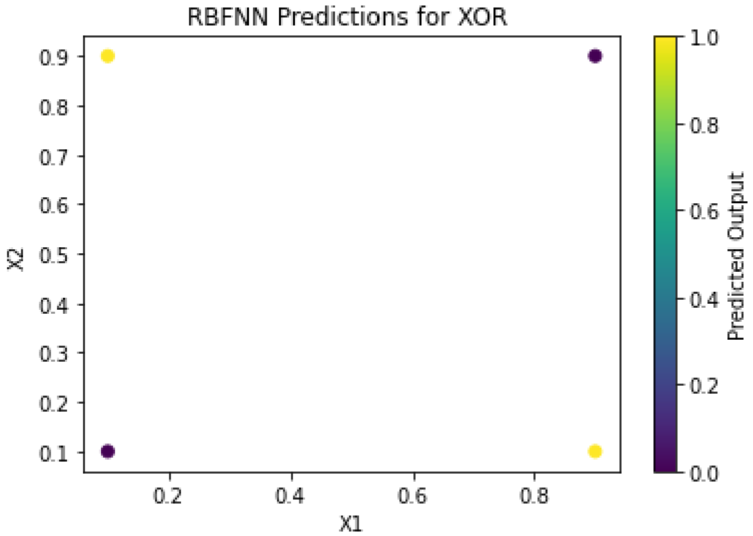
In [24]:

```
import numpy as np
import matplotlib.pyplot as plt

class RBFNN:
    def __init__(self, sigma):
        self.sigma = sigma
        self.centers = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
        self.weights = None

    def _gaussian(self, x, c):
        return np.exp(-np.linalg.norm(x - c) ** 2 / (2 * self.sigma ** 2))

    def _calculate_activation(self, X):
        activations = np.zeros((X.shape[0], self.centers.shape[0]))
        for i, center in enumerate(self.centers):
            for j, x in enumerate(X):
                activations[j, i] = self._gaussian(x, center)
        return activations

    def fit(self, X, y):
        # Calculate activations
        activations = self._calculate_activation(X)

        # Solve for weights
        self.weights = np.linalg.pinv(activations.T @ activations) @ activations.T @

    def predict(self, X):
        activations = self._calculate_activation(X)
        return activations @ self.weights


# Example usage:
if __name__ == "__main__":
    # Define XOR dataset
    X = np.array([[0.1, 0.1], [0.1, 0.9], [0.9, 0.1], [0.9, 0.9]])
    y = np.array([0, 1, 1, 0])

    # Initialize and train RBFNN
    rbfnn = RBFNN(sigma=0.1)
    rbfnn.fit(X, y)

    # Predict
    predictions = rbfnn.predict(X)
    print("Predictions:", predictions)

    # Calculate mean squared error
    mse = np.mean((predictions - y) ** 2)
    print("Mean Squared Error:", mse)

    # Plot the results
    plt.scatter(X[:, 0], X[:, 1], c=predictions, cmap='viridis')
    plt.colorbar(label='Predicted Output')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.title('RBFNN Predictions for XOR ')
    plt.show()
```

```
Predictions: [ 3.39868340e-17  1.00000000e+00  1.00000000e+00 -1.30570525e-17]
Mean Squared Error: 2.498330116506209e-32
```



In [32]:

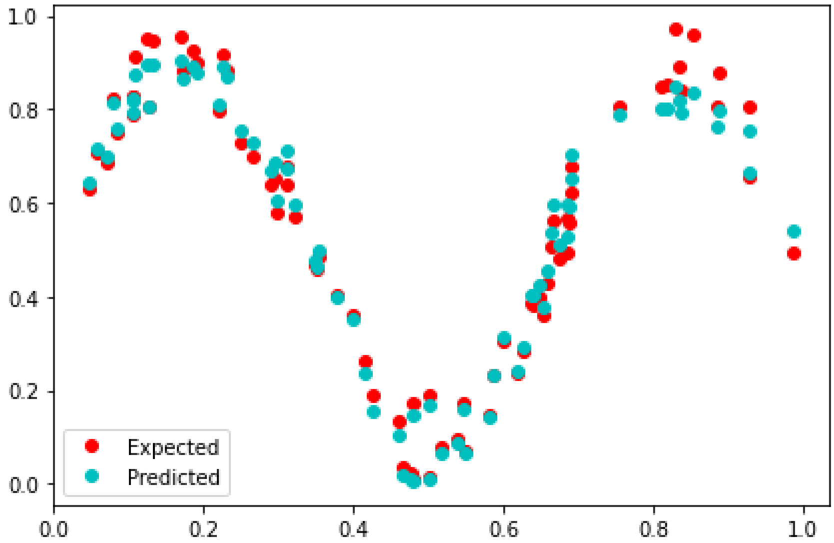
```
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import statistics as stat

def generate_data_points(num_data_points):
    """Generate random data points with noise"""
    y_list = []
    desired_y_list = []
    x_list = []

    for i in range(num_data_points):
        x = np.random.uniform(0.0, 1.0)
        x_list.append(x)
        y = 0.5 + 0.4 * math.sin(3 * math.pi * x)
        noise = np.random.uniform(-0.1, 0.1)
        y_noise = y + noise
        y_list.append(y_noise)
        desired_y_list.append(y)
    return x_list, y_list, desired_y_list

def kmeans(data, num_clusters):
    """K-means clustering algorithm"""
    clusters_x = np.random.choice(np.squeeze(data[0]), size=num_clusters)
    clusters_y = np.random.choice(np.squeeze(data[1]), size=num_clusters)
    clusters = np.array([clusters_x, clusters_y])
    prev_clusters = clusters.copy()

    # Initialize variance
    variance = np.zeros(num_clusters)

    # Initialize distance matrix
    dp, num_clusters = (len(data[0]), num_clusters)
    distance = np.array([[[0.0, 0.0, 0.0] for i in range(dp)] for j in range(num_clu

    # Iteratively update clusters until convergence
    converged = False
    while not converged:
        for i in range(num_clusters):
            cluster = [clusters[0][i], clusters[1][i]]
            for j in range(len(data[0])):
                dp = [data[0][j], data[1][j]]
                squared_distance = (cluster[0] - dp[0])**2 + (cluster[1] - dp[1])**2
                distance[i][j][0] = squared_distance
```

```
                distance[i][j][1] = dp[0]
                distance[i][j][2] = dp[1]

        distanceT = distance.transpose(1,0,2)
        current_cluster_index = 0
        smallest_data_point_x = 0
        smallest_data_point_y = 0
        smallestDistance = 1000
        clusters.fill(0)
        num_dp_belongs_to_each_cluster = [1 for i in range(num_clusters)]
        cluster_dp_x = [[] for i in range(num_clusters)]
        cluster_dp_y = [[] for i in range(num_clusters)]

        for i in range(len(distanceT)):
            for j in range(len(distanceT[i])):
                dis = distanceT[i][j][0]
                if dis < smallestDistance:
                    smallestDistance = dis
                    smallest_data_point_x = distanceT[i][j][1]
                    smallest_data_point_y = distanceT[i][j][2]
                    current_cluster_index = j
            smallestDistance = 1000
            num_dp_belongs_to_each_cluster[current_cluster_index] += 1
            clusters[0][current_cluster_index] += smallest_data_point_x
            cluster_dp_x[current_cluster_index].append(smallest_data_point_x)
            clusters[1][current_cluster_index] += smallest_data_point_y
            cluster_dp_y[current_cluster_index].append(smallest_data_point_y)

        for i in range(num_clusters):
            clusters[0][i] = clusters[0][i] / num_dp_belongs_to_each_cluster[i]
            clusters[1][i] = clusters[1][i] / num_dp_belongs_to_each_cluster[i]

        converged = np.linalg.norm(clusters - prev_clusters) < 1e-6
        prev_clusters = clusters.copy()

    clusters = clusters.transpose()
    clustersWithNoPoints = []
    for i in range(num_clusters):
        dp_for_cluster = num_dp_belongs_to_each_cluster[i]
        if dp_for_cluster < 2:
            clustersWithNoPoints.append(i)
            continue
        else:
            distance_dp_to_cluster = []
            for j in range(len(cluster_dp_x[i])):
                cluster_x = clusters[i][0]
                clsuter_y = clusters[i][1]
                dp_x = cluster_dp_x[i][j]
                dp_y = cluster_dp_y[i][j]
                delta_x_square = (cluster_x - dp_x)**2
                delta_y_square = (clsuter_y - dp_y)**2
                distance_dp_to_cluster.append(math.sqrt(delta_x_square + delta_y_squ
            if len(distance_dp_to_cluster) < 2:
                variance[i] = 0
            else:
                variance[i] = stat.variance(distance_dp_to_cluster)

    if len(clustersWithNoPoints) > 0:
        avg_variance_all_other_clusters = []
        for i in range(num_clusters):
            if i not in clustersWithNoPoints:
                avg_variance_all_other_clusters.append(variance[i])
        variance[clustersWithNoPoints] = np.mean(avg_variance_all_other_clusters)
```

```
    all_same_variance = np.mean(variance)
    all_same_variance = np.array([all_same_variance for i in range(len(variance))])

    return clusters, variance

class RBFNet(object):
    """Implementation of a Radial Basis Function Network"""
    def __init__(self, k=3, lr=0.01, epochs=100):
        """Initialize the RBFNet with hyperparameters and random weights and biases"
        self.k = k
        self.lr = lr
        self.epochs = epochs
        self.w = np.random.randn(k)
        self.b = np.random.randn(1)

    def fit(self, X, y):
        """Train the RBFNet"""
        self.centers, self.variance = kmeans(X, self.k)
        X = X.transpose()
        for epoch in range(self.epochs):
            for i in range(X.shape[0]):
                # Forward pass
                a = np.array([rbf(X[i], center, variance) for center, variance in zi
                F = a.T.dot(self.w) + self.b
                loss = (y[i] - F).flatten() ** 2

                # Backward pass
                error = -(y[i] - F).flatten()

                # Online update
                self.w = self.w - self.lr * a * error
                self.b = self.b - self.lr * error

    def predict(self, X):
        """Make predictions"""
        y_pred = []
        X = X.transpose()
        for i in range(len(X)):
            a = np.array([rbf(X[i], center, variance) for center, variance in zip(se
            F = a.T.dot(self.w) + self.b
            y_pred.append(F)
        return y_pred

def rbf(x, centers, variance):
    """Radial Basis Function"""
    return np.exp(-np.linalg.norm(centers - x) ** 2)

# Generate random data points
x_list, y_list, desired_y_list = generate_data_points(75)
data = np.array([x_list, y_list])

# Create and train the RBFNet
rbf_net = RBFNet(lr=0.01, k=8, epochs=100)
rbf_net.fit(data, desired_y_list)

# Make predictions
y_pred = rbf_net.predict(data)

# Plot the result
plt.plot(data[0], y_list, 'ro', label='Expected')
plt.plot(data[0], y_pred, 'co', label='Predicted')

# Calculate the accuracy and mse
num_correct_prediction_points = 0
```

```
for i in range(len(y_list)):
    if abs(y_pred[i] - y_list[i]) < 1e-1:
        num_correct_prediction_points += 1
print('Prediction accuracy of points: ', num_correct_prediction_points/len(y_list))



plt.legend()
plt.tight_layout()
plt.show()

# Convert lists to NumPy arrays
y_pred_array = np.array(y_pred)
desired_y_array = np.array(desired_y_list)

# Calculate Mean Squared Error
mse = np.mean((y_pred_array - desired_y_array) ** 2)
print('Mean Squared Error:', mse)
```

```
Prediction accuracy of points:  0.9733333333333334
```




```
Mean Squared Error: 0.15263163469602117
```

In [ ]:

In [ ]: