

Extended Application of Burnside's Lemma to Higher Dimension using Matrix and Linear Transformation

楊記綱

2022 年 9 月 17 日

概 要

1 `module Burnside where`

針對清大課程 Burnside's Lemma 去從線性轉換與矩陣、向量的角度去做延伸。令嘗試將此想法推廣到更高維度、非正多面體與較複雜之正多面體。

目錄

How to Read

0.1 Defining Notations

0.1.1 Vector

All vectors within this document (no matter if it's in Haskell code or TeX equations) refers to column vectors (even if it's written in style of row vector).

0.1.2 Matrix

Within this document, we'll often use matrix as a denotation for an ordered-list of vectors.

$$[v_1, \dots, v_k] = \left[\begin{pmatrix} v_{1,1} \\ \vdots \\ v_{1,n} \end{pmatrix} \quad \dots \quad \begin{pmatrix} v_{k,1} \\ \vdots \\ v_{k,n} \end{pmatrix} \right] = \begin{bmatrix} v_{1,1} & \dots & v_{k,1} \\ \vdots & \ddots & \vdots \\ v_{1,n} & \dots & v_{k,n} \end{bmatrix}$$

k = amount of vectors, n = vectors' dimension

And since we're treating matrix as a list (or a set, since it's guaranteed that each vector within it is unique¹), for simplicity, we'll be using following two expression quite often.

$$\text{column vector } v \text{ that exists in } M: v \in M \tag{1a}$$

$$\text{for all column vector } v \text{ in } M: \forall v \in M \tag{1b}$$

¹The sole reason I still prefer to call it a list instead of a set is basically due to its ordered nature. Cause in math, sets are unordered, but here it's critical to main that order-ness, as we're identifying faces through that order.

1 Brief Introduction to Matrix and Linear Transformation

So what does matrix in linear transformation means? Well, for any $n \times n$ matrix, you could think of each column as representing each axis' unit vector's position after the transformation. So for example:

$$\text{Let } T = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad (2)$$

$$\text{means, shifting } \hat{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \hat{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \hat{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ to} \quad (3)$$

$$\hat{i}' = \begin{bmatrix} a \\ d \\ g \end{bmatrix}, \hat{j}' = \begin{bmatrix} b \\ e \\ h \end{bmatrix}, \hat{k}' = \begin{bmatrix} c \\ f \\ i \end{bmatrix} \quad (4)$$

Hence by these definitions we could conclude that:

{data}

定理 1 (最少資料量)

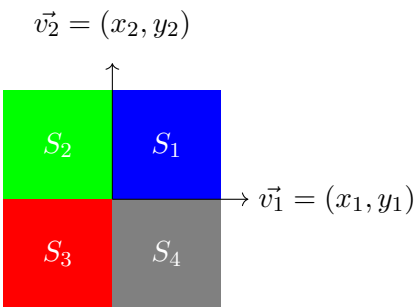
For any object within n -Dimensional space, by specifying n -Surfaces' location could unique identify an object's orientation.

2 Describing Object's State

定義 1.1 (Structure-Matrix)

To describe an object's current orientation, we'll use a set of vectors (which is collected into a matrix).

For example, a 2×2 square could be denoted by two 2-D vectors (aka a 2×2 matrix) like following:

$$M = \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \quad (5)$$


in which we could clearly see by giving two 2-D vectors, we could rigidly define our 4 different squadron (in counterclockwise fashion).

Or like a cube with 6 faces to color, we could denote it as following:

$$\text{令所有面之向量之列表 } L = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \quad (6)$$

Sidenote: On Duplicated Information

One dangerous thing about duplicated info is that you might break structural relationship. As you may have noticed, the matrix above that's being used to denote a cube contains vectors that can be calculated base from others. If we number them from left to right you can see that

$$v_3 = -v_1 \quad (7a)$$

$$v_4 = -v_2 \quad (7b)$$

$$v_6 = v_1 \times v_2 \quad (7c)$$

$$v_5 = -v_6 \quad (7d)$$

Which meant we only need 2 vectors^a to denote a cube. But since in Burnside's Lemma, we're only considering rotation and reflection, leaving duplicated information won't make any difference (except adding some computational time), as rotation and reflection simply won't break an object's structure.

Yet this is still something you should keep in mind, if you're brute forcing all possible outcome by simply shuffling orders of those vectors (instead of using matrices to calculate them). Since you're not using matrix, it'll be your responsibility to ensure those 4 condition listed above are satisfied (suppose you use our initial L).

^aSince our matrix represents an ordered-list, we won't need the third vector to determine orientation. ($[v_1, v_2] \neq [v_2, v_1]$)

3 Operations upon Object's State

Now we can apply transformations onto these vectors. But what could be counted as a valid transformation?

Well, since we already know linear transformation wouldn't break the structure of the object, the only other condition to met is for a transformation to maintain the outline of the object. Or more precisely:

定義 1.1 (Shuffle Transformation)

$$\forall v \in TM (v \in \{v \mid v \in M\}) \quad (8)$$

```
2  -- Check if no new vectors are created and no old vectors are altered
3  validVectors1 :: Matrix -> Matrix -> Bool
4  --           sample    subject    result
5  validVectors1 (Matrix s) (Matrix x) = sort s' == sort x'
6                                     where s' = nub s
7                                     x' = nub x
```

4 Collection of all valid Transformations

當我們在談到旋轉一物體的時候我們可以將其總結成：

$$\mathbb{T} = \{T \mid T \cdot x \in \mathbb{X}, x \in \mathbb{X}\} \quad (9)$$

其中 \mathbb{T} 代表所有合法的 $n \times n$ 矩陣使得作用於 \mathbb{X} 中任意元件會得結果 $T \cdot x = x' \ni x, x' \in \mathbb{X}$ 。那也就是換成講義裡的記號 $|G| = \#\mathbb{T}$ 。

4.1 Calculate $\#\mathbb{T}$

還記得在定理??中所說的最少資料量嗎？現在對我們的 n 維物體來標示各種 orientation 的話，舉 \mathbb{R}^3 物體為例，可靠標記其中三個面的方位即可。所以（為了後續計算方便）我們就永遠選相鄰的三個邊，以數對 (A, B, C) 表示²。那現在令所有可能的數對之集合：

$$\mathbb{P} = \{(D, E, F) \mid \angle DOE = \angle AOB, \angle DOF = \angle AOC, \angle EOF = \angle BOC\} \quad (10)$$

其中我們叫這保角性質我們所選之數對 (A, B, C) 的 Structure，而單純允許旋轉而不允許映射的情況下，這是一個該被確保的 Structure。另外須注意這 Structure 是有方向之分的，通常 A, B, C 採逆時鐘排列（從原點向外指，右手方向）。

那現在定義完 \mathbb{P} ，要計算旋轉方式就簡單多了。首先我們知道 $\forall P \in \mathbb{P}, \exists T \in \mathbb{T} \ni T \cdot (A, B, C) = P$ （其中 $T \cdot (A, B, C) = (TA, TB, TC)$ ）所以 $\#\mathbb{P} = \#\mathbb{T} = |G|$ 。而至於 \mathbb{P} 則可用排列組合推出，對立方體舉例：

先任意選一面 A
再選一面相鄰 A 的面 B
最後再選唯一一個在這兩面逆時鐘方向的面 C
得 $\#\mathbb{P} = C_1^6 \times C_1^4 \times C_1^1 = 24$

對正四面體亦同：

先任意選一面 A
再選一面相鄰 A 的面 B
最後再選唯一一個在這兩面逆時鐘方向的面 C
得 $\#\mathbb{P} = C_1^4 \times C_1^3 \times C_1^1 = 12$

或者嘗試將它 Generalized 對任意正多面體：

$$|G| = (\text{面數}) \times (\text{一面的邊數、一面相鄰的面數}) \quad (11)$$

4.2 Calculate \mathbb{T}

那算完他的數量，我們有沒有從它回推 \mathbb{T} 裡面的內容呢？有的（至少用電腦算式簡單的），我們甚至能算出他的組數（那個 k^n 的 n ）一樣拿立方體舉例：

$$\text{令所有面之向量之列表 } L = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \quad (12a)$$

²本文中所有表示面之數，皆為向量，且皆由原點指向該面之重心

$$\text{舉例取轉換 } T = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (12b)$$

$$\text{得 } TL = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \quad (12c)$$

其中我們可看到，縱列 1~4 向右位移了一格，而 5,6 不變。因此可得對變換 T (z 軸左手旋轉 90 度) 可分成三組，也就是對 L, TL 做 disjoint set 後數他的數量。
就此我們就可以算出他的塗色可能性了。

```

8 listTracks :: Matrix -> Matrix -> [[Int]]
9 listTracks = curry$joinTrack.uncurry trackShift
10
11 trackShift :: Matrix -> Matrix -> [(Int, Int)]
12 trackShift (Matrix von) (Matrix zu) = map (search von') zu'
13     where von' = zip [1..length von] von
14           zu'   = zip [1..length zu] zu
15           search xs (id, x) = (fst.head$filter (\(_, x') -> x'==x) xs, id)
16
17 joinTrack :: [(Int, Int)] -> [[Int]]
18 joinTrack xs = nub $ map ((nub.sort) . makeChain xs) xs
19 -- ~~~ Btw, using `nub` won't break the chain (aka it'll only remove first/last element.
20 makeChain :: [(Int, Int)] -> (Int, Int) -> [Int]
21 makeChain xs (init, termin)
22     | termin /= x = termin:init:makeChain xs (x, termin)
23     | otherwise = [init, termin]
24     where x = fst $ xs!!(init-1)

```

5 Extend Research Topics (未寫)

5.1 2x2 魔術方塊

To be honest, the main reason I'm introducing matrix/linear transformation into Burnside's Lemma is purely due to its elegancy. By abstracting all the ideas of motion and how a set of faces needs to be the same color into matrix, makes it very easy to solve using computer (at least I hope so, originally). But it turns out I'm only halfway right. It's true we could find all possible transformation more confidently either by brute forcing or 排列組合 with some idea I've developed through this theory, but still there's still some parts we have to hack it through. A minor one will be that we'll have to compare two matrices and then group them by who've swapped to whose slot, but that's still acceptable as we're able to write program to do that. But when we enter the realm of Rubik's Cube, we're faced with the greatest obstacle throughout this project.

問題 1

How do we rotate one layer of the Rubik's Cube, while leaving the other layers still?

I'm sorry to say it, but now we're going to enter the “*programming*” realm. Why? Because the best method I could think of is just group layers, then apply transformation seperately.

Well, with our strategy decided, let's start modeling the Rubik's Cube first.

```

25 type Rubiks2x2 = (Matrix, Matrix)
26 --           |           ^ Each Faces
27 --           |> Each Block
28 {- For Rubiks2x2 P F:
29 P = [ Corners*8   , Edges*12   ]
30 F = [ Corners*8*3 , Edges*12*2 ]
31 -}
```

The way I model 2x2 Rubik's Cube is by first giving a position vector p_i and then a facing vector f_i which tells you which direction is the first face facing.

$$P = \begin{bmatrix} p_1 & \dots & p_8 \end{bmatrix}, \text{ each component of } p_i \text{ is either 1 or -1} \quad (13a)$$

$$F = \begin{bmatrix} f_1 & \dots & f_8 \end{bmatrix}, f_i \text{ is one of the 6 different (directional) unit vector} \quad (13b)$$

And know with (p_i, f_i) we could denote any block we want, where we can then give a list of 3 colours (order-sensitive) which will be coloured counterclockwise.³

```

32 attachID :: [[Int]] -> [[Int]]
33 attachID xs = map (uncurry (++)) $ zip (map singleton [1..length xs]) xs
34 p2x2 = Matrix $ attachID $
35     map ([3]++) $
36     (:) <$> [1, -1] <*>
37     ((\a b -> [a, b]) <$> [1, -1] <*> [1, -1])
38 f2x2 = Matrix $ attachID $ concatMap genFace $ stripMatrix p2x2
39 twoByTwo = (p2x2, f2x2)
```

```

40 genFace :: [Int] -> [[Int]]
41 genFace (i:g:x:y:z:_) = map (i:) $ (fx x) ++ (fy y) ++ (fz z)
42     where fx v | v==1=[right] | v==(-1)=[left ] | otherwise = []
43           fy v | v==1=[rear ] | v==(-1)=[front ] | otherwise = []
44           fz v | v==1=[top  ] | v==(-1)=[bottom] | otherwise = []
```

```

45 -- Some shortcut for readability
46 top    = [ 0,  0,  1]
47 bottom = [ 0,  0, -1]
48 right  = [ 1,  0,  0]
49 left   = [-1,  0,  0]
50 front  = [ 0, -1,  0]
51 rear   = [ 0,  1,  0]
```

³Since Haskell only support variables' name starting with lower case, we'll have to follow the rule. So just keep in mind that `p2x2` meant P , and `f2x2` meant F .

5.2 Getting Layers

```
52 -- Grouping of 2x2 Rubik's Cube
53 getLayer :: Vector -> Rubiks2x2 -> Rubiks2x2
54 getLayer uv (Matrix p, Matrix f) = (Matrix (corner++edge), Matrix (f'))
55     where f' = filter (\(i:d:x) -> x==uv) f
56           -- (fC, fE) = span (\x -> 5 == length x) f'
57           corner = map ((p !!) . (\(g:_) -> g-1)) f' -- fC
58           edge = []
59           -- edge = map (((p !!).(8 +)).flip div 2.fst.(\(id, x)->(id-24, x))) fE
60 -- TODO: Impl for 3x3
```

5.3 Join Everything Together

```
61 idn :: Int -> Vector
62 idn n
63     | n < 1 = [1]
64     | otherwise = 0:idn (n-1)
65 padU :: Matrix -> Matrix
66 padU (Matrix m) = Matrix $ (1:replicate (length m) 0) : map (0:) m
```

```
67 rot90Z=Matrix [[0,1,0], [-1,0,0], [0,0,1]]
```

```
68 patchLayer' :: Matrix -> Matrix -> Matrix
69 patchLayer' x (Matrix []) = x
70 patchLayer' (Matrix x) (Matrix (t:m)) = patchLayer' (Matrix (a++[t]++b)) (Matrix m)
71     where (a, b') = break (\(i:_) -> i==head t) x
72           b = tail b'
73 patchLayer :: Rubiks2x2 -> Rubiks2x2 -> Rubiks2x2
74 patchLayer (a, b) (p, q) = (patchLayer' a p, patchLayer' b q)
```

```
75 apply :: Matrix -> Vector -> Rubiks2x2 -> Rubiks2x2
76 --      Transform Layer      Input      Output
77 apply t v x = patchLayer x (pt*a, pt*b)
78     where (a, b) = getLayer v x
79           pt = padU.padU $ t
80 -- TODO: Check result!!
```

5.3.1 Proof of Completeness

A Matrix

```
81 type Vector = [Int]
82 vector :: [Int] -> Matrix
83 vector a = Matrix [a]
84
85 newtype Matrix = Matrix [[Int]]
86
87 stripMatrix :: Matrix -> [[Int]]
88 stripMatrix (Matrix x) = x
```

```
89 instance Show Matrix where
90     show (Matrix [x:xs])
91         | null xs = "[" ++ show x ++ "]"
92         | otherwise = "[" ++ show x ++ "]\n" ++ show (Matrix [xs])
93     show (Matrix rows)
94         | length (head rows) > 1 = "[" ++ intercalate ", " (map (show.head) rows) ++ "]\n"
95                                     ++ show (Matrix (map tail rows))
96         | otherwise = "[" ++ intercalate ", " (map (show.head) rows) ++ "]"
97 dumpMatrix = print.stripMatrix
```

```
98 a = Matrix [[1,2],[2,4],[2,0],[2,2]]
99 b = Matrix [[2,4],[2,0],[1,2],[2,2]]
```

```
100 concatM :: Matrix -> Matrix -> Matrix
101 concatM (Matrix x) (Matrix y) = Matrix $ x ++ y
102 diagFlip' :: [[a]] -> [[a]]
103 diagFlip' xs
104     | length (head xs) > 1 = map head xs : diagFlip' (map tail xs)
105     | otherwise = [map head xs]
106 diagFlip :: Matrix -> Matrix
107 diagFlip (Matrix xs) = Matrix $ diagFlip' xs
```

```
108 instance Num Matrix where
109     (Matrix [xs]) + (Matrix [ys]) = Matrix [zipWith (+) xs ys]
110     Matrix (x:xs) + Matrix (y:ys) = concatM (Matrix [zipWith (+) x y]) (Matrix xs + Matrix ys)
111     (Matrix []) + a@(Matrix _) = a
112     a@(Matrix _) + (Matrix []) = a
113     x * (Matrix ys) = Matrix $ map f ys
114                     where f y = map (sum.zipWith (*) y) xs'
115                     (Matrix xs') = diagFlip x
```

B Sage Graphics

B.1 2x2 Rubik's Cube

```
def colorRect3D(x, c, l, f): # f: 0 (xy), 1 (xz), 2 (yz)
    x = vector(x)
    if f == 0:
        sv1 = vector((-1, 0, 0))
        sv2 = vector((0, -1, 0))
    elif f == 1:
        sv1 = vector((-1, 0, 0))
        sv2 = vector((0, 0, -1))
    elif f == 2:
        sv1 = vector((0, -1, 0))
        sv2 = vector((0, 0, -1))
    Gph = Graphics()
    Gph += polygon3d([x, x+sv1, x+sv1+sv2, x+sv2])
    return Gph

def colorBlock(p, f, c):
    Gph = Graphics()
    baseVector = p * 0.5
    for cl in c:
        if f[2]!=0:
            Gph += colorRect3D(baseVector, c, 1, 0)
    return Gph

def plotRubiks2x2(P, F, cs):
    Gph = Graphics()
    for p, f, c in zip(P, F, cs):
        Gph += colorBlock(p, f, c)
    # save3D(Gph) # Comment out if you don't want auto-export
    return Gph

def save3D(g, n="plot"):
    filename = "/tmp/"+n+".html"
    g.save(filename)
    os.system("sed -i 's/\\usr\\share/..\\usr\\share/g' "+filename)
    print("Plot3D saved to: "+filename)
```

C Ternary Operator

```
116 -- https://wiki.haskell.org/Ternary\_operator
117 data Cond a = a :? a
118
119 infixl 0 ?
120 infixl 1 :?
121
122 (?) :: Bool -> Cond a -> a
123 True ? (x :? _) = x
124 False ? (_, :? y) = y
```

LICENSE

Codes

Documentation