Homework 2

Aldo Tapia

Note: All the code was executed in my local machine instead of using codespaces for this task.

Problem 1: Shuffled complex evolution (SCE) optimization (25 points)

Read the original paper by Duan et al. (1992) on the Shuffled Complex Evolution (SCE) optimization algorithm (https://doi.org/10.1007/BF00939380). Summarize the main steps of the SCE algorithm in your own words. Additionally discuss the problems that the SCE algorithm is designed to solve, and how it compares to other optimization algorithms (e.g., gradient descent, genetic algorithms).

Answer:

SCE mixes different optimization strategies to find the global minimum of the objective function, these approaches are a combination between random and deterministic search, clustering, systematic evolution of the points in direction of the global minimum, and shuffling of the points to avoid local minima. The steps of the SCE algorithm are as follows:

Step 0 - Initialization: the population is initialized by computing the sample size ($s = p \times m$) and generating s random points in the parameter space. p is the number of complexes and m is the number of points in each complex.

Step 1 - Generate samples: s points are generated in the parameter space

Step 2 - Rank points: the s points are sorted based on the increasing function value generating a D array.

Step 3 - Partition into complexes: the s points are partitioned into p complexes, each complex contains m points.

Step 4 - Evolve each complex: each complex is evolved independently using a competitive evolution strategy.

Step 5 - Shuffle complexes: the evolved complexes are shuffled together and sorted to form a new D array.

Step 6 - Check convergence: if the convergence criterion is met, the algorithm stops. Otherwise, it goes back to Step 3.

In the competitive complex evolution algorithm, each complex is evolved independently, and the worse performing points in each complex are replaced by new candidate points generated through a simplex-like reflection, contraction, or expansion step based on the better-performing members of the complex (like the Nelder-Mead simplex method). This allows for local search within each complex, while the shuffling step allows for global mixing of information across complexes, helping to avoid premature convergence to local minima.

The problems that SCE is designed to solve are the multiple local minima, when the shape of the objective function surface is non-smooth or is discontinuous, parameter with different sensibilities, and when the surface near the true solution is nonconvex. All these problems are common in hydrological model calibration, which is the main application of SCE.

For comparing SCE with other optimization algorithms, we need to analyze the following key points:

Gradient: SCE does not require to compute gradients, same with other optimization algorithms like Genetic Algorithms, Nelder-Mead, and Particle Swarm Optimization. In contrast, gradient-based methods like gradient descent require the objective function to be differentiable and can get stuck in local minima.

Parallelization: SCE can be parallelized by evolving each complex independently, which can speed up the optimization process. Genetic algorithms and particle swarm optimization can also be parallelized, while gradient descent is typically not parallelizable.

Convergence speed:: depending the surface characteristics, gradient descent can converge faster than SCE when the objective function is smooth and convex, but it can get stuck in local minima. SCE is designed to avoid local minima and can perform better on non-smooth or non-convex surfaces, but it may require more function evaluations to converge. Same applies to genetic algorithms and particle swarm optimization, which can also be slower than gradient descent but are more robust to local minima.

Global optimization: SCE has a great ability to find the global minimum of the objective function (that's the main point of the algorithm), while gradient descent can get stuck in local minima. Genetic algorithms and particle swarm optimization are also designed for global optimization, but they may require more function evaluations than SCE to find the global minimum.

Problem 2: Regression for streamflow prediction (25 points)

In `/data/LeafRiverDaily.csv` you have been given daily temperature, precipitation, and streamflow data for the Leaf River in Mississippi. Your task is to build a regression model to predict daily streamflow based on the temperature and precipitation data. Perform the following steps:

1. Load the data into a pandas DataFrame and perform any necessary preprocessing (e.g., handling missing values, feature scaling). Consider a sample as a 90 day history of temperature and precipitation data, and the target variable as the streamflow on the 91st day.
2. Split the data into training and testing sets (e.g., 80% training, 20% testing) to evaluate the performance of your regression model.
3. Train a linear regression model on the training data and evaluate its performance on the testing data using appropriate metrics (e.g., R-squared, mean absolute error). Use a sample size where inputs are the previous 90 days of temperature and precipitation data, and the target variable is the streamflow on the 91st day. You may need to reshape your data accordingly to fit this format.

Answer:

Step 1

Data loading, summary statistics, and checking for missing values:

```
path = 'assets/LeafRiverDaily.csv'
df = pd.read_csv(path)
df.describe()
df.isna().sum()
```

	Precipitation	Temperature	Streamflow
count	10960	10960	10960
mean	3.869	2.912	1.338
std	10.042	1.882	2.831
min	0.000	0.000	0.069
25%	0.000	1.329	0.202
50%.	0.000	2.562	0.445
75%	2.478	4.327	1.210
max	221.519	8.497	64.014

```
Precipitation    0
Temperature      0
Streamflow       0
dtype: int64
```

Since there are no missing values, no need for imputation. Then, I created a function to reshape the data into samples of 90 days of temperature and precipitation as input, and the streamflow on the 91st day as the target variable:

```
def reshape_to_samples(df, sample_size = 90):
    num_samples = len(df) - sample_size - 1
    print(f'num_samples available: {num_samples}')
    forcings = []
    target = []
    precip_list = []

    for i in range(num_samples):
        sample_temp = df.iloc[i:(i+sample_size + 1),:]
        temp = sample_temp['Temperature'].values[:-1]
        precip = sample_temp['Precipitation'].values[:-1]
        streamflow = sample_temp['Streamflow'].values[-1]
        forcings.append(np.concatenate([temp, precip]))
        target.append(streamflow)
        precip_list.append(sample_temp['Precipitation'].values[-1])
    return np.array(forcings), np.array(target),
           num_samples, np.array(precip_list)
```

Step 2

Using the function above, I reshaped the data into samples and split it into training and testing sets, where the training set contains the first 80% of the samples and the testing set contains the remaining 20% of the samples:

```
X, y, num_samples, precip = reshape_to_samples(df)
train_len = np.floor(num_samples*0.8).astype(int)
X_train, y_train = X[:train_len], y[:train_len]
X_test, y_test = X[train_len:], y[train_len:]
```

Step 3

For fitting the linear regression model, I used the `statsmodels` library to fit an Ordinary Least Squares (OLS) regression model to the training data:

```
X_train = sm.add_constant(X_train)
model = sm.OLS(y_train, X_train).fit()
X_test = sm.add_constant(X_test)
y_modeled = model.predict(X_test)
```

The performance of the model was evaluated using R-squared and Mean Absolute Error (MAE):

R^2: 0.730.
MAE: 0.931

Analyzing the p-values of the coefficients using Wald test, I found that some days of temperature are statistically significant ($p < 0.05$) for predicting streamflow (6 days), while many days of precipitation (close to the target day) are statistically significant for predicting streamflow (33 days). This suggests that precipitation has a stronger influence on streamflow than temperature in this dataset, which is consistent with our understanding of hydrological processes. The days with $p > 0.05$ for temperature and precipitation are as follows:

Temperature days with $p < 0.05$: 68, 69, 87, 8,8 89, 90.
Precipitation days with $p < 0.05$: 26, 48, 49, 50, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 63, 66, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90

The hydrograph of the observed streamflow vs. the predicted streamflow from the linear regression model is shown below:

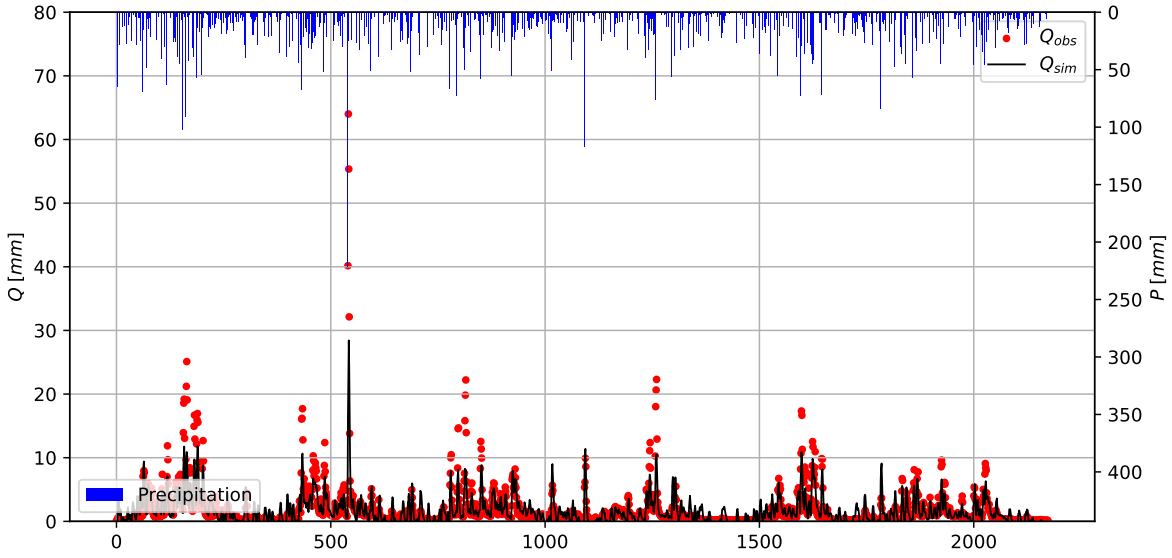
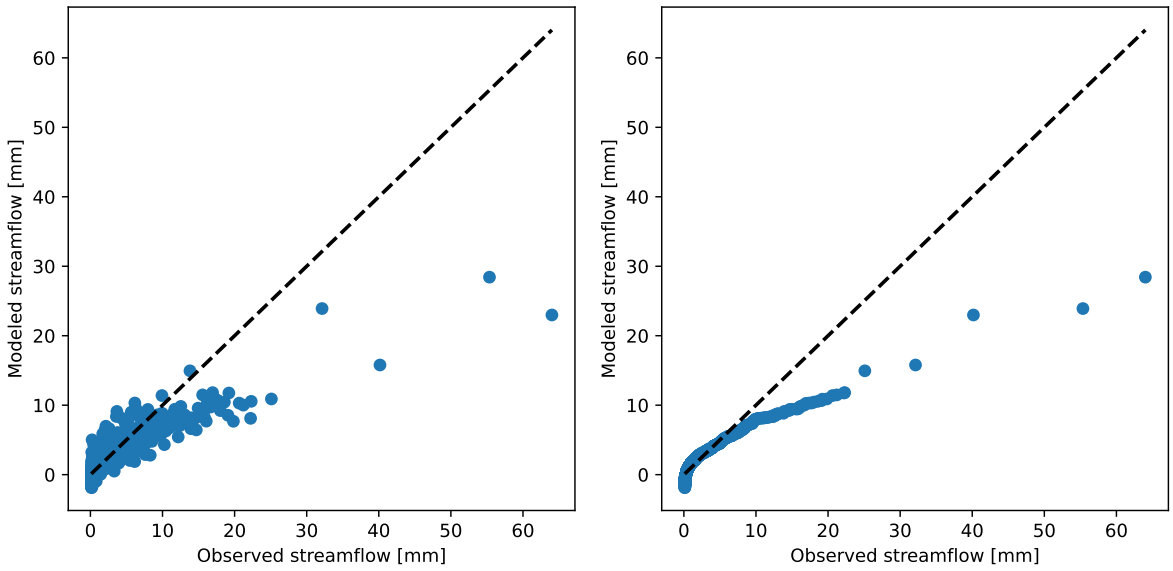


Figure 1: Stream discharge hydrograph of observed streamflow vs. predicted streamflow

The hydrograph shows that the modeled streamflow is not able to capture the peaks of the observed streamflow, and there is no recession curve in the modeled streamflow, exposing the non-account of storage and routing. This plots does not show something important, which is days with negative streamflow. The following plots show the scatter plot of observed vs. predicted streamflow, and the Q-Q plot of observed vs. predicted streamflow for the testing dataset:



(a) Scatter plot of observed vs. predicted streamflow (b) Q-Q plot of observed vs. predicted streamflow

Figure 2: Comparison of testing dataset

The above plots shows the relatively good performance of the linear regression model for predicting streamflow for low streamflow values, while for high streamflow values, the model underestimates the streamflow. Also, we can see that there are some negative streamflow values in the modeled data.

Problem 3: Calibration of a simple hydrological model (25 points)

For this problem you will implement and calibrate a nonlinear hydrological model using the SCE optimization algorithm. The model is a conceptual bucket model expressed as a state-space ODE. The single state variable $S(t)$ represents the water storage in the catchment, and its evolution is governed by:

$$\frac{dS}{dt} = P(t) - a \cdot \max(T(t), 0) - b \cdot S(t)^c$$

Where:
- $S(t)$ is the catchment storage at time t (the state variable).
- $P(t)$ is the precipitation at time t (a forcing input).
- $T(t)$ is the temperature at time t (a forcing input).
- $a \cdot \max(T(t), 0)$ represents evapotranspiration, assumed proportional to temperature when

temperature is positive.
- $b \cdot S(t)^c$ is a nonlinear storage-discharge relationship that produces streamflow.

The predicted streamflow is then given by the output equation:

$$Q(t) = b \cdot S(t)^c$$

The parameters to calibrate are:
- a — evapotranspiration coefficient.
- b — discharge coefficient.
- c — nonlinearity exponent of the storage-discharge relationship.

Perform the following steps: 1. Implement the model using scipy's `solve_ivp` (or `odeint`) function. You will need to define a function that computes dS/dt given the current state S and the forcing inputs $P(t)$ and $T(t)$. Since the forcing data is daily, you will need to interpolate P and T to evaluate them at arbitrary times requested by the ODE solver. You can use `scipy.interpolate.interp1d` for this purpose. 2. Define an objective function that computes the mean squared error between the observed streamflow and the model-predicted streamflow $Q(t)$ for a given set of parameters. 3. Use the SCE optimization algorithm to find the optimal parameters a, b, c, and d that minimize the objective function using the SCE-UA algorithm as implemented with the `spotpy` library.

Note: you should use the spotpy documentation to help you implement the calibration: https://spotpy.readthedocs.io/en/latest/Calibration_with_SCE-UA/

Answer:

Step 1

The implementation of the model was done using `scipy.integrate.solve_ivp` to solve the ODE, and I did a manual interpolation of precipitation and temperature to evaluate them at arbitrary times requested by the ODE solver, with a small constraint if the step size is too small. The function `dSdt_decorated` computes the derivative of storage with respect to time, given the current storage and the forcing inputs.

Finally, to speed up the computation of the derivative, I used `numba` to compile the function `dSdt_decorated` with the `@njit` decorator, which allows for faster execution. Also, I constrained the derivative of storage to avoid negative storage values.

```
@njit
def dSdt_decorated(t, S, P, T, a, b, c):
    n = len(T)
    idx = int(np.floor(t))
    if idx >= n:
        idx = n - 1

    if np.abs(idx - t) > 0.000001:
        Tt = T[idx]
        Pt = P[idx]
    else:
        Tt = T[idx] + (T[idx+1] - T[idx]) * (t - idx)
        Pt = P[idx] + (P[idx+1] - P[idx]) * (t - idx)

    E = a*(Tt if Tt > 0.0 else 0.0)
    Q = b*(S**c) if S > 0.0 else 0.0
    dS =  Pt - E - Q

    # constraint to avoid negative storage
    if S <= 0.0 and dS < 0.0:
        dS = 0.0
    return dS
```

The evaluation of the streamflow based on the storage, is done outside of the ODE solver, since it is not needed for the integration of the storage. The function `streamflow_from_storage` computes the streamflow given the storage and the parameters b and c.

```
def streamflow_from_storage(S, b, c):
    S = np.asarray(S, dtype=float)
    Q = np.zeros_like(S)
    m = S > 0.0
    Q[m] = b * (S[m] ** c)
    return Q
```

The following code snippet shows how to run the model with a given set of parameters and evaluate the predicted streamflow using the ODE solver. I also calibrated the initial storage, since the function is constrained to avoid negative storage, we cannot use 0 as initial storage or the model will try to decrease the storage and it will be constrained to 0:

```
T = df_train['Temperature'].values
P = df_train['Precipitation'].values
Q = df_train['Streamflow'].values
```

```
t_span = [0, len(T)-1]
S0 = [0]
t_eval = np.arange(len(T))

sol = solve_ivp(fun = dSdt_numba(P, T, a, b, c),
                t_span = t_span, y0 = [x[3]],
                t_eval=t_eval,
                method="RK23",
                max_step=1.0,
                rtol=1e-4, atol=1e-6)
S = sol.y[0]
streamflow_modeled = streamflow_from_storage(S, b=b, c=c])
```

Step 2

Since MSE is the metric that we want to minimize, I defined the objective function as the mean squared error between the observed streamflow and the model-predicted streamflow:

```
def mse(y_true, y_pred):
    mse = np.mean((y_true - y_pred) ** 2)
    return mse
```

Step 3

To calibrate the model using the SCE optimization algorithm, I defined a class `Spotpy_setup` that includes the parameters to calibrate, the simulation function that runs the model and returns the predicted streamflow, the evaluation function that returns the observed streamflow, and the objective function that computes the MSE between the observed and predicted streamflow.

```
class Spotpy_setup(object):
    a = Uniform(low=0, high=2, optguess=0.1)
    b = Uniform(low=0, high=2, optguess=0.5)
    c = Uniform(low=0, high=3, optguess=0.5)
    initial_storage = Uniform(low=0, high=200, optguess=50)

    def __init__(self,
                 forcings = pd.DataFrame,
                 target = pd.Series):
```

```
        self.temperature = forcings['Temperature'].values
        self.precipitation = forcings['Precipitation'].values
        self.target = target.values

    def simulation(self, x):
        sol = solve_ivp(fun = dSdt_numba(self.precipitation, self.temperature,
                                         x[0], x[1], x[2]),
                        t_span = t_span, y0 = [x[3]], t_eval=t_eval,
                        method="RK23",
                        max_step=1.0, rtol=1e-4, atol=1e-6)
        S = sol.y[0]
        streamflow_modeled = streamflow_from_storage(S, b=x[1], c=x[2])
        return streamflow_modeled

    def evaluation(self):
        return self.target

    def objectivefunction(self, simulation, evaluation):
        return mse(evaluation,simulation)


sp = Spotpy_setup(forcings=df_train[['Temperature', 'Precipitation']],
                  target=df_train['Streamflow'])

sampler = spotpy.algorithms.sceua(sp, dbname='SCEUA_dsdt_rk23',
                                  dbformat='csv', save_sim = False)

max_model_runs = 10000

sampler.sample(max_model_runs)

results = spotpy.analyser.load_csv_results('SCEUA_dsdt_rk23')
```

The following plots show the convergence of the MSE during the optimization process, and the comparison of the observed streamflow with the predicted streamflow from the calibrated model for the testing dataset:

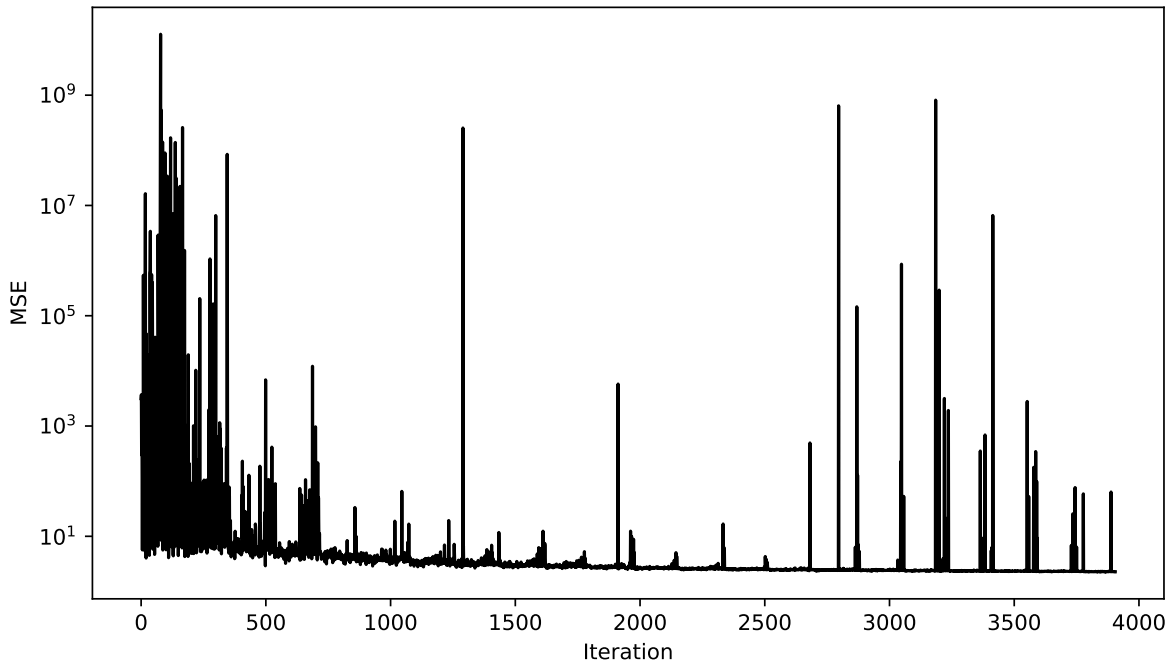


Figure 3: MSE convergence of SCE optimization

Same hydrograph of the observed streamflow vs. the predicted streamflow from the calibrated model:

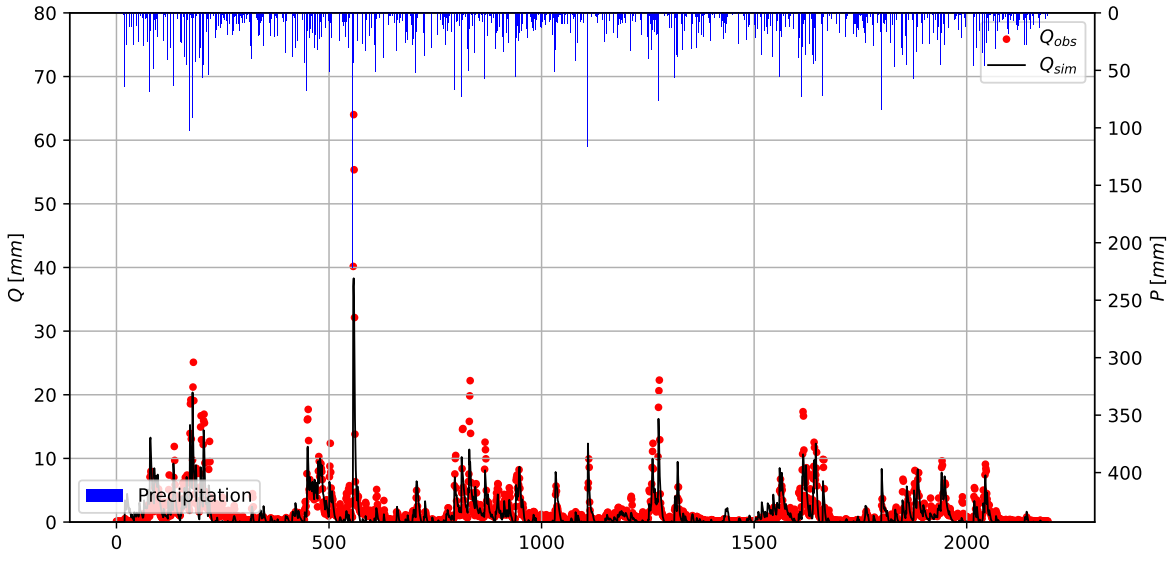
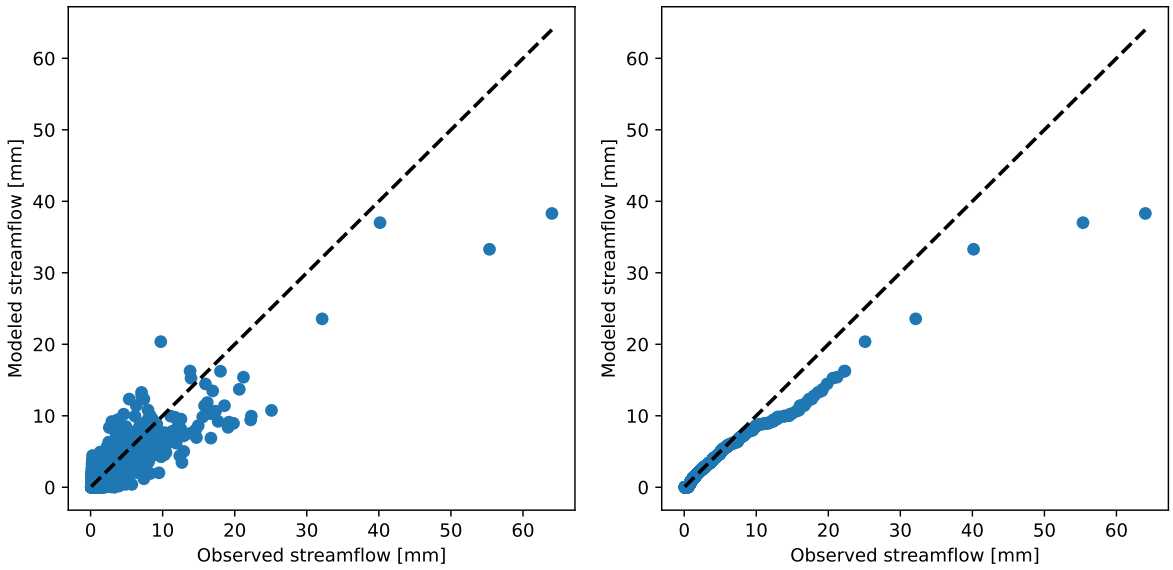


Figure 4: Stream discharge hydrograph of observed streamflow vs. predicted streamflow

We can observe that the calibrated model is able to capture the peaks of the observed streamflow better than the linear regression model, but since this model only has one state variable, it is not able to capture the recession curve of the observed streamflow.



(a) Scatter plot of observed vs. predicted streamflow (b) Q-Q plot of observed vs. predicted streamflow

Figure 5: Comparison of testing dataset

The above plots show that the calibrated model has a similar performance to the linear regression model for low streamflow values, while for high streamflow values, the calibrated model performs relatively better than the linear regression model, although it still underestimates the streamflow.

Problem 4: Comparison of linear regression and SCE optimization (25 points)

Compare the performance of the linear regression model and the SCE optimization algorithm for predicting streamflow from the previous problems based on the same dataset. Perform the following steps:

1. Calculate the Overall NSE (Nash-Sutcliffe Efficiency) for both models on the testing data. The NSE is defined as:

$$NSE = 1 - \frac{\sum_{i=1}^{n}(Q_{obs,i} - Q_{pred,i})^2}{\sum_{i=1}^{n}(Q_{obs,i} - \bar{Q}_{obs})^2}$$

Where $Q_{obs,i}$ is the observed streamflow, $Q_{pred,i}$ is the predicted streamflow, and $\bar{Q}_{obs}$ is the mean of the observed streamflow.

2. Create a figure that compares the observed streamflow with the predictions from both models over time. Include appropriate labels, legends, and titles to clearly distinguish between the observed data and the predictions from the linear regression and SCE models.

3. Discuss the results: Which model performs better in terms of NSE? What are the strengths and weaknesses of each model? How might you improve the performance of each model?

Answer:

Step 1

The following functions compute the Kling-Gupta Efficiency (KGE) and the Nash-Sutcliffe Efficiency (NSE). Although NSE is required, I computed the KGE to interpret the components of the KGE.

```
def kge(y_true, y_pred):
  '''Kling-Gupta Efficiency (KGE)'''
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    y_pred = y_pred[~np.isnan(y_true)]
    y_true = y_true[~np.isnan(y_true)]

    y_true_mean = np.mean(y_true)
    y_pred_mean = np.mean(y_pred)

    y_true_std = np.std(y_true)
    y_pred_std = np.std(y_pred)

    xy = np.sum((y_true-y_true_mean)*(y_pred-y_pred_mean))
    corr = xy/((len(y_true))*y_true_std*y_pred_std)

    alpha = y_pred_std/y_true_std
    beta = y_pred_mean/y_true_mean
    kge = 1 - np.sqrt((corr-1)**2 + (alpha-1)**2 + (beta-1)**2)
```

```
    return kge, corr, alpha, beta

def nse(y_true, y_pred):
    '''Nash-Sutcliffe Efficiency (NSE)'''
    numerator = np.sum((y_true - y_pred) ** 2)
    denominator = np.sum((y_true - np.mean(y_true)) ** 2)
    nse =  1 - numerator / denominator
    return nse
```

These are the results for the NSE and KGE for both models:

Model	NSE	KGE	α_{KGE}	β_{KGE}	r_{KGE}
Liner regression	0.695	0.633	0.855	0.700	0.938
Simple hydrological model	0.562	0.628	0.757	0.788	0.815

In terms of stricly NSE values, the linear regression model performs better than the simple hydrological model. However, the KGE values are relatively similar for both models, with the linear regression model having a slightly higher KGE than the simple hydrological model. Also, we need to remember than the linear regression models negative values of streamflow.

Is important to note that the linear regression model evaluates 90 days of data, while the simple hydrological model evaluates the streamflow based on the storage, which is a state variable that evolves over time, and it is not directly related to the previous 90 days of data, specially since it doesn't include routing or storage in different buckets.

In terms of the components, the linear regression model has α_{KGE} closer to 1 which indicates that the variability of the predicted streamflow is closer to the variability of the observed streamflow for the linear regression model. The β_{KGE} is also closer to 1 for the linear regression model, which in long term, indicates that the bias of the predicted streamflow is lower for the linear regression model compared to the simple hydrological model. The correlation coefficient r_{KGE} is higher for the linear regression model, but since we used a linear regression model, it is expected to have a higher correlation coefficient than the simple hydrological model, which is a nonlinear model.

Step 2

The hydrograph of the observed streamflow vs. the predicted streamflow from both models is shown below:

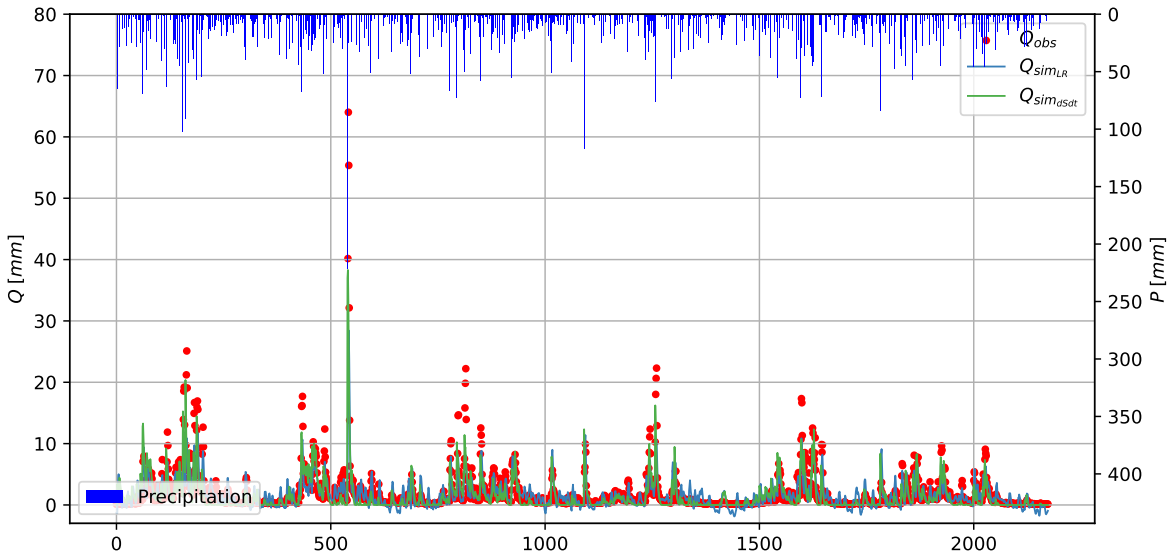


Figure 6: Comparison between Linear Regression and Simple Hydrological Model

The hydrograph summaries the performance of both models and the points expressed in previous paragraphs.

Step 3

Calling back some of the points made, the linear regression includes 90 days of data as input, which allows to capture the dependency of the streamflow on the previous days, but in a linear way, which is not able to capture the nonlinearity of the system. The simple hydrological model includes a state variable that evolves over time, which allows to capture the storage, but not the routings or multiple processes that can happen in the catchment, which is why it is not able to capture the recession curve of the observed streamflow.

To improve the performance for the linear regression, we may include non-linear terms, such polynomial terms. For the simple hydrological model, we can include more state variables to capture the storage in different buckets, and include routing to capture the delay between the precipitation and the streamflow. Also, we can include more processes such as interflow, evapotranspiration, and groundwater flow to capture the complexity of the system.

A test for the latter was implemented and it's presented in the Appendix, where I implemented a hymod-like model with 4 state variables, which includes more processes and storage in different buckets.

Appendix

Hymod-like model using the same approach of this HW:

For trying a more complex hydrological model, I implemented a hymod-like model with 4 states and more parameters included (the evapotranspiration I left it as is stated for the problem, modifying the number of buckets and the equations for the storage-discharge relationship).

The following equations govern the model:

$$\frac{dS_1}{dt} = P(t,k) - Pe(t) - E(t) - Rg(t) - IF(t)$$

$$\frac{dS_2}{dt} = Rg(t) - BF(t)$$

$$\frac{dS_{3a}}{dt} = Pe(t,k) + IF(t) - Y(t)$$

$$\frac{dS_{3b}}{dt} = Y(t) - SF(t)$$

Where:
- S_1 is the soil moisture storage.
- S_2 is the deep groundwater storage.
- S_{3a} is 1st bucket for route water.
- S_{3b} is 2nd bucket for route water.
- $P(t,k)$ is the precipitation at time t with a transmissivity parameter k.
- $T(t)$ is the temperature at time t (a forcing input).
- $Pe(t,k)$ is precipitation excess, which accounts when the first bucket is saturated and the ponding when the transmissivity is considered.
- $E(t)$ is evapotranspiration, assumed proportional to temperature when temperature is positive.
- $Rg(t)$ is the recharge to groundwater.
- $IF(f)$ is the interflow.
- $Y(t)$ is the flow from one routing bucket to the next.
- $BF(t)$ is the baseflow.
- $SF(t)$ is the surface flow.

With this, streamflow is defined as:

$$Q(t) = SF(t) + BF(t)$$

Some new components are parametized as $\theta_x \cdot S_i$ where θ_x is a parameter to calibrate for the x variable and S_i is the storage of the corresponding bucket i. The different variables are:

$$Pe(t,k) = \max(0, P(t) - \max(0, S_{1_{\max}} - S_1)) + \max(0, P(t) - k)$$

Where $S_{1_{\max}}$ is the maximum storage of the soil bucket. Here Pe(t, k) accounts for the precipitation excess, which is the amount of precipitation that cannot be stored in the soil bucket, and the ponding when the transmissivity is considered.

Also, the recharge to groundwater is defined as:

$$Rg = u \left(\frac{S_1}{S_{1_{\max}}} \right)^{\alpha} \cdot S_1$$

Where u is a parameter to calibrate that controls the maximum recharge, and α is a parameter to calibrate that controls the nonlinearity of the recharge. u is defined as $b(1-c)$ where b and c are the parameters that control the interflow and the level of recharge.

Since the recharge to groundwater is non-linear, the baseflow is also non-linear and is defined as:

$$BF = e \cdot S_2^{\beta}$$

Where e is a parameter to calibrate that controls the maximum baseflow, and β is a parameter to calibrate that controls the nonlinearity of the baseflow.

The new model is summarized in the following figure:

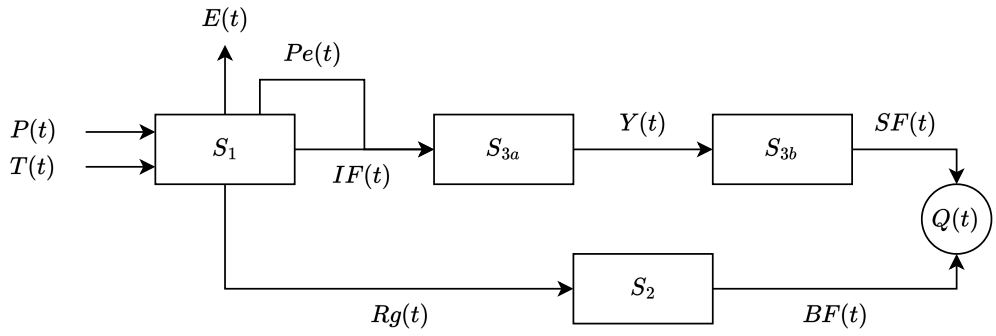


Figure 7: Hymod-like model structure

The model was calibrated as the same way as the simple hydrological model, using the SCE optimization algorithm with the `spotpy` library, the following plot shows the convergence of the MSE during the optimization process:

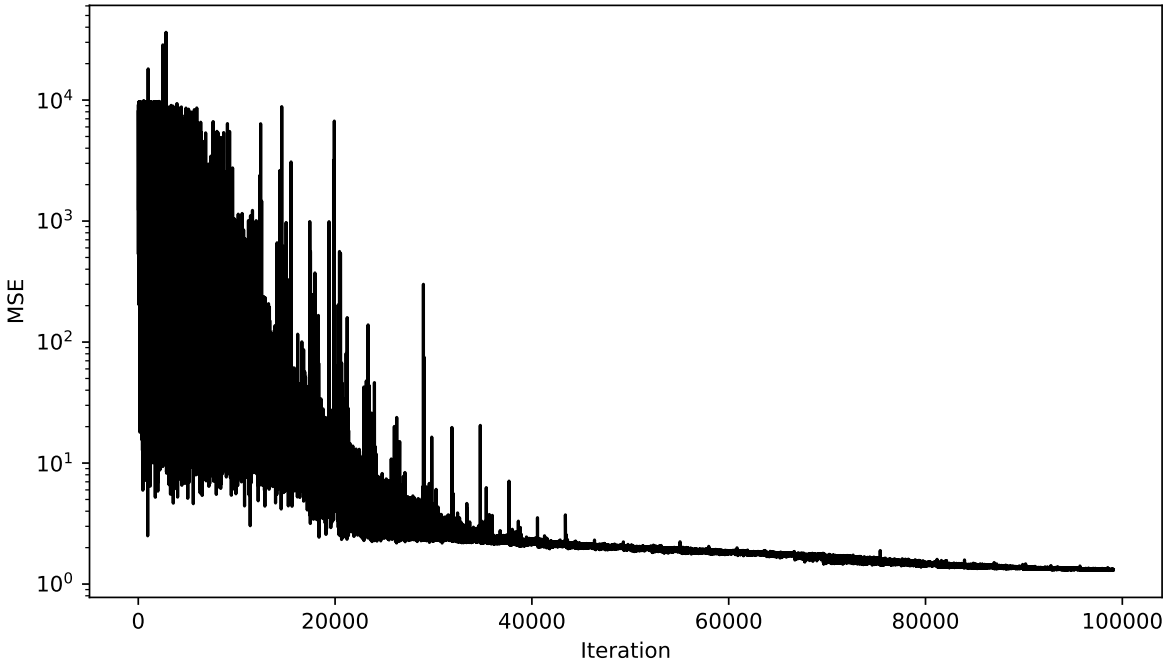


Figure 8: MSE convergence of SCE optimization for the hymod-like model

The parameters obtained from the calibration are the following:

- `a`: 1.14623.
- `b`: 0.00366244.
- `c`: 0.000201689.
- `d`: 249.075.
- `e`: 0.0164868.
- `f`: 0.603221.
- `g`: 0.595915.

- `alpha`: 1.99607.
- `beta`: 1.08919.
- `k`: 0.165918.
- `initial_S1`: 56.3323.
- `initial_S2`: 41.0105.
- `initial_S3a`: 2.98454.
- `initial_S3b`: 0.418529.

The KGE obtained by this method was 0.891 ($\alpha_{KGE} = 0.980$, $\beta_{KGE} = 1.04$, $r_{KGE} = 0.905$), which is a significant improvement compared to the previous models, and the hydrograph of the observed streamflow vs. the predicted streamflow from this model is shown below:

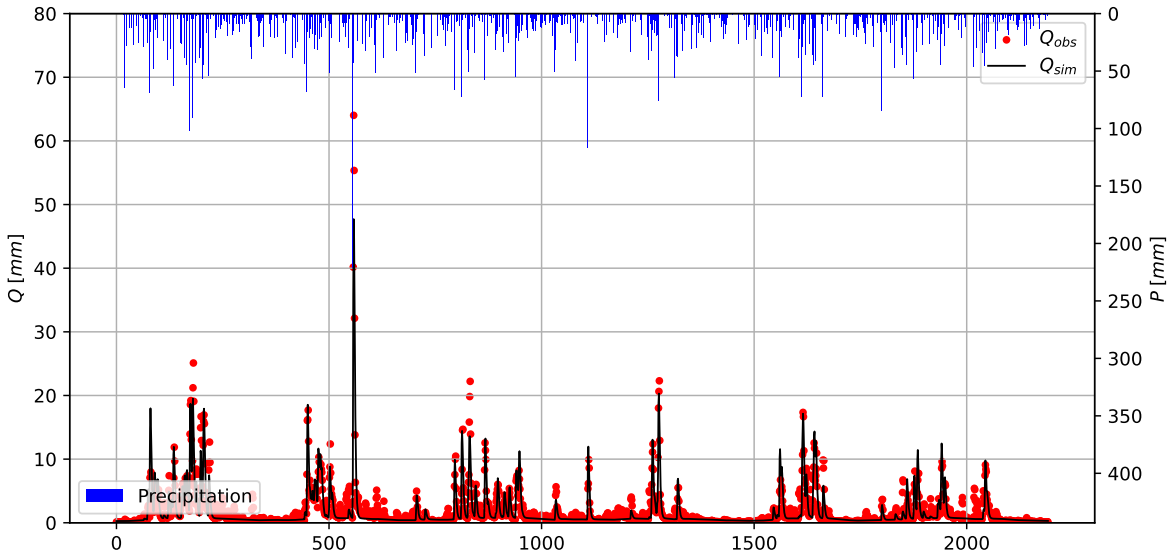


Figure 9: Hydrograph of the proposed model

The hydrograph shows that the proposed model is able to capture the peaks of the observed streamflow better than the previous models, but the random noise in the observed streamflow is not captured by the model, which mean some processes are not correctly captured by the model, and still there is room for improvement. Something interesting from this model, is the dynamics of the storage states:

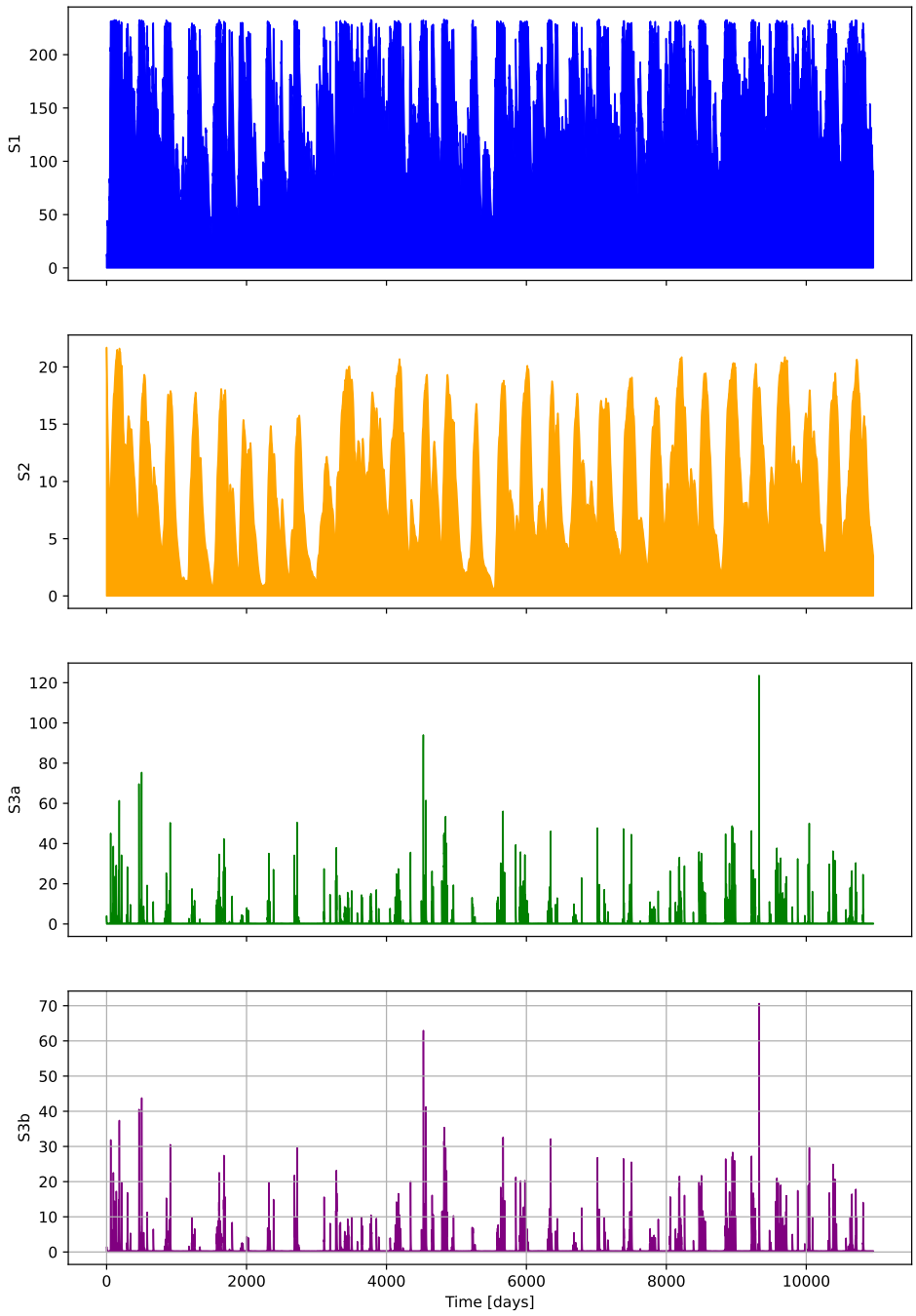


Figure 10: Storage dynamics of the proposed model

The state variables showen in the plot include both the training and testing dataset, while the hydrograph only includes the testing dataset (same with the computed metrics). The above

plot shows that the first bucket is highly dynamic, and the channel routing ($S3a$ and $S3b$) exhibit a higher storage dynamics than the groundwater storage ($S2$). Hence, for this system is more important to capture the dynamics of the soil moisture and the channel routing, than the groundwater storage.

The code used is the following:

```
import pandas as pd
import numpy as np
import spotpy
from spotpy.parameter import Uniform
from scipy.integrate import solve_ivp
from numba import njit
from spotpy.objectivefunctions import rmse, mae, mse

path = 'assets/LeafRiverDaily.csv'

df = pd.read_csv(path)

train_len = int(np.floor(df.shape[0]*0.8))

df_train = df.iloc[:train_len,:]
df_test = df.iloc[train_len:,:]


T = df_train['Temperature'].values
P = df_train['Precipitation'].values
Q = df_train['Streamflow'].values

t_span = [0, len(T)-1]
S0 = [0]
t_eval = np.arange(len(T))

@njit
def dSdt_decorated(t, S1, S2, S3a, S3b,
                   P, T, a, b, c, d, e,
                   f, g, alpha, beta, k):

    n = T.shape[0]

    idx = int(np.floor(t))
    if idx < 0:
        idx = 0
```

```
    if idx >= n - 1:
        idx = n - 1

    if (abs(t - idx) < 1e-9) or (idx == n - 1):
        Tt = T[idx]
        Pt = P[idx]
    else:
        frac = t - idx
        Tt = T[idx] + (T[idx + 1] - T[idx]) * frac
        Pt = P[idx] + (P[idx + 1] - P[idx]) * frac

    S1p = max(0.0, S1)
    S2p = max(0.0, S2)
    S3ap = max(0.0, S3a)
    S3bp = max(0.0, S3b)

    # First bucket
    max_S = d # max storage
    u = b*(1 - c) # to avoid extracting more than 1x S1
    E = a * (Tt if Tt > 0.0 else 0.0) * (S1p/max_S)
    R = u * ((S1p/max_S)**alpha) * S1p
    IF = c * S1p
    Ptk = Pt - k # infiltration factor
    Pek = max(0.0, Pt - Ptk) # effective precipitation

    # precipitation excess
    deficit = max_S - S1p
    Pe = max(0.0, Pt - max(0.0, deficit)) + Pek

    # Second bucket
    BF = e * (S2p**beta)

    # Channel routing
    Y  = f * S3ap
    SF = g * S3bp

    # State derivatives
    dS1 = Ptk - Pe - E - R - IF
    dS2 = R - BF
    dS3a = Pe + IF - Y
    dS3b = Y - SF
```

```
    # Prevent negative storage from decreasing further
    if S1 <= 0.0 and dS1 < 0.0:
        dS1 = 0.0
    if S2 <= 0.0 and dS2 < 0.0:
        dS2 = 0.0
    if S3a <= 0.0 and dS3a < 0.0:
        dS3a = 0.0
    if S3b <= 0.0 and dS3b < 0.0:
        dS3b = 0.0

    # Return an array (solve_ivp likes 1D array-like)
    return np.array([dS1, dS2, dS3a, dS3b], dtype=np.float64)


def dSdt_numba(P, T, a, b, c, d, e, f, g, alpha, beta, k):
    # Make sure they are numpy arrays with float dtype (helps Numba)
    P = np.asarray(P, dtype=np.float64)
    T = np.asarray(T, dtype=np.float64)

    def rhs(t, y):
        return dSdt_decorated(t, y[0], y[1], y[2], y[3],
                              P, T, a, b, c, d, e, f, g,
                              alpha, beta, k)

    return rhs

def streamflow_from_storage(S2, S3b, e, g):
    return e * S2 + g * S3b

class Spotpy_setup(object):
    a = Uniform(low=0, high=2.5, optguess=1.99)
    b = Uniform(low=0, high=1, optguess=0.17)
    c = Uniform(low=0, high=1, optguess=0.09)
    d = Uniform(low=10, high=250, optguess=150)
    e = Uniform(low=0, high=0.1, optguess=0.001)
    f = Uniform(low=0, high=1, optguess=0.05)
    g = Uniform(low=0, high=1, optguess=0.0)
    alpha = Uniform(low=0, high=2, optguess=0.5)
    beta = Uniform(low=0, high=2, optguess=0.5)
    k = Uniform(low=0, high=100, optguess=50)
    initial_S1 = Uniform(low=0, high=80, optguess=1)
    initial_S2 = Uniform(low=0, high=80, optguess=1)
```

```
    initial_S3a = Uniform(low=0, high=80, optguess=1)
    initial_S3b = Uniform(low=0, high=80, optguess=1)

    def __init__(self,
                 forcings = pd.DataFrame,
                 target = pd.Series):

        self.temperature = forcings['Temperature'].values
        self.precipitation = forcings['Precipitation'].values
        self.target = target.values

    def simulation(self, x):
        sol = solve_ivp(fun = dSdt_numba(self.precipitation, self.temperature,
                                         x[0], x[1], x[2], x[3], x[4],
                                         x[5], x[6], x[7], x[8], x[9]),
                        t_span = t_span, y0 = [x[10], x[11], x[12], x[13]],
                        t_eval=t_eval, method="RK23",
                        max_step=1.0, rtol=1e-4, atol=1e-6)
        S1, S2, S3a, S3b = sol.y
        streamflow_modeled = streamflow_from_storage(S2, S3b, x[4], x[6])
        return streamflow_modeled

    def evaluation(self):
        return self.target

    def objectivefunction(self, simulation, evaluation):
        return mse(evaluation,simulation)


sp = Spotpy_setup(forcings=df_train[['Temperature', 'Precipitation']],
                  target=df_train['Streamflow'])

sampler = spotpy.algorithms.sceua(sp, dbname='new_SCEUA_dsdt_rk23_e',
                                  dbformat='csv', save_sim = False,
                                  parallel='mpi')

max_model_runs = 200000

sampler.sample(max_model_runs, ngs=100)
```

And to run the code: `mpirun -c 100 python spotpy_multiple.py`