Computing quotients of finitely generated abelian groups

Fredrik Meyer

April 25, 2016

Abstract

Say you are given an integer matrix A. This correspond to a map $\mathbb{Z}^r \xrightarrow{A} \mathbb{Z}^s$, and the cokernel is a finitely-generated abelian group. How do you see which one?

This is a short tutorial on how to do this in practice (with the help of Python in large examples).

1 The problem

Say you are given an integer matrix $A \in M_{n\times m}(\mathbb{Z})$. This correspond to a linear map from $\mathbb{Z}^n$ to $\mathbb{Z}^m$. One often wants to compute the cokernel, which measures the extent to which the map is *not* surjective. By definition the cokernel is $\mathbb{Z}^m/\operatorname{im} A$.

It is not always easy to see what the cokernel should be. For example, given the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},$$

is it obvious that the cokernel is $\mathbb{Z}/2$? What would be nice, is if the matrix were in a nicer form, such as

$$A' = \begin{pmatrix} 1 & 2 \\ 0 & 2 \end{pmatrix}.$$

In this case, it is quite apparent that the image is $\mathbb{Z} \oplus 2 \cdot (1,1)$. We note that we can change basis of $\mathbb{Z}^2$ by matrices in $\mathrm{SL}_2(\mathbb{Z})$, and that the matrix

$$S = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

send the vectors $(1,0)$ and $(1,1)$ to $(1,0)$ and $(0,1)$. This shows that the image of the previous matrix is $\mathrm{SL}_2(\mathbb{Z})$-equivalent to $\mathbb{Z} \oplus 2\mathbb{Z}$. Hence the quotient is $\mathbb{Z}/2$.

It is a general fact that if the matrix is in upper diagonal form, then one can read off the quotient from the diagonal entries, because basises constructed from such matrices are always $\mathrm{SL}_2(\mathbb{Z})$-equivalent to the standard basis.

Hence we would want a process to change basis such that the representing matrix is in upper diagonal form. This is possible by performing row operations involving only elements of $\mathrm{SL}_2(\mathbb{Z})$. This correspond to left-multiplying the matrix A by an elementary matrix E, which again correspond to changing the basis of the target space.

Remark. *By also allowing column operations, one can get the matrix on diagonal form, but this is not necessary to read off the quotient (but it can be useful to read off the kernel). By the way: note that the kernel is always $\mathbb{Z}^k$ for some $k \geq n$: there is no torsion factor. This is because the kernel is a subgroup of $\mathbb{Z}^n$, and $\mathbb{Z}^n$ has no torsion.*

2 The solution

Here we present an algorithm to transform a matrix A into what is sometimes called the (nonreduced) Hermite normal form of a matrix. We say nonreduced, because it is possible to further simplify the matrix by similar operations, but this is not necessary to spot the cokernel.

Below we present an algorithm. The given data is an integer matrix A.

1. Find the smallest non-zero entry a in the first column of A. By row operations, move the corresponding row to the top. If there are no such entries, proceed to step 3.

2. For each row not equal to the top row, let d be the greatest common divisor of the first entry in this row (call this b) and a. Then there is a relation $xa + yb = d$ (Bezout), which can be used to eliminate this entry (see example below for explanation). Repeat.

3. Now we do Step 1 with the submatrix obtained by removing the column and leftmost row from A. In the end we get a 1×1-matrix, which is the base case.

The second step is a bit vague because it is easier to explain with an example. Consider the 2×2-matrix

$$A = \begin{pmatrix} 12 & 2 \\ 18 & 5 \end{pmatrix}.$$

The greatest common divisor between 12 and 18 is 6, and we have a Bezout relation $6 = 12 \cdot (-1) + 18 \cdot 1$. Now consider the matrix

$$S = \begin{pmatrix} -1 & 1 \\ -3 & 2 \end{pmatrix}.$$

The first row are the coefficients in the Bezout relation, and the second are a "divided Koszul relation": it is $18/6$ and $12/6$. More generally, if we have a matrix

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

then let $D = xa + yc$ be a Bezout relation. Then form the matrix

$$S = \begin{pmatrix} x & y \\ -c/D & a/D \end{pmatrix}.$$

Then SA is

$$\begin{pmatrix} D & bx + dy \\ 0 & (ad - bc)/D \end{pmatrix}.$$

Note that S has determinant 1 by construction. This generalizes to larger matrices, except that it's harder to keep track of indices.

By repeated application of this process, we get a upper diagonal matrix. Then, as noted above, the cokernel can be read off as $\mathbb{Z}/d_1 \oplus \ldots \oplus \mathbb{Z}/d_m$.

3 Implementation in Python

It is possible to work with matrices in the `numpy` library in Python, but it uses only floating point numbers. Since we are working with matrices over the integers, we implement our own matrix class instead. The code for the implementation is in the Appendix A.

Here is the code for the Python function taking an integer matrix as input, and returning a upper diagonal form.

The implementation is recursive, starting with the base case of a 1×1-matrix.

```
def triangular(M):
        '''
        Input: an integer matrix M.
        Output: an upper triangulization
               U of M over the integers.
        '''
        if len(M.L) == 1:
                if M.L[0][0] < 0:
                        return -M
                return M
        (index, smallest) = (0,M.L[0][0])
        for i in range(len(M.L)):
                if abs(M.L[i][0])  < smallest:
                        (index, smallest) = (i,M.L[i][0])
        if smallest == 0:
                return concat(M,triangular(submatrix(M)))
        if index != 0:
                N = M.switchRows(0,index)
        else:
                N = Matrix(M.L)
        for i in range(1,len(M.L)):
                d = ntheory.gcd(smallest, N.L[i][0])
                bez = ntheory.bezout(smallest, N.L[i][0])
                I = identity(len(N.L))
                I.L[0][0] = bez[0]
                I.L[0][i] = bez[1]
                I.L[i][0] = N.L[i][0]/d
                I.L[i][i] = -N.L[0][0]/d
                N = I * N
```

```
                if N.L[0][0] < 0:
                        N = N.mult(-1)
        return concat(N,triangular(submatrix(N)))
```

Try to compare what the function does with the algorithm above. Here is an example output of the above function:

```
>>> M = Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> print M
3x3-matrix: [1, 2, 3]
            [4, 5, 6]
            [7, 8, 9].
>>> print triangular(M)
3x3-matrix: [1, 2, 3]
            [0, 3, 6]
            [0, 0, 0].
```

Hence the cokernel in this case is $\mathbb{Z}/3 \oplus \mathbb{Z}$.

A Implementation of integer matrices in Python

Note that many functions are not yet written. Perhaps it is better to just Google my git account for more readable code.

```
import ntheory
from operator import mul

class Matrix:
        """
        Construct a matrix object from a double list.
        """
        def __init__(self, L):
                self.L = L
                self.m = len(L[0]) # number of columns
                self.n = len(L) # number of rows

        def checkFormat(self,L):
                '''
                To be written. Checks if the matrix is
                well-defined.
                '''
                return True
```

```
    def __add__(self,M):
        '''
        Returns the sum of self and M.
        '''
        newL = []
        for r in range(self.m):
            row = [self.L[r][i] + M.L[r][i]
                   for i in range(self.n)]
            newL += [row]
        return Matrix(newL)

    def __sub__(self,N):
        return (self + (-N))

    def __neg__(self):
        newL = [[-r for r in R] for R in self.L]
        return Matrix(newL)

    def __mul__(self,N):
        '''
        Returns the product of self and N.
        '''
        NT = N.transpose()
        newL = []
        for i in range(self.n):
            newR = []
            for j in range(N.m):
                newR += [sum([self.L[i][k]*NT.L[j][k]
                              for k in range(self.m)])]
            newL += [newR]
        return Matrix(newL)

    def transpose(self):
        '''
        Returns the transpose of self.
        '''
        newL = [[] for i in range(self.m)]
        for i in range(len(self.L)):
            for j in range(self.m):
```

```
                        newL[j] += [self.L[i][j]]
        return Matrix(newL)

def trace(self):
        '''
        If self is square, return trace.
        '''
        if self.n != self.m:
                return "NOT SQUARE"
        return sum([self.L[i][i]
                    for i in range(self.n)])

def mult(self,c):
        newL = [[c*r for r in R] for R in self.L]
        return Matrix(newL)

def switchRows(self,i,j):
        '''
        Returns the matrix obtained by
        switching rows i,j in self.
        Counting starts at 0.
        '''
        newL = list(self.L)
        newL[i], newL[j] = newL[j], newL[i]
        return Matrix(newL)

def multRow(self,i,c = -1):
        '''
        Multiplies row i with c. (plus minus 1)
        '''
        newL = list(self.L)
        newL[i] = [c*r for r in newL[i]]
        return Matrix(newL)

def addRow(self,i,j,a=1):
        '''
        Adds a times row j to row i.
        '''
        newL = list(self.L)
        rowj = list(self.L[j])
```

```
                newL[i] = [self.L[i][j]+a*rowj[j]
                           for j in range(len(rowj))]
                return Matrix(newL)

        def _prodDiagonal(self):
                return reduce(mul,[self.L[i][i]
                            for i in range(len(self.L))],1)

        def det(self):
                return triangular(self)._prodDiagonal()



        def __str__(self):
                s = "{0}x{1}-matrix: ".format(self.m,self.n)
                          + str(self.L[0]) + "\n"
                for row in self.L[1:]:
                         s+= 12*" " + str(row) + "\n"
                return s[:-1] + "."

def identity(n):
        L = []
        for i in range(n):
                L += [[int(j == i) for j in range(n)]]
        return Matrix(L)

def concat(M,N):
        '''
        Input: M nxm matrix.
               N n-1 x m-1 matrix.
        Output: A new matrix with Q with N the
         submatrix obtained by removing first col and row.
         '''
        L = [M.L[0]]
        for i in range(1,len(M.L)):
                R = [M.L[i][0]]
                for j in range(len(N.L[0])):
                        R += [N.L[i-1][j]]
                L += [R]
        return Matrix(L)
```

```
def submatrix(M,c=0,r=0):
        '''
        The submatrix obtained by removing col c and row r.
        '''
        L = []
        for i in range(len(M.L)):
                if i != r:
                        R = []
                        for j in range(len(M.L[0])):
                                if j != c:
                                        R += [M.L[i][j]]
                        L += [R]
        return Matrix(L)
```