Optimal Assignment of the Keyboard Number Row

Derryl Sayo

```python
import math
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
```

```
(CVXPY) Apr 15 11:57:37 AM: Encountered unexpected exception importing solver GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected < 9.8.0. Please open a
(CVXPY) Apr 15 11:57:37 AM: Encountered unexpected exception importing solver PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected < 9.8.0. Please open a
```

Background

On a standard computer keyboard, the base-10 numbers are arranged in sequential order:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

As context, I have designed and built ergonomic mechanical keyboards in hopes of reducing wrist, hand, and finger strain as a large portion of my work is done digitially through programming, note-taking, etc. My current design iteration reduces the number of keys from a standard 108 to 48 arranged in a 4×12 grid (ie. "ortholinear") fashion. As a result, I have had to move many of the characters to multiple layers accessed by holding buttons underneath my thumbs. In the small keyboards community, these layers are mostly left as an "exercise for the reader" where empirical experimentation leads to the variety of keymappings shared around the internet.

As a result, my compromise/solution for typing the numbers is to access it via holding a layer button and assigning the home row to the sequential order as noted above with the left pink on 0 and the right on 9. However, this sequential order does not reflect the frequency these numbers appear in text. For example, in my experience 0 has been a widely used character in typing dates, prices, etc. yet it is placed underneath my second weakest finger the right pinky.

In pursuit of a solution, one idea would be to use integer programming as a canonical assignment problem to assign each number to keyboard location which we will explore here.

Problem Statement

On a keyboard with the base-10 numbers arranged in sequential order in a single row, we have $i = 0, \dots, 9$ positions with $i = 0$ representing the left pinky and $i = 9$ representing the right pinky and $j = 0, \dots 9$ numbers.

Assign each position to a number to minimize the cost of typing frequently used numbers.

Decision Variables

Let x_{ij} be assignment of position i with number j. $x_{ij} = 1$ if the assignment is true and 0 if not.

```
X = cp.Variable((10, 10), integer=True)
```

Costs

The cost of assigning a number to a location comes is the combination of two factors:

1. The relative comfort level of each finger pressing a key. This will be subjective and we will be able to parametrize it to suit each individual. Higher values imply less comfortable so should incur a higher cost.
2. The frequency ranking of the number in text. Higher values are assigned to more frequently used numbers as there is more repeated finger stress. Some ideas where this can be generated is by counting frequencies by feeding in a training dataset (eg. Linux kernel code, online shopping prices…) however as a proof of concept the rankings are assigned by personal experience.

We take the outer product of these two vectors to form a 10×10 matrix where each position c_{ij} represents a cost of assigning finger i to number j:

```
# Ranking the home row finger positions from 1-10
# NOTE: The index is the finger position.
comfort_levels = [10, 8, 3, 4, 6, 7, 1, 2, 5, 9]

# Approximate rankings for number frequency in
# NOTE: The index is the number on the keyboard.
frequency_ranking = [1/1, 1/2, 1/3, 1/4, 1/5, 1/7, 1/8, 1/9, 1/6, 1/10]

# Construct the cost matrix
C = np.outer(comfort_levels, frequency_ranking)

print(np.round(C, 3))
```

```
[[10.     5.     3.333  2.5    2.     1.429  1.25   1.111  1.667  1.   ]
 [ 8.     4.     2.667  2.     1.6    1.143  1.     0.889  1.333  0.8  ]
 [ 3.     1.5    1.     0.75   0.6    0.429  0.375  0.333  0.5    0.3  ]
 [ 4.     2.     1.333  1.     0.8    0.571  0.5    0.444  0.667  0.4  ]
 [ 6.     3.     2.     1.5    1.2    0.857  0.75   0.667  1.     0.6  ]
 [ 7.     3.5    2.333  1.75   1.4    1.     0.875  0.778  1.167  0.7  ]
 [ 1.     0.5    0.333  0.25   0.2    0.143  0.125  0.111  0.167  0.1  ]
 [ 2.     1.     0.667  0.5    0.4    0.286  0.25   0.222  0.333  0.2  ]
 [ 5.     2.5    1.667  1.25   1.     0.714  0.625  0.556  0.833  0.5  ]
 [ 9.     4.5    3.     2.25   1.8    1.286  1.125  1.     1.5    0.9  ]]
```

Objective Function

We try to find the best *overall* assignment by minimizing the total cost. This can be written as the objective function $\min \sum_{i,j} c_{ij}x_{ij}$

```
objective = cp.Minimize(cp.sum(cp.multiply(C,X)))
```

Constraints

Standard to the assignment problem we have the constraints that

1. Each position is assigned to a single number:

$$\sum_j x_{ij} = 1$$

2. Each number is assigned to a single position:

$$\sum_i x_{ij} = 1$$

```
constraint1 = [X >= 0]
constraint2 = [cp.sum(X[i,:]) == 1 for i in range(10)]
constraint3 = [cp.sum(X[:,j]) == 1 for j in range(10)]
constraints = constraint1 + constraint2 + constraint3
```

We can add other constraints to further improve the comfort of the assignment. Some ideas that we can explore is adding a limit/penalty to typing multiple frequently used numbers on the same hand, moving commonly used bigrams (ie. different numbers used in succession) to different hands.

Solution

We use CVXPY to solve this integer programming problem:

```
problem = cp.Problem(objective, constraints)
problem.solve()
X.value
```

```
array([[-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.],
       [-0., -0., -0., -0., -0., -0.,  1., -0., -0., -0.],
       [-0., -0.,  1., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0.,  1., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1., -0., -0.]])
```

So, our optimal layout is:

$$\begin{bmatrix} 9 & 6 & 2 & 3 & 8 & 5 & 0 & 1 & 4 & 7 \end{bmatrix}$$

However, this solution is pretty trivial and close to how we would intuitively rearrange the keys if we knew the relative frequencies of each key.

Let's see if we can do better by modifying the costs or adding more constraints.

Costs of Re-Arranging Keys

We can add extra cost to the objective function to account for muscle memory retraining. One natural conclusion would be that moving the key further would incur more "cost" in finding the optimal layout. In addition, to prevent completely dominating the weights, we can potentially scale the distance cost as I have done below.

Let $C = [c_{ij}]$ be the cost of assigning number j to position i. Like before, this is computed by the outer product of the comfort and frequency ranking vectors with an added element-wise multiplication by the distance costs (scaled by some selected value as stated above).

```
dist_scale = 0.2

# The numbers are offset by 1 from their positional index
# (ie. index 0 is number 1, index 9 is number 0)
distances = np.zeros((10, 10))
for i in range(10):
        for j in range(1,11):
                distances[j - 1, i] = abs(j - i - 1)

print("Distance cost matrix: \n", distances)
```

```
Distance cost matrix:
 [[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
 [1. 0. 1. 2. 3. 4. 5. 6. 7. 8.]
 [2. 1. 0. 1. 2. 3. 4. 5. 6. 7.]
 [3. 2. 1. 0. 1. 2. 3. 4. 5. 6.]
 [4. 3. 2. 1. 0. 1. 2. 3. 4. 5.]
 [5. 4. 3. 2. 1. 0. 1. 2. 3. 4.]
 [6. 5. 4. 3. 2. 1. 0. 1. 2. 3.]
 [7. 6. 5. 4. 3. 2. 1. 0. 1. 2.]
 [8. 7. 6. 5. 4. 3. 2. 1. 0. 1.]
 [9. 8. 7. 6. 5. 4. 3. 2. 1. 0.]]
```

```
C = np.outer(comfort_levels, frequency_ranking) + (dist_scale * distances)

print("Cost matrix: \n", np.round(C, 3))
```

```
Cost matrix:
 [[10.     5.2    3.733  3.1    2.8    2.429  2.45   2.511  3.267  2.8  ]
 [ 8.2    4.     2.867  2.4    2.2    1.943  2.     2.089  2.733  2.4  ]
 [ 3.4    1.7    1.     0.95   1.     1.029  1.175  1.333  1.7    1.7  ]
 [ 4.6    2.4    1.533  1.     1.     0.971  1.1    1.244  1.667  1.6   ]
 [ 6.8    3.6    2.4    1.7    1.2    1.057  1.15   1.267  1.8    1.6   ]
 [ 8.     4.3    2.933  2.15   1.6    1.     1.075  1.178  1.767  1.5   ]
 [ 2.2    1.5    1.133  0.85   0.6    0.343  0.125  0.311  0.567  0.7   ]
 [ 3.4    2.2    1.667  1.3    1.     0.686  0.45   0.222  0.533  0.6   ]
 [ 6.6    3.9    2.867  2.25   1.8    1.314  1.025  0.756  0.833  0.7   ]
 [10.8    6.1    4.4    3.45   2.8    2.086  1.725  1.4    1.7    0.9   ]]
```

Now, we can re-formulate the integer programming problem using the new costs in the objective function and solve again using CVXPY:

```
objective = cp.Minimize(cp.sum(cp.multiply(C,X)))
problem = cp.Problem(objective, constraints)
problem.solve()
X.value
```

```
array([[-0., -0., -0., -0., -0., -0.,  1., -0., -0., -0.],
       [-0., -0.,  1., -0., -0., -0., -0., -0., -0., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0.,  1., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.]])
```

This time, we get our optimal layout as

$$\begin{bmatrix} 6 & 2 & 1 & 3 & 4 & 5 & 0 & 7 & 8 & 9 \end{bmatrix}$$

which only the 6 being moved the most and the most common keys under the most comfortable fingers.

Conclusion, Thoughts, and Future Improvements

As seen above, finding the optimal number row layout can be formulated and solved as an integer programming problem by minimizing the cost of key reassignment, number frequency, and the relative comfort levels of each finger position. Additional costs and constraints can be added to further tailor the generated layout to a person's typing style and workload.

One aspect I would like to improve upon is to vary the number frequencies based on typing workload by finding some datasets that more accurately count the number of times each number is pressed. For example, manual price entry on an online shopping website might use 9 (ie. $19.99) more often compared to a computer programmer who might use 1s and 0s more.