Python Code for Data Analysis and Polynomial Regression

1. Importing Libraries

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error

data = pd.read_csv('/content/displacement_data.csv')
dfx = data[["x", "y", "u_x"]]
dfy = data[["x", "y", "u_y"]]
```

2. Plotting 3D Scatter Plot for X and Y

```
x1 = dfx[['x', 'y']]
y1 = dfx['u_x']
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(dfx['x'], dfx['y'], dfx['u_x'], c='b', marker='o')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U_X')
plt.title("3D␣Scatter␣Plot␣of␣X,␣Y␣and␣UX")
plt.show()

sns.scatterplot(x="x", y="u_x", data=dfx)
plt.title("x␣v/s␣u_x")
plt.xlabel("x")
plt.ylabel("u_x")
plt.show()

sns.scatterplot(x="y", y="u_x", data=dfx)
plt.title("y␣v/s␣u_x")
plt.xlabel("y")
plt.ylabel("u_x")
plt.show()
```

3. Applying Polynomial Regression

```
degree = 3
poly = PolynomialFeatures(degree)
X_poly = poly.fit_transform(x1)
model_x = LinearRegression()
model_x.fit(X_poly, y1)
x_range = np.linspace(dfx['x'].min(), dfx['x'].max(), 100)
y_range = np.linspace(dfx['y'].min(), dfx['y'].max(), 100)
X_grid, Y_grid = np.meshgrid(x_range, y_range)
X_grid_poly = poly.transform(np.c_[X_grid.ravel(), Y_grid.ravel()])
Z_grid = model_x.predict(X_grid_poly).reshape(X_grid.shape)

# Plot the original data and the polynomial plane
fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

# Plot the original scatter data
ax.scatter(dfx['x'], dfx['y'], dfx['u_x'], color='blue', alpha=0.5, label
    ="Original␣Data")

# Plot the polynomial plane
ax.plot_surface(X_grid, Y_grid, Z_grid, color='orange', alpha=0.7,
    rstride=100, cstride=100, edgecolor='k')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U_x')
ax.set_title('3D␣Polynomial␣Regression␣Fit')

plt.legend()
plt.show()
u_x_pred = model_x.predict(X_poly)
print("Mean␣Squared␣Error:", mean_squared_error(y1, u_x_pred))
```

4. Estimating Ux as a Function of (x, y)

```
y_fixed = np.full_like(dfx['x'], dfx['y'].mean())
X_x_projection = pd.DataFrame({'x': dfx['x'], 'y': y_fixed})
X_x_projection_poly = poly.transform(X_x_projection)

u_y_x_projection = model_x.predict(X_x_projection_poly)

# Plot the original data projection along the x-axis
plt.figure(figsize=(8, 6))
plt.scatter(dfx['x'], dfx['u_x'], color='blue', alpha=0.5, label='
    Original␣Data')
plt.plot(dfx['x'], u_y_x_projection, color='orange', label='Polynomial␣
    Model␣Projection')
plt.xlabel('X')
plt.ylabel('U_X')
plt.title('X␣vs␣U_X␣Projection')
```

```
plt.legend()
plt.show()
```

5. Polynomial Regression Equation

```
coefficients_x = model_x.coef_
print(coefficients_x)
intercept_x = model_x.intercept_
print(intercept_x)
terms = poly.get_feature_names_out(['x', 'y'])
equation = f"{intercept_x} "

for coef, term in zip(coefficients_x[1:], terms[1:]):
    equation += f"+ ({coef}) * {term} "

print("Polynomial Regression Equation for u_x:")
print(equation)
```

6. Plotting 3D Scatter Plot for UY

```
x2 = dfy[['x', 'y']]
y2 = dfy['u_y']
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(dfy['x'], dfy['y'], dfy['u_y'], c='b', marker='o')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U_Y')
plt.title("3D Scatter Plot of X, Y and UY")
plt.show()

sns.scatterplot(x="x", y="u_y", data=dfy)
plt.title("x v/s u_y")
plt.xlabel("x")
plt.ylabel("u_y")
plt.show()

sns.scatterplot(x="y", y="u_y", data=dfy)
plt.title("y v/s u_y")
plt.xlabel("y")
plt.ylabel("u_y")
plt.show()
```

7. Polynomial Regression Estimating as a Function of (x, y) for UY

```
degree = 3
poly = PolynomialFeatures(degree)
X_poly = poly.fit_transform(x2)

model_y = LinearRegression()
model_y.fit(X_poly, y2)

u_y_pred = model_y.predict(X_poly)

print("Polynomial Coefficients:", model_y.coef_)
print("Intercept:", model_y.intercept_)
print("Mean Squared Error:", mean_squared_error(y2, u_y_pred))
coefficients_y = model_y.coef_
intercept_y = model_y.intercept_
terms = poly.get_feature_names_out(['x', 'y'])
equation = f"{intercept_y} "

for coef, term in zip(coefficients_y[1:], terms[1:]):
    equation += f"+ ({coef}) * {term} "

print("Polynomial Regression Equation for u_y:")
print(equation)
```

8. 3D Polynomial Regression Fit for UY

```
degree = 3
poly = PolynomialFeatures(degree)
X_poly = poly.fit_transform(x2)

model_y = LinearRegression()
model_y.fit(X_poly, y2)

u_y_pred = model_y.predict(X_poly)

print("Polynomial Coefficients:", model_y.coef_)
print("Intercept:", model_y.intercept_)
print("Mean Squared Error:", mean_squared_error(y2, u_y_pred))
coefficients_y = model_y.coef_
intercept_y = model_y.intercept_
terms = poly.get_feature_names_out(['x', 'y'])
equation = f"{intercept_y} "

for coef, term in zip(coefficients_y[1:], terms[1:]):
    equation += f"+ ({coef}) * {term} "

print("Polynomial Regression Equation for u_y:")
print(equation)
```

9. Y vs UY Projection

```
x_fixed = np.full_like(dfy['y'], dfy['x'].mean())
X_y_projection = pd.DataFrame({'x': x_fixed, 'y': dfy['y']})
X_y_projection_poly = poly.transform(X_y_projection)

u_y_y_projection = model_y.predict(X_y_projection_poly)

# Plot the original data projection along the y-axis
plt.figure(figsize=(8, 6))
plt.scatter(dfy['y'], dfy['u_y'], color='blue', alpha=0.5, label='
    Original␣Data')
plt.plot(dfy['y'], u_y_y_projection, color='orange', label='Polynomial␣
    Model␣Projection')
plt.xlabel('Y')
plt.ylabel('U_Y')
plt.title('Y␣vs␣U_Y␣Projection')
plt.legend()
plt.show()


\subsection*{10. Storing Coefficients of \( U_x \) in Array}
\begin{lstlisting}[style=python]
from sympy import symbols, diff
x, y = symbols('x␣y')

ux = (
    intercept_x +
    coefficients_x[1]*x +
    coefficients_x[2]*y +
    coefficients_x[3]*x**2 +
    coefficients_x[4]*x*y +
    coefficients_x[5]*y**2 +
    coefficients_x[6]*x**3 +
    coefficients_x[7]*x**2*y +
    coefficients_x[8]*x*y**2 +
    coefficients_x[9]*y**3
)
print(ux)
```

10. Storing Coefficients of U_x in Array

```
from sympy import symbols, diff
x,y = symbols('x␣y')
ux=(
    intercept_x+
    coefficients_x[1]*x+
    coefficients_x[2]*y+
    coefficients_x[3]*x**2+
```

```
    coefficients_x[4]*x*y+
    coefficients_x[5]*y**2+
    coefficients_x[6]*x**3+
    coefficients_x[7]*x**2*y+
    coefficients_x[8]*x*y**2+
    coefficients_x[9]*y**3
)
print(ux)
```

11. Storing Coefficients of U_y in Array

```
uy = (
    intercept_y +
    coefficients_y[1]*x +
    coefficients_y[2]*y +
    coefficients_y[3]*x**2 +
    coefficients_y[4]*x*y +
    coefficients_y[5]*y**2 +
    coefficients_y[6]*x**3 +
    coefficients_y[7]*x**2*y +
    coefficients_y[8]*x*y**2 +
    coefficients_y[9]*y**3
)
print(uy)
```

12. Calculating Strains

```
exx_1 = diff(ux, x)
eyy_1 = diff(uy, y)

exx = sum(term for term in exx_1.as_ordered_terms() if term.has(x, y))
eyy = sum(term for term in eyy_1.as_ordered_terms() if term.has(x, y))

print(exx)
print(eyy)
```

13. Calculating Stresses in Terms of λ and μ

```
s_xx_lambda = sum([exx, eyy])
s_yy_lambda = s_xx_lambda
s_xx_mew = 2*exx
s_yy_mew = 2*eyy

print(s_xx_lambda)
print(s_yy_lambda)
print(s_xx_mew)
print(s_yy_mew)
```

14. Applying Boundary Condition for $\int \sigma \, dy$ = Reaction Force

```
R_x_1 = s_xx_lambda.subs(x, 1)
R_x_2 = s_xx_mew.subs(x, 1)
print(R_x_1, R_x_2)

from sympy import integrate
R_x_lambda = integrate(R_x_1, (y, 0, 1))
R_x_mew = integrate(R_x_2, (y, 0, 1))
print(R_x_lambda)
print(R_x_mew)

R_y_1 = s_yy_lambda.subs(y, 1)
R_y_2 = s_yy_mew.subs(y, 1)

Reaction = pd.read_csv('/content/reaction_data.csv')
Reaction.head()

R_y_lambda = integrate(R_y_1, (x, 0, 1))

R_x = Reaction.iloc[1, 1]
R_y = Reaction.iloc[3, 1]
C = np.array([R_x, R_y], dtype=float)
print(C)

R_y_mew = integrate(R_y_2, (x, 0, 1))
print(R_y_lambda)
print(R_y_mew)
```

15. Lame's Constant Calculation

```
A = np.array([[R_x_lambda, R_x_mew], [R_y_lambda, R_y_mew]], dtype=float)
print(A)
l = np.linalg.solve(A, C)
print(f"Lame's␣constants␣are:␣{l}")
```

16. Stress Heat Diagram

```
s_xx = l[0]*s_xx_lambda + (l[1]*s_xx_mew)
s_yy = l[0]*s_yy_lambda + (l[1]*s_yy_mew)

from sympy import lambdify
s_x = lambdify((x, y), s_xx)
s_y = lambdify((x, y), s_yy)
e_x = lambdify((x, y), exx)
e_y = lambdify((x, y), eyy)
```

```
data['stress_x'] = np.vectorize(s_x)(data['x'], data['y'])
data['stress_y'] = np.vectorize(s_y)(data['x'], data['y'])
data['e_x'] = np.vectorize(e_x)(data['x'], data['y'])
data['e_y'] = np.vectorize(e_y)(data['x'], data['y'])

data.head()
```

17. Strain Heat Diagram

```
plt.figure(figsize=(10, 6))
contour = plt.tricontourf(data['x'], data['y'], data['stress_y'], levels
    =20, cmap="jet")

# Step 2: Add a color bar for reference
cbar = plt.colorbar(contour)
cbar.set_label("Stress")

# Optional: Add labels, title, etc.
plt.xlabel("X␣Coordinate")
plt.ylabel("Y␣Coordinate")
plt.title("Stress␣in␣y")

plt.show()
```

18. Txy Calculation

```
Txy = (l[1]) * (diff(ux, y) + diff(uy, x))
```

19. Txy Heat Diagram

```
T_xy = lambdify((x, y), Txy)
data['Txy'] = np.vectorize(T_xy)(data['x'], data['y'])
data.head()

plt.figure(figsize=(10, 6))
contour = plt.tricontourf(data['x'], data['y'], data['Txy'], levels=20,
    cmap="jet")

# Step 2: Add a color bar for reference
cbar = plt.colorbar(contour)
cbar.set_label("Stress")

# Optional: Add labels, title, etc.
plt.xlabel("X␣Coordinate")
plt.ylabel("Y␣Coordinate")
```

```
plt.title("Txy")

plt.show()
```