```python
In [13]:  # Import the necessary libraries
          import numpy as np
          import matplotlib.pyplot as plt
          from sklearn.svm import SVC
          from sklearn.preprocessing import StandardScaler

          # Data points: two classes (+1 and -1)
          X = np.array([[1, 1], [-1, 1], [2, 1], [1, -2], [-2, 1]])
          y = np.array([1, 1, -1, -1, -1])

          # Scale the data (standardization helps SVM performance)
          scaler = StandardScaler()
          X_scaled = scaler.fit_transform(X)

          # Create and fit the SVM model with a polynomial kernel (degree 2)
          svm_model = SVC(kernel='poly', degree=2, C=1.0)
          svm_model.fit(X_scaled, y)

          # Create a grid of points to plot decision boundary
          xx, yy = np.meshgrid(np.linspace(-3, 3, 500), np.linspace(-3, 3, 500))
          grid_points = np.c_[xx.ravel(), yy.ravel()]
          grid_points_scaled = scaler.transform(grid_points)

          # Predict decision boundary
          Z = svm_model.decision_function(grid_points_scaled)
          Z = Z.reshape(xx.shape)

          # Plot the results
          plt.figure(figsize=(8, 8))
          plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='black')  # Decision
          boundary

          # Plot the data points
          plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y, cmap=plt.cm.coolwarm, s=100,
          edgecolors='k')

          # Highlight the support vectors
          plt.scatter(svm_model.support_vectors_[:, 0], svm_model.support_vectors_[:,
          1],
                      s=200, facecolors='none', edgecolors='k', linewidths=2)

          plt.title('SVM with Polynomial Kernel (Degree 2)')
          plt.xlabel('Feature 1 (scaled)')
          plt.ylabel('Feature 2 (scaled)')
          plt.show()

          # Extracting the support vectors from the trained SVM model
          support_vectors = svm_model.support_vectors_

          # Inverse transform to get back to the original scale
          support_vectors_original = scaler.inverse_transform(support_vectors)

          # Given Support Vectors
          given_support_vectors = np.array([
              [1.22, 0.5],
              [0.54, -2.0],
              [-1.50, 0.5],
              [0.54, 0.5],
              [-0.82, 0.5]
          ])

          # Display the calculated support vectors
```

```python
print("** Calculated Support Vectors in original coordinates:**")
for idx, sv in enumerate(support_vectors_original, start=1):
    print(f"Calculated Support-Vektor {idx}: {tuple(sv)}")

# Display the given support vectors
print("\n***** Given Support Vectors: *****")
for idx, sv in enumerate(given_support_vectors, start=1):
    print(f"Given Support-Vektor {idx}: {tuple(sv)}")




def polynomial_kernel(x, y, degree=2, coef=1):
    """Berechnet den polynomialen Kernel zwischen zwei Vektoren."""
    return (np.dot(x, y) + coef) ** degree

# Definierte Datenpunkte
data_points = np.array([[1, 1], [-1, 1], [2, 1], [1, -2], [-2, 1]])

# Berechnung der polynomialen Kernelwerte für alle Kombinationen der Datenpun
kte
kernel_matrix = np.zeros((data_points.shape[0], data_points.shape[0]))

for i in range(data_points.shape[0]):
    for j in range(data_points.shape[0]):
        kernel_matrix[i, j] = polynomial_kernel(data_points[i], data_points
[j])

# Ausgabe der Kernel-Matrix
print("\n **** Kernel-Matrix für die gegebenen Datenpunkte: ****")
print(kernel_matrix)
```