I have two functions that should display the "bias-variance tradeoff" and they do not behave exactly the same. Can you help me to understand why? One is using scikit-learn for most functionality, while the other is using homemade functions. These functions are validated against scikit-learn in other places so they should produce identical results, but when I use the two different implementations in this figure, they are off from each other.Below is some code that eventually plots plot*bias*variance*tradeoff*polynomial*degree*sklearn() and plot*bias*variance*tradeoff*polynomial*degree*own*way():def OLS*parameters(X, y): """ Calculates the optimal parameters, theta, using the ordinary least squares method.

Theta_OLS = inv(X.T @ X) @ X.T @ y

Returns

---

theta : numpy array shape (n)

the optimal parameters, theta as given by the

OLS method.

Parameters

X : numpy array shape (n,f)

Feature matrix for the data, where n is the number

of data points and f is the number of features.

y : numpy array shape (n)

Y values of the data set.

"""

#calculate X^T*X and take the inverse

XTX = X.T@X

XTX_i = np.linalg.inv(XTX)

#calculate X^T*y

XT_y = X.T @ y

#calculate theta

theta = XTX_i @ XT_y

return theta

def variance(y_predict): """ Calculates the variance of the predicted values

Returns

----

variance: float

  Variance value

Parameters

----

y_predict: numpy array shape (n)

  Predicted y values of the data set.

"""

variance = np.mean(np.var(y_predict, axis=1))

return variance

def squared*bias(y, y*predict): """ Calculates the squared bias between true and predicted values. Squaring bias so it is better for plotting.

Returns

----

squared_bias: float

Bias value

Parameters

---

y : numpy array shape (n)

Y values of the data set.

y_predict: numpy array shape (n)

Predicted y values of the data set.

"""

squared_bias = np.mean((y - np.mean(y_predict, axis=1))**2)

return squared_biasdef MSE(y, y_predict):

"""

Calculates Mean Squared Error (MSE)

between true and predicted values

Returns

---

mse : float

MSE error value

Parameters

---

y : numpy array shape (n)

Y values of the data set.

y: numpy array shape (n)

Predicted y values of the data set.

""" n = np.size(y*predict) mse = (1/n) * np.sum((y - y*predict)**2) return mse def plot*bias*variance*tradeoff*polynomial*degree*sklearn(x, y, p=65, bootstraps=200):

x_train, x_test, y_train, y_test = train_test_split(

x[:, None], y, random_state=1, test_size=0.2

)

degrees = np.arange(1, p + 1, step=5)

mses = np.zeros(degrees.shape)

variances = np.zeros(degrees.shape)

biases = np.zeros(degrees.shape)

for i, degree in enumerate(degrees):

model = make_pipeline(

PolynomialFeatures(degree=degree),

LinearRegression(fit_intercept=True)

)

preds = [] # will hold predictions on x_test for each bootstrap (shape per item: (n_test,))

for j in range(bootstraps):

x_train_re, y_train_re = resample(

x_train, y_train,

random_state=j

```
        )

        model.fit(x_train_re, y_train_re)
        preds.append(model.predict(x_test))  # (n_test,)

    # Stack to shape (n_test, bootstraps)
    preds = np.column_stack(preds)

    mses[i] = MSE(y_test[:, None], preds)
    variances[i] = variance(preds)
    biases[i] = squared_bias(y_test, preds)

plt.figure(figsize=(6, 4))
plt.plot(degrees, mses, label="MSE")
plt.plot(degrees, variances, label="Variance")
plt.plot(degrees, biases, label="Bias^2")
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend(fontsize=12)
plt.ylabel("Prediction error (logarithmic scale)", fontsize=12)
plt.xlabel("Polynomial degree", fontsize=12)
plt.yscale("log")
plt.savefig("Bias_variance_tradeoff_degree.png", bbox_inches='tight')
plt.show()def polynomial_features(x, p,intercept=False):

"""
Generates a polynomial feature matrix with or without
intercept, based on the values of x.
```

```
Returns
-------
X : numpy vector shape(n,p), if intercept shape(n,p+1)
   the resulting feature matrix of all polynomial combinations
   up to a given degree. Vandermonde format.



Parameters
----------
x : numpy vector shape(n)
   x values from dataset

p : int
   number of degrees

intercept : Bool
   Bool to determine if intercept should be included or not:
   False : no intercept
   True : include intercept
"""

n = len(x)

#handling the intercept column
#to avoid branching in loop
if intercept:
   matrix_p = p+1
```

```
    start_col = 1
    i_offs = 0

    X = np.zeros((n, matrix_p))
    X[:,0] = np.ones(n)

else:
    matrix_p = p
    start_col = 0
    i_offs = 1

    X = np.zeros((n, matrix_p))

for i in range(start_col,matrix_p):
        X[:,i] = np.power(x,i+i_offs)

return X
```

np.random.seed(350) x*noise* = *np.linspace(-1, 1, num=1000)* *y*noise = (1 / (1 + 25 * x*noise***2))* + *np.random.normal(0, 0.1, size=x*noise.size) p = 65 X*noise* = *polynomial*features(x*noise, p, intercept=use*intercept)

test and train dataset, and scaling of X*train and X*test

X*train*noise, X*test*noise, y*train*noise, y*test*noise = train*test*split(X*noise, y*noise, random*state=1, test*size=0.2)plot*bias*variance*tradeoff*polynomial*degree*esklearn(x*noise, y*noise, p=p)plot*bias*variance*tradeoff*polynomial*degree*own*way(X*train*noise, y*train*noise, X*test*noise, y*test_noise)Let me know if I seem to have missed any function that is needed

TL;DR – What's most likely going wrong

1. **Numerical solver difference (the big one):**\

Your homemade OLS_parameters uses the explicit normal equation inverse\ \[\theta = (X^\top X)^{-1} X^\top y]\ which is **numerically unstable** (and can outright fail) for high-degree polynomials (up to 65!), because the Vandermonde matrix is ill-conditioned.\ **scikit-learn's** LinearRegression uses an SVD/least-squares solver (np.linalg.lstsq) which is much more stable and robust. This alone can produce visibly different curves at moderate–high degrees.

1. **Intercept handling mismatch:**\

In your sklearn pipeline you have:

```
1  PolynomialFeatures(degree=degree)   # include_bias=True by
default -> adds a column of 1s
2  LinearRegression(fit_intercept=True)  # also fits an intercept
```

That's a **double intercept** (a duplicated all-ones column). LinearRegression can handle this because it uses a robust solver. But if your own path also includes an intercept column and then tries to invert (X^\top X), you will often get a singular or near-singular system → large numerical errors.

1. **Possibly inconsistent feature sets per degree:**\

In the sklearn path, features are **re-generated for each degree** inside the loop. In your own path you prebuilt

```
1  X_noise = polynomial_features(x_noise, p,
intercept=use_intercept)
```

If your plot_bias_variance_tradeoff_polynomial_degree_own_way(...) is not slicing the first d columns correctly for each degree (or is reusing all p columns), the models won't be comparable.

1. **Train/test split + bootstrapping must be identical:**\

In the sklearn function you:

```
1  - Split on `x`/`y`\.
2  - Bootstrap only the **training set** \(with
`random_state=j`\)\.
3  - Predict on the **same fixed** `x_test` each bootstrap\.
```

Your own function must do the *same*. If it bootstraps differently (or bootstraps the test set, or uses different seeds), the curves will not be identical.

1. **"Scaling" (mentioned in a comment) can be fine but must be consistent:**\

Scaling does not change predictions for plain OLS if applied consistently to train and test, but it can interact with numerical conditioning. If your own path scales and the sklearn path does not, results can differ slightly (or even more at very high degree).

1. **Shape conventions for metrics:**\

Your MSE, variance, and squared_bias expect preds to have shape (n_test, bootstraps); any transpose mismatch in the custom path will change results. Your versions look correct for the sklearn path.

Concrete fixes to make them match (strongly recommended)

1) Replace the explicit inverse with a stable least-squares solve

```
def OLS_parameters(X, y):
    """
    Stable OLS: solves min ||X theta - y||_2 using least squares (SVD),
    not the explicit (X^T X)^(-1) X^T y.
    """
    # Returns theta of shape (n_features,)
    theta, *_ = np.linalg.lstsq(X, y, rcond=None)
    return theta
```

This change alone usually makes your own implementation line up with scikit-learn for high-degree polynomials.

---

2) Make intercept handling identical in both paths

Pick **one** of these two consistent setups:

Option A (recommended):

- **scikit-learn:** PolynomialFeatures(degree=d, include_bias=False), and LinearRegression(fit_intercept=True)
- **own:** polynomial_features(x, d, intercept=False), and your OLS_parameters does **not** add an intercept column; if you want an intercept, add a column of ones yourself and let OLS_parameters solve it.

Option B:

- **scikit-learn:** PolynomialFeatures(degree=d, include_bias=True), and LinearRegression(fit_intercept=False)
- **own:** polynomial_features(x, d, intercept=True)

Personally I prefer **A** (no bias column in features, let the linear model fit the intercept).

Update your sklearn function accordingly:

```
model = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False),
    LinearRegression(fit_intercept=True)
)
```

And ensure your own polynomial_features(..., intercept=False) does not add a bias column.

---

3) Ensure both loops use the same degree grid

In sklearn you use:

```
degrees = np.arange(1, p + 1, step=5)  # -> 1, 6, 11, ..., 61 (for p=65)
```

Use the **same** degrees in your own function.

When you prebuild X_noise up to p, make sure for a degree d you pass only the first d columns (plus 1 if you kept an intercept column), e.g.:

```
# If intercept=False in polynomial_features:
X_train_d = X_train_noise[:, :d]   # x, x^2, ..., x^d
X_test_d  = X_test_noise[:, :d]
```

---

4) Bootstrap in the same way

Mirror the scikit-learn path:

- Keep x_test and y_test **fixed** across bootstraps.
- Resample **only** (X_train_d, y_train) for each bootstrap.
- Use resample(..., random_state=j) so seeds match.

---

5) Keep metrics identical

Your helper metrics are fine for the sklearn path. To make intent crystal-clear, I'd recommend these "explicit" versions:

```
def mse_matrix(y_true, preds):  # preds: (n_test, B)
    return np.mean((y_true[:, None] - preds)**2)

def var_matrix(preds):          # preds: (n_test, B)
```

```
    return np.mean(np.var(preds, axis=1, ddof=0))

def bias2_matrix(y_true, preds):# preds: (n_test, B)
    mu = np.mean(preds, axis=1)
    return np.mean((y_true - mu)**2)
```

They're mathematically equivalent to yours and make the broadcasting explicit.

A reference "own-way" plotting function (aligned with sklearn)

Below is a version that mirrors your sklearn function as closely as possible (uses np.linalg.lstsq, consistent intercept handling, same bootstrapping, same degrees):

```
import numpy as np
from sklearn.utils import resample
import matplotlib.pyplot as plt

def plot_bias_variance_tradeoff_polynomial_degree_own_way(x, y, p=65, bootstraps=200):
    # Split exactly like the sklearn path
    from sklearn.model_selection import train_test_split
    x_train, x_test, y_train, y_test = train_test_split(
        x, y, random_state=1, test_size=0.2
    )

    degrees = np.arange(1, p + 1, step=5)
    mses = np.zeros(degrees.shape)
    variances = np.zeros(degrees.shape)
    biases = np.zeros(degrees.shape)

    for i, degree in enumerate(degrees):
        # Build polynomial features with NO intercept column
        X_train = polynomial_features(x_train, degree, intercept=False)
        X_test  = polynomial_features(x_test,  degree, intercept=False)

        preds = []
        for j in range(bootstraps):
            Xb, yb = resample(X_train, y_train, random_state=j)
```

```
25
26              # Fit with stable least-squares (SVD)
27              theta, *_ = np.linalg.lstsq(Xb, yb, rcond=None)
28              preds.append(X_test @ theta)
29
30          preds = np.column_stack(preds)  # (n_test, bootstraps)
31
32          mses[i] = np.mean((y_test[:, None] - preds)**2)
33          variances[i] = np.mean(np.var(preds, axis=1, ddof=0))
34          biases[i] = np.mean((y_test - np.mean(preds,
axis=1))**2)
35
36      plt.figure(figsize=(6, 4))
37      plt.plot(degrees, mses, label="MSE")
38      plt.plot(degrees, variances, label="Variance")
39      plt.plot(degrees, biases, label="Bias^2")
40      plt.xticks(fontsize=12)
41      plt.yticks(fontsize=12)
42      plt.legend(fontsize=12)
43      plt.ylabel("Prediction error (logarithmic scale)",
fontsize=12)
44      plt.xlabel("Polynomial degree", fontsize=12)
45      plt.yscale("log")
46      plt.show()
```

And in your sklearn path, make the intercept choice match:

```
1  model = make_pipeline(
2      PolynomialFeatures(degree=degree, include_bias=False),
3      LinearRegression(fit_intercept=True)
4  )
```

Quick debugging checklist (to locate the exact mismatch)

If they still differ:

1. **Compare predictions for one degree and one bootstrap:**

```
1  # Using the same x_train/x_test/y_train and degree, same
bootstrap seed j
2  # A) sklearn preds_skl : (n_test,)
3  # B) own     preds_own : (n_test,)
4  print("max abs diff:", np.max(np.abs(preds_skl - preds_own)))
```

If this is ~1e-10…1e-12 → they're equivalent. If it's big, it's one of the issues above.

1. **Print the rank / condition number:**

```
import numpy.linalg as LA
print("cond(X_train):", LA.cond(X_train))
```

If the condition number skyrockets at higher degrees, any explicit inverse will go off the rails.

1. **Ensure degree slicing is correct** if you prebuilt X up to p.
2. **Verify shapes:** preds should be (n_test, bootstraps) for the metrics you wrote.

(Optional) Tips to make the experiment more stable

- Use **Chebyshev** or **Legendre** polynomial bases (orthogonal on (\[-1,1])) to improve conditioning at high degrees.
- Or add a tiny **ridge** regularization in your own path:

\[\theta = (X^\top X + \lambda I)^{-1} X^\top y] This will alter the "pure OLS" setting, but can prevent blowups at very high degree.

What I still need from you (if it's still off)

You mentioned:

- plot_bias_variance_tradeoff_polynomial_degree_own_way(…) — I don't see its code.
- use_intercept — I don't see its value.
- You commented "and scaling of X_train and X_test" — I don't see that scaling code either.

If you paste those three pieces, I can point to the exact line(s) that are causing the divergence.

If you apply the two main fixes — **use **np.linalg.lstsq in your OLS and **make intercept handling identical** — your curves should match the scikit-learn ones very closely. Want me to adapt your existing functions exactly as-is and produce a minimal, fully reproducible script?