Nonnegative and Positive Diophantine Solutions

John Burkardt

22 May 2020

We suppose we are given a linear Diophantine equation in n variables:

$$\sum_{i=1}^{n} a_i x_i = b$$

where the coefficients a_i and the right hand side b are assumed to be strictly positive integers. It is desired to determine all solution vectors x which are integers, and either nonnegative, or strictly positive. Under the assumptions on a and b, there can no more than finitely many nonnegative or strictly positive integer solutions. We use k to represent the number of solutions of a given problem.

This text describes a pair of MATLAB functions, with the signatures

```
x = diophantine_nd_nonnegative(a,b)
x = diophantine_nd_positive(a,b)
```

which accept the coefficient n-vector a and right hand side b of a Diophantine equation, and return the set of solutions in an $n \times k$ array x.

We regard the nonnegative solution procedure as the more fundamental of the two. If we have such a procedure, we can construct a corresponding procedure for all strictly positive solutions. A brute force approach would generate all nonnegative solutions and then discard those that are not strictly positive. But a more satisfactory approach defines an auxilliary problem to be treated by the nonnegative procedure, whose result can be transformed into a solution of the strictly positive problem.

Thus, assume for a moment that we have a nonnegative solver. Say we are given the information for a Diophantine equation, but we wish to find only the strictly positive solutions. In that case, from the data a and b, we can define a new right hand side β, and use the nonnegative solution procedure to find solution vectors χ, from which the solution set x of the original problem can be recovered.

The following algorithm suggests how this is done:

Algorithm 1 Solver for positive solutions of Diophantine equation

```
procedure DIOPHANTINE_ND_POSITIVE($a, b$)
    $n \leftarrow$ length of a
    $\beta \leftarrow b - \sum_{i=1}^{n} a_i$
    if $\beta < 0$ then
        $x \leftarrow \emptyset$
    else
        $\chi \leftarrow$ diophantine_nd_nonnegative($a, \beta$).
        for $1 \le j \le n$ do
            $x_j \leftarrow \chi_j + 1$
        end for
    end if
    return $x$
end procedure
```

Now we can concentrate on the algorithm *diophantine_nd_nonnegative()* which seeks nonnegative solutions to a Diophantine problem. Given the data a and b, we plan to construct each possible solution vector

x one component at a time. The step counter j starts at 1. We start by evaluating the "surplus", $r = b - \sum i = 1^{j-1} a_i x_j$. On step j, we assume we have already set components 1 through j. If $j < n$, then we can increment j and choose x_j to be any value between 0 and $\lfloor \frac{r}{a_j} \rfloor$, where for any real value r, the `floor()` function, $\lfloor r \rfloor$ returns the greatest integer less than or equal to r. Here, we take the greedy choice $x_j = \lfloor \frac{r}{a_j} \rfloor$, and we are ready for the next step.

However, if $j = n$, we have constructed a complete proposed solution. If the solution satisfies the equation, we add it to our list. In any case, we then search our current solution x for the largest previous index k for which x_k was nonzero. If we find such an index, we decrement x_k by 1, set $j = k$, and restart the algorithm from that index to consider the next possible solution. But if no such index k exists, then we exhausted our options and the search terminates. An outline of this algorithm follows:

Algorithm 2 Solver for nonnegative solutions of Diophantine equation

```
procedure DIOPHANTINE_ND_NONNEGATIVE($a, b$)
    $n \leftarrow$ length of a
    $x \leftarrow \emptyset$
    $j \leftarrow 0$
    $r \leftarrow 0$
    while true do                      ▷ Construct a vector Y that is a possible solution.
        $r \leftarrow b - \sum_{i=1}^{j} a_i\, y_i$
        if $j < n$ then                ▷ We have a partial vector Y. Get next component.
            $j \leftarrow j + 1$
            $y(j) \leftarrow \lfloor \frac{r}{a(j)} \rfloor$     ▷ Largest possible y(j), given y(1:j-1)
        else                           ▷ We have a full vector Y.
            if $r == 0$ then           ▷ Is Y a solution?
                $x \leftarrow x \cup y$
            end if
            while $0 < j$ do           ▷ Find last nonzero Y entry, decrease by 1.
                if $0 < y(j)$ then
                    $y(j) \leftarrow y(j) - 1$
                    break
                end if
                $j \leftarrow j - 1$
            end while
            if $j == 0$ then
                return $x$
            end if
        end if
    end while
end procedure
```

There are several ways to improve this procedure. At an early step $j < n$, if the residual r is zero, the remaining entries of y can be set to zero immediately and a solution accepted. Moreover, if at any step j, the residual is not divisible by the greatest common divisor of the coeffients indexed from $j + 1$ through n, the partial proposed solution is untenable; the last nonzero entry in y should be decremented and the search restarted from that index. For the moment, these efficiencies have not been implemented.

As a small demonstration of this procedure, consider the calculation of the nonnegative solutions of

$$12\, x_1 + 9\, x_2 + 7\, x_3 = 30$$

The following diagram suggests the progress of the algorithm. The first line suggests that algorithm begins by setting y_1 to its maximum value of 2. It then produces the partial vector (2,0) and the full vector (2,0,0), which is rejected because it does not satisfy the equation. On the second line, we see that the algorithm steps back to the last nonzero component of y, namely y_1, reduces it by 1, and extends the vector to (1,2) and then (1,2,0). This vector satisfies the equation and so is accepted as the first solution. In the third line, we see that the algorithm steps back to the last nonzero component of y, namely y_2, decrements it to 1, and generates the vector (1,1,1), which is rejected. Subsequent lines show the orderly generation of tentative vectors, and the discovery of a second solution.

```
2; 2 0; 2 0 0
1; 1 2; 1 2 0  Solution #1
   1 1; 1 1 1
        1 1 0
   1 0; 1 0 2
        1 0 1
        1 0 0
0; 0 3; 0 3 0
   0 2; 0 2 1
        0 2 0
   0 1; 0 1 3  Solution #2
        0 1 2
        0 1 1
        0 1 0
   0 0; 0 0 4
        0 0 3
        0 0 2
        0 0 1
        0 0 0
```

As a result of the way that the possibilities are examined, the accepted solutions are automatically accumulated in a sorted order. If you regard each solution as the digits of a number, they show up in descending numeric order. For example, the $k = 10$ strictly positive solutions of $2\,x_1 + 3\,x_2 + 5\,x_3 + 6\,x_4 + 7\,x_5 = 35$ are returned in a 5×10 array x. To illustrate the ordering, we print the transpose of x, so that each solution appears as a row, and can be read as a 5 digit number:

```
7,1,1,1,1
4,3,1,1,1
4,1,1,2,1
3,2,2,1,1
2,2,1,1,2
2,1,3,1,1
1,5,1,1,1
1,3,1,2,1
1,1,2,1,2
1,1,1,3,1
```

Copies of the two MATLAB procedures are available from the web page

```
http://people.sc.fsu.edu/~jburkardt/m_src/diophantine_nd/diophantine_nd.html,
```

which also provides a procedure for prechecking the values of a and b, and points to a related directory containing some tests, and the text of this document.