Linearization of the Rubik's Cube

Stephen Huan

March 15, 2020

1 Rational

Why make Rubik's cube moves matricies?

See the previous lecture, here. I will re-iterate them here anyways.

1. A series of moves becomes a single matrix - the product of the moves in the series.

$$BAx = (BA)x = Tx$$

where $T = BA$. Thus, R U R' U' = $(U^{-1}R^{-1}UR)$ (reversed because of the order of applying matrix transformations). This allows sequences of moves to be compressed into constant memory, and the application of a sequence of moves to be done by a single matrix mutiplication, which is trivially parallelizable.

2. Systematizes cube logic, by replacing logic with pure linear algebra and ties nicely into group theory. For example, it's commonly known that to inverse a series of moves one can read the series backwards and inverse each move in the sequence. This is an example of $(AB)^{-1} = B^{-1}A^{-1}$.

3. Might inspire novel solving algorithms - the problem of solving the Rubik's Cube becomes a matrix factorization problem, to decompose a target matrix into the product of a series of given matricies.

4. It makes Wikipedia's notation more palatable if you think of moves as matricies.

2 Previous Failure

Eight months ago, I attempted this exact task of making moves matricies along with the cube state. The idea is to represent a cube state as a 6x9 matrix, and transitions as 6x6. Then, if a given cube is represented as S and an R move as R, then applying an R move is equivalent to computing $S' = RS$. However, as hard as I tried I couldn't get it to work. With my new knowledge of linear algebra, I realized that such a task is mathematically impossible.

Start with the solved cube given by:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \end{bmatrix}$$

and a R-turn on the solved cube given by:

$$\begin{bmatrix} 0 & 0 & 5 & 0 & 0 & 5 & 0 & 0 & 5 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 4 & 2 & 2 & 4 & 2 & 2 & 4 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 0 & 4 & 4 & 0 & 4 & 4 & 0 \\ 5 & 5 & 2 & 5 & 5 & 2 & 5 & 5 & 2 \end{bmatrix}$$

As one can see, some columns are unaffected while others have stickers moving around. If one thinks of a matrix A times a vector $\mathbf{x}$ as a linear transformation, and matrix multiplication AB as the application of the same linear transformation A on each column of B, then leaving some columns alone while permuting others is impossible in general. To have $A\mathbf{x} = \mathbf{x}$ in general, then A must be the identity. However, if A is the identity, then it can't permuate the matrix in the other columns.

The problem I was having was that it is possible to solve for a particular solution. If $AB = M$ then $A = MB^{-1}$ which will work for a particular B and a particular M. However, by the above logic, it won't work in general.

3 Novel Work

3.1 Permutation Matricies

If we can't transform different columns differently, put all the stickers in the same column! Let the cube state be represented by a 54x1 vector, in which case the transformations are 54x54. Now, the transformation matricies are permutation matricies, since they swap entries of the state vector. A permutation matrix is defined as a matrix with exactly one 1 in each row and each column, and can be generated by permuting the rows of the identity matrix. Let $\mathbf{e}_j$ be the jth row vector of the identity matrix, *not* the more standard column. Given a permutation π:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 2 & 5 & 3 \end{pmatrix}$$

The permutation matrix P_π permutes the vector $\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$ such that $P_\pi \mathbf{x} = \begin{bmatrix} 1 \\ 4 \\ 2 \\ 5 \\ 3 \end{bmatrix}$.

$$P_\pi = \begin{bmatrix} \mathbf{e}_{\pi(1)} \\ \mathbf{e}_{\pi(2)} \\ \vdots \\ \mathbf{e}_{\pi(m)} \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_4 \\ \mathbf{e}_2 \\ \mathbf{e}_5 \\ \mathbf{e}_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The way to understand this is that $\mathbf{e}_j \cdot \mathbf{x}$ gives the jth entry in $\mathbf{x}$. Thus, to match the permutation it suffices to add corresponding $\mathbf{e}_{\pi(j)}$ rows to P, one by one.

3.2 Application to Cubing

First we need to generate the permutation matricies from a cube with completely distinct stickers, not the standard solved cube (because it's not possible to infer all the sticker swaps from a solved cube - doing a U move swaps the white stickers, but it's impossible to tell). All code is given here.

```
def import_cube(data: list) -> cube.Cube:
    """ Returns a cube, given a length 54 list. """
    c = [cube.list_mat(data[:9])] + \
        [[data[i:i + 3] for i in range(j, 43, 12)] for j in range(9, 21, 3)] + \
        [cube.list_mat(data[-9:])]

    obj = cube.Cube()
    obj.cube = cube.str_cubies(c)
    return obj

def distinct_cube() -> cube.Cube:
    """ Returns a cube with distinct ids for each sticker. """
    return import_cube(list(range(54)))
```

Then, we can flatten the cube into a 54x1 vector and generate the equivalent permutation matrix from the vector.

```
def flatten(c: cube.Cube) -> np.array:
    """ Flatten a cube into a 54x1 vector. """
    return np.array([x for row in np.array(c.to_face()) for x in row])

def perm_mat(perm: list) -> np.array:
    """ Generates a permutation matrix, assuming the standard is 0, 1, ... n."""
    n = len(perm)
    M, I = np.identity(n), np.identity(n)
    for i in range(n):
        M[i] = I[perm[i]]
    return M
```

However, note that when I create a cube object from a 54x1 vector, it natually permutes the order, because I read in this format:

```
       W W W
       W W W
       W W W
O O O  G G G  R R R  B B B
O O O  G G G  R R R  B B B
O O O  G G G  R R R  B B B
       Y Y Y
       Y Y Y
       Y Y Y
```

To account for this, I define a matrix M which encapsulates the reading permutation.

```
# permutation from importing
M = np.linalg.inv(perm_mat(flatten(distinct_cube())))
```

M is the inverse of the permutation matrix of the just reading in the cube.

```
c = distinct_cube()

x = flatten(c)
c.turn("R")
y = flatten(c)
R = perm_mat(y)

T = R @ M

assert np.array_equal(T @ x, y)
```

The transformation matrix T now encapsulates the R move, and it's equal to R times M in order to cancel out the natural permutation of reading in the cube.

To make a move with this new matrix,

```
c = cube.Cube()
x = flatten(c)
c = import_cube(M @ T @ x)
```

Again, multiplying by M cancels the transformation in import_cube.

Now, we can generate the 18 standard moves (U, D, F, B, R, L and their double moves and inverses). To generate the double moves for a move A, compute A^2 and to calculate the inverse compute $A^{-1} = A^T$.

```python
def gen_moves() -> dict:
    """ Generates a move dictionary for the 18 standard moves. """
    d = {}
    c = distinct_cube()
    for move in cube.MOVES:
        c.turn(move)
        P = perm_mat(flatten(c))
        d[move] = P @ M
        d[move + "'"] = d[move].T
        d[move + "2"] = d[move] @ d[move]
        c.turn(move + "'")
    return d
```

As mentioned earlier, a sequence of moves can become a single matrix. If applying R U R' U' with matrix multiplication, then the actual order will be $U^{-1}(R^{-1}(U(R\mathbf{x})))$ and by matrix multiplication associativity we can compute $U^{-1}R^{-1}UR$.

```python
def move_mat(seq: str) -> np.array:
    """ Turns a sequence of moves into a transformation matrix. """
    T = np.identity(54)
    for move in cube.tokenize(seq):
        T = moves[move] @ T
    return T
```

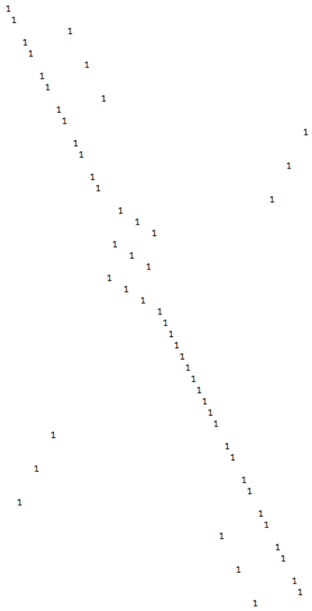
To apply a transformation, we use the logic given earlier.

```python
def apply(T: np.array, x: cube.Cube) -> cube.Cube:
    """ Applies the given transformation matrix to the cube. """
    return import_cube(M @ T @ flatten(x))
```

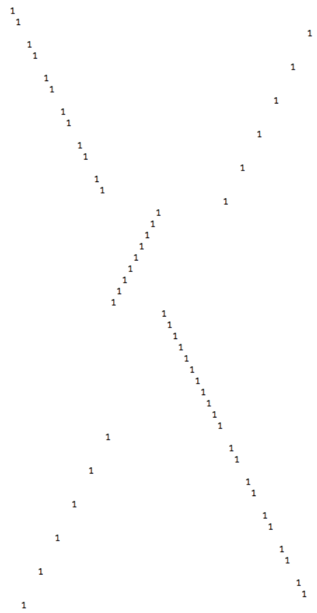
Also, we can draw pretty pictures by rendering only the ones of the matrix.

```python
def pretty(T: np.array) -> str:
    """ Pretty-formats a transformation matrix. """
    return "\n".join("".join(str(int(x)) for x in row).replace("0", " ") for row in T)
```

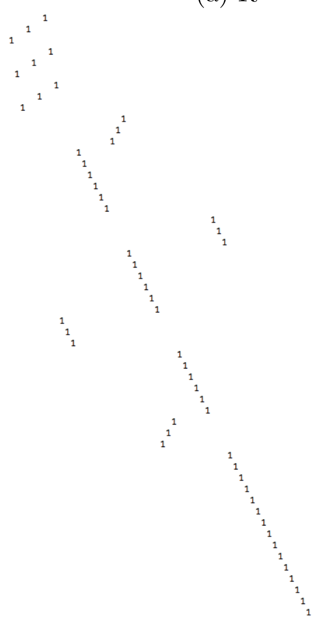
3.3 Pictures



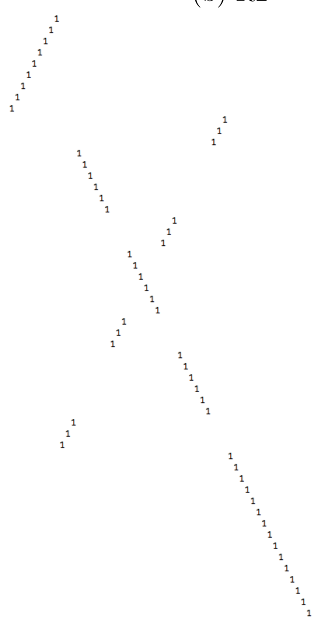
(a) R



(b) R2

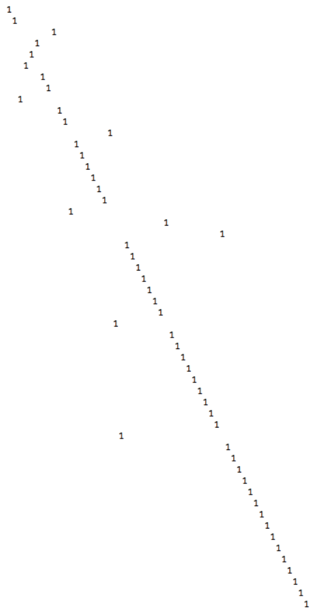


(c) U

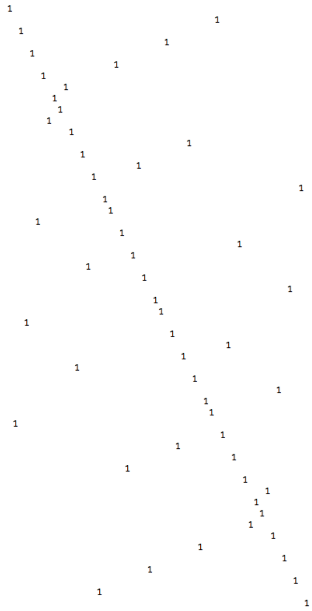


(d) U2

Figure 1: Matricies for selected standard moves



(a) T-perm



(b) Superflip

Figure 2: Matricies for selected algorithms

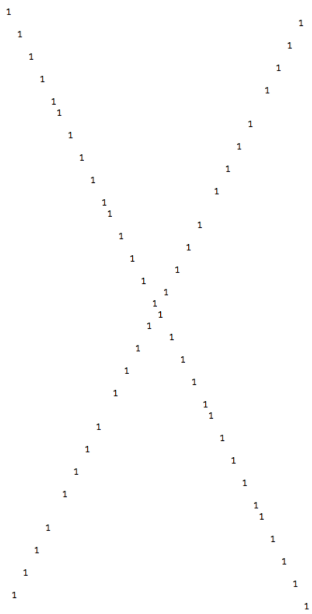


Figure 3: Checkerboard

Personally, checkerboard is my favorite.

3.4 Miscellaneous Uses

What state is the cube in after U100000? Since 100000 is divisable by 4, the cube must be solved. In general, each series of moves has a given "order", the the amount of times it can be repeated before the cube is solved again. One application of matricies is to compute these orders quickly, as matrix exponentation can be done quickly with repeated squaring.

```
def mat_exp(A: np.array, k: int) -> list:
    """ Does fast matrix exponentation """
    v = np.identity(len(A))
    while k > 0:
        if k & 1 == 1:
            v = v @ A
        k >>= 1
        A = A @ A
    return v

# largest group
T = move_mat("R U2 D' B D'")
print(apply(mat_exp(T, 1260), c))

# what's the state after (R U F L B)100000?
T = move_mat("R U F L B")
print(apply(mat_exp(T, 100000), c))
```

4 Mathematical Proofs

Lemma 1: The transpose of a permutation matrix is its inverse.
Proof: By definition of the inverse, $AA^{-1} = I$. Thus, it suffices to calculate AA^T. Think of this product as the sum of the outer products between each column in A and each row in A^T.

$$AA^T = \begin{bmatrix} col_1(A) & col_2(A) & \dots & col_n(A) \end{bmatrix} \begin{bmatrix} row_1(A^T) \\ row_2(A^T) \\ \dots \\ row_n(A^T) \end{bmatrix}$$

$$= col_1(A)row_1(A^T) + \cdots + col_n(A)row_n(A^T)$$

Each $col_j(A)row_j(A^T)$ product yields a matrix of all 0's except for a single 1 at the index where there is a 1 in $col_j(A)$. Adding them up yields a matrix which has a 1 on each entry on the diagonal (because each 1 position in the columns of A is distinct).

Thus, $col_1(A)row_1(A^T) + \cdots + col_n(A)row_n(A^T) = I_n$
Collorary to lemma 1: Because every matrix can be transposed, every permutation matrix is invertible.

Lemma 2: The mutiplication of permutation matricies is closed, that is, for a permutation matrix A and permutation matrix B, AB is a permutation matrix.
Proof: Multiplying a matrix by a permutation matrix is equivalent to permuting the rows of the matrix. If the rows are swapped, then they must still have exactly one 1 in each row. Since swapping rows doesn't change the number of 1's in each column, each column must also still have exactly one 1. Thus, the product is a permutation matrix.

Lemma 3: There are a finite number of permutation matricies, namely, there are $N!$ permutation matricies for matricies of size NxN.
Proof: Each permutation matrix is uniquely determined by π. There are $N!$ possible π's, so there are $N!$ permutation matricies.

Lemma 4: $(P^k)^{-1} = (P^{-1})^k$
Proof: $(A_1A_2\dots A_n)^{-1} = A_n^{-1}A_{n-1}^{-1}\dots A_1^{-1}$
$(PP\dots P)^{-1} = P^{-1}P^{-1}\dots P^{-1} = (P^{-1})^k$

Theorem 1: For every permutation matrix P, there exists a k such that $P^k = I$.
Proof: By lemma 2, each power of P must be a permutation matrix. By lemma 3, there are a finite number of permutation matrices, thus there must be point x where $\forall z \geq x$ $P^z = P^y$ for some $y < x$. Picking a particular $z > x$, if $P^z = P^y$, then $P^z(P^y)^{-1} = I$ so $P^z(P^{-1})^y = I$, thus $P^{z-y} = I$, completing the proof.

This proves that repeating any sequence of moves on a solved Rubik's cube will bring the cube backed to a solved state eventually.

Theorem 2: The inverse of a series of moves is the the series formed by reversing the original series, inversing each move.
Proof: Let $S = A_1A_2\dots A_n$. $S^{-1} = A_n^{-1}A_{n-1}^{-1}\dots A_1^{-1}$.

Theorem 3: NISS. The "scramble" and "solution" lie in a cycle.
Proof: Let a scramble be $ABCD$ and a solution be $pqrs$. By the definition of a solution, $ABCDpqrs = I$. $sABCDpqrs = s$ and $sABCDpqrss^{-1} = ss^{-1}$, so $sABCDpqr = I$. Repeating these operations,

$$ABCDpqrs = I$$
$$sABCDpqr = I$$
$$rsABCDpq = I$$
$$qrsABCDp = I$$
$$pqrsABCD = I$$
$$\dots$$

Thus, one can either think of the scramble as $ABCD$ and the solution as $pqrs$, or the scramble as $qrsA$ and the solution as $BCDp$, or any variant, so as long the scramble plus the solution is a cyclic rotation of the original scramble and solution.

5 Future Work

Parity derivations, group theory tie-ins, and eigenvalue stuff! Maybe a faster solving algorithm eventually.