GAUSS ELIMINATION AS A TERM REWRITING SYSTEM

P.H. Rodenburg

Department of Philosophy
University of Utrecht

D.J. Hoekzema

Department of Philosophy
Department of Physics
University of Utrecht

Logic Group
Preprint Series
No. 30

Department of Philosophy

University of Utrecht

GAUSS ELIMINATION AS A TERM REWRITING SYSTEM

P.H. Rodenburg

Department of Philosophy
University of Utrecht

D.J. Hoekzema
Department of Philosophy
Department of Physics
University of Utrecht

January 1988

Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

GAUSS ELIMINATION AS A TERM REWRITING SYSTEM

D.J. Hoekzema

Philosophy Department and Physics Department, Rijksuniversiteit te Utrecht

P.H. Rodenburg

Philosophy Department, Rijksuniversiteit te Utrecht

We specify the Gauß elimination algorithm algebraically, in a format described in our earlier report [RH]. The specification is to be considered as a term rewriting system; as such, it is complete, *modulo* substitution of a complete term rewriting system in its parametrized module NUMBERS.

Note: The authors were supported by the Dutch government through SPIN under the PRISMA project (Parallel Inference and Storage Machine).

1. Introduction

In many respects, the present paper is a continuation of an earlier one, [RH]. It contains a specification of an algorithm, with some brief comments. For its background, we refer to [RH]. Indeed, some of the modules given here, in particular BOOL, NAT and LISTS, are copied from [RH]. The specification is parametrized; it requires a particular sort X with certain operations on it (see the module NUMBERS below) about which nothing further is said. If an appropriate number system is substituted into the algorithm, binding the parameter P in NUMBERS, it does Gaussian elimination. We have taken care that the rational number system as specified in [RH] (or the derived system of complex rationals) *is* appropriate.

2. Gaussian elimination

The Gaussian elimination algorithm is a well-known method for solving systems of linear equations with coefficients in some field S. A system of n linear equations with m unknowns is represented by an $n\times(m+1)$ matrix; for instance,

$$a_{11}x + a_{12}y = a_{13}$$
$$a_{21}x + a_{22}y = a_{23}$$

is represented by

$$\begin{pmatrix} a_{11} a_{12} a_{13} \\ a_{21} a_{22} a_{23} \end{pmatrix}$$

The algorithm is based on the fact that the solution space of a system of linear equations does not change if a multiple of one of its elements is added to other elements; in the above example,

$$\begin{pmatrix} a_{11}+ba_{21} & a_{12}+ba_{22} & a_{13}+ba_{23} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

has the same solution space as the original matrix. Solving the system of equations now means that the corresponding matrix is rewritten into the most readable representation of the solution space. Suppose we are dealing with a system represented by a matrix $A = (a_{ij})_{i<n,j\leq m}$. Let $A_0 = (a_{ij})_{i<n,j<m}$ be the matrix of coefficients. Recall that the rank of a matrix is the dimension of the space spanned by its column vectors. There are three cases:

(i) $\text{rank}(A_0) \neq \text{rank}(A)$. This may be the case if $n > m$: since then for any system of m vectors in an n-dimensional vector space, there exist independent vectors. The solution space is empty. This shows by the appearance of a row of the form $(0 \ldots 0\ a)$, i.e. $\sum_i 0 \cdot x_i = a$, with $a \neq 0$.

(ii) $\text{rank}(A_0) = \text{rank}(A) = m$, the solution space consists of a single point. A_0 can be diagonalized, and the $(n+1)$-th column gives the values of the variables at the solution.

(iii) $\text{rank}(A_0) = \text{rank}(A) < m$, the solution space is an $(m-\text{rank}(A))$-dimensional subspace of S^n. The result of rewriting is a simplified version (determining the same solution space) of the original system of equations. For example, the result of rewriting the matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}$$

is of the form

$$\begin{pmatrix} 1 & 0 & b_{13} & b_{14} \\ 0 & 1 & b_{23} & b_{24} \end{pmatrix},$$

representing the equations

$$x + b_{13}z = b_{14}$$
$$y + b_{23}z = b_{24}$$

The details of the rewriting process are best explained under the modules that specify them.

3. The specification

```
module  BOOL                                --booleans
begin
  exports
    begin
      sorts B
      functions T: → B
                F: → B
                _∧_: B × B → B
                _∨_: B × B → B
    end
  variables
    p: → B
  equations
  [1] T ∧ p = p
  [2] p ∧ T = p
  [3] F ∧ F = F
  [4] T ∨ T = T
  [5] p ∨ F = p
  [6] F ∨ p = p
end BOOL
```

```
module  NAT                          --natural numbers
begin
  exports
    begin
      sorts N
      functions 0: → N
                S_: N → N
                _+_: N × N → N
                _·_: N × N → N
    end
  variables
    m, n: → N
  equations
  [1] m + 0 = m
  [2] m + Sn = S(m+n)
  [3] m · 0 = 0
  [4] m · Sn = m·n + m
end NAT
```

The modules BOOL and NAT are both complete, cf. [RH].

```
module NUMBERS -- numbers
begin
  parameters
    P begin
      sorts X
      functions 0: → X
                1: → X
                †X: → X
                -_: X → X
                _+_: X × X → X
                1/_:X → X
                _·_:X × X → X
                _≈_: X × X → B
    end P
  imports BOOL
end NUMBERS
```

By itself, NUMBERS specifies only a signature; the module's behaviour depends entirely on the binding of the parameter P. Numbers is in fact merely a list of features which an admissible number system should have, in order to make the Gaussian algorithm work at all, i.e. there must be elements $\underline{0}$ and $\underline{1}$, there must be addition and multiplication, etc. Beyond this, the number system that one inserts must meet some additional demands if the algorithm is to work properly. We shall assume that its canonical model is a field. The module CORT (complex rationals) in [RH] satisfies these requirements (with C for X). Its submodule RADD (rational addition) would do as well. The choice of operations to figure as $\underline{0}$, $\underline{1}$, etc. is rather obvious in either case — in particular, in RADD $\underline{1}$ should be `+[Z]`; for $\dagger X$, the representative of $1/\underline{0}$ is a good candidate. CORT and RADD fail in one minor respect: they do not have an operation $\approx$. For both of them, such a feature is specified in the appendix below. For there to be a canonical model at all, we must of course assume that the specification of the number system is semicomplete. Moreover we claim that, if it is *complete*, every module will be complete too. The straightforward verification of this claim (using the normal extension theorem formulated in [RH]§4) we leave to the reader.

```
module LISTS -- lists
begin
  parameters  M
    begin
      sorts X
      functions †X: → X
    end M
  exports
    begin
      sorts LX
      functions _;_: X × LX → LX
                _;_: LX × LX →  LX
                (_)_:LX × N → X
                †LX: → LX
                R_: LX → LX
    end
  imports NAT
  variables
    n: → N
    x: → X
    v,w: → LX
  equations
  [1] (†LX)n = †X
  [2] (x;v)0 = x
  [3] (x;v)Sn =(v)n
  [4] †LX;v = v
  [5] (x;v);w = x;(v;w)
  [6] R †LX = †LX
  [7] R(x;v) = (Rv);(x;†LX)
end LISTS
```

Lists is essentially a conjunction of the modules SIM (simple lists) and REV (reversion) in [RH]. Furthermore, we added the possibility to select the *n*-th entry of a list, because this is a feature which we shall need later on in the modules SUB and TRIAN.

```
module LS -- list structure
begin
  exports
    begin
      functions _+_: LX × LX → LX
                _·_: X × LX → LX
    end
  imports NUMBERS, LISTS {M bound to numbers}
  variables x,y :→ X
            v,w :→ LX
  equations
  [1] x;v + y;w = (x+y);(v+w)
  [2] v + †LX = †LX
  [3] †LX + v = †LX
  [4] x·(y;v) = (x·y) ; (x·v)
  [5] x·†LX = †LX
end LS
```

LS defines some additional structure for lists of numbers. Such lists can now be added to each other, and they can be multiplied with a scalar. · works as in SCAM (scalar multiplication) in [RH]; + may be compared to $\wedge$ in DIM (diagonal multiplication, ibid.).

```
module IFTE -- if then else
begin
  parameters
    P begin
      sorts X
    end P
  exports
    begin
      functions If: B × X × X → X
    end
  imports BOOL
  variables x,y: → X
  equations
  [1] If(T,x,y) = x
  [2] If(F,x,y) = y
end IFTE
```

```
module MAT -- matrices
begin
  imports LS, LISTS {renamed by [LX → MX, †LX →   †MX, ; → :],
                          P bound by [X → LX , †X → †LX] to LS}
end MAT
```

In this module, matrices are introduced as lists of lists — henceforth called rows — of numbers. It may be noted that the renaming ; → : is supposed to apply to *both* functions by that name in LISTS.

The rows in our matrices need not be equally long. Equation [4] of LISTS will have the effect that shorter rows behave as if they had been supplemented with occurrences of †*X* on the right. As a result, a single matrix in the ordinary sense will have many representing normal forms of MAT. We leave this as it is from lack of interest in equality of matrices.

```
module PROT -- put row on top
begin
  exports
    begin
      functions Pt: N × MX → MX
    end
  imports MAT
  functions H: N × MX × MX → MX
  variables a,b: → MX
            n: → N
            v: → LX
  equations
  [1] H(0,a,v:b) = v:(Ra:b)
  [2] H(Sn,a,v:b) = H(n,v:a,b)
  [3] H(n,a,†MX) = Ra
  [4] Pt(n,a) = H(n,†MX,a)
end PROT
```

The effect of applying *Pt* to *n* and *a* is that the *n*-th row of the matrix *a* is removed and put on top. If *n* exceeds the highest row number of *a*, then *Pt*(*n*,*a*) returns *a* itself, after wasting some time in reversing it twice.

```
module SUB -- subtract row
begin
  exports
    begin
      functions Sub: N × LX × MX → MX
                Norm: N × LX → LX
    end
  imports MAT, IFTE {P bound by [X → LX] to MAT}
  functions H: N × LX × MX → MX
  variables n: → N
            v,w: → LX
            a: → MX
            x: → X
  equations
  [1] Norm(n,v) = If((v)n ≈ 0 ∨ (v)n ≈ †X, v, 1/(v)n · v)
  [2] H(n,v,w:a) = (w + -(w)n ·Norm(n,v)): H(n,v;a)
  [3] H(n,v,†MX) = †MX
  [4] Sub(n,v,a) = If((v)n ≈ 0 ∨ (v)n ≈ †X, a, H(n,v,a))
end SUB
```

The module SUB defines two functions, *Norm* and *Sub*. *Norm*(n,v) is the result of normalizing the row v on its n-th component, i.e. the result of dividing the whole row by $(v)_n$, if that is possible, and if it is not, that is, if $(v)_n$ is $\underline{0}$ or undefined, simply v. The calculation proposed for *Sub*(n,v,a) consists in normalizing v on its n-th component, and then subtracting multiples of the resulting row from all rows of a, in such a way that the n-th column of a comes to consist entirely of zeroes. Since clearly such an outcome cannot be guaranteed if $(v)_n$ is $\underline{0}$ or undefined, equation [4] makes an exception for this case.

```
module TRIAN --triangulate matrix
begin
  exports
    begin
      functions Tr: MX → MX
    end
  imports PROT
          SUB
  functions H1: N × MX → MX
            H2: N × MX → LX
            Row: N × MX → N
  variables n: → N
            v: → LX
            a: → MX
  equations
  [1] Row(n,v:a) = If((v)n ≈ 0 ∨ (v)n ≈ †X, SRow(n,a), 0)
  [2] Row(n,†MX) = 0
  [3] H1(n,a) = Pt(Row(n,a),a)
  [4] H2(n,v:a) = Norm(n,v) : H2(Sn, H1(Sn, Sub(n,v,a)))
  [5] H2(n,†MX) = †MX
  [6] Tr(a) = H2(0,H1(0,a))
end TRIAN
```

The various functions in TRIAN work as follows. *Row*(n,a) looks for the first row in a with a welldefined nonzero entry in its n-th place. It returns the position of this row. If every row has 0 or † in n-th position, then *Row* returns max+1, where max is the index of the last row. This will happen in particular if n exceeds the row length of a. $H_1(n,a)$ puts the row pointed out by *Row*(n,a) on top, leaving the matrix unchanged if *Row*(n,a) = max+1. $H_2(n,a)$ normalizes the top row, $(a)_0$, on its n-th component and produces zeros in the rest of the n-th column (in the pathological case that $(a)_{0,n}$ is 0 or †X, $H_2(n,a) = a$). The process now continues in the next column of a (so the top row v, having been normalized, drops out of consideration). Because in the right hand sides of [3] and [6] the arguments of `H2` have the form `H1(s,t)`, we can be sure that if there is nothing wrong with the matrix, such as undefined entries or rows of unequal length, either the top row of the matrix can indeed be normalized on its n-th component, or the entire n-th column already consisted of zeros to begin with. Accordingly, we can be sufficiently sure that the function *Tr* produces a triangulated matrix, i.e. a matrix $b = (b_{ij})$ with only zeros beneath the diagonal (b_{ii}).

```
module GAUSS -- gaussian elimination
begin
  exports
    begin
      functions G: MX → MX
    end
  imports TRIAN
  functions H: MX → MX
            C: LX → N
  variables n,m: → N
            a: → MX
            v: → LX
            x: → X
  equations
  [1] C(x;v) = If(x≈0 ∨ x≈†X, SC(v), 0)
  [2] C(†LX) = 0
  [3] H(v:a) = v : H(Sub(C(v), v, a))
  [4] H(†MX) = †MX
  [5] G(a) = RH(RTr(a))
end GAUSS
```

For a row v, $C(v)$ is the position of its first proper nonzero entry. H is comparable to H_2 in TRIAN: it clears a column below the top row. The function G defined by [5] is the operation of Gauß elimination that we set out to specify. It subjoins to triangulation a cleaning of the matrix, from bottom to top. The ideal result of triangulation — supposing to begin with that we are working with an $n\times(n+1)$ matrix — has the form (taking n=3 for definiteness)

$$\begin{pmatrix} 1 & a_{12} & a_{13} & a_{14} \\ 0 & 1 & a_{23} & a_{24} \\ 0 & 0 & 1 & a_{34} \end{pmatrix} .$$

This is first reversed; and then H is applied, and multiples of the last row (which is now on top) are subtracted from the other rows in such a way that a_{23} and a_{13} are replaced by zeros. Next a multiple of the second row is subtracted from the old top row, leaving a zero in second position. The resulting matrix is reversed, and we end up with something of the form

$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \end{pmatrix} ,$$

representing the solution

$x = a$
$y = b$
$z = c$.

In less ideal circumstances, we get equations representing a larger solution space, or a contradiction, as announced in §2. If there are undefined entries in the input (in particular if the rows are of unequal length), we will get an output, though it will not make sense.

Appendix: equality on complex rationals

The complex rationals in [RH] were built from the positive integers in a number of steps. We follow these steps in defining equality. We make use of the module BOOL, which introduces the booleans T and F (of sort B), and conjunction $\wedge$.

The positive integers (module PINT, sort P) are specified in unary notation, as `1`, `S1`, `SS1`, `SSS1`, ... ; we use this representation to define equality with recursion.

```
module EQPINT -- equality on positive integers
begin
  exports
    begin
      functions _≈_: P × P → B
    end
  imports PINT, BOOL
  variables p,q: → P
  equations
  [1] 1≈1 = T
  [2] 1 ≈ Sp = F
  [3] Sp ≈ 1 = F
  [4] Sp ≈ Sq = p≈q
end EQPINT
```

The integers (sort Z in module INT) are represented as `0` or $+t$ or $-t$, with t a normal form of sort P of PINT (in other words, t is a term S^n1 as above). Since the rationals are constructed from the positive rationals in exactly the same way, we first give a parametrized definition of equality, and then fill in the parameters for the case of the integers.

```
module EQSIGN -- equality on signed objects
begin
  parameters  M
    begin
      sorts X, Y
      functions _≈_: X × X → B
                0: → Y
                +_: X → Y
                -_: X → Y
    end M
  exports
    begin
      functions _≈_: Y × Y → B
    end
  imports BOOL
  variables w,x: → X
  equations
  [1] 0≈0 = T
  [2] 0 ≈ +x = F
  [3] +x ≈ 0 = F
  [4] 0 ≈ -x = F
  [5] -x ≈ 0 = F
  [6] +w ≈ +x = w≈x
  [7] +w ≈ -x = F
  [8] -w ≈ +x = F
  [9] -w ≈ -x = w≈x
end EQSIGN

module EQPINT + INT
begin
  imports INT
        EQPINT
end EQPINT + INT
```

```
module EQI -- equality on integers
begin
  imports EQPINT + INT
          EQSIGN
            {M bound by [X → P, Y → Z, 0 → 0]
             to EQPINT + INT}
end EQI
```

The positive rationals are represented as lists of integers: as it happens, every positive rational q has a unique representation

$$q = p_0^{n_0} \cdot \ldots \cdot p_{k-1}^{n_{k-1}}$$

with $p_0, p_1, \ldots$ the primes 2, 3,... in ascending order, $n_0, \ldots, n_{k-1} \in \mathbb{Z}$, and $n_{k-1} \neq 0$. In general, equality of lists reduces to equality of their elements. SIM (simple lists) is a module in [RH] that takes a parameter `M` containing a sort `X`; objects of that sort are gathered into lists $x_1;(x_2;(x_3(\ldots;x_n\ldots)))$; `[X]` stands for the empty list.

```
module EQL -- equality on lists
begin
  parameters  M'
    begin
      functions _≈_: X × X → B
    end M'
  exports
    begin
      functions _≈_: LX × LX → B
    end
  imports SIM, BOOL
  variables x,y: → X
            a,b: → LX
  equations
  [1] [X] ≈ [X] = T
  [2] [X] ≈ x;a = F
  [3] x;a ≈ [X] = F
  [4] x;a ≈ y;b = x≈y ∧ a≈b
end EQL
```

```
module EQREX -- equality on rational numbers as lists of
exponents
begin
  imports EQI
        EQL
          {renamed by [LX → LZ, [X] → [Z]]
          M' bound by [X → Z]to EQI}
end EQREX
```

The rationals are constructed from lists of integers just as the integers from the positive integers; and the construction of equality is analogous. The rationals are sort *Q* in the module RAT in [RH].

```
module EQREX + RAT
begin
  imports RAT, EQREX
end EQREX + RAT


module REQ -- rational equality
begin
  imports EQREX + RAT, EQSIGN {M bound by [X → LZ, Y → Q, 0 →
0]
                                        to EQREX + RAT}
end REQ
```

Finally, the complex rationals, sort *C* in the module CORT of [RH], are pairs of rationals, and equality for them is defined as usual for pairs.

```
module CREQ -- complex rational equality
begin
  exports
    begin
      functions _≈_: C × C → B
    end
  imports CORT, REQ
  variables x, y, u, v: → Q
  equations
  [1] (x,y) ≈ (u,v) = x≈u ∧ y≈v
end CREQ
```

Acknowledgement. We profited from De Vries' MIRANDA program for Gaussian elimination [V]. F. Wiedijk put a derivate of the specification through his syntax checker, and pointed out a number of improvements.

References

[RH] P.H. Rodenburg & D.J. Hoekzema, Specification of the Fast Fourier Transform algorithm as a term rewriting system, to appear as No. 27 in the Logic Group Preprint Series, Dept. of Philosophy, Rijksuniversiteit Utrecht.

[V] F.-J. de Vries, A functional program for Gaussian elimination, Logic Group Preprint Series No. 23, Dept. of Philosophy, Rijksuniversiteit Utrecht.

Logic Group Preprint Series
Department of Philosophy
University of Utrecht
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

nr. 1 C.P.J. Koymans, J.L.M. Vrancken, *Extending Process Algebra with the empty process*, September 1985.
nr. 2 J.A. Bergstra, *A process creation mechanism in Process Algebra*, September 1985.
nr. 3 J.A. Bergstra, *Put and get, primitives for synchronous unreliable message passing*, October 1985.
nr. 4 A.Visser, *Evaluation, provably deductive equivalence in Heyting's arithmetic of substitution instances of propositional formulas*, November 1985.
nr. 5 G.R. Renardel de Lavalette, *Interpolation in a fragment of intuitionistic propositional logic*, January 1986.
nr. 6 C.P.J. Koymans, J.C. Mulder, *A modular approach to protocol verification using Process Algebra* , April 1986.
nr. 7 D. van Dalen, F.J. de Vries, *Intuitionistic free abelian groups*, April 1986.
nr.8 F. Voorbraak, *A simplification of the completeness proofs for Guaspari and Solovay's R*, May 1986.
nr.9 H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, *A semantic framework for the COLD-family of languages*, May 1986.
nr.10 G.R. Renardel de Lavalette, *Strictheidsanalyse*, mei 1986.
nr.11 A. Visser, *Kunnen wij elke machine verslaan? Beschouwingen rondom Lucas' argument*, Juli 1986.
nr.12 E.C.W. Krabbe, *Naess's dichotomy of tenability and relevance*, June 1986.
nr.13 Hans van Ditmarsch, *Abstractie in wiskunde, expertsystemen en argumentatie*, Augustus 1986
nr.14 A.Visser, *Peano's Smart Children, a provability logical study of systems with built-in consistency* , October 1986.
nr.15 G.R.Renardel de Lavalette, *Interpolation in natural fragments of intuitionistic propositional logic*, October 1986.
nr.16 J.A. Bergstra, *Module Algebra for relational specifications*, November 1986.
nr.17 F.P.J.M. Voorbraak, *Tensed Intuitionistic Logic*, January 1987.
nr.18 J.A. Bergstra, J. Tiuryn, *Process Algebra semantics for queues*, January 1987.
nr.19 F.J. de Vries, *A functional program for the fast Fourier transform*, March 1987.
nr.20 A. Visser, *A course in bimodal provability logic*, May 1987.
nr.21 F.P.J.M. Voorbraak, *The logic of actual obligation, an alternative approach to deontic logic*, May 1987.
nr.22 E.C.W. Krabbe, *Creative reasoning in formal discussion*, June 1987.
nr.23 F.J. de Vries, *A functional program for Gaussian elimination*, September 1987.
nr.24 G.R. Renardel de Lavalette, *Interpolation in fragments of intuitionistic propositional logic*, October 1987.(revised version of no. 15)
nr.25 F.J. de Vries, *Applications of constructive logic to sheaf constructions in toposes*, October 1987.
nr.26 F.P.J.M. Voorbraak, *Redeneren met onzekerheid in expertsystemen*, November, 1987.
nr.27 P.H. Rodenburg, D.J. Hoekzema, *Specification of the fast Fourier transform algorithm as a term rewriting system*, December, 1987.

nr.28 D. van Dalen, *The war of the frogs and the mice, or the crisis of the Mathematische Annalen,* December, 1987.

nr.29 A. Visser, *Preliminary Notes on Interpretability Logic,* January, 1988.

nr.30 D.J. Hoekzema, P.H. Rodenburg, *Gauß elimination as a term rewriting system*, January, 1988.