Homework 2

Sandy Auttelet

Due: 01/23/2024 at 11:59PM

1. (Graded) Consider the linear program given below. Use software to find optimal solutions for values of δ from 0.5 to 1.2 in steps of 0.1. Compare the solutions and make some observations "regarding active constraints, basic solutions and degeneracy." (original question). Edit: ...regarding how solutions change with respect to δ. Repeat the solutions for the corresponding integer program ($x \in \mathbb{Z}^4$) making some observations about any relationships between the two sets of solutions. (other than integer/non-integer).

$$\max_x \delta x_1 + \delta x_2 + x_3 + x_4 \text{ subject to}$$

$$x_1 + 2x_2 + x_3 + 2x_4 \leq 70,$$

$$2x_1 + x_2 + 2x_3 + x_4 \leq 50,$$

$$x_1 + x_2 + x_3 + 3x_4 \leq 62,$$

$$x \geq 0,$$

$$x \in \mathbb{R}^4.$$

Answer:

```
[In]
import numpy as np
import scipy.optimize as opt

deltas = np.linspace(0.5,1.2,8)
for delta in deltas:
    c=np.array([delta, delta, 1.0, 1.0])

    A = np.array([[ 1.0, 2.0, 1.0, 2.0],\
    [ 2.0, 1.0, 2.0, 1.0],\
    [ 1.0, 1.0, 1.0, 3.0]])

    b = np.array([70.0,50.0, 62.0])

    bounds=((0,np.inf),(0,np.inf),(0,np.inf),(0,np.inf))

    isint=[0,0,0,0] #For all x's to be real numbers
```

```
        #isint=[1,1,1,1] #For all x's to be integers

        res=opt.linprog(-c,A,b,bounds=bounds,integrality=isint)

        print("Delta = ",delta)
        print("x: ",res['fun'])
        print("x-vector: ",res['x'])
        print("Norm of x-vector =", np.round(np.linalg.norm(res['x'])
    ,2), "\n")

    [Out] #For all real x's
    Delta =  0.5
    x:  -32.4
    x-vector:  [ 0.   0.  17.6 14.8]
    Norm of x-vector = 23.0

    Delta =  0.6
    x:  -32.4
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  0.7
    x:  -34.3
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  0.8
    x:  -36.2
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  0.9
    x:  -38.1
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  1.0
    x:  -40.0
    x-vector:  [10. 19.  0. 11.]
    Norm of x-vector = 24.12

    Delta =  1.1
    x:  -44.0
    x-vector:  [10. 30.  0.  0.]
    Norm of x-vector = 31.62

    Delta =  1.2
    x:  -48.0
    x-vector:  [10. 30.  0.  0.]
    Norm of x-vector = 31.62


    [Out] #For all integer x's
    Delta =  0.5
```

```
    x:  -32.0
    x-vector:  [ 0.  4. 16. 14.]
    Norm of x-vector = 21.63

    Delta =  0.6
    x:  -32.4
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  0.7
    x:  -34.3
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  0.8
    x:  -36.2
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  0.9
    x:  -38.1
    x-vector:  [ 0. 19. 10. 11.]
    Norm of x-vector = 24.12

    Delta =  1.0
    x:  -40.0
    x-vector:  [10. 30.  0.  0.]
    Norm of x-vector = 31.62

    Delta =  1.1
    x:  -44.0
    x-vector:  [10. 30.  0.  0.]
    Norm of x-vector = 31.62

    Delta =  1.2
    x:  -48.0
    x-vector:  [10. 30.  0.  0.]
    Norm of x-vector = 31.62

```

Because I had completed this problem before your edit, I did in fact attempt to answer the initial question. This paragraph contains that attempt. Degeneracy in the physics definition implies there is more than one possible quantum state for a particle with a specific energy. To apply this definition to this linear program, I assume degeneracy implies there is more than one possible solution to my problem. From our textbook, this degeneracy is defined as more than 4 constraints are active at x since we are in $\mathbb{R}^4$. In this problem's case, since our system is not subject to a strictly equal condition containing all our variables, I assume there is only one solution to the x vector with our maximum value, potentially resulting in a non-degenerate case as this is related to our problem and physics. In scipy, these active conditions are called b_e generated from the matrix A_e and represents the equality constraint of the function *opt.linprog*,

which are by default None.

For the real valued x-vector, when we change δ, we can see that x-vector[0] and x-vector[1] increase with δ. However, x-vector[2] and x-vector[3] decrease as δ increases. The optimized result of the objective function, called x, also decreases as δ increases. The change in x goes $\Delta x = [0, -1.9, -1.9, -1.9, -1.9, -4.0, -4.0]$ with each iteration of δ which seems odd and arbitrary. Additionally, we can see that the norm of the x-vector increases as δ increases. These changes occur at the same δ as when Δx changes value.

From analysis of the outputs, when $\delta = 0.5$ then the x-vector has different results depending on if we are in the integers. At 0.5, logically, we vary from real numbers to integers in the x-vector. We also see a change in the x variable, the optimized result of the objective function. However, x-vector[1] is 0 for the real solution and 4.0 for the integer solution. Interestingly, the norm of the x-vector is relatively close to the same when rotating between integer and real number solutions at this point, but there is a small shift. The other point of interest is when $\delta = 1.0$ as that also causes a shift in the x-vector. However, the objective function's result does not change from -40.0. In this case, the norms of the x-vector are not very close as we seem to take the increasing step earlier when we are restricted to integers. You see another shift from 0 at x-vector[3] to 11 when looking at the real versus integer solutions. I considered the norms here as I like to ponder the results geometrically. When restricting our problem to integer solutions, sometimes we are forced to "turn" earlier or later than if we are allowed to have real numbered solutions. I am curious why the turn happened earlier when $\delta = 1.0$ considering the x-vector was full of integers regardless of constraints.

2. (Not graded) Practice obtaining solutions to linear and mixed-integer programs. You can find many examples in the books or online. Work toward problems in high dimensions where matrices are more challenging to construct.

   Answer:
   As part of my independent research project, I have played with this kind of problem before. I have used this *linprog* function in my work, as well as other methods to solve systems of equations. Utilizing Newton's Iteration Method, Newton's Directional Method and the Method of Least Squares, I attempted to solve my linear programming problem. One case is listed in my code below representing Newton's Directional Method utilizing *np.linalg.solve*. Your class has not introduced me to the concepts of Jacobian and Hessian matrices as their construction was necessary in my work. I built these matrices for arbitrarily high dimensional problems and constructed general solutions as such:

```
1     def newton_iter(wn, vectors, C1s, C2s, C3s):
2         """
3         Iterates through the jacobian and hessian of flat norm to
    minimize weights.
4
5         Parameters
6         ----------
7         wn : list of floats
```

```
            Initial guess of weights to minimize flat norm functional
    .
        vectors : list of lists of arrays of two floats
            Vectors list generated from graph class for a specific
    vertex.
        C1s : list of floats
            List of constant 1 value for each vector.
        C2s : list of floats
            List of constant 2 value for each vector.
        C3s : list of list of floats
            Symmetric matrix of constant 3 values for each vector and
     all its pairs.
        num : int
            number of time iteration is used to approximate constant
    3s integrand.

        Returns
        -------
        newton_dir : float
            Approximate direction of newton's method.

        """
        jacobian = jacob(wn, vectors, C1s, C2s, C3s)
        hessian = hess(wn, vectors, C1s, C3s)
        newton_dir = np.linalg.solve(hessian,-jacobian)
        return newton_dir


    def hess(guess, vectors, C1s, C3s):
        """
        Builds hessian matrix for all weights specific to each vertex
    .

        Parameters
        ----------
        guess : list of floats
            Initial guess of weights to minimize flat norm functional
    .
        vectors : list of lists of arrays of two floats
            Vectors list generated from graph class for a specific
    vertex.
        C1s : list of floats
            List of constant 1 value for each vector.
        C3s : list of list of floats
            Symmetric matrix of constant 3 values for each vector and
     all its pairs.
        num : int
            number of time iteration is used to approximate constant
    3s integrand.

        Returns
        -------
        hessian: array of arrays of floats
            Symmetric matrix of values of second derivative of flat
```

```
    norm function with respect to alpha weight and beta weight.

        """
        hessian = []
        for j in range(0,len(vectors)):
            hessian_row = []
            for k in range(0,len(vectors)):
                w_alpha = guess[j]
                w_beta = guess[k]
                double_derivative = d2F_dwa_dwb(w_alpha, w_beta,
vectors, C1s[j], C3s[j][k], j, k)
                hessian_row.append(double_derivative)
            hessian.append(hessian_row)
        return np.array(hessian)

```

Considering this course is devoted to optimization, I look forward to learning how to make my code run faster. I am told list comprehension is a good speed up, and I have integrated this method into my currently used functions, but I look forward to building a basis of knowledge on this topic to make intuitively optimized neural networks with ease.