Ex

Exercise 13.1

- 5(a), 7, 10, 11, 12, 15, 16

Exercise 13.2

- 2, 7, 13, 15, 20, 21, 22, 23, 24, 25

Com Ex

Computer Exercise 13.1

- 2(a)

Computer Exercise 13.2

- 1(a), 1(e)

In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

In [ ]:
```
def golden_section_search(f, a, b, tol=1e-10):
    gratio = (np.sqrt(5) - 1) / 2

    a1 = b - gratio * (b - a)
    a2 = a + gratio * (b - a)

    while abs(b - a) > tol:
        func1 = f(a1)
        func2 = f(a2)
        if func1 > func2:
            a = a1
            a1 = a2
            a2 = a + gratio * (b - a)
        else:
            b = a2
            a2 = a1
            a1 = b - gratio * (b - a)

    x_opt = (a + b) / 2
    return x_opt

print("Golden Section Search fo Sin(x)")
x_golden = golden_section_search(np.sin, 0, np.pi/2)
print(f"x = {x_golden}, f(x) = {np.sin(x_golden)}")

print("Scipy Minimize for Sin(x)")
```

```
x_scipy = sp.optimize.minimize_scalar(np.sin, bounds=(0, np.pi / 2), method='bou
print(f"x = {x_scipy.x}, f(x) = {np.sin(x_scipy.x)}")
```

```
Golden Section Search fo Sin(x)
x = 4.515392518989481e-11, f(x) = 4.515392518989481e-11
Scipy Minimize for Sin(x)
x = 5.786836745895891e-06, f(x) = 5.786836745863593e-06
1.0
```

In [26]:
```
def newton_method(f, grad_f, hess_f, x0, tol=1e-6, max_iter=100):
    x = x0.copy()
    for i in range(max_iter):
        g = grad_f(x)
        H = hess_f(x)
        if np.linalg.norm(g) < tol:
            print(f"Converged at iteration {i}")
            break
        try:
            p = np.linalg.solve(H, g)
        except np.linalg.LinAlgError:
            print("Hessian is singular or not positive definite.")
            break
        x = x - p
    return x

def bfgs(f, grad_f, x0, tol=1e-6, max_iter=100):
    n = len(x0)
    x = x0.copy()
    B = np.eye(n)
    for i in range(max_iter):
        g = grad_f(x)
        if np.linalg.norm(g) < tol:
            print(f"Converged at iteration {i}")
            break

        p = -np.linalg.solve(B, g)
        alpha = 1.0
        x_new = x + alpha * p

        s = x_new - x
        y = grad_f(x_new) - g
        ys = np.dot(y, s)

        if ys == 0.0:
            print("Division by zero in BFGS update.")
            break

        Bs = B @ s
        B += np.outer(y, y) / ys - np.outer(Bs, Bs) / (s @ Bs)

        x = x_new

    return x

def nelder_mead(f, x0, alpha=1, gamma=2, rho=0.5, sigma=0.5, tol=1e-6, max_iter=
    n = len(x0)
    simplex = [x0]
    for i in range(n):
        y = x0.copy()
        y[i] += 1.0
        simplex.append(y)
    simplex = np.array(simplex)

    for iteration in range(max_iter):
        vals = np.array([f(x) for x in simplex])
        idx = np.argsort(vals)
        simplex = simplex[idx]
        vals = vals[idx]

        centroid = np.mean(simplex[:-1], axis=0)
        xr = centroid + alpha * (centroid - simplex[-1])
        fr = f(xr)

        if vals[0] <= fr < vals[-2]:
            simplex[-1] = xr
        elif fr < vals[0]:
            xe = centroid + gamma * (xr - centroid)
            fe = f(xe)
            if fe < fr:
                simplex[-1] = xe
            else:
                simplex[-1] = xr
        else:
            xc = centroid + rho * (simplex[-1] - centroid)
            fc = f(xc)
            if fc < vals[-1]:
                simplex[-1] = xc
            else:
                for i in range(1, n+1):
                    simplex[i] = simplex[0] + sigma * (simplex[i] - simplex[0])

        if np.std(vals) < tol:
            print(f"Converged at iteration {iteration}")
            break

    return simplex[0]

def simulated_annealing(f, x0, T0=1.0, Tmin=1e-6, alpha=0.95, max_iter=10000, st
    x_curr = x0.copy()
    f_curr = f(x_curr)
    x_best = x_curr.copy()
    f_best = f_curr
    T = T0

    for i in range(max_iter):
        # 랜덤 이웃점 생성 (여기서는 단순히 각 차원에 step_size 범위 내에서 노이
        x_new = x_curr + np.random.uniform(-step_size, step_size, size=len(x0))
        f_new = f(x_new)
        delta = f_new - f_curr

        if delta < 0 or np.random.rand() < np.exp(-delta / T):
            x_curr = x_new
            f_curr = f_new
            if f_new < f_best:
                x_best = x_new
                f_best = f_new

        T *= alpha
        if T < Tmin:
            print(f"Temperature below Tmin at iteration {i}")
```

```
                break

        return x_best

rosenbrock = lambda x: sum(100.0 * (x[1:] - x[:-1]**2.0)**2.0 + (1 - x[:-1])**2.
grad_rosenbrock = lambda x: np.array([-400 * x[0] * (x[1] - x[0]**2) - 2 * (1 -
hess_rosenbrock = lambda x: np.array([[1200 * x[0]**2 - 400 * x[1] + 2, -400 * x

print(f"Newton's Method for Rosenbrock: {newton_method(rosenbrock, grad_rosenbro
print(f"BFGS for Rosenbrock: {bfgs(rosenbrock, grad_rosenbrock, np.array([-1.2,
print(f"Nelder-Mead for Rosenbrock: {nelder_mead(rosenbrock, np.array([-1.2, 1])
print(f"Simulated Annealing for Rosenbrock: {simulated_annealing(rosenbrock, np.
print(f"Scipy Minimize BFGS for Rosenbrock: {sp.optimize.minimize(rosenbrock, np
print(f"Scipy Minimize Nelder-Mead for Rosenbrock: {sp.optimize.minimize(rosenbr
print(f"Scipy Minimize Simulated Annealing for Rosenbrock: {sp.optimize.dual_ann
print()

woods = lambda x: (
    100 * (x[0] ** 2 - x[1]) ** 2 + (x[0] - 1) ** 2 +
    90 * (x[2] ** 2 - x[3]) ** 2 + (x[2] - 1) ** 2 +
    10.1 * ((x[1] - 1) ** 2 + (x[3] - 1) ** 2) +
    19.8 * (x[1] - 1) * (x[3] - 1)
)

grad_woods = lambda x: np.array([
    400 * x[0] * (x[0]**2 - x[1]) + 2 * (x[0] - 1),
    -200 * (x[0]**2 - x[1]) + 20.2 * (x[1] - 1) + 19.8 * (x[3] - 1),
    360 * x[2] * (x[2]**2 - x[3]) + 2 * (x[2] - 1),
    -180 * (x[2]**2 - x[3]) + 20.2 * (x[3] - 1) + 19.8 * (x[1] - 1)
])

hess_woods = lambda x: np.array([
    [
        1200 * x[0]**2 - 400 * x[1] + 2,
        -400 * x[0],
        0,
        0
    ],
    [
        -400 * x[0],
        220.2,
        0,
        19.8
    ],
    [
        0,
        0,
        1080 * x[2]**2 - 360 * x[3] + 2,
        -360 * x[2]
    ],
    [
        0,
        19.8,
        -360 * x[2],
        200.2
    ]
])

print(f"Newton's Method for Woods: {newton_method(woods, grad_woods, hess_woods,
print(f"BFGS for Woods: {bfgs(woods, grad_woods, np.array([-3, -1, -3, -1]))}")
```

```python
print(f"Nelder-Mead for Woods: {nelder_mead(woods, np.array([-3, -1, -3, -1]))}"
print(f"Simulated Annealing for Woods: {simulated_annealing(woods, np.array([-3,
print(f"Scipy Minimize BFGS for Woods: {sp.optimize.minimize(woods, np.array([-3
print(f"Scipy Minimize Nelder-Mead for Woods: {sp.optimize.minimize(woods, np.ar
print(f"Scipy Minimize Simulated Annealing for Woods: {sp.optimize.dual_annealin
```

```
Converged at iteration 6
Newton's Method for Rosenbrock: [1. 1.]
Converged at iteration 67
BFGS for Rosenbrock: [1. 1.]
Converged at iteration 87
Nelder-Mead for Rosenbrock: [1.00082096 1.0016435 ]
Temperature below Tmin at iteration 269
Simulated Annealing for Rosenbrock: [ 0.1211715  -0.00186836]
Scipy Minimize BFGS for Rosenbrock: [0.9999955  0.99999099]
Scipy Minimize Nelder-Mead for Rosenbrock: [1.00002202 1.00004222]
Scipy Minimize Simulated Annealing for Rosenbrock: [1.00002523 1.00004788]

Converged at iteration 13
Newton's Method for Woods: [-0.96797404  0.94713917 -0.96951629  0.95124763]
Division by zero in BFGS update.
BFGS for Woods: [ 3.55250936e-71 -1.40745583e+68  4.15383749e+34  1.56526148e+69]
Converged at iteration 11
Nelder-Mead for Woods: [0 0 0 0]
Temperature below Tmin at iteration 269
Simulated Annealing for Woods: [-0.66446579  0.48187861 -1.11412755  1.2783952 ]
Scipy Minimize BFGS for Woods: [0.9999998  0.99999962 1.00000008 1.00000017]
Scipy Minimize Nelder-Mead for Woods: [0.99999777 0.99999832 1.00000621 1.0000122
]
Scipy Minimize Simulated Annealing for Woods: [-1. -1. -1. -1.]
```