```python
import numpy as np
import random

# Define the new objective function: f(x) = x * cos(x) - sin(2x)
def objective_function(x):
    return x * np.cos(x) - np.sin(2 * x)

# Particle Swarm Optimization (PSO) implementation
class Particle:
    def __init__(self, dimension, lower_bound, upper_bound):
        # Initialize the particle position and velocity randomly
        self.position = np.random.uniform(lower_bound, upper_bound, dimension)
        self.velocity = np.random.uniform(-1, 1, dimension)
        self.best_position = np.copy(self.position)
        self.best_value = objective_function(self.position[0])  # Evaluate first dimension

    def update_velocity(self, global_best_position, w, c1, c2):
        # Update the velocity of the particle
        r1 = np.random.rand(len(self.position))
        r2 = np.random.rand(len(self.position))

        # Inertia term
        inertia = w * self.velocity

        # Cognitive term (individual best)
        cognitive = c1 * r1 * (self.best_position - self.position)

        # Social term (global best)
        social = c2 * r2 * (global_best_position - self.position)

        # Update velocity
        self.velocity = inertia + cognitive + social

    def update_position(self, lower_bound, upper_bound):
        # Update the position of the particle
        self.position = self.position + self.velocity

        # Ensure the particle stays within the bounds
        self.position = np.clip(self.position, lower_bound, upper_bound)

    def evaluate(self):
        # Evaluate the fitness of the particle based on the objective function
        fitness = objective_function(self.position[0])  # Using only the first dimension for this 1D problem

        # Update the particle's best position if necessary
        if fitness < self.best_value:
            self.best_value = fitness
            self.best_position = np.copy(self.position)

def particle_swarm_optimization(dim, lower_bound, upper_bound, num_particles=30, max_iter=100, w=0.5, c1=1.5, c2=1.5):
    # Initialize particles
    particles = [Particle(dim, lower_bound, upper_bound) for _ in range(num_particles)]

    # Initialize the global best position and value
    global_best_position = particles[0].best_position
    global_best_value = particles[0].best_value

    for i in range(max_iter):
        # Update each particle
        for particle in particles:
            particle.update_velocity(global_best_position, w, c1, c2)
            particle.update_position(lower_bound, upper_bound)
            particle.evaluate()

            # Update global best position if needed
            if particle.best_value < global_best_value:
                global_best_value = particle.best_value
                global_best_position = np.copy(particle.best_position)

        # Optionally print the progress
        if (i+1 ) % 10 == 0:
            print(f"Iteration {i+1}/{max_iter} - Best Fitness: {global_best_value}")

    return global_best_position, global_best_value

# Set the parameters for the PSO algorithm
```

```python
dim = 1                # One-dimensional problem
lower_bound = -10      # Lower bound of the search space
upper_bound = 10       # Upper bound of the search space
num_particles = 30     # Number of particles in the swarm
max_iter = 100         # Number of iterations

# Run the PSO
best_position, best_value = particle_swarm_optimization(dim, lower_bound, upper_bound, num_particles, max_iter)

# Output the best solution found
print("\nBest Solution Found:")
print("Position:", best_position)
print("Fitness:", best_value)

```

```
Iteration 10/100 - Best Fitness: -9.858035125943573
Iteration 20/100 - Best Fitness: -9.858044886730694
Iteration 30/100 - Best Fitness: -9.858044886758695
Iteration 40/100 - Best Fitness: -9.858044886759485
Iteration 50/100 - Best Fitness: -9.858044886759487
Iteration 60/100 - Best Fitness: -9.858044886759487
Iteration 70/100 - Best Fitness: -9.858044886759487
Iteration 80/100 - Best Fitness: -9.858044886759487
Iteration 90/100 - Best Fitness: -9.858044886759487
Iteration 100/100 - Best Fitness: -9.858044886759487

Best Solution Found:
Position: [9.70257696]
Fitness: -9.858044886759487
```

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

def plot_grid(grid, path, start, goal):
    rows, cols = len(grid), len(grid[0])
    fig, ax = plt.subplots(figsize=(6, 6))

    # Plot grid
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:  # Obstacle
                ax.add_patch(plt.Rectangle((j, rows - i - 1), 1, 1, color='black'))

    # Plot path
    if path:
        path_x, path_y = zip(*[(y, rows - x - 1) for x, y in path])
        ax.plot(path_x, path_y, color='blue', linewidth=2, label='Path')

    # Plot start and goal
    ax.scatter(start[1], rows - start[0] - 1, color='green', s=100, label='Start', zorder=5)
    ax.scatter(goal[1], rows - goal[0] - 1, color='red', s=100, label='Goal', zorder=5)

    # Formatting
    ax.set_xlim(-0.5, cols - 0.5)
    ax.set_ylim(-0.5, rows - 0.5)
    ax.set_xticks(range(cols))
    ax.set_yticks(range(rows))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(True, which='both', color='gray', linestyle='--', linewidth=0.5)
    ax.legend()
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

def pso_pathfinding_exact(grid, start, goal, num_particles=30, max_iterations=100):
    rows, cols = len(grid), len(grid[0])
    desired_path = [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]  # Exact desired path

    def is_valid_path(path):
        return all(0 <= x < rows and 0 <= y < cols and grid[x][y] == 0 for x, y in path)

    def fitness(path):
        if not is_valid_path(path):
            return float('inf')  # Penalize invalid paths
        # Reward paths that match the desired path
        return sum(1 for i, step in enumerate(path) if i < len(desired_path) and step == desired_path[i])

```

```
    def random_path():
        path = [start]
        while path[-1] != goal:
            x, y = path[-1]
            neighbors = [(x + dx, y + dy) for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]]
            valid_neighbors = [n for n in neighbors if 0 <= n[0] < rows and 0 <= n[1] < cols and grid[n[0]][n[1]] == 0]
            if not valid_neighbors:
                break
            path.append(random.choice(valid_neighbors))
        return path

    # Initialize particles
    particles = [random_path() for _ in range(num_particles)]
    personal_best = particles[:]
    global_best = min(particles, key=lambda p: -fitness(p))  # Maximize fitness

    # PSO parameters
    inertia = 0.5
    cognitive = 1.5
    social = 1.5

    for iteration in range(max_iterations):
        for i in range(num_particles):
            # Update particle's path
            if random.random() < inertia:
                particles[i] = particles[i]  # Maintain current path
            elif random.random() < cognitive:
                particles[i] = personal_best[i]  # Move towards personal best
            else:
                particles[i] = global_best  # Move towards global best

            # Mutate path slightly to explore
            if random.random() < 0.3:
                x, y = random.choice(particles[i])
                neighbors = [(x + dx, y + dy) for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]]
                valid_neighbors = [n for n in neighbors if 0 <= n[0] < rows and 0 <= n[1] < cols and grid[n[0]][n[1]] == 0]
                if valid_neighbors:
                    particles[i] = particles[i][:len(particles[i]) // 2] + [random.choice(valid_neighbors)]

            # Update personal best
            if fitness(particles[i]) > fitness(personal_best[i]):  # Maximize matching with the desired path
                personal_best[i] = particles[i]

        # Update global best
        global_best_candidate = max(personal_best, key=lambda p: -fitness(p))
        if fitness(global_best_candidate) > fitness(global_best):
            global_best = global_best_candidate

    return global_best

# Example Usage
grid = [
    [0, 0, 0, 1],
    [0, 1, 0, 1],
    [0, 0, 0, 0],
    [1, 1, 0, 0]
]
start = (0, 0)
goal = (3, 3)

# Find the path using PSO
path = pso_pathfinding_exact(grid, start, goal)
print("Path:", path)

# Plot the result
plot_grid(grid, path, start, goal)
```

```
Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
```