

Computer Laboratory

Exercises in

**LINEAR PROGRAMMING AND  
COMBINATORIAL OPTIMIZATION**



**LUND INSTITUTE OF TECHNOLOGY**

**DEPARTMENT OF MATHEMATICS**

**1999**



# Preparation for the Labs

If you want to change your password, give the command `yppasswd` and follow the instructions.

Download the matlab files needed for the laboration at

<http://www.maths.lth.se/matematiklth/personal/kalle/kombopt/lab1.tar.gz>

Copy this file to your home directory, decompress and unpack it.

```
gzip -d lab1.tar.gz
tar xvf lab1.tar
```

Now there is a subdirectory `lab1`. This is a tree structure. You change directory with the command `cd`, e.g. `cd lab1`. The command `cd ..` puts you in the parent directory. The command `ls` shows the files and the subdirectories at the present level in the “tree”.

## Emacs

During the laboration you need to edit some files. The editor **Emacs** in the UNIX-system is here briefly described.

**Start:** Give the command `emacs&` or `emacs filename&`.

**Start Emacs from MATLAB:** Write `!emacs filename&`. Here the sign `!` makes it possible to give UNIX commands during a MATLAB run.

**Read a file into Emacs:** `C-X C-F` (means: press `control` and `x` simultaneously, then press `control` and `f` simultaneously) and answer with the filename.

**Save a file:** `C-X C-S`.

**Exit Emacs permanently and save files:** Give the command `C-X C-C`. There is no need to do this before you have finished the session. It is convenient to keep the Emacs window during the whole run. When you wish to edit a file, just move the mouse to the Emacs window, make the changes and save the file.

It is easy to press a wrong key, especially some control key. Regretting a command is generally done with `C-G`.

Further information can be obtained by the built in help function (`C-H`).

## Some Useful MATLAB Functions

First an explanation of the command is given with the MATLAB `help` command and then an example is given:

```
FIND Find indices of the non-zero elements.  
I = FIND(X) returns the indices of the vector X that are  
non-zero. For example, I = FIND(A>100), returns the  
indices of A where A is greater than 100. See RELOP.
```

```
>> b=[3 5 -2 0 9 -3 -4 8];  
>> find(b>0)  
ans =  
     1     2     5     8
```

```
>> help max
```

```
MAX Largest component.  
For vectors, MAX(X) is the largest element in X. For  
matrices, MAX(X) is a vector containing the maximum element  
from each column. [Y,I] = MAX(X) stores the indices of the  
maximum values in vector I. MAX(X,Y) returns a matrix the  
same size as X and Y with the largest elements taken from X  
or Y. When complex, the magnitude MAX(ABS(X)) is used.
```

```
>> max(b)  
ans =  
     9  
>> [maxvalue,index]=max(b)  
maxvalue =  
     9  
index =  
     5
```

```
>> A=[3 5 4 2 0;7 6 3 5 1;8 6 9 2 0]  
A =  
     3     5     4     2     0  
     7     6     3     5     1  
     8     6     9     2     0  
>> [maxv,index]=max(A)
```

```
maxv =  
      8      6      9      5      1  
index =  
      3      2      3      2      2
```

```
>> help sum
```

```
SUM      Sum of the elements.  
For vectors, SUM(X) is the sum of the elements of X.  
For matrices, SUM(X) is a row vector with the sum over  
each column. SUM(DIAG(X)) is the trace of X.
```

```
See also PROD, CUMPROD, CUMSUM.
```

```
>> sum(A)  
ans =  
     18     17     16      9      1  
>> sum(sum(A))  
ans =  
     61
```

Note also the element-by-element operations, for example, multiplication :

```
>> B=[1 1 1 1 1;0 0 0 0 0;2 2 2 2 2]  
B =  
      1      1      1      1      1  
      0      0      0      0      0  
      2      2      2      2      2  
>> A.*B  
ans =  
      3      5      4      2      0  
      0      0      0      0      0  
     16     12     18      4      0
```

# Laboratory Exercise 1

This lab is about linear programming. First you should run an available MATLAB program that demonstrates the simplex method. You choose the pivot element on every iteration. Then you have to modify the program so that the pivot elements are chosen automatically. Finally you should test your program on some examples.

You should also make a version that produces the result as fast as possible, and investigate how the execution time increases with the number of constraints and variables.

Cycling can be prevented with *Bland's rule*:

1. If there are more than one variable to enter the basis (columns with negative reduced costs), then choose the one with the lowest index.
2. If there are more than one variable to leave the basis, then choose the one with the lowest index. (Note: the lowest index of a *basis* variable.)

The problem to solve at this session is

$$\begin{array}{ll} \text{maximize} & z = \mathbf{c}^T \mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{x} \geq \mathbf{0}. \end{array}$$

It is assumed that a feasible choice of basic variables is given. This is easy if the problem has been converted from its standard form with  $b \geq 0$ . Why?

**The simplex method in MATLAB.** The following procedure requires that the user assigns the correct pivot element on every iteration. Observe that the initial tableau is constructed with the function `checkbasic1` from `inlämningsuppgift1`.

```
function [y,basicvars,steps]=simplmovie(A,b,c,basicvars)
% [y,basicvars,steps] = simplmovie(A,b,c,basicvars)
%
% A    m*n-matrix
% b    m*1-matrix, b>=0
% c    n*1-matrix
% basicvars 1*m - matrix with indices for feasible basic variables.
%
% Shows a movie of how the simplex method works
% on the problem
%
%           max(c'x), when Ax=b, x>=0.
%
[m,n]=size(A);
```

```

% Create a tableau with slack variables
[y,x,basic,feasible,optimal]=checkbasic1(A,b,c,basicvars);
clc, disp('initial tableau')
y
steps=0;

% Loop until all reduced costs are non-negative
while min(y(m+1,1:n)) < 0
    % input pivot indices (p,q)
    q=input('entering variable   q = ');
    p=input('bounding constraint p = ');
    % Divide the pivot row p with the pivot element y(p,q)
    clc, disp(['pivot around element ( ' int2str(p) ', ' int2str(q) ')'])
    y(p,:)=y(p,+)/y(p,q)
    basicvars(p)=q;
    steps=steps+1;
    pause
    % Subtract multiples of the pivot row from all other rows
    for row=[1:p-1 p+1:m+1]
        clc, disp(['eliminate row ' int2str(row)])
        y(row,:)=y(row,)-y(row,q)*y(p,+)
        pause
    end
    if min(y(:,n+1))<0
        disp('*****')
        disp('You have chosen an incorrect pivot element.')
        disp('Restart!')
        disp('*****')
        break
    end
end
end
if min(y(m+1,1:n))>=0 & min(y(:,n+1))>=0
    disp('Congratulation! You have understood the simplex method.')
    disp('(At least for this example.)')
end
end

```

PREPARATORY EXERCISE. Rewrite the program above and name it `simp(A,b,c,basicvars)`, so that the pivot elements are chosen automatically. Try to produce the answer as fast as possible. The MATLAB functions `min` and `find` are useful. Unbounded problems should be detected. The problem with cycling can be solved by using Bland's rule, but this is difficult and not compulsory. The number

of steps of the simplex algorithms is returned from the function. The number of flops is easily counted with the MATLAB function `flops`:

```
flops(0)
```

```
. . .
```

```
flops
```

(You can also use the function `etime`)

## At the computer

1. Go to the subdirectory `lab1`.
2. Suitable test matrices of size  $m \times n$  can be obtained with the procedure `init`:

```
A=9*rand(m,n)+1;  
b=ones(m,1);  
c=ones(n,1);  
A=[A eye(m)];  
c=[c;zeros(m,1)];  
basicvars=(n+1):(m+n);
```

A trial run with a  $3 \times 3$ -matrix is thus done by:

```
>>m=3  
>>n=3  
>>init  
>>[y,basicvars,steps]=simplmovie(A,b,c,basicvars)
```

3. To study the phenomenon cycling, write `chvatal` to create the matrices

```
A=[0.5 -5.5 9 -2.5 ;  
    0.5 -1.5 1 -0.5 ;  
    1 0 0 0]  
b=[0 0 1]'  
c=[-10 57 24 9]'
```

Choose the pivot elements according to “the most negative reduced cost should enter the basis” and “if more than one variable can leave the basis, then choose the one with the lowest index”. (The cycling indices are  $(p, q) = (1, 1), (2, 2), (1, 4), (2, 3), (1, 5), (2, 6)$ .)

4. Make the copy

```
>>!cp simpmovie.m simp.m
```

and adjust `simp` with Emacs according to the preparatory exercise. Test your program on Example 1 in section 2.1 of the book (tableau 2.1, 2.3 and 2.4). Test random matrices (created by `init`) of sizes from  $10 \times 10$  to  $100 \times 100$  and make a table of the number of flops or elapsed time. How does the execution time seem to depend on  $m$  and  $n$ ? How does the number of steps grow with  $m$  and  $n$ ? Exponential or polynomial? If you have solved the problem with cycling, then try the Chvatal example, too.

5. For every rule on how to choose the pivot element it is possible to construct examples that make the simplex method very slow. Take a look at `simpb` (use emacs or write `type simpb` from matlab) which is a simplex method using a 'Bland'-like rule for selecting the incoming basic variable ( $q$ ). The script `badexample` creates a difficult example for this simplex algorithm. The variable  $d$  sets the size of the matrices  $A$  of the problem.

```
>>d=3
```

```
>>badexample
```

```
>>[y,basicvars,steps]=simpb(A,b,c,basicvars)
```

How does the number of steps grow with  $d$ ? Is simplex a polynomial time solution for the linear programming problem?

