for _, row in df_top40.iterrows():
    plt.text(
        row["GDP_PC_RANK_2022"] + 0.2,
        row["LIFE_EX_YRS_RANK_2022"],
        row["Country Name"],
        fontsize=8
    )

plt.xlabel("GDP per capita rank (1 = richest)")
plt.ylabel("Life expectancy rank (1 = longest-lived)")
plt.title(f"Top 40 Wealthiest Countries: Over / Under-Performers\n  = {tau_top:.3f}")

plt.gca().invert_xaxis()
plt.gca().invert_yaxis()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



The analysis of country rank in terms of GDP per capita vs life expectancy using Kendall’s tau tells us that overall life expectency is correlated with GDP per capita in terms of how contries compare to eachother (rank), however the correlation is much worse in the rich countries, why might that be?

2.7 CONCEPTUAL QUESTIONS

1. Is Kendall's Tau a good way to measure the correlation of these two variables?

   answer: Its more approriate to look at how the Life Expectency values compare to the GDP per capita directly using Pearson's R, which compares one number to another. Kendall's Tau is for comparing the ranks, so the colored rank figure shows us that countries below the line, are under performing in terms of life expectancy vs GDP relative to their neighbors. It is also telling us the the GDP life expectency relationship breaks down at higher levels of GDP or is less meaningful.

2. Why might this relationship breakdown when subsetting the data to only the richest countries?

2.8 Spatial Rank Correlation

Using Kendalls Tau for suitability mapping sensitivity Analysis

In a Weighted Linear Combination (WLC) or other GIS-MCDA, you often generate suitability rasters under different weighting schemes, e.g.:

- Scenario A: baseline weights (e.g., 40% slope, 30% soil, 30% rainfall)
- Scenario B: modified weights (e.g., 30% slope, 40% soil, 30% rainfall)

Each raster cell gets a suitability score. You can rank cells (1 = most suitable) for each scenario, then compute Kendall's τ between the two rankings.

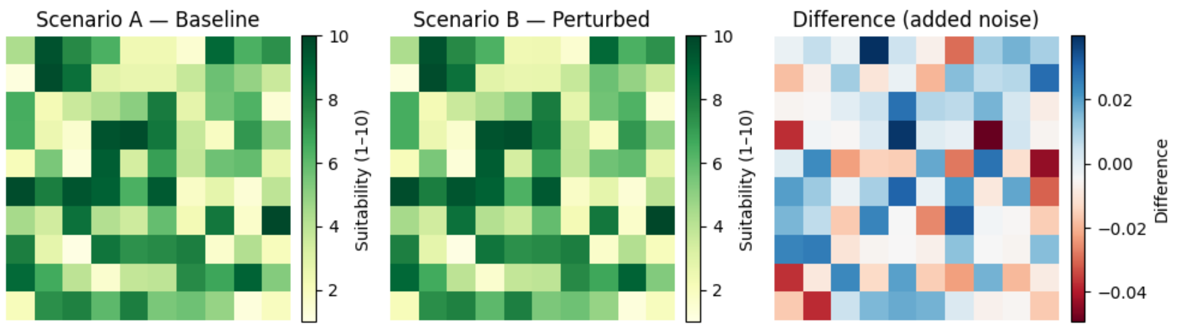
2.9 simulated MCDA suitability

We will build two small 10x10 rasteres (100 cells):

suitability_A –> Baseline Scenario suitability_B –> slightly perturbed version (change one weight layer)

1. Then flatted both to 1D arrays (each cell = one observation)
2. compute τ for the full map
3. visualize where ranks changed the most.

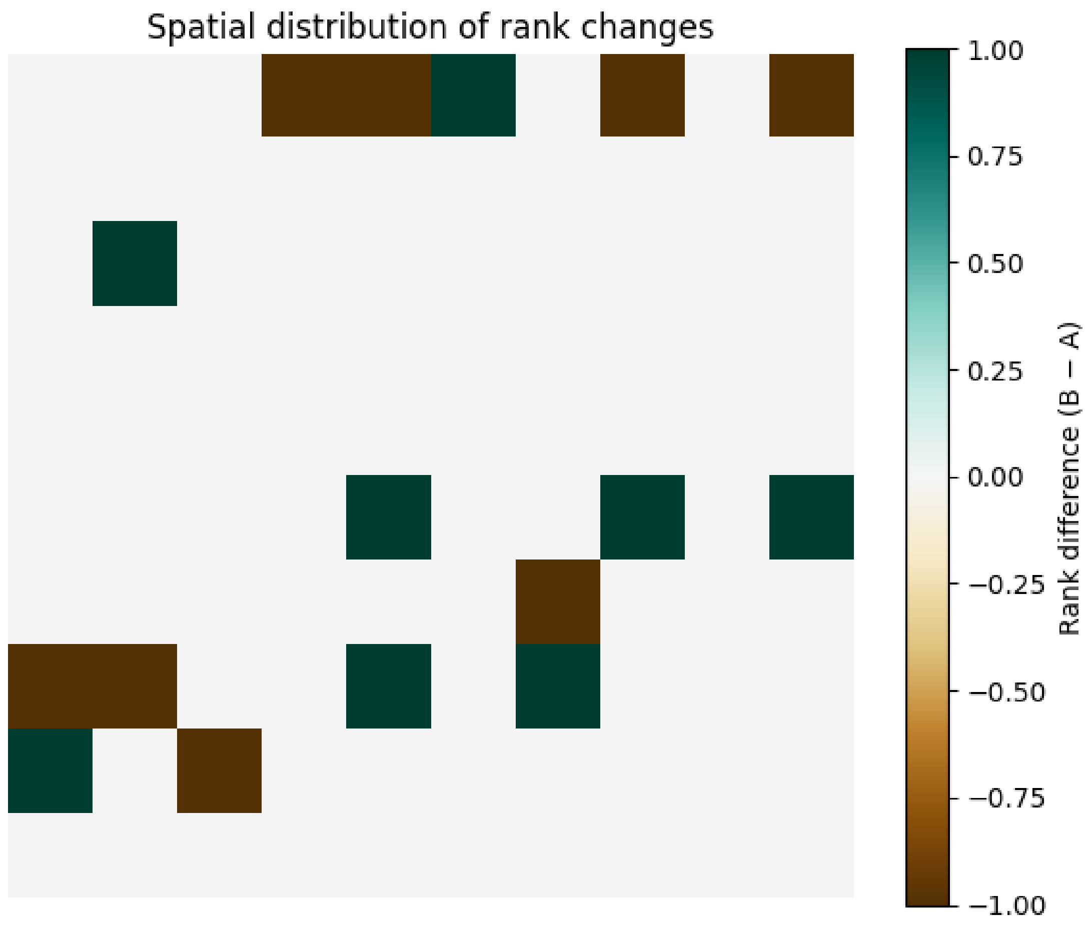
```
np.random.seed(42)

# --- Step 1: create two synthetic suitability grids (values 1-10) ---
grid_size = 10
suitability_A = np.random.rand(grid_size, grid_size) * 9 + 1  # values in [1,10]
suitability_B = suitability_A + np.random.normal(0, 0.02, (grid_size, grid_size))  # add mild
suitability_B = np.clip(suitability_B, 1, 10)  # keep within same range

diff = suitability_A - suitability_B

# --- Step 2: plot them side-by-side ---
fig, axes = plt.subplots(1, 3, figsize=(10, 4))

im1 = axes[0].imshow(suitability_A, cmap="YlGn", vmin=1, vmax=10)
axes[0].set_title("Scenario A - Baseline")
axes[0].axis("off")
plt.colorbar(im1, ax=axes[0], fraction=0.046, pad=0.04, label="Suitability (1-10)")

im2 = axes[1].imshow(suitability_B, cmap="YlGn", vmin=1, vmax=10)
axes[1].set_title("Scenario B - Perturbed")
axes[1].axis("off")
plt.colorbar(im2, ax=axes[1], fraction=0.046, pad=0.04, label="Suitability (1-10)")

im3 = axes[2].imshow(diff, cmap = 'RdBu', vmin = diff.min(), vmax = diff.max())
axes[2].set_title("Difference (added noise)")
axes[2].axis('off')
plt.colorbar(im3, ax=axes[2], fraction=0.046, pad = 0.04, label="Difference")

plt.tight_layout()
plt.show()
```



```
# --- Step 3: compute Kendall's tau on the flattened ranks ---
A_flat = suitability_A.flatten()
B_flat = suitability_B.flatten()

rank_A = pd.Series(A_flat).rank()
rank_B = pd.Series(B_flat).rank()

tau, pval = kendalltau(rank_A, rank_B)
print(f"Overall spatial Kendall's   = {tau:.3f} -- Pvalue: {pval:.4f}")
```

```
Overall spatial Kendall's   = 0.997 -- Pvalue: 0.0000
```

```
rank_diff = (rank_B - rank_A).values.reshape(grid_size, grid_size)

plt.figure(figsize=(6,5))
plt.imshow(rank_diff, cmap="BrBG", vmin=rank_diff.min(), vmax=rank_diff.max())
plt.colorbar(label="Rank difference (B - A)")
plt.title("Spatial distribution of rank changes")
plt.axis("off")
plt.tight_layout()
plt.show()
```


Spatial distribution of rank changes
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Rank difference (B − A)

3 Morris Sensitivity Analysis (Elementary Effects Method)

In global sensitivity analysis, the **Morris Method** (also called the *Method of Elementary Effects*) is a screening technique used to identify which input variables in a model have the greatest influence on the output. It was introduced by Max D. Morris in 1991 as a computationally efficient way to explore sensitivity in models with many inputs, without requiring an enormous number of model runs.

The basic idea is:

- Change one input at a time by a small step,
- See how much the output changes,
- Repeat this from different starting points across the input space,
- Summarize how consistently (or inconsistently) each input causes change.

That small one-at-a-time change in an input and the resulting change in the model output is called an **elementary effect**.

3.1 Why was Morris developed?

Before Morris (1991), sensitivity analysis often lived in two extremes:

- **Local / derivative-based sensitivity**:

  This asks "If I make a tiny change to one parameter around the current baseline, how much does the output change?"
  This is basically a partial derivative.

  Problem: it only tells you about behavior *near one point* in parameter space, and it assumes smooth / linear behavior.

- **Full global variance-based methods (like Sobol indices)**:

  These methods try to quantify how much of the total output variance is explained by each input and by their interactions.

  They're extremely informative — but also computationally expensive, because they require a lot of model evaluations.

Max Morris was looking for something in-between:

- A method that is **global** (explores the whole parameter space, not just one point),
- But still **cheap enough** to run early, even for high-dimensional problems (many inputs),
- And able to flag inputs that are likely important, nonlinear, or interacting.

So the Morris method is often described as a **screening method**: it's a first pass that tells you which inputs matter and how they matter, so you know where to focus more detailed analysis later.

3.2 What problems does the Morris method solve?

Imagine you have a model:

$$y = f(x_1, x_2, x_3, \dots, x_k)$$

where each x_i is an input (a criterion weight, a threshold, a soil parameter, etc.), and y is some decision score (e.g. total suitability, predicted recharge, contaminant load, habitat score).

You want to know:

1. Which inputs have basically **no effect** on y? (Those might be safely fixed or ignored.)
2. Which inputs have a **large overall effect** on y? (These are important drivers.)
3. Which inputs behave **nonlinearly** or **interact** with other inputs?
   (For example, "slope only matters once soil permeability is high," or "forest cover and precipitation together change infiltration potential in a way you don't get by looking at either alone.")

The Morris method gives you exactly that information with two summary statistics per input.

3.3 Core concept: the Elementary Effect

For each input x_i, we define an *elementary effect* as:

$$EE_i = \frac{f(x_1, \dots, x_i + \Delta, \dots, x_k) - f(x_1, \dots, x_i, \dots, x_k)}{\Delta}$$

Where:

- Δ is a small step in the value of x_i,
- All other inputs are held constant for that step,
- The numerator is "how much the output changed when we nudged just x_i."

Interpretation:

- EE_i is basically: "If I change only x_i a little, how much does the model output respond?"

But — and this is the key difference from local sensitivity — we don't do this just once from one baseline. We repeat this from *multiple random locations* in the input space. Each repetition gives us another possible elementary effect for that same input.

So for each input x_i, we don't just get one number. We get a distribution of elementary effects across the space of plausible inputs.

3.4 Morris summary metrics

After computing many elementary effects for each input, we summarize them. The two most common summaries are:

1. μ^* (mu star):

   The mean of the *absolute value* of the elementary effects for that input.

   - High μ^* means: changing this input tends to cause a big change in the output overall.

   - This is interpreted as "overall importance" or "influence strength."

   We use the absolute value so positive and negative effects don't cancel each other out.

2. σ (sigma):

   The standard deviation of the elementary effects for that input.

- High σ means: the effect of this input is not consistent — sometimes it has a big effect, sometimes small, sometimes positive, sometimes negative.

- That usually indicates **nonlinearity** or **interactions with other inputs**.

Intuition: if an input only mattered under certain combinations of other inputs, you'd see a wide spread in its elementary effects → high .

This gives you a beautiful diagnostic plot: * on the x-axis (importance) vs on the y-axis (interaction / nonlinearity).

- Inputs with **low *** and **low** → mostly irrelevant.
- Inputs with **high *** and **low** → consistently important, mostly linear effect on the output.
- Inputs with **high *** and **high** → important but tricky: nonlinear or involved in interactions.

That's the classic "Morris scatter plot."

3.5 How it works (conceptually)

1. Define ranges (or distributions) for each input x_i.
   Example: slope weight in WLC could vary from 0.1 to 0.4, precipitation weight from 0.2 to 0.6, etc.

2. Sample a sequence of points in that input space (called "trajectories" or "paths").
   Each path walks through the space one input at a time, changing one variable by Δ while keeping the others fixed, then moving on to the next variable, etc.

3. For each step along that path, compute the elementary effect EE_i.

4. Aggregate all the elementary effects for each input across all paths → get μ^* and σ.

5. Rank or plot inputs based on μ^* and to decide which inputs matter.

This is global because you're sampling across the full allowable range of inputs — not just perturbing around a single baseline point.

3.6 Why this is used

The Morris method is widely used in:

- Environmental modeling and hydrology (e.g., identifying which hydrogeologic parameters most influence recharge estimates or contaminant transport),
- Ecological and habitat suitability modeling,
- Groundwater recharge / infiltration models,
- Flood and erosion models,
- Multi-criteria decision analysis (MCDA), including GIS-based suitability mapping, to see which criteria weights dominate the final suitability score, and where there are strong interactions.

In practice:

- You run Morris first to screen out unimportant variables (so you don't waste computation on them),
- Then you apply heavier methods (like Sobol variance decomposition) on the variables that survived screening.

So Morris is both:

1. A science tool (which parameters actually matter?),
2. A workflow tool (where should I spend my expensive computation time?).

3.7 Summary

- The Morris method is a global, one-factor-at-a-time sensitivity screening method introduced by Max D. Morris in 1991.
- It's built around **elementary effects**: the change in model output when you nudge one input while holding others constant.
- By repeating that across many starting points, you get a *distribution* of effects for each input.
- You then summarize each input with:
  - μ^* (mean absolute elementary effect): how influential this input is overall,
  - σ (stdev of elementary effects): how nonlinear or interaction-heavy its influence is.
- This is incredibly helpful in decision-support models (like WLC suitability mapping) because it tells you:
  - which criteria weights dominate suitability,

  - which ones only matter in combination,
  - and which ones are basically irrelevant.

Next, we'll:

1. Build a tiny synthetic model in Python so you can *see* elementary effects for a toy function,
2. Compute μ^* and σ for each input manually,
3. Reproduce what SALib's `morris` routines do,
4. Visualize μ^* vs σ,
5. Then connect that to a spatial WLC / MCDA setting.

   Now let's start generating elementary effects for a simple model in Python.

```python
## Import libraries and set random seed for repeatability

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


np.random.seed(12)
```

1. Lets define a sample model

We want the model to have the following attributes:

- depends on multiple inputs
- is at least a little nonlinear
- has some interactions

$$f(x_1, x_2, x_3) = 2x_1 + 0.5x_2^2 + 3x_1x_3$$

features of this function:

- $2x_1$ linear effect of x_1
- $0.5x_2^2$ nonlinear effect of x_2
- $3x_1x_3$ interaction between x_1 and x_3

```python
# function to test for elementary effects

def model(x1,x2,x3):
    return 2*x1 + 0.5*(x2**2) + 3*x1*x3
```

2. Define and measure elementary effects

For a given input x_i the elementary effect is:

$$EE_i = \frac{f(..., x_i + \Delta, ...) - f(..., x_i, ...))}{\Delta}$$

We will create a function that does the following:

- pick a random starting point (x_i) from (x_1, x_2, x_3)
- nudge just one variable by Δ
- compute the change in output per unit step

  notes: 1. we clip the perturbed value so we stay in a valid range [0,1]. 2. we gaurd the denominator in case delta pushes us out and gets clipped to the same value.

```python
def elementary_effect(model_func, x, i, delta):
    """
    compute the elementary effect of variable i at point x.

    model_func: callable f(x1,x2,x3)
    x: np.array shape (3,) representing [x1,x2,x3]
    i: which index to perturb (0,1,2)
    delta: step sie to add to x[i]

    returns: EE_i float
    """
    x_base = x.copy()
    x_perturbed = x.copy()
    x_perturbed[i] = x_perturbed[i] + delta
    x_perturbed[i] = np.clip(x_perturbed[i], 0, 1)

    y0 = model_func(*x_base)
    y1 = model_func(*x_perturbed)

    return (y1 - y0) / (x_perturbed[i] - x_base[i] + 1e-12)
```

3. Sample multiple points in input space and gather EEs

We will:

- draw random points in [0,1]^3,
- for each point, compute EEs for x1, x2, x3,
- repeat for, say, 50 random points.

```python
def sample_elementary_effects(model_func, n_samples=50, delta=0.1):
    """
    For n_samples random base points in [0,1]^3,
    compute elementary effects for each of the 3 inputs.

    Returns: DataFrame with columns:
      ['x1_EE', 'x2_EE', 'x3_EE']
    """
    records = []

    for _ in range(n_samples):
        # random point in [0,1]^3
        x = np.random.rand(3)

        ee_x1 = elementary_effect(model_func, x, i=0, delta=delta)
        ee_x2 = elementary_effect(model_func, x, i=1, delta=delta)
        ee_x3 = elementary_effect(model_func, x, i=2, delta=delta)

        records.append({
            "x1_EE": ee_x1,
            "x2_EE": ee_x2,
            "x3_EE": ee_x3,
        })

    return pd.DataFrame(records)

ee_df = sample_elementary_effects(model, n_samples=50, delta=0.1)
ee_df.head()
```

	x1_EE	x2_EE	x3_EE
0	2.789945	0.790050	0.462489
1	4.756241	0.064575	1.601218
2	4.870848	0.083421	2.702145
3	3.818250	0.333828	0.411628
4	2.006778	0.902736	2.832675

4. Next we compute μ^* and σ

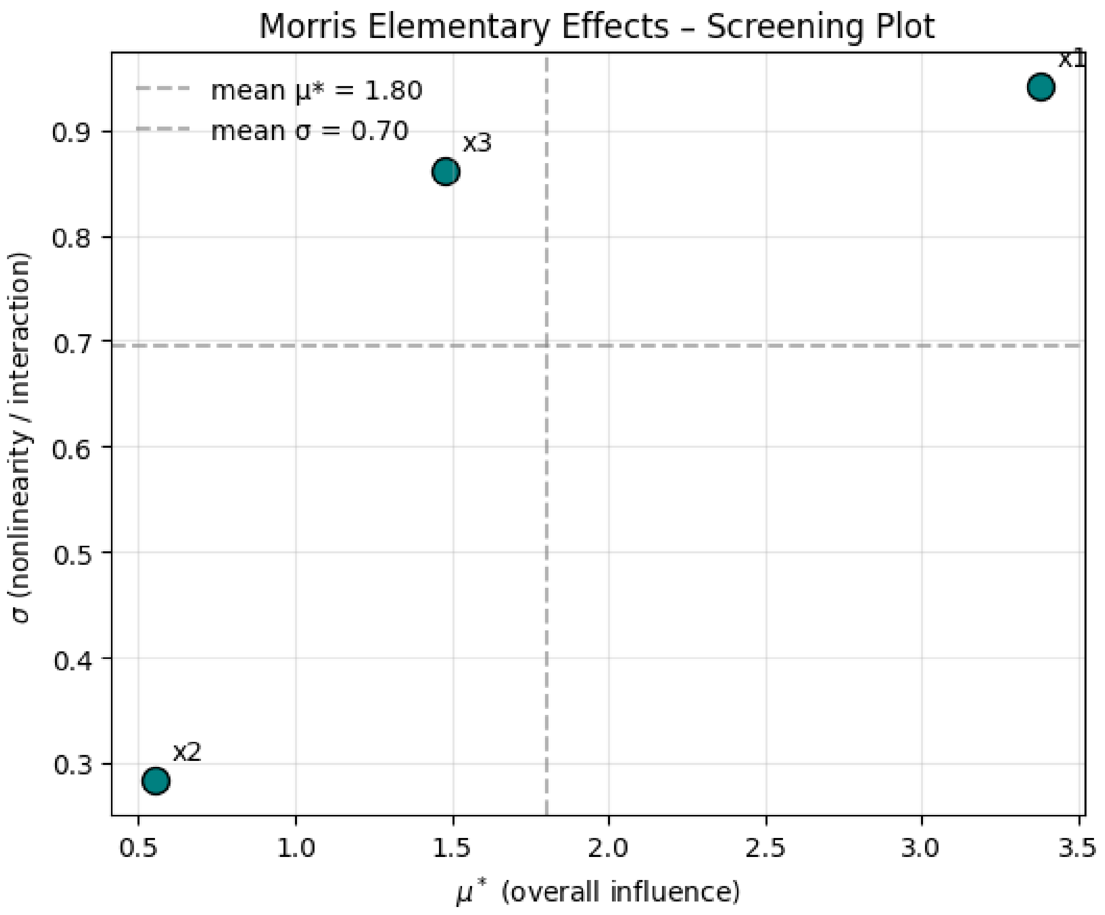
```
summary = pd.DataFrame({
    "mu_star": [
        ee_df["x1_EE"].abs().mean(),
        ee_df["x2_EE"].abs().mean(),
        ee_df["x3_EE"].abs().mean()
    ],
    "sigma": [
        ee_df["x1_EE"].std(ddof=1),
        ee_df["x2_EE"].std(ddof=1),
        ee_df["x3_EE"].std(ddof=1)
    ]
}, index=["x1", "x2", "x3"])

summary
```

	mu_star	sigma
x1	3.378779	0.940584
x2	0.554154	0.283397
x3	1.480260	0.861859

5. plot μ^* vs σ

```
import matplotlib.pyplot as plt

plt.figure(figsize=(6,5))

# --- base scatter ---
plt.scatter(summary["mu_star"], summary["sigma"],
            s=100, color="teal", edgecolor="k")

# --- reference lines (mean   and mean   ) ---
mu_mean = summary["mu_star"].mean()
sigma_mean = summary["sigma"].mean()

plt.axvline(mu_mean, color="gray", linestyle="--", alpha=0.6,
            label=f"mean  * = {mu_mean:.2f}")
plt.axhline(sigma_mean, color="gray", linestyle="--", alpha=0.6,
            label=f"mean   = {sigma_mean:.2f}")
```

```
# --- labels for each variable ---
for var_name in summary.index:
    plt.text(summary.loc[var_name, "mu_star"] + 0.05,
             summary.loc[var_name, "sigma"] + 0.02,
             var_name, fontsize=10)

# --- axes & title ---
plt.xlabel(r"$\mu^*$ (overall influence)")
plt.ylabel(r"$\sigma$ (nonlinearity / interaction)")
plt.title("Morris Elementary Effects - Screening Plot")
plt.legend(loc="upper left", frameon=False)
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



3.7.1 interpretation:

The graph shows the following:

- x_2 has low influence and low interaction
- x_1 has high interaction and high influence
- x_3 has high interaction but low influence

This makes sense looking at our model:

$$f(x_1, x_2, x_3) = 2x_1 + 0.5x_2^2 + 3x_1x_3$$

features of this function:

- $2x_1$ linear effect of x_1
- $0.5x_2^2$ small but nonlinear effect of x_2
- $3x_1x_3$ interaction between x_1 and x_3

4 Applying Morris Elementary Effects to a Real Dataset: The Palmer Penguins

To make the concept of **Morris Elementary Effects** more concrete, we'll apply it to a simple, well-known dataset — the **Palmer Penguins** dataset.
This dataset provides morphological measurements for three penguin species (*Adelie*, *Chinstrap*, and *Gentoo*) collected from islands in the Palmer Archipelago, Antarctica.

It is often used as a modern alternative to the classic *Iris* dataset because: - It contains a small number of continuous, interpretable features. - The relationships between variables are biologically intuitive (e.g., larger flipper length → heavier penguin). - It contains both linear and nonlinear interactions, which make it a good demonstration dataset for **sensitivity analysis** methods like Morris.

4.1 Goal

We'll use the **Morris method** to analyze which morphological features most influence a penguin's **body mass**.

Specifically: 1. **Inputs (factors)**:
- Bill length (mm)
- Bill depth (mm)
- Flipper length (mm)
2. **Output (model response)**:
- Body mass (g)

We'll fit a simple regression model (f(x_1, x_2, x_3)) that predicts body mass from these inputs, then treat this model as our "black box."
Afterward, we'll apply the **Morris Elementary Effects method** to quantify: - (^*) → the overall (average) influence of each input, and
- ( ) → the variability or nonlinearity of that influence across the input space.

4.2 Why this example works

- **Interpretability**: It's easy to reason about which traits should matter (flipper length and bill length should correlate with mass).
- **Dimensional simplicity**: With only 3 numeric inputs, the results are easy to visualize in 2D (– plot).
- **Interaction potential**: The relationships are not purely linear — for instance, the effect of flipper length might depend on species or bill size.

By working with this dataset, we can see how the Morris method distinguishes between *strong, consistent* influences (high , low) and *context-dependent, interacting* ones (high , high), even in an everyday biological system.

```
import seaborn as sns
import pandas as pd

# Load dataset
penguins = sns.load_dataset("penguins")
penguins.head()
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Female

```
# keep only columns we need and drop NAs

cols = [
    "bill_length_mm",
    "bill_depth_mm",
    "flipper_length_mm",
    "body_mass_g"
]
```

```
df = penguins[cols].dropna().copy()
df.head(), df.shape
```

```
(   bill_length_mm  bill_depth_mm  flipper_length_mm  body_mass_g
 0            39.1           18.7              181.0       3750.0
 1            39.5           17.4              186.0       3800.0
 2            40.3           18.0              195.0       3250.0
 4            36.7           19.3              193.0       3450.0
 5            39.3           20.6              190.0       3650.0,
 (342, 4))
```

4.2.1 fit simple regression model

We'll use scikit-learns linear regression as our block-box model.

This gives us $\hat{y} = f(x_1, x_2, x_3)$ that we can later probe with Morris.

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

X = df[["bill_length_mm", "bill_depth_mm", "flipper_length_mm"]]
y = df["body_mass_g"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=0
)

model = LinearRegression()
model.fit(X_train, y_train)

print("R^2 on train:", model.score(X_train, y_train))
print("R^2 on test :", model.score(X_test, y_test))
model.coef_, model.intercept_
```

```
R^2 on train: 0.7694159737586672
R^2 on test : 0.7210757528501677
```

```
(array([ 7.12002238,  7.65512529, 48.21508216]),
 np.float64(-5931.683059218843))
```

4.2.2 wrap fitted model as function

Morris neds a function that takes inputs and returns a scalar output.

We'll define a function that:

- accepts three inputs(bill_length, bill_depth, flipper_length),
- packs them into the shape scikit-learn expects
- returns the predicted body mass in grams

```
def penguin_mass_model(bill_length_mm, bill_depth_mm, flipper_length_mm):
    """
    Predict penguin body mass (g) from morphology using our trained linear model.
    Inputs are scalars.
    Returns a scalar prediction (grams).
    """
    X_input = pd.DataFrame([{
        "bill_length_mm": bill_length_mm,
        "bill_depth_mm": bill_depth_mm,
        "flipper_length_mm": flipper_length_mm
    }])
    y_pred = model.predict(X_input)
    return float(y_pred[0])
```

4.2.3 define realistic ranges for each input

to run morris, we need to tell it what ranges each input can take.

We'll base that on the observed min/max in the data

```
feature_ranges = {
    "bill_length_mm": (
        df["bill_length_mm"].min(),
        df["bill_length_mm"].max()
    ),
    "bill_depth_mm": (
        df["bill_depth_mm"].min(),
        df["bill_depth_mm"].max()
    ),
    "flipper_length_mm": (
        df["flipper_length_mm"].min(),
        df["flipper_length_mm"].max()
    )
}

feature_ranges
```

```
{'bill_length_mm': (np.float64(32.1), np.float64(59.6)),
 'bill_depth_mm': (np.float64(13.1), np.float64(21.5)),
 'flipper_length_mm': (np.float64(172.0), np.float64(231.0))}
```

```
# --- 1. Define the sampling and EE computation function ---
def sample_elementary_effects_real_model(model_func, feature_ranges, n_samples=100, delta=0.
    """
    Compute elementary effects for each continuous input variable in the model.

    Parameters
    ----------
    model_func : callable
        A function that takes three inputs (bill_length, bill_depth, flipper_length)
        and returns a scalar prediction (body mass in g).
    feature_ranges : dict
        Dictionary mapping feature names to (min, max) tuples.
    n_samples : int
        Number of random points to sample.
    delta : float
        Fractional perturbation of each variable (e.g., 0.05 = 5% of its range).
    """
    features = list(feature_ranges.keys())
    records = []

    for _ in range(n_samples):
        # Randomly sample a base point inside the observed feature space
        base_point = {}
        for feat, (low, high) in feature_ranges.items():
            base_point[feat] = np.random.uniform(low=low + 0.05*(high-low), high=high - 0.05

        # Baseline prediction
        y0 = model_func(base_point[features[0]], base_point[features[1]], base_point[feature

        ee_point = {}
        for feat in features:
            low, high = feature_ranges[feat]
            step = delta * (high - low)
            perturbed_point = base_point.copy()
            perturbed_point[feat] = np.clip(base_point[feat] + step, low, high)

            y1 = model_func(
                perturbed_point[features[0]],
                perturbed_point[features[1]],
                perturbed_point[features[2]]
            )

            ee_point[f"{feat}_EE"] = (y1 - y0) / step

        records.append(ee_point)

    return pd.DataFrame(records)

# --- 2. Run the sampling ---
ee_df = sample_elementary_effects_real_model(
    penguin_mass_model,
    feature_ranges,
    n_samples=100,
    delta=0.05
)

ee_df.head()
```

	bill_length_mm_EE	bill_depth_mm_EE	flipper_length_mm_EE
0	7.120022	7.655125	48.215082
1	7.120022	7.655125	48.215082
2	7.120022	7.655125	48.215082
3	7.120022	7.655125	48.215082
4	7.120022	7.655125	48.215082

```
summary = pd.DataFrame({
    "mu_star": [ee_df[c].abs().mean() for c in ee_df.columns],
    "sigma": [ee_df[c].std(ddof=1) for c in ee_df.columns]
}, index=[c.replace("_EE", "") for c in ee_df.columns])

summary
```

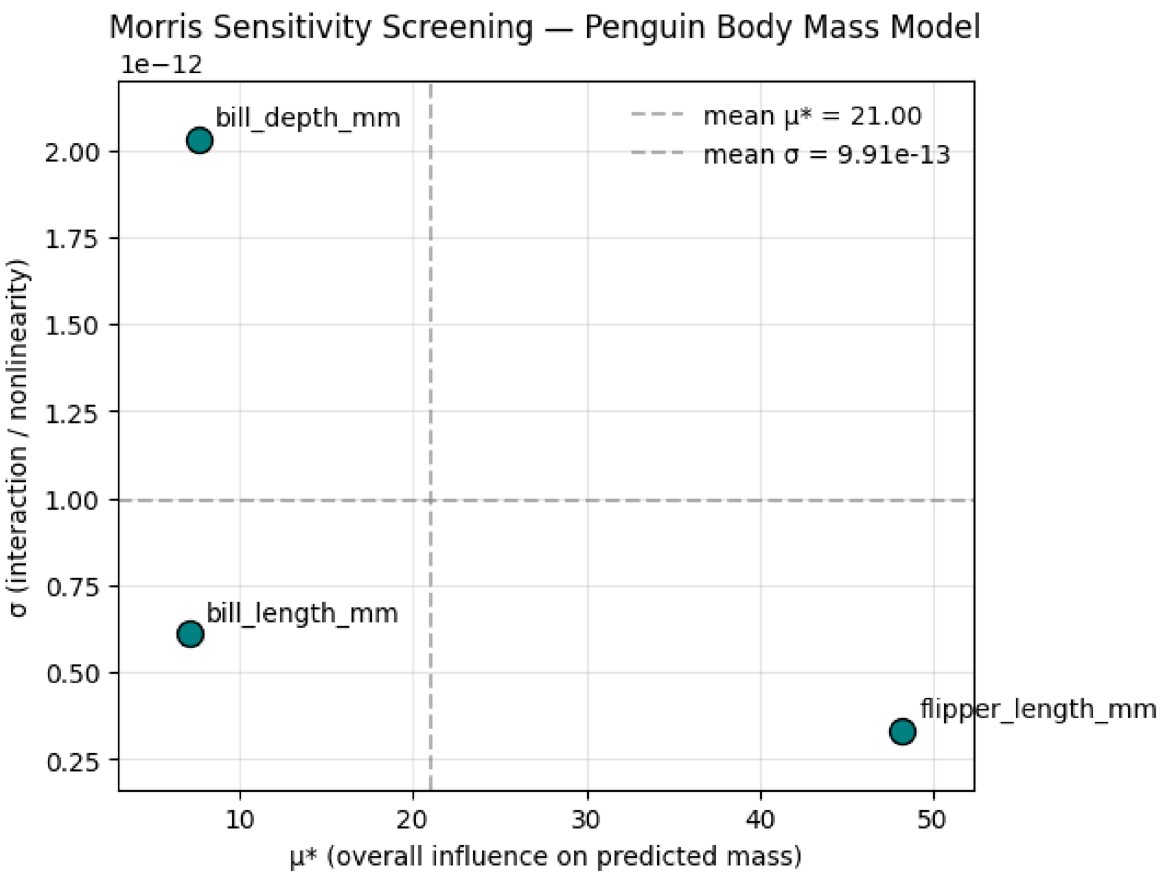
	mu_star	sigma
bill_length_mm	7.120022	6.091025e-13
bill_depth_mm	7.655125	2.030373e-12
flipper_length_mm	48.215082	3.332289e-13

```python
print(summary)
print()

print("mu_star stats:")
print(" min:", summary["mu_star"].min())
print(" max:", summary["mu_star"].max())
print(" mean:", summary["mu_star"].mean())

print("\nsigma stats:")
print(" min:", summary["sigma"].min())
print(" max:", summary["sigma"].max())
print(" mean:", summary["sigma"].mean())

print("\nAny NaN in summary?")
print(summary.isna().any())
```

```
                     mu_star         sigma
bill_length_mm      7.120022  6.091025e-13
bill_depth_mm       7.655125  2.030373e-12
flipper_length_mm  48.215082  3.332289e-13

mu_star stats:
 min: 7.120022379018935
 max: 48.21508215626039
 mean: 20.996743274284498

sigma stats:
 min: 3.3322894109700684e-13
 max: 2.030373237990764e-12
 mean: 9.909015515269688e-13

Any NaN in summary?
mu_star    False
sigma      False
dtype: bool
```

```
mu_vals = summary["mu_star"].values.astype(float)
sigma_vals = summary["sigma"].values.astype(float)
names = summary.index.tolist()

mu_mean = float(np.mean(mu_vals))
sigma_mean = float(np.mean(sigma_vals))

# define plotting limits with a small % padding around actual data
mu_min = mu_vals.min()
mu_max = mu_vals.max()
sigma_min = sigma_vals.min()
sigma_max = sigma_vals.max()

mu_pad = 0.1 * (mu_max - mu_min if mu_max > mu_min else 1.0)
sigma_pad = 0.1 * (sigma_max - sigma_min if sigma_max > sigma_min else 1.0)

x_lo = mu_min - mu_pad
x_hi = mu_max + mu_pad
y_lo = sigma_min - sigma_pad
y_hi = sigma_max + sigma_pad

fig, ax = plt.subplots(figsize=(6,5))

# scatter points
ax.scatter(mu_vals, sigma_vals,
           s=100, color="teal", edgecolor="k", zorder=3)

# adaptive label offset: 2% of axis span instead of hardcoded 0.02
x_offset = 0.02 * (x_hi - x_lo)
y_offset = 0.02 * (y_hi - y_lo)

for x, y, label in zip(mu_vals, sigma_vals, names):
    ax.text(x + x_offset, y + y_offset, label,
            fontsize=10, zorder=4)

# reference lines
ax.axvline(mu_mean, color="gray", linestyle="--", alpha=0.6,
           label=f"mean  * = {mu_mean:.2f}")
ax.axhline(sigma_mean, color="gray", linestyle="--", alpha=0.6,
           label=f"mean   = {sigma_mean:.2e}")
```

```
# set sane limits
ax.set_xlim(x_lo, x_hi)
ax.set_ylim(y_lo, y_hi)

ax.set_xlabel(" * (overall influence on predicted mass)")
ax.set_ylabel("  (interaction / nonlinearity)")
ax.set_title("Morris Sensitivity Screening - Penguin Body Mass Model")
ax.grid(alpha=0.3)
ax.legend(loc="upper right", frameon=False)

plt.show()
```



4.2.4 Interpreting the Morris screening for penguin mass

We used a linear regression model to predict penguin body mass (g) from bill length, bill depth, and flipper length. We then applied the Morris Elementary Effects method to measure how sensitive the predicted mass is to each morphological input.

Results: - `flipper_length_mm` has the highest *, meaning it is the most influential predictor of body mass overall. Small changes in flipper length lead to large changes in predicted body mass. -* `bill_length_mm` *and* `bill_depth_mm` *also affect predicted mass, but less strongly (their* values are much smaller than flipper length). - All three inputs have values that are ~0. This means the effect of each input is essentially constant across the range of values we sampled. In other words, the model's response to each input is linear and does not depend much on the other inputs.

This is exactly what we'd expect from a plain linear regression with no interaction terms: each predictor contributes additively and with a roughly constant slope. Because is near zero, we see no evidence of strong interactions or nonlinear behavior in this model.

In a more complex ecological model (or in a spatial MCDA with nonlinear suitability thresholds), we would expect to see higher , which would indicate that certain inputs matter only under certain conditions or in combination with other inputs.

4.3 Spatial Sensitivity Analysis with Synthetic Rasters

Now that we understand how Morris' *elementary effects* method works conceptually, let's explore how it applies in a **spatial MCDA (Multi-Criteria Decision Analysis)** setting — the kind used in suitability or recharge mapping.

In a spatial context, our "model" is often a **weighted linear combination (WLC)** of several raster criteria:

- slope
- soil permeability
- rainfall
- land cover, etc.

Each criterion is spatially continuous, and we assign **weights** to express their relative importance. The Morris method lets us vary these weights systematically to see:

- which weights most strongly influence the overall suitability outcome ( ), and
- which weights interact in nonlinear ways (> 0).

To demonstrate this, we'll create a **synthetic dataset of rasters** that mimics realistic environmental layers and intentionally includes **nonlinearity and interactions**.

Our synthetic study area will include:

- `slope` — higher toward one corner (representing uplands),
- `permeability` — highest in a central "valley,"

- `rainfall` — decreasing west to east, with sinusoidal north–south variability.

We'll then define four "factors" that influence a hypothetical **recharge suitability score**:

Symbol	Description	Behavior
w	Weight on slope suitability	favors gentle slopes (nonlinear decay)
w	Weight on soil permeability	roughly linear (more permeable → better)
w	Weight on rainfall suitability	"Goldilocks" response — midrange rainfall best
w	Weight on interaction term	captures synergy between rainfall and permeability

We'll define a composite suitability model:

$$\text{Suitability}(i,j) = w_{\text{slope}}, f_{\text{slope}}(s_{ij}) + w_{\text{perm}}, f_{\text{perm}}(p_{ij}) + w_{\text{rain}}, f_{\text{rain}}(r_{ij}) + w_{\text{int}}, f_{\text{int}}(p_{ij}, r_{ij})$$

Then we'll summarize each run by a single management-style indicator:

% of the landscape with suitability 0.7

Finally, we'll apply the **Morris elementary-effects** method to the weights $((w_{\text{slope}}, w_{\text{perm}}, w_{\text{rain}}, w_{\text{int}}))$ to estimate and — showing which weights matter most and which behave nonlinearly or interactively.

4.3.1 Step 1. Generate synthetic rasters with spatial structure

What we will do:

slope: increases toward one corner (like uplands vs basin floor).

perm: hotspot of high permeability (like an alluvial fan or paleo-channel).

rain: west-to-east gradient plus a sinusoidal north-south climate band.

```
np.random.seed(42)

# grid size
nx, ny = 50, 50

# create coordinate grids (to allow gradients across space)
```

```
x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny)
X, Y = np.meshgrid(x, y, indexing="ij")  # X[i,j], Y[i,j] in [0,1]

# synthetic "slope": higher slope in the NE corner + noise
slope = 30 * (0.3*X + 0.7*Y) + np.random.normal(0, 2, (nx, ny))
slope = np.clip(slope, 0, 30)  # degrees

# synthetic "soil permeability" (perm): better in valley-like band
perm = 0.6 + 0.4*np.exp(-((X-0.5)**2 + (Y-0.2)**2)/0.05)
perm += np.random.normal(0, 0.05, (nx, ny))
perm = np.clip(perm, 0, 1)

# synthetic "rainfall": gradient + hump
rain = 200 + 600*(1 - X) + 80*np.sin(3*np.pi*Y)
rain += np.random.normal(0, 20, (nx, ny))
rain = np.clip(rain, 200, 800)  # mm/yr

slope.shape, perm.shape, rain.shape
```

```
((50, 50), (50, 50), (50, 50))
```

4.3.2 Step 2. define the transformed suitability sub-scores

Now we define the $f_{slope}, f_{perm}, f_{rain}$, and the interaction term.

Design choices:

- We want nonlinearity: e.g. very steep slopes get penalized sharply.
- We want “Goldilocks” rainfall: mid-range rain gives best recharge; too little = no water, too much = maybe runoff.
- We want interaction between perm and rain: rain only helps if perm is high.

```
def score_slope(slope_grid):
    # high score for gentle slopes, decays nonlinearly
    # slope in degrees 0..30
    # we'll do an exponential decay so it's strongly nonlinear
    return np.exp(-slope_grid / 10.0)  # near 1 at 0 deg, ~exp(-3)=0.05 at 30 deg

def score_perm(perm_grid):
    # more permeable is better, roughly linear
    # perm already in [0,1]
    return perm_grid  # identity for now

def score_rain(rain_grid):
    # peak around moderate rainfall (e.g. 500 mm),
    # penalize too dry or too wet via a Gaussian-like curve
    return np.exp(-((rain_grid - 500.0)**2) / (2*(100.0**2)))
    # ~1.0 at 500 mm, drops off as you move away

def score_interaction(perm_grid, rain_grid):
    # interaction: rain only "counts" if perm is high
    # e.g. multiply them, so high perm + decent rain is very good
    # but high rain with low perm won't help
    rain_norm = (rain_grid - 200.0) / (800.0 - 200.0)  # scale rain to [0,1]
    rain_norm = np.clip(rain_norm, 0, 1)
    return perm_grid * rain_norm
```

4.3.3 Step 3. Define the suitability model with weights

we will treat the weights as the “inputs” then we will perturb them with Morris, these are like decision-maker priorities in a WLC

```
def suitability_from_weights(w_slope, w_perm, w_rain, w_int):
    """
    Given a set of weights, compute a final suitability raster,
    then return a management-style scalar metric:
    % of the landscape above a suitability threshold.
    """
    S_slope = score_slope(slope)
    S_perm  = score_perm(perm)
    S_rain  = score_rain(rain)
    S_int   = score_interaction(perm, rain)

    # weighted linear combo including interaction layer
    suitability = (
        w_slope * S_slope +
        w_perm  * S_perm  +
        w_rain  * S_rain  +
        w_int   * S_int
    )
    # normalize by sum of weights so scores stay in a comparable range
    w_sum = (w_slope + w_perm + w_rain + w_int) + 1e-12
    suitability = suitability / w_sum

    # management-style scalar output:
    # fraction of cells with suitability >= 0.7
    high_priority = (suitability >= 0.7).mean()  # this is a single number between 0 and 1

    return float(high_priority)
```

4.3.4 Step 4. Run Morris-style elementary effects on the weights

This is almost identical to what we did with penguins, but now:

- We're sampling weights instead of morphology.
- We're perturbing one weight at a time.
- We'll say each weight varies in [0.1, 1.0] — like minimum importance to strong importance — and we'll use a fractional step.

```
weight_ranges = {
    "w_slope": (0.1, 1.0),
    "w_perm":  (0.1, 1.0),
    "w_rain":  (0.1, 1.0),
    "w_int":   (0.1, 1.0),
}

def sample_elementary_effects_wlc(model_func, weight_ranges, n_samples=100, delta=0.1):
    """
    model_func: suitability_from_weights
    weight_ranges: dict of {weight_name: (min,max)}
    n_samples: number of random base weight sets to test
    delta: step as a fraction of that weight's range
    """
    wnames = list(weight_ranges.keys())
    records = []

    for _ in range(n_samples):
        # pick a random baseline weight set, not at the extreme edges
        base_w = {}
        for wname, (lo, hi) in weight_ranges.items():
            base_w[wname] = np.random.uniform(
                lo + 0.1*(hi-lo),
                hi - 0.1*(hi-lo)
            )

        # baseline model output
        y0 = model_func(
            base_w["w_slope"],
            base_w["w_perm"],
            base_w["w_rain"],
            base_w["w_int"]
        )

        ee_point = {}
        # perturb each weight in turn
        for wname, (lo, hi) in weight_ranges.items():
            step = delta * (hi - lo)

            perturbed_w = base_w.copy()
            perturbed_w[wname] = np.clip(base_w[wname] + step, lo, hi)

            y1 = model_func(
                perturbed_w["w_slope"],
                perturbed_w["w_perm"],
                perturbed_w["w_rain"],
                perturbed_w["w_int"]
            )

            # elementary effect for this weight
            denom = step if step > 1e-12 else 1e-12
            ee_point[f"{wname}_EE"] = (y1 - y0) / denom

        records.append(ee_point)

    return pd.DataFrame(records)

ee_wlc_df = sample_elementary_effects_wlc(
    suitability_from_weights,
    weight_ranges,
    n_samples=100,
    delta=0.1
)
```

```
ee_wlc_df.head()
```

	w_slope_EE	w_perm_EE	w_rain_EE	w_int_EE
0	0.008889	0.013333	0.013333	-0.017778
1	-0.106667	0.066667	0.044444	-0.053333
2	-0.137778	0.075556	0.080000	-0.111111
3	0.000000	0.044444	0.008889	-0.008889
4	-0.004444	0.026667	0.031111	-0.022222

Now we have:

- Each row = one baseline weighting scenario.
- Each column = how sensitive the "% high-priority land" is to each weight at that baseline.
- This is a spatial decision model, with real interactions baked in (w_int affects that interaction layer we defined).

4.3.5 Step 5. Summarize μ^* and σ for the weights

```
summary_wlc = pd.DataFrame({
    "mu_star": [ee_wlc_df[c].abs().mean() for c in ee_wlc_df.columns],
    "sigma":   [ee_wlc_df[c].std(ddof=1)  for c in ee_wlc_df.columns],
}, index=[c.replace("_EE", "") for c in ee_wlc_df.columns])

summary_wlc
```

	mu_star	sigma
w_slope	0.059733	0.048370
w_perm	0.053911	0.026679
w_rain	0.047244	0.035660
w_int	0.053956	0.034369

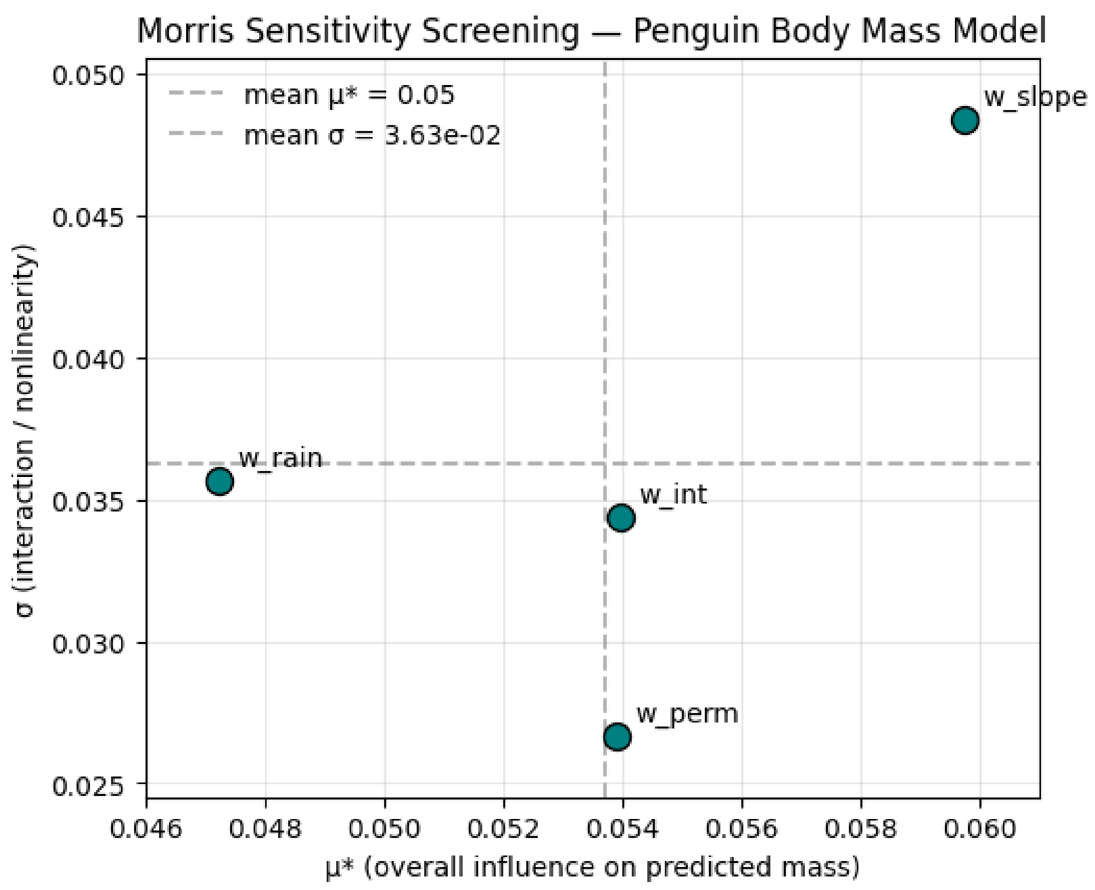
4.3.5.1 Conceptual Questions

Which weight most strongly controls (μ^*) how much of the map is deemed suitable or good (gte 7) Which weight's effect is stable vs. only matters in some regimes? Does the interaction weight (w_int) have a high σ? *(it should because we designed it to matter more when both permeability and rain are favorable)*

4.3.6 Step 6. plot μ^* vs σ for the spatial model

```
mu_vals = summary_wlc["mu_star"].values.astype(float)
sigma_vals = summary_wlc["sigma"].values.astype(float)
names = summary_wlc.index.tolist()

mu_mean = float(np.mean(mu_vals))
sigma_mean = float(np.mean(sigma_vals))

# define plotting limits with a small % padding around actual data
mu_min = mu_vals.min()
mu_max = mu_vals.max()
sigma_min = sigma_vals.min()
sigma_max = sigma_vals.max()

mu_pad = 0.1 * (mu_max - mu_min if mu_max > mu_min else 1.0)
sigma_pad = 0.1 * (sigma_max - sigma_min if sigma_max > sigma_min else 1.0)

x_lo = mu_min - mu_pad
x_hi = mu_max + mu_pad
y_lo = sigma_min - sigma_pad
y_hi = sigma_max + sigma_pad

fig, ax = plt.subplots(figsize=(6,5))

# scatter points
ax.scatter(mu_vals, sigma_vals,
           s=100, color="teal", edgecolor="k", zorder=3)

# adaptive label offset: 2% of axis span instead of hardcoded 0.02
x_offset = 0.02 * (x_hi - x_lo)
y_offset = 0.02 * (y_hi - y_lo)

for x, y, label in zip(mu_vals, sigma_vals, names):
    ax.text(x + x_offset, y + y_offset, label,
            fontsize=10, zorder=4)

# reference lines
ax.axvline(mu_mean, color="gray", linestyle="--", alpha=0.6,
           label=f"mean  * = {mu_mean:.2f}")
ax.axhline(sigma_mean, color="gray", linestyle="--", alpha=0.6,
           label=f"mean   = {sigma_mean:.2e}")
```

```
# set sane limits
ax.set_xlim(x_lo, x_hi)
ax.set_ylim(y_lo, y_hi)

ax.set_xlabel(" * (overall influence on predicted mass)")
ax.set_ylabel("  (interaction / nonlinearity)")
ax.set_title("Morris Sensitivity Screening - Penguin Body Mass Model")
ax.grid(alpha=0.3)
ax.legend(loc="upper left", frameon=False)

plt.show()
```



4.3.6.1 Interpretation

The plot above shows the Morris – diagram for our synthetic spatial suitability model. Each point represents one of the four input weights varied in the analysis:

- μ^* (x-axis) — measures the overall influence of that weight on the model output (mean absolute elementary effect). A larger μ^* means that changing this weight tends to change the suitability outcome more strongly — i.e., it is a more influential parameter overall.
- σ (y-axis) — measures the variability of those effects across the parameter space. A higher σ indicates nonlinear behavior or interactions with other parameters. In other words, the effect of that weight depends on the values of other weights or on specific regions of the input space.

4.3.6.1.1 What the plot shows:

w_{slope} shows the highest μ^* — meaning that the slope criterion is the dominant control on how much land is classified as suitable. This makes intuitive sense: slope strongly affects recharge potential, and our transformation for slope was nonlinear, creating steep contrasts between gentle and steep terrain.

w_{perm} and w_{rain} have moderate μ^* values — they influence suitability, but to a smaller degree. Their relatively low σ values suggest that their effects are more linear and consistent across different weight combinations.

w_{int} (the interaction weight) has a slightly lower μ^* than w_{slope} but a comparable or higher σ — meaning it contributes less to the total suitability on average but behaves nonlinearly. This reflects the fact that its impact depends on the combination of permeability and rainfall, not on either one alone.

therefore: - The slope weight is the most critical lever — changing it will have the greatest and most consistent effect on the area classified as "high-suitability."

- The interaction term is important to monitor — it introduces context-dependent variability and captures coupled effects of rainfall and permeability.
- The other weights behave more linearly, meaning that their influence can be anticipated more easily and modeled with less uncertainty.

This is a hallmark Morris result:

- Parameters with high μ^* and low $\sigma \rightarrow$ strong, predictable influence.
- Parameters with moderate μ^* but high $\sigma \rightarrow$ interactive or nonlinear influence.
- Parameters with low μ^* and low $\sigma \rightarrow$ negligible or near-linear effect.

4.4 Review Questions

4.4.1 Conceptual

1. What is the main purpose of the Morris elementary effects method in sensitivity analysis?
2. How does Morris differ from a simple one-at-a-time (OAT) sensitivity analysis?
3. In the Morris method, what does the elementary effect represent for a given input parameter?
4. Why do we calculate both μ^* (mu-star) and σ (sigma), and what does each measure tell us?
5. What does a high μ^* combined with a low σ imply about a model parameter?
6. What does a high σ (standard deviation of elementary effects) indicate about a parameter's behavior?
7. How do nonlinearities or interactions among variables influence σ in Morris results?
8. Why is Morris sometimes called a "screening" method rather than a "full" global sensitivity method?
9. What is the role of Δ (delta) in the Morris method, and how is it typically chosen?
10. In a spatial MCDA context, what might a model input represent, and what would the model output represent in a Morris analysis?

4.4.2 Interpretation

11. On a μ^*–σ plot, where would you expect to find a highly linear and dominant variable?
12. What does it mean if a point is high on the axis but low on ?
13. In the penguin example, why did values collapse toward zero?
14. In the synthetic raster model, why does the interaction term (w_int) often have a high σ?
15. How might a decision-maker interpret high σ in a spatial suitability model?

4.4.3 Answers

1. The main purpose of the Morris elementary effects method is to identify which input variables have the greatest overall influence on a model's output, and to detect whether their effects are linear, nonlinear, or involve interactions with other inputs.

2. Unlike a simple one-at-a-time (OAT) analysis that tests variables around a single baseline, Morris repeats OAT experiments across many random starting points in the input space, allowing it to capture global (not just local) sensitivity patterns.

3. The elementary effect for a given parameter represents the change in model output resulting from a small perturbation of that parameter, divided by the size of the change (Δ). It approximates the local slope of the model's response surface.

4. μ^* measures the average magnitude of influence of an input (its overall importance), while measures how variable that influence is across the input space (nonlinearity or interaction effects).

5. A high μ^* and low σ indicate a parameter that has a strong, consistent, and largely linear effect on the model output.

6. A high σ means the parameter's influence changes across the input space — suggesting nonlinear behavior or interactions with other parameters.

7. Nonlinearities or interactions cause σ to increase because the sign or magnitude of the elementary effects vary depending on where in the parameter space the sample is taken.

8. Morris is called a "screening" method because it efficiently identifies which parameters are important without requiring full global variance decomposition (like Sobol or FAST). It's often used as a first step before more detailed analyses.

9. Δ (delta) defines the size of the perturbation for each elementary effect. It is typically a small fraction (e.g., 0.1–0.2) of the parameter's total range to ensure that local changes approximate the model's gradient while still exploring meaningful variability.

10. In a spatial MCDA, model inputs might represent layer weights (e.g., slope, permeability, rainfall importance), while the output could represent an aggregated measure such as overall suitability, recharge potential, or % of land above a suitability threshold.

---

11. A highly linear and dominant variable appears toward the lower right of the μ^**–*σ *plot — high* μ^**(strong influence) and low* σ *(consistent, linear behavior).*

12. A point high on the σ *axis but low on* μ^* *represents a parameter with weak average influence but strong nonlinear or context-dependent effects.*

13. In the penguin example, σ *values collapsed toward zero because the regression model was purely linear and additive, so each variable's effect was constant and did not vary with the others.*

14. In the synthetic raster model, the interaction weight (w_int) often has a high because its influence depends jointly on permeability and rainfall — the effect is strong only where both variables are favorable.

15. A decision-maker might interpret high σ *as an indicator of uncertainty or context dependency — meaning that the importance of that criterion changes across the landscape or depends on how other factors are weighted.*

5 Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.