Empirical Dynamic Modeling for Beginners

Author Name: GOSSET Ferdinand
Research Centre: Institute of Oceanography, National Taiwan University
Research Project Title: Empirical Dynamic Modelling for Beginners using pyEDM
Primary Supervisor: Professor Chih-hao Hsieh

1. Project Summary

This online source provides documentation for the Python codes to reproduce the training exercises and results for the paper “Empirical Dynamic Modeling for Beginners” (Chang et al. 2017), originally based on rEDM (Park et al. 2024). The Python implementation is based on pyEDM (Park and Sugihara 2025). This project aims to introduce beginners to the fundamental concepts of Empirical Dynamic Modeling (EDM) through exercises that employ various analytical techniques. Users will utilize simplex projection (Sugihara and May 1990) to visualize state-space reconstructions, enabling the identification of patterns in time-series data. S-map analysis (Sugihara 1994, Deyle et al. 2016) will be conducted to uncover nonlinear relationships and predict future states based on historical data. Convergent Cross-Mapping (CCM) (Sugihara et al. 2012) will help infer the causal relationships between variables, providing insights into the system’s dynamics. Multivariate and multi-view embedding techniques will be applied to analyze interactions between multiple variables and integrate diverse data perspectives (Dixon et al. 1999, Ye and Sugihara 2016). Additionally, perturbation scenario exploration will assess the system’s response to external disturbances (Deyle et al. 2013). Through this exercise, users will gain a basic understanding of complex system behaviors and develop essential skills in data analysis and interpretation using EDM techniques.

2. Research Plan

This online source aims to introduce beginners to EDM using Python codes. It also allows users to gain an understanding of the theoretical aspect of EDM, as well as different techniques (Section 2.2)

2.1 Short-term Goals

The short-term goals of this exercise are to be comfortable with the different techniques of EDM :

1. Simplex Projection (Sugihara and May 1990)
2. S-map analysis (Sugihara 1994, Deyle et al. 2016)
3. Convergent Cross-mapping (Sugihara et al. 2012)
4. Univariate, multivariate, and multi-view embedding (Dixon et al. 1999, Ye and Sugihara 2016)
5. Scenario exploration (Deyle et al. 2013)

2.2 Long-term Goals

The project's long-term goals are to understand the aforementioned techniques, both practically and theoretically, and use them in data analysis when dealing with a nonlinear dynamical system.

References

Chang, C.-W., M. Ushio, and C.-h. Hsieh. 2017. Empirical dynamic modeling for beginners. Ecological Research 32:785-796.

Deyle, E. R., M. Fogarty, C. H. Hsieh, L. Kaufman, A. D. MacCall, S. B. Munch, C. T. Perretti, H. Ye, and G. Sugihara. 2013. Predicting climate effects on Pacific sardine. Proceedings of the National Academy of Sciences, USA 110:6430-6435.

Deyle, E. R., R. M. May, S. B. Munch, and G. Sugihara. 2016. Tracking and forecasting ecosystem interactions in real time. Proceedings of the Royal Society of London B: Biological Sciences 283:20152258.

Dixon, P. A., M. J. Milicich, and G. Sugihara. 1999. Episodic fluctuations in larval supply. Science 283:1528-1530.

Park, J., C. Smith, G. Sugihara, E. Deyle, E. Saberski, and H. Ye. 2024. rEDM: An R package for Empirical Dynamic Modeling.

Park, J., and G. Sugihara. 2025. Python/Pandas toolset for Empirical Dynamic Modeling.

Sugihara, G. 1994. Nonlinear forecasting for the classification of natural time series. Philosophical Transactions of the Royal Society of London, Series A 348:477-495.

Sugihara, G., R. May, H. Ye, C. H. Hsieh, E. Deyle, M. Fogarty, and S. Munch. 2012. Detecting causality in complex ecosystems. Science 338:496-500.

Sugihara, G., and R. M. May. 1990. Nonlinear forecasting as a way of distinguishing chaos from measurement error in a data series. Nature 344:734-741.

Ye, H., and G. Sugihara. 2016. Information leverage in interconnected ecosystems: Overcoming the curse of dimensionality. Science 353:922.

3. Time Series

All the EDM analyses are performed using the Python pyEDM package.

```
import numpy as np
from numpy.random import default_rng
import pandas as pd
from pandas.plotting import autocorrelation_plot
import scipy.stats as st
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from sklearn.preprocessing import StandardScaler
from pyEDM import *
```

3.1 Red noise

The red noise time series is generated using a custom function ***generate_red_noise***, which simulates the well-known stochastic system with high autocorrelation :

```
def generate_red_noise(n=1000, alpha=0.8):
    noise = np.random.normal(size=n)
    red = np.zeros(n)
    for t in range(1, n):
        red[t] = alpha * red[t - 1] + noise[t]
    return red
```

The red noise generator function generates a time series by iteratively applying the red noise equation. It takes the following parameters :

- **n (int, default=1000) :** Number of time points to generate.
- alpha (float, default=0.8) : Autoregression coefficient controlling the memory of the process.
    - If alpha = 0 → pure white noise (no memory).
    - If alpha is close to 1 → highly autocorrelated, smoother red noise.
- Returns a *numpy.ndarray* of length n, containing the red noise time series.

3.2 Logistic map

The logistic map time series is generated using a custom function ***generate logistic map***, which simulates the well-known chaotic system :

```
def generate_logistic_map(n=1000, r=3.7, x0=0.5):
    x = np.zeros(n)
    x[0] = x0
    for t in range(1, n):
        x[t] = r * x[t - 1] * (1 - x[t - 1])
    return x
```

The logistic map function generates a time series by iteratively applying the logistic map equation. It takes the following parameters :

- n (int, default=1000) : The number of time steps to simulate. The output will be a time series of length n.

- r (float, default=3.7) : The growth rate parameter of the logistic map. controling the system's behavior :
    - For 1 < r < 3, the system stabilizes to a fixed point.
    - For 3 < r < ~3.57, it oscillates periodically.
    - For r > ~3.57, it enters chaotic dynamics.
- x0 (float, default=0.5) : The initial value of the population (x_0), typically between 0 and 1.
- Returns a *numpy.ndarray* of length n, representing the dynamics of the logistic map.

3.3 Normalization

These time series are then normalized to zero mean and unit variance using ***normalized_series*** :

```
def normalize_series(series):
    scaler = StandardScaler()
    return scaler.fit_transform(series.reshape(-1, 1)).flatten()
```

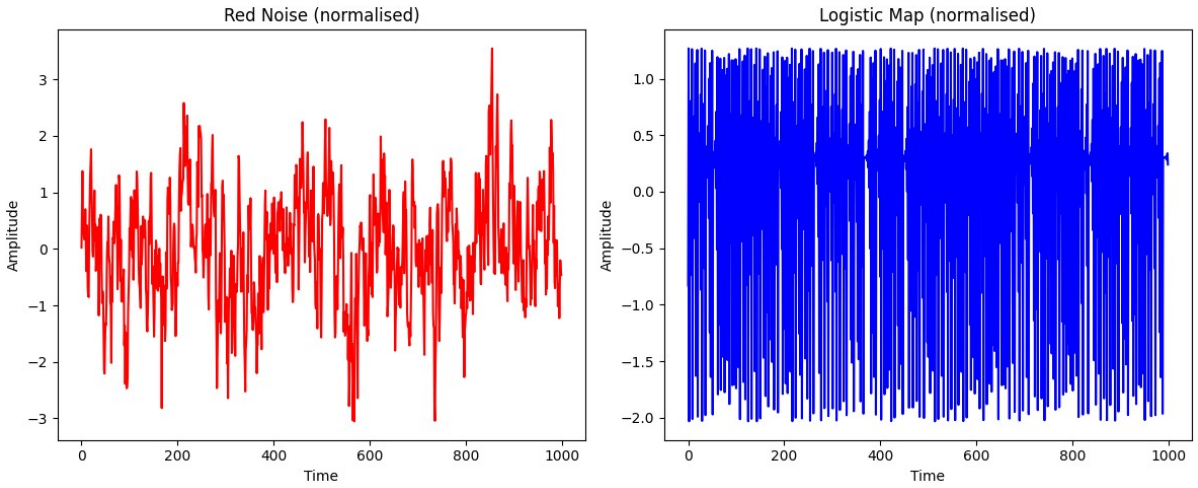


Figure 1 : Time series generated from Red noise (left) and Logistic map (right)

4. Simplex Projection

Using simplex projection (***run_simplex***), we aim to forecast the next time point (i.e., step t+1) based on the past dynamics of a system such as red noise or the logistic map. Before applying this method, each time series is standardized to have zero mean and unit variance.

We split the time series into two equal parts :

- The first half serves as the library (training data) for reconstructing the attractor (manifold).
- The second half is used as the prediction set to evaluate out-of-sample forecast performance.

We apply the simplex projection over a range of embedding dimensions, typically from E = 2 to E = 8, and observe how the forecasting skill (denoted ρ, the correlation between predicted and observed values) varies with E. The ***run_simplex*** function is applied, using the ***pyEDM.Simplex*** function within it.

```
def run_simplex(ts, E, lib_range, pred_range):
```

```
    df = pd.DataFrame({'Time': np.arange(1, len(ts) + 1), 'X': ts})
    preds = Simplex(
        dataFrame=df,
        lib=lib_range,
        pred=pred_range,
        E=E,
        columns="X",
        target="X",
        showPlot=False
    )
    # Verifying the columns exist
    if 'Observations' not in preds or 'Predictions' not in preds:
        return np.nan

    valid = ~preds['Observations'].isna()
    if valid.sum() == 0:
        return np.nan

    rho = preds.loc[valid, ['Observations',
'Predictions']].corr().iloc[0, 1]
    return rho
```

The ***pyEDM.Simplex*** function takes as arguments :

- **dataFrame :** A pandas DataFrame containing at least:
  - A column named "Time" (1-indexed), and
  - The variable(s) to analyze (e.g., "X").

- **lib :** A string specifying the library range (training data), e.g. "1 500".
- **pred :** A string specifying the prediction range (testing data), e.g. "501 1000".
- E : The embedding dimension, i.e. the number of time-lagged coordinates used to reconstruct the system's state space.
- **columns :** The name(s) of the column(s) used for embedding (e.g., "X" for univariate, or "X Y Z" for multivariate).
- **target :** The variable to predict — usually the same as columns in univariate analysis.
- **showPlot (optional) :** If True, automatically displays a prediction plot.

→ The function returns a dictionary-like object with the following key:

"Predictions": a DataFrame containing :

- Time: the time index of each prediction
- Observations: the true values of the target variable
- Predictions: the predicted values using simplex projection.

The ***evaluate_dataframe*** function is then used to evaluate the forecasting skill (ρ) of a time series over a range of embedding dimensions (E) using Simplex projection (***run_simplex***), and returns the results as a clean Pandas DataFrame :

```
def evaluate_dataframe(ts):
    data = []
    for E in E_values:
        rho = run_simplex(ts, E, lib, pred)
        data.append({'E': E, 'rho': rho})
    return pd.DataFrame(data)
```

The optimal E can then be found for both the Red noise and the Logistic map, and the predictive skill can be plotted as an E function :

```
sim_r = evaluate_dataframe(Red)
sim_l = evaluate_dataframe(Logi)

# Finding optimal E values
E_r = sim_r.loc[sim_r['rho'].idxmax(), 'E']
E_l = sim_l.loc[sim_l['rho'].idxmax(), 'E']

print(f"🔴 Optimal E for red noise = {E_r}")
print(f"🔵 Optimal E for logistic map = {E_l}")

# Visualisation with optimal E lines
plt.figure(figsize=(12, 5))

# Panel 1: Red noise
plt.subplot(1, 2, 1)
plt.plot(sim_r['E'], sim_r['rho'], marker='o', color='red', label='Red
noise')
plt.axvline(E_r, color='red', linestyle='--', label=f'Optimal E =
{E_r}')
plt.xlabel("Embedding dimension (E)")
plt.ylabel("Predictive skill (ρ)")
plt.title("Red Noise")
plt.legend()
plt.grid(True)

# Panel 2: Logistic map
plt.subplot(1, 2, 2)
plt.plot(sim_l['E'], sim_l['rho'], marker='o', color='blue',
label='Logistic map')
plt.axvline(E_l, color='blue', linestyle='--', label=f'Optimal E =
{E_l}')
plt.xlabel("Embedding dimension (E)")
```

```
plt.title("Logistic Map")
plt.legend()
plt.grid(True)

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

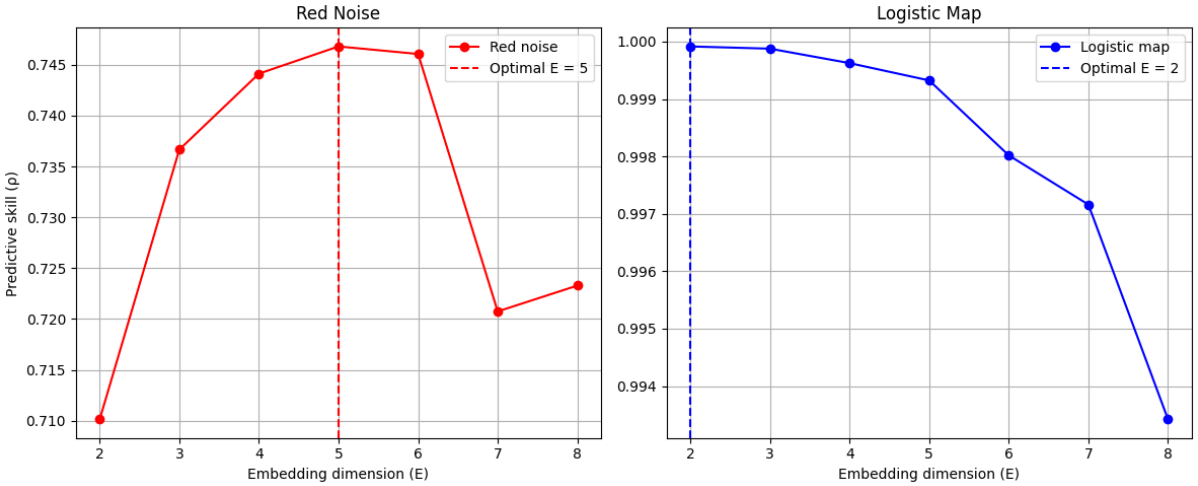


Figure 2 : Simplex Projection : Predictive skill (ρ) as a function of Embedding dimension (E) , with the optimal Embedding dimensions for the Red noise and the Logistic map indicated by the vertical lines

5. Smap analysis

We apply S-map analysis to perform one-step-ahead predictions using the SMap() function from pyEDM. The time series is split into two equal parts :

- The first half is used as the library (training data),
- The second half is used for out-of-sample prediction.

The optimal embedding dimension E for each time series (red noise and logistic map) has already been determined via Simplex projection.

We then test a range of state-dependence parameters θ (theta) from 0 to 2, in increment of 0.1. For each θ, we compute the forecast skill (ρ). The value of θ that gives the highest predictive skill is selected as the optimal state-dependency level.

This process illustrates how the forecasting ability varies with θ and allows us to assess the nonlinear, state-dependent nature of the system.

```
# Defining theta values for the analysis
theta_values = np.arange(0, 2.1, 0.1)
```

The ***run_smap*** function evaluates the forecasting skill (ρ) of a time series using the S-map method over a range of state-dependency parameters θ (theta). It helps analyze how nonlinear and state-dependent the dynamics of the system are. It uses the ***pyEDM*** function, ***pyEDM.SMap***

```
def run_smap(ts, E, lib_range, pred_range, thetas):
    df = pd.DataFrame({'Time': np.arange(1, len(ts) + 1), 'X': ts})
    results = []
    for theta in thetas:
        res = SMap(
            dataFrame=df,

            lib=lib_range,
            pred=pred_range,
            E=E,
            columns="X",
            target="X",
            theta=theta,
            showPlot=False
        )
        # res is expected to be a dictionary with 'predictions' key
        preds = res.get('predictions')  # or a similar key depending on
the library's output


        if preds is None:
            rho = np.nan
        else:
            # preds should be a DataFrame with 'Observations' and
'Predictions'
            valid = ~preds['Observations'].isna()
            if valid.sum() == 0:
                rho = np.nan
            else:
                rho = preds.loc[valid, ['Observations',
'Predictions']].corr().iloc[0, 1]
        results.append({'theta': theta, 'rho': rho})
    return pd.DataFrame(results)
```

The ***pyEDM.SMap*** takes :

- **dataFrame :** A pandas DataFrame containing :
  - A "Time" column (starting from 1), and
  - One or more time series columns (e.g., "X").
- **lib :** A string indicating the library range for training (e.g., "1 500").
- **pred :** A string indicating the prediction range for testing (e.g., "501 1000").

- **E :** The embedding dimension (number of lags used to reconstruct the system's attractor).
- **columns :** The variable(s) used to construct the embedding. This can be a single ("X") or multiple (e.g., "X Y Z")variables.
- **target :** The variable to predict.
- **theta :** A non-negative float that controls the degree of nonlinearity / state-dependence :
    - If theta = 0, SMap reduces to a global linear model.
    - If theta > 0, the model becomes more sensitive to local dynamics.
- **showPlot (optional) :** If True, displays a built-in forecast plot.
- **embedded (optional) :** If True, showing the Smap coefficients.

The SMap() function returns a dictionary with at least the following keys:

- **"predictions" :** A DataFrame containing:
    - **Time :** time index of predictions.
    - **Observations :** true values of the target variable.
    - **Predictions :** predicted values based on S-map.
- "coefficients" (if embedded=True) : A DataFrame showing the Smap coefficients (i.e., partial derivatives), which reflect how each variable influences the target variable over time.

The SMap analysis can then be executed for both Time Series and then plotted with :

```
# 🔴 Red noise
plt.figure(figsize=(6, 4))
plt.plot(smap_r['theta'], smap_r['rho'], color='red')
plt.ylim(0.4, 1.0)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\rho$')
plt.title('Red noise')
plt.grid(True)
plt.show()

# 🔵 Logistic map
plt.figure(figsize=(6, 4))
plt.plot(smap_l['theta'], smap_l['rho'], color='blue')
plt.ylim(0.6, 1)
plt.xlabel(r'$\theta$')
plt.ylabel(r'$\rho$')
plt.title('Logistic map')
plt.grid(True)
plt.show()



the_r = smap_r.loc[smap_r['rho'].idxmax(), 'theta']
the_l = smap_l.loc[smap_l['rho'].idxmax(), 'theta']

print(f"🔴 Optimal theta for red noise = {the_r}")
print(f"🔵 Optimal theta for logistic map = {the_l}")
```

```
print(smap_r)
```

That also prints the optimal thetas :

🔴 Optimal theta for red noise = 0.9, but exhibits almost no variation with different theta. This result indicates that Red noise represents a linear stochastic process.

🔵 Optimal theta for logistic map = 2.0. This result indicates that Logistic map represents a nonlinear dynamical system.

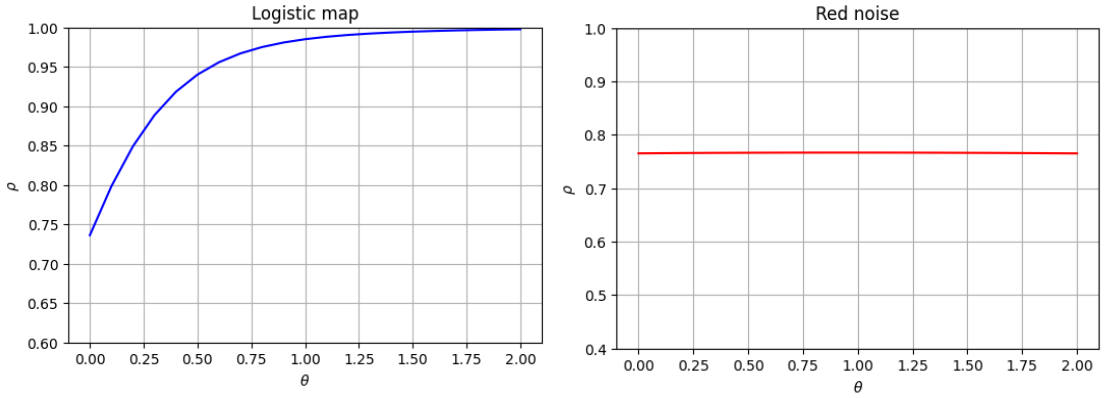


Figure 3 : SMap, the predictive/forecasting skill as a function of theta for the Red noise and the Logistic map

6. Convergent Cross-Mapping

6.1 Implementing the Moran Model

A .csv file is created, using the code below, from which the useful data will then be extracted :

```
# Moran type Data (for example : synchrones oscillations + noise)
t = np.arange(1000)
N1 = 0.5 + 0.2 * np.sin(2 * np.pi * t / 50) + 0.05 *
np.random.randn(1000)
N2 = 0.4 + 0.2 * np.sin(2 * np.pi * t / 50 + np.pi / 6) + 0.05 *
np.random.randn(1000)

df_moran = pd.DataFrame({'time': t, 'N1': N1, 'N2': N2})
df_moran.to_csv('ESM3_Data_moran.csv', index=False)

# competition type data: mirror dynamics + noise
M1 = 0.5 + 0.3 * np.sin(2 * np.pi * t / 60) + 0.05 *
np.random.randn(1000)
M2 = 1.0 - M1 + 0.05 * np.random.randn(1000)

df_compet = pd.DataFrame({'time': t, 'M1': M1, 'M2': M2})
```

```
df_compet.to_csv('ESM4_Data_competition.csv', index=False)
```

Extracting the useful data :

```
from sklearn.preprocessing import StandardScaler

# Loading the file and normalizing the time series data
def load_and_normalize(fp):
    df = pd.read_csv(fp)
    ts = df.iloc[:, 1:]  # Assuming the first column is time or index
    return pd.DataFrame(StandardScaler().fit_transform(ts),
columns=ts.columns)
dam = load_and_normalize('ESM3_Data_moran.csv')
dac = load_and_normalize('ESM4_Data_competition.csv')
```

The Best Embedding dimension can then be found (the E with the best forecasting skill), using ***pyEDM.CCM*** within the ***find_best_E*** function.

```
def find_best_E(df, col_lib, col_target):
    results = []
    col_name = f"{col_lib}:{col_target}"

    for E in range(2, 9):
        out = CCM(
            dataFrame=df,
            columns=col_lib,
            target=col_target,
            E=E,
            Tp=-1,
            libSizes="10 990 980",  # libSize min max step
            sample=1,
            showPlot=False
        )

        if col_name not in out.columns:
            raise ValueError(f"Expected column '{col_name}' not found.
Got: {out.columns}")

        rho = out[col_name].iloc[-1]  #Last value of the correlation
coefficient
        results.append((E, rho))
```

```
    return max(results, key=lambda x: x[1])[0]


E_n1 = find_best_E(dam, "N1", "N2")
E_n2 = find_best_E(dam, "N2", "N1")
E_m1 = find_best_E(dac, "M1", "M2")
E_m2 = find_best_E(dac, "M2", "M1")
print("E pour N1→N2:", E_n1, "– N2→N1:", E_n2)
print("E pour M1→M2:", E_m1, "– M2→M1:", E_m2)
```

A bootstrapped CCM analysis for robust estimation of ρ across library sizes can then be performed using ***run_ccm_bootstrap*** :

```
from scipy.stats import kendalltau
def run_ccm_bootstrap(df, col_lib, col_trg, E, libs, samples=200,
seed=2301):
    out = CCM(
        dataFrame=df, columns=col_lib, target=col_trg,
        E=E, Tp=0, libSizes=" ".join(map(str, libs)),
        sample=samples, seed=seed,
        showPlot=False, includeData=True
    )
    print("Keys returned by pyEDM.CCM:", out.keys())
    return out["PredictStats1"]  # or PredictStats2, depends on the
causality's direction
```

The forecasting skills calculated using Convergent Cross-Mapping can then be plotted using ***summarize_trend*** that summarizes ρ (forecast skill) across bootstrapped replicates and the ***plot_ccm*** functions :

```
     def summarize_trend(df_rho, col_name):
         stats = df_rho.groupby('LibSize')[col_name].quantile([0.25,
0.5, 0.75]).unstack()
         tau = {}
         for q in stats.columns:
             tau[q] = kendalltau(stats.index.astype(float),
stats[q]).correlation
         return stats, tau


     libs = list(range(20, 81, 20)) + list(range(100, 1001, 100))

     for label, df, E in [
```

```
            ("N1→N2", dam, E_n1),
            ("N2→N1", dam, E_n2),
            ("M1→M2", dac, E_m1),
            ("M2→M1", dac, E_m2),
        ]:
            col_lib, col_trg = label.split("→")
            df_rho = run_ccm_bootstrap(df, col_lib, col_trg, E, libs)
            col_name = f"{col_lib}:{col_trg}"
            stats, tau = summarize_trend(df_rho, "rho")
            print(f"\n{label}, tau quantiles : {tau}")

        def plot_ccm(stats, df, cols, libs, color, label):
            x = stats.index  # LibSizes

            # Tracé de la médiane + intervalle interquartile
            plt.plot(x, stats[0.5], color=color, label=label + " median")
            plt.fill_between(x, stats[0.25], stats[0.75], color=color,
alpha=0.3)

            # Affichage des colonnes disponibles
            print(f"\n➡ Verification of the columns in the DataFrame :
{df.columns.tolist()}")

            col1, col2 = cols

            if col1 in df.columns and col2 in df.columns:
                corr0 = df[col1].corr(df[col2])
                print(f"✅ Correlation between{col1} and {col2} : {corr0}")

                if not pd.isna(corr0):
                    plt.axhline(corr0, color=color, linestyle='--',
label=f"corr({col1},{col2})")
                else:
                    print(f"⚠ The correlation is NaN. Dotted line non
tracée.")
            else:
                print(f"❌ Error : one of the columns {col1} or {col2} does not
exist in df.")

        # Recuperating the results for N1→N2 and N2→N1
        res, _ = summarize_trend(run_ccm_bootstrap(dam, "N1", "N2", E_n1,
libs), "rho")
```

```
    print(type(res))
    print(res)
    print(res.columns if hasattr(res, 'columns') else "No columns")
    print(res.index if hasattr(res, 'index') else "No index")

    # Réutilisation des fonctions et affichage
    plt.figure(figsize=(8, 5))

    stats_1, _ = summarize_trend(run_ccm_bootstrap(dam, "N1", "N2",
E_n1, libs), "rho")
    plot_ccm(stats_1, dam, ("N1", "N2"), libs, "red", "N1→N2")

    stats_2, _ = summarize_trend(run_ccm_bootstrap(dam, "N2", "N1",
E_n2, libs), "rho")
    plot_ccm(stats_2, dam, ("N2", "N1"), libs, "blue", "N2→N1")


    plt.xlabel("Library size")
    plt.ylabel("ρ CCM")
    plt.title("CCM - Moran Model")
    plt.ylim(-1, 1)  # to insure that every line is visible
    plt.legend()
    plt.show()
```

The correlation values can also be printed :

✅ Correlation between N1 and N2 : 0.7698
✅ Correlation between N2 and N1 : 0.7698

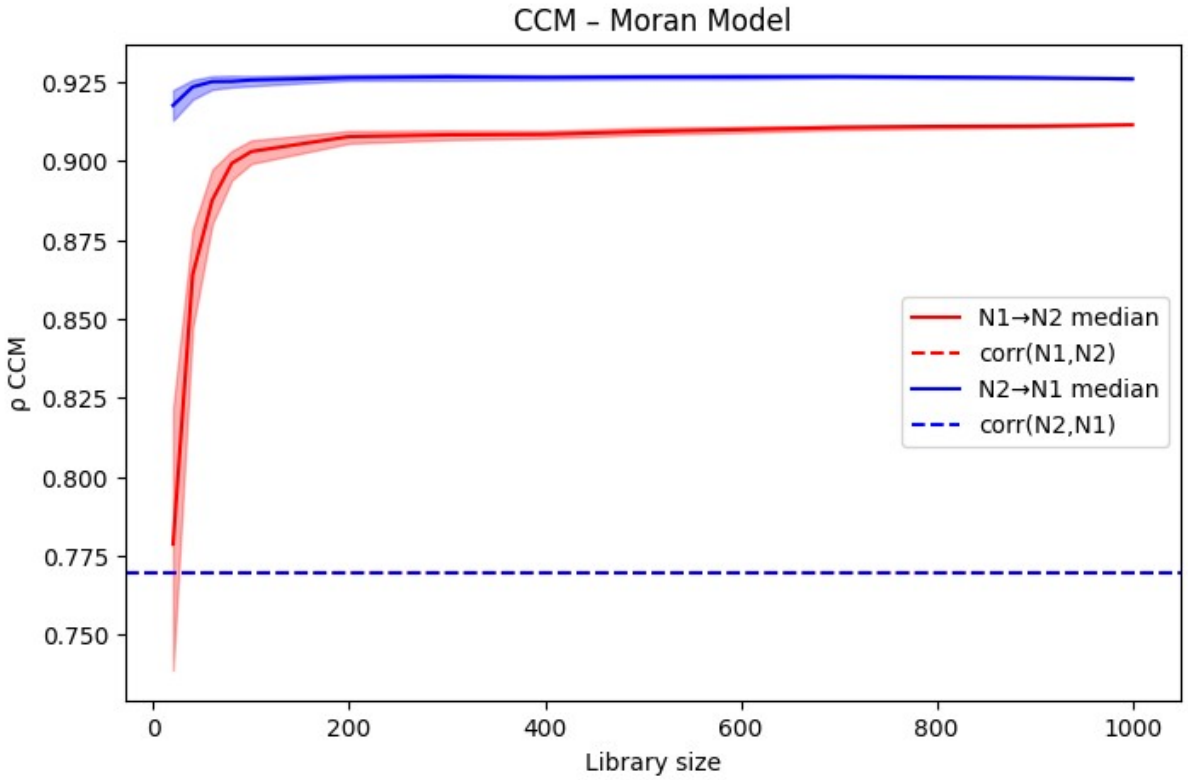


Figure 4 : Convergent Cross-Mapping, Forecasting skills between variables N1 and N2, and their correlation value (N1 and N2 part of the Moran model).

7. Univariate, multivariate and multi-view embeddings

7.1 Data Generation

We start by generating an ecological model with an ODE system with 5 variables (R, C1, C2, P1, P2). This model has a Michaelis-Menten type feeding response in the equations (saturating functional response). It includes terms for growth, consumption, mortality, and interaction strengths.

```
from scipy.integrate import solve_ivp

def food_web(t, y, params):
    R, C1, C2, P1, P2 = y

    # Prevent numerical issues (e.g., division by zero)
    R = max(R, 1e-8)
    C1 = max(C1, 1e-8)
    C2 = max(C2, 1e-8)
    P1 = max(P1, 1e-8)
    P2 = max(P2, 1e-8)
```

```
    # Unpack parameters
    nu1 = params['nu1']
    nu2 = params['nu2']
    lambda1 = params['lambda1']
    lambda2 = params['lambda2']
    C1star = params['C1star']
    C2star = params['C2star']
    mu1 = params['mu1']
    mu2 = params['mu2']
    kappa1 = params['kappa1']
    kappa2 = params['kappa2']
    Rstar = params['Rstar']
    k = params['k']

    # ODEs
    dRdt = R * (1 - R / k) - mu1 * kappa1 * (C1 * R) / (R + Rstar) -
mu2 * kappa2 * (C2 * R) / (R + Rstar)
    dC1dt = mu1 * kappa1 * (C1 * R) / (R + Rstar) - nu1 * lambda1 * (P1
* C1) / (C1 + C1star) - mu1 * C1
    dC2dt = mu2 * kappa2 * (C2 * R) / (R + Rstar) - nu2 * lambda2 * (P2
* C2) / (C2 + C2star) - mu2 * C2
    dP1dt = nu1 * lambda1 * (P1 * C1) / (C1 + C1star) - nu1 * P1
    dP2dt = nu2 * lambda2 * (P2 * C2) / (C2 + C2star) - nu2 * P2

    return [dRdt, dC1dt, dC2dt, dP1dt, dP2dt]

params = {
    'nu1': 0.1, 'nu2': 0.07,
    'lambda1': 3.2, 'lambda2': 2.9,
    'C1star': 0.5, 'C2star': 0.5,
    'mu1': 0.15, 'mu2': 0.15,
    'kappa1': 2.5, 'kappa2': 2.0,
    'Rstar': 0.3,
    'k': 1.2
}

# --- Initial conditions: [R, C1, C2, P1, P2] ---
y0 = [1.0, 0.5, 0.8, 0.7, 0.8]

# --- Time settings ---
tmax = 10000
tau = 5
dt = 0.01
```

```
burn_in_duration = 200
t_eval = np.arange(0, tmax, tau)

# --- Burn-in phase (before integration) ---
Xi = np.array(y0)
for _ in range(int(burn_in_duration / dt)):
    dXi = food_web(0, Xi, params)
    Xi = Xi + dt * np.array(dXi)

# --- Integrate ODE system with post-burn initial condition ---
sol = solve_ivp(
    fun=food_web,
    t_span=(0, tmax),
    y0=Xi,
    args=(params,),
    t_eval=t_eval,
    method='RK45',
    rtol=1e-6,
    atol=1e-9
)
# Convert solution to DataFrame
df = pd.DataFrame({
    'time': sol.t,
    'Resource': sol.y[0],
    'Consumer1': sol.y[1],
    'Consumer2': sol.y[2],
    'Predator1': sol.y[3],
    'Predator2': sol.y[4]
})
```

Where :

- **R:** a resource (e.g., plant or nutrient),
- **C1, C2:** two consumers (e.g., herbivores),
- **P1, P2:** two predators, each preying on one of the consumers.

The mathematical model of ODEs (Ordinary differential equations) composed of :

- **Resource Dynamics** (R) , where k is the carrying capacity.
- **Consumer Dynamics**, (C1 and C2)
- **Predator Dynamics**, (P1 and P2)

The population density can be plotted to confirm that it is indeed a chaotic system :

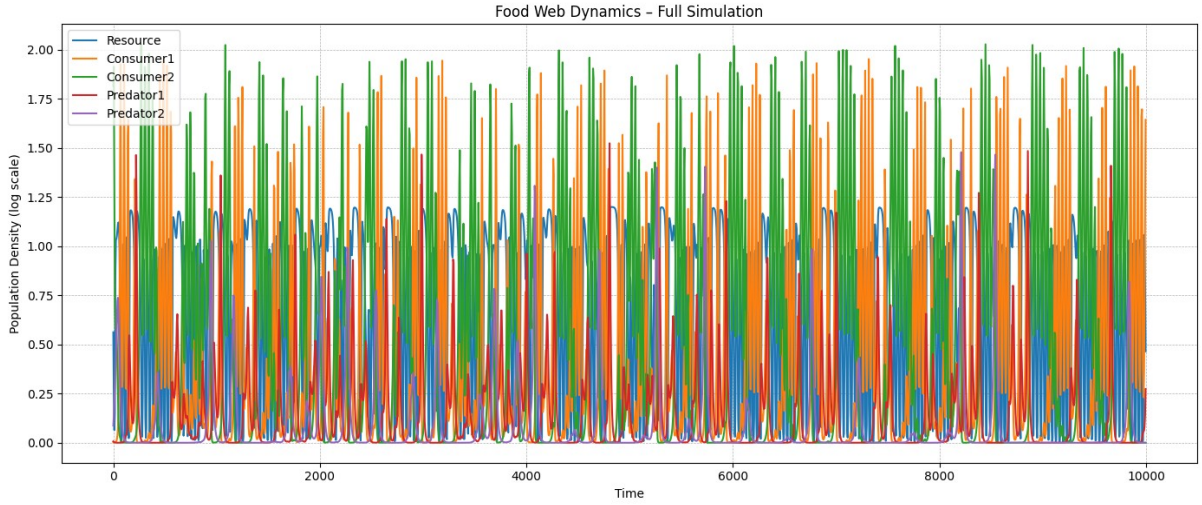


Figure 5 : 5-species foodweb dynamics through time

With a closer look, as well as a logarithmic scale :

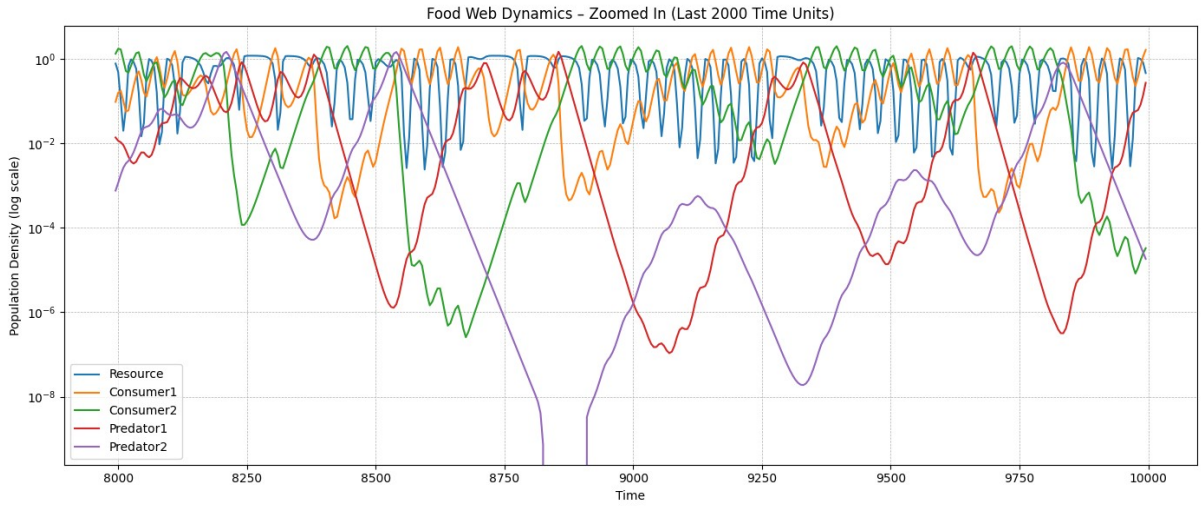


Figure 6 : Last 2000 time steps of the 5-species foodweb dynamics through time at log scale

7.2 Univariate Embedding

We also create the library and the prediction range (*lib* used for learning dynamics and *pred* used for testing/predicting), and create a usable Data frame for the ***pyEDM.Simplex*** that will be used:

```
df_C1 = pd.DataFrame({
    "time": sol.t,
    "C1": sol.y[1]  # Consumer1 time series
})

lib_start = 1
```

```
    lib_end = len(df_C1) // 2
    pred_start = lib_end + 1
    pred_end = len(df_C1)
```

The best embedding dimension can then be found :

```
results_list = []

# Looping on different embedding dimensions E
for E in range(1, 20):
    print(f"\nTesting E={E}")
    res = Simplex(
        dataFrame=df_C1,
        lib=f"{lib_start} {lib_end}",
        pred=f"{pred_start} {pred_end}",
        E=E,
        columns="C1",
        target="C1",
    )
 # Deleting rows with NaN in Predictions or Observations
    res = res.dropna(subset=["Predictions", "Observations"])

    obs_valid = res['Observations'].values
    pred_valid = res['Predictions'].values

    print(res.head())
    print(f"Number of valid predictions for E={E} : {len(obs_valid)}")
```

The Mean Absolute Error can now be calculated and the results plotted :

```
    if len(obs_valid) > 0:
        mae = np.mean(np.abs(obs_valid - pred_valid))
    else:
        mae = np.nan

    print(f"Estimated MAE for E={E} = {mae}")
    results_list.append({"E": E, "MAE": mae})

# Results as a DataFrame
results_df = pd.DataFrame(results_list)

# Selecting and printing the best embedding dimension E based on MAE
if results_df['MAE'].notna().any():
```

```
    bestE = results_df.loc[results_df['MAE'].idxmin(), 'E']
    print(f"\nBest embedding dimension : E = {bestE}")

    # Run Simplex with best E
    simp_C1 = Simplex(
        dataFrame=df_C1,
        lib=f"{lib_start} {lib_end}",
        pred=f"{pred_start} {pred_end}",
        E=int(bestE),
        columns="C1",
        target="C1",
    )

    simp_C1 = simp_C1.dropna(subset=["Predictions", "Observations"])
    obs_valid = simp_C1['Observations'].values
    pred_valid = simp_C1['Predictions'].values

    # Create figure with 2 subplots side by side
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Left panel: MAE vs E
    axes[0].plot(results_df['E'], results_df['MAE'], marker='o')
    axes[0].set_xlabel("Embedding dimension E")
    axes[0].set_ylabel("MAE")
    axes[0].set_title("Best embedding dimension based on MAE")
    axes[0].grid(True)

    # Right panel: Observed vs Predicted scatter
    axes[1].scatter(obs_valid, pred_valid, alpha=0.6)
    axes[1].plot([obs_valid.min(), obs_valid.max()], [obs_valid.min(),
obs_valid.max()], 'r--')
    axes[1].set_xlabel("Observed")
    axes[1].set_ylabel("Prediction")
    axes[1].set_title(f"Univariate simplex projection (E={bestE})")
    axes[1].grid(True)

    plt.tight_layout()
    plt.show()

    # Rho calculation
    rho = np.corrcoef(obs_valid, pred_valid)[0, 1]
    print(f"Prediction skill (ρ) : {rho:.3f}")
```

```
else:
    print("Error : No valid MAE values found for any embedding
dimension E.")
```

```
Best embedding dimension : E = 3
Prediction skill (ρ) : 0.985
```

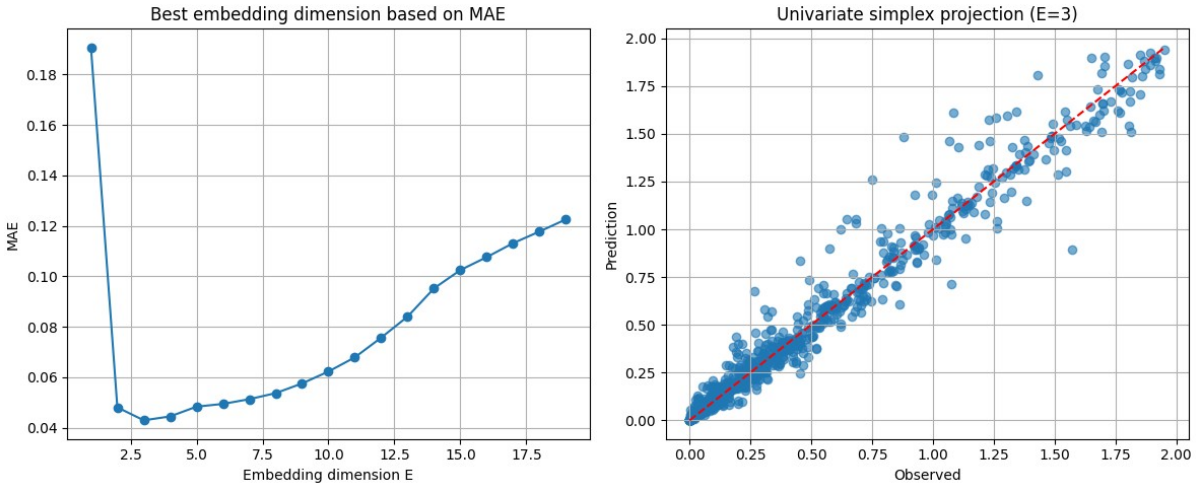


Figure 7 : Univariate simplex projection : Mean Absolute Error as function of Embedding dimension and the Observed vs Predicted values using an E=3 (the best E with lowest MAE).

7.3 Multivariate Embedding

Here, multivariate nonlinear time series are analysed using Simplex projection from pyEDM, focusing on forecasting species C1 based on its own dynamics and interactions with R and P1.

The same five species model is used here :

```
# 1. Extract variables from your solution
C1 = sol.y[1]  # Consumer1
R = sol.y[0]   # Resource
P1 = sol.y[3]  # Predator1

# 2. Normalize each time series
C1 = (C1 - np.mean(C1)) / np.std(C1)
R = (R - np.mean(R)) / np.std(R)
P1 = (P1 - np.mean(P1)) / np.std(P1)
```

We then create the Data Frame, in order to evaluate the data using the ***pyEDM.Simplex*** function :

```
# 3. Create DataFrame
T = len(sol.t)
d = pd.DataFrame({
    "time": np.arange(1, T + 1),
    "C1": C1,
```

```
        "R": R,
        "P1": P1
    })

    # 4. Prepare multivariate block and normalize
    embedding = ["C1", "R", "P1"]
    block = d[embedding]
    block = (block - block.mean()) / block.std()
    block["time"] = np.arange(1, T + 1)  # Required by pyEDM

    # 5. Define library and prediction ranges
    lib_start = 1
    lib_end = len(block) // 2
    pred_start = lib_end + 1
    pred_end = len(block)

    lib = f"{lib_start} {lib_end}"
    pred = f"{pred_start} {pred_end}"

    print("lib =", lib)
    print("pred =", pred)
```

The best embedding dimension can then be found :

```
    # --- 6. Find best embedding dimension E by minimizing MAE ---
    results_list = []
    max_E = 20  # Max embedding dimension to test; adjust as needed

    for E in range(1, max_E + 1):
        print(f"\nTesting multivariate E={E}")
        res = Simplex(
            dataFrame=block,
            lib=lib,
            pred=pred,
            E=E,
            columns="C1 R P1",
            target="C1",
            embedded=True
        )
        res = res.dropna(subset=["Predictions", "Observations"])
        if len(res) > 0:
            mae = np.mean(np.abs(res["Observations"] -
res["Predictions"]))
```

```
        else:
            mae = np.nan
        print(f"MAE for E={E}: {mae}")
        results_list.append({"E": E, "MAE": mae})

    results_df = pd.DataFrame(results_list)
    bestE = results_df.loc[results_df['MAE'].idxmin(), 'E']
    print(f"\nBest multivariate embedding dimension E = {bestE}")
```

We can then use the Simplex Projection like before, filter out the Nan values, and plot the results :

```
    # --- 7. Run Simplex with best E ---
    mult_simp_C1 = Simplex(
        dataFrame=block,
        lib=lib,
        pred=pred,
        E=bestE,
        columns="C1 R P1",
        target="C1",
        embedded=True
    )

    # Remove rows with NaNs
    mult_simp_C1 = mult_simp_C1.dropna(subset=["Predictions",
"Observations"])
    C1_pred_mult = mult_simp_C1["Predictions"].values
    C1_obs_mult = mult_simp_C1["Observations"].values

    # --- 8. Compute forecast skill (correlation) ---
    valid = ~np.isnan(C1_obs_mult) & ~np.isnan(C1_pred_mult)

    if valid.sum() == 0:
        print("No valid points: forecast skill cannot be computed.")
        rho = np.nan
    else:
        rho = np.corrcoef(C1_obs_mult[valid], C1_pred_mult[valid])[0,
1]
        print(f"\nForecast skill (ρ) = {rho:.3f}")

    # --- 9. Plot observed vs predicted ---
    plt.figure(figsize=(6, 6))
    plt.scatter(C1_obs_mult[valid], C1_pred_mult[valid], alpha=0.6)
```

```
    plt.plot([C1_obs_mult.min(), C1_obs_mult.max()],
             [C1_obs_mult.min(), C1_obs_mult.max()],
             'r--')
    plt.xlabel("Observed C1")
    plt.ylabel("Predicted C1")
    plt.title(f"Multivariate Simplex Projection\n(C1 ~ C1, R, P1),
E={bestE}, ρ={rho:.2f}")
    plt.grid()
    plt.show()
```

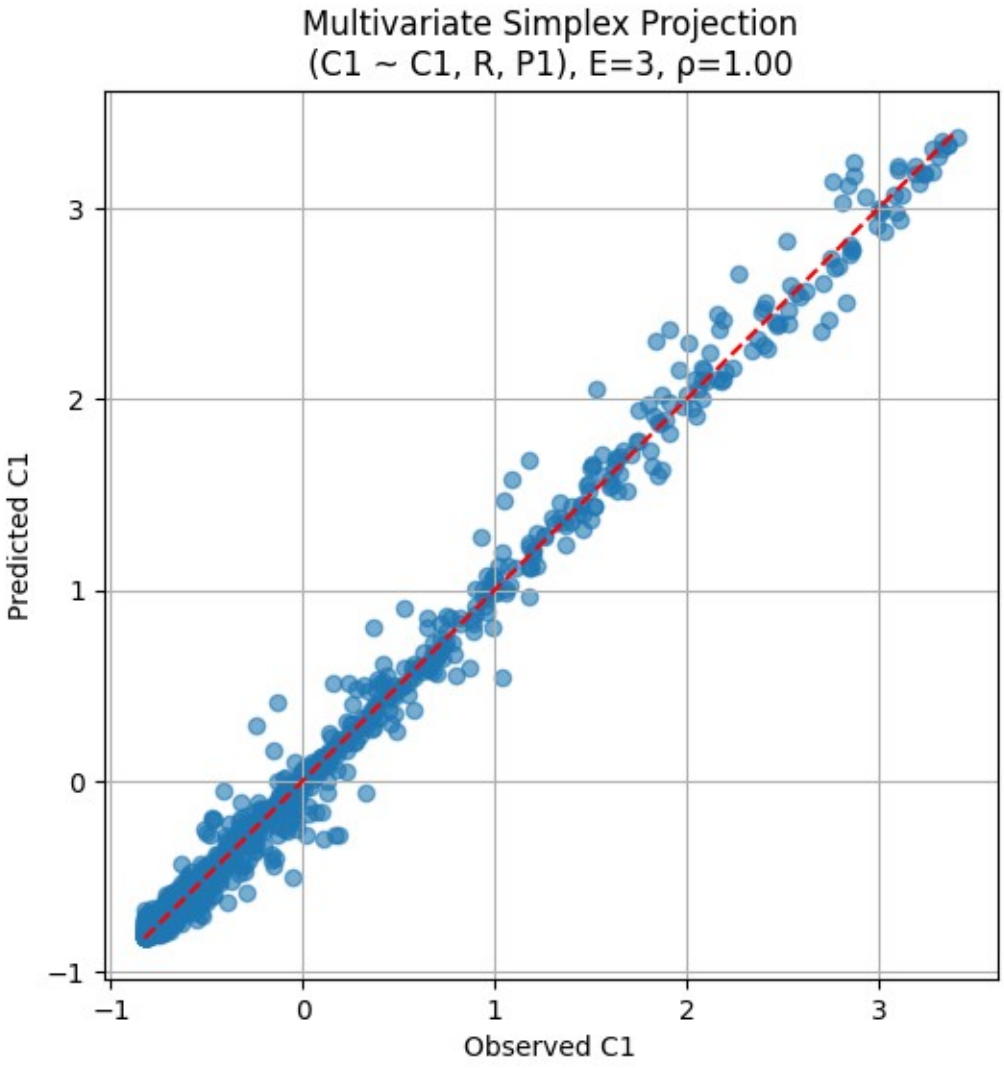


Figure 8 : Multivariate Simplex Projection (Best multivariate embedding dimension E = 3)

7.4 Multi-view Embedding

Multiview Embedding Forecasting can then be performed on the five-species chaotic system (C1, R, P1) using pyEDM.

```
    # --- 1. Extract variables from solution ---
```

```
    C1 = sol.y[1]  # Consumer1
    R = sol.y[0]   # Resource
    P1 = sol.y[3]  # Predator1

    # --- 2. Normalize ---
    C1 = (C1 - np.mean(C1)) / np.std(C1)
    R = (R - np.mean(R)) / np.std(R)
    P1 = (P1 - np.mean(P1)) / np.std(P1)

    # --- 3. Create DataFrame ---
    T = len(sol.t)
    data_used = np.arange(T)
    d = pd.DataFrame({
        "time": data_used + 1,
        "C1": C1,
        "R": R,
        "P1": P1
    })

    # --- 4. Create multiview input block and normalize ---
    embedding = ["C1", "R", "P1"]
    block = d[embedding]
    block = (block - block.mean()) / block.std()
    block["time"] = data_used + 1  # required by pyEDM

    # --- 5. Define library and prediction ranges ---
    lib_start = 1
    lib_end = len(block) // 2
    pred_start = lib_end + 1
    pred_end = len(block)

    lib = f"{lib_start} {lib_end}"
    pred = f"{pred_start} {pred_end}"

    print("lib =", lib)
    print("pred =", pred)
```

The best Embedding dimension can then be found :

```
# --- 6. Find best embedding dimension E by minimizing MAE with Simplex
---
max_E = 10  # max E to try, adjust if needed
results_list = []
```

```
for E in range(2, max_E + 1):
    print(f"Testing Simplex E={E}")
    res = Simplex(
        dataFrame=block,
        lib=lib,
        pred=pred,
        E=E,
        columns="C1 R P1",
        target="C1",
        embedded=True
    )
    res = res.dropna(subset=["Predictions", "Observations"])
    if len(res) > 0:
        mae = np.mean(np.abs(res["Observations"] - res["Predictions"]))
    else:
        mae = np.nan
    print(f"MAE: {mae:.4f}")
    results_list.append({"E": E, "MAE": mae})

results_df = pd.DataFrame(results_list)
bestE = results_df.loc[results_df['MAE'].idxmin(), 'E']
print(f"\nBest embedding dimension found: E = {bestE}")
```

After extracting the variables from the previous system, the ***pyEDM.Multiview*** function can then be applied, using the optimal Embedding Dimension, and the results plotted :

```
# --- 7. Run Multiview forecasting with best E ---
multiview_C1 = Multiview(
    dataFrame=block,
    lib=lib,
    pred=pred,
    columns="C1 R P1",
    target="C1",
    E=bestE
)

# --- 8. Extract predictions and observations ---
C1_obs_multv = multiview_C1['Predictions']['Observations'].values
C1_pred_multv = multiview_C1['Predictions']['Predictions'].values

# Optional: limit points for plotting if dataset is large
plot_limit = 200
C1_obs_multv = C1_obs_multv[:plot_limit]
```

```
C1_pred_multv = C1_pred_multv[:plot_limit]

# Filter NaNs
valid = ~np.isnan(C1_obs_multv) & ~np.isnan(C1_pred_multv)
C1_obs_multv = C1_obs_multv[valid]
C1_pred_multv = C1_pred_multv[valid]

# --- 9. Compute and print forecast skill ---
if len(C1_obs_multv) == 0:
    print("No valid points: forecast skill cannot be computed.")
    rho = np.nan
else:
    rho = np.corrcoef(C1_obs_multv, C1_pred_multv)[0, 1]
    print(f"Forecast skill (ρ) = {rho:.3f}")

# --- 10. Plot observed vs predicted ---
plt.figure(figsize=(6, 6))
plt.scatter(C1_obs_multv, C1_pred_multv, alpha=0.6)
plt.plot([C1_obs_multv.min(), C1_obs_multv.max()],
         [C1_obs_multv.min(), C1_obs_multv.max()],
         'r--')
plt.xlabel("Observed C1")
plt.ylabel("Predicted C1")
plt.title(f"Multiview Forecast (C1 ~ C1, R, P1), E={bestE},
ρ={rho:.2f}")
plt.grid()
plt.show()
```

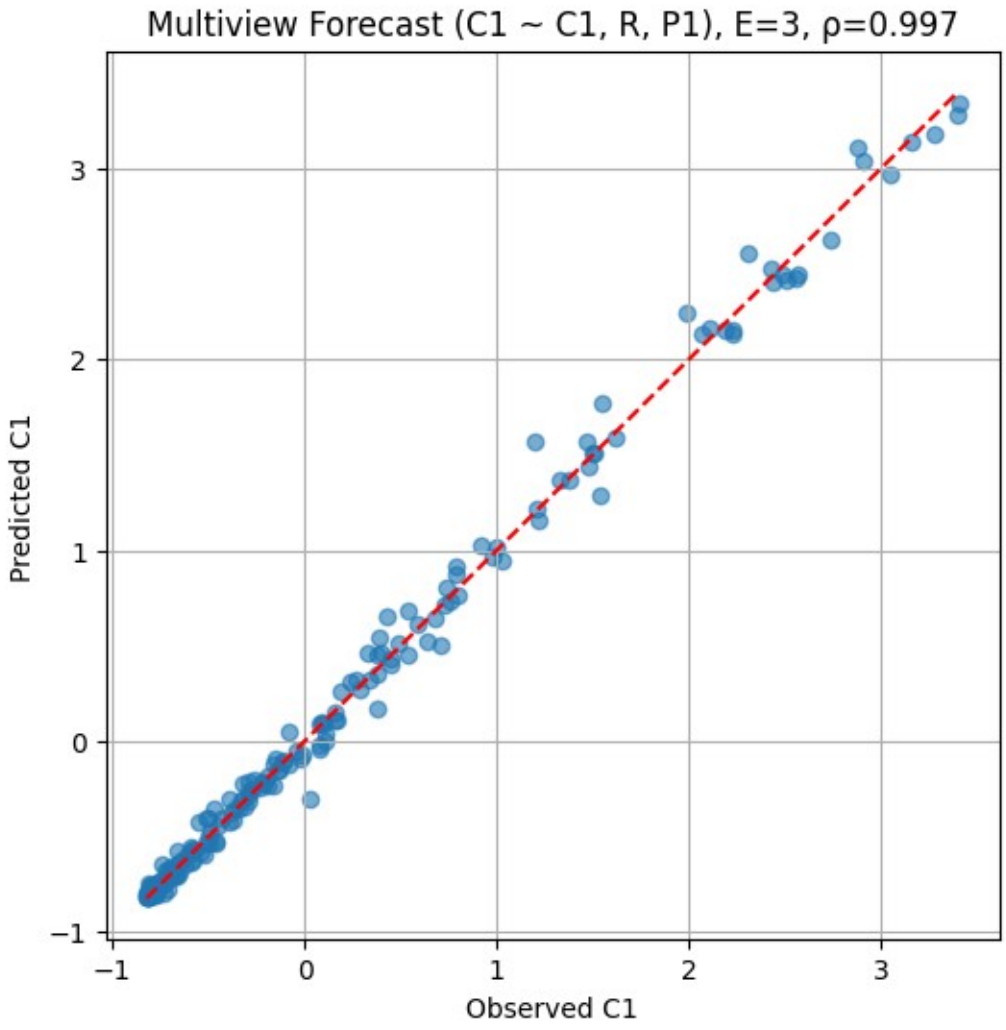
The ***pyEDM.Multiview*** function takes as arguments :

- **dataFrame :** The input time series data as a Pandas DataFrame. In our case, the block contains the standardized variables C1, R, and P1 plus time.
- **lib :** The library range as a string, e.g. "1 500". This defines the rows of dataFrame used for training (model building).
- **pred :** The prediction range as a string, e.g. "501 1000". This defines the rows used for forecasting and evaluation.
- **columns :** A string specifying which variables to use for embedding (e.g., "C1 R P1"). These are used to construct all possible combinations of E variables.
- **target :** The variable you are trying to predict. Here, you're forecasting C1.
- **E :** The embedding dimension. This tells pyEDM to consider all combinations of E=3 variables from the set of columns for building each view.

And it returns a Dataframe with the following columns :

- **Time :** The time index corresponding to each forecast.
- **Observations :** The actual value of the target variable (here, C1) at each time step.

- **Predictions :** The combined forecast of the target variable, averaged over the top multiviews.
- **View1, View2, ...(optional)** : Individual predictions from each view, if requested.




```
Forecast skill (ρ) = 0.997
```

Figure 9 : Multiview Embedding Forecast

7.5 Compare univariate, multivariate, and multiview embeddings

After testing all those methods : univariate embedding, multivariate embedding and multiview embedding, we can now compare them, and see which one would be the most efficient for our system :

```
# rho univariate (Simplex univariate)
rho_univariate = np.corrcoef(obs_valid, pred_valid)[0, 1]  # from the
last univariate simplex projection

# rho multivariate (Simplex multivariate)
rho_multivariate = np.corrcoef(C1_obs_mult, C1_pred_mult)[0, 1]

# rho multiview
```

```python
C1_pred_multv = multiview_C1['Predictions'].Predictions[1:200]
C1_obs_multv = multiview_C1['Predictions'].Observations[1:200]
rho_multiview = np.corrcoef(C1_obs_multv, C1_pred_multv)[0, 1]

# Printing the forecasting skills
print(f"Forecasting skill (ρ) Univariate : {rho_univariate:.3f}")
print(f"Forecasting skill (ρ) Multivariate : {rho_multivariate:.3f}")
print(f"Forecasting skill (ρ) Multiview : {rho_multiview:.3f}")

# Reassembling the rhos
rhos = {
    "Univariate": rho_univariate,
    "Multivariate": rho_multivariate,
    "Multiview": rho_multiview
}

# Plotting the barplot
plt.figure(figsize=(7, 5))
bars = plt.bar(rhos.keys(), rhos.values(), color=["C0", "C1", "C2"])
plt.ylim([min(rhos.values()) - 0.02, max(rhos.values()) + 0.05])
plt.ylabel(r"Forecasting skill ($\rho$)")
plt.title("Comparison of Forecasting Skill by Embedding Method")
plt.grid(axis="y", linestyle="--", alpha=0.7)

for bar in bars:
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height + 0.005,
        f"{height:.3f}",
        ha='center',
        va='bottom',
        fontsize=10,
        fontweight='bold'
    )

plt.show()
```

```
Forecasting skill (ρ) Univariate : 0.985
Forecasting skill (ρ) Multivariate : 0.995
```

```
Forecasting skill (ρ) Multiview : 0.996
```

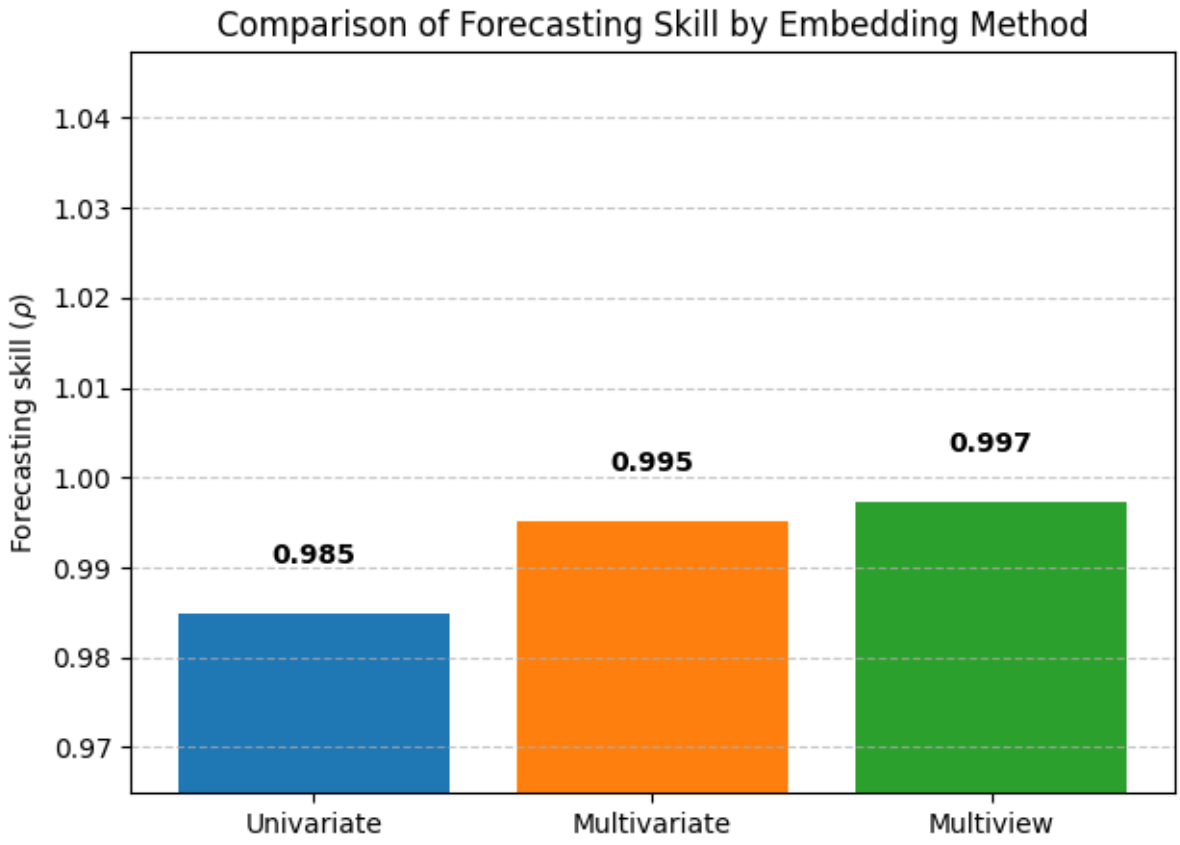


Figure 10 : Comparing Univariate, Multivariate and multiview embeddings

8. Tracking interaction strength using S-map

8.1 Simulation and Time Series

The goal of this part is to visualise the interaction strengths among variables through time. To be able to observe this, the same 5-species ecological system will be simulated over time.

```
# --- Step 1: Extract variables from the food web solution ---
# The variables indices in sol.y:
# sol.y[0] = Resource (R)
# sol.y[1] = Consumer 1 (C1)
# sol.y[2] = Consumer 2 (C2)
# sol.y[3] = Predator 1 (P1)
# sol.y[4] = Predator 2 (P2) - excluded here for simplicity

# --- Step 1: Extract variables from the food web solution ---
R = sol.y[0]
C1 = sol.y[1]
C2 = sol.y[2]
P1 = sol.y[3]
```

```
# --- Step 2: Normalize each variable ---
def normalize(x):
    return (x - np.mean(x)) / np.std(x)

R = normalize(R)
C1 = normalize(C1)
C2 = normalize(C2)
P1 = normalize(P1)

# --- Step 3: Create DataFrame ---
df = pd.DataFrame({
    "R": R,
    "C1": C1,
    "C2": C2,
    "P1": P1
})

df["Time"] = np.arange(len(df)) + 1  # 1-based time index for pyEDM

# --- Step 4: Define library and prediction intervals ---
data_length = len(df)
lib_start = 1
lib_end = data_length // 2
pred_start = lib_end + 1
pred_end = data_length

lib = f"{lib_start} {lib_end}"
pred = f"{pred_start} {pred_end}"
```

8.2 S-Map analysis

We can now implement a complete S-Map (sequential locally weighted global linear map) analysis using pyEDM to understand the time-varying interactions in a nonlinear ecological system.

First we prepare the Dataframe, and choose the range of thetas through which we will try to find the best one :

```
# --- Step 5: Define embedding and theta values ---
embedding = ["R", "C1", "C2", "P1"]
df_used = df.iloc[lib_start - 1:pred_end].reset_index(drop=True)

thetas = [0, 1e-4, 3e-4, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 0.5, 0.75,
1, 1.5, 2, 3, 4, 6, 8]
results = []
```

A first ***pyEDM.SMap*** function can then be applied to find the best theta :

```
# --- Step 6: Tune theta based on prediction of C1 ---
for theta in thetas:
    res = SMap(
        dataFrame=df_used,
        lib=lib,
        pred=pred,
        columns=embedding,
        target="C1",
        theta=theta,
        E=len(embedding),
        embedded=True,
        showPlot=False
    )
    observations = res['predictions'].Observations.to_numpy()
    predictions = res['predictions'].Predictions.to_numpy()
    valid = ~np.isnan(observations) & ~np.isnan(predictions)
    mae = mean_absolute_error(observations[valid], predictions[valid])
    results.append((theta, mae))

# --- Step 7: Plot MAE vs Theta ---
df_theta = pd.DataFrame(results, columns=["theta", "mae"])
plt.figure(figsize=(8, 4))
plt.plot(df_theta["theta"], df_theta["mae"], marker='o')
plt.xlabel("Theta")
plt.ylabel("MAE")
plt.title("Looking for the best theta (S-Map on C1)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Best Theta : 8.0
```

And the results can then plotted :

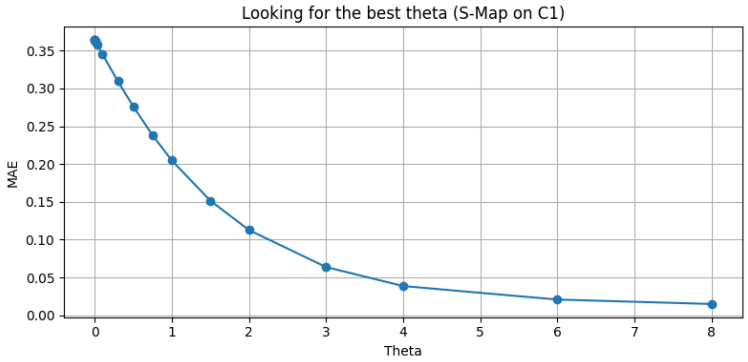


Figure 11 : SMap, Mean Absolute Error (MAE) as a function of theta (the best theta will be the one with the lowest MAE theta=8)

Then, a second ***pyEDM.SMap*** can be applied, to compare the predicted values to the observed ones :

```
# --- Step 8: Best theta and final SMap run ---
best_theta = df_theta.loc[df_theta["mae"].idxmin(), "theta"]
print(f"Best Theta : {best_theta}")

smap_res = SMap(
    dataFrame=df_used,
    lib=lib,
    pred=pred,
    columns=embedding,
    target="C1",
    theta=best_theta,
    E=len(embedding),
    embedded=True,
    showPlot=False
)

# --- Step 9: Observed vs Predicted ---
predictions_data = smap_res['predictions']
plt.figure(figsize=(6, 6))
plt.scatter(predictions_data.Observations,
predictions_data.Predictions, alpha=0.5)
min_val = min(predictions_data.Observations.min(),
predictions_data.Predictions.min())
max_val = max(predictions_data.Observations.max(),
predictions_data.Predictions.max())
plt.plot([min_val, max_val], [min_val, max_val], 'r--')
plt.xlabel("Observed")
```

```
plt.ylabel("Predicted")
plt.title(f"S-Map Prediction (Target: C1)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

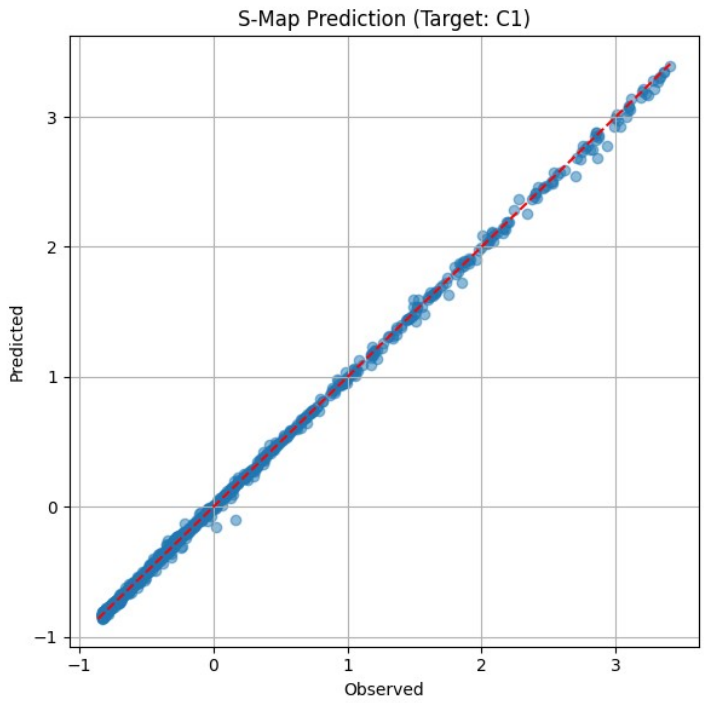


Figure 12 : SMap, predicted values vs observed values for the best theta (theta=8)

Finally, the different time-varying partial derivatives (Smap coefficients) can be extracted and then plotted :

```
# --- Step 10: Plot Coefficients over Time ---
coefficients = smap_res['coefficients']
print("Coefficient columns:", coefficients.columns.tolist())

time_range = range(min(200, len(coefficients)))  # limit to 200 time
steps
plt.figure(figsize=(10, 5))
```

```
plt.plot(time_range, coefficients.loc[time_range, 'dC1/dR'],
label='dC1/dR', color="royalblue", linewidth=2)
plt.plot(time_range, coefficients.loc[time_range, 'dC1/dC2'],
label='dC1/dC2', color="red", linewidth=2)
plt.plot(time_range, coefficients.loc[time_range, 'dC1/dP1'],
label='dC1/dP1', color="springgreen", linewidth=2)
plt.axhline(0, color="black", linestyle="dashed", linewidth=0.8)
plt.xlabel("Time")
plt.ylabel("Interaction strength")
plt.title("S-map Coefficients (Time-varying interaction strengths)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

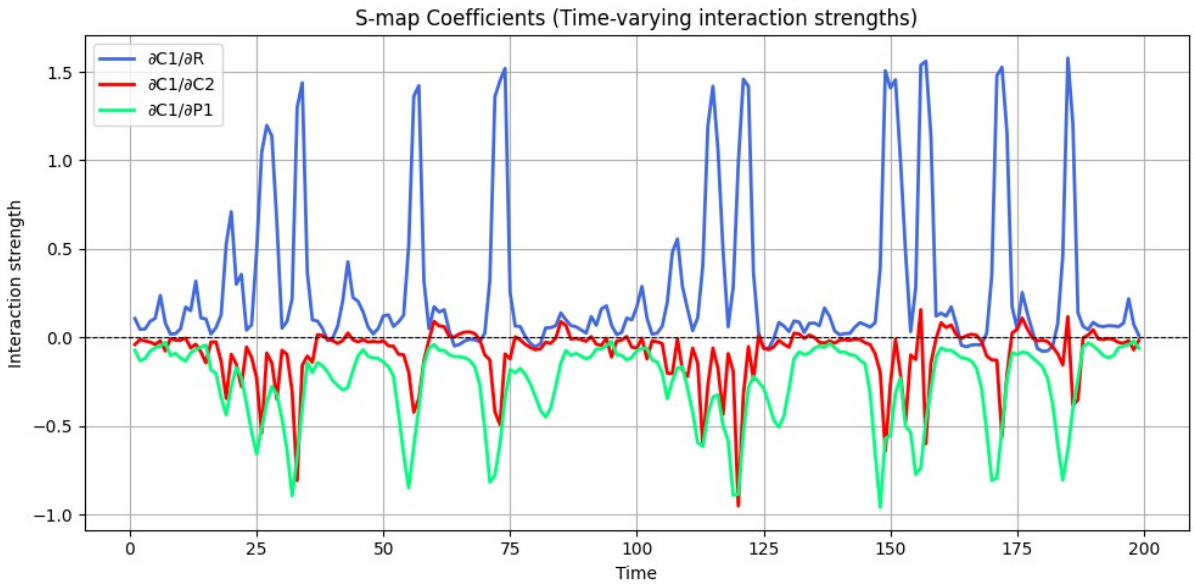


Figure 13 : SMap, Time Variation of the different coefficients (partial derivatives)

9. Scenario exploration

9.1 Generating a system with five species

For the scenario exploration, a similar system will be simulated, using the same number of species, and parameters, we removed the burn-in period, and subsampled the time series (tau=5) to remove strong autocorrelation due to ODE simulation :

```
# --- Food web ODE model ---
def food_web(t, y, params):
    R, C1, C2, P1, P2 = y
```

```
    R, C1, C2, P1, P2 = map(lambda x: max(x, 1e-8), (R, C1, C2, P1,
P2))
    nu1, nu2 = params['nu1'], params['nu2']
    lambda1, lambda2 = params['lambda1'], params['lambda2']
    C1star, C2star = params['C1star'], params['C2star']
    mu1, mu2 = params['mu1'], params['mu2']
    kappa1, kappa2 = params['kappa1'], params['kappa2']
    Rstar, k = params['Rstar'], params['k']

    dRdt = R * (1 - R / k) - mu1 * kappa1 * (C1 * R) / (R + Rstar) -
mu2 * kappa2 * (C2 * R) / (R + Rstar)
    dC1dt = mu1 * kappa1 * (C1 * R) / (R + Rstar) - nu1 * lambda1 * (P1
* C1) / (C1 + C1star) - mu1 * C1
    dC2dt = mu2 * kappa2 * (C2 * R) / (R + Rstar) - nu2 * lambda2 * (P2
* C2) / (C2 + C2star) - mu2 * C2
    dP1dt = nu1 * lambda1 * (P1 * C1) / (C1 + C1star) - nu1 * P1
    dP2dt = nu2 * lambda2 * (P2 * C2) / (C2 + C2star) - nu2 * P2
    return [dRdt, dC1dt, dC2dt, dP1dt, dP2dt]

# --- Parameters ---
params = {
    'nu1': 0.1, 'nu2': 0.07,
    'lambda1': 3.2, 'lambda2': 2.9,
    'C1star': 0.5, 'C2star': 0.5,
    'mu1': 0.15, 'mu2': 0.15,
    'kappa1': 2.5, 'kappa2': 2.0,
    'Rstar': 0.3,
    'k': 1.2
}

# --- Initial condition + burn-in ---
y0 = [1.0, 0.5, 0.8, 0.7, 0.8]
Xi = np.array(y0)
dt = 0.01
for _ in range(int(200 / dt)):
    Xi += dt * np.array(food_web(0, Xi, params))

# --- Integration ---
tmax = 12000
tau = 5
t_eval = np.arange(0, tmax, tau)
sol = solve_ivp(food_web, (0, tmax), Xi, args=(params,), t_eval=t_eval,
rtol=1e-6, atol=1e-9)
```

```
# --- Store result ---
df = pd.DataFrame({
    'R': sol.y[0],
    'C1': sol.y[1],
    'C2': sol.y[2],
    'P1': sol.y[3],
    'P2': sol.y[4],
})

# --- Normalize ---
df_norm = (df - df.mean()) / df.std()
```

9.2 Applying Simplex to determine the response to the perturbation

In this last part, we are trying to answer the following question : “How would the future of C1 change if R was artificially increased or decreased?” by using simplex projection.

As always, we start by looking for the best embedding dimension :

```
# --- Find best E for C1 ---
N = len(df_norm)
lib_end = int(N * 2 / 3)
pred_start = lib_end + 1
results = []

for E in range(3, 11):
    res = Simplex(dataFrame=df_norm, lib=f"1 {lib_end}",
pred=f"{pred_start} {N}",
                  columns="C1", target="C1", E=E, verbose=False)
    obs = res["Observations"].dropna().values
    pred = res["Predictions"].dropna().values
    mae = np.mean(np.abs(obs - pred)) if len(obs) == len(pred) and
len(obs) > 0 else np.nan
    results.append({"E": E, "MAE": mae})

results_df = pd.DataFrame(results)
E_C1 = results_df.loc[results_df["MAE"].idxmin(), "E"]
print(f"Best embedding dimension for C1: {E_C1}")
```

The goal is to find the Embedding Dimension with the lowest MAE. In our case, it is E_C1=6.

The time delay Embedded Matrix can then be created with E_C1 columns. And R is added as an exogenous variable: this is a multivariate embedding, but it is focused on C1's past and R_t. R is treated as the potential driver of C1.

```
    # --- Create multivariate embedding: C1 past + R now ---
    embed_C1 = np.column_stack([
        df_norm["C1"].values[i:N - E_C1 + i + 1] for i in range(E_C1)
    ])
    R_trim = df_norm["R"].values[E_C1 - 1:]
    block0 = np.column_stack([embed_C1, R_trim])
    col_names = [f"C1_t_{i}" for i in reversed(range(E_C1))] +
["R_t"]
```

We simulate two scenarios :

```
# --- Scenarios: R up/down ---
block_inc0 = np.column_stack([embed_C1, R_trim + 0.5])
block_dec0 = np.column_stack([embed_C1, R_trim - 0.5])
```

Two counterfactual datasets are created :

- One where R is increased by 0.5 (in normalized units).
- One where R is decreased by 0.5.

The different scenario DataFrames can then be stacked and prepared into prediction blocks, and the original data can be predicted :

```
df_block0 = pd.DataFrame(block0, columns=col_names)
pred_nochange = Simplex(df_block0, lib=f"1 {len(block0)}", pred=f"1
{len(block0)}",
                        columns=col_names, target=col_names[0], E=E_C1,
verbose=False)

# --- Scenario predictions ---
n0 = len(block0)
block_inc_all = np.vstack([block0, block_inc0])
block_dec_all = np.vstack([block0, block_dec0])
df_block_inc = pd.DataFrame(block_inc_all, columns=col_names)
df_block_dec = pd.DataFrame(block_dec_all, columns=col_names)

step = 100
predictions_inc, predictions_dec = [], []
```

After predicting with the original R, the same procedure is done with the decreased and the increased R :

```
for start_pred in range(n0 + 1, n0 + len(block_inc0) + 1, step):
    end_pred = min(start_pred + step - 1, n0 + len(block_inc0))
```

```
    res_inc = Simplex(df_block_inc, lib=f"1 {n0}", pred=f"{start_pred}
{end_pred}",
                      columns=col_names, target=col_names[0], E=E_C1,
verbose=False)
    predictions_inc.append(res_inc["Predictions"].dropna())

    res_dec = Simplex(df_block_dec, lib=f"1 {n0}", pred=f"{start_pred}
{end_pred}",
                      columns=col_names, target=col_names[0], E=E_C1,
verbose=False)
    predictions_dec.append(res_dec["Predictions"].dropna())

pred_increase = pd.concat(predictions_inc)
pred_decrease = pd.concat(predictions_dec)

# --- Plot ---
trange = range(50)
plt.figure(figsize=(10, 6))
plt.plot(pred_nochange.iloc[trange]["Observations"].values,
label="Observed", color="black", linewidth=2)
plt.scatter(trange, pred_nochange.iloc[trange]["Predictions"].values,
label="Predicted", color="black", marker='o')
plt.scatter(trange, pred_increase.values[:len(trange)], label="R
Increased", color="red", marker='^')
plt.scatter(trange, pred_decrease.values[:len(trange)], label="R
Decreased", color="blue", marker='v')
plt.xlabel("Time")
plt.ylabel("Normalized C1")
plt.title("Scenario Exploration on C1 (ODE, tau=5)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

And then plot the results :

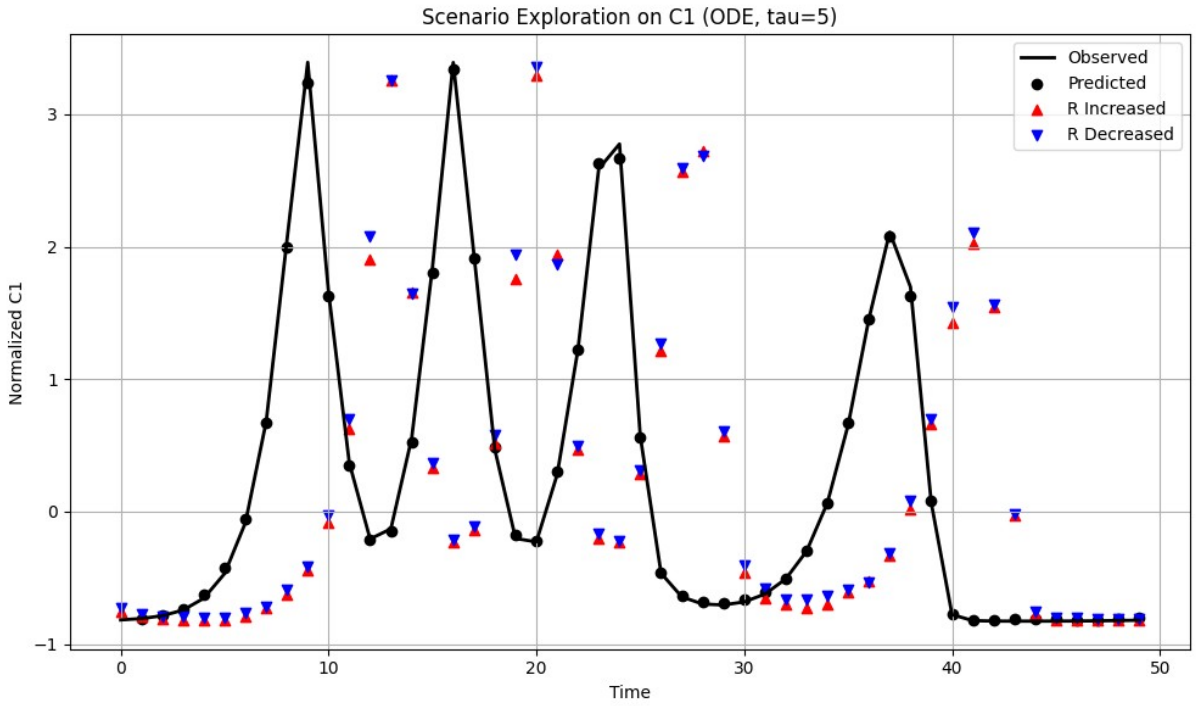


Figure 14 : Scenario Exploration, C1 Observed values vs Predicted values vs Predicted values with R increased/decreased

Conclusion

• Simplex Projection is the foundational forecasting method, using delay embeddings and nearest neighbors to predict future values.

• S-map analysis extends Simplex with locally weighted linear models, detecting nonlinearity and the degree of state-dependence.

• CCM (Convergent Cross Mapping) tests for causality by verifying if historical states of variable X can reconstruct variable Y .

• Multiview Embedding increases robustness by exploring and averaging over multiple embedding strategies. It mitigates the risk of choosing suboptimal coordinates.

• Scenario Exploration allows probing system responses to hypothetical changes, which is valuable for sensitivity analysis and decision support.

Together, these methods form a versatile toolkit for modeling complex, nonlinear systems from time series data, offering complementary insights into predictability, causality, and robustness.