Numerical Linear Algebra

Alex Zhou

April 2019

1 Introduction

We consider algorithms for computing algebraic invariants for linear maps in a vector over a field F. In particular, we are interested in $F = GF(p)$, the finite or Galois field of p elements, represented by integer modulo p for some prime number p.

2 Division

A useful procedure in a finite field $GF(p)$ for prime number p is finding the multiplicative inverse of a number. An implementation in Python is written below. Note that we should memorise the results to reduce computational burden for later use.

```
def find_inverses(p):
    if not is_prime(p):
        raise Exception("p is not prime")
    inverses = [0] * p # leave 0 alone
    inverses[1] = 1 # set 1 to 1
    for a in range(2, p):
        inverses[a] = -(p // a) * inverses[p % a] % p
    return inverses
```

We use the extended Euclidean algorithm recursively to compute the inverses. We express $ax + py = 1$, where $ax = 1 \pmod{p}$. We build x which is the inverse of a using smaller numbers' inverses. Notice that

$$p = \left\lfloor \frac{p}{a} \right\rfloor a + (p \bmod a),$$

so we can relate the inverse of a to the inverse of a strictly smaller number $p \bmod a$. Explicitly, we have $p = qa + r$ for $q = \left\lfloor \frac{p}{a} \right\rfloor$ and $r = p \bmod a$. Then

$$r = -qa \pmod{p} \quad \Rightarrow \quad rr^{-1}a^{-1} = a^{-1} = -qr^{-1} \pmod{p},$$

which gives the recurrence

$$a^{-1} = -\left\lfloor \frac{p}{a} \right\rfloor (p \bmod a)^{-1} \pmod{p}.$$

We tabulate the test output for $p = 11$

```
[Output]
    a    = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    a^-1 = [0, 1, 6, 4, 3, 9, 2, 8, 7, 5, 10]
```

If we instead iterate through the list of numbers and naively check each a with each b, then this has time complexity $O(p^2)$. One improvement is to use the extended Euclidean algorithm or Fermat's little theorem and iterate over the list. These methods have $O(p \log p)$ time complexity, as Euclidean division and modular exponentiation both are logarithmic. Our algorithm is in fact linear $O(p)$ as it uses the existing lower value inverses to compute successive inverses. Python actually has an inbuilt pow function which takes a base, an exponent (which is -1 in our case) and a modulus.

3 Gaussian Elimination

Recall that a matrix $M = (m_{ij}) \in F_{m,n}$ with m rows and n columns is in reduced row echelon form if

- For some $1 \leq r \leq m$, the last $m - r$ rows have zero entries.
- For each $1 \leq i \leq r$, there is a number $1 \leq l(i) \leq n$ such that $m_{ij} = 0$ for $j < l(i)$ and $m_{ij} = 1$ for $j = l(i)$.
- For $1 \leq i_1 < i_2 \leq r$, we have $l(i_1) < l(i_2)$.
- for each $2 \leq k \leq r$, we have $m_{ij} = 0$ when $j = l(k)$ and $i < k$.

The rank of such a matrix is the dimension of the row-space which is r. The following operations leave the row-space unaltered

1. $T(i, j)$, transpose rows i and j;
2. $D(i, a)$, divide row i by $a \in F - \{0\}$;
3. $S(i, a, j)$, subtract $a \in F - \{0\}$ times row $j \neq i$ from row i.

The purpose of Gaussian elimination is to use these three operations to transform any arbitrary matrix into reduced row echelon form. We can see a relatively simple algorithm for Gaussian elimination modulo p which is adapted from usual Gaussian elimination in Python below:

```python
def gauss_elim(M, p):
    n_rows, n_cols = M.shape
    M = M.copy() % p
    h = k = 0
    inverses = find_inverses(p)

    while h < n_rows and k < n_cols:
        pivot_row = -1
        for row in range(h, n_rows): # Find the pivot row
            if M[row, k] % p != 0:
                pivot_row = row
                break
        if pivot_row == -1: # No pivot found in column
            k += 1
            continue

        if pivot_row != h: # Swap current row with pivot row
            M[[h, pivot_row]] = M[[pivot_row, h]]

        # Scale the pivot row
        pivot = M[h, k] % p
        M[h] = (inverses[pivot] * M[h]) % p

        for row in range(n_rows): # Eliminate all the other rows
            if row != h and M[row, k] % p != 0:
                M[row] = (M[row] - M[row, k] * M[h]) % p
        h += 1
        k += 1
    return M
```

Using this algorithm on the matrices,

$$A_1 = \begin{pmatrix} 11 & 1 & 7 & 2 & 0 \\ 8 & 0 & 2 & 5 & 11 \\ 2 & 1 & 2 & 6 & 5 \\ 7 & 4 & 5 & 3 & 1 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 0 & 1 & 1 & 3 & 5 & 2 \\ 1 & 2 & 3 & 8 & 9 & 0 \\ 0 & 1 & 1 & 2 & 3 & 2 \\ 2 & 1 & 3 & 7 & 9 & 1 \\ 2 & 1 & 3 & 8 & 10 & 0 \end{pmatrix},$$

we have the reduced row echelon forms:

```
[Output]
    ref(A_1, 5)     ref(A_1, 11)    ref(A_2, 23)
    [1 0 0 4 0]     [1 0 3 0 0]     [1 0 1 0 0 0]
    [0 1 0 0 4]     [0 1 7 0 0]     [0 1 1 0 0 0]
    [0 0 1 4 3]     [0 0 0 1 0]     [0 0 0 1 0 0]
    [0 0 0 0 0]     [0 0 0 0 1]     [0 0 0 0 1 0]
                                    [0 0 0 0 0 1]
```

Hence, the ranks are 3, 4 and 5 and their non-zero rows form a basis for their row-spaces respectively. This is in part due to the fact that elementary row operations do not alter the row-space and also the fact that the dimension of the row-space is equal to the dimension of the column-space which is the dimension of the image, namely the rank.

4 Kernels

Let A be an m by n matrix and $x = (x_j)$ be a column vector over F. The kernel or null-space of A is the space of solutions to $Ax = 0$. Note that the kernel is unchanged when applying elementary row operations, since they correspond to multiplying by an invertible matrix (row operations are clearly invertible). Therefore, a basis for the kernel can be found by putting A into reduced row echelon form and then expressing $x_{l(1)}, \dots, x_{l(r)}$ in terms of the other columns x_j which correspond to the free variables. This algorithm is realised in Python below:

```python
def null_basis(M, p):
    n_rows, n_cols = M.shape
    M_rref = gauss_elim(M, p)
    rows_to_pivot = {}
    # Find the basic columns
    for row in range(n_rows):
        if np.any(M_rref[row]): # Skip zero rows
            for col in range(n_cols):
                if M_rref[row, col] == 1 and all(M_rref[r, col] == 0 for r in range(row + 1,
                    rows_to_pivot[row] = col
                    break

    # Basic and free columns are complementary
    basic_cols = set(rows_to_pivot.values())
    free_cols = set(range(n_cols)) - basic_cols

    kernel = []
    # Create a vector for each free column
    for free_var in free_cols:
        vec = np.zeros(n_cols, dtype = int)
        vec[free_var] = 1 # Set free varibles to 1
        # Compute dependent variables based on equations
        for row, pivot_col in rows_to_pivot.items():
            # Calculate sum of the other terms in this row
            val = sum((M_rref[row, col] * vec[col]) % p for col in range(n_cols)
                      if col != pivot_col and M_rref[row, col] != 0)
            vec[pivot_col] = (-val) % p
        kernel.append(vec)
    return kernel
```

The bases for A_1 modulo 5, 7 and 13 are as follows:

```
[Output]
    p = 5   [1, 0, 1, 1, 0], [0, 1, 2, 0, 1]
    p = 7   [5, 1, 6, 0, 1]
    p = 13  [5, 9, 11, 1, 1]
```

Running the algorithm for A_2 at every prime $p < 30$, we obtain:

```
[Output]
    2   [1,  1,  1, 0, 0, 0]    13  [12, 12, 1, 0, 0, 0]
    3   [2,  2,  1, 0, 0, 0]    17  [16, 16, 1, 0, 0, 0]
    5   [4,  4,  1, 0, 0, 0]    19  [18, 18, 1, 0, 0, 0]
    7   [6,  6,  1, 0, 0, 0]    23  [22, 22, 1, 0, 0, 0]
    11  [10, 10, 1, 0, 0, 0]    29  [28, 28, 1, 0, 0, 0]
```

which seems to suggest the pattern that $\ker(A_2)_{F_p} = \langle (p-1, p-1, 1, 0, 0, 0)^\top \rangle$.

5 Annihilators

Let U be the subspace of the space of row vectors F^n. The annihilator U° consists of the set of column vectors x satisfying $ux = 0$ for every $u \in U$. It is therefore a subspace of the space of column vectors. Incidentally, if U is the row-space of a matrix A, then U° is precisely the kernel of A. Conversely, given a matrix whose rows form a basis of U, since the dimensions of the row-space and column-space coincide, the dimension of U is equal to the rank of A). Therefore U° has dimension equal to the nullity of A. By the rank-nullity theorem,

$$\dim(U) + \dim(U^\circ) = n.$$

Similarly, if S is a subspace of the space of column vectors, then we make the analogous definition of S° as the space of row vectors t satisfying $ts = 0$ for every $s \in S$. Certainly, by definition we have $(U^\circ)^\circ = U$. Let us verify this fact by computing U° and $U^\circ)^\circ$ where U is the row-space of A_1 in the finite field $GF(21)$. The row-space is unchanged under elementary row operations, hence

$$\operatorname{rref}(A_1) = \begin{pmatrix} 1 & 0 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 & 14 \\ 0 & 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 1 & 5 \end{pmatrix} \pmod{23}$$

has the kernel spanned by $(15, 9, 16, 18, 1)^\top$. Taking this as a matrix with one row which spans U°, we perform Gaussian elimination to obtain

$$\begin{pmatrix} 1 & 19 & 21 & 15 & 20 \end{pmatrix}.$$

A basis for the kernel of this matrix and therefore for $(U^\circ)^\circ$ is

$$(4, 1, 0, 0, 0)^\top, (2, 0, 1, 0, 0)^\top, (8, 0, 0, 1, 0)^\top, (3, 0, 0, 0, 1)^\top.$$

Finally, by constructing a matrix with these vectors as rows and performing Gaussian elimination gives

$$B_1 = \begin{pmatrix} 4 & 1 & 0 & 0 & 8 \\ 2 & 0 & 1 & 0 & 0 \\ 8 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathrm{rref}(B_1) = \begin{pmatrix} 1 & 0 & 0 & 0 & 8 \\ 0 & 1 & 0 & 0 & 14 \\ 0 & 0 & 1 & 0 & 7 \\ 0 & 0 & 0 & 1 & 5 \end{pmatrix} \pmod{23},$$

which is the same as $\mathrm{rref}(A_1)$ as desired.

6 Intersections and Sums of Annihilators

Let U and W be two subspaces of F^n. It is known that

$$(U+W)^\circ = U^\circ \cap V^\circ \quad \text{and} \quad (U \cap W)^\circ = U^\circ + V^\circ.$$

Given matrices A and B with row-spaces U and V, we want to compute the bases for U, V, $U+V$ and $U \cap V$. The bases for U and V can be computed as before, by taking the non-zero rows in the reduced row echelon form. Also, $U+V$ is spanned by the union of bases of U and V, so a basis can be computed by considering the matrix

$$\begin{pmatrix} A \\ B \end{pmatrix}$$

and again, performing Gaussian elimination. The intersection of two bases is harder to compute, but we can make use of the second identity from above . Then

$$U \cap W = ((U \cap W)^\circ)^\circ = (U^\circ + V^\circ)^\circ.$$

So we proceed as follows: compute bases for the kernels of A and B; take both collection of vectors as the rows of a matrix N which we put into reduced row echelon form; finally compute a basis of the kernel of the non-zero rows of N.

```
def row_basis(M, p):
    n_rows, n_cols = M.shape
    M_rref = gauss_elim(M, p)
    coimage = []
    for row in range(n_rows):
        if np.any(M_rref[row]):
            coimage.append(M_rref[row])
    return coimage

def sum_basis(M, N, p):
    if M.shape[1] != N.shape[1]:
        raise Exception("Row vectors of M and N do not have the same length")
    S = np.concatenate((M, N), axis = 0)
    return row_basis(S, p)

```

```python
def intersect_basis(M, N, p):
    M_rref = gauss_elim(M, p)
    N_rref = gauss_elim(N, p)
    ker_M = np.row_stack(null_basis(M_rref, p))
    ker_N = np.row_stack(null_basis(N_rref, p))
    sum_ker = np.row_stack(sum_basis(ker_M, ker_N, p))
    return null_basis(sum_ker, p)
```

Let us compute bases for U, W, $U+W$ and $U \cap W$ where U and W are the row-space and kernel (transposed) of the following matrix:

$$A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 5 & 0 & 1 & 6 & 3 & 0 \\ 0 & 0 & 5 & 0 & 2 & 0 & 0 \\ 2 & 4 & 0 & 0 & 0 & 5 & 1 \\ 4 & 3 & 0 & 0 & 6 & 2 & 6 \end{pmatrix},$$

modulo 19 and 7.

```
[Output]
            p = 19                          p = 7
    U       (1, 0, 0, 0, 0, 6, 1),          (1, 0, 0, 0, 0, 0, 0),
            (0, 1, 0, 0  0, 3, 14),         (0, 1, 0, 0, 0, 3, 2),
            (0, 0, 1, 0, 0, 16, 9),         (0, 0, 1, 0, 0, 0, 0),
            (0, 0, 0, 1, 0, 0, 8),          (0, 0, 0, 1, 0, 2, 4),
            (0, 0, 0, 0, 1, 17, 6)          (0, 0, 0, 0, 1, 0, 0)

    W       (1, 0, 9, 18, 6, 1, 12),        (0, 1, 0, 0, 0, 3, 2),
            (0, 1, 0, 2, 0, 4, 14)          (0, 0, 0, 1, 0, 2, 4)

    U+W     (1, 0, 0, 0, 0, 0, 0),          (1, 0, 0, 0, 0, 0, 0),
            (0, 1, 0, 0, 0, 0, 0),          (0, 1, 0, 0, 0, 3, 2),
            (0, 0, 1, 0, 0, 0, 0),          (0, 0, 1, 0, 0, 0, 0),
            (0, 0, 0, 1, 0, 0, 0),          (0, 0, 0, 1, 0, 2, 4),
            (0, 0, 0, 0, 1, 0, 0),          (0, 0, 0, 0, 1, 0, 0)
            (0, 0, 0, 0, 0, 1, 0),
            (0, 0, 0, 0, 0, 0, 1)

    U^W     Empty                           (0, 4, 0, 5, 0, 1, 0),
                                            (0, 5, 0, 3, 0, 0, 1)
```

This matches the expected relationship between the dimensions of the spaces,

$$\dim(U+W) = \dim(U) + \dim(W) - \dim(U \cap W).$$

In the modulo 19 case, $U+W$ has full rank with trivial $U \cap W$, hence

$$\operatorname{im}(A_3^\top) \oplus \ker(A_3) = GF(19).$$

Under the field of real or complex numbers endowed with the standard inner product $\langle \cdot, \cdot \rangle$, this holds for any matrix A. For $v \in \ker(A)$, we have

$$\langle r_1, v \rangle, \ldots, \langle r_n, v \rangle = 0$$

so v is orthogonal to every row of A, which means that $\ker(A)$ is orthogonal to $\mathrm{im}(A)$. Therefore, the sum is equal to F^n and their intersection is trivial. However, this conclusion is not necessarily correct for a finite field $GF(p)$, as demonstrated in the modulo 7 case. Orthogonality requires an extra inner product structure on the vector space and more specifically, there is no natural way to satisfy positive definiteness on $GF(p)$.

7 Linear Algebra in Python

Python has a whole wealth of linear algebra libraries that are available. Some commonly used functions in NumPy are linalg.solve to solve a linear system $Ax = y$; linalg.inv the find an inverse A^{-1}; and eig to find eigenvalues and eigenvectors. SciPy also contains many decomposition algorithms such as LU, QR and SV decompositions. The following is a demonstration of some of these functions:

```python
import numpy as np

A = np.array([[4, 3, -5],
              [-2, -4, 5],
              [8, 8, 0]])
y = np.array([2, 5, -3])
x = np.linalg.solve(A, y)

A_inv = np.linalg.inv(A)
x = np.dot(A_inv, y)

a = np.array([[2, 2, 4],
          [1, 3, 5],
          [2, 3, 4]])
w, v = eig(a)
```

with the following output:

```
[Output]
    [2.20833333 -2.58333333 -0.18333333]
    [2.20833333 -2.58333333 -0.18333333]
    E-value: [-1. 4.]
    E-vector [-0.89442719 -0.4472136 ], [0.4472136 -0.89442719]
```