大阪市立科学館研究報告 35, 37 – 40 (2025)

# 数理最適化を用いた勤務シフトの作成

江 越　航 *

## 概　要

科学館での勤務シフトを作成するため、数理最適化の手法を用いる方法を試みた。科学館ではプラネタリウムの投影を、担当学芸員が交代で行っている。スタッフの勤務シフトを組むにあたり、年間の勤務日数など種々の制約条件を考慮して、各日の出勤者の割り振りが必要になる。現在この作成を手作業で行っているが、これを Python の数理最適化ライブラリである PuLP を用いて自動的に作成することを試みた。本報では、その手法について述べる。

### 1. はじめに

科学館ではプラネタリウムの投影を、休館日を除く毎日、平日は 7 回、土日祝日は 8 回行っている。これを 7 名のプラネタリウム担当学芸員、および数名の補助スタッフが交代で担当している。

スタッフの年間の勤務日数は決まっていることから、シフトを組んで交代勤務を行うことになる。

シフトを組むにあたり、年間の勤務日数に加えて、1 週間当たりの勤務日数、一日当たりの必要スタッフ数、職責や経験に応じた出勤者の組み合わせなどを考慮して、各日の出勤者を決める必要がある。一方で、土日祝の休みはなるべく多くしたいという希望もある。勤務シフト表を作成する際には、これらの制約を考慮しながら、出勤者を割り当てていく作業が必要になる。

現在この作成を手作業で行っているが、今回 Python の数理最適化ライブラリである PuLP を用いて自動的に作成することを試みた。以下では、その手法について述べる。

### 2. 数理最適化

数理最適化とは、与えられた制約条件の下で、目的となる値を最大または最小にすることである。例えば、トラックの配送計画や、工場での生産計画、インフラの整備計画、最適な乗換ルートの検索などがその例である。

数理最適化の簡単な例として、以下の不等式

$$x \geqq 0 \quad y \geqq 0$$
$$x + 3y \leqq 10 \quad 2x + y \leqq 9$$

を満たす領域 $D$ を考え、この領域 $D$ における $x+y$ の最大値を求めることを考える。この領域 $D$ が制約条件であり、$x+y$ は目的関数と呼ばれる。

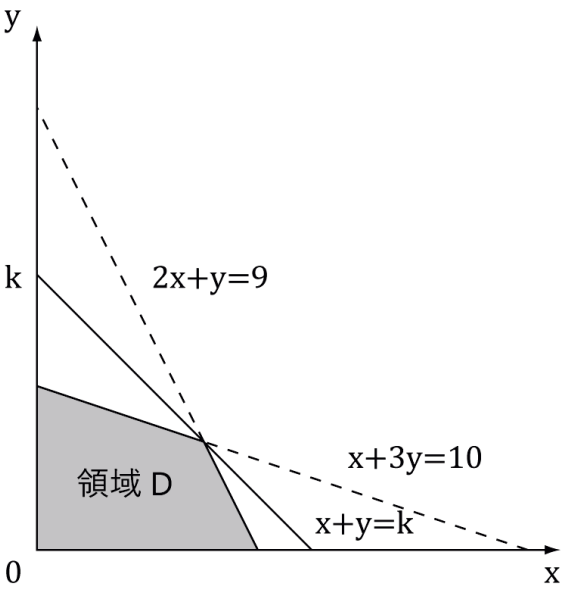


**図1　領域問題**

$x+y=k$ とおけば、これは傾き -1 の直線となり、その $y$ 切片が $k$ である。$k$ を最大にするには、領域 $D$ の右上の点 $x$=3.4、$y$=2.2 を選べば、$x+y$=5.6 となり、最大値

*大阪市立科学館 学芸員
e-mail:egoshi@sci-museum.jp

を取ることが求められる。

これを Python で実行するには、以下のようなスクリプトを作ればよい。

ライブラリをインポートした後、pulp.LpProblem で数理モデルを格納するオブジェクトを定義する。最大化問題なので、引数にはpulp.LpMaximizeを指定する。

次に、pulp.LpVariable により、変数 $x$、$y$ を定義する。cat='Continuous'にて、変数の種類として、実数を指定する。

```
# ライブラリのインポート
import pulp

# 最適化モデルの定義
prob =
 pulp.LpProblem('Domain_Problem',pulp.
  LpMaximize)

# 変数の定義
x =
 pulp.LpVariable('x',cat='Continuous')
y =
 pulp.LpVariable('y',cat='Continuous')
```

制約条件は prob +=の形式により定義する。prob +=に続いて不等式または等式で制約式を記述することで、制約条件を表現する。ここでは図 1 の領域 $D$ を指定している。

```
# 領域の定義
prob += x >= 0
prob += y >= 0
prob += x + 3 * y <= 10
prob += 2 * x + y <= 9
```

次に、目的関数を定義する。目的関数も制約条件と同じく prob +=の形式で指定するが、不等式または等式を含まない式となる。

```
# 目的関数の定義
prob += x + y
```

以上の準備をもとに、prob.solve()とすることで、定義した数理モデルが解かれる。

```
# 最適化実行
status = prob.solve()
```

最適化された結果は、x.value()などにより、その値を表示できる。

```
# 結果表示
print('x', x.value())
print('y', y.value())
```

この結果、解として $x$=3.4、$y$=2.2 が求められたことがわかる。

以上のように、数理最適化ライブラリである PuLP により、制約条件、目的関数を定義すれば、数理モデルを解いて、目的関数を最大化する $x$、$y$ を求めることができる。

## 3. シフト表の作成

### 3-1. カレンダーの作成

次に、PuLP をシフト表作成に応用する方法を考える。

準備として、元のカレンダーとなる表 1 のような csv ファイルを用意する。このファイルでは、列方向に日付と曜日を、行方向に staff 名を記載している。また後の便宜のために、休館日、土日祝の列も設け、該当する行に 1 を記載しておく。

staff1～staff7 の列には、あらかじめ出勤が決まっている場合には、1 を記載しておく。

表 1 シフト表カレンダー

| 日付 | 曜 | 休館日 | 土日祝 | Staff1 | … | Staff7 |
|---|---|---|---|---|---|---|
| 4/1 | 火 | | | | | |
| 4/2 | 水 | | | 1 | | 1 |
| 4/3 | 木 | | | | | |
| 4/4 | 金 | | | | | |
| 4/5 | 土 | | 1 | | | |
| 4/6 | 日 | | 1 | | | |
| 4/7 | 月 | 1 | | | | |
| … | … | | | | | |
| 3/31 | 火 | | | | | |

### 3-2. スクリプトの作成

Python のスクリプトは以下のようになる。最初にライブラリとして PulP をインポートする。また、テーブル形式のデータを扱うには、Pandas を使うと便利なので、これもインポートしておく。

```
#ライブラリの用意
import pulp
import pandas as pd
```

次に、csv ファイルを読み込んで、必要なデータを抽出する。Pandas の ilocメソッドを用いて、日付、曜日、休館日、土日祝の各列を読み出している。スタッフ名

については、columns を用いて、列名を取得している。

```
# リストの定義
# CSV ファイルの読み込み
df = pd.read_csv('shift_data.csv')

# 必要なデータの抽出
calendar = df.iloc[:, 0]   # A 列（日付）
dayofweek = df.iloc[:, 1]  # B 列（曜日）
closed_day = df.iloc[:, 2]  # C 列（休館日）
weekends_holidays = df.iloc[:, 3]
                                # D 列（土日祝）
staff = df.columns[4:]
                          # E 列以降（スタッフ名）
```

次に、最適化モデルを定義する。引数には目的関数を最小化するため、pulp.LpMinimize を指定した。

```
# 最適化モデルの定義
prob =
 pulp.LpProblem('PlanetariumShift',
  pulp.LpMinimize)
```

各スタッフの毎日の出勤・休みを表す変数として、x[cal, sf]を定義する。cal が日付、sf がスタッフを指す。x の値が 1 の時は出勤、0 の時は休みを意味する。

```
# 変数の定義
cs = [(cal, sf) for cal in calendar for
 sf in staff]
x = pulp.LpVariable.dicts('x', cs,
 cat='Binary')
```

以後、定義したモデルに制約条件を加えていく。まず、各スタッフの休暇日数に制約を加えるため、pulp.lpSum により年間の出勤人数の合計を計算している。年間に必要な休暇日数は 123 日のため、出勤日数の合計は 242 日となる。

```
# 絶対条件 #
# 必要休暇日数を守る
for sf in staff:
 prob += pulp.lpSum([x[cal, sf] for cal in
  calendar]) == 242
```

また、日曜日から土曜日の間に、必ず 2 日の休日が必要となるため、この制約条件を加える。range(5, 365 - 6, 7)は、2025 年度は最初の日曜日が 0 から数えて 5 日目なので、これをスタートとし、7 日ごとに範囲を区切っている。年度末は、最低 7 日ある場合に制約を加えている。

```
# 日曜日から土曜日の間の 7 日ごとに必ず 2 日休む
for sf in staff:
 for start in range(5, 365 - 6, 7):
  # 最初の日曜日から 7 日ごと
   prob += pulp.lpSum([x[calendar[i], sf]
 for i in range(start, start + 7)]) <= 5
```

科学館の休館日は全員休みとするため、x[cal, sf]の値を 0 に設定する。

```
# 休館日は全員休み
for i, cal in enumerate(calendar):
  if closed_day[i] == 1:  # 休館日
    for sf in staff:
      prob += x[cal, sf] == 0
```

1 日当たり、最低 4 名の出勤が必要となるため、この条件を加える。開館日の場合、closed_day は値なし（NaN）、または 0 となっているため、この日の出勤者数の合計が 4 人以上になるようにする。

```
# 1 日当たりの最低必要出勤者数を 4 人以上にする
for i, cal in enumerate(calendar):
  if pd.isna(closed_day[i]) or
   closed_day[i] == 0:  # 開館日
    prob += pulp.lpSum([x[cal, sf] for sf
     in staff]) >= 4
```

初期条件として、元のカレンダーの csv ファイルで、あらかじめ出勤が定められている日は、x[cal, sf]の値を 1 に設定する

```
# 出勤があらかじめ決まっている日は x を 1 に
for i, cal in enumerate(calendar):
  for sf in staff:
    if df.at[i, sf] == 1:
      prob += x[cal, sf] == 1
```

以上で、必ず必要な制約条件が設定できたことから、prob.solve により数理モデルを解けば、求める勤務シフトを得ることができる。

```
# 最適化実行
status = prob.solve()
```

### 3-3. 目的関数の設定

前項のスクリプトにより、必要な条件を満たした勤務シフトを得ることができる。しかし、機械的に出勤パターンを作成するため、休みが飛び飛びになったり、土日祝に関しても特に制限なく出勤を割り振ったりと、必ず

しも満足度が高いシフト表になっていない。

そこで、必ずではないが、可能な範囲で条件を満たす、という制限を設けることで、より満足度の高い出勤パターンにする。これは目的関数を定義して、その目的関数を最小化することで、実現することができる。

例えば、土日祝の出勤者をなるべく減らすには、以下のような目的関数を定義する。ここでは、`lpSum([x[cal, sf])`により、土日祝の各スタッフの出勤日数を数え上げ、これをなるべく少なくなるようにすることで、土日祝の出勤を減らすことになる。

```
# 目的関数
# 土日祝の出勤を減らす
prob += pulp.lpSum([x[cal, sf]
 for i, cal in enumerate(calendar) for sf
 in staff if weekends_holidays[i] == 1])
```

また、職責や経験から、特定のスタッフ同士がなるべく同じ日の出勤になるようにしたい場合がある。

この場合、まず各日ごとに値を取る差分の補助変数`diff`を定義する。この変数は、最低値が0で、整数の値を取るように定義する。

次に、各日において、staff1 と staff2 の出勤の差分を計算する。この際、

```
x[cal, 'staff1'] - x[cal, 'staff2']
x[cal, 'staff2'] - x[cal, 'staff1']
```

の2通りの計算をしているが、これはPuLPの制約条件で絶対値を使うことができないためである。変数 `diff` は0以上と定義されているので、staff1とstaff2の出勤パターンが異なれば、正の値を取る。

目的関数として、変数 `diff` の値が小さくなるようにすれば、結果的にstaff1とstaff2の出勤パターンが一致することになる。

```
# 差分の補助変数
diff = pulp.LpVariable.dicts('diff',
 calendar, lowBound=0, cat='Integer')

# 出勤パターンを一致させる制約
for cal in calendar:
  prob += diff[cal] >=
   x[cal, 'staff1'] - x[cal, 'staff2']
  prob += diff[cal] >=
   x[cal, 'staff2'] - x[cal, 'staff1']
```

```
# 目的関数
prob += pulp.lpSum([x[cal, sf]
 pulp.lpSum(diff[cal] for cal in
  calendar)
```

このほか、条件をうまく設定することにより、飛び石で休みになるパターンを減らす、土日祝の休みをなるべく公平に割り当てる、などの条件を加えることが可能になる。

### 3-4. 結果

最適化の実行により、各スタッフの毎日の出勤・休みを表す変数である`x[cal, sf]`の値が求まる。これを適当なメソッドで書き出すことで、目的とするシフト表が得られる。

しかしながら、休みが飛び飛びになったり、連休が続いたりと、あまり満足度が高くないパターンになることがある。

条件を厳しくし過ぎると、最適解がなかったり、目的関数がなかなか収束しないことがある。その場合は、以下のように制限時間を設けたり、許容ギャップを設定することで、現実的な時間で解を得るようにすることができる。

```
# 最適化実行
solver = pulp.PULP_CBC_CMD(msg=True,
 timeLimit=60, gapRel=0.05)
status = prob.solve(solver)
```

## 4. まとめ

Python の数理最適化ライブラリである PuLP を用いて、数理最適化の手法を用いてシフト表を作成することを試みた。

条件を満たしたシフト表が簡単に作成できるが、必ずしも満足度が高いものになるとは限らない。これは、どのような制約条件や目的関数を設定するかで変わってくるが、曖昧な希望だと、スクリプトに書き下しにくいこともある。

現在のところ、手作業で作成した方が、より満足度の高い勤務シフトを作ることができるが、そのための補助として、さらに将来的には条件を適切に設定することで、自動的にシフト表を作成することができると思われる。