Solving Simultaneous Linear Diophantine Equations in APL

Bob Smith
Sudley Place Software
Originally Written
11 Jul 2021
Updated
16 Oct 2023

Introduction

The problem posed by Eric Lescasse (through Roy Sykes[1]) back in 2018, intrigued me as initially it looked like it might be solvable through Matrix Divide. It turned out that its solution **is** related to Matrix Divide, but only when restricted to integers:

Given an integer matrix's row & column sums (each, of course, summing to the same grand total), return an integer matrix with the same row & column sums. Of course, there may be several solutions.

The "several solutions" turned out to be either the solution is unique or there are an infinite number of solutions.

Example

For example, (origin-1 and multi-precision integers throughout)

```
      ⎕←M←?4 3⍴10x
4  6  10
4  5   1
3  1   7
5  4   9
      ⎕←RS←+/M ⍝ The row sums
20 10 11
      ⎕←CS←+⌿M ⍝ The column sums
18 16 16 27
      ⎕←nRC←×/(nR nC)←⍴M ⍝ The #rows/cols/entries
12
```

Encoding The Information

Although the original matrix M is unknown to us, we can represent it symbolically as follows:

m_{11} m_{12} m_{13}
m_{21} m_{22} m_{23}
m_{31} m_{32} m_{33}
m_{41} m_{42} m_{43}

Now, the nR equations for row sums (RS) are represented as (I apologize for mixing notations, but you get the idea)

$m_{11} + m_{12} + m_{13} = RS[1]$
$m_{21} + m_{22} + m_{23} = RS[2]$
$m_{31} + m_{32} + m_{33} = RS[3]$
$m_{41} + m_{42} + m_{43} = RS[4]$

These equations can be encapsulated into a Boolean coefficient matrix of nR rows (equations) with nRC columns (unknowns):

```
      ⎕←RS_EQ←nR nRC⍴(nRC+nC)↑nC⍴1
m11 m12 m13  m21 m22 m23  m31 m32 m33  m41 m42 m43
1   1   1    0   0   0    0   0   0    0   0   0  = RS[1]
0   0   0    1   1   1    0   0   0    0   0   0  = RS[2]
0   0   0    0   0   0    1   1   1    0   0   0  = RS[3]
0   0   0    0   0   0    0   0   0    1   1   1  = RS[4]
```

Similarly, the nC equations for the column sums (CS) are

```
m11 + m21 + m31 + m41 = CS[1]
m12 + m22 + m32 + m42 = CS[2]
m13 + m23 + m33 + m43 = CS[3]
```

These equations can be encapsulated into a Boolean coefficient matrix of nC rows (equations) with nRC columns (unknowns):

```
      ⎕←CS_EQ←⊃,/nR⍴⊂∘.=⍨⍳nC
m11 m12 m13  m21 m22 m23  m31 m32 m33  m41 m42 m43
1   0   0    1   0   0    1   0   0    1   0   0   = CS[1]
0   1   0    0   1   0    0   1   0    0   1   0   = CS[2]
0   0   1    0   0   1    0   0   1    0   0   1   = CS[3]
```

Joining these two Boolean coefficient matrices along the 1st coordinate (RS_EQ⍪CS_EQ) yields nR+nC equations, each with nRC unknowns as follows (call this Boolean coefficient matrix, A):

```
1 1 1 0 0 0 0 0 0 0 0 0 = RS[1]
0 0 0 1 1 1 0 0 0 0 0 0 = RS[2]
0 0 0 0 0 0 1 1 1 0 0 0 = RS[3]
0 0 0 0 0 0 0 0 0 1 1 1 = RS[4]
1 0 0 1 0 0 1 0 0 1 0 0 = CS[1]
0 1 0 0 1 0 0 1 0 0 1 0 = CS[2]
0 0 1 0 0 1 0 0 1 0 0 1 = CS[3]
```

With C←RS,CS, the problem is to solve for the length nRC vector X in the matrix equation: C≡A+.×X, where (by definition) X←,M is always a solution, as in C≡A+.×,M.

Simultaneous Linear Diophantine Equations

Specifically, we are asking to solve a set of Simultaneous Linear Diophantine Equations (SLDEs, pronounced as “slides”) where the name Diophantine refers to the 3rd century A.D. Greek Mathematician Diophantus[2] who pioneered the study of integer solutions to integer equations.

Such equations were first solved by H.J.S. Smith in his 1861 paper[3]. In

it, he defines a Smith Normal Form (SNF) which along with two associated matrices can be used to solve SLDEs.

Calculating The Smith Normal Form

Calculation of the SNF[4] produces three matrices as follows:

```
(U B V)←⌹'s' A
```

The code behind this variant on Domino implements a multi-precision algorithm with is a proposed addition for the FLINT[Error: Reference source not found] library to calculate the two associated SNF unimodular matrices. The matrix `B` is the SNF matrix (and is diagonal only) and `U` and `V` are unimodular (integer matrices whose determinate is ±1, and are thus integer-invertible). If the shape of `A` is `m` by `n`, then the shape of `B` is the same as `A` (`m` by `n`), `U` is `m` by `m`, and `V` is `n` by `n`.

The algorithm to calculate the SNF starts with `U` and `V` as identity matrices with `B` as a copy of `A`, which preserves the identity

```
B≡U+.×A+.×V
```
(1)

The algorithm continues by using elementary row and column operations, where row operations on `B` are reflected in `U` and column operations on `B` are reflected in `V`. Throughout, the goal is to transform `B` into a diagonal matrix, and, at all stages, **preserve the above identity**.

The first part of the SNF algorithm looks very much like **Gaussian Elimination** in that its goal is to transform the lower left triangle of `B` to all zeros. However, instead of division to perform the reduction, it uses GCD because the goal is also to preserve the Diophantine nature of the matrices. Similarly, the upper right triangle of `B` is reduced to all zeros through the same type of transformations, but using column operations. In other words, one part of SNF could call a **Gaussian Elimination operator** with its own operand as opposed to (say) the code to calculate a determinant (`-.×R`) which would use a different operand.

The second part of the SNF algorithm is to ensure uniqueness by converting the diagonal entries such that they are successively divisors of the next higher entry (Pairwise Chain Divisible). That is,

```
∧/0=2|/1 1⌽B
```

At the end, the SNF matrix `B` is diagonal, meaning that all entries outside the diagonal are zero.

Starting with equation (1):

```
     B              ≡U+.×A+.×V
↔ (B+.× ⌹V)       ≡U+.×A+.×V+.×⌹V right-divide by V
↔ (B+.× ⌹V)       ≡U+.×A           identity matrix
↔ (B+.×(⌹V)+.×X)≡U+.×A+.×X        right-multiply by X
↔ (B+.×(⌹V)+.×X)≡U+.×C            because        C≡A+.×X
↔ (B+.×(⌹V)+.×X)≡D                substitute     D←U+.×C
```

It's possible for the trailing entries on the diagonal of `B` also to be zero, meaning that because `B` is central to all these calculations, whatever is multiplied by those zero entries is irrelevant. That's how we can end up with an infinite number of solutions. The count of leading non-zero entries in the diagonal of `B` is

```
fx←+/∧\0≠1 1⌽B
```

Essentially, this value counts the number of fixed unknowns, and `fr←n-fx` counts the number of free unknowns. That is, if an element on the diagonal of `B` is zero, whatever it is multiplied by has no effect on the result, and so the multiplied-by value is "free".

Now, construct the vector

```
Y←n↑(fx↑D)÷fx↑1 1⌽B
```

By construction, because `B` is a diagonal matrix

```
B+.×Y ↔ (1 1 ⌽B)×Y
```

and because of the way `Y` was constructed (the elements of `B`'s diagonal are in the denominator of `Y`)

```
D≡B+.×Y     ⍝ ⍴D ↔ n
```

but we can only be sure of this equation for the 1st `fx` entries. If any of the remaining entries in `D` (remember that `D←U+.×C`) are non-zero, (`∨/0≠fx↓D`), there are no solutions.

Moreover, this makes sense only if the entries of `Y` are all integers (`^/Y=⌊Y`); if not, there are no solutions.

Continuing the above sequences of equations

```
     B                  ≡U+.×A+.×V
↔ (B+.×  ⌹V)            ≡U+.×A+.×V+.×⌹V right-divide by V
↔ (B+.×  ⌹V)            ≡U+.×A           identity matrix
↔ (B+.×(⌹V)+.×X)≡U+.×A+.×X               right-multiply by X
↔ (B+.×(⌹V)+.×X)≡U+.×C                   because          C≡A+.×X
↔ (B+.×(⌹V)+.×X)≡D                       substitute       D←U+.×C
↔ (B+.×(⌹V)+.×X)≡B+.×Y                   substitute       D≡B+.×Y
↔ (     (⌹V)+.×X)≡Y                      left-multiply by ⌹B
↔                X ≡V+.×Y                left-multiply by V
```

Assuming there are any solutions, if there are no free unknowns (`fr=0`) the solution is unique, if there are free unknowns (`fr>0`), the number of solutions is infinite.

Solving The Set of SLDEs

The solution[6] appears on Wikipedia, and is translated into APL in the following function:

```
    ∇ (V Y fr)←C sldeSolve A;U B D fx m n
[1]   ⍝ Calculate X such that C≡A+.×X
[2]   ⍝  where ⍴C ↔ m
[3]   ⍝        ⍴A ↔ m n
[4]   ⍝        ⍴X ↔ n
[5]   (m n)←⍴A ◇ :Assert (,m)≡⍴C
[6]   (U B V)←⎕SVD 's' A ⍝ Calculate Smith Normal Form
[7]   ⍝ ⍴U ↔ m m   ⍴B ↔ m n   ⍴V ↔ n n
[8]   :Assert B≡⊃+.×/U A V
[9]   ⍝ Calculate (fx fr) ↔ # fixed/free unknowns
[10]  fx←+/∧\0≠1 1⌽B ◇ fr←n-fx
[11]  D←U+.×C ⍝ Helper matrix
[12]
[13]  Y←n↑(fx↑D)÷fx↑1 1⌽B    ⍝ ⍴Y ↔ n
[14]
[15]  ⍝ Check for solution:  Y must be all integers,
[16]  ⍝ and the <fr> trailing entries of D are all 0
[17]  ⎕ERROR((1∊Y≠⌊Y)∨1∊0≠fx↓D)/'NO SOLUTION'
[18]  :Assert D≡B+.×Y
[19]
[20]  ⍝ The solution to C≡A+.×X is X←V+.×Y
[21]  :Assert C≡A+.×V+.×Y
    ∇
```

which is called as

```
      (V Y fr)←C sldeSolve A
```

Assuming there are solutions, the vector `X←V+.×Y` solves the matrix equation `C≡A+.×X`.

Free Unknowns

If there are free unknowns, the trailing zeros in `Y` may be any arbitrary values and `V+.×Y` will still generate a solution with the same row & column sums as the input matrix. Assuming `fr>0`, the trailing `fr` elements of `Y` may be any random integers (say, `RI←?frρ¯∞`). That is,

if `V+.×Y` is a solution, so is

```
V+.×Y+(-ρY)↑RI                                    (2)
```

Because the trailing `fr` elements of `Y` are zero, the trailing `fr` columns of `V` are irrelevant, and we can break them out and re-write (2) as

```
      (V+.×Y)+((-fr)↑[2]V)+.×RI
←→          X+((-fr)↑[2]V)+.×RI      because X←V+.×Y
```

This means that because `X` is a solution to the SLDEs, so is

```
X+((-fr)↑[2]V)+.×RI
```

Because we have free variables which may assume an infinite number of values, the probability that `X` is the same as `M` is zero, although the infinite number of solutions all have the same row & column sums as `M`. We knew from the beginning that (by definition) `,M` would satisfy the original matrix equation `C≡A+.×,M`.

Even though there are an infinite number of free variables, there is one set of them that can re-create the original matrix. That is, we want to solve the matrix equation:

```
(  ,M)    ≡X+((-fr)↑[2]V)+.×RI
((,M)-X)≡   ((-fr)↑[2]V)+.×RI     Subtract X
```

where the vector `RI` is the vector of unknowns to calculate.

Just as before, when we wanted to solve for `X` in the SLDE

```
C≡A+.×X
```

we called

```
(V Y fr)←C sldeSolve A
```

In this case, we call

```
(V2 Y2 fr2)←((,M)-X) sldeSolve (-fr)↑[2]V
```

Note that this time, `fr2=0`, meaning that there no free unknowns for this solution (i.e., it's unique), and that `RI←V2+.×Y2` provides exactly the correct random integers to re-create the original matrix:

```
M≡(ρM)ρX+((-fr)↑[2]V)+.×RI
```

Note that even when the solution is unique (`fr=0` and `M≡X`), the above equation is still valid, and `sldeSolve` is passed a zero-column matrix on the right and returns `V2` a zero by zero matrix and `Y2` an empty vector. Then `V2+.×Y2` is an empty vector, multiplying it on the left by a zero-column matrix produces a vector of zeros which added to `X` leaves `X` unchanged, and matches `M`. In other words, empty arrays just work!

Applications

Chemistry[7]

- balancing chemical equations[8]
- determining the molecular formula of a compound

Transportation and Logistics

Because these industries work so often in integer units (people, trucks, boxes, warehouses, etc.), it is natural to use Diophantine equations.

Biology[9]

- analysis of S-shaped curves of the growth of microorganisms

Cryptography[10]

- elliptic curve cryptography is based on doing calculations in finite field (also called Galois fields) for a diophantine equation of degree 3 in two variables

And many, many more.

Downloads

This paper is an ongoing effort and can be out-of-date the next day. To find the most recent version, goto http://sudleyplace.com/APL/ and look for the title of this paper on that page.

The latest beta version of NARS2000 implements the Smith Normal Form primitive. It may be found online at

https://nars2000.org/download/binaries/beta/

in either 32- or 64-bit versions. The APL code above may be found in the workspace online at

http://nars2000.org/download/workspaces/SNF.ws.nars.

References

1. Personal communication, 18 Dec 2018.
2. Diophantus, Wikipedia, https://en.wikipedia.org/wiki/Diophantus
3. Smith, Henry J. Stephen, Wikipedia (1861). "On systems of linear indeterminate equations and congruences". Phil. Trans. R. Soc. Lond. 151 (1): 293–326. doi:10.1098/rstl.1861.0016. JSTOR 108738. Reprinted (pp. 367–409) in The Collected Mathematical Papers of Henry John Stephen Smith, Vol. I, edited by J. W. L. Glaisher. Oxford: Clarendon Press (1894), xcv+603 pp.
4. Smith Normal Form, Wikipedia, https://en.wikipedia.org/wiki/Smith_normal_form
5. FLINT library (https://flintlib.org) with a proposed addition written by and copyright © 2015 Alex J. Best. https://github.com/alexjbest/flint2/tree/alex/old/fmpz_mat/snf_*.c
6. Wikipedia, Systems of Linear Diophantine Equations, https://en.wikipedia.org/wiki/Diophantine_equation#System_of_linear_Diophantine_equations
7. Crocker, Roger. (1968). “Application of diophantine equations to problems in chemistry”. Journal of Chemical Education - J CHEM EDUC. 45. 10.1021/ed045p731. https://pubs.acs.org/doi/10.1021/ed045p731

8. 2019 JETIR June 2019, Volume 6, Issue 6, “Applications of diophantine equations in chemical equations” https://www.jetir.org/papers/JETIR1906I86.pdf
9. Klykov SP (2021) “Application of diophantine equations for practical problems solution in biology”. Open J Bac 5(1): 013-016. DOI: 10.17352/ojb.000019, https://www.peertechzpublications.com/articles/OJB-5-119.php
10. “Algorithmic solution of Diophantine equations and applications to cryptography II”, https://graz.elsevierpure.com/en/projects/algorithmic-solution-of-diophantine-equations-and-applications-to