EECS 127: Homework 9

Michael Pham

Fall 2024

Problems

1 Weak Duality

Problem 1.1. Using Weak Duality, show that the following set is empty:

$$\left\{x \in \mathbb{R}^2 : x_1^2 + x_2^2 = 1, x_1 + x_2 \geq 2\right\}.$$

Solution. To begin with, we note that this can be rephrased as:

$$\begin{aligned} \min \quad & 0 \\ \text{s.t.} \quad & x_1^2 + x_2^2 = 1 \\ & x_1 + x_2 \geq 2 \end{aligned}$$

With that in mind, we can look at the dual of the problem. To do this, we first consider $L(x, \lambda, \mu)$ to be:

$$\begin{aligned} L(x, \lambda, \mu) &= 0 + \mu(x_1^2 + x_2^2 - 1) + \lambda(2 - x_1 - x_2) \\ &= 0 + (\mu x_1^2 - \lambda x_1) + (\mu x_2^2 - \lambda x_2) - \mu + 2\lambda \end{aligned}$$

And we note that since this is separable in x_1, x_2, as we did in the lecture, we can look at $\min_{x_1} \mu x_1^2 - \lambda x_1$ and $\min_{x_2} \mu x_2^2 - \lambda x_2$. Note that for $\mu > 0$, it is convex, and so we take the partial with respect to x_1, x_2 respectively to eventually get:

$$\begin{aligned} x_1^*(\lambda, \mu) &= \frac{\lambda}{2\mu} \\ x_2^*(\lambda, \mu) &= \frac{\lambda}{2\mu} \end{aligned}$$

With all of this in mind then, $d(\lambda, \mu)$ when $\mu > 0$ is:

$$\begin{aligned} d(\lambda, \mu) &= \mu\left(\left(\frac{\lambda}{2\mu}\right)^2 + \left(\frac{\lambda}{2\mu}\right)^2 - 1\right) + \lambda\left(2 - \frac{\lambda}{2\mu} - \frac{\lambda}{2\mu}\right) \\ &= \mu\left(2\left(\frac{\lambda}{2\mu}\right)^2 - 1\right) + \lambda\left(2 - 2\frac{\lambda}{2\mu}\right) \\ &= 0.5\frac{\lambda^2}{\mu} - \mu + 2\lambda - \frac{\lambda}{\mu} \\ &= -0.5\frac{\lambda^2}{\mu} - \mu + 2\lambda \end{aligned}$$

So, we select $\lambda = \mu = 1$ to get that $d(1, 1) = -0.5 - 1 + 2 = 0.5 > 0$. Thus, we can conclude that, indeed, the equations $x_1^2 + x_2^2 = 2$ and $x_1 + x_2 \geq 2$ has no solutions; the set is empty as desired. ■

2 Conditions

Problem 2.1.

3 Approximations from Limited Measurements

Problem 3.1. ...

Solution. We begin by solving the problem in Python as follows:

Code: Problem 4.1 Code

```
import cvxpy as cp
import numpy as np

n = 200
threshold = 0.01
success = 0
trials = 20
for trial in range(trials):
        y = np.random.normal(size = n)
        z = np.random.uniform(low = 0, high = 1, size = n)

        X_star = np.outer(y, z)

        m = 0.1 * n**2
        pairs = []
        count = 0

        while count < m:
                i = np.random.randint(0, 200)
                j = np.random.randint(0, 200)
                pair = (i, j)

                flag = True

                for p in pairs:
                if p == pair:
                        flag = False
                        break
                if flag:
                        pairs.append(pair)
                        count += 1

        measurements = np.zeros((n, n))
        for pair in pairs:
        i = pair[0]
        j = pair[1]
        measurements[i, j] = X_star[i, j]

        X_hat = cp.Variable((n, n))
        constraints = [X_hat[i, j] == X_star[i, j] for (i, j) in pairs]
        objective = cp.Minimize(cp.normNuc(X_hat))
        problem = cp.Problem(objective = objective, constraints = constraints)
        problem.solve()

        x_hat = X_hat.value
        if np.linalg.norm(x_hat - X_star, ord=2) <= threshold:
                success += 1

print(f"Empirical Success Rate: {success / trials}")
```

And we note that after running the code, we got a success rate of 95%. ■

4 Numerical Analysis Flashbacks?!

Problem 4.1. content...

Solution. First, we write the following code in cvxpy:

Code: Problem 5.1 Code

```
import cvxpy as cp
import numpy as np

n = 1000
a = np.random.uniform(0, 1, size = n)
b = np.random.uniform(0, 1, size = n)

def global_min():
        x = cp.Variable(n)
        objective = cp.Minimize(
                cp.sum(x**4) + cp.sum(cp.multiply(a, x))**2 + cp.sum(cp.multiply(b, x))
        )

        problem = cp.Problem(objective)
        problem.solve()

        return x.value

def newton(a, b, s, tolerance=1e-4, max_iters=200):
        x = np.ones(n)

        def objective(x, a, b):
                return np.sum(x**4) + (np.dot(a, x))**2 + np.dot(b, x)

        def gradient(x, a, b):
                return 4 * x**3 + 2 * a * np.dot(a, x) + b

        def hessian(x, a):
                return np.diag(12 * x**2 + 2 * a**2) + 2 * np.outer(a, a)

        for iter in range(max_iters):
                grad = gradient(x, a, b)
                hess = hessian(x, a)

                grad_norm = np.linalg.norm(grad, 2)
                if grad_norm < tolerance:
                        print(f"Converged at iteration {iter}.")
                        return objective(x, a, b), iter

                delta_x = np.linalg.solve(hess, grad)
                x = x - s * delta_x

        print(f"Did not converge after {max_iters} iterations")
        return objective(x, a, b), iter

step_sizes = [10, 1.0, 0.1, 0.01]

for s in step_sizes:
        print(f"Running Newton's algorithm with step size s = {s}")
        result, num_iters = newton(a, b, s)
        print(f"Final objective value: {result}, Iterations: {num_iters}")
```

Running through this, we note that for step sizes which are too large, we will in fact diverge; a step size of 10 was too large.

On the other hand, a step size of 1 converged within around 200 steps. We note that smaller step sizes will (eventually) converge as well. However, it will converge a lot slower; we see that for a step size of 0.1 and less, they all were slowly converging, but couldn't complete within 200 steps. ■

Problem 4.2a. content...

Solution. We note that this inequality is trying to change the step-size dynamically.

That is, at first, we'll be trying out larger step-sizes. However, we'll then keep on decreasing it until the objective value's decreased enough based off of its gradient at the given point.

This is better as, firstly, it prevents us from overshooting and thus diverging, but also at the start, we can try out larger values which can help speed up the search for the global minimum. ■

Problem 4.2b. content...

Solution. First, we have the following code:

Code: Problem 5.2ii Code

```
import numpy as np
import cvxpy as cp
import random
import matplotlib.pyplot as plt
n = 1000
a = np.random.uniform(0, 1, n)
b = np.random.uniform(0, 1, n)
max_iter = 2000
x_k = []

def global_min():
        x = cp.Variable(n)
        objective = cp.Minimize(
                cp.sum(x**4) + cp.sum(cp.multiply(a, x))**2 + cp.sum(cp.multiply(b, x))
        )

        problem = cp.Problem(objective)
        problem.solve()

        return x.value

def newton_algorithm_backtracking(s=10, alpha=0.5):
        def objective(x, a, b):
                return np.sum(x**4) + (np.dot(a, x))**2 + np.dot(b, x)

        def gradient(x, a, b):
                return 4 * x**3 + 2 * a * np.dot(a, x) + b

        def hessian(x, a):
                return np.diag(12 * x**2) + 2 * np.outer(a, a)

        x = np.ones(n)
        for i in range(max_iter):
                x_k.append(x)
                grad = gradient(x, a, b)
                hess = hessian(x, a)

                if np.linalg.norm(grad) < 0.0001:
                        return i+1

                step_dir = np.linalg.solve(hess, grad)
                step_size = s

                while objective(x - step_size * step_dir, a, b) >= objective(x, a, b):
                        step_size *= alpha
                x = x - step_size * step_dir
        return max_iter

num_iters = newton_algorithm_backtracking()
print(f"Backtracking converged in {num_iters} iterations")
```

```
x_star = global_min()
x_diff = [np.linalg.norm(x_star - i) for i in x_k]
plt.semilogy(range(len(x_diff)), x_diff)
plt.show()
```

Idk?! ■