Bias-Variance Tradeoff

1. Importing necessary libraries

First, we need to import the libraries required for performing numerical computations, implementing the linear regression model, feature preprocessing (such as polynomial transformation), testing, and visualisation.

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

2. Preparing the dataset

Next, we define and generate the data. For this, we first define a function `true_function(x)` that represents the actual underlying relationship we want to model (in this case, we are taking it to be `sin(x)`). Then, we create 100 data points (`X`) with values ranging from 0 to 10, and calculate the true values using `true_function`. We also add random noise (noise) to simulate real-world data with uncertainty. Finally, the actual data points (`y`) are created by combining the true values with noise.

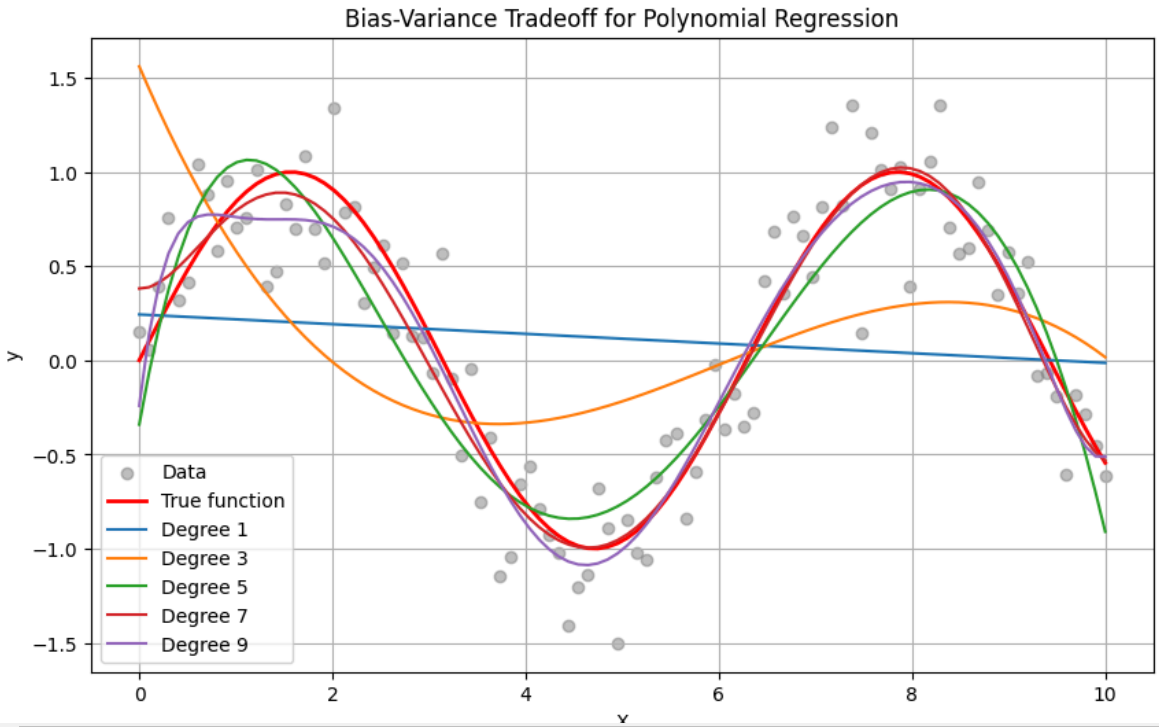
```
np.random.seed(42)

def true_function(x):
    return np.sin(x)

X = np.linspace(0, 10, 100).reshape(-1, 1)
y_true = true_function(X)
noise = np.random.normal(0, 0.3, size=y_true.shape)
y = y_true + noise
```

3. Building and training the model

Now, we split the generated data into training and testing sets, and train the regression model by iterating through different complexities (polynomial degrees). We also plot the predicted curve for each specific degree on the main visualisation.

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

degrees = [1, 3, 5, 7, 9]
train_errors = []
test_errors = []

# Creating a plot to visualize the results
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='gray', label='Data', alpha=0.5)
plt.plot(X, y_true, color='red', label='True function', lw=2)

for deg in degrees:
    # Transforming the input features to polynomial features
    poly = PolynomialFeatures(deg)
    X_poly_train = poly.fit_transform(X_train)
    X_poly_test = poly.transform(X_test)

    # Fitting the linear regression model
    model = LinearRegression()
    model.fit(X_poly_train, y_train)

    # Predicting on training and test data
    y_train_pred = model.predict(X_poly_train)
    y_test_pred = model.predict(X_poly_test)

    # Calculating the MSE for both training and testing sets
    train_error = np.mean((y_train - y_train_pred) ** 2)
    test_error = np.mean((y_test - y_test_pred) ** 2)

    train_errors.append(train_error)
    test_errors.append(test_error)

    # Plotting the model's predictions
    X_range = np.linspace(0, 10, 100).reshape(-1, 1)
    X_range_poly = poly.transform(X_range)
    y_range_pred = model.predict(X_range_poly)
    plt.plot(X_range, y_range_pred, label=f'Degree {deg}')

# Finalising the plot
```

```
plt.title('Bias-Variance Tradeoff for Polynomial Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```



4. Visualising the bias-variance tradeoff

Finally, we plot the training and testing errors (mean squared errors) for each polynomial degree. This allows us to observe how model complexity (degree) affects the errors and visualize the bias-variance trade-off.

```
plt.figure(figsize=(8, 6))
plt.plot(degrees, train_errors, label='Train Error', marker='o')
plt.plot(degrees, test_errors, label='Test Error', marker='o')
plt.title('Bias-Variance Tradeoff')
plt.xlabel('Polynomial Degree (Model Complexity)')
plt.ylabel('Mean Squared Error (MSE)')
plt.legend()
plt.grid(True)
plt.show()
```