This `CWEB` program implements the Simplex algorithm with **integer** arithmetic. You may also wish to read the `CWEB` program GAUSS.

1. The problem. Given a matrix $A[1\,..\,\check{m}][1\,..\,\check{n}]$ and vectors $b[1\,..\,\check{m}]$ and $c[1\,..\,\check{n}]$, we wish to find a vector x such that

$$A\cdot x \equiv b,\quad x\ge 0,\quad \text{and}\quad c\cdot x\ \text{ is minimum}$$

(i.e., $c\cdot x \le c\cdot \check{x}$ for any $\check{x}\ge 0$ such that $A\cdot\check{x}\equiv b$). The problem may have no solution for two reasons: either there is no $x\ge 0$ such that $A\cdot x = b$ or there is no such x that will minimize $c\cdot x$. In the first case we say that the problem is *infeasible*; in the second case, we say the problem is *unbounded*.

2. It is convenient to bunch A, b, and c into a single matrix: add c to A as an extra row and add b to A as an extra column. If the resulting matrix is $D[1\,..\,m][1\,..\,n]$ then

$$A\equiv D[1\,..\,m{-}1][1\,..\,n{-}1],\qquad b\equiv D[1\,..\,m{-}1][n],\qquad \text{and}\qquad c\equiv D[m][1\,..\,n{-}1] \tag{2.1}$$

(the value of $D[m][n]$ being arbitrary).

3. Simple matrices. In order to solve our problem, we shall transform matrix D until it becomes "simple". Our definition of "simple" shall be given through informal pictures, following a few conventions. In a picture of a matrix $E[1\,..\,m][1\,..\,n]$, we shall assume that the index of the bottom row is m, while the indices of the other rows are some unspecified permutation of $1\,..\,m{-}1$. Similarly, we shall assume that the index of the rightmost column is n, while the indices of the other columns are $1\,..\,n{-}1$ in some order.

Some entries of E shall be represented by numbers, like 0 and 1; these are to be taken literally. Other entries may be represented by greek letters; two entries represented by the same greek letter *are not necessarily equal.*

There are three kinds of *simple* matrices. A matrix E is *simple solvable* if it fits the pattern suggested by the picture on the left below, where the α s stand for arbitrary numbers and the β s stand for nonnegative numbers ($\beta\ge 0$). We may have any number, including zero, of null rows where our picture shows only one.

$$\begin{array}{ccccccc}
1&0&0&\alpha&\alpha&\alpha&\beta\\
0&1&0&\alpha&\alpha&\alpha&\beta\\
0&0&1&\alpha&\alpha&\alpha&\beta\\
0&0&0&0&0&0&0\\
0&0&0&\beta&\beta&\beta&\alpha
\end{array}
\qquad
\begin{array}{ccccccc}
\alpha&\alpha&\alpha&\alpha&\alpha&\alpha&\alpha\\
\gamma&\gamma&\gamma&\gamma&\gamma&\gamma&\zeta\\
\alpha&\alpha&\alpha&\alpha&\alpha&\alpha&\alpha\\
\alpha&\alpha&\alpha&\alpha&\alpha&\alpha&\alpha\\
\alpha&\alpha&\alpha&\alpha&\alpha&\alpha&\alpha
\end{array}
\qquad
\begin{array}{ccccccc}
1&0&0&\alpha&\alpha&\gamma&\beta\\
0&1&0&\alpha&\alpha&\gamma&\beta\\
0&0&1&\alpha&\alpha&\gamma&\beta\\
0&0&0&\alpha&\alpha&0&0\\
0&0&0&\alpha&\alpha&\xi&\alpha
\end{array}$$

A matrix E is *simple infeasible* if it fits the pattern suggested by middle picture above, where the α s stand for arbitrary numbers, the γ s stand for nonpositive numbers ($\gamma\le 0$), and ζ stands for a positive number ($\zeta>0$). The opposite is also acceptable: we can have γ s standing for nonnegative numbers and ζ standing for a negative number. The special row is an *infeasibility row*.

A matrix E is *simple unbounded* if it fits the pattern of the right picture above, where the α s stand for arbitrary numbers, the β s stand for nonnegative numbers ($\beta\ge 0$), the γ s stand for nonpositive numbers ($\gamma\le 0$), and ξ stands for a negative number ($\xi<0$). The column containing the γ s is an *unboundedness column*.

4. It is easy to see that if D is simple solvable then the corresponding minimization problem has a solution. (See future sections for details.) Similarly, is D is simple infeasible then the corresponding problem is infeasible. And if D is simple unbounded then the corresponding problem is unbounded.

5. Problem reformulated and restricted to integers. Since we wish to restrict ourselves to integer data, it is convenient to introduce a slightly more general notion of simplicity: For any nonnull integer d, a matrix E is *d-simple* if the matrix E/d is simple.

Now our minimization problem can be reformulated as follows: Given an *integer* matrix D, we wish to find an *integer* matrix E, *integer* matrices F and G, and a nonnull integer d such that

$$E \text{ is } d\text{-simple (solvable, infeasible, or unbounded)},$$
$$G \cdot D \equiv E, \quad F \cdot G \equiv d \cdot I, \quad \text{and} \quad G[\,][m] \equiv d \cdot I[\,][m],$$

where I denotes the identity matrix and $G[\,][m]$ denotes column m of G. The rows of all the matrices are indexed by $1\,..\,m$. The columns of D and E are indexed by $1\,..\,n$, while the columns of F, G, and I are indexed by $1\,..\,m$. (Incidentally, we assume neither $m \le n$ nor $m \ge n$.)

⟨ Basic global variables 5 ⟩ ≡

matrix D;
int m, n;
long d;
matrix G;
matrix F; ▷ $F \cdot G \equiv d \cdot I$
matrix E; ▷ $G \cdot D \equiv E$

See also sections 6 and 59.

This code is used in section 88.

6. If E turns out to be simple infeasible, the index of an infeasibility row shall be denoted by h. If E turns out to be simple unbounded, the index of an unboundedness column shall be denoted by k. If E turns out to be simple solvable, we shall set $h = k = 0$.

⟨ Basic global variables 5 ⟩ +≡

int h, k;

7. Our vectors and matrices will be allocated dinamically. Since the entries of our vectors will be **long**, a **vector** will be a pointer to **long**. Similarly, a **matrix** will be a pointer to a pointer to **long**. We shall also need a data types to store vectors with **int** entries.

⟨ Typedefs 7 ⟩ ≡

typedef long ∗∗**matrix**;
typedef long ∗**vector**;
typedef int ∗**ivector**;

See also section 32.

This code is used in section 88.

8. Memory allocation routines. Memory allocation is a rather routine matter. Let's get it done now, so we can move on to more interesting stuff. The first thing we need is someone to call when the computer runs out of memory.

⟨ Memory allocation functions 8 ⟩ ≡

```
void failure(void) {
  fprintf(stderr, "\n\n␣Out␣of␣memory:␣unable␣to␣allocate␣vector␣or␣matrix\a\n\n");
  fprintf(ofile, "\n\n␣Out␣of␣memory:␣unable␣to␣allocate␣vector␣or␣matrix\n\n\n\f\n\n");
  exit(1);
}
```

See also sections 9, 10, and 11.

This code is used in section 88.

9. We shall follow *Numerical Recipies* (W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, 2nd. edition, Cambridge University Press, 1994) in programming the memory allocation of matrices and vectors. Our first function allocates memory for an **int** vector whose entries are indexed by $1, \ldots, n$. Our second function allocates a **long** vector whose entries are indexed by $1, \ldots, n$.

⟨ Memory allocation functions 8 ⟩ +≡

```
ivector allocate_ivector(int n) {
  size_t num_bytes;
  ivector v;

  num_bytes = n * sizeof(int);
  v = (ivector) malloc(num_bytes);
  if (v ≡ Λ) failure();
  v −= 1;
  return v;
}

vector allocate_vector(int n) {
  size_t num_bytes;
  vector v;

  num_bytes = n * sizeof(long);
  v = (vector) malloc(num_bytes);
  if (v ≡ Λ) failure();
  v −= 1;
  return v;
}
```

10. The next function allocates a **long** matrix with rows indexed by $1, \ldots, m$ and columns indexed by $1, \ldots, n$.

⟨ Memory allocation functions 8 ⟩ +≡

```
matrix allocate_matrix(int m, int n) {
  size_t num_bytes;
  int i;
  matrix A;
  num_bytes = m * sizeof(long *);
  A = (matrix) malloc(num_bytes);
  if (A ≡ Λ) failure();
  A −= 1;
  num_bytes = m * n * sizeof(long);
  A[1] = (vector) malloc(num_bytes);
  if (A[1] ≡ Λ) failure();
  o, A[1] −= 1;
  for (i = 2; i ≤ m; ++i) o, A[i] = A[i − 1] + n;
  return A;
}
```

11. Sometimes we must undo the memory allocation. First, we undo allocation done by *allocate_ivector*; then we undo *allocate_vector*; finally, we free the space allocated by *allocate_matrix*. The standard function *free* receives only a pointer to the beginning of the block of bytes to be freed; it knows *how many* bytes must be freed.

⟨ Memory allocation functions 8 ⟩ +≡

```
void deallocate_ivector(ivector v) {
  free((void *) (v + 1));
}
void deallocate_vector(vector v) {
  free((void *) (v + 1));
}
void deallocate_matrix(matrix A) {
  o, free((void *) (A[1] + 1));
  free((void *) (A + 1));
}
```

12. Counting mems. You must have noticed the little *o*s preceding some expressions in the memory allocations routines. They are there to count the number of unavoidable accesses to memory executed by the critical routines of our program (see D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*, ACM Press and Addison-Wesley, 1993). A **long** variable *mems* is used to record this number. (Unfortunately, if the number of memory accesses turns out to be greater than the capacity of a **long** variable then the overflow of *mems* will go undetected and we shall get a wrong answer.)

Note that the evaluation of an expression like $A[i]$ requires only one unavoidable access to memory: we pretend that the variables A and i reside in registers and not in memory.

#define *o* *mems*++
#define *oo* *mems* += 2
#define *ooo* *mems* += 3
#define *oooo* *mems* += 4

⟨ Other global variables 12 ⟩ ≡
  static long *mems* = 0_L;

See also sections 40, 92, 95, and 116.

This code is used in section 88.

13. The pivoting operation. The heart of all Simplex algorithms is the following "pitoting" operation. Suppose d is a nonnull integer and let $E[1\,..\,m][1\,..\,n]$ be an integer matrix. Assuming $E[p][k] \neq 0$, a *pivot about row p and column k* is the following operation: for each i distinct from p, replace vectors $E[i]$ and $G[i]$ (i.e., rows i of G and E) by the vectors

$$\frac{Epk \cdot E[i] - Eik \cdot E[p]}{d} \quad \text{and} \quad \frac{Epk \cdot G[i] - Eik \cdot G[p]}{d}$$

respectively. Having done this, replace d by Epk.

Here, $E[p]$ denotes row p of E while Epk is our sloppy abbreviation for $E[p][k]$. Vectors $E[p]$ and $G[p]$ remain unchanged. Incidentally, we shall have $p \neq m$ and $k \neq n$ whenever we use this piece of code in the future.

When this pivoting is done in proper context, the divisions by d will generate no fractions: for all j, the value of $Ehk \cdot E[i][j] - Eik \cdot E[h][j]$ will be divisible by d for all j.

⟨ Pivot about row p and column k and update d 13 ⟩ ≡ {

```
  long t, Epk, Eik;
  Epk = E[p][k];
  for (i = 1; i ≤ m; ++i)
    if (i ≠ p) {
      Eik = E[i][k];
      for (j = 1; j ≤ n; ++j) {
        t = Epk * E[i][j] - Eik * E[p][j];
        E[i][j] = t/d;
      }
      for (j = 1; j ≤ m; ++j) {
        t = Epk * G[i][j] - Eik * G[p][j];
        G[i][j] = t/d;
      }
    }
  d = Epk;
}
```

This code is used in sections 19 and 24.

14. Suppose that, before the pivoting operation, matrix E fits the pattern suggested by the picture on the left below. Then the matrix that emerges from the pivoting operation will have the pattern suggested by the picture on the right below. (As usual, the α s stand for arbitrary integers, not necessarily all equal; a similar observation goes for the α' s.)

$$\begin{array}{c|ccccccc}
 & d & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \\
 & 0 & d & \alpha & \alpha & \alpha & \alpha & \alpha \\
p & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \\
 & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \\
 & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \\
 & & & k & & & &
\end{array}
\qquad
\begin{array}{c|ccccccc}
 & d' & 0 & 0 & \alpha' & \alpha' & \alpha' & \alpha' \\
 & 0 & d' & 0 & \alpha' & \alpha' & \alpha' & \alpha' \\
p & 0 & 0 & d' & \alpha' & \alpha' & \alpha' & \alpha' \\
 & 0 & 0 & 0 & \alpha' & \alpha' & \alpha' & \alpha' \\
 & 0 & 0 & 0 & \alpha' & \alpha' & \alpha' & \alpha' \\
 & & & k & & & &
\end{array}$$

15. Here is a fundamental observation about the effect of the pivoting operation on column n of E. We shall say that a column k is *good* for row p if $k < n$ and

$$\frac{Epn}{Epk} \geq 0 \tag{15.1}$$

(we assume, of course that $Epk \neq 0$). If k is good for p then we shall have $Epn/d \geq 0$ *after* the pivoting operation. Now let's ask a similar question of a row i distinct from p. Suppose

$$Ein/d \geq 0 \tag{15.2}$$

before the pivoting. Under what circumstances is it true that $Ein/d \geq 0$ *after* pivoting? In other words, under what circumstances is it true that

$$\frac{Epk \cdot Ein - Eik \cdot Epn}{d \cdot Epk} \geq 0$$

before the pivoting. Here is the answer: the inequality is true if

$$Eik/d \leq 0 \qquad \text{or} \qquad \frac{Ein}{Eik} \geq \frac{Epn}{Epk} \ . \tag{15.3}$$

To prove our claim, supose first that $Eik/d \leq 0$. Then

$$\frac{Epk \cdot Ein - Eik \cdot Epn}{d \cdot Epk} \equiv \frac{Ein}{d} - \frac{Eik}{d} \cdot \frac{Epn}{Epk} \geq 0 \ ,$$

as claimed. Now suppose $Eik/d > 0$ and the second alternative in (15.3) holds. Then $Epk \cdot Ein \geq Eik \cdot Epn$ if Eik and Epk have the same sign and $Epk \cdot Ein \leq Eik \cdot Epn$ otherwise. In other words,

$$(Epk \cdot Ein - Eik \cdot Epn) \cdot Eik \cdot Epk \geq 0 \ .$$

This inequality is equivalent to

$$\frac{Epk \cdot Ein - Eik \cdot Epn}{d \cdot Epk} \geq 0$$

since Eik and d have the same sign.

16. The basic heuristic. The well-known Simplex algorithm solves our problem if we do not insist on G being integer: upon receiving D, it produces a rational — not necessarily integer — matrix G such that $G \cdot D$ is simple. In order to solve our problem as stated, we shall resort to a *variant* of the Simplex algorithm. I am not sure who is to be credited for this variant; the names of Cramer, Chio, and Edmonds come to mind. For lack of a better name, we shall call it "Simplex–Chio".

Actually, our first versions of the precedure will not be true algorithms since they may go into an endless cycle for some inputs. For this reason, we shall call them *heuristics*.

17. The function *simplex_0* is a naive implementation of the Simplex–Chio heuristic. It receives an integer matrix $D[1\,..\,m][1\,..\,n]$ and returns an integer d and an integer matrix G that will solve our problem. However, the function *may not converge* and, even if it converges, it may not produce the desired results due to *arithmetic overflow*† during the computations.

If all goes well, the function also produces an integer matrix E and an integer matrix F such that E equal to $G \cdot D$ and $F \cdot G \equiv d \cdot I$.

⟨ The basic heuristic 17 ⟩ ≡

```
long simplex_0(matrix D, int m, int n, matrix F, matrix G, matrix E) {
  int h, k, p, i, j;
  long d;
  for (i = 1; i ≤ m; ++i)
    for (j = 1; j ≤ n; ++j) E[i][j] = D[i][j];      ▷ E = D
  for (i = 1; i ≤ m; ++i)
    for (j = 1; j ≤ m; ++j) F[i][j] = G[i][j] = i ≡ j ? 1 : 0;      ▷ F = G = I
  d = 1;
  ⟨ Phase 1: deal with rows 1 .. m − 1 24 ⟩
  ⟨ Phase 2: deal with row m 19 ⟩
}
```

This code is used in section 88.

18. The heuristic has two phases. Each iteration of phase 1 begins with a matrix E that fits the pattern suggested by the picture on the left below. In that picture, we assume that the rows are indexed by $1\,..\,m$ from top to bottom and that the rightmost column has index n. Moreover, the αs and βs are integer and $\beta/d \geq 0$ for each β.

$$\begin{array}{cccccccc} & d & 0 & \alpha & \alpha & \alpha & \alpha & \beta \\ & 0 & d & \alpha & \alpha & \alpha & \alpha & \beta \\ h & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \\ & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \\ & 0 & 0 & \alpha & \alpha & \alpha & \alpha & \alpha \end{array} \qquad \begin{array}{ccccccc} d & 0 & 0 & \alpha & \alpha & \alpha & \beta \\ 0 & d & 0 & \alpha & \alpha & \alpha & \beta \\ 0 & 0 & d & \alpha & \alpha & \alpha & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha & \alpha & \alpha & \alpha \end{array}$$

Each iteration of phase 2 begins with a matrix E that fits the pattern suggested by the picture on the right above, where $\beta/d \geq 0$ for each β.

† unfortunately, overflows go undetected

19. *Phase 2.* At the beginning of each iteration of phase 2 we have $F \cdot G/d \equiv I$, $G[\,][m]/d \equiv I[\,][m]$ and (compare with 15.2)

$$Ein/d \;\geq\; 0 \tag{19.1}$$

for each i in $1\,..\,m{-}1$. The nominal goal of phase 2 is to enforce the inequality $Emk/d \geq 0$ for all k in $1\,..\,n{-}1$. Phase 2 may not achieve its nominal goal if it runs into an unboundedness column (and also if it goes into an endless cycle).

⟨ Phase 2: deal with row m 19 ⟩ ≡

```
while (1) {
  for (k = 1; k < n ∧ E[m][k] ∗ d ≥ 0; ++k) ;     ▷ 1
  if (k < n) {     ▷ 2
    for (p = 1; p < m; ++p)     ▷ 3
      if (E[p][k] ∗ d > 0) break;     ▷ 4
    if (p < m) {     ▷ 5
      for (i = p + 1; i < m; ++i)     ▷ 6
        if (E[i][k] ∗ d > 0)     ▷ 7
          if (E[i][k] ∗ E[p][n] − E[p][k] ∗ E[i][n] > 0) p = i;     ▷ 8
      ⟨ Pivot about row p and column k and update d 13 ⟩     ▷ 9
      for (i = 1; i ≤ m; ++i) F[i][p] = D[i][k];     ▷ 10
    }
    if (p ≡ m) return d;     ▷ 11
  }
  else return d;     ▷ 12
}
```

This code is used in section 17.

20. Line 1 looks for k in $1\,..\,n{-}1$ such that $Emk/d \geq 0$. If no such k is found, E is simple solvable; line 2 sends us to line 12 and there the execution of *simplex_0* terminates.

In order to avoid a division operation, we write $Emk \cdot d \geq 0$ instead of $Emk/d \geq 0$. Actually, the product $Epk \cdot d$ could be replaced by $Epk \cdot s$, where s is the *sign* of d. We shall put this observation to use in our next implementation.

21. At the beginning of line 3, k is such that $Emk/d < 0$. To come closer to our nominal goal, we would like to do a pivoting operation about column k and some row p; after such pivot, we would have $Emk/d \equiv 0$.

In order to preserve the identity $G[\,][m]/d \equiv I[\,][m]$, we must have $p < m$. In order to preserve 19.1, p must satisfy the conditions we learned in 15.3. Lines 3 to 8 try to find p in $1\,..\,m{-}1$ satisfying two conditions: $Epk/d > 0$ and

$$Epn/Epk \leq Ein/Eik \quad \text{for each } i \text{ in } 1\,..\,m{-}1 \text{ such that } \; Eik/d > 0\,.$$

We say that such p is *safe in* $1\,..\,m{-}1$. If there is no safe p (i.e., if $p \equiv m$ at the end of lines 3–4) we stop striving for our nominal goal: matrix E is simple unbounded. In such case, line 5 sends us to line 11 and the execution of *simplex_0* stops.

(I wrote "**if** $(p \equiv m)$" instead of as "**else**" in line 11 in order to emphasize the formal parallel with line 29 in the code for phase 1.)

The inequalities in the definition of safeness can be restated without divisions. For example, $Epk/d > 0$ is equivalent to $Epk \cdot d > 0$. Moreover, if $Eik \cdot d > 0$ and $Epk \cdot d > 0$ then $Epn/Epk \leq Ein/Eik$ is equivalent to $Eik \cdot Epn - Epk \cdot Ein \leq 0$.

22. Line 9 performs a pivoting operation about p and k. Because of our careful choice of p, 19.1 will remain true after the pivoting.

The remarkable thing about the pivoting operation is that *all the divisions by d are exact*: there are no truncations and no fractions. The reasons for this are well-known but by no means obvious: at the beginning of each iteration, *d and each entry in G and in E is the determinant of some submatrix of D.*

23. Line 10 updates column p of matrix F. This update preserves the identity $F \cdot G/d \equiv I$ at the beginning of each iteration.

24. *Phase 1.* The code for phase 1 is surprisingly similar to that of phase 2. At the beginning of each iteration of phase 1 we have $F \cdot G/d \equiv I$, $G[\,][m]/d \equiv I[\,][m]$ and (compare with 15.2)

$$Ein/d \;\geq\; 0 \tag{24.1}$$

for each i in $1\,..\,h{-}1$. The nominal goal of phase 1 is to make this inequality valid also for $i \equiv h$. But phase 1 may not achieve this goal if it runs into an infeasibility row.

Each iteration in phase 1 deals with a row h; at the end of each iteration, h may be incremented or remain unchanged.

$\langle$ Phase 1: deal with rows $1\,..\,m-1$ 24 $\rangle \equiv$

```
h = 1;
while (h < m) {
  if (E[h][n] < 0)    ▷ 13
    for (k = 1; k < n ∧ E[h][k] ≥ 0; ++k) ;    ▷ 14
  else if (E[h][n] > 0)
    for (k = 1; k < n ∧ E[h][k] ≤ 0; ++k) ;    ▷ 16
  else
    for (k = 1; k < n ∧ E[h][k] ≡ 0; ++k) ;    ▷ 18
  if (k < n) {    ▷ 19
    for (p = 1; p < h; ++p)    ▷ 20
      if (E[p][k] * d > 0) break;    ▷ 21
    if (p < h) {    ▷ 22
      for (i = p + 1; i < h; ++i)    ▷ 23
        if (E[i][k] * d > 0)    ▷ 24
          if (E[i][k] * E[p][n] − E[p][k] * E[i][n] > 0) p = i;    ▷ 25
      if ((E[h][k] * E[p][n] − E[p][k] * E[h][n]) * d * E[h][k] ≥ 0) p = h;    ▷ 26
    }
    ⟨ Pivot about row p and column k and update d 13 ⟩    ▷ 27
    for (i = 1; i ≤ m; ++i) F[i][p] = D[i][k];    ▷ 28
    if (p ≡ h) ++h;    ▷ 29
  }
  else if (E[h][n] ≡ 0) ++h;    ▷ 30
  else return d;    ▷ 31
}
```

This code is used in section 17.

25. Lines 13 to 18 choose a good k, i.e., an index $k < n$ such that $Ehk \neq 0$ and $Ehn/Ehk \geq 0$. If there is no good k, row h has one of the following three forms: either $Eh[1 \mathinner{..} n] \equiv 0$; or $Eh[1 \mathinner{..} n{-}1] \leq 0$ and $Ehn > 0$; or $Eh[1 \mathinner{..} n{-}1] \geq 0$ and $Ehn < 0$. In the first case, there is nothing to be done on row h: lines 19 and 30 send us to the next iteration with $h + 1$ in place of h. In the two last cases, E is simple infeasible: lines 19 and 30 send us to line 31 and *simplex_0* terminates.

26. At the beginning of line 20, we have a good k. If we pivot about h and k, the inequality 24.1 will become valid for $i \equiv h$ but may become invalid for some of the other indices in $1 \mathinner{..} h{-}1$. Lines 20–26 look for p in $1 \mathinner{..} h$ such that after a pivot about p and k the inequality 24.1 will remain valid for all i in $1 \mathinner{..} h{-}1$ and perhaps will become valid for $i \equiv h$.

Lines 20 to 25 (compare with lines 3 to 8 in the code of phase 2) look for a safe p in $1 \mathinner{..} h{-}1$, i.e., an index p such that $Epk/d > 0$ and

$$Epn/Epk \leq Ein/Eik \quad \text{for each } i \text{ in } 1 \mathinner{..} h{-}1 \text{ such that} \quad Eik/d > 0\,.$$

Suppose there is no such p, i.e., suppose $p \equiv h$ at the end of lines 20–21. Then we may pivot (line 27) about h and k without fear of disturbing 24.1. Now suppose $p < h$ at the end of lines 20–21. Then lines 23–25 finish the job of finding a safe p in $1 \mathinner{..} h{-}1$.

27. At the beginning of line 26, p is safe in $1 \mathinner{..} h{-}1$. Line 26 checks whether

$$Epn/Epk \geq Ehn/Ehk.$$

Note that this inequality can be restated without divisions: since $Epk/d > 0$, it is equivalent to $(Ehk \cdot Epn - Epk \cdot Ehn) \cdot d \cdot Ehk \geq 0$.

Suppose the inequality is true. Then it is also true that $Ehn/Ehk \leq Ein/Eik$ for each i in $1 \mathinner{..} h{-}1$ such that $Eik/d > 0$. Hence, we can pivot about h and k without disturbing 24.1.

Now suppose the inequality is false. Then we must pivot about p and k and hope that things get better in the next iteration.

28. If the pivoting operation was done about h (and k), line 29 increments h before starting a new iteration. Note that p may have become equal to h on two different occasions: either in lines 20–21 or in line 26.

If the pivoting was *not* done about h, we start a new iteration with the same h as before. Even though h did not change, our matrix E is somehow "better". On rare occasions, however, this loop may get forever stuck with a certain value of h.

29. Line 27 performs a pivoting operation about p and k. As in phase 2, all the divisions by d are exact. Finally, line 28 updates column p of matrix F. This update preserves the identity $F \cdot G/d \equiv I$ at the beginning of each iteration.

30. Arithmetic overflow. The magnitude† of the numbers generated by the Simplex–Chio procedure may become very large, even if the magnitudes of the entries in the data matrix D are very small. In particular, the numbers generated by *simplex_0* may easily exceed the capacity of a **long** variable. If such overflow occurs, the results will be corrupted without warning.

31. How can we detect an overflow in an arithmetic operation before it actually happens? Before explaining the trick, we must understand the range of **long** integers.

Since **sizeof**(**long**) $\equiv 4$, a **long** integer has 32 bits. Since our computer uses two's complement notation, the range of representable integers goes from -2^{31} to $2^{31}-1$. (These numbers are called `LONG_MIN` and `LONG_MAX` respectively in the header file `limits.h`.) We shall, however, discard the use of -2^{31} and restrict ourselves to the interval $-2^{31}+1 \mathrel{..} 2^{31}-1$. In other words, we shall make sure that the magnitude of all our integers is allways strictly smaller than 2^{31}. Incidentaly, the representation of $2^{31}-1$ is 2,147,483,647 in decimal notation. The hexadecimal representation is given next.

#define `TWO31M1` $^{\#}$`7fffffff`$_{\mathrm{L}}$ ▷ $2^{31}-1$

32. In order to detect an overflow during a **long** arithmetic operation, we shall use **long long** variables. The **long long** type is not part of the ANSI standard, but is recognized by the GNU C compiler. Since **sizeof**(**long long**) $\equiv 8$, a **long long** integer has 64 bits and therefore the range of representable integers goes from -2^{63} to $2^{63}-1$ (these numbers are called `LLONG_MIN` and `LLONG_MAX` respectively in the file `limits.h`). Incidentally, the decimal representation of $2^{63}-1$ is 9,223,372,036,854,775,807. The hexadecimal representation of 2^{31} is given next.

#define `TWO31` $^{\#}$`0000000080000000`$_{\mathrm{LL}}$ ▷ 2^{31}

⟨ Typedefs 7 ⟩ +≡

typedef long long llong;

33. Our computations will be done as follows. Suppose that the magnitudes of the **long** variables a, x, b and y are smaller than 2^{31}. Then the evaluation of the expression

$$(\mathbf{llong})\ a * (\mathbf{llong})\ x - (\mathbf{llong})\ b * (\mathbf{llong})\ y$$

produces no overflow. This is so because, for any positive integers α and ξ,

$$\text{if } \alpha < 2^{31} \text{ and } \xi < 2^{31} \text{ then } \alpha \cdot \xi < 2^{62}, \qquad \text{and} \qquad \text{if } \alpha < 2^{62} \text{ and } \xi < 2^{62} \text{ then } \alpha + \xi < 2^{63}.$$

Now, if the value of the **llong** expression is less than `TWO31`, it can be safely stored in a **long** variable. Otherwise, there is nothing to do but cry "Overflow!".

† magnitude = absolute value

34. *Predicting a bound on all matrix entries.* (The next two sections are included just for fun: the corresponding code will not be actually used in our implementations of the Simplex-Chio algorithm.)

There is a way to predict, before actually running the algorithm, an upper bound on the magnitude of all matrix entries that will be generated during the execution of the algorithm. We shall denote such upper bound by ω.

All the numbers generated in the course of the Simplex–Chio algorithm (in particular, all entries of G and E) are determinants of square submatrices of the data matrix D; this is a theorem. It is not difficult to see that the number

$$\omega_1 \;=\; \textstyle\prod_{i=1}^{m} \, (1 + D[i][1] + \cdots + D[i][n])$$

is a bound on the magnitude of any subdeterminant of D. Similarly, the number

$$\omega_2 \;=\; \textstyle\prod_{j=1}^{n} \, (1 + D[1][j] + \cdots + D[m][j]) \; .$$

is a bound on the magnitude of any subdeterminant of D. The upper bound ω mentioned above may be defined as $\min(\omega_1, \omega_2)$. Unfortunately, this bound is usually too loose to be useful; only for some rare matrices ω is tight. Hence, we shall compute ω just out of curiosity.

35. The function *omega1* will receive a **matrix** $D[1\,..\,m][1\,..\,n]$ and return the value of its ω_1. If the parameter cannot be computed correctly in **long** arithmetic due to overflow, the function will return `LONG_MIN`, which is equal to -2^{31}.

Our function assumes that the magnitude of each entry of D is strictly smaller than 2^{31}. In order to detect an overflow, we shall do the computations in **llong** arithmetic.

⟨ Auxiliary functions 35 ⟩ ≡

```
long omega1(matrix D, int m, int n) {
  llong term, sum, prod;
  int i, j;
  prod = 1LL;
  for (i = 1; i ≤ m; ++i) {
    sum = 1LL;
    for (j = 1; j ≤ n; ++j) {
      term = (llong) D[i][j];
      if (term < 0LL) term = −term;
      sum += term;
      if (sum ≥ TWO31) return LONG_MIN;      ▷ TWO31 ≡ 2^31
    }
    prod *= sum;
    if (prod ≥ TWO31) return LONG_MIN;
  }
  return (long) prod;
}
```

See also sections 36, 118, 121, 122, 123, 124, 125, 127, 130, 133, and 137.

This code is used in section 88.

36. The function *omega2* is similar: it attempts to computes the bound ω_2.

⟨ Auxiliary functions 35 ⟩ +≡

```
long omega2(matrix D, int m, int n) {
  llong term, sum, prod;
  int i, j;
  prod = 1LL;
  for (j = 1; j <= n; ++j) {
    sum = 1LL;
    for (i = 1; i <= m; ++i) {
      term = (llong) D[i][j];
      if (term < 0LL) term = -term;
      sum += term;
      if (sum >= TWO31) return LONG_MIN;
    }
    prod *= sum;
    if (prod >= TWO31) return LONG_MIN;
  }
  return (long) prod;
}
```

37. First implementation of the heuristic. Our first implementation of the Simplex–Chio heuristic (*simplex_0* does not really count) makes sure that no overflow will go undetected. (But we shall do nothing, in this implementation, about the cycling nuisance.)

The function *simplex_1* receives matrix D and, *if it converges*, returns an integer d. It returns $d \equiv -2^{31}$ if the computations were interrupted due to an imminent overflow; the values of G, E and F are meaningless in this case. It returns $d \neq -2^{31}$ if the computations went through without any overflow; in this case, the quadruple G, d, E, F is a solution to our problem.

The function assumes that the magnitude of each entry of the data matrix D is strictly smaller than 2^{31}. If there is no overflow, the magnitude of d and of each entry of G and E will also be smaller than 2^{31}.

⟨ First implementation of the heuristic 37 ⟩ ≡

```
long simplex_1(matrix D, int m, int n, matrix F, matrix G, matrix E, int *inf, int *unb) {
  int h, p, k, i, j;
  long d, sd, sdE;
  vector Di, Eh, Ep, Ei, Gp, Gi, Fi;
  long Ehk, Ehn, Epk, Epn, Eik, Ein;
  llong t;
  ⟨ Set E = D, F = G = I, and d = 1 39 ⟩
  ⟨ Initialize num_its and maxmag 41 ⟩
  ⟨ Phase 1 of first implementation 42 ⟩
  ⟨ Phase 2 of first implementation 47 ⟩
}
```

This code is used in section 88.

38. In order to indicate the kind of simple matrix it found, the function produces integers $*inf$ and $*unb$: if the matrix E turns out to be simple infeasible then $*inf$ is the index of an infeasibility row; and if E turns out to be unbounded then $*unb$ is the index of an unboundedness column. If $*inf \equiv 0$ and $*unb \equiv 0$ then E is simple solvable.

39. A note on *mems* counting: The evaluation of an expression like $E[i][j]$ requires two accesses to memory: one the get the value of $E[i]$ and the other to get $E[i][j]$ proper. (As usual, we shall pretend that the variables E, i and j reside in registers and not in memory.) If all entries in row i must be processed, we may reduce the number of memory accesses by copying $E[i]$ to a register, say Ei, and then writing $Ei[j]$ instead of $E[i][j]$.

⟨ Set $E = D$, $F = G = I$, and $d = 1$ 39 ⟩ ≡

```
for (i = 1; i ≤ m; ++i) {
  oo, Ei = E[i], Di = D[i];
  for (j = 1; j ≤ n; ++j) oo, Ei[j] = Di[j];
}
for (i = 1; i ≤ m; ++i) {
  oo, Fi = F[i], Gi = G[i];
  for (j = 1; j ≤ m; ++j) oo, Fi[j] = Gi[j] = i ≡ j ? 1L : 0L;
}
d = 1L;
```

This code is used in section 37.

40. To satisfy the curiosity of the user, we shall keep track of the number of iterations in each phase; global **llong** variables *num_its_ph1* and *num_its_ph2* will be used for this purpose. We shall also keep track of the largest magnitude among all entries in G and E throughout the computations; a global **llong** variable *maxmag* will be used to store this value.

⟨ Other global variables 12 ⟩ +≡

```
llong num_its_ph1, num_its_ph2, maxmag;
```

41. The initial value of *maxmag* is the largest magnitude among all entries of matrices D and I, since at this point $E \equiv D$ and $G \equiv I$.

⟨ Initialize *num_its* and *maxmag* 41 ⟩ ≡

```
maxmag = 1LL;    ▷ largest entry in matrix G
for (i = 1; i ≤ m; ++i) {
  o, Ei = E[i];
  for (k = 1; k ≤ n; ++k) {
    o, Eik = Ei[k];
    if (Eik < 0L) Eik = −Eik;
    if (maxmag < (llong) Eik) maxmag = (llong) Eik;
  }
}
num_its_ph1 = num_its_ph2 = 0LL;
```

This code is used in sections 37, 63, and 69.

42. The implementation of phase 1 is copied down from the basic version of the heuristic with some stylistic changes.

⟨ Phase 1 of first implementation 42 ⟩ ≡

```
h = 1;
while (h < m) {
  ++num_its_ph1;
  ⟨ Choose a good column k; set k = n if no such column exists 43 ⟩
  if (k < n) {
    ⟨ Choose safe p in 1 .. h − 1; set p = h if there are no candidates 44 ⟩
    ⟨ Set p = h if pivoting about h and k is ok, after all 45 ⟩
    ⟨ Pivot about p and k and update d 50 ⟩
    for (i = 1; i ≤ m; ++i) oooo, F[i][p] = D[i][k];
    if (p ≡ h) ++h;
  }
  else if (o, Eh[n] ≡ 0L) ++h;
  else ⟨ E is simple infeasible; return 46 ⟩
}
```

This code is used in section 37.

43. Nothing new here, except for *mems* counting.

⟨ Choose a good column k; set $k = n$ if no such column exists 43 ⟩ ≡

```
o, Eh = E[h];
if (o, Eh[n] < 0L)
  for (k = 1; k < n ∧ (o, Eh[k] ≥ 0L); ++k) ;
else if (o, Eh[n] > 0L)
  for (k = 1; k < n ∧ (o, Eh[k] ≤ 0L); ++k) ;
else
  for (k = 1; k < n ∧ (o, Eh[k] ≡ 0L); ++k) ;
```

This code is used in sections 42, 65, and 71.

44. This section and the next use some of our overflow-avoiding tricks. At the end of this piece of code we have $Ep \equiv E[p]$, $Epk \equiv Ep[k]$, and $Epn \equiv Ep[n]$.

⟨ Choose safe p in $1 \mathrel{..} h-1$; set $p = h$ if there are no candidates 44 ⟩ ≡

```
sd = d ≥ 0L ? +1L : −1L;
for (p = 1; p < h; ++p)
  if (oo, E[p][k] ∗ sd > 0L) break;
ooo, Ep = E[p], Epk = Ep[k], Epn = Ep[n];
for (i = p + 1; i < h; ++i) {
  ooo, Ei = E[i], Eik = Ei[k], Ein = Ei[n];
  if (Eik ∗ sd > 0L) {
    t = (llong) Epk ∗ (llong) Ein − (llong) Eik ∗ (llong) Epn;
    if (t < 0LL) p = i, Ep = Ei, Epk = Eik, Epn = Ein;
  }
}
```

This code is used in sections 42 and 47.

45. We assume that $Ep \equiv E[p]$, $Epk \equiv Ep[k]$, and $Epn \equiv Ep[n]$ before entering this piece of code. The first two relations remain true when we leave this piece of code.

⟨ Set $p = h$ if pivoting about h and k is ok, after all 45 ⟩ ≡

```
if (p < h) {
  oo, Ehk = Eh[k], Ehn = Eh[n];
  t = (llong) Ehk ∗ (llong) Epn − (llong) Epk ∗ (llong) Ehn;
  sdE = sd ∗ Ehk ≥ 0L ? +1L : −1L;
  if (t ∗ (llong) sdE ≥ 0LL) p = h, Ep = Eh, Epk = Ehk;
}
```

This code is used in sections 42, 65, and 71.

46. ⟨ E is simple infeasible; **return** 46 ⟩ ≡ {

```
  oo, ∗inf = h, ∗unb = 0;
  return d;
}
```

This code is used in sections 42, 65, and 71.

47. The implementation of phase 2 is the same as in the basic version except for some stylistic changes. In order to re-use some of the code from phase 1, we shall write h where we should be writing m; of course $h \equiv m$ throughout phase 2.

⟨ Phase 2 of first implementation 47 ⟩ ≡

```
while (1) {        ▷ h ≡ m
  ++num_its_ph2;
  sd = d ≥ 0L ? +1L : −1L;
  o, Eh = E[h];
  for (k = 1; k < n ∧ (o, Eh[k] ∗ sd ≥ 0L); ++k) ;
  if (k < n) {
    ⟨ Choose safe p in 1 .. h − 1; set p = h if there are no candidates 44 ⟩
    if (p < h) {
      ⟨ Pivot about p and k and update d 50 ⟩
      for (i = 1; i ≤ m; ++i) oooo, F[i][p] = D[i][k];
    }
    else ⟨ E is simple unbounded; return 49 ⟩
  }
  else ⟨ E is simple solvable; return 48 ⟩
}
```

This code is used in section 37.

48. ⟨ E is simple solvable; **return** 48 ⟩ ≡ {

```
  oo, ∗inf = ∗unb = 0;
  return d;
}
```

This code is used in sections 47, 67, and 74.

49. ⟨ E is simple unbounded; **return** 49 ⟩ ≡ {

```
  oo, ∗unb = k, ∗inf = 0;
  return d;
}
```

This code is used in sections 47, 67, and 74.

50. The pivoting operation depends heavily of our overflow-control tricks. If an overflow occurs, there is nothing to do but abort the computation. This piece of code assumes that $Ep \equiv E[p]$ and $Epk \equiv Ep[k]$.

⟨ Pivot about p and k and update d 50 ⟩ ≡ {

```
  register llong magt;
  o, Gp = G[p];
  for (i = 1; i ≤ m; ++i)
    if (i ≠ p) {
      oo, Ei = E[i], Eik = Ei[k];
      ⟨ Update row i of E 51 ⟩
      o, Gi = G[i];
      ⟨ Update row i of G 52 ⟩
    }
  d = Epk;
  ⟨ If verbose, print p, G and E 54 ⟩
}
```

This code is used in sections 42, 47, 65, 67, 71, and 74.

51. ⟨ Update row i of E 51 ⟩ ≡

```
for (j = 1; j ≤ n; ++j) {    ▷ Ep ≡ E[p] and Epk = Ep[k]
  oo, t = (llong) Epk * (llong) Ei[j] − (llong) Eik * (llong) Ep[j];    ▷ t is divisible by d
  t /= (llong) d;
  magt = t < 0LL ? −t : t;
  if (magt ≥ TWO31) ⟨ Imminent overflow! 53 ⟩
  o, Ei[j] = (long) t;
  if (maxmag < magt) maxmag = magt;
}
```

This code is used in section 50.

52. ⟨ Update row i of G 52 ⟩ ≡

```
for (j = 1; j ≤ m; ++j) {
  oo, t = (llong) Epk * (llong) Gi[j] − (llong) Eik * (llong) Gp[j];    ▷ t is divisible by d
  t /= (llong) d;
  magt = t < 0LL ? −t : t;
  if (magt ≥ TWO31) ⟨ Imminent overflow! 53 ⟩
  o, Gi[j] = (long) t;
  if (maxmag < magt) maxmag = magt;
}
```

This code is used in section 50.

53. ⟨ Imminent overflow! 53 ⟩ ≡ {

```
  if (verbose ≡ on) {
    fprintf(ofile, "\n\n␣Unable␣to␣update␣E[%d][%d]␣or", i, j);
    fprintf(ofile, "␣G[%d][%d]␣without␣overflow!\n", i, j);
  }
  return LONG_MIN;    ▷ LONG_MIN = −2^31
}
```

This code is used in sections 51 and 52.

54. A printout of G and E at the end of each iteration may help if you are studying the behaviour of the heuristic.

⟨ If *verbose*, print p, G and E 54 ⟩ ≡

```
if (verbose ≡ on) {
  fprintf(ofile, "\n\n␣G␣and␣E␣=␣G*D␣after␣pivot␣on␣row␣%d:", p);
  printmatrices(G, E, m, n);
}
```

This code is used in section 50.

55. Convergence. Why does our heuristic diverge for some inputs? Let's begin by reviewing our criteria for choosing k and p. At the beginning of each iteration of phase 1, index k is chosen in $1\,..\,n{-}1$ so that $Emk/d < 0$. Then, p is chosen in $1\,..\,m{-}1$ so that

$Epk/d > 0$ and
$Epn/Epk \le Ein/Eik$ for each i in $1\,..\,m{-}1$ such that $Eik/d > 0$.

At the beginning of each iteration of phase 1, k is chosen in $1\,..\,n{-}1$ so that $Ehk \neq 0$ and $Ehn/Ehk \ge 0$. Then, p is chosen in $1\,..\,h$ so that

either $p \equiv h$ or $Epk/d > 0$ and
$Epn/Epk \le Ein/Eik$ for each i in $1\,..\,m{-}1$ such that $Eik/d > 0$.

As long as there is only one way to choose k and p, convergence is not under threat (essencially because Ehn will come stictly closer to zero with each iteration). But if there are *ties*, i.e., if there is more than one k or more than one p satisfying the requirements, we may go into an endless cycle, unless the ties are not broken in a consistent way.

Two tie-breaking rules are known to work: the *lexicographic method* and *Bland's rule*. The latter is named after R. Bland (*New finite pivoting rules for the simplex method*, Mathematics of Operations Research, 2 (1977), pp.103–107). We shall implement both rules in future sections. After this is done, we shall feel entitled to use the term "algorithm" instead of "heuristic".

In order to implement either of these rules we must have an explicit representation of the row and column bases of matrix E.

56. Row and column bases. Before we can tackle the cycling issue, we must introduce the concept of row and column bases. By doing so, we shall also gain a more formal understanding of the structure of simple matrices. Consider an integer matrix $E[1\,..\,m][1\,..\,n]$ and a nonnull integer d. A *column basis* for the pair E, d is an array $\varphi[1\,..\,n-1]$ such that

all entries of φ are in $0\,..\,m-1$,
the nonnull entries of φ are pairwise distinct, and
if $\varphi[j] \neq 0$ then $E[\,][j]/d \equiv I[\,][\varphi[j]]$.

Here, $E[\,][j]$ stands for $E[1\,..\,m][j]$ and I is the identity matrix indexed by $1\,..\,m$. Given φ, we shall say that a column index j is *basic* if $\varphi[j] \neq 0$ and that it is *nonbasic* otherwise.

It is more conveninent to work with the 'inverse' of a column basis, as given by the following definition. The *row basis* associated with φ is the array $\psi[1\,..\,m-1]$ defined by the pair of conditions

$$\text{if } \varphi[j] \neq 0 \text{ then } \psi[\varphi[j]] = j \quad \text{and} \quad \text{if } \psi[i] \neq 0 \text{ then } \varphi[\psi[i]] = i\,.$$

Given ψ, we shall say that a row index i is *basic* if $\psi[i] \neq 0$ and that it is *nonbasic* otherwise.

57. *Simple matrices: formal definition.* Suppose ψ is a row basis for a matrix E. Let A' stand for the matrix $E[1\,..\,m-1][1\,..\,n-1]$, let b' stand for the vector $E[1\,..\,m-1][n]$, and let c' stand for the vector $E[m][1\,..\,n-1]$. Matrix E is *d-simple solvable* if the following conditions hold: (1) $b'/d \geq 0$; (2) $\psi[i] \equiv 0$ implies $A'[i][\,] \equiv 0$ and $b'[i] \equiv 0$; and (3) $c'/d \geq 0$.

Matrix E is *d-simple unbounded* if there exists $k < n$ such that (1) $b'/d \geq 0$; (2) $A'[\,][k]/d \leq 0$; (3) $c'[k]/d < 0$; and (4) $\psi[i] \equiv 0$ implies $A'[i][k] \equiv b'[i] \equiv 0$.

Matrix E is *d-simple infeasible* if there exists $h < m$ such that either (1) $A'[h][\,] \leq 0$ and $b'[h] > 0$ or (2) $A'[h][\,] \geq 0$ and $b'[h] < 0$.

58. *Entering and leaving the basis.* Suppose a pivot operation occurs about a row p and a column k in the middle of some iteration. We shall then set $\varphi[k] = p$ and say that k *enters the basis*. In order to preserve the proper relation between φ and ψ we must also set $\psi[p] = k$.

Before the pivoting operation, $\varphi[k]$ is null but $\psi[p]$ may be nonnull. If $\psi[p] \neq 0$ before the pivot operation, we say then that the column $\psi[p]$ *leaves the basis*.

59. I will use an **ivector** *psi* to represent the basis array ψ; it is convenient to make *psi* indexed by $1\,..\,m$ with *psi*$[m]$ permanently set to 0.

⟨ Basic global variables 5 ⟩ +≡

```
ivector psi;    ▷ psi ≡ ψ
```

60. It is not difficult to find a row basis for E.

⟨ Compute row basis *psi* of E 60 ⟩ ≡

```
for (i = 1; i ≤ m; ++i) psi[i] = 0;
for (j = 1; j < n; ++j) {
  register int ii;
  for (i = 1; i < m ∧ E[i][j] ≡ 0L; ++i) ;
  if (i < m ∧ E[i][j] ≡ d ∧ psi[i] ≡ 0) {
    for (ii = i + 1; ii ≤ m ∧ E[ii][j] ≡ 0L; ++ii) ;
    if (ii > m) psi[i] = j;
  }
}
```

This code is used in section 104.

61. Once a basis is known, matrix F becomes redundant. It can be inferred from ψ and D at any moment: if $\psi[i] \equiv 0$ then $F[\,][i] = I[\,][i]$ else $F[\,][i] = D[\,][\psi[i]]$.

⟨ Compute F from psi and D 61 ⟩ ≡

for $(j = 1;\ j \le m;\ {+\!+}j)$
  if $(o, psi[j])$ **for** $(i = 1;\ i \le m;\ {+\!+}i)$ $oooo, F[i][j] = D[i][psi[j]];$
  else for $(i = 1;\ i \le m;\ {+\!+}i)$ $oo, F[i][j] = i \equiv j\ ?\ 1_{\mathrm{L}} : 0_{\mathrm{L}};$

This code is used in section 104.

62. Simplex algorithm with Bland's rule. In order to avoid cycling, the indices k and p must be chosen very carefully. Bland has shown that the following rule forces convergence: choose

the smallest possible k and the smallest possible $\psi[p]$

that are consistent with all the other requirements. The proof that the rule avoids cycling is not too easy.

We have already been choosing k according to Bland's rule anyway. The second part of the rule only makes a difference when there is a tie between two candidates for the rôle of p.

63. Since we keep explicit track of the row basis (vector *psi*), there no need to compute matrix F. The argument D will also be eliminated: we shall agree to set $E = D$ before calling *simplex_bland*.

⟨ Implementation with Bland's rule 63 ⟩ ≡

```
long simplex_bland(matrix E, int m, int n, matrix G, ivector psi, int *inf, int *unb) {
    int h, p, k, i, j;
    long d, sd, sdE;
    vector Eh, Ep, Ei, Gp, Gi;
    long Ehk, Ehn, Epk, Epn, Eik, Ein;
    ⟨ Set G = I, psi = 0, and d = 1 64 ⟩
    ⟨ Initialize num_its and maxmag 41 ⟩
    ⟨ Phase 1 of second implementation 65 ⟩
    ⟨ Phase 2 of second implementation 67 ⟩
}
```

This code is used in section 88.

64. ⟨ Set $G = I$, *psi* $= 0$, and $d = 1$ 64 ⟩ ≡

```
for (i = 1; i ≤ m; ++i) {
    o, Gi = G[i];
    for (j = 1; j ≤ m; ++j) o, Gi[j] = i ≡ j ? 1L : 0L;
}
for (i = 1; i ≤ m; ++i) o, psi[i] = 0;
d = 1L;
```

This code is used in sections 63 and 69.

65. The skeleton of phase 1 is the same as that of *simplex_1*, except that the code for updating the row and column bases takes the place of the code for updating F.

At the beginning of each iteration of phase 1 we have $Ein/d \geq 0$ for each i in $1 \mathinner{..} h{-}1$ (compare with 24.1). In particular, the inequality holds for each i such that $psi[i] \neq 0$, because our implementation of phase 1 examines the rows in increasing order.

```
⟨ Phase 1 of second implementation 65 ⟩ ≡
  h = 1;
  while (h < m) {
    llong t;
    ++num_its_ph1;
    ⟨ Choose a good column k; set k = n if no such column exists 43 ⟩        ▷ Bland's rule
    if (k < n) {
      ⟨ Choose p in 1 .. h according to Bland's rule 66 ⟩
      ⟨ Set p = h if pivoting about h and k is ok, after all 45 ⟩
      ⟨ Pivot about p and k and update d 50 ⟩
      o, psi[p] = k;      ▷ k enters the basis
      if (p ≡ h) ++h;
    }
    else if (o, Eh[n] ≡ 0_L) ++h;
    else ⟨ E is simple infeasible; return 46 ⟩
  }
```

This code is used in section 63.

66. This is the only part of phase 1 affected by Bland's rule; the effect of the rule appears in the condition governing a single **if**.

```
⟨ Choose p in 1 .. h according to Bland's rule 66 ⟩ ≡
  sd = d ≥ 0_L ? +1_L : −1_L;
  for (p = 1; p < h; ++p)
    if (oo, E[p][k] * sd > 0_L) break;
  ooo, Ep = E[p], Epk = Ep[k], Epn = Ep[n];
  for (i = p + 1; i < h; ++i) {
    ooo, Ei = E[i], Eik = Ei[k], Ein = Ei[n];
    if (Eik * sd > 0_L) {
      t = (llong) Epk * (llong) Ein − (llong) Eik * (llong) Epn;
      if (t < 0_LL ∨
      (t ≡ 0_LL ∧ (oo, psi[i] < psi[p])))      ▷ Bland's rule
        p = i, Ep = Ei, Epk = Eik, Epn = Ein;
    }
  }
```

This code is used in sections 65 and 67.

67. The skeleton of phase 2 is essentially the same as that of *simplex_1*.

⟨ Phase 2 of second implementation 67 ⟩ ≡

```
while (1) {        ▷ h ≡ m
  llong t;
  ++num_its_ph2;
  sd = d ≥ 0L ? +1L : −1L;
  o, Eh = E[h];
  for (k = 1; k < n ∧ (o, Eh[k] ∗ sd ≥ 0L); ++k) ;        ▷ Bland's rule
  if (k < n) {
    ⟨ Choose p in 1 .. h according to Bland's rule 66 ⟩
    if (p < h) {
      ⟨ Pivot about p and k and update d 50 ⟩
      o, psi[p] = k;
    }
    else ⟨ E is simple unbounded; return 49 ⟩
  }
  else ⟨ E is simple solvable; return 48 ⟩
}
```

This code is used in section 63.

68. The lexicographic version of Simplex. The *the lexicographic method* for forcing the convergence of Simplex depends on maintaining the set of column indices in a certain *order*. In other words, it depends on maintaining a certain *permutation* $c_1, c_2, \ldots, c_n$ of $1, 2, \ldots, n$. This permutation is not fixed: it must be readjusted at the end of some iterations. But we shall always have $c_1 \equiv n$.

The permutation $c_1, c_2, \ldots, c_n$ allows us to talk about a lexicographic order among vectors. Suppose x and y are two distinct vectors indexed by $1 \mathinner{..} n$. We say that x is *lexically smaller than* y if $x[c_l] < y[c_l]$, where l is the smallest index for which $x[c_l] \neq y[c_l]$.

The lexicographic method establishes a policy for breaking ties during the choice of p in phase 1 or phase 2. The rule requires that p be chosen so that

the vector $E[p]/Epn$ is lexically smallest.

69. Here is our version of the Simplex–Chio algorithm with the lexicographic rule.

⟨ Implementation with the lexicographic rule 69 ⟩ ≡

```
long simplex_lexicographic(matrix E, int m, int n, matrix G, ivector psi, int *inf, int *unb) {
  int h, p, k, i, j;
  long d, sd, sdE;
  vector Eh, Ep, Ei, Gp, Gi;
  long Ehk, Ehn, Epk, Epn, Eik;
  llong t;
  ivector c;
  int l, cl;
  ⟨ Set G = I, psi = 0, and d = 1 64 ⟩
  ⟨ Initialize permutation c 70 ⟩
  ⟨ Initialize num_its and maxmag 41 ⟩
  ⟨ Lexicographic implementation of phase 1 71 ⟩
  ⟨ Lexicographic implementation of phase 2 74 ⟩
}
```

This code is used in section 88.

70. We must set $c[1] = n$. The other entries in c are arbitrary. Actually, a lexicographic comparison never has to look at nonbasic columns; hence, there is no need to initialize $c[2 \mathinner{..} n]$.

⟨ Initialize permutation c 70 ⟩ ≡

```
c = allocate_ivector(n);
o, c[1] = n;
```

This code is used in section 69.

71. The skeleton of phase 1 is the same as that of our previous implementations of Simplex, except for the addition of code to update c everytime we pivot about row h. At the beginning of each iteration of phase 1, our permutation c has the following property for each basic row i:

$$\text{the vector } E[i][1\,..\,n]/d \text{ is lexically positive.} \tag{71.1}$$

(A vector $x[1\,..\,n]$ is *lexically positive* if $x[c_l] > 0$ for the smallest l such that $x[c_l] \neq 0$.) Since $c_1 \equiv n$, this property generalizes our invariant 24.1.

⟨ Lexicographic implementation of phase 1 71 ⟩ ≡

```
h = 1;
while (h < m) {
  ++num_its_ph1;
  ⟨ Choose a good column k; set k = n if no such column exists 43 ⟩
  if (k < n) {
    ⟨ Choose p in 1 .. h according to the lexicographic rule 73 ⟩
    ⟨ Set p = h if pivoting about h and k is ok, after all 45 ⟩
    ⟨ Pivot about p and k and update d 50 ⟩
    o, psi[p] = k;
    if (p ≡ h) {
      ⟨ Update permutation c 72 ⟩
      ++h;
    }
  }
  else if (o, Eh[n] ≡ 0_L) ++h;
  else ⟨ E is simple infeasible; return 46 ⟩
}
```

This code is used in section 69.

72. Suppose a pivoting operation was performed about row h and column k. Then h is a new basic row. Before starting a new iteration, we must update permutation c so that 71.1 remains valid. There is an easy way to do this: let all basic columns precede all nonbasic columns in $c_2, \ldots, c_n$. Actually, there is no need to record the nonbasic columns in c because no lexicographic comparison ever looks at nonbasic columns.

⟨ Update permutation c 72 ⟩ ≡

```
for (l = i = 1; i ≤ h; ++i)
  if (o, psi[i] ≠ 0) oo, c[++l] = psi[i];
```

This code is used in section 71.

73. The lexicographic search examines rows $E[i]$ and $E[p]$ and stops at the first l for which

$$E[i][c_l]/E[i][k] - E[p][c_l]/E[p][k] \neq 0. \tag{73.1}$$

For $l = 1$, the left-hand side of 73.1 becomes $E[i][n]/E[i][k] - E[p][n]/E[p][k]$. So, our lexicographic method begins with exactly the same test we have been doing since our first outline of Simplex; if this first comparison is an equality, we proceede with $l = 2$ and so on. Hence the lexicographic rule is a true generalization of the Simplex heuristic.

It is not too difficult to show that 73.1 holds for some $l \le n$; in fact, it holds for some l such that c_l is a basic column.

⟨ Choose p in 1 .. h according to the lexicographic rule 73 ⟩ ≡

```
sd = d ≥ 0_L ? +1_L : −1_L;
for (p = 1; p < h; ++p)
  if (oo, E[p][k] ∗ sd > 0_L) break;
oo, Ep = E[p], Epk = Ep[k];
for (i = p + 1; i < h; ++i) {
  oo, Ei = E[i], Eik = Ei[k];
  if (Eik ∗ sd > 0_L) {
    for (l = 1; ; ++l) {      ▷ lexicographic comparison
      o, cl = c[l];
      oo, t = (llong) Epk ∗ (llong) Ei[cl] − (llong) Eik ∗ (llong) Ep[cl];
      if (t ≠ 0_LL) break;
    }
    if (t < 0_LL) p = i, Ep = Ei, Epk = Eik;
  }
}
o, Epn = Ep[n];
```

This code is used in sections 71 and 74.

74. The skeleton of phase 2 is identical to that of our previous implementations of Simplex. The permutation c does not change during this phase. At the beginning of each iteration, for each basic row i, the vector $E[i][1..n]/d$ is lexically positive. Since $c_1 \equiv n$, this property generalizes our invariant 19.1.

⟨ Lexicographic implementation of phase 2 74 ⟩ ≡

```
while (1) {      ▷ h ≡ m
  ++num_its_ph2;
  sd = d ≥ 0_L ? +1_L : −1_L;
  o, Eh = E[h];
  for (k = 1; k < n ∧ (o, Eh[k] ∗ sd ≥ 0_L); ++k) ;
  if (k < n) {
    ⟨ Choose p in 1 .. h according to the lexicographic rule 73 ⟩
    if (p < h) {
      ⟨ Pivot about p and k and update d 50 ⟩
      o, psi[p] = k;
    }
    else ⟨ E is simple unbounded; return 49 ⟩
  }
  else ⟨ E is simple solvable; return 48 ⟩
}
```

This code is used in section 69.

75. Why does the lexicographic rule rule work? Suppose we are in phase 1 and supppose we are going through a sequence of iterations in which the size of the basis does not increase (i.e., $p < h$ in each iteration). Since the vector $E[i][1\,..\,n]/d$ is lexically positive for each basic row i, the value of $E[h][n]$ comes closer to zero after each pivot operation. This is ends up forcing the convergence.

76. Back to the minimization problem. Let's go back now to the basic minimization problem: find a vector x that minimizes $c \cdot x$ subject to

$$A \cdot x \equiv b \quad \text{and} \quad x \geq 0, \tag{76.1}$$

where A, b, and c are parts of the matrix D as indicated in 2.1. Suppose that we found a nonnull integer d and an invertible integer matrix G such that $G \cdot D$ is d-simple and $G[\,][m] \equiv d \cdot I[\,][m]$. Let $E = G \cdot D$.

77. Supppose E is d-simple infeasible. Then our minimization problem has no solution. To make this more evident, we shall exhibit a vector $v'[1\,..\,m{-}1]$ such that

$$\text{either} \quad v' \cdot A \leq 0 \text{ and } v' \cdot b > 0 \quad \text{or} \quad v' \cdot A \geq 0 \text{ and } v' \cdot b < 0.$$

We say that such v' is an "infeasibility vector". It constitutes a verifiable proof of the nonexistence of a vector x satisfying 76.1. The function below receives the index h of an infeasibility row in matrix E and produces an *integer* infeasibility vector *vprime*.

⟨ Solution of the minimization problem 77 ⟩ ≡

```
void infeasibility(matrix G, int h, vector vprime) {
  int i;
  for (i = 1; i < m; ++i) vprime[i] = G[h][i];
}
```

See also sections 78 and 79.

This code is used in section 88.

78. Now suppose E is d-simple solvable. It is a trivial job to find a vector x satisfying 76.1. In particular, there is only one such x satisfying the additional condition $x[j] \equiv 0$ for every nonbasic j. In order to certify the minimality of $c \cdot x$, it is sufficient to exhibit a vector $y[1\,..\,m{-}1]$ such that

$$y \cdot A \leq c \quad \text{and} \quad c \cdot x \equiv y \cdot b\,.$$

We say that such y is a "solution to the dual problem"; it proves the minimality of $c \cdot x$ because $c \cdot \check{x} \geq \check{y} \cdot b$ for all $\check{x}$ satisfying 76.1 and for all $\check{y}$ satisfying $\check{y} \cdot A \leq c$.

The function below receives E together with its row basis *psi*. Rather than producing x and y, it produces these vectors multiplied by d. We call the resulting vectors u and v:

$$u \;\equiv\; dx \qquad \text{and} \qquad v \;\equiv\; dy\,.$$

The vectors u and v are *integer*. If $d > 0$, then $u \geq 0$ and $v \cdot A \leq dc$. If $d < 0$, then $u \leq 0$ and $v \cdot A \geq dc$. In either case, $A \cdot u \equiv db$.

⟨ Solution of the minimization problem 77 ⟩ +≡

```
void solution(matrix E, ivector psi, vector u, vector v) {
  int i, j;
  for (j = 1; j < n; ++j) u[j] = 0L;
  for (i = 1; i < m; ++i)
    if (psi[i] ≠ 0) u[psi[i]] = E[i][n];
  for (i = 1; i < m; ++i) v[i] = −G[m][i];
}
```

79. Finally, suppose E is d-simple unbounded. In order to show that the minimization problem is unbounded, it is sufficient to exhibit a vector x satisfying 76.1 and a vector x' such that

$$A \cdot x' \equiv 0, \quad x' \geq 0, \quad \text{and} \quad c \cdot x' < 0.$$

Such a pair x, x' proves the unboundedness of the minimization problem because, for any number $\lambda > 0$, no matter how large, the vector $x + \lambda x'$ satisfies 76.1 and $c \cdot (x + \lambda x') \equiv \lambda\,(c \cdot x')$. We say that x' is an "unboundedness vector".

The following function receives the index k of the unboundedness column and a row basis *psi*. Rather than producing x and x', it produces the product of these vectors by d. We call the resulting vectors u and u':

$$u \;\equiv\; dx \qquad \text{and} \qquad u' \;\equiv\; dx'\,.$$

The vectors u and u' are *integer*. If $d > 0$, then $u \geq 0$, and $u' \geq 0$, and $c \cdot u' < 0$. If $d < 0$, then $u \leq 0$, and $u' \leq 0$, and $c \cdot u' > 0$. Argument *uprime* will play the role of u'.

⟨ Solution of the minimization problem 77 ⟩ +≡

```
void unboundedness(matrix E, int k, ivector psi, long d, vector u, vector uprime) {
  int i, j;
  for (j = 1; j < n; ++j) u[j] = uprime[j] = 0L;
  uprime[k] = d;
  for (i = 1; i < m; ++i)
    if (psi[i] ≠ 0) {
      u[psi[i]] = E[i][n];
      uprime[psi[i]] = −E[i][k];
    }
}
```

80. Printing routines. Unfortunately we must write a lot of code for the rather pedestrian job of printing our matrices.

81. We like to print our matrices so that each column is just wide enough to accomodate all entries in that column. The *findwidth* function receives a **matrix** $B[1\,..\,m][1\,..\,n]$ and sets $w[j]$ to the width of column j of B.

The width of a column is the width of its widest entry. We use *sprintf* to figure out the width of each entry. The expression *sprintf*(*buffer*, "%ld", b) places output (followed by the null character) in consecutive bytes starting at *buffer* and returns the number of characters transmitted (not including the null character). Since the magnitude of all our numbers is less than $2^{31} = 2{,}147{,}483{,}648$, *buffer* must have at least $1 + 10 + 1$ characters.

There is an exception, however: if the *given_width* option is nonnull then all entries of w are set to *given_width*.

⟨ Printing functions 81 ⟩ ≡

```
void findwidth(matrix B, int m, int n, ivector w) {
   int i, j, wi;
   char buffer[12];
   if (given_width)
      for (j = 1; j ≤ n; ++j) w[j] = given_width;
   else
      for (j = 1; j ≤ n; ++j)
         for (w[j] = 0, i = 1; i ≤ m; ++i) {
            wi = sprintf(buffer, "%ld", B[i][j]);      ▷ wi is width of Bij
            if (w[j] < wi) w[j] = wi;
         }
}
```

See also sections 82, 83, 84, 85, and 86.

This code is used in section 88.

82. The function *printmatrix* sends matrix $A[1\,..\,m][1\,..\,n]$ to the output file.

⟨ Printing functions 81 ⟩ +≡

```
void printmatrix(matrix A, int m, int n) {
   int i, j;
   ivector w;
   w = allocate_ivector(n);
   findwidth(A, m, n, w);
   fprintf(ofile, "\n");
   for (i = 1; i ≤ m; ++i) {
      fprintf(ofile, "\n");
      for (j = 1; j ≤ n; ++j) fprintf(ofile, "␣%*ld", w[j], A[i][j]);
   }
   deallocate_ivector(w);
}
```

83. We often wish to print the pair of matrices $G[1\,..\,m][1\,..\,m]$ and $E[1\,..\,m][1\,..\,n]$. We shall print them side-by-side if this can be done comfortably; otherwise, E will be printed after G.

⟨ Printing functions 81 ⟩ +≡

```
void printmatrices(matrix G, matrix E, int m, int n) {
   int i, j, totalw;
   ivector wG, wE;
   wG = allocate_ivector(m); wE = allocate_ivector(n);
   findwidth(G, m, m, wG); findwidth(E, m, n, wE);
   totalw = m + n;
   for (i = 1; i ≤ m; ++i) totalw += wG[i];
   for (j = 1; j ≤ n; ++j) totalw += wE[j];
   fprintf(ofile, "\n");
   if (totalw ≤ 85)     ▷ print G and E side-by-side
     for (i = 1; i ≤ m; ++i) {
       fprintf(ofile, "\n");
       for (j = 1; j ≤ m; ++j) fprintf(ofile, "␣%*ld", wG[j], G[i][j]);
       fprintf(ofile, "␣␣␣␣␣");
       for (j = 1; j ≤ n; ++j) fprintf(ofile, "␣%*ld", wE[j], E[i][j]);
     }
   else {     ▷ print G, then E
     for (i = 1; i ≤ m; ++i) {
       fprintf(ofile, "\n");
       for (j = 1; j ≤ m; ++j) fprintf(ofile, "␣%*ld", wG[j], G[i][j]);
     }
     fprintf(ofile, "\n");
     for (i = 1; i ≤ m; ++i) {
       fprintf(ofile, "\n");
       for (j = 1; j ≤ n; ++j) fprintf(ofile, "␣%*ld", wE[j], E[i][j]);
     }
   }
   deallocate_ivector(wG); deallocate_ivector(wE);
}
```

84. The user may wish to print the matrices G/d and E/d in *floating point format*. The following function prints E/d. The rows of E are indexed by $1\,..\,m$ and the columns by $1\,..\,n$.

⟨ Printing functions 81 ⟩ +≡

```
void printmatrix_float(matrix E, int m, int n, long d) {
   int i, j;
   fprintf(ofile, "\n");
   for (i = 1; i ≤ m; ++i) {
     fprintf(ofile, "\n");
     for (j = 1; j ≤ n; ++j) fprintf(ofile, "␣%9.2e", (float) E[i][j]/(float) d);
   }
}
```

85. The user may also wish to print the matrix E/d in *rational format*: given an integer matrix E and a nonnull integer d, we print, for each row i and column j, a pair x, y of integers such that

$$x/y = E[i][j]/d \ ,$$
y has no common divisors with x ,
$$y > 0 \ .$$

The number x is the *numerator* and y is the *denominator* of entry i, j.

Before handling the printing, we must write Euclid's algoritm to find the greatest common divisor of integers x and y. We assume that $x \geq 0$ and $y > 0$. (If both x and y were 0 then there would be no *maximum* common divisor, and our algorithm will go into a loop.) The observation that explains the workings of Euclid's algorithm is this: if $x > y$ then $gcd(x, y) \equiv gcd(x \,\%\, y, x)$.

⟨ Printing functions 81 ⟩ +≡

```
long euclid(long x, long y) {
  long t;
  do {
    x = x % y;
    t = x, x = y, y = t;
  } while (y ≠ 0L);
  return x;
}
```

86. The function *printmatrix_rational* will print the matrix E/d in rational format. We shall store the numerators of all entries of E in a matrix X and the denominators in a matrix Y.

We add one more embellishment: For each entry of the form x/y, if $x \equiv 0$ or $y \equiv 1$, we supress $/y$ and write only x. If all entries in a column are of this kind, the width of the y-part of the column must be adjusted.

The function will also compute (just for the record) the largest magnitude, $*mX$, among all entries of X and the largest magnitude, $*mY$, among all entries of Y.

⟨ Printing functions 81 ⟩ +≡

```
void printmatrix_rational(matrix E, int m, int n, long d, long *mX, long *mY) {
  matrix X, Y;
  ivector wX, wY;
  long maxX = 0L, maxY = 0L;
  int i, j;
  X = allocate_matrix(m, n); Y = allocate_matrix(m, n);
  ⟨ Compute matrices X and Y 87 ⟩
  wX = allocate_ivector(n); wY = allocate_ivector(n);
  findwidth(X, m, n, wX); findwidth(Y, m, n, wY);
  for (j = 1; j ≤ n; ++j)
    if (wY[j] ≡ 1) {
      for (i = 1; i ≤ m ∧ Y[i][j] ≡ 1; ++i) ;
      if (i > m) wY[j] = 0;    ▷ adjust wY in case all entries are 1
    }
  fprintf(ofile, "\n");
  for (i = 1; i ≤ m; ++i) {
    fprintf(ofile, "\n");
    for (j = 1; j ≤ n; ++j)
      if (X[i][j] ≡ 0L ∨ Y[i][j] ≡ 1L) fprintf(ofile, "␣%*ld␣%-*s", wX[j], X[i][j], wY[j], "");
      else fprintf(ofile, "␣%*ld/%-*ld", wX[j], X[i][j], wY[j], Y[i][j]);
  }
  deallocate_matrix(X); deallocate_matrix(Y);
  deallocate_ivector(wX); deallocate_ivector(wY);
  *mX = maxX; *mY = maxY;
}
```

87. ⟨ Compute matrices X and Y 87 ⟩ ≡ {

```
long sd, magd;   ▷ sign of d and magnitude of d
long sEij, magEij;   ▷ sign of Eij and magnitude of Eij
long gcd, Xij, Yij;
sd = +1L, magd = d;
if (magd < 0L) sd = -1L, magd = -magd;
for (i = 1; i <= m; ++i) {
  for (j = 1; j <= n; ++j) {
    sEij = +1L, magEij = E[i][j];
    if (magEij < 0L) sEij = -sEij, magEij = -magEij;
    gcd = euclid(magEij, magd);
    Xij = magEij / gcd;
    if (maxX < Xij) maxX = Xij;
    Yij = magd / gcd;
    if (maxY < Yij) maxY = Yij;
    X[i][j] = sd * sEij * Xij;
    Y[i][j] = Yij;
  }
}
}
```

This code is used in section 86.

88. MAIN. Our C program has the following structure:

```
#define PROGRAM "Simplex-Chio-Cramer-Edmonds␣algorithm"
#define AUTHOR "Paulo␣Feofiloff"
#define DATE "5/3/98"
  ⟨ Header files 119 ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Typedefs 7 ⟩
  ⟨ Basic global variables 5 ⟩
  ⟨ Other global variables 12 ⟩
  ⟨ Memory allocation functions 8 ⟩
  ⟨ Printing functions 81 ⟩
  ⟨ Auxiliary functions 35 ⟩
  ⟨ The basic heuristic 17 ⟩
  ⟨ First implementation of the heuristic 37 ⟩
  ⟨ Implementation with Bland's rule 63 ⟩
  ⟨ Implementation with the lexicographic rule 69 ⟩
  ⟨ Solution of the minimization problem 77 ⟩
  void main(int argc, char *argv[]) {
    ⟨ Local variables 90 ⟩
    ⟨ Process command line 89 ⟩
    ⟨ Open input and output files 94 ⟩
    ⟨ Read data 96 ⟩
    ⟨ Run the algorithm 104 ⟩
    ⟨ Print results 109 ⟩
    ⟨ Check results 120 ⟩
    ⟨ Close files 117 ⟩
  }
```

89. *Command line: name of input file.* Our input file must have an "`.in`" suffix. The corresponding prefix must be the first argument on the command line. The name of the output file will consist of the same prefix followed by "`.out`". We assume the prefix has no more than 20 characters.

```
⟨ Process command line 89 ⟩ ≡
  fprintf(stdout, "\n***␣%s", PROGRAM);
  if (argc ≥ 2) {
    if (strlen(argv[1]) > 20)
      early_quit("Prefix␣of␣input␣file␣must␣have␣at␣most␣20␣characters!");
    sprintf(ifilename, "%s.in", argv[1]);
    sprintf(ofilename, "%s.out", argv[1]);
  }
```

See also section 93.

This code is used in section 88.

90. The name of the input file will be at most 24 characters long: 20 for the prefix, 3 more for the "`.in`", plus 1 for the null character. The name of the output file will be at most 25 characters long.

We take this opportunity to declare a few other *factotum* local variables.

⟨ Local variables 90 ⟩ ≡

char *ifilename*[24];
char *ofilename*[25];
int *i*, *j*;
long *t*;

See also sections 91, 97, 99, 103, and 111.

This code is used in section 88.

91. *Command line: options.* The remaining arguments on the command line specify a few options. In order to choose one of the implementations of the algorithm, the user must say

`-b` to set *implem* = *bland*;
`-l` to set *implem* = *lexicographic*;

The default value of *implem* is *heuristic*. The program interprets *implem* as follows: if *implem* ≡ *heuristic* then *simplex_1* will be executed; if *implem* ≡ *bland* then *simplex_bland* will be executed; if *implem* ≡ *lexicographic* then *simplex_lexicographic* will be executed.

⟨ Local variables 90 ⟩ +≡

enum {
heuristic, *bland*, *lexicographic*
} *implem*;

92. The other options will give the user control over how matrices are printed. When *verbose* is *on*, the program will print matrices G e E after each pivoting operation. When *want_float* is *on*, the program will print the matrices G/d and E/d in floating point format at the end of the run. When *want_rational* is *on*, the program will print the matrices G/d and E/d in rational format at the end of the run. When *given_width* $\neq 0$, the program pretends that each entry of each matrix can be printed using at most *given_width* characters; if *given_width* $\equiv 0$, our program will compute the width of each column of each matrix. The user must say

`-v` to set *verbose* = *on*;
`-r` to set *want_rational* = *on*;
`-f` to set *want_float* = *on*;
`-w`W to set *given_width* = W.

(These option variables are global because they must be visible not only to *main* but also to some of the other functions.)

⟨ Other global variables 12 ⟩ +≡

enum { *off*, *on* } *verbose*, *want_rational*, *want_float*;
int *given_width*;

93. ⟨ Process command line 89 ⟩ +≡

```
if (argc ≡ 1) {
  fprintf(stderr, "\n\n␣Type␣prefix␣of␣input␣file␣followed␣by␣options:");
  fprintf(stderr, "\n␣-b␣␣...␣Bland's␣rule");
  fprintf(stderr, "\n␣-l␣␣...␣lexicographic␣method");
  fprintf(stderr, "\n␣-v␣␣...␣verbose");
  fprintf(stderr, "\n␣-r␣␣...␣print␣output␣matrices␣in␣rational␣format");
  fprintf(stderr, "\n␣-f␣␣...␣print␣output␣matrices␣in␣floating␣point␣format");
  fprintf(stderr, "\n␣-wW␣...␣on␣output,␣assume␣each␣entry␣is␣W␣characters␣wide\n\n");
  exit(1);
}
implem = heuristic;
verbose = want_rational = want_float = off;
given_width = 0;
while (−− argc > 1)
  if (sscanf(argv[argc], "-w%d", &given_width) ≡ 1) ;
  else if (strcmp(argv[argc], "-b") ≡ 0) implem = bland;
  else if (strcmp(argv[argc], "-l") ≡ 0) implem = lexicographic;
  else if (strcmp(argv[argc], "-v") ≡ 0) verbose = on;
  else if (strcmp(argv[argc], "-r") ≡ 0) want_rational = on;
  else if (strcmp(argv[argc], "-f") ≡ 0) want_float = on;
```

94. Having processed the command line, we are ready to open and prepare the input and output files.

⟨ Open input and output files 94 ⟩ ≡

```
ifile = fopen(ifilename, "r");
if (ifile ≡ Λ) early_quit("Unable␣to␣open␣the␣input␣file!");
ofile = fopen(ofilename, "a");
if (ofile ≡ Λ) early_quit("Unable␣to␣open␣the␣output␣file!");
fprintf(ofile, "\n***␣%s␣(by␣%s,␣%s)", PROGRAM, AUTHOR, DATE);
fprintf(ofile, "\n***␣Given␣an␣integer␣matrix␣D,");
fprintf(ofile, "\n***␣finds␣a␣nonnull␣integer␣d␣and␣integer␣matrices␣F␣and␣G");
fprintf(ofile, "\n***␣such␣that␣G*D␣is␣d-simple,␣F*G␣=␣d*I,␣and␣G[][m]␣=␣d*I[][m]");
fprintf(ofile, "\n***␣All␣matrix␣entries␣are␣smaller␣than␣%lld", TWO31);   ▷ 2^31
```

This code is used in section 88.

95. ⟨ Other global variables 12 ⟩ +≡

```
FILE *ifile;   ▷ input file
FILE *ofile;   ▷ output file
```

96. *Input file: first line.* The first line in the input file contains a sequence of at most 126 arbitrary characters (this is used as a caption for the file); it will be stored in string *caption*. To avoid problems with a line longer than 126 characters, we shall read it using the *fgets* function. The command *fgets*(*caption*, *k*, *infile*) copies characters from *infile* to *caption* until (1) a `'\n'` character is copied, or (2) the end of the file is reached, or (3) $k-1$ characters have been copied before a `'\n'` or the end of the file have been found. A null character is placed in *caption* after the last character read. If the end of the file is encountered *before any character has been read*, the contents of *caption* is undefined and *fgets* returns Λ. Otherwise, *fgets* returns *caption*.

⟨ Read data 96 ⟩ ≡

```
if (fgets(caption, 128, ifile) ≡ Λ) quit("Input␣file␣is␣empty!");
caption[127] = '\n';
for (i = 0; caption[i] ≠ '\n'; ++i) ;
if (i ≥ 127) quit("Something␣wrong␣with␣first␣line␣of␣input␣file!");
caption[i] = '\0';
fprintf(stdout, "\n\n␣Caption␣of␣input␣file:\n␣\"%s\"", caption);
fprintf(ofile, "\n\n␣Caption␣of␣input␣file:\n␣\"%s\"", caption);
```

See also sections 98, 100, and 101.
This code is used in section 88.

97. ⟨ Local variables 90 ⟩ +≡

```
char caption[128];
```

98. *Input file: second line.* The second line of the input file must contain the values of m and n. We *assume* each is greater than 0 and less than `INT_MAX`, which on our system has value $2^{31}-1$.

⟨ Read data 96 ⟩ +≡

```
fscanf(ifile, "%d%d", &m, &n);
fprintf(ofile, "\n␣%d␣rows␣and␣%d␣columns\n", m, n);
```

99. *Input file: remaining lines.* From the third line on we expect to find a matrix D with rows indexed by $1\,..\,m$ and columns indexed by $1\,..\,n$. How are these rows and columns arranged on the lines of the file? The entries of matrix D could be arranged in the most obvious and natural way: *D11*, *D12*, … on the third line of the file, *D21*, *D22*, … on the fourth line, and so on, where *Dij* stands for $D[i][j]$. We shall, however, adopt a more elaborate arrangement. Suppose, for example, that $m \equiv 3$ and $n \equiv 4$; then the input file will have the following form from the third line on (with $r_3 \equiv 3$ and $c_4 \equiv 4$).

$$
\begin{array}{ccccc}
 & c_1 & c_2 & c_3 & c_4 \\
r_1 & Dr_1c_1 & Dr_1c_2 & Dr_1c_3 & Dr_1c_4 \\
r_2 & Dr_2c_1 & Dr_2c_2 & Dr_2c_3 & Dr_2c_4 \\
r_3 & Dr_3c_1 & Dr_3c_2 & Dr_3c_3 & Dr_3c_4
\end{array}
$$

⟨ Local variables 90 ⟩ +≡

```
ivector r, c, check;
int cj, ri;
```

100. First, we read $c_1, c_2, \ldots, c_n$. This must be a permutation of $1, \ldots, n$ (usually $c_n \equiv n$). We *assume* each is greater than 0 and less than `INT_MAX`, which on our system has value $2^{31}-1$.

⟨ Read data 96 ⟩ +≡

```
c = allocate_ivector(n);
check = allocate_ivector(n); for (j = 1; j ≤ n; ++j) check[j] = 0;
for (j = 1; j ≤ n; ++j) {
  fscanf(ifile, "%d", &cj);
  if (cj < 1 ∨ cj > n) quit("Bad column label.");
  if (check[cj] ≠ 0) quit("Column labels not a permutation of 1,..,n.");
  check[cj] = 1;
  c[j] = cj;
}
deallocate_ivector(check);
```

101. Now we read the remaining lines. Remember that $r_1, \ldots, r_m$ must be a permutation of $1, \ldots, m$ (usually $r_m \equiv m$). We *assume* each is greater than 0 and less than `INT_MAX`.

Even though $D[m][n]$ is irrelevant, the input file must supply a value for this entry.

⟨ Read data 96 ⟩ +≡

```
r = allocate_ivector(m);
check = allocate_ivector(m); for (i = 1; i ≤ m; ++i) check[i] = 0;
D = allocate_matrix(m, n);
for (i = 1; i ≤ m; ++i) {
  fscanf(ifile, "%d", &ri);
  if (ri < 1 ∨ ri > m) quit("Bad row label.");
  if (check[ri] ≠ 0) quit("Row labels not a permutation of 1,..,m.");
  check[ri] = 1, r[i] = ri;
  for (j = 1; j ≤ n; ++j) ⟨ Read entry of D belonging to row r[i] and column c[j] 102 ⟩
}
deallocate_ivector(check);
fprintf(ofile, "\n Order of columns in input file:\n");
for (j = 1; j ≤ n; ++j) fprintf(ofile, " %d", c[j]);
fprintf(ofile, "\n Order of rows in input file:\n");
for (i = 1; i ≤ m; ++i) fprintf(ofile, " %d", r[i]);
deallocate_ivector(c); deallocate_ivector(r);
```

102. The magnitude of all entries of matrix D must be strictly smaller than 2^{31}, which corresponds to 2,147,483,648 in decimal notation. Hence, any integer with 9 of fewer decimal digits is of the desired kind. Any such number will fit comfortably in a **long** variable.

Suppose, for one moment, that one of the entries of D in the input file has more than 9 decimal digits. If we were to read the input file without any precautions, the most significant digits of that entry would be truncated, without any warning, during the read operation. To avoid this, we shall read the matrix entries as strings and convert them to integers after checking that they are not too long. The conversion uses the standard function *atol*.

(We could simplify this piece of code by using *strtol* instead of *atol*. The manual says that *strtol* returns `LONG_MAX` or `LONG_MIN` if the input string is too long; but then the manual adds "*strtol*( ) no longer accepts values greater than `LONG_MAX` as valid input".)

#define `MAX_DIGITS` 9

⟨ Read entry of D belonging to row $r[i]$ and column $c[j]$ 102 ⟩ ≡ {

```
fscanf(ifile, "%s", buffer);
if (strlen(buffer) ≤ MAX_DIGITS ∨ (strlen(buffer) ≡ MAX_DIGITS + 1 ∧ buffer[0] ≡ '-'))
  D[r[i]][c[j]] = atol(buffer);
else quit("The␣magnitude␣of␣some␣entry␣in␣the␣data␣matrix␣is␣too␣large!");
}
```

This code is used in section 101.

103. We shall *assume* that no entry of D in the input file occupies more than 127 characters.

⟨ Local variables 90 ⟩ +≡

```
char buffer[128];
```

104. *Run the algorithm.* We allocate space for all our matrices and vectors and then run the implementation dictated by the variable *implem*.

⟨ Run the algorithm 104 ⟩ ≡

```
fprintf(ofile, "\n\n␣Data␣matrix␣D:");
printmatrix(D, m, n);
E = allocate_matrix(m, n);
G = allocate_matrix(m, m);
F = allocate_matrix(m, m);
psi = allocate_ivector(m);
mems = 0L;
switch (implem) {
case heuristic:
  fprintf(ofile, "\n\n\n␣Running␣Simplex-Chio␣HEURISTIC␣(no␣convergence␣rule)");
  d = simplex_1(D, m, n, F, G, E, &h, &k);
  if (d ≡ LONG_MIN) quit("Aborted␣due␣to␣overflow␣during␣pivoting!");
  ⟨ Compute row basis psi of E 60 ⟩
  break;
case bland:
  fprintf(ofile, "\n\n\n␣Running␣Simplex-Chio␣algorithm␣with␣BLAND's␣rule");
  ⟨ Make E = D 105 ⟩
  d = simplex_bland(E, m, n, G, psi, &h, &k);
  if (d ≡ LONG_MIN) quit("Aborted␣due␣to␣overflow␣during␣pivoting!");
  ⟨ Compute F from psi and D 61 ⟩
  break;
case lexicographic:
  fprintf(ofile, "\n\n\n␣Running␣LEXICOGRAPHIC␣version␣of␣Simplex-Chio␣algorithm");
  ⟨ Make E = D 105 ⟩
  d = simplex_lexicographic(E, m, n, G, psi, &h, &k);
  if (d ≡ LONG_MIN) quit("Aborted␣due␣to␣overflow␣during␣pivoting!");
  ⟨ Compute F from psi and D 61 ⟩
}
⟨ Print maxmag and num_its 106 ⟩
⟨ Print mems 107 ⟩
⟨ Print omega 108 ⟩
```

This code is used in section 88.

105. ⟨ Make $E = D$ 105 ⟩ ≡

```
for (i = 1; i ≤ m; ++i) {
  register vector Ei, Di;
  oo, Ei = E[i], Di = D[i];
  for (j = 1; j ≤ n; ++j) oo, Ei[j] = Di[j];
}
```

This code is used in section 104.

106. ⟨ Print *maxmag* and *num_its* 106 ⟩ ≡

```
fprintf(ofile, "\n\n␣Largest␣magnitude␣among␣entries␣of␣G␣and␣E");
fprintf(ofile, "\n␣throughout␣the␣execution␣of␣the␣algorithm:␣%lld", maxmag);
fprintf(ofile, "\n␣Number␣of␣iterations␣in␣phase␣1:␣%lld", num_its_ph1);
fprintf(ofile, "\n␣Number␣of␣iterations␣in␣phase␣2:␣%lld", num_its_ph2);
```

This code is used in section 104.

107. The Simplex–Chio algorithm executes at most $5m^2(m+n)$ arithmetic operations; I guess the number of memory accesses in our *simplex* functions is also in the order of $m^2(m+n)$.

⟨ Print *mems* 107 ⟩ ≡

```
fprintf(ofile, "\n␣Number␣of␣mems␣used␣by␣the␣simplex␣routine:");
fprintf(ofile, "␣%ld␣=␣%4.2f␣m*m*(m+n)", mems, (float) mems/(float) (m ∗ m ∗ (m + n)));
```

This code is used in section 104.

108. To satisfy the curiosity of the user, we print the *a priori* bounds on the magnitude of the matrix entries encountered in the course of computations.

⟨ Print *omega* 108 ⟩ ≡

```
t = omega1(D, m, n);
if (t ≡ LONG_MIN) fprintf(ofile, "\n␣Curiosity:␣omega1␣>␣LONG_MAX");
else fprintf(ofile, "\n␣Curiosity:␣omega1␣=␣%ld", t);
t = omega2(D, m, n);
if (t ≡ LONG_MIN) fprintf(ofile, "␣␣and␣␣omega2␣>␣LONG_MAX␣");
else fprintf(ofile, "␣␣and␣␣omega2␣=␣%ld␣", t);
```

This code is used in section 104.

109. *Print results.* If the *verbose* option is *on*, matrices G and E have already been printed from within the *simplex* funcitons, so we don't print them again.

⟨ Print results 109 ⟩ ≡

```
if (verbose ≡ off) {
  fprintf(ofile, "\n\n␣Matrices␣G␣and␣E␣=␣G*D:");
  printmatrices(G, E, m, n);
}
else {
  fprintf(ofile, "\n\n␣Matrix␣F:");
  printmatrix(F, m, m);
}
if (want_rational ≡ on) ⟨ Print matrices in rational format 110 ⟩
if (want_float ≡ on) ⟨ Print matrices in floating point format 112 ⟩
if (h ≠ 0) ⟨ Deal with infeasible case 113 ⟩
else if (k ≠ 0) ⟨ Deal with unbounded case 114 ⟩
else ⟨ Deal with solvable case 115 ⟩
```

This code is used in section 88.

110. If *want_rational* is *on*, we are expected to print the entries of matrices G/d and E/d as explicit fractions, with relatively prime numerator and denominator.

⟨ Print matrices in rational format 110 ⟩ ≡

```
{
  fprintf(ofile, "\n\n\n␣Matrices␣G/d␣and␣E/d␣in␣rational␣format:");
  printmatrix_rational(G, m, m, d, &mGX, &mGY);
  printmatrix_rational(E, m, n, d, &mEX, &mEY);
  fprintf(ofile, "\n\n␣Largest␣magnitude␣of␣numerators␣in␣G␣and␣E:␣%ld",
      mGX ≥ mEX ? mGX : mEX);
  fprintf(ofile, "\n␣Largest␣magnitude␣of␣denominators:␣␣␣␣␣␣␣␣␣␣%ld",
      mGY ≥ mEY ? mGY : mEY);
}
```

This code is used in section 109.

111. ⟨ Local variables 90 ⟩ +≡

```
long mGX, mGY, mEX, mEY;
```

112. If *want_float* is *on*, we are expected to print the entries of matrices G/d and E/d in floating point format.

⟨ Print matrices in floating point format 112 ⟩ ≡

```
{
    fprintf(ofile, "\n\n\n␣Matrices␣G/d␣and␣E/d␣in␣floating␣point␣format:");
    printmatrix_float(G, m, m, d);
    printmatrix_float(E, m, n, d);
  }
```

This code is used in section 109.

113. ⟨ Deal with infeasible case 113 ⟩ ≡

```
{
    fprintf(ofile, "\n\n\n␣Matrix␣E␣is␣d-simple␣infeasible,␣with␣d␣=␣%ld", d);
    fprintf(ofile, "␣(infeasibility␣row␣is␣%d)", h);
    fprintf(ofile, "\n\n␣Let␣A␣=␣D[1..%d][1..%d]␣and␣b␣=␣D[1..%d][%d]", m − 1, n − 1, m − 1, n);
    fprintf(ofile, "\n␣and␣consider␣the␣problem␣of");
    fprintf(ofile, "␣minimizing␣c*x␣subject␣to␣A*x␣=␣b␣and␣x␣>=␣0");
    fprintf(ofile, "\n␣The␣vector␣y'␣below␣proves␣infeasibility␣of␣the␣problem");
    if (E[h][n] > 0L) fprintf(ofile, "\n␣(since␣␣y'*A␣<=␣0␣␣and␣␣y'*b␣>␣0)");
    else fprintf(ofile, "\n␣(since␣␣y'*A␣>=␣0␣␣and␣␣y'*b␣<␣0)");
    vprime = allocate_vector(m − 1);
    infeasibility(G, h, vprime);
    fprintf(ofile, "\n\n␣␣␣␣␣␣y'␣=␣");
    for (i = 1; i < m; ++i) fprintf(ofile, "␣%ld", vprime[i]);
  }
```

This code is used in section 109.

114. ⟨ Deal with unbounded case 114 ⟩ ≡

```
{
    fprintf(ofile, "\n\n\n␣Matrix␣E␣is␣d-simple␣unbounded,␣with␣d␣=␣%ld", d);
    fprintf(ofile, "␣(unboundedness␣column␣is␣%d)", k);
    fprintf(ofile, "\n\n␣Let␣A␣=␣D[1..%d][1..%d],␣b␣=␣D[1..%d][%d],␣and␣c␣=␣D[%d][1..%d]",
        m − 1, n − 1, m − 1, n, m, n − 1);
    fprintf(ofile, "\n␣Consider␣the␣problem␣of");
    fprintf(ofile, "␣minimizing␣c*x␣subject␣to␣A*x␣=␣b␣and␣x␣>=␣0");
    fprintf(ofile, "\n␣The␣vectors␣x␣and␣x'␣below␣prove␣unboundedness");
    fprintf(ofile, "\n␣(since␣A*x␣=␣b,␣␣x␣>=␣0,␣␣A*x'␣=␣0,␣␣x'␣>=␣0,␣and␣␣c*x'␣<␣0)");
    u = allocate_vector(n − 1);
    uprime = allocate_vector(n − 1);
    unboundedness(E, k, psi, d, u, uprime);
    fprintf(ofile, "\n\n␣␣␣␣␣␣␣d*x␣=␣");
    for (j = 1; j < n; ++j) fprintf(ofile, "␣%ld", u[j]);
    fprintf(ofile, "\n␣␣␣␣␣␣d*x'␣=␣");
    for (j = 1; j < n; ++j) fprintf(ofile, "␣%ld", uprime[j]);
  }
```

This code is used in section 109.

115. ⟨ Deal with solvable case 115 ⟩ ≡ {

```
fprintf(ofile, "\n\n\n␣Matrix␣E␣is␣d-simple␣solvable,␣with␣d␣=␣%ld", d);
fprintf(ofile, "\n\n␣Let␣A␣=␣D[1..%d][1..%d],␣b␣=␣D[1..%d][%d]␣and␣c␣=␣D[%d][1..%d]",
    m - 1, n - 1, m - 1, n, m, n - 1);
fprintf(ofile, "\n␣Consider␣the␣problem␣of");
fprintf(ofile, "␣minimizing␣c*x␣subject␣to␣A*x␣=␣b␣and␣x␣>=␣0");
fprintf(ofile, "\n␣The␣vector␣x␣below␣solves␣the␣problem");
fprintf(ofile, "␣and␣y␣proves␣the␣minimality␣of␣c*x");
fprintf(ofile, "\n␣(since␣A*x␣=␣b,␣␣x␣>=␣0,␣␣y*A␣<=␣c,␣␣and␣␣c*x␣=␣y*b)");
u = allocate_vector(n - 1);
v = allocate_vector(m - 1);
solution(E, psi, u, v);
fprintf(ofile, "\n\n␣␣␣␣␣␣d*x␣=␣");
for (j = 1; j < n; ++j) fprintf(ofile, "␣%ld", u[j]);
fprintf(ofile, "\n␣␣␣␣␣␣d*y␣=␣");
for (i = 1; i < m; ++i) fprintf(ofile, "␣%ld", v[i]);
}
```

This code is used in section 109.

116. ⟨ Other global variables 12 ⟩ +≡

```
vector u, v, uprime, vprime;
```

117. ⟨ Close files 117 ⟩ ≡

```
fclose(ifile);
fprintf(stdout, "\n␣Output␣was␣sent␣to␣file␣%s\n\n", ofilename);
fprintf(ofile, "\n\n\f\n\n");
fclose(ofile);
deallocate_matrix(D);
deallocate_matrix(E);
deallocate_matrix(G);
deallocate_matrix(F);
deallocate_ivector(psi);
```

This code is used in section 88.

118. The *quit* and *early_quit* functions print a message before aborting the program.

⟨ Auxiliary functions 35 ⟩ +≡

```
void quit(char *message) {
  fprintf(stderr, "\n\n␣%s\a\n\n", message);
  fprintf(ofile, "\n\n␣%s\n\n\n\n\f\n\n", message);
  exit(1);
}
void early_quit(char *message) {
  fprintf(stderr, "\n\n␣%s\a\n\n", message);
  exit(1);
}
```

119. We must include `stdio.h` because we are using *stderr*, *fprintf*, *sprintf*, etc. We must include `stdlib.h` bacause we use *malloc*, *free* and *atol*. Our use of `LONG_MIN` requires the inclusion of `limits.h`. Our use of *strlen* and *strcmp* requires the inclusion of `string.h`.

Attention! The *llabs* function (header file `stdlib.h`) does not seem to work properly. Its prototype is **extern llong** *llabs*(**llong**); it should return the absolute value of its argument; yet, $llabs(2_{\mathrm{LL}}) \equiv 4294967294$. Just in case, we also avoid using *abs* and *labs*.

⟨ Header files 119 ⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
```

This code is used in section 88.

120. Checking the results. *(This part of the code should be deleted after the program has been properly debugged.)* First, we check results of the reformulated problem; a little later, we shall check the results of the minimization problem.

⟨ Check results 120 ⟩ ≡

```
check_G_m();
check_F_times_G();
check_G_times_D();
if (h ≠ 0) check_inf();
else check_unb_or_solv();
```

See also section 126.

This code is used in section 88.

121. First, we check the identity $G[\,][m] \equiv d \cdot I[\,][m]$. The inputs for function *check_G_m* are the global variables G, m, and d.

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_G_m(void) {
  int i;
  for (i = 1; i < m; ++i)
    if (G[i][m] ≠ 0L) goto wrong;
  if (G[m][m] ≠ d) goto wrong;
  return;
wrong:
  fprintf(stderr, "\n\n ERROR!\a\n");
  fprintf(ofile, "\n\n ERROR! G[][m] != dI[][m]\n");
  return;
}
```

122. Next, we check the identity $F \cdot G \equiv d \cdot I$. The inputs for function *check_F_times_G* are the global variables F, G, m, n and d. We shall do the calculations in **llong** arithmetic in order to detect a possible **long** overflow. We must remember that the magnitude of each entry of matrix D is strictly smaller than 2^{31}. The magnitude of d and each entry of F and G satisfies a similar bound.

```
#define TWO62 #4000000000000000LL    ▷ 2^62
⟨ Auxiliary functions 35 ⟩ +≡
  void check_F_times_G(void) {
    int i, j, k;
    llong dd;
    llong term, sum;
    for (i = 1; i ≤ m; ++i)
      for (j = 1; j ≤ m; ++j) {
        sum = 0LL;
        for (k = 1; k ≤ m; ++k) {
          term = (llong) F[i][k] * (llong) G[k][j];
          sum += term;
          if (sum ≥ TWO62) goto oflo;
        }
        if (i ≡ j) dd = (llong) d; else dd = 0LL;
        if (sum ≠ dd) goto wrong;
      }
    return;
  oflo:
    fprintf(stderr, "\n\n Overflow WARNING!\a");
    fprintf(ofile, "\n\n WARNING! Unable to compute F*G without overflow");
    return;
  wrong:
    fprintf(stderr, "\n\n ERROR!\a\n");
    fprintf(ofile, "\n\n ERROR! F*G != d*I\n");
    return;
  }
```

123. Now we check the identity $G \cdot D \equiv E$. The inputs for function *check_G_times_D* are the global variables G, D, E, m and n.

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_G_times_D(void) {
  int i, j, k;
  llong term, sum;
  for (i = 1; i ≤ m; ++i)
    for (j = 1; j ≤ n; ++j) {
      sum = 0LL;
      for (k = 1; k ≤ m; ++k) {
        term = (llong) G[i][k] * (llong) D[k][j];
        sum += term;
        if (sum ≥ TWO62) goto oflo;
      }
      if (sum ≠ (llong) E[i][j]) goto wrong;
    }
  return;
oflo:
  fprintf(stderr, "\n\n␣Overflow␣WARNING!\a");
  fprintf(ofile, "\n\n␣WARNING!␣Unable␣to␣compute␣G*D␣without␣overflow");
  return;
wrong:
  fprintf(stderr, "\n\n␣ERROR!\a\n");
  fprintf(ofile, "\n\n␣ERROR!␣G*D␣!=␣E\n");
  return;
}
```

124. If h is nonnull then E should be simple infeasible. This is very easy to check. The inputs for function *check_inf* are the global variables E, m, n, and h. We assume that $0 < h < m$.

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_inf(void) {
  int j;
  if (E[h][n] ≡ 0L) goto wrong;
  if (E[h][n] < 0L) {
    for (j = 1; j < n; ++j)
      if (E[h][j] < 0L) goto wrong;
  }
  else {
    for (j = 1; j < n; ++j)
      if (E[h][j] > 0L) goto wrong;
  }
  return;
wrong:
  fprintf(stderr, "\n\n␣ERROR!\a");
  fprintf(ofile, "\n\n␣ERROR!␣Matrix␣E␣is␣NOT␣simple");
  return;
}
```

125. We assume here that $h \equiv 0$ and $0 \le k < n$. If $k \ne 0$ then k should be the index of an unboundedness column (and therefore E should be simple unbounded); otherwise, E should be simple solvable. The inputs for function *check_unb_or_solv* are the global variables E, m, n, d, k, and the row basis vector *psi*.

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_unb_or_solv(void) {
  int i, j;
  long sd;
  sd = d > 0L ? +1L : −1L;
  for (i = 1; i < m; ++i)
    if ((psi[i] ≡ 0 ∧ E[i][n] ≠ 0L) ∨ (psi[i] ≠ 0 ∧ E[i][n] ∗ sd < 0L)) goto wrong;    ▷ bad column n
  if (k ≠ 0) {
    for (i = 1; i < m; ++i)
      if ((psi[i] ≡ 0 ∧ E[i][k] ≠ 0L) ∨ (psi[i] ≠ 0 ∧ E[i][k] ∗ sd > 0L)) goto wrong;    ▷ bad column k
    if (E[m][k] ∗ sd ≥ 0L) goto wrong;    ▷ bad E[m][k]
  }
  else {
    for (i = 1; i < m; ++i)
      if (psi[i] ≡ 0)
        for (j = 1; j < n; ++j)
          if (E[i][j] ≠ 0L) goto wrong;    ▷ error on nonbasic row
    for (j = 1; j < n; ++j)
      if (E[m][j] ∗ sd < 0L) goto wrong;    ▷ error on row m
  }
  return;
wrong:
  fprintf(stderr, "\n\n␣ERROR!\a");
  fprintf(ofile, "\n\n␣ERROR!␣Matrix␣E␣is␣NOT␣simple");
  return;
}
```

126. Second checking of the results. *(This part of the code should be deleted after the program has been properly debugged.)* Here we shall check the results of the original minimization problem. Given all the checking we did above, this second checking is an overkill; but we shall do it anyhow.

⟨ Check results 120 ⟩ +≡

```
if (h ≠ 0) check_vprime();
else {
  check_u();
  if (k ≠ 0) check_uprime();
  else check_v();
}
```

127. Suppose our program decided that the minimization problem is infeasible. Then it must have found a vector v' such that

$$v' \cdot A \le 0 \text{ and } v' \cdot b > 0 \quad \text{or} \quad v' \cdot A \ge 0 \text{ and } v' \cdot b < 0\ ,$$

where A, b, and c are parts of D as indicated in 2.1. We must check that v' does have this property. The inputs for function *check_vprime* are the global variables D, m, n, and *vprime*. Our error messages say "y'" where you would expect "v'/d".

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_vprime(void) {
  int i, j;
  llong s, term, summ;
  ⟨ Let summ = vprime * b; in case of overflow, go to oflo 128 ⟩
  if (summ ≡ 0LL) goto wrong;
  ⟨ If vprime * A has wrong sign go to wrong; in case of overflow, go to oflo 129 ⟩
  return;
oflo:
  fprintf(stderr, "\n\n␣Overflow␣WARNING!\a");
  fprintf(ofile, "\n\n␣WARNING!␣Unable␣to␣compute␣vprime*A␣or␣vprime*b␣without␣overflow");
  return;
wrong:
  fprintf(stderr, "\n\n␣ERROR!\a");
  fprintf(ofile, "\n\n␣ERROR!␣y'␣is␣not␣an␣infeasibility␣vector");
  return;
}
```

128. ⟨ Let *summ* = *vprime* ∗ *b*; in case of overflow, go to *oflo* 128 ⟩ ≡

```
summ = 0LL;
for (i = 1; i < m; ++i) {
  term = (llong) vprime[i] * (llong) D[i][n];
  summ += term;
  if (summ ≥ TWO62) goto oflo;
}
```

This code is used in section 127.

129. ⟨ If *vprime* ∗ *A* has wrong sign go to *wrong*; in case of overflow, go to *oflo* 129 ⟩ ≡

```
s = summm < 0LL ? -1LL : +1LL;
for (j = 1; j < n; ++j) {
  summm = 0LL;
  for (i = 1; i < m; ++i) {
    term = (llong) vprime[i] * (llong) D[i][j];
    summm += term;
    if (summm ≥ TWO62) goto oflo;
  }
  if (s * summm > 0LL) goto wrong;
}
```

This code is used in section 127.

130. Suppose our program decided that the minimization problem is not infeasible. Then it must have found a vector u such that

$$A \cdot u/d \equiv b \quad \text{and} \quad u/d \geq 0\ ,$$

where A, b, and c are parts of D as indicated in 2.1. We must check that u does, in fact, possess these properties. The inputs for function *check_u* are the global variables D, m, n, d, and u. Our error messages say "x" where you would expect "u/d".

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_u(void) {
  int i, j;
  long sd;
  llong term, sum;
  sd = d < 0L ? -1L : +1L;
  ⟨ If u/d is not ≥ 0 then go to wrong 131 ⟩
  ⟨ If A * u/d is not = b then go to wrong; in case of overflow, go to oflo 132 ⟩
  return;
oflo:
  fprintf(stderr, "\n\n Overflow WARNING!\a");
  fprintf(ofile, "\n\n WARNING! Unable to compute A*x without overflow");
  return;
wrong:
  fprintf(stderr, "\n\n ERROR!\a\n");
  fprintf(ofile, "\n\n ERROR! x != 0 or A*x != b\n");
  return;
}
```

131. ⟨ If *u*/*d* is not ≥ 0 then go to *wrong* 131 ⟩ ≡

```
for (j = 1; j < n; ++j)
  if (sd * u[j] < 0L) goto wrong;
```

This code is used in section 130.

132. ⟨ If $A * u/d$ is not $= b$ then go to *wrong*; in case of overflow, go to *oflo* 132 ⟩ ≡

```
for (i = 1; i < m; ++i) {
  sum = 0LL;
  for (j = 1; j < n; ++j) {
    term = (llong) D[i][j] * (llong) u[j];
    sum += term;
    if (sum ≥ TWO62) goto oflo;
  }
  if (sum ≠ (llong) d * (llong) D[i][n]) goto wrong;
}
```

This code is used in section 130.

133. Suppose our program decided that the minimization problem is unbounded. Then it must have found a vector u' such that

$$A \cdot u'/d \equiv 0, \quad u'/d \geq 0 \quad \text{and} \quad c \cdot u'/d < 0 \ ,$$

where A, b, and c are parts of D as indicated in 2.1. We must check that u' does, in fact, possess these properties. The inputs for function *check_uprime* are the global variables D, m, n, d, and *uprime*. Our error messages say "x'" where you would expect "u'/d".

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_uprime(void) {
  int i, j;
  long sd;
  llong term, sum;
  sd = d < 0L ? −1L : +1L;
  ⟨ If uprime/d is not ≥ 0 then go to wrong 134 ⟩
  ⟨ If A * uprime/d is not = 0 then go to wrong; in case of overflow, go to oflo 135 ⟩
  ⟨ If c * uprime/d is not < 0 then go to wrong; in case of overflow, go to oflo 136 ⟩
  return;
oflo:
  fprintf(stderr, "\n\n␣Overflow␣WARNING!\a");
  fprintf(ofile, "\n\n␣WARNING!␣Unable␣to␣compute␣A*x␣or␣A*x'␣or␣c*x'␣without␣overflow");
  return;
wrong:
  fprintf(stderr, "\n\n␣ERROR!\a\n");
  fprintf(ofile, "\n\n␣ERROR!␣x'␣!>=␣0␣or␣A*x'␣!=␣0␣or␣c*x'␣!<␣0\n");
  return;
}
```

134. ⟨ If *uprime*/d is not ≥ 0 then go to *wrong* 134 ⟩ ≡

```
for (j = 1; j < n; ++j)
  if (sd * uprime[j] < 0L) goto wrong;
```

This code is used in section 133.

135. ⟨ If $A * uprime/d$ is not $= 0$ then go to *wrong*; in case of overflow, go to *oflo* 135 ⟩ ≡

```
for (i = 1; i < m; ++i) {
   sum = 0LL;
   for (j = 1; j < n; ++j) {
      term = (llong) D[i][j] * (llong) uprime[j];
      sum += term;
      if (sum ≥ TWO62) goto oflo;
   }
   if (sum ≠ 0LL) goto wrong;
}
```

This code is used in section 133.

136. ⟨ If $c * uprime/d$ is not < 0 then go to *wrong*; in case of overflow, go to *oflo* 136 ⟩ ≡

```
sum = 0LL;
for (j = 1; j < n; ++j) {
   term = (llong) D[m][j] * (llong) uprime[j];
   sum += term;
   if (sum ≥ TWO62) goto oflo;
}
if ((llong) sd * sum ≥ 0LL) goto wrong;
```

This code is used in section 133.

137. Suppose our program decided that the minimization problem is solvable. Then it must have found vectors u and v such that

$$(v/d) \cdot A \le c \quad \text{and} \quad c \cdot u \equiv v \cdot b\ ,$$

where A, b, and c are parts of D as indicated in 2.1. We must check that u and v do, in fact, possess these properties. The inputs for function *check_v* are the global variables D, m, n, d, u, and v. Our error messages say "x" and "y" where you would expect "u/d" and "v/d".

⟨ Auxiliary functions 35 ⟩ +≡

```
void check_v(void) {
   int i, j;
   long sd;
   llong term, sum, summ;
   sd = d < 0L ? −1L : +1L;
   ⟨ If (v/d) ∗ A is not ≤ c then go to wrong; in case of overflow, go to oflo 138 ⟩
   ⟨ If c ∗ u/d is not = (v/d) ∗ b then go to wrong; in case of overflow, go to oflo 139 ⟩
   return;
oflo:
   fprintf(stderr, "\n\n␣Overflow␣WARNING!\a");
   fprintf(ofile,
        "\n\n␣WARNING!␣Unable␣to␣compute␣A*x␣or␣y*A␣or␣y*b␣or␣c*x␣without␣overflow");
   return;
wrong:
   fprintf(stderr, "\n\n␣ERROR!\a\n");
   fprintf(ofile, "\n\n␣ERROR!␣y*A␣!<=␣c␣or␣c*x␣!=␣y*b\n");
   return;
}
```

138. ⟨ If $(v/d) * A$ is not $\le c$ then go to *wrong*; in case of overflow, go to *oflo* 138 ⟩ ≡

```
for (j = 1; j < n; ++j) {
  summ = 0LL;
  for (i = 1; i < m; ++i) {
    term = (llong) v[i] * (llong) D[i][j];
    summ += term;
    if (summ ≥ TWO62) goto oflo;
  }
  if ((llong) sd * summ > (llong) sd * (llong) d * (llong) D[m][j]) goto wrong;
}
```

This code is used in section 137.

139. ⟨ If $c * u/d$ is not $= (v/d) * b$ then go to *wrong*; in case of overflow, go to *oflo* 139 ⟩ ≡

```
sum = 0LL;
for (j = 1; j < n; ++j) {
  term = (llong) D[m][j] * (llong) u[j];
  sum += term;
  if (sum ≥ TWO62) goto oflo;
}
summ = 0LL;
for (i = 1; i < m; ++i) {
  term = (llong) v[i] * (llong) D[i][n];
  summ += term;
  if (summ ≥ TWO62) goto oflo;
}
if (sum ≠ summ) goto wrong;
```

This code is used in section 137.

140. Index. The index shows the section whre each of the identifiers in this module is defined and used.

SIMPLEX

Written by Paulo Feofiloff, at IME–USP, in February 1998. Based on a previous version written in Pascal in March 1996.

Keywords: Simplex algorithm, Simplex method, linear programming, Cramer's rule, integer arithmetic.