THE SIMPLEX METHOD

T. Hürlimann

Working Paper No. 207

INSTITUT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG

INSTITUT FÜR INFORMATIK DER UNIVERSITÄT FREIBURG

Institute of Informatics, University of Fribourg, site Regina Mundi

rue de Faucigny 2

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

The SIMPLEX Method

Tony Hürlimann, Dr. rer. pol.

Key-words: LP, algorithm, Simplex, sparse matrixes, linear algebra

Abstract: This paper starts where most textbooks on linear programming stop: at the point where the simplex is to be implemented. The task is not easy, becauce many aspects from linear algebra, numerical analysis, optimization, and design of efficient data strucutres and algorithms are intertwined in a complex manner. Theses branches are seldom thought together, which makes the task even harder.
Nevertheless the task is a fascinating subject, since it shows many "real-live" aspects of an implementation.

This paper gives a condensed synopsis of the implementation of the Simplex method to solve linear systems. I did not found any textbook, which treats all aspects of the Simplex from the implementational point of view. In some sense, this paper starts where most textbooks in LP stop.
Linear Algebra, fundamentals in numerical analysis, optimization and efficient programming are seldomly taught together. They are all needed to implement the simplex efficiently. It is easy to write a program which implements the simplex for small, trivial problems. It tasks some hours, but it is very difficult to write a good implementation for non-trivial problems; this is much more exciting. Theoretical background is needed as well as good programming skills. This paper presents an *condense* overview of many problems encounted when implementing the simplex. Three implementations are presented: The first part exposes the normal Simplex presented in most textbooks, the second part contains the implementation of the revised Simplex (with full matrix), and the third part implements the revised Simplex using exploiding sparsity. The first part explains how to pivot directly in the tableau, whereas Part two uses the

LU-decomposition method and some *LU*-updating method to solve larger problems.

Stichworte: LP, Algorithmus, Simplex

Zusammenfassung:

1. INTRODUCTION

"Though this be madness, yet there is method in't."
Shakespeare - Hamlet

"If you can look into the seeds of time,
And say which grain will grow and which will not,
Speak then to me ..."
Shakespeare - Macbeth

"...it leads
A long, steep jourey, through sunk
gorges, o'er mountains in snow."
Matthew Arnold.

This paper presents a routine collection and some theoretical background to implement the Simplex algorithm. The paper supposes that the reader is already familiar with the simplex and the PASCAL programming language.

To implement a efficient and stable simplex on the computer is not an easy job. In fact, many

1. INEQUALITY AND STANDARD FORM

(Goal: converting to the standard form....)
The general LP problem can be formulated in the following **inequality form**

$$\begin{aligned} &\text{maximize } cx \\ &\text{subject to } Ax \le b \qquad (1) \\ &\qquad x \ge 0 \end{aligned}$$

or in the following **standard form**

$$\begin{aligned} &\text{maximize } cx \\ &\text{subject to } Ax = b \qquad (2) \\ &\qquad x \ge 0 \end{aligned}$$

where *A* is a mxn matrix, *c* is a n-row vector, *b* is a m-column vector, and *x* is a n-column vector.
The first step in formulating an LP consists in transforming the problem in the inequality or the standard form.
The transformation into the inequality form can be done using the following rules.

1. A **minimizing function** can be replaced by a maximizing function in changing the sign of *c* ("minimize *cx*" is replaced by "maximize –*cx*").

2. A **greater-equal constraint** can be replaced by a less-or-equal restriction by changing the sign ("$Ax \geq b$" is replaced by "$-Ax \leq -b$").

3. A **free** (or unrestricted) **variable** x_j can be replaced by a pair of non-negative variables $x_j' \geq 0, x_j'' \geq 0$ by substituting $x_j = x_j' - x_j''$. Another way to treat free variables is to pivot them into the basis as soon as possible and to leave them there (they must not be pivoted anymore)(freeze the corresponding row). (Note: if all variables are free and all have been pivoted into the basis, such that some elements of the cost vector c stay positive, then the problem is unbounded).

4. A **lower bound** $l_j \leq x_j$ (different from zero) on a variable x_j can be changed to a zero bound by replacing the variable x_j by a new variable $x_j' \geq 0$ where $x_j' = x_j - l_j$.
An alternative is to convert the lower bound by a added constraints. But this works only, if the lower bound is positive, otherwise a translation as descripted must take place.

5. An **upper bound** $x_j \leq u_j$ (different from infinity) on a variable x_j can be introduced explicitly as a constraint. Another way to get around the upper bounds is in making the pivoting step slidely more complex (see below).

6. **Inequality constraints** $Ax \leq b$ can be replaced by equality constraints by adding a **slack** variable vector z (replace $Ax \leq b$ by $Ax + z = b$), where z is bounded at zero ($z \geq 0$).

7. **Equality constraints** $Ax = b$ can be replaced by a pair of inequality constraints $Ax \leq b, \ -Ax \geq -b$. Another way to get rid of equality constraints is to pivot them out of the basis as soon as possible and to delete the corresponding columns. Still another way is to replace the equality constraints by inequality constraints $Ax \leq b$ and by just adding one constraint $-dx <= -\beta$ where d is a row-vector and β is a scalar. If the indices of the equality constraints are $i_1, i_2, \ldots, i_e$ (supposing we have e equality constraints), then we have $d_{i_k} = \sum_{k=1}^{e} a_{i_k j}$, and $\beta = \sum_{k=1}^{e} b_{i_k}$.(see Suhl p. 33).

Examples:

1. THE TWO STAGE SIMPLEX (THEORY)

The solution of a LP problem $\max\{cx: Ax \le b, x \ge 0\}$ consists of two distinct, sequential steps:

Step 1: solve the stage I problem: (find a feasible solution)

Step 2: solve the stage II problem: (find the optimal solution).

At each step we apply the Simplex method to solve our problem.
The simplex described below starts from a tableau in standard form

Simplex: Das so erhaltene Tableau widerspiegelt nicht unbedingt eine feasible Basis.
1. Programm mit welchem nun beliebige Handpivotierung durchgeführt werden kann.
2. Programm mit Pivotierung:

Pivot Rule for not feasible basis:
1. if $\forall(i)(b_i \ge 0)$ (all *b* are non-negative) then we have a feasible basis, GOTO 6
2. Choose any row *r* such that $b_r < 0$ and *r* is maximal (the last row with $b_r < 0$)
3. if $\forall(j)(a_{rj} \ge 0)$ (all elements in the row *r* are non-negative then STOP. Problem is infeasible in this row *r*.
4. if $r = m$ (*r* is the last row) then choose any column *c* such that $a_{mc} < 0$,
 else $(r < m)$ then choose any column *c* such that $a_{rc} < 0$, and compute $\min_{i>r}(\frac{b_r}{a_{rc}}, \frac{b_i}{a_{ic}} : a_{ic} > 0)$. Choose the row *r* which minimizes this expression.
5. Pivot on a_{rc} and GOTO 1.

6. We have now a feasible basis ($\forall(i)(b_i \ge 0)$ is the feasible condition): if $\forall(j)(c_j \le 0)$ (all c_j are non-negative) then STOP. The optimum was found.
7. Choose any column *j* such that $c_j > 0$.
8. if all $\forall(i)(a_{ij} \le 0)$ (all a_{ij} are non-positive in column *j*), then STOP: Problem is unbounded.
9. Compute $\min_{1<=i<=m}(\frac{b_i}{a_{ij}} : a_{ij} > 0)$ and choose the row *i* which minimizes this expression.
10. Pivot on a_{ij} .and GOTO 6.

```
procedure simplex
begin
  opt:='no', unbounded:='no';
  while opt='no' and unbounded='no' do
    if $c_j$<=0 for all j then opt:='yes'
    else begin
      choose any j such that $c_j$>0;
```

```
        if $a_{ij}$<=0 for all i then unbounded:='yes'
        else
          find $\min_{(i \mid a_{ij}>0)} \left[\frac{b_i}{a_{ij}}\right] = \frac{b_k}{a_{kj}}$
          pivot on $a_{kj}$
      end
    end
```

primal simplex (all $b_i > 0$, feasible condition):

or : pivot column: choose any column j such that $c_j < 0$

pivot row: choose a row i such that

$$\frac{b_i}{a_{ij}} = \min_i \left\{ \frac{b_i}{a_{ij}} \text{ such that } a_{ij} > 0 \right\}$$

dual simplex (all $c_j > 0$, feasible condition):

or : pivot row: choose any row i such that $b_i < 0$

pivot column: choose a column j such that

$$\frac{c_j}{a_{ij}} = \max_j \left\{ \frac{c_j}{a_{ij}} \mid a_{ij} < 0 \right\}$$

dual simplex: suppose the system is in canonical form and all $c_j >=0$.

step 1.: if all $b_i>=0$ then optimal, Stop

step 2 : if there exists an i such that $b_i<0$ and $a_{ij}>=0$ for all j, then the system is infeasible,.stop.

step 3 : choose any row i such that $b_i<0$

step 4: choose a column j such that

$$\frac{c_i}{a_{ij}} = \max_i \left\{ \frac{c_j}{a_{ij}} \text{ such that } a_{ij} < 0 \right\}$$

step 5 : pivot on a_{ij}. goto step 1. (the system will remain in the canonical form with all $c_j>=0$.

Suppose the basic variables at this representation are $x_{j(1)}, x_{j(2)}, \dots , x_{j(m)}$. Define the mxm matrix $B = [A^{j(1)}, A^{j(2)}, \dots , A^{j(m)}]$ and the 1xm row vector $c_B = c_{j(1)}, c_{j(2)}, \dots , c_{j(m)}$]. We present the tableau in the following foramt:

$$\begin{array}{c|c} z_o & c \\ \hline b & A \end{array}$$

After a number of pivoting the table looks is

$$\begin{array}{c|c} z_o^* & c^* \\ \hline b^* & A^* \end{array}$$

where $z_o^* = z_o - c_B B^{-1} b$, $c^* = c - c_B B^{-1} A$, $b^* = B^{-1} b$, and $A^* = B^{-1} A$

primal problem in canonical max form

max cx

subject to

$$Ax \leq b \qquad (7)$$

$$0 \leq x \leq Infinity$$

dual problem of (7) is

min by

subject to

$$A^T y \geq c \qquad (8)$$

$$0 \leq y \leq Infinity$$

To translate the dual problem (8) into a canonical form, we need the following rules:

1. replace *min* by *max* and *b* by *-b.*
2. replace $A^T y \geq b$ by $-A^T \geq -b$.

Variable upper bounds: the pivot step

The pivot steps becomes more complex, if upper bounds are not explicitly translated into constraints. In this case, we may alloy the non-basic variable to by zero *or* at its upper bound (Murtagh p.78). (Algorithm and a small example see Calvert a.i. p. 509-514).

The complexity of the Simplex

The simplex is a remarkable fast algorithm. Different statistical tests and many practical problems have shown that the Simplex takes about 2m-6m pivot steps to find the optimum, where m are the number of constraints. This is a very small number of all feasible corner points which contains $\binom{n}{m}$ corner points. The empirical tests suggest that the mean complexity of the Simplex is linear! It is still one of the most exiting open question, why the Simplex performs so good. The worst-case complexity of the Simplex, however, is much less pleasing. LP examples can be constructed, for which we can prove that the Simplex takes exponential time to solve them for any known pivot strategy. Klee (in: Chvatal p. 47) has constructed an example of this type for the largest cost rule.

$$\text{maximize } z = \sum_{j=1}^{n} 10^{n-j} x_j$$
$$\text{subject to } 2\sum_{j=1}^{i-1} 10^{i-j} x_j + x_i \le 100^{i-1} \quad \text{with } 2 \le i \le n$$
$$x_i \ge 0 \quad \text{with } 1 \le i \le n$$

For this model the Simplex could take 300,000 years to solve it, supposing we have only(!) 50 variables and we make 100 iterations per second!

DATA STRUCTURE

A matrix

$$\begin{pmatrix} 1.0 & 0 & 0 & 2.0 \\ 0 & 7.0 & 0 & 6.0 \\ 0 & 3.0 & 5.0 & 4.0 \\ 8.0 & 0 & 0 & 0 \end{pmatrix}$$

can be stored in different manner. The most simplest is to define a two-dimensional array such as

```
CONST
  MAX = ....;
TYPE
  TA = array[1..MAX,1..MAX] of real;
```

Two important disadvantages are: the space must be declared at compile time, and the maximal row as well as the maximal column numbers are limited to MAX.

add a constraint to a solution tableau:

consider following example (Thie p.153ff) (MODEL1.LPL)

max z : 11x1+4x2+x3+15x4

sobject to

3x1+x2+2x3+4x4 <= 28

8x1+2x2-x3+7x4 <= 50

x1, x2, x3, x4 >= 0

the tableau is:

```
x5    3    1    2    4    1    0    28
```

x6	8	2	-1	7	0	1	50
	-11	-4	-1	-15	0	0	0

the solution tableau is:

x4	-2	0	5	1	2	-1	6
x2	11	1	-18	0	-7	4	4
	3	0	2	0	2	1	106

so we have now the following system:

max : 3x1+2x3+2x5+x6 (=106)

-2x1+5x3+x4+2x5-x6 = 6

11x1+x2-18x3-7x5+4x6 = 4

adding now the following constraint

3x1+x2+3x4 <= 20

we must first drive the basic variable out of the restriction:

3x1+ (-11x1+18x3+7x5-4x6+4) + 3(2x1-5x3-2x5+x6+6) <= 20

giving

-2x1+3x3+x5-x6 <= -2

We now add this to the solution tableau and apply the dual simplex.

Gomory's Cutting Plane algorithm:

example: (Thie p195ff) (MODEL2.LPL)

min x1-3x2

sobject to

x1-x2 <= 2

2x1+4x2 <= 15

x1, x2 >= 0 and integral

simplex solution is

x3	3/2	0	1	1/4	23/4
x2	1/2	1	0	1/4	15/4
	5/2	0	0	3/4	45/5

step 1 : if all RHS are integral, then we have a solution, stop

step 2 : choose any row r with a non-integral RHS and add the following constraint to the system:

$\sum_j f_{rj}x_j - f_r >= 0$ and integral (add a new integral slack)

where f_{rj} is the fractional part of a_{rj} $\{a_{rj}\text{-trunc}(a_{rj})\}$ and f_r is the fractional part of b_r.

Applied to our example usinf the first constraints we add the new constraint:

$1/2x1+1/4x4-x5 = 3/4$ (where x5 is the slack)

expressed in the non-basic variables this is

$2x1+(15-2x1-4x2) >= 3$

simplified we get

$x2 <= 3$

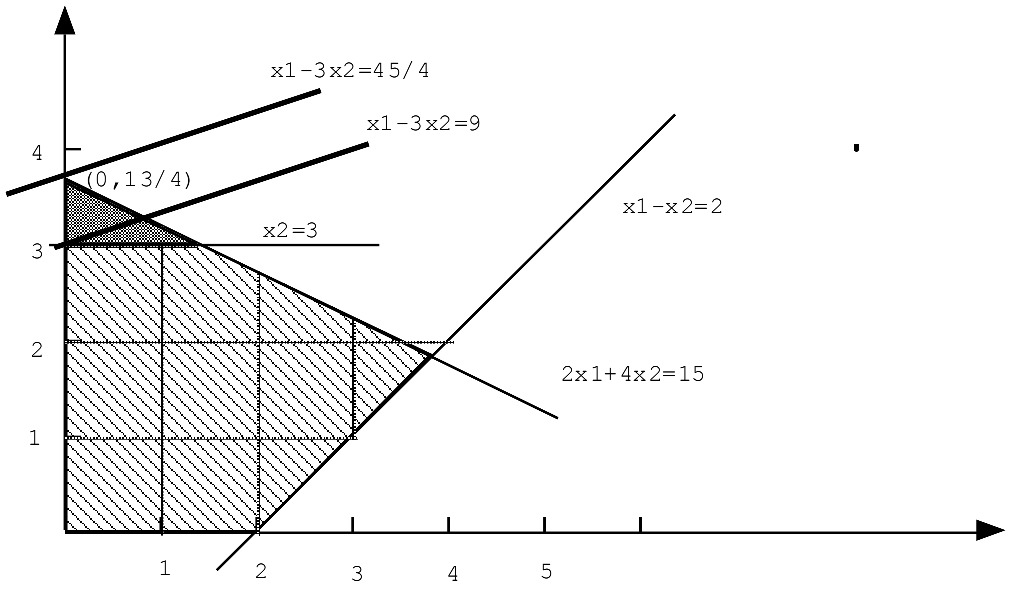

Abb 1 (Ref.: Thie p.197)

Want happens geometrically is that the new added constraints cuts the dark space out of the feasible space. Now the new optimal solution of the LP relaxation is (0,3).

Branch and Bound.

Instead of expanding the original problem as in the Cutting Plane two new problems are generated one with the new bound $x_j >= trunc(b_i)+1$, and the other with the bound $x_j <= trunc(b_i)$.

Williams p. 165:

```
TYPE BestNode, N1, N2, N3 : NODE
BestNode.obj:=-INFINITY  (if maximization)
try to find a lower bound LB : then BestNode.obj:=LB (=cut-off value)
define node N1 as the LP
solve N1
if (integer solution) then BestNode:=N1; exit;
stack N1
```

```
while stack not empty do begin
  N1:=popHeighestObj;  (critical step: how to choose the waiting node!!!!)
  create two nodes N2, N3 from N1  (critical how to choose the braching variable!!)
  solve N2
  if infeasible(N2) or obj(N2)<obj(BestNode) then delete(N2)
  if (integer solution) and obj(BestNode)<obj(N2) then BestNode:=N2
  if (fractal solution) then stack N2
  solve N3
  if infeasible(N3) or obj(N3)<obj(BestNode) then delete(N3)
  if (integer solution) and obj(BestNode)<obj(N3) then BestNode:=N3
  if (fractal solution) then stack N3
endwhile
if BestNode.obj=-INFINITY then the problem has no integer solution
```

2. LINEAR ALGEBRA

PUBLIC DOMAIN SOFTWARE IN NUMERICAL ANALYSIS:

LINPACK (linear equation systems)
EISPACK (eingenvalue problems)
QUADPACK (integration)
MINPACK (function minimization)
LAPACK++ (Schwaller Thomas)

Collected Algorithms of the ACM
in Transactions on Mathematical Software

NAG package (over 600 user-callable functions)

Professional packages:
MA** (Duff, Reid), Y12M (Zlatev, Wasniewski),

VECTOR NORMS AND MATRIX NORMS

The norm of a vector x, denoted by $\|x\|$, is used as a means of determining whether one vector is, in some sense, larger than another. A norm should have the following properties:

(i) $\|x\| > 0$ for any nonzero vector x

(ii) for any scalar $\gamma, \|\gamma x\| = |\gamma| \|x\|$

(iii) for any two vectors x and $y, \|x+y\| \le \|x\| + \|y\|$

The p-norm of a n-vector is defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

A finite limit exists as $p \to \infty$.

In practice, most common norms correspond to $p = 1, 2, \textit{ and } \infty$

We have

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$
$$\|x\|_2 = \sqrt{x'x}$$
$$\|x\|_\infty = \max_{1 \le i \le n} |x_i|$$

The norm of a matrix A is defined satisfies also the same three properties as a vector norm. Furthermore the forth property is required:

(iv) $\|AB\| = \|A\| \|B\|$

We have

$$\|A\|_1 = \max_j \left\| a_j \right\|_1$$
$$\|A\|_2 = \sigma_1(A) \quad \text{(the largest singular value)}$$
$$\|\mathrm{A}\|_\infty = \max_i \left\| a_i' \right\|_1$$

(Note: the singular values of a *mxn* matrix A are found by a singular value decomposition: Any matrix A can be written as

$$A = USV'$$

where U and V are appropriate orthogonal matrices and S is a diagonal matrix with

$$S = diag(\sigma_1, \sigma_2, \mathrm{K}\ , \sigma_p) \quad \text{where} \quad p = \min(n, m)$$

The convention is usually adopted, that σ_1 is the largest singuar value).

ILL-CONDITIONING AND INSTABILITY

Whenever a relatively small change to the parameters of a problem results in a disproportionately large change to the solution, we say that the problem is ill-conditioned. Note that ill-conditioning has nothing to do with the method chosen, it is a property of the problem itself. A striking example for a ill-conditioned problem is the Wilkinson polynom (Gill p.44). This polynom is defined as

$$W(x) = x^{20} + 210x^{19} + \ldots + 20! = (x-1)(x-2)\ldots(x-19)(x-20)$$

whose exact roots are obviously the positive integers 1,2,...,20. Now suppose we defined another polynom $W1(x)$ which is a small perturbation of $W(x)$ defined as

$$W1(x) = W(x) - 2^{-23}x^{19}$$

The exact roots (rounded to 10 digits of $W1(x)$ are

1.000000000	10.095266145 ± 0.643500904i
2.000000000	11.793633881 ± 1.652329728i
3.000000000	13.992358137 ± 2.518830070i
4.000000000	16.730737466 ± 2.812624894i
4.999999928	19.502439400 ± 1.940330347i
6.000006944	
6.999697234	
8.007267603	
8.917250249	
20.84690810	

Some real roots have changed considerablely, but ten of the roots have moved into the complex plane! Note that this ill-conditioning of the problem (small change in the original problem result in big changes in the solution) has nothing to do with the rounding errors on a computer. It is entirely specific to the problem. Using exact calulations this solution will be obtained indepentently of the method used.

The reader may assume that ill-conditioning is restricted to non-linear problems. On the contrary, we will see that ill-conditioning is a serious problem also for linear systems. An example show how ill-conditioning can arise in linear systems. Consider the matrix *A* and the vectors *b* and *b'* defined as (example from Strang p. 56)

$$A = \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0001 \end{pmatrix} \quad b = \begin{pmatrix} 2 \\ 2 \end{pmatrix} \quad b' = \begin{pmatrix} 2 \\ 2.0001 \end{pmatrix}$$

The solution vector *x* of $Ax = b$ is $x = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$, whereas the solution of $Ax = b'$ is $x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

Despite a very small change in the right hand side vector, the solution has changed considerably. The 'reason' for this unexpected behaviour is, that the matrix *A* is nearly singular. To measure the 'closeness' of ill-conditioning, a well developed concept is introduced: the condition number.

CONDITION NUMBER OF A LINEAR SYSTEM

Consider a system of linear equations

$$Ax = b$$

If a small perturbation on *A* or *b* is considered, then the solution vector *x* will change too as defined in

$$A(x + \delta x) = b + \delta b$$

$$(A + \delta A)(x + \delta x) = b$$

To measure the amount of perturbation of *x*, the condition number is introduced and defined as

$$cond(A) = \|A\|\,\|A^{-1}\|$$

The relative perturbation of x can be measured with

$$\frac{\|\delta x\|}{\|x\|} \le cond(A)\frac{\|\delta b\|}{\|b\|}$$

$$\frac{\|\delta x\|}{\|x + \delta x\|} \le cond(A)\frac{\|\delta A\|}{\|A\|}$$

We say that a matrix is ill-conditioned, if its condition number is large, and well-conditioned if it is small. The identity matrix is well-conditioned, since its condition number is 1. The closer the matrix A is to being singular the larger the value of *cond(A)* will be.

Another example for an ill-conditioned matrix *A* is (Gill p.46)

$$A = \begin{pmatrix} 0.550 & 0.423 \\ 0.484 & 0.372 \end{pmatrix}$$

Together a vector *b*, a linear system is defined with the exact solution *x*

$$b = \begin{pmatrix} 0.127 \\ 0.112 \end{pmatrix} \qquad \text{and} \qquad x = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

It is easy to see that a small perturbation on *b* will produce a large perturbation on *x*

$$b = \begin{pmatrix} 0.127 + 0.00007 \\ 0.112 + 0.00028 \end{pmatrix} \qquad \text{and} \qquad x = \begin{pmatrix} 1.7 \\ -1.91 \end{pmatrix}$$

The reason, again, for this very different solution is that *A* is almost singular. Since the inverse of *A* rounded to 6 digits is

$$A^{-1} = \begin{pmatrix} 1278.35 & -1453.61 \\ -1659.79 & 1890.03 \end{pmatrix}$$

one can easy calculate the different norms of *A*.

As a rule of thumb, $\log_{10}(cond(A))$ is approximately the number of decimal digits lost in the computed solution vector *x* in $Ax = b$ (Hopkins p.166).

A lower bound on the condition number of a upper triangular matrix *U* is given by (Gill p.152)

$$cond(U) = \frac{\max_i |u_{ii}|}{\min_i |u_{ii}|}$$

Given the *LU* factorization of a matrix *A*, a related guideline is that any ill-conditioning in *A* is likely to be reflected in *U*. (Gill p. 91).

The Hilbert matrix definition can be used to generate ill-conditioned matrices. They are defined as (Knuth 1, p.37)

$$H_{ij} = 1/_{i+j-1}$$

and have the following properties

- the inverse is $H_{ij}^{-1} = \frac{(-1)^{i+j}(i+n-1)!(j+n-1)!}{(i+j-1)(i-1)!^2(j-1)!^2(n-i)!(n-j)!}$
- the condition number increase rapidly with the order n

Relation between condition number and drop tolerance: (Zlatev in: Evans, p.213): Small numbers in a matrix can be considered as zero. How small them can be gives the following formula

$$\varepsilon \approx cond(A)^{-1}$$

FINITE PRECISION AND INSTABLILITY (ROUNDOFF ERRORS)

Floating-point numbers are stored in the computer using scientific notation on the power of two. Since only a certain fixed number of digits can be stored, the numbers cannot be stored exactly and there introduce rounding or truncation errors in calculations which may accululate in a sequence of arithmetic operations. Even seemingly exact number are sometimes not stored exact in the computers memory. An example was given by Beale 1988 p. 6

	Fraction	Exact decimal	Binary representation (8 places)	Decimal representation of binary
A	2/5	0.4	0.01100110	0.39843750
B	3/5	0.6	0.10011001	0.59765625
A+B	1	1.0	0.11111111	0.99609375

A simple example from non-linear arithmetics to calculate the following integral (HOPKINS p 10ff)

$I_n = \frac{1}{e}\int_0^1 e^x x^n dx$ using the recurrence relation $\begin{aligned} I_0 &= 1 - e^{-1} \\ I_n &= 1 - nI_{n-1} \qquad n = 1,2,\ldots \end{aligned}$

The following program coded in PASCAL can be executed to show the accumulation of the error. The error accumulation will be slower using double precision representation, but even with double precision the error of I_n makes the result meaningless.

```
program AcuError;

const MAX=20;
type TVector = array[0..MAX] of real {single, double};
var
  Result, Exact : TVector;
  i,j :integer;

begin
 Exact[0] := 0.6321205588286;
 Exact[1] := 0.3678794411714;
 Exact[2] := 0.2642411176571;
 Exact[3] := 0.2072766470287;
 Exact[4] := 0.1708934118854;
 Exact[5] := 0.1455329405731;
 Exact[6] := 0.1268023565615;
 Exact[7] := 0.1123835040693;
 Exact[8] := 0.1009319674456;
 Exact[9] := 0.9161229298966E-1;
 Exact[10]:= 0.8387707010339E-1;
 Exact[11]:= 0.7735222886266E-1;
 Exact[12]:= 0.7177325364803E-1;
 Exact[13]:= 0.6694770257562E-1;
 Exact[14]:= 0.6273216394138E-1;
 Exact[15]:= 0.5901754087930E-1;
 Exact[16]:= 0.5571934593124E-1;
 Exact[17]:= 0.5277111916900E-1;
 Exact[18]:= 0.5011985495809E-1;
 Exact[19]:= 0.4772275579621E-1;
 Exact[20]:= 0.4554488407582E-1;

 Result[0] := 0.6321205588286;
 for i:=1 to MAX do
   Result[i] := 1 - i*Result[i-1];

 for i:=0 to 20 do
   writeln(Result[i],' ',Exact[i]);

end.
```

The method (here the recurrence formula) is the source of the meaningless result. Therefore, we should use another method - for which the error to not accumulate - to solve this problem.

Finite precision is disastrous for ill-conditioned problem, but can also be disastrous for well defined problems, if the wrong method is used to solve the problem. An example drawn again from Strang p. 56 is the following linear system

$$A = \begin{pmatrix} .0001 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

The solution vector x of $Ax = b$ is $x = \begin{pmatrix} \frac{10000}{9999} \\ \frac{9998}{9999} \end{pmatrix}$. Roundoff error on three positions will produce the solution $x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. That's fine. But now suppose, we use *LU* decomposition (see below), where the first equation is $.0001x_1 + x_2 = 1$ and the second equation will be transformed to $-9999x_2 = -9998$. Backsubstitution then assign first $x_2 = 1$ (after round error). If we substitute $x_2 = 1$ into the first equation then will be $.0001x_1 + 1 = 1$ (instead of $.0001x_1 + \frac{9998}{9999} = 1$), and the solution is $x_1 = 0$ (instead of $x_1 = 1$). Note that the matrix A is well-conditioned and the origin of the error is clearly in the method to solve the problem and not in the problem itself. Another sequence of numerical manipulations will find another result.

The simplex method and related calculation as the *LU* decomposition are long sequence of (simple) calculations. To prevent the the accululation of errors one may

- equilibrate the components in the matrix
- monitor the growth of the rounding errors
- introduce error tolerances
- choose a different pivot element at a specific stage of decomposition.

Equilibration is a process which makes the largest (or the smallest) element of each row and column the same (say 1.0). First, each element in the row is divided by the largest and the largest is stored separatly, then each column is divided by the largest. This process is also called **scaling** of a matrix. More elaborated methods are available (see Hamming 1971). **Monitoring** may be done by checking the growth of

$$e = Ax - b$$

If some elements of e are not within a defined error tolerance, then this means a excessive error and the system should take some preventions (re-invert or iterative improvement).

Error tolerances: Even in a short sequence of operations, an small error may be introduced in an unexpected way. For example one may try to compare the following two numbers

```
first number:   A=0
second number   B=1-(30/58)*(58/30)
```

Depending on the compiler, this two numbers may not be the same. So the test

```
IF A=B THEN ...
```

may produce a unpredicted result. In a reliable implementation of the simplex, relational tests on reals should be replaced by tolerance functions. The equality may be replace by

```
IF IsZero(A-B) THEN ...
```

where IsZero() is a user defined function, which compares A and B within a defined tolerance:

```
function IsZero(x:real) : boolean;
```

```
begin
  IsZero := (x>-EPSILON) and (x<EPSILON);
end;
```

where EPSILON is a small number (say 1E-8). The code transparency is also enhanced using such functions. (In the programming language C, the functions should be implemented as macros to speed up the execution time).
Different tolerances values may be defined for different purposes. Murtagh (p.34) recommands the following values for a 60-bit floting-point word:

- test of the reduced cost for optimality test (1E-5)
- tolerance within a a_{ij} element is considered as zero (1E-10)
- tolerence for a pivot element (1E-8)
- tolerance for the test in $Ax - b$ (1E-6).

LINEAR EQUATIONS

Given a matrix nonsingular *A* and a vector *b*, a fundamental problem in numerical linear algebra is to find a vector *x* such that

$$Ax = b$$

If A is singular or not-quadratic, then we must find a submatrix of maximal rank to solve the problem. In the following text, we suppose that *A* is quadratic and non-singular. They are direct and iterative solution methods.

DIRECT METHODS

The simplest method is **Gaussian elimination method** which consists of eliminate one variable in turn as shown in the following sequence:

$$\begin{array}{lll} x_1 + \ \ x_2 + 2x_3 = 3 & \qquad x_1 + x_2 + 2x_3 = 3 & \qquad x_1 + x_2 + 2x_3 = 3 \\ 2x_1 + 3x_2 + \ \ x_3 = 2 & \qquad x_2 - 3x_3 = -4 & \qquad x_2 - 3x_3 = -4 \\ 3x_1 - \ \ x_2 - \ \ x_3 = 6 & \qquad -4x_2 - 7x_3 = -3 & \qquad -19x_3 = -19 \end{array}$$

x_1 is eliminated from the second and third equation by subtracting a multiple (2) of the first equations from the second and another multiple (3) from the third. Then we eliminate x_2 from the third equation by subtracting a multiple (4) of the second equation. It is now easy to solve the problem by back-substitution beginning with x_3. x_3 can be calculated using the third equation which gives $x_1 = 1$. Inserting this result into the second equation gives $x_2 - 3(1) = -4$, so that $x_2 = -1$. Inserting these results into the first equation gives $x_1 + (-1) + 2(1) = 3$, so that $x_1 = 2$.

In matrix notation, this procedure could be expressed as a sequence of pre-multiplying elementary matrices

$$\begin{pmatrix}1 & 1 & 2\\ 0 & 1 & -3\\ 0 & 0 & -19\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -4 & 1\end{pmatrix}\cdot\begin{pmatrix}1 & 1 & 2\\ 0 & 1 & -3\\ 0 & -4 & -7\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 3 & 0 & 1\end{pmatrix}\cdot\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -4 & 1\end{pmatrix}\cdot\begin{pmatrix}1 & 1 & 2\\ 2 & 3 & 1\\ 3 & -1 & -1\end{pmatrix}$$

The right hand side vector b must also be pre-multiplyed by the same elementary matrices

$$\begin{pmatrix}3\\ -4\\ -19\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -4 & 1\end{pmatrix}\cdot\begin{pmatrix}3\\ -4\\ -3\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 3 & 0 & 1\end{pmatrix}\cdot\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -4 & 1\end{pmatrix}\cdot\begin{pmatrix}3\\ 2\\ 6\end{pmatrix}$$

For this example, Gaussian elimination is nothing else than to replace

$Ax = b$ by $\quad E_2E_1Ax = E_2E_1b$

Note, that the multiplication E_2E_1 is a lower triangular matrix and E_2E_1A is an upper triangular matrix, since

$$E_2E_1=\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 3 & 4 & 1\end{pmatrix}\quad\text{and}\quad E_2E_1A=\begin{pmatrix}1 & 1 & 2\\ 0 & 1 & -3\\ 0 & 0 & -19\end{pmatrix}$$

In general, the matrix A is reduced to an upper triangular matrix by an elimination process in $k=1,2,\ldots,n-1$ stages. The matrix A and the vector b at stage k are denoted by $A^{(k)}$ and $b^{(k)}$ and we write

$$A^{(k)}=\begin{pmatrix}a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)}\\ 0 & a_{22}^{(2)} & a_{22}^{(2)} & \cdots & a_{22}^{(2)}\\ 0 & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)}\\ \cdots & \cdots & \vdots & \ddots & \vdots\\ 0 & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)}\end{pmatrix}\quad\text{and}\quad b^{(k)}=\begin{pmatrix}b_1^{(1)}\\ b_2^{(2)}\\ b_k^{(k)}\\ \ldots\\ b_n^{(k)}\end{pmatrix}$$

where the elements $a_{ik}^{(k)}, i>k$ at stage k are eliminated by calculating

$$a_{ij}^{(k+1)}=a_{ij}^{(k)}-(a_{ik}^{(k)}/a_{kk}^{(k)})\cdot a_{kj}^{(k)}\quad,\quad i,j>k.$$

or by replacing A with $E_{n-1}\ldots E_2E_1A$

The element $a_{kk}^{(k)}$ is called the **pivot** at stage k of the Gaussian elimination.

The vector b is also replaced by another vector at each stage $k=1,2,\ldots,n-1$, by calculating

$$b_i^{(k+1)}=b_i^{(k)}-(a_{ik}^{(k)}/a_{kk}^{(k)})\cdot b_k^{(k)}\quad,\quad i,j>k$$

or by replacing b with $E_{n-1}\ldots E_2E_1b$.

To sumarize, the Gaussian elimination method requrires two steps

- elimination of the variables by replaces the system $Ax=b$ with the system

  $E_{n-1}\ldots E_2E_1Ax=E_{n-1}\ldots E_2E_1b$ which is the same as $A^{(n)}x=b^{(n)}$.

- back substitution by calculating $x_k=(b_k^{(k)}-\sum_{j=k+1}^{n}a_{kj}^{(k)}x_j)/a_{kk}^{(k)}$

  for $k=n,n-1,\ldots,1$ (note that the sum is zero for k=n).

The ***LU*-decompisition** method is closly related to the Gaussian elimination method. It consists of replacing $Ax = b$ with $LUx = b$, where L is a lower-triangular and U is an upper-triangular matrix. The Gaussian method and the LU-decomposition method are related by the following folmulas

$$L = (E_{n-1} \mathrm{K}\ E_2 E_1)^{-1} = E_1^{-1} E_2^{-1} \mathrm{K}\ E_{n-1}^{-1} \quad \text{and} \qquad U = E_{n-1} \mathrm{K}\ E_2 E_1 A$$

The LU-decomposition method requires three independent steps to solve the original problem

- find the decomposition $A = LU$
- solve the linear system $Ly = b$
- solve the linear system $Ux = y$

The first step is essentially the same as the Gaussian elimination step and requires $\frac{1}{3} n^3$ flops. (note: a flop is the work associated with the computation of the form $fl(a \cdot b + c)$, one multiplication and one addition, see Gill p.39).
(Strang: p. 17 exercice: A PC can do 8'000 (double-precision) flops per second, VAX 80'000 and the CRAY X-MP/2 does 12 million. So how long takes a decomposition if n=600?)

It can be executed by pre-multiplying A by a sequence of elementary matrices as $E_{n-1} E_{n-2} \mathrm{K}\ E_2 E_1 A = U$ where n is the order of the matrix. L is get by inverting the elementary matrix sequence: $L = (E_{n-1} E_{n-2} \mathrm{K}\ E_2 E_1)^{-1}$, which is easy since L is a lower triangular matrix. Practically and from a different point of view, we obtain the elements of L and U by calculating (Duff 1986, p. 49):

$$l_{ij} = (a_{ij} - \sum_{p=1}^{j-1} l_{ip} u_{pj}) / u_{jj} \quad , i > j$$

$$u_{ij} = (a_{ij} - \sum_{p=1}^{i-1} l_{ip} u_{pj}) \quad , i \le j$$

Both matrices L and U can be calulated in one step, provided the elements are calculated in a certain order. The Doolittle algorithm calculates the elements (normally column-wise) in the sequence $l_{k1}, l_{k2}, \mathrm{K}\ , l_{k,k-1}, u_{kk}, \mathrm{K}\ , u_{kn}$. The Crout algorithm places the ones on the diagonals in the U matrix rather than in the L matrix (as the Doolittle method). And the elements are calculated (normally row-wise) using

$$l_{ij} = (a_{ij} - \sum_{p=1}^{j-1} l_{ip} u_{pj}) \ , i \ge j$$

$$u_{ij} = (a_{ij} - \sum_{p=1}^{i-1} l_{ip} u_{pj}) / l_{ii} \quad , i < j$$

The second step – called **forward substitution** – is easy and requires $\frac{1}{2} n^2$ flops. We may calculate the vector y by using

$$y_k = (b_k - \sum_{j=1}^{k-1} l_{kj} y_j) / l_{kk} \quad , k = 1, 2, \ldots, n$$

The third step – called **backward substitution** – requires also $\frac{1}{2}n^2$ flops. As already seen, x is calculated using

$$x_k = (y_k - \sum_{j=k+1}^{n} u_{kj} x_j) / u_{kk} \quad , k = n, n-1, \ldots, 1$$

It seems that to solve the linear system $Ax = b$ using the LU-decomposition is more complicated and takes longer than the simple Gaussian elimation method. Nevertheless the LU-decomposition is prefered and widely used in software packages. The reasons are manifold:

- when b changes, only steps 2 and 3 must be executed
- even if A changes slightly, the LU can be updated efficiently
- the stability can be better controlled
- the LU-decomposition of a sparse A is often also sparse

An essential ingredient in the Gaussian elimination process as well as in the LU-decomposition (also called LU-factorization) is the existence of a nonzero pivot at every stage, otherwise the procedure breaks down entirely, even if A is nonsingular. To escape this difficulty, the rows or the columns or both may be permuted such that the chosen diagonal component is different from zero. A pre-multiplication of a permuation matrix P produces a re-ordering of the rows, whereas a post-multiplication of Q produces a permutation of the columns. Therefore, we may solve the equivalent system

$$\bar{A}\bar{x} = \bar{b} \quad \text{with } \bar{A} = PAQ, \; \bar{x} = Q'x, \; \bar{b} = Pb$$

An example is

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 2 \\ 4 & 5 & 6 \end{pmatrix}$$

Exchanging the first and the second row will produce the matrix $\bar{A}$ and is eqivalent to multipy A by the following permutation matrices P and Q

$$\bar{A} = PAQ \quad \text{where } P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } Q = I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

In general, the permutation cannot be fixed at the beginning of the decomposition procedure, but must be considered at each stage of the process, and one should decide to permute two rows or columns. This produces the sequence

$$E_{n-1}P_{n-1}E_{n-2}P_{n-2} \ldots E_2P_2E_1P_1A = U$$

Unfortunately, avoiding zero pivots does not protect against disaster in the decomposition procedure. One should also avoid *small* pivot elements. This is so, because small pivot elements in stage *k* may produce large numbers in the $A^{(k+1)}$ matrix. An overflow is likely to occur in some subsequent stages. Different pivoting strategies have been proposed to avoid small pivots (to guard against numerical instability). To review them, let us first define the active matrix: The quadratic submatrix of $A^{(k)}$ containing the last *n-k* rows and columns is called **the active matrix**. We call it $AA^{(k)}$.

Complete pivoting is the strategy to choose the largest element in $AA^{(k)}$ and to exchange two rows and two columns accordingly to bring the largest pivot at position *(k,k)* at every stage *k*. Formally formulated: find the largest entry $a_{rc}^{(k)}$ in the active matrix $AA^{(k)}$such that $a_{rc}^{(k)} = \max_{ij}\left|a_{ij}^{(k)}\right|$.

Partial pivoting restricts the choose of the largest element in the active row or the active column *k*. At stage *k* of the elimination process, let γ_k denote the maximum magnitude of any subdiagonal (right-diagonal) component, then we choose the pivot row (column) of the largest magnitude $\gamma_k = \max_{i \geq k} \left|a_{ik}^{k}\right|$

Formally formulated: find the largest column (row) entry $a_{kc}^{(k)}$ ($a_{rk}^{(k)}$) in the active matrix $AA^{(k)}$ such that $a_{kc}^{(k)} = \max_j \left|a_{kj}^{(k)}\right|$ ($a_{rk}^{(k)} = \max_i \left|a_{ik}^{(k)}\right|$).

Threshold pivoting: an interchange is performed at step *k* only if the diagonal pivot element is much smaller than the largest subdiagonal element γ_k . Formally, row *k* and l are interchanged only if

$$\left|a_{kk}^{(k)}\right| < \tau\gamma_k, \quad \text{with} \quad 0 < \tau < 1$$

τ is called the threshold tolerence. Typically values of τ are $1/10$ (Gill p.90), Duff [1986] p.98 (MA28), 0.01 in LP-code (see Suhl).

(In other words: find a column entry $a_{kc}^{(k)}$ in the active matrix $AA^{(k)}$ such that $a_{kc}^{(k)} \geq \tau \max_j \left|a_{kj}^{(k)}\right|$ with $0 \leq \tau \leq 1$).

For sparse systems, another concern is **to minimize the fill-ins**, that is to prevent zero elements to becomming non-zeroe s. An deterministic algorithm to find a decomposition with the minimum fill-in seems difficult, because the problem is NP-complete. But there a sereral heuristics (strategies). For sparsity consideration, the pivotal strategies can be divided into three groups (Zlatev in: Evans, p.188, see there also tables comparing the run-time and fill-ins, see also Zlatev 1991 p. 71ff):

- Strategies with priori interchanges. (Permute the matrix at increasing number of non-zeroes on rows/columns before decomposition begins, and at each stage chosen the row with the minimal row count that satisfies the threshold condition as pivot row).
- Strategies using the Markowitz cost function (see below).

- Strategies based on local minimization of the number of fill-ins.

The most promising strategies seems to lay on the Markowitz cost function. If the entry $a_{rc}^{(k)}$ is chosen as pivot element, then $(lenr_r^{(k)}-1)(lenc_c^{(k)}-1)$ is the number of fill-ins at stage *k*, where $lenr_r^{(k)}$ and $lenc_c^{(k)}$ are the number of non-zeroes in row *r* and column *c* of the matrix $A^{(k)}$ at stage *k*. This expression is called the Markowitz count. The recommended chosing as pivot is the nonzero entry in the active matrix $AA^{(k)}$ which minimizes the Markowitz count. Stability and minimizing the fill-ins are often conflicting goals. Therefore, the final strategy in chosing a pivot element is a compromise, which might be formulated as following: *At each stage k, search through the nonzero entries in the active matrix (or a part of it) in order of increasing Markowitz count until one is found satisfying the threshold condition.* Since low Markowitz count are likely to come early in ascending row and column count, one need only to search in few rows and columns. Numerous varieties of this strategy might be formulated.

UPDATING (GILL P.125FF)

In many optimization algorithms, as in the simplex, we need to solve a sequence of linear systems in which each successive matrix is closely related to the previous matrix.
Optimization algorithms:
solve $A_1x=b,\ A_2x=b, \ldots,\ A_nx=b$ in this sequence. If A_{i-1}, A_i for $i=1,2,\ldots,n$ are two similar matrices, it might be more efficient to <u>update</u> the *LU* decomposition somehow ($L_{i-1}U_{i-1}$ to L_iU_i for $i=1,2,\ldots,n$) rather than to re-factor each matrix A_i from scratch. Especially in the simplex, where the subsequent matrices *A* differ only in a column, the updating might be more efficient.
The update could be described as
1. Replace the corresponding column *c* within the upper-triangular matrix *U*. The new matrix $\overline{U}$ is, in general, no longer upper-triangular (Fig 1a).
2. Shuffle the column c to the right by exchanging each column j with j+1 for j>=c. This produces a upper Hessenberg matrix *H*, which is a upper-triangular matrix except for the nonzero subdiagonal elements in columns c through n-1.

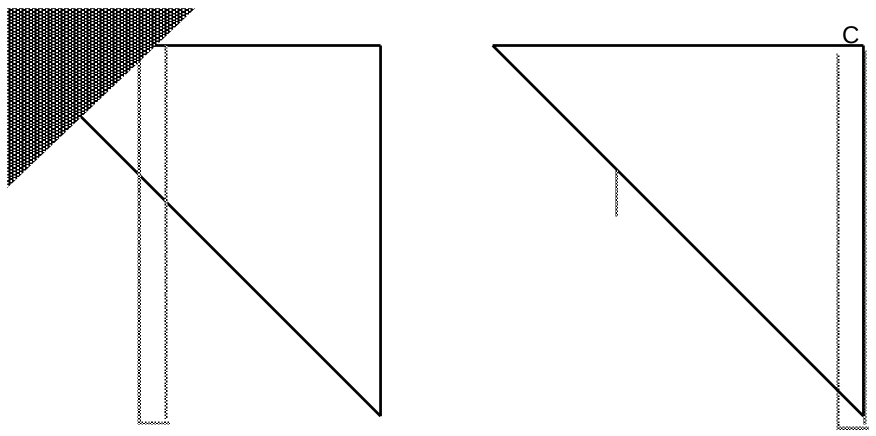

Fig 1a+b

3. Place H in upper-triangular form. Different methods have been proposed for doing this. The Bartels-Golup method (Bartels 1969) eliminates the subdiagonal nonzeros in order from column c to column n-1. At each column j=c,c+1,...,n-1, the elements at position (j,j) and (j+1,j) (the diagonal element and the subdiagonal element) are compared. If the subdiagonal element has a greater magnitude, then row j+1 and j exchanged. Finally, we pivot at the position (j,j) (=pre-multiply by an simple elementary matrix containing only one nonzero element beneath the diagonal in position (j+1,j) (Fig 1c). The main advantage of this method id that it is numerically stable.

The Forrest-Tolmin method (Forrest 1972) proposed to move row c to the bottom and any row >c one position up. Consequently, the first n-1 rows are placed in upper-triangular form (Fig 1d). Although this method does not ensure numerical stability, it is attractive for preserving sparsity. The Bartels-Golup method, on the other side, can cause fill-ins, in rows c through n.

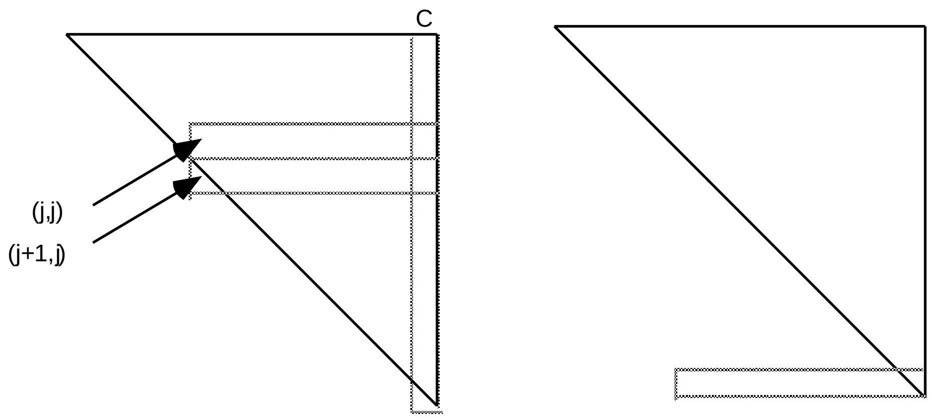

Fig 1c+d.

Reid (Reid 1982) has proposed another method:

Saunders (Saunders 1975) proposed an alternative to Reid's method, which allows L and most of U to reside on disk.

Gay

..... selective annihilation, stabilized elementary transformations, sweeps, updating LU p.146......

The **QR-Decomposition method**: A=QR, where Q is orthogonal ($QQ' = I$) and R is upper-triangular. stable, used in interior-point algorithm, for symetric matrices. More fill-ins but stable.

The **Gauss-Jordan elimination method** is similar to the Gaussian method, but it avoids the back-substitution step. Each variable is eliminated from every equation except one. Taken the numerical example above, one would process by eliminate x_1 from all equation except the first (like in Gaussian elimination. The second variable x_2 will be eliminated from all equations except from the second one, and x_3 is eliminated from all equations except the third one. This gives the sequence:

$$\begin{array}{llll}
x_1 + x_2 + 2x_3 = 3 & x_1 + x_2 + 2x_3 = 3 & x_1 \quad - 5x_3 = -7 & x_1 \quad = 2 \\
2x_1 + 3x_2 + x_3 = 2 & \quad x_2 - 3x_3 = -4 & \quad x_2 - 3x_3 = -4 & \quad x_2 \quad = -1 \\
3x_1 - x_2 - x_3 = 6 & -4x_2 - 7x_3 = -3 & \quad -19x_3 = -19 & \quad x_3 = 1
\end{array}$$

This method is the same as to pre-multiply the augmented matrix $(A \mid b)$ by a sequence of elementary matrices with a full column. Unfortunately, this method involves more work than the *LU*-decomposition.

The **inverse method**. From a mathematical point of view, the simplest method to solve the linear system $Ax = b$ might be to build the inverse A^{-1} and multiply it by the *b* vector, since $x = A^{-1}b$. Unfortunately, this method cannot be recommended. It needs 3 times more flops than the *LU*-decomposition. Another reason is that the inverse of *A* tends to be dense, even if *A* is sparse, and it is, in gereral, more difficult to control the stability. The inverse should ***never*** be calculated explicitly. Even in a matrix product $C = A^{-1}S$, when *A*, *C*, and *S* are given, where we may be tempted to form the inverse, we do not need to calculate the inverse explicitly. An alternative may be to form the *LU*-factorization of A , and then solving $LUc_j = s_j$ for each vector column *j*. (Gill p.104).

ITERATIVE METHODS

There are other methods to solve $Ax = b$. Expressing A as a sum of two matrices $A = M - N$, and $A = D - L - U$ such that D is a diagonal matrix, U is a upper triangular, and L is a lower triangular matrix, we can define a stationary iterative scheme in the form (Evans p. 75)

$$Mx^{(k+1)} = Nx^{(k)} + b$$

where $x^{(k)}$ is the k-th aproximation to the solution vector. The **Jacobi method** uses the partitioning $M = D$, $N = L + U$. The **Gauss-Seidel method** uses the partitioning $M = D - L$, $N = U$. The **SOR method** (Successive Over-relaxation) method uses $M = \omega^{-1}(D - \omega L)$, $N = \omega^{-1}((1 - \omega)D + \omega U)$ as partitioning where ω is a scalar in the range $0 < \omega < 2$. It is easy to see, that the SOR method becomes the Gauss-Seidel method when $\omega = 1$. In practice, the most difficult part is to estimate the optimal relaxation factor ω. (more information gives Schendel Chap 6.).

Iterative methods may be used together with direct method, if the accuracy is a problem. For very big problems, the following method is often used

1 Decompose A using the LU-factorization, but drop any fill-ins that are smaller than a defined tolerance.
2 Approach the 'true' LU by a iterative method. (see also Zlatev 1991).

For the simplex we do not use iterative method, but use LU-decomposition.

beyond nonsingularity.

GRAPH THEORY

Each quadratic matrix which has a traversal can be associated with a digraph (directed graph), where every row (column) is a vertex and every non-zero entry is a edge. Each m×n matrix without a traversal can be associated with a bipartite graph, where the rows and the columns form two disjoint vertex set and every non-zero matrix entry is an edge. Finally, every symmetric matrix can be associated with a undirected graph.

- The graphs are invariant under symmetric permutations (only the vertex labeling change).
- the graph exposes the underlying structure.

Reduction to lower-triangular block form

1. find a full traversal: (=maximum assignment or matching of the bigraph). Goal: find a permutation of rows and columns such that the diagonal is filled with non-zeroes: The algorithm of Hall:

```
for k:=1 to n do
    1. if A[k,k] = 0 then
        if any A[i,j] <> 0 such that i>k and j>k then
            interchange row i and k, and interchange column j and k
        else find a path with an odd length with alternate columns and rows
            $l_1, l_2, K, l_i, K, l_r$ beginning with a column
            such that (see Fig 1)
            $l_1 = k$, $l_r > k$, and $a_{l_i l_{i+1}} \neq 0$ for $i = 2, 3, K, r-1$
            now interchange each row $l_i$ with row $l_{i-2}$
             for $i = r-1, r-2, K, 4$
            and interchange column $l_r$ with column k.
            (if not such path exist, stop: matrix is singular).
```

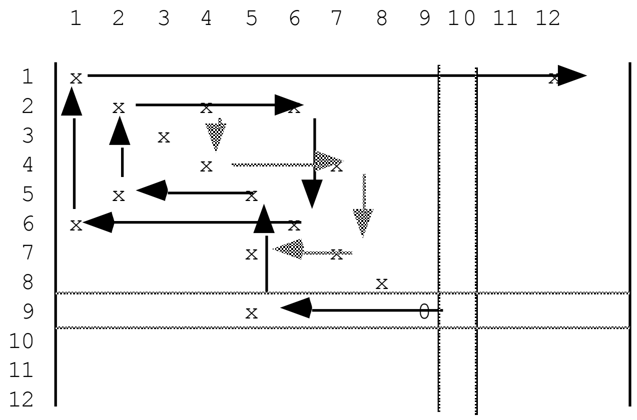

Fig 1

The algorithm of Hopcroft and Karp: Maximum matching on the bigraph $G = (V, E)$. A subset of the edges $M \subseteq E$ is a matching (or assignment) if no vertex is incident with more than one edge. A maximum matching is a matching with maximum cardinality. A vertex is free if it is incident with no edge in *M*. A path without repeated vertices is an augmenting path relative to the matching M if its endpoints are both free vertices and its edges are alternatively in *E-M* and in *M*. If an augenting path can be found, then another matching *M'* with cardinality $|M| + 1$ can be obtained by taking all edges of the path which are not in the original *M*. (Fig 2a). The initial matching with the edges {(1,5),(3,8)} is given by the fat lines in Fig 2a. A augmenting path can by constructed containing the edges {(6,1),(1,5),(5,3),(3,8),(8,4)} with the endpoints 6 and 4 (Fig 2b). This produces a new matching {(1,6),(3,5),(4,8)} in Fig 2c. Another augmenting path is shown in 2d, such that we obtain a maximum matching in Fig 2e.

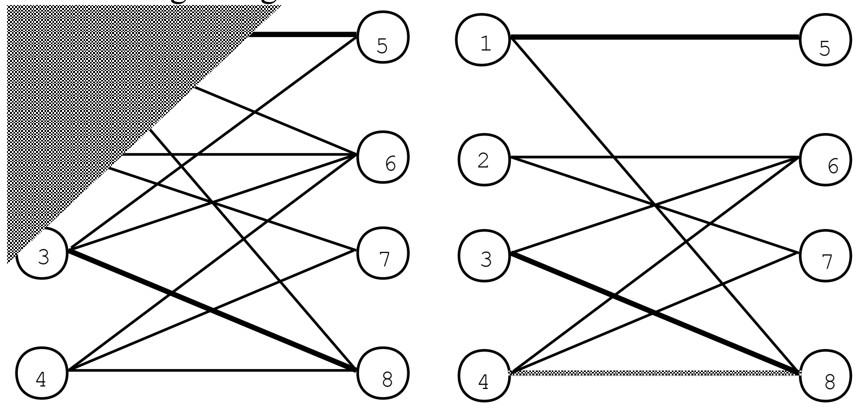

Fig 2a+b

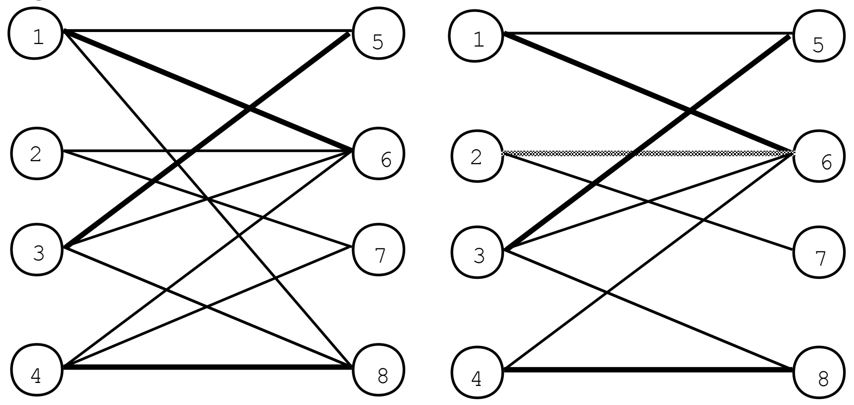

Fig 2c+d

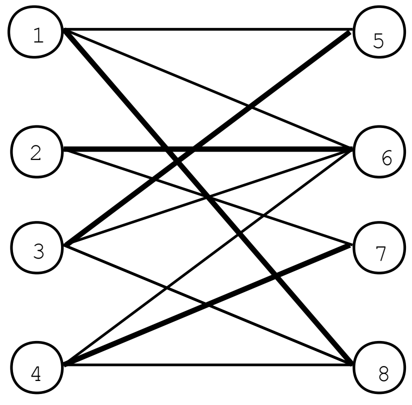

Fig 2e

The algorithm of Hall has a higher asymptotic bound, but has been found to perform better than the algorithm of Hopcroft and Karp on many practical matrices.

2. finding the lower triangular block structure: (=finding the strong components

in a digraph): the algorithm of Tarjan (see Pissanetsky p.189).

1. DATA STRUCTURES

A matrix

$$\begin{pmatrix} 1.0 & 0 & 0 & 2.0 \\ 0 & 7.0 & 0 & 6.0 \\ 0 & 3.0 & 5.0 & 4.0 \\ 8.0 & 0 & 0 & 0 \end{pmatrix}$$

can be stored in different manner. The most simplest is to define a two-dimensional array such as

```
CONST
  MAX = ....;
TYPE
  TA = array[1..MAX,1..MAX] of real;
```

Two important disadvantages are: the space must be declared at compile time, and the maximal row as well as the maximal column numbers are limited to MAX.

unsorted sparse row-wise format (USR)

A	1.0	2.0	6.0	7.0	3.0	4.0	5.0	8.0
IA	1	4	4	2	2	4	3	1
IPA	1	3	5	8	9			

The entries in row i are A[k] with IPA[i] <= k <= IPA[i+1]-1

sorted sparse row-wise format (SSR)

A	1.0	2.0	7.0	6.0	3.0	5.0	4.0	8.0
IA	1	4	2	4	2	3	4	1
IPA	1	3	5	8	9			

sorted sparse column-wise format (SSC)

A	1.0	8.0	7.0	3.0	5.0	2.0	6.0	4.0
IA	1	4	2	3	3	1	2	3
IPA	1	3	5	6	9			

If the rows and columns must be accessed quickly then the Gustavson's format may be the right choice:

unsorted row-wise Gustavson's format (URG) (see Duff 1986, p.32)

A	1.0	2.0	6.0	7.0	3.0	4.0	5.0	8.0
IA	1	4	4	2	2	4	3	1
IPA	1	3	5	8	9			
JA	1	4	3	2	3	2	3	1
JPA	1	3	5	6	9			

sorted row-wise Gustavson's format (URG)

A	1.0	2.0	7.0	6.0	3.0	5.0	4.0	8.0
IA	1	4	2	4	2	3	4	1
IPA	1	3	5	8	9			
JA	1	4	2	3	3	1	2	3
JPA	1	3	5	6	9			

linked list representation: Evans p. 6,

REFERENCES

On LP and Simplex

BEALE E.M.L., [1988], Introduction to Optimization, John Wiley & Sons, Chichester.

BREARLEY A.L., MITRA G., WILLIAMS H.P., [1975], Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm, in: Mathematical Programming 8 (1975), p. 54-83.

CALVERT J.E., VOXMAN W.L., [1989], Linear Programming, Harcourt Brace Jovanovich, Publishers, Academic Press, San Diego.

CHVATAL V. [1983], Linear Programming, W.H. Freeman and Comp, New York.

KARLOFF H., [1991], Linear Programming, Birkhäuser, Boston.

MURTAGH B.A., [1981], Advanced Linear Programming: Computation and Practice, McGraw-Hill International Book Company, New York.

HO J.K., SUNDARRAJ R.P., [1989], DECOMP: an Implementation of Dantzig-Wolfe Decomposition for Linear Programming, Lecture Notes in Economics and Mathematical Systems, Vol 338, Springer-Verlag, Berlin.

STRAYER J.K., [1989], Linear Programming and Its Applications, Springer, NY.

SUHL U., [1975], Entwicklung von Algorithmen für ganzzahlige Optimierungsmodelle, Beiträge zur Unternehmensforschung, Institut für Unternehmensführung, Heft 6, Freie Universität Berlin, Berlin.

linear algebra

GILL P.E., MURRAY W., WRIGHT M.H., [1991], Numerical Linear Algebra and Optimization, Vol 1, Addison-Wesley, New York.

HOPKINS T., PHILLIPS C., [1988], Numerical Methods in Practice, Using the NAG Library, Addison-Wesley, New York.

STRANG G., [1988], Linear Albegra and its Applications, Third Edition, Harcourt Brace Jovanovich, Publ. San Diego.

On Sparse Matrices:

BUNCH J.R., ROSE D.J., [1975], Sparse Matrix Computations, Akademic Press, NewYork.

COLEMAN T.F., [1984], Large Sparse Numerical Optimization, Lecture Notes in Computer Science, Vol 165, Springer-Verlag, Berlin.

DUFF I.S., REID J.K., [1974], A Comparison of Sparsity Orderings for Obtaining a Pivotal Sequence in Gaussian Elimination, Journal of the Institute of Mathematics and its Applications, Vol 14, No: 3, p. 281-291.

DUFF I.S., STEWART G.W. (eds), [1978], Sparse Matrix Proceedings 1978, SIAM, Philadelphia.

DUFF I.S., [1981], Sparse Matrices and their Uses, Academic Press, London.

DUFF I.S., ERISMAN A.M., REID J.K., [1986], Direct Methods for Sparse Matrices, Clarendon Press, Oxford.

EVANS D.J. (ed), [1985], Sparsity and its Applications, Cambridge University Press. London.

GUSTAVSON F., [1981], A Survey of some Sparse Matrix Theory and Techniques, in: Jahrbuch Überbicke Mathematik 1981, Wissenschaftsverlag, Bibliographisches Institut, Mannheim.

HAMMING R.W., [1971], Introduction to Applied Numerical Analysis, McGraw-Hill, New York.

ØSTERBY O, ZLATEV Z., [1983], Direct Methods for Sparse Matrices, Lecture Notes in Computer Science, Vol 157, Springer-Verlag, Berlin.

PISSANETSKY S., [1984], Sparse Matrix Technology, Academic Press, London.

SCHENDEL U., [1989], Sparse Matrices, Ellis Howwood Lim. Publ., Chichester.

ZLATEV Z., WASNIEWSKI J., SCHAUMBURG K., [1981], Y12M, Lecture Notes in Computer Science, Vol 121, Springer-Verlag, Berlin.

ZLATEV Z., [1991], Computational Methods for General Sparse Matrices, Kluwer Acad. Publ. Dordrecht.

On Updating Bases

BARTELS R.H., GOLUP G.H., [1969], The Simplex Method of Linear Programming Using LU Decomposition, Communication of the ACM, 12, p.266-268.

FORREST J.J.H., TOLMIN J.A., [1972], Updating Triangular Factors of the Basis to Maintain Sparsity in the Product Form Simplex Method, Mathematical Programming, 2, p.263-278.

REID J.K., [1982], A Sparsity-Expoiting Variant of the Bartels-Golup Decomposition for Linear Programming Bases, Math. Programming 24(1982), p.55-69.

SAUNDERS M.A., [1975], A Fast, Stable Implementation of the Simpley Method Using Bartels-Golub Updating, in: Bunch J.R., p.213-226.