Max-plus Linear Algebra with Scilab

Stéphane Gaubert and Max P. Scilab

ALAPEDES Max-Plus SOFTWARE WORKSHOP
INRIA June 18-19, 1998

Abstract

This document is a tutorial session in Scilab, which presents the max-plus linear algebra facilities currently under development. The implementation is more than tentative: remarks and suggestions are welcome.

All this session is contained in the Scilab exec file `TPALGLIN.sce`, that you can be execute via the command `exec TPALGLIN.sce` or, if you wish a step by step demonstration, via `exec('TPALGLIN.sce',7)`

The notions and mathematical notations used here can be found in standard books on max-plus algebra (e.g. [1]), or are detailed in [6], [5].

Contents

I. Solving Linear Equations of the Form $x = Ax \oplus b$

Let us first recall the following celebrated result:

Theorem 1: Let A denote a $n \times n$ matrix, and b a n-dimensional column vector, all with entries in the semiring $\overline{\mathbb{R}}_{\max} = (\mathbb{R} \cup \{-\infty, +\infty\}, \max, +)$. The minimal n-dimensional column vector x with entries in $\overline{\mathbb{R}}_{\max}$, such that

$$x = Ax \oplus b$$

is given by

$$x = A^*b$$

where, by definition,

$$A^* = A^0 \oplus A \oplus A^2 \oplus A^3 \oplus \cdots \ .$$

Moreover, if all the entries of A are strictly less than $+\infty$, then all the entries of A^* are strictly less than $+\infty$ iff $A^* = A^0 \oplus \cdots \oplus A^k$, for all $k \geq n-1$. The syntax in `Scilab` is simply `star(A)`, where A is a full max-plus matrix. Let us try some basic values:

```
a=#(2)
 a  =
    2.
b=star(a)
 b  =
   Inf
a=#(-1)
 a  =
  - 1.
b=star(a)
 b  =
    0.
a=%0
 a  =
  -Inf
b=star(a)
 b  =
     0.
a=%1
 a  =
     0.
b=star(a)
 b  =
     0.
```

The same syntax is valid for matrices (our implementation uses Jordan algorithm [7, Ch. 3,§ 4.3], which requires $O(n^3)$ time).

```
a=%zeros(2,2)
 a  =

! -Inf  -Inf !
! -Inf  -Inf !

b=star(a)
 b  =

!   0.  -Inf !
! -Inf    0. !

type(b)
```

Stéphane.Gaubert@inria.fr, http://amadeus.inria.fr/gaubert
Scilab@inria.fr, http://www-rocq.inria.fr/scilab

```
 ans  =

    257.
```

(the type of usual full matrices is 1, the type of max-plus full matrices is 257). Here is a more complicated example:

```
 a=#([-1 2; %0 -3])
 a  =

! - 1.    2. !
! -Inf  - 3. !

star(a)
 ans  =

!   0.    2. !
! -Inf    0. !
```

Yet a more complicated example:

```
a=#([%0 2 3 ; -2 -10 -1 ; -5 -2 %1])
 a  =

! -Inf    2.     3. !
! - 2.  - 10.  - 1. !
! - 5.  - 2.     0. !

b=star(a)
 b  =

!   0.    2.    3. !
! - 2.    0.    1. !
! - 4.  - 2.    0. !
```

We check that the star operation is idempotent:

```
star(star(a))==star(a)
 ans  =

! T T T !
! T T T !
! T T T !
```

We perform a second consistency check:

```
star(a)==(a^0+a)^2
 ans  =

! T T T !
! T T T !
! T T T !
```

Since star(a) is finite, the answer to the following test must be true

```
(a^0+a)^2==(a^0+a)^3
 ans  =

! T T T !
! T T T !
! T T T !
```

Here, `star(a)` is finite because all the circuits of a have negative or zero weight. To find nodes in circuits with exactly zero weight, we have to compute the zero diagonal entries of the matrix

$$A^+ = A \oplus A^2 \oplus A^3 \oplus \cdots = AA^* \ .$$

```
b=plus(a)
 b  =

!   0.    2.    3. !
! - 2.    0.    1. !
! - 4.  - 2.    0. !
```

Is it correct ?

```
b==a*star(a)
 ans  =

! T T T !
! T T T !
! T T T !
```

Since $b(1,1) = b(2,2) = b(3,3) = 0$, each entry of a belongs to a circuit of weight 0. Let us modify this:

```
a(2,1)=-10
 a  =

! -Inf     2.     3. !
! - 10.  - 10.  - 1. !
! - 5.   - 2.     0. !

plus(a)
 ans  =

! - 2.    2.    3. !
! - 6.  - 3.  - 1. !
! - 5.  - 2.    0. !
```

What happens if a circuit has strictly positive weight ?

```
a(3,1)=6
 a  =

! -Inf     2.     3. !
! - 10.  - 10.  - 1. !
!   6.   - 2.     0. !

plus(a)
 ans  =

!  Inf   Inf   Inf !
!  Inf   Inf   Inf !
!  Inf   Inf   Inf !
```

Mixing $+\infty$ and $-\infty$:

```
a=#([2 3; %0 -1])
 a  =

!   2.    3. !
! -Inf  - 1. !

star(a)
 ans  =

!  Inf   Inf !
! -Inf    0. !
```

Random large example:

```
a=#(rand(64,64))
 a  =

         column 1 to 5

!   0.2113249    0.3760119    0.6212882...
!   0.7560439    0.7340941    0.3454984...
[suppressed output]

b=star(a)
 b  =

         column  1 to 11

!  Inf   Inf   Inf   Inf   Inf   Inf...
!  Inf   Inf   Inf   Inf   Inf   Inf...
[supressed output]
```

To make a^* convergent, we have to make sure that all circuits have at most zero weigth, e.g. by using the following normalization:

```
a=(%ones(1,size(a,1))*a*..
%ones(size(a,2),1))ˆ(-1)*a;
```

We check that the new matrix has maximum 0:

```
 max(plustimes(a))==0
 ans  =

  T
```

Since it seems correct, let us put it in a macro:

```
deff('[b]=normalize(a)',..
'b=(%ones(1,size(a,1))*a..
*%ones(size(a,1),1))ˆ(-1)*a')
```

We check that the macro is correct (empty answer=ok)

```
find(a<>normalize(a))
 ans  =

     []
```

Now, star(a) should be finite. Indeed,

```
b=star(a)
 b  =

         column 1 to 5

!   0.          - 0.0410985  - 0.0491906...
! - 0.1255879    0.          - 0.0943671...
[suppressed output]
```

Let us check the answer (recall that $A^* = (\mathrm{Id} \oplus A)^{n-1}$, provided that it converges).

```
find(b<>((aˆ0+a)ˆ8)ˆ8)
 ans  =

     []
```

The naive $(\cdot)^{64}$ operation would have been a bit slow for such a "large" matrix. Indeed, a^* is computed in $O(n^3)$ time, and $(a^0 + a)^n$ requires an $O(n^4)$ time, unless we use dichotomic powers.

Most probably, the star converges in less than 64 steps

```
c=(aˆ0+a)ˆ8;
```

The following shows how many entries of $b = a^*$ are distinct from $c = (\mathrm{Id} \oplus a)^8$, and c^2, respectively:

```
size(find(b<>c),2)
 ans  =
     2.

size(find(b<>cˆ2,2))
 ans  =
     0
```

Hence, $a^* = (\mathrm{Id} \oplus a)^{16}$. This raises the interesting question of understanding how fast the star of a random matrix converges. Finally, we find the minimal solution of the equation $x = ax \oplus b$ using Theorem 1 above.

```
a=#(-1)
 a  =

  - 1.
b=#(2)
 b  =

    2.

x=star(a)*b
 x  =

    2.
```

Idem for matrices

```
a=#([%0 %1 %0; %0 %0 -1; %1 %0 %0])
 a  =

! -Inf    0.  -Inf !
! -Inf  -Inf  - 1. !
!   0.  -Inf  -Inf !
```

```
b=#([10; %0; %0])
 b  =

!   10. !
! -Inf  !
! -Inf  !

x=star(a)*b
 x  =

!   10. !
!   9.  !
!   10. !

x==a*x+b
 ans  =

! T !
! T !
! T !
```

The star of sparse matrices is not implemented yet (by the way, computing the star of sparse matrices is not allways a sensible thing to do, since the result is generically full).

Among desirable further developments, let us mention the development of sparse algorithms to compute $x = a^*b$, when a is a sparse matrix and b a full or sparse column vector. We plan to implement two algorithms:

1. *value iteration*, which computes the sequence $x(k) = ax(k-1) \oplus b, x(0) = \mathbb{0}$. If a^*b is finite, the sequence converges in a finite (possibly small) time to the minimal solution. Of course, Gauss-Seidel refinements can be implemented (all this is fairly easy to do).
2. *policy iteration*. This is a joint work with Jean Cochet-Terrasson: there is a fixed point analogue of the max-plus spectral policy iteration algorithm à la Howard which is detailed below. In the case of the equation $x = ax \oplus b$, we can prove that this policy iteration algorithm allways requires less steps than value iteration. It remains to implement it.

II. Solving the Spectral Problem $Ax = \lambda x$

A. Computing the Maximal Circuit Mean

We first recall the following classical result.

Theorem 2: An irreducible matrix A with entries in the max-plus semiring $\mathbb{R}_{\max}$ has a unique eigenvalue $\rho(A)$, which is given by the maximal mean weight of the circuits of A.
In algebraic terms, for an $n \times n$ matrix:

$$\rho(A) = \mathrm{tr}(A) \oplus (\mathrm{tr}(A^2))^{(1/2)} \oplus \cdots \oplus (\mathrm{tr}(A^n))^{(1/n)} \quad ,$$

where $\mathrm{tr}(A) = A_{11} \oplus \cdots \oplus A_{nn}$. This formula yields a naive algorithm to compute the eigenvalue:

```
file: naiveeigenv.sci

function t=mptrace(a)
//max-plus trace
d=diag(a)
t=%ones(1,size(d,1))*d

//we overload the entrywise
//exponent operator, named .ˆ
//so that it works for maxplus matrices
//(see help overload)

function b=%talg_j_s(a,s)
b=#(plustimes(s)*plustimes(a))


function rho=naiveeigenv(a)
n=size(a,1)
x=a
t=mptrace(a)
for i=2:n
x=x*a
t=t + (mptrace(x)).ˆ(1/i)
end
rho=t
```

We can load this macro in Scilab with:

```
getf('naiveeigenv.sci')
```

Let us check the macro for scalars

```
naiveeigenv(#(1))
 ans  =

    1.
naiveeigenv(%0)
 ans  =

  -Inf
naiveeigenv(%top)
 ans  =

   Inf
```

Let us try now matrices

```
a=#([1,4;-1,%0])
 a  =

!   1.    4. !
! - 1.  -Inf !

rho=naiveeigenv(a)
 rho  =

    1.5
```

Is it correct ?

```
b=rhoˆ(-1)*a
 b  =

! - 0.5    2.5 !
! - 2.5  -Inf  !
```

```
c=plus(b)
 c  =

!   0.      2.5 !
! - 2.5    0.  !
```

The answer should be zero:

```
mptrace(c)
 ans  =

    0.
```

Let us try a larger matrix

```
a=#([-1 -3 0 ; -10 -5 2; -1 -4 0])
 a  =

! - 1.   - 3.    0. !
! - 10.  - 5.    2. !
! - 1.   - 4.    0. !

// Guess what the eigenvalue is...

naiveeigenv(a)

[answer suppressed]
```

Since a is irreducible, the cyclicity theorem tells us that $a^{k+c} = \rho^c a^k$, for some $k, c \geq 1$. Let us look manually for the least k and c (in fact, we know from the theory that $c = 1$).

```
a==a^2
 ans  =

! T F T !
! F F T !
! T T T !

a^2==a^3
 ans  =

! T T T !
! T T T !
! T T T !
```

Hence, $k = 2, c = 1$. In general, the length of the transient (i.e. the minimal value of k) can be arbitrarily large. Let us build such a pathological example:

```
a(1,3)=-1
 a  =

! - 1.   - 3.  - 1. !
! - 10.  - 5.    2. !
! - 1.   - 4.    0. !

a^2==a^3
 ans  =

! T F T !
! T T T !
! T T T !

a^3==a^4
 ans  =

! T T T !
! T T T !
! T T T !

//

a(2,3)=-5
 a  =

! - 1.   - 3.  - 1. !
! - 10.  - 5.  - 5. !
! - 1.   - 4.    0. !

a(1,3)=-5
 a  =

! - 1.   - 3.  - 5. !
! - 10.  - 5.  - 5. !
! - 1.   - 4.    0. !

a^3==a^4
 ans  =

! F F T !
! T T T !
! T T T !

a^8==a^7
 ans  =

! T T T !
! T T T !
! T T T !

a^7==a^6
 ans  =

! T F T !
! T T T !
! T T T !
```

Exercise: explain why, for this example, the length of the transient increases to infinity when $a(1,3)$ and $a(2,3)$ both decrease to $-\infty$.

(Of course, the use of hash tables in SEMIGROUPE allows much more efficient algorithms to compute the least k, and its non-commutative generalizations).

Let us try now a big matrix

```
a=#(rand(64,64));

timer(); rho=naiveeigenv(a)
 rho  =
    0.9913730
timer()
 ans  =

    7.316374
```

The execution time is not brilliant. Fortunately, there are faster algorithms, e.g., Karp's [8].

```
timer(); rho2=karp(a)
 rho2  =

    0.9913730
timer()
 ans  =

    0.349986
```

This is much better, but is the result of `karp` correct ?

```
rho==rho2
 ans  =

  F
```

The answer is false, but we should not panic... we did quite complex computations in `naiveeigenv`, and arithmetical errors have accumulated. Let us check that this is the case ...

```
plustimes(rho)-plustimes(rho2)
 ans  =

  - 1.110D-16
```

B. *Computing the Cycle Time via Karp's and Howard's algorithms*

Now, it is time to give more technical details about Karp's algorithm. Karp proved that if A is irreducible, for all index i,

$$\rho(A) = \max_{\substack{1\leq j\leq n \\ (A^n)_{ij}\neq -\infty}} \min_{1\leq k\leq n} \frac{(A^n)_{ij} - (A^{n-k})_{ij}}{k} \quad . \tag{1}$$

In fact, the original redaction of Karp exchanges the role of i and j, but this is a detail, and we will see soon why (1) is preferable.

The purists wanting to avoid this (rather monstruous) crossing of algebras should write, with the max-plus notation:

$$\rho(A) = \bigoplus_{\substack{1\leq j\leq n \\ (A^n)_{ij}\neq \mathbb{0}}} \bigwedge_{1\leq k\leq n} \left(\frac{(A^n)_{ij}}{(A^{n-k})_{ij}}\right)^{\frac{1}{k}}$$

It turns out that Karp's algorithm is also interesting in the case of reducible matrices. To explain the more general quantity that it computes, we need the following definition.

Definition 1: The *cycle time* of a $n \times n$ matrix A with entries in the max-plus semiring $\mathbb{R}_{\max}$ is the n-dimensional column vector $\chi(A)$, given by

$$\chi_i(A) = \lim_k (A^k x)_i^{(1/k)}, \quad i = 1 \dots n,$$

where x is an arbitrary *finite* vector.

Theorem 3 (SG, unpublished) Karp's formula (1), invoked at index i, returns the i-th coordinate of the cycle time vector of the matrix A.

The function `karp` that we have implemented here takes a second optional argument, which is precisely the index i. By default, $i = 1$. The function returns the i-th coordinate of the cycle time of A.

```
a=#([ 2 %0; %0 3])
 a  =

!   2.  -Inf !
! -Inf    3. !

karp(a)
 ans  =

    2.
karp(a,1)
 ans  =

    2.
karp(a,2)
 ans  =

    3.
a(1,2)=2
 a  =

!   2.    2. !
! -Inf    3. !

karp(a,1)
 ans  =

    3.
karp(a,2)
 ans  =

    3.
```

Is it correct ?

```
a^100*%ones(2,1)
 ans  =

!   299. !
!   300. !
```

We know from the theory that $\chi_i(A)$ is equal to the max of the eigenvalues of the strongly connected components of the graph of A to which i has access. Let us check this.

```
a(2,2)=1
 a  =

!   2.    2. !
! -Inf    1. !

karp(a,1)
```

```
 ans  =

    2.
karp(a,2)
 ans  =

    1.
a^100*%ones(2,1)
 ans  =

!   200. !
!   100. !
```

Fine ... but if we want to compute the n entries of the cycle time vector, shall we invoke `karp` n times ? Of course, no ... the cycle time vector is constant on each strongly connected component of the graph of A, hence, it is enough to invoke `karp` only *once* per strongly connected component.

We next show how we can compute these components using metanet. First, we build the adjacency matrix of the graph of A (the first argument that `spget` returns is a $n \times 2$ vector ij: the k-th arc of the graph goes from $ij(k, 1)$ to $ij(k, 2)$).

```
ij=spget(sparse(a))
 ij  =

!   1.    1. !
!   1.    2. !
!   2.    2. !
```

We turn it to a 0-1 adjacency matrix for use by metanet

```
adjacency=sparse(ij,ones(1,size(ij,1)))
 adjacency  =

(    2,    2) sparse matrix

(    1,    1)        1.
(    1,    2)        1.
(    2,    2)        1.
```

Let us see how it looks

```
full(adjacency)
 ans  =

!   1.    1. !
!   0.    1. !

g=mat_2_graph(adjacency,1,'node-node')
 g  =

       g(1)

         column 1 to 8

!graph  name  directed  node_number

[suppressed output]
```

The argument 1 in the last expression stands for directed. This is a huge list... for the graph may contains much more information than its adjacency structure.

```
show_graph(g)
 ans  =

    1.
[ncomp,nc]=strong_connex(g)
 nc  =

!   2.    1. !
 ncomp  =

    2.
```

We found that the graph has 2 strongly connected components, which are {2} and {1}, respectively. Let us see what happens if we modify the graph. First, we automatize the process, by creating the macro `mp_2_graph`, which transforms a max-plus matrix to a graph for use in metanet.

```
getf('mp_2_graph.sci')

a(1,3)=2
 a  =

!   2.    2.    2. !
! -Inf    1.  -Inf !

a(3,3)=1
 a  =

!   2.    2.    2. !
! -Inf    1.  -Inf !
! -Inf  -Inf    1. !

g2=mp_2_graph(a);

show_graph(g2);

[ncomp,nc]=strong_connex(g2)
 nc  =

!   3.    2.    1. !
 ncomp  =

    3.
a(3,1)=0
 a  =

!   2.    2.    2. !
! -Inf    1.  -Inf !
!   0.  -Inf    1. !

g3=mp_2_graph(a);

show_graph(g3);
```

```
[ncomp,nc]=strong_connex(g3)
 nc  =

!   2.    1.    2. !
 ncomp  =

    2.
```

It is now easy to buil the irreducible blocks of A. E.g., here is the second connected component of the graph:

```
I=find(nc==2)
 I  =

!   1.    3. !
```

and here is the $I \times I$ submatrix of a:

```
A=a(I,I)
 A  =

!   2.    2. !
!   0.    1. !
```

We could use this to compute efficiently the cycle time of a. However, another algorithm, namely, Howard's policy iteration, computes directly *all* the coordinates of the cycle time vector, and in a faster way. The algorithm is documented in [3]. The Scilab primitive is named `howard`:

```
chi=howard(A)
 chi  =

!   2. !
!   2. !
```

Optionnaly, `howard` returns a *bias vector* (which is defined below):

```
[chi,v]=howard(A)
 v  =

!   2. !
!   0. !
 chi  =

!   2. !
!   2. !
```

When A is irreducible, the bias vector v is nothing but an eigenvector:

```
A*v
 ans  =

!   4. !
!   2. !

v
 v  =

!   2. !
!   0. !
```

In the reducible case, by definition, the bias vector v is such that

$$a(v + k \times \chi) = v + (k+1)\chi \quad ,$$

for all k large enough. Let us check this with the above reducible matrix:

```
[chi,v]=howard(a)
 v  =

!   2. !
!   1. !
!   0. !
 chi  =

!   2. !
!   1. !
!   2. !
//(tentative dirty conversions...)

v1=plustimes(v)+plustimes(chi)
 v1  =

!   4. !
!   2. !
!   2. !

a*#(v1)==#(plustimes(v1)+plustimes(chi))
 ans  =

! T !
! T !
! T !

v1=plustimes(v1)+plustimes(chi)
 v1  =

!   6. !
!   3. !
!   4. !

a*#(v1)==#(plustimes(v1)+plustimes(chi))
 ans  =

! T !
! T !
! T !
```

Let us see how fast these three algorithms are for large matrices.

```
a=#(rand(100,100));

timer();h=howard(a);timer()
 ans  =

    0.08333

timer();k=karp(a);timer()
 ans  =
    0.266656

k==h(1)
 ans  =

  T
```

karp and howard yield less numerical errors than naiveeigenv, hence, the answer was true here. Much of the time is spent in the interface for such relatively small matrices. The advantage of howard becomes clear for large matrices, particularly for sparse ones.

```
a=#(sprand(500,500,0.02));

timer();h=howard(a);timer()
 ans  =

    0.066664

//karp

timer();k=karp(a);timer()
 ans  =

    0.91663

k==h(1)
 ans  =

  T
```

Yet a larger one:

```
timer();a=#(sprand(2000,2000,0.01));timer()
 ans  =

    0.983294

h=howard(a);timer()
 ans  =

    0.783302
```

In other words, computing the cycle time vector via howard takes a time which is comparable to the generation of the random matrix. Using karp here would be too slow for the demo (howard takes experimentally an almost linear (=O(number of arcs)) time, karp takes an $O(n \times$ number of arcs) time).

C. Computing the Eigenspace

Possibly after dividing A by $\rho(A)$, we may always assume that $\rho(A) = \mathbb{1}(= 0)$. We will only consider here the case an an irreducible matrix (the reducible case involves decomposing first A in irreducible blocks, see [4][chap 4] and [6] for the characterization of the spectrum in this case). Then, the minimal generating family[1] of the eigenspace is obtained by selecting exactly one column of A^* per strongly connected component of the *critical graph* of A (which is the subgraph of the graph of A composed of the circuits whose mean weight is $\rho(A)$).

Consider

```
a=#([0 -2 -10 ; 0 -3 -5; -1 5 -8])
 a  =

!   0.  - 2.  - 10. !
!   0.  - 3.  - 5.  !
! - 1.    5.  - 8.  !
```

We first compute an eigenvector of a using howard

```
[chi,v]=howard(a)
 v  =

!   0. !
!   0. !
!   5. !
 chi  =

!   0. !
!   0. !
!   0. !
```

Then, we perform a diagonal change of variables

```
getf('mpdiag.sci')

deff('[b]=dadinv(a,v)',..
'b=mpdiag(v^(-1))*a*mpdiag(v)')

b=dadinv(a,v)
 b  =

!   0.  - 2.  - 5. !
!   0.  - 3.    0. !
! - 6.    0.  - 8. !
```

We compute the saturation graph, whose non-trivial strongly connected components form the critical graph.

```
[ir,ic]=find(b==#(0))
 ic  =

!   1.    1.    2.    3. !
 ir  =

!   1.    2.    3.    2. !

adjacency=sparse([ir',ic'],..
ones(1,size(ir,2)))
 adjacency  =

(    3,    3) sparse matrix

(    1,    1)        1.
(    2,    1)        1.
```

[1]The minimal generating family is unique, up to a permutation and a scaling.

```
(    2,    3)        1.
(    3,    2)        1.
full(adjacency)
 ans  =

!   1.    0.    0. !
!   1.    0.    1. !
!   0.    1.    0. !

g=mat_2_graph(adjacency,1,'node-node');

show_graph(g)
 ans  =

    1.
[ncomp,nc]=strong_connex(g)
 nc  =

!   1.    2.    2. !
 ncomp  =

    2.
c=plus(b)
 c  =

!   0.  - 2.  - 2. !
!   0.    0.    0. !
!   0.    0.    0. !
```

We select one node per strongly connected component of the saturation graph.

```
critical=[]
 critical  =

     []

basis=#([])
 basis  =

     []

for i=1:ncomp
j=min(find(nc==i))
critical(i)=j
if (c(j,j)==#(0))
basis=[basis,c(:,j)]
end
end
 j  =

    1.
 critical  =

    1.
 basis  =

!   0. !
!   0. !
!   0. !
 j  =

    2.
 critical  =

!   1. !
!   2. !
 basis  =

!   0.  - 2. !
!   0.    0. !
!   0.    0. !
```

Now, basis is a minimal generating family of the eigenspace. Let us automatize this process

```
getf('eigenspace.sci')

a=#([3 0 %0; 0 3 %0 ; 2 1 2])
 a  =

!   3.    0.  -Inf !
!   0.    3.  -Inf !
!   2.    1.    2. !


[v,rho]=eigenspace(a)
 rho  =

    3.
 v  =

!   0.  - 3. !
! - 3.    0. !
! - 1.  - 2. !

// Consistency check

a*v==rho*v
 ans  =

! T T !
! T T !
! T T !
```

The first output argument of `eigenspace` is (of course) a generating family of the eigenspace for the maximal eigenvalue of the matrix. The second (optional) output argument is the maximal eigenvalue of the matrix.

D. Computing the Spectral Projector

If A has maximal eigenvalue $\mathbb{1}$, the matrix P, defined by

$$\lim_{k\to\infty} A^k A^* = P$$

satisfies $AP = PA = P = P^2$. The matrix P is called the spectral projector of A, for its image is precisely the eigenspace of A (we call image of A its column space, i.e. the set of vectors of the form Ax, where x is an arbitrary column vector of appropriate size).

```
getf('projspec.sci')
P=projspec(a)
 P  =

!   0.  - 3.  -Inf !
! - 3.    0.  -Inf !
! - 1.  - 2.  -Inf !
```

Let us check this value by simulation

```
b=rho^(-1)*a
 b  =

!   0.  - 3.  -Inf !
! - 3.    0.  -Inf !
! - 1.  - 2.  - 1. !

Q=b^100*(b^0+b)^100
 Q  =

!   0.  - 3.  -Inf   !
! - 3.    0.  -Inf   !
! - 1.  - 2.  - 100. !
```

Let us now enlarge a, creating another strongly connected component of the critical graph. First, we add a circuit with mean $6/2 = 3 = \rho(a)$.

```
a(4,5)=5;
a(5,4)=1;
 a  =

!   3.    0.  -Inf  -Inf  -Inf !
!   0.    3.  -Inf  -Inf  -Inf !
!   2.    1.    2.  -Inf  -Inf !
! -Inf  -Inf  -Inf  -Inf    5. !
! -Inf  -Inf  -Inf    1.  -Inf !
```

Second, we add other non-critical arcs

```
a(1,4)=-8;
a(5,3)=-7
 a  =

!   3.    0.  -Inf  - 8.  -Inf !
!   0.    3.  -Inf  -Inf  -Inf !
!   2.    1.    2.  -Inf  -Inf !
! -Inf  -Inf  -Inf  -Inf    5. !
! -Inf  -Inf  - 7.    1.  -Inf !
```

Let us compute the eigenspace

```
[v,rho]=eigenspace(a)
 rho  =

    3.
 v  =

!   0.   - 3.   - 11. !
! - 3.     0.   - 14. !
! - 1.   - 2.   - 12. !
! - 9.   - 10.    0.  !
! - 11.  - 12.  - 2.  !

a*v==rho*v
 ans  =

! T T T !
! T T T !
! T T T !
! T T T !
! T T T !
```

Let us compute the spectral projector

```
P=projspec(a)
 P  =

!   0.   - 3.   - 19.  - 11.  - 9.  !
! - 3.     0.   - 22.  - 14.  - 12. !
! - 1.   - 2.   - 20.  - 12.  - 10. !
! - 9.   - 10.  - 8.     0.     2.  !
! - 11.  - 12.  - 10.  - 2.     0.  !
```

Let us compare it with the result of simulation

```
b=rho^(-1)*a;
Q=b^100*(b^0+b)^100
 Q  =

!   0.   - 3.   - 19.  - 11.  - 9.  !
! - 3.     0.   - 22.  - 14.  - 12. !
! - 1.   - 2.   - 20.  - 12.  - 10. !
! - 9.   - 10.  - 8.     0.     2.  !
! - 11.  - 12.  - 10.  - 2.     0.  !

P==Q
 ans  =

! T T T T T !
! T T T T T !
! T T T T T !
! T T T T T !
! T T T T T !
```

Let us check that the spectral projector leaves the eigenspace invariant:

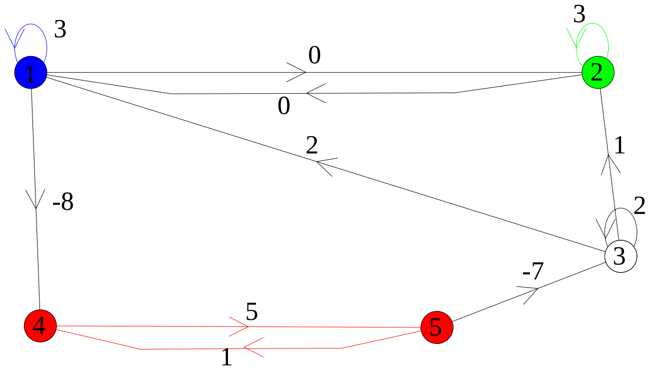
```
P*v==v
 ans  =

! T T T !
! T T T !
! T T T !
! T T T !
! T T T !
```

E. *Displaying the Critical Graph*

Finally, the macro `spectral_analysis` generates the graph of a matrix, and distinguishes the different strongly connected components of the critical graph by colors.

```
getf('spectral_analysis.sci')
g=spectral_analysis(a);
show_graph(g)
```

We edited the graph via metanet, saved it in a file, and generated a `xfig` file via `plot_graph`. Currently, the .fig output is less nice than what we see on the metanet window. Thus, we had to modify it slightly in `xfig` to make it prettier: we choosed better colors, fonts of appropriate size, made the nodes opaque, and slightly reshaped some arcs (of course this should be automatized). Here is the result:



III. Solving the Inverse Problem $Ax = b$ via Residuation

A. *Mere Residuation*

Let A, B, X denote matrices with entries in the completed max-plus semiring $\overline{\mathbb{R}}_{\max}$. We recall the following basic result of residuation theory.

Theorem 4: The maximal solution of $AX \leq B$ is given by $X = A\backslash B$, where

$$(A\backslash B_{ij} = \min_k(-A_{ki} + B_{kj})$$

Generically, $AX = b$ has no solution. The matrix $A\backslash B$ is called the *left residual* of A and B. Dually, the maximal solution of $YA \leq B$ is denoted by the *right residual* B/A. When A is invertible, $A\backslash B$ coincides with $A^{(-1)}B$.

```
a=#(3)
 a  =

    3.
b=#(4)
 b  =

    4.
a/b
 ans  =

  - 1.
a==(a/b)*b
 ans  =

  T
a/%0
 ans  =

   Inf
a/%top
 ans  =

  -Inf
%0/%0
 ans  =

   Inf
%top/%top
 ans  =

   Inf
```

Matrix case:

```
a=#([2,3;%0,1])
 a  =

!   2.     3. !
! -Inf     1. !

b=#([10; 100])
 b  =

!   10.  !
!   100. !

a\b
 ans  =

!   8. !
!   7. !
```

Exercise: prove that:

$$(a/a)a = a; \quad (a/a)^2 = a/a \ .$$

These properties allow a consistency check:

```
p=a/a
 p  =

!   0.     2. !
! -Inf     0. !

p*a==a
 ans  =

! T T !
! T T !

p==p^2
 ans  =
! T T !
! T T !
```

Invertible case:

```
a=#([%0 %1 %0; %0 %0 %1; %1 %0 %0])
 a  =

! -Inf    0.  -Inf !
! -Inf  -Inf    0. !
!   0.  -Inf  -Inf !

b=#([1;2;3])
 b  =

!   1. !
!   2. !
!   3. !

a\b
 ans  =

!   3. !
!   1. !
!   2. !

a'*b
 ans  =

!   3. !
!   1. !
!   2. !
```

(the inverse of the permutation matrix a is its transpose).

Residuation allows us to determine if a vector b belongs to the image of a matrix A. Indeed, b belongs to Im A iff $b = A(A\backslash b)$. Exercise: draw the image of the following matrix:

```
a=#([0,2;%0,0])
 a  =

!   0.    2. !
! -Inf    0. !
```

Answer

```
b=#([4;0])
 b  =

!   4. !
!   0. !

b==a*(a\b)
 ans  =

! T !
! T !

b=#([3;0])
 b  =

!   3. !
!   0. !

b==a*(a\b)
 ans  =

! T !
! T !

b=#([2;0])
 b  =

!   2. !
!   0. !

b==a*(a\b)
 ans  =

! T !
! T !

b=#([1;0])
 b  =

!   1. !
!   0. !

b==a*(a\b)
 ans  =

! T !
! F !

b=#([0;0])
 b  =

!   0. !
!   0. !

b==a*(a\b)
 ans  =

! T !
! F !
```

(Im a is the set of column vectors (x_1, x_2) such that $x_1 \geq 2+x_2$).

B. *Computing Minimal Generating Families*

Let F denote a finite set of pairwise non-proportional vectors of $(\mathbb{R}_{\max})^n$. We say that a vector v in this set F is *redundant* if it belongs the the semimodule generated by the vectors of F distinct of v.

Theorem 5: By deleting redundant vectors of the finite set F, we obtain a minimal generating set G of the semimodule that it generates. This set G is unique, up to multiplication of its elements by invertible constants.

Since residuation allows us to determine redundant vectors, we can easily build minimal generating families. In fact, we do not check that $b = A(A\backslash b)$, but we rather use the "east-europe" variant of this algorithm (which can be found e.g. in U. Zimmermann's book [9] or in P. Butkovic's survey [2]). The algorithm is readily obtained from the following result:

Theorem 6: The vector $b \in (\mathbb{R}_{\max})^n$ belongs to the image of $A \in (\mathbb{R}_{\max})^{n\times p}$ iff

$$\bigcup_{1\leq i\leq p} \operatorname*{arg\,min}_{1\leq j\leq n}(-A_{ji} + b_j) = \{1, \dots, n\} \ .$$

Checking this is twice faster than checking that $A(A\backslash b) = b$. The Scilab macro is named `inspan`. `inspan`(a, b) returns true if the vector b is in the image of the matrix a.

```
inspan(a,b)
 ans  =

  F
b=#([3;0])
 b  =

!   3. !
!   0. !

inspan(a,b)
 ans  =

  T
```

Similarly, `includespan`(A, B) returns true if $\operatorname{Im} B$ is included in $\operatorname{Im} A$, and `equalspan`(A, B) returns true if $\operatorname{Im} A$ is equal to $\operatorname{Im} B$.

```
includespan(a,b)
 ans  =

  T
includespan(b,a)
 ans  =

  F
b=%ones(2,1)
 b  =

!   0. !
!   0. !

includespan(a,b)
 ans  =

  F
equalspan(a,b)
 ans  =

  F
equalspan(a,a)
 ans  =

  T
equalspan(b,b)
 ans  =


  T
```

`weakbasis`(A) returns a matrix whose colums form a minimal generating set of the column space of A:

```
weakbasis(a)
 ans  =


!   2.    0. !
!   0.  -Inf !
```

Finitely generated subsemimodules of $(\mathbb{R}_{\max})^2$ have minimal generating sets with 0, 1, or 2 elements

```
a=#([0 2 3; 7 5 2])
a  =

!   0.    2.    3. !
!   7.    5.    2. !

b=weakbasis(a)
 b  =



!   3.    0. !
!   2.    7. !

a=#(rand(2,20))
 a  =
         column 1 to 5

!   0.7093614    0.2281042    0.5695345...
!   0.3137576    0.3097598    0.0957654...
[output suppressed]
b=weakbasis(a)
 b  =

!   0.0405107    0.7819632 !
!   0.7767725    0.1604007 !

equalspan(a,b)
 ans  =
  T
```

Finitely generated subsemimodules of $(\mathbb{R}_{\max})^3$ can have minimal generating sets of arbitrarily large cardinality.

```
a=#([0,0,0;0,-1,-2;0,1,2])
 a  =

!   0.    0.    0. !
!   0.  - 1.  - 2. !
!   0.    1.    2. !

weakbasis(a)
 ans  =

!   0.    0.    0. !
! - 2.    0.  - 1. !
!   2.    0.    1. !

a=[a,#([0;-3;3])]
 a  =

!   0.    0.    0.    0. !
!   0.  - 1.  - 2.  - 3. !
!   0.    1.    2.    3. !

weakbasis(a)
 ans  =

!   0.    0.    0.    0. !
! - 3.    0.  - 1.  - 2. !
!   3.    0.    1.    2. !

a=[a,#([0;-4;4])]
 a  =

!   0.    0.    0.    0.    0. !
!   0.  - 1.  - 2.  - 3.  - 4. !
!   0.    1.    2.    3.    4. !

weakbasis(a)
 ans  =

!   0.    0.    0.    0.    0. !
! - 4.    0.  - 1.  - 2.  - 3. !
!   4.    0.    1.    2.    3. !
```

For an $n \times p$ matrix, `weakbasis` runs in np^2 time

```
a=#(rand(10,200));

timer();

b=weakbasis(a);

timer()
 ans  =

   1.33328
size(b)
 ans  =

!   10.    195. !

equalspan(a,b)
 ans  =

  T
```

IV. Solving $Ax = Bx$

This is an interesting research subject. Please ask privately to see the demo of the currently implemented algorithm.

Appendix

I. Loading the Max-plus Environment

The following Scilab command, which can be executed directly in Scilab, or put in the `~/scilab.star` initialization file, links incrementally Scilab with the max-plus libraries, and defines some max-plus macros.

```
exec(SCI+'/routines/maxplus/mploader.sce')
```

II. Availability

The max-plus toolbox requires the version 2.4 of Scilab, which will be released in the next few days. The max-plus toolbox will be made available via the web pages of the authors, as soon as released (hopefully not much later than the version 2.4 of Scilab).

References

[1] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[2] Peter Butkovič. Strong regularity of matrices — a survey of results. *Discrete Applied Mathematics*, 48:45–68, 1994.

[3] J. Cochet-Terrasson, Guy Cohen, Stéphane Gaubert, Michael Mc Gettrick, and Jean-Pierre Quadrat. Numerical computation of spectral elements in max-plus algebra. In *IFAC Conference on System Structure and Control*, Nantes, France, July 1998.

[4] S. Gaubert. *Théorie des systèmes linéaires dans les dioïdes*. Thèse, École des Mines de Paris, July 1992.

[5] S. Gaubert. Two lectures on max-plus algebra. In *Proceedings of the 26-th Spring School on Theoretical Computer Science and Automatic Control*, Noirmoutier, May 1998.

[6] S. Gaubert and M. Plus. Methods and applications of (max,+) linear algebra. In R. Reischuk and M. Morvan, editors, *STACS'97*, number 1200 in LNCS, Lübeck, March 1997. Springer.

[7] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, Paris, 1979. Engl. transl. *Graphs and Algorithms*, Wiley, 1984.

[8] R.M. Karp. A characterization of the minimum mean-cycle in a digraph. *Discrete Maths.*, 23:309–311, 1978.

[9] U. Zimmermann. *Linear and Combinatorial Optimization in Ordered Algebraic Structures*. North Holland, 1981.

Index of Primitives for Max-plus Linear Algebra

These primitives are written in C and interfaced with Scilab. The other functionalities presented here (except basic matrix operations, including residuation, which are at FORTRAN level) are Scilab macros, which make use of the above primitives and of the general Scilab facilities for handling max-plus objects.