GLPK

GNU Linear Programming Kit
Version 1.1
Library Reference
December 2000

Andrew Makhorin

The GLPK package is a part of the GNU project, released under the aegis of GNU.

Copyright © 2000 Andrew Makhorin, Department for Applied Informatics, Moscow Aviation Institute, Moscow, Russia. All rights reserved.

Free Software Foundation, Inc., 59 Temple Place — Suite 330, Boston, MA 02111, USA.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1 Introduction 1

2 GLPAVL: AVL Search Trees 3

2.1 Prelude 3
2.2 `create_avl` — create AVL tree 5
2.3 `insert_by_key` — insert new node with given key 6
2.4 `next_node` — search for the next node 7
2.5 `prev_node` — search for the previous node 10
2.6 `find_by_key` — search for the first node with given key . . . 11
2.7 `next_by_key` — search for the next node with the same key 12
2.8 `insert_by_pos` — insert new node to given position 13
2.9 `find_by_pos` — search for node placed in given position . . 14
2.10 `delete_node` — delete given node 14
2.11 `rotate_avl` — restore balance of AVL subtree 16
2.12 `delete_avl` — delete AVL tree 18
References 19

3 GLPDUFF: Duff Search Technique 21

3.1 Prelude 21
3.2 `create_duff` — create Duff scheme 22
3.3 `include_obj` — include object into scheme 22
3.4 `exclude_obj` — exclude object from scheme 22
3.5 Determining all objects with given length 23
3.6 Determining length of given object 23
3.7 Changing length of given object 23
3.8 `reset_duff` — remove all objects from scheme 23
3.9 `delete_duff` — delete Duff scheme 23
References 24

4 GLPEFI: Elimination Form of the Inverse (EFI) 25

4.1 Prelude 25
4.2 `create_efi` — create EFI 27
4.3 `build_efi` — build EFI for given basis matrix 27
4.4 `update_efi` — update EFI for adjacent basis matrix 27
4.5 `efi_ftran` — perform forward transformation (FTRAN) . . 28
4.6 `efi_btran` — perform backward transformation (BTRAN) 29
4.7 `delete_efi` — delete EFI 29

5 GLPETA: Implementation of Eta File 31

5.1 Prelude . . . 31
5.2 `create_eta` — create eta file . . . 32
5.3 `app_term` — append new eta term . . . 32
5.4 `h_solve` — solve system $Hx = b$ . . . 33
5.5 `ht_solve` — solve system $H'x = b$ . . . 34
5.6 `reset_eta` — reset eta file . . . 34
5.7 `delete_eta` — delete eta file . . . 35

6 GLPGEL: Sparse Gaussian Elimination 37

6.1 Unsymmetric Gaussian Elimination . . . 37
6.1.1 `gel` — transform sparse matrix to upper triangular form using gaussian elimination . . . 37
6.1.2 `gel_bg` — transform sparse matrix of special structure to upper triangular form using Bartels and Golub technique . . . 43
6.1.3 `gel_ft` — transform sparse matrix of special structure to upper triangular form using Forrest and Tomlin technique . . . 47
6.1.4 `min_bump` — minimize bump of sparse matrix using Reid technique . . . 52
6.2 LU Factorization . . . 55
6.2.1 `create_lu` — create LU factorization . . . 56
6.2.2 `build_lu` — compute LU factorization of given sparse matrix . . . 56
6.2.3 `monit_lu` — standard monitoring routine . . . 61
6.2.4 `solve_lu` — solve linear system using LU factorization . . . 61
6.2.5 `delete_lu` — delete LU factorization . . . 62
References . . . 62

7 GLPMAT: Operations on Vectors and Matrices 63

7.1 Prelude . . . 63
7.2 `check_mat` — check sparse matrix for correctness . . . 66
7.3 `check_per` — check permutation matrix for correctness . . . 67
7.4 `clear_line` — clear row or column of sparse matrix . . . 67
7.5 `clear_mat` — clear sparse matrix ($A := 0$) . . . 68
7.6 `copy_mat` — copy sparse matrix ($B := A$) . . . 68
7.7 `copy_per` — copy permutation matrix ($Q := P$) . . . 69
7.8 `count_nz` — count non-zeros of sparse matrix . . . 69
7.9 `create_mat` — create sparse matrix . . . 70
7.10 `create_per` — create permutation matrix . . . 70
7.11 `delete_mat` — delete sparse matrix . . . 70
7.12 `delete_per` — delete permutation matrix . . . 71
7.13 `eq_scaling` — implicit equilibration scaling . . . 71
7.14 `gm_scaling` — implicit geometric mean scaling . . . 72

7.15 `inv_per` — invert permutation matrix ($P := P'$) 73
7.16 `iper_vec` — permute vector elements in reverse order ($y := P'x$) 73
7.17 `l_solve` — solve lower triangular system $Lx = b$ 74
7.18 `lt_solve` — solve transposed lower triangular system $L'x = b$ 74
7.19 `mat_per` — permute matrix columns ($A := AP$) 75
7.20 `mat_vec` — multiply sparse matrix on vector ($y := Ax$) . . 76
7.21 `new_elem` — create new element of sparse matrix 76
7.22 `per_mat` — permute matrix rows ($A := PA$) 77
7.23 `per_vec` — permute vector elements ($y := Px$) 77
7.24 `reset_per` — reset permutation matrix ($P := I$) 78
7.25 `scrape_mat` — remove tiny elements from sparse matrix . . 78
7.26 `sort_mat` — sort row and column lists of sparse matrix . . 79
7.27 `submatrix` — copy submatrix of sparse matrix 79
7.28 `test_mat_d` — create test sparse matrix of $D(n,c)$ class . . 80
7.29 `test_mat_e` — create test sparse matrix of $E(n,c)$ class . . 80
7.30 `tmat_vec` — multiply transposed sparse matrix on vector ($y := A'x$) 81
7.31 `trn_mat` — transpose sparse matrix ($A := A'$) 82
7.32 `u_solve` — solve upper triangular system $Ux = b$ 82
7.33 `ut_solve` — solve transposed upper triangular system $U'x = b$ 83
7.34 `v_solve` — solve implicit upper triangular system $Vx = b$ 84
7.35 `vt_solve` — solve implicit transposed upper triangular system $V'x = b$ 85
References 85

8 GLPMPS: Processing LP Problems Using MPS Format 87

8.1 Linear programming (LP) problem 87
8.2 MPS Format Specification 88
8.3 MPS data structures 95
8.4 `load_mps` — load MPS file 97
8.5 `free_mps` — free MPS file 98

9 GLPRFI: Reid Form of the Inverse (RFI) . . 99

9.1 Prelude 99
9.2 `create_rfi` — create RFI 102
9.3 `build_rfi` — build RFI for given basis matrix 102
9.4 `update_rfi` — update RFI for adjacent basis matrix 103
9.5 `rfi_ftran` — perform forward transformation (FTRAN) 104
9.6 `rfi_btran` — perform backward transformation (BTRAN) 104
9.7 `delete_rfi` — delete RFI 105
References 105

10 GLPSET: Auxiliary Routines 107

10.1 Communicative Routines 107
10.1.1 `print` — print informative message 107
10.1.2 `error` — print error message and continue program execution 107
10.1.3 `fault` — print error message and terminate program execution 108
10.2 Memory Management Routines 108
10.2.1 `umalloc` — allocate memory block 109
10.2.2 `ucalloc` — allocate memory block 109
10.2.3 `ufree` — free memory block 110
10.3 Memory Pools 110
10.3.1 `create_pool` — create memory pool 111
10.3.2 `get_atom` — allocate atom of fixed size 112
10.3.3 `free_atom` — free atom 112
10.3.4 `get_atomv` — allocate atom of variable size 112
10.3.5 `clear_pool` — free all atoms 113
10.3.6 `delete_pool` — delete memory pool 113

11 GLPSTR: Operations on Charater Strings 115

11.1 Ordinary Character Strings 115
11.1.1 `str2int` — convert character string to value of integer type 115
11.1.2 `str2dbl` — convert character string to value of double type 115
11.1.3 `strspx` — remove all spaces from character string 116
11.1.4 `strtrim` — remove trailing spaces from character string 116
11.2 Segmented Character Strings 116
11.2.1 `clear_str` — clear segmented character string 117
11.2.2 `compare_str` — compare segmented character strings 118
11.2.3 `create_str` — create segmented character string 118
11.2.4 `create_str_pool` — create pool for segmented character strings 119
11.2.5 `delete_str` — delete segmented character string 119
11.2.6 `get_str` — extract value of segmented character string 119
11.2.7 `set_str` — assign value to segmented character string 120

Index 121

1 Introduction

GLPK (GNU Linear Programming Kit) is a set of routines written in the ANSI C programming language and organized in the form of a library. This package is intended for solving large scale linear programming and other related problems.

This document describes GLPK routines, which implement some important algorithms and data structures (for example, operations on sparse matrices) and which may be used independently.

For the sake of convenience all GLPK routines are splitted into partitions. Each partition has a name, which is identical to the corresponding header file in the library. For example, declarations of data structures and routines related to operations on sparse matrices are contained in the header file `glpmat.h`, therefore the name of that partition is GLPMAT.

Some GLPK routines (for example, routines that implement AVL search trees) are not related directly to solving LP problems. However, those routines play important role on preparing input data in case of large scale problems.

Descriptions of some routines include *time complexity*. This characteristic shows how the time of an operation depends on the size of data. For example, the `find_by_key` routine, which searches for a node with the given key in AVL tree, has time complexity $O(\log n)$, where n is current number of nodes in the AVL tree. This means that if this operation takes t time units in case of n nodes, it will take $10t$ time units in case of $1000n$ nodes, i.e. increasing number of nodes in 1000 times involves increasing the time of operation approx. in 10 times, because $\log 1000 \approx 10$ (the logarithm base is not essential). Time complexity $O(1)$ means that an operation takes the constant time, i.e. the time of operation doesn't depend on the size of data. Knowing time complexity of GLPK routines the user can estimate this characteristic of its own routines based on GLPK routines.

2 GLPAVL: AVL Search Trees

2.1 Prelude

Adelson-Velsky and Landis search tree (AVL tree) is a well known data structure, which is intended mainly for organizing symbol tables[1].

The user can consider AVL tree just as an ordered set of *nodes* (or table entries), where each node has a *key*, i.e. some value, which identifies the node. Data type, which is used for representing keys of a particular AVL tree, should be supplied by the user and may be of arbitrary nature. However, all keys should be comparable on "less than", "equal to", and "greater than".

AVL trees allow performing main operations (inserting a new node, searching for a node, and deleting a node) for logarithmic time and, thus, combine advantages of linked lists and binary search.

AVL trees are based on binary trees. *Binary tree* is a recursive data structure, where one node is a *root*, and other nodes are divided into two parts: the *left subtree* and the *right subtree*, thereto each subtree also have the same structure of binary tree. The ordering of nodes of a binary tree is defined in the following way: if some node has the key K, then all nodes in the left subtree have keys that are less than K, and all nodes in the right subtree have keys that are greater than K. Example of a binary tree is shown on Fig. 2.1.

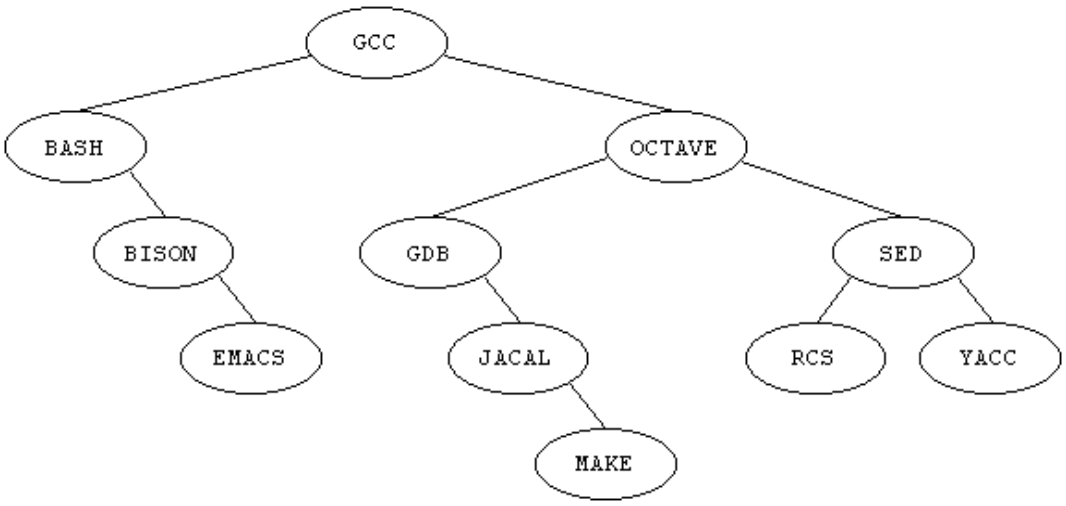

Fig. 2.1. Example of a binary tree

The ordering nodes by keys allows efficient searching for a node with the given key K. In order to find that node we should start from the root of a binary tree and go to the left if K is less than the key of the current node or to the right if K is greater than the key of

[1] Another approach for organizing symbol tables is based on *hash tables*.

the current node. The same way may be used in order to find the point for inserting a new node.

The main characteristic of a binary tree is its *height*, which is a maximal number of nodes in a path from the root to any other node. It is understood that the height of a binary tree is a crucial parameter, since it determines the number of elementary steps needed for performing main operations on a binary tree. If there are no additional limitations on the structure of a binary tree, its height may change from $\log_2(N+1)$ (ideal case) to N (degenerative case), where N is total number of nodes. Therefore if a binary tree approaches to the degenerative case, all main operations become inefficient as in the case of ordinary linear linked lists.

In order to avoid the degeneracy of a binary tree Adelson-Velsky and Landis [1] proposed the following limitation on its structure: the difference between heights of the left and the right subtrees for each node of a binary tree should be not greater than one. The binary tree, which is satisfied to such limitation, is called *AVL tree*. (Note that the binary tree shown on fig. 2.1 is not AVL tree.) In [1] it also was shown that the height of any AVL tree can't exceed $O(\log N)$, therefore AVL trees allow all main operations to be efficient as in the case of the binary search.

For more detailed discussion of AVL and binary trees see the excellent book [2].

In order to represent AVL tree on the whole the `AVLTREE` structure is used:

```
typedef struct AVLTREE AVLTREE;
struct AVLTREE
{     POOL *pool;
      int (*comp)(void *, void *);
      int size;
      AVLNODE *root;
      int height;
};
```

where:

`pool` memory pool (see Section 10.3) that holds all nodes of the tree;

`comp` user-supplied routine for comparing node keys. If this pointer is `NULL`, the ordering by keys is not used;

`size` tree size (total number of nodes in the tree);

`root` pointer to the root node;

`height` total height of the tree.

The representation of a particular node is defined by the `AVLNODE` structure:

```
typedef struct AVLNODE AVLNODE;
struct AVLNODE
{     void *key;
      int rank;
      int type;
      void *link;
      AVLNODE *up;
      short int flag;
      short int bal;
```

```
      AVLNODE *left;
      AVLNODE *right;
};
```

where:

`key` pointer to the key (data type used for representing keys is supplied by the user);

`rank` node rank (the relative position of the node in its own subtree = number of nodes in the left subtree plus one);

`type` node type (this field is intended for the user);

`link` pointer to the node extension (this field is intended for the user);

`up` pointer to the parent node or `NULL` (if this node is the root of the entire tree and has no parent node);

`flag` node flag:

- `0` this node is the left child of its parent (or this node is the root of the entire tree and has no parent node);
- `1` this node is the right child of its parent.

`bal` balance flag (the difference between heights of the right and the left subtrees):

- `-1` the left subtree is higher than the right one;
- ` 0` the left and the right subtrees has equal heights;
- `+1` the left subtree is lower that the right one.

`left` pointer to the root of the left subtree;

`right` pointer to the root of the right subtree.

2.2 `create_avl` — create AVL tree

Synopsis

```
#include "glpavl.h"
AVLTREE *create_avl(int (*comp)(void *, void *));
```

Description

The `create_avl` routine creates an empty AVL tree, which has no nodes.

The parameter `comp` specifies a routine for comparing node keys (data type representing keys should be supplied by the user). The comparing routine should have the following heading:

```
int comp(void *key1, void *key2);
```

where `key1` and `key2` are pointers to the first key and to the second key respectively. This routine should return one of the following values depending on the result of comparison (below [`p`] denotes a value of the object, which `p` points to):

`< 0` if [`key1`] is less than [`key2`];

`= 0` if [`key1`] and [`key2`] are identical;

`> 0` if [`key1`] is greater than [`key2`].

If node keys are character strings, the following statement may be used for creating AVL tree:

```
... = create_avl((int (*)(void *, void *))strcmp);
```

If node keys are not used, it is allowed to specify `NULL` instead the comparing routine. This case assumes inserting, searching, and deleting nodes by their positions (sequential numbers).

Returns

The `create_avl` routine returns a pointer to the created AVL tree.

2.3 `insert_by_key` — insert new node with given key

Synopsis

```
#include "glpavl.h"
AVLNODE *insert_by_key(AVLTREE *tree, void *key);
```

Description

The `insert_by_key` routine creates a new node with the given `key` and inserts this node into the AVL `tree`. (This operation requires key nodes, therefore the key comparing routine should be defined on creating the tree; see Section 2.2.)

The insertion is performed independently on whether there are nodes with the same keys in the tree or not. In order to check that a node with the same key already exists the `find_by_key` routine may be used (see Section 2.6).

It is assumed that the object, which `key` points to, has yet been allocated. This allocation should be kept at least while the node is in the tree or while the tree exists.

On exit the fields `type` and `link` of the structure `AVLNODE`, that represents the inserted node, will contain binary zeros. These fields are available through the pointer returned by the routine and may be used for arbitrary purposes.

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

The `insert_by_key` routine returns a pointer to the inserted node.

Algorithm

At first the routine searches for the point, where a new node with the given key K should be inserted. In order to do that the routine starts from the root of the tree. If $K \leq K'$, where K' is the key of the current node, the routine goes to its left child; otherwise, if $K > K'$, the routine goes to its right child. Such descent is continued until the routine has found an empty left or right subtree, instead which it inserts new node.

After insertion the tree may become disbalanced. It is obvious that only those subtrees can be disbalanced, which were visited on inserting the node. So, the routine rises from the new node up to the root of the tree and restores the balance of each visited node (now these nodes are visited in the reverse order).

Let us assume that the routine visits a node, the left subtree of which was increased as the result of the insertion. Depending on the difference between heights of subtrees of that node the following three cases can take a place (see Fig. 2.2):

a) before insertion $bal = +1$, after insertion $bal = 0$. Since in this case the height of the subtree, the root of which is the visited node, was not changed, further movement up to the tree is terminated;

b) before insertion $bal = 0$, after insertion $bal = -1$. Since in this case the height of the subtree, the root of which is the visited node, was increased, further movement up to the tree should be continued;

c) before insertion $bal = -1$, after insertion $bal = -2$. In this case the routine performs the auxiliary transformation called *rotation*. This transformation, keeping the order of nodes, restores the balance (for details see Section 2.11). After rotation the height of the subtree, the root of which is the visited node, remains unchanged, so further movement up to the tree is terminated.

Now let us assume that the routine visits a node, the right subtree of which was increased as the result of the insertion. The corresponding cases are mirror reflections of cases described above (see Fig. 2.3).

2.4 `next_node` — search for the next node

Synopsis

```
#include "glpavl.h"
AVLNODE *next_node(AVLTREE *tree, AVLNODE *node);
```

Description

The `next_node` routine searches for the next node of the AVL `tree`, which follows the given `node`. If the parameter node is `NULL`, the routine searches for the first node of the tree. The key of the next node is always not less than the key of the given node.

The following statement may be used for visiting all nodes of the tree in the sequence of non-decreasing their keys:

```
for (n = next_node(t, NULL); n != NULL; n = next_node(t, n)) ...
```

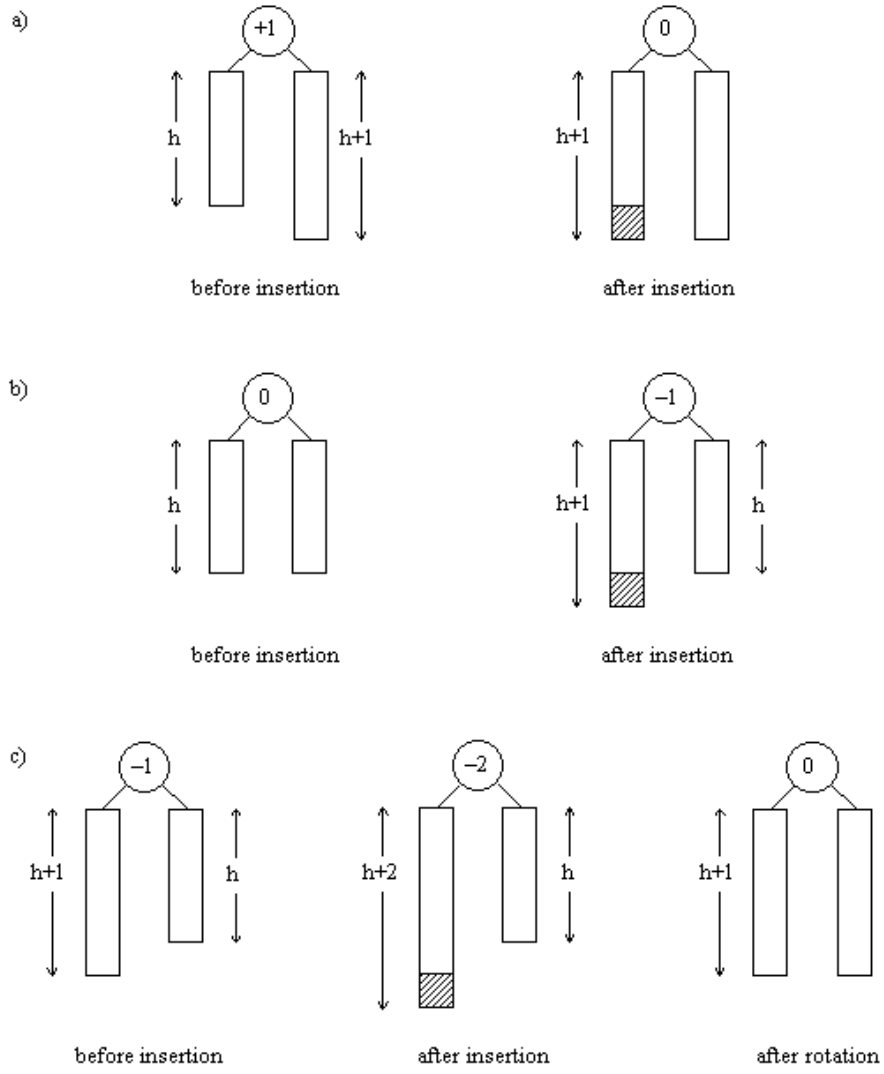

Fig. 2.2. New node was inserted into the left subtree

Note that if the tree has nodes with identical keys, these nodes will be also visited only once, however, the order of visiting these indistinguishable nodes is undefined.

This operation is applicable if nodes has no keys. In such case the next node is that one, which has the next sequential number (see Algorithm).

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

If the parameter `node` is `NULL`, the `next_node` routine returns a pointer to the first node of the tree, or `NULL`, if the tree is empty. If the parameter `node` points to the given node, the `next_node` routine returns a pointer to the next node or `NULL`, if the next node does not exist.

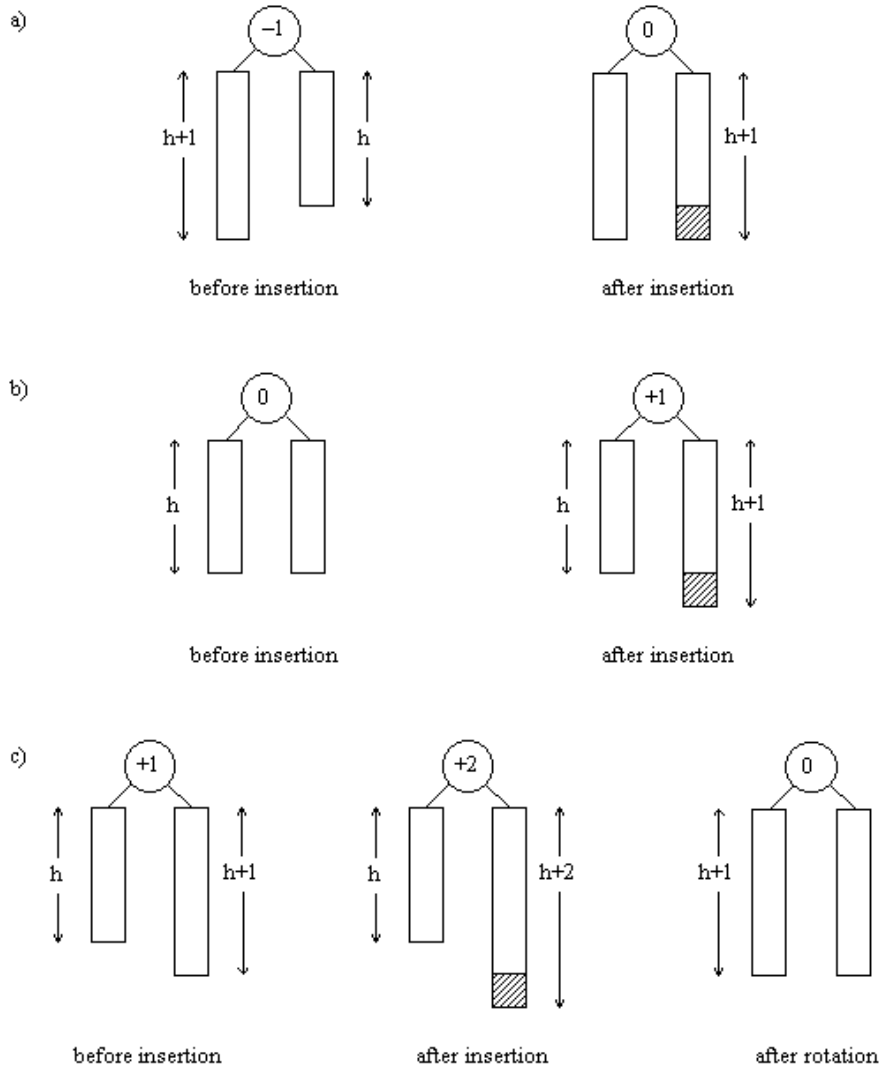

Fig. 2.3. New node was inserted into the right subtree

Algorithm

Visiting all nodes of AVL tree in the straight sequence may be explained by the following recursive procedure:

1) visit all nodes of the left subtree using the same procedure;

2) visit the root;

3) visit all nodes of the right subtree using the same procedure.

However, the `next_node` routine uses non-recursive version of this procedure. Two cases, which may take a place, are schematically shown on Fig. 2.4.

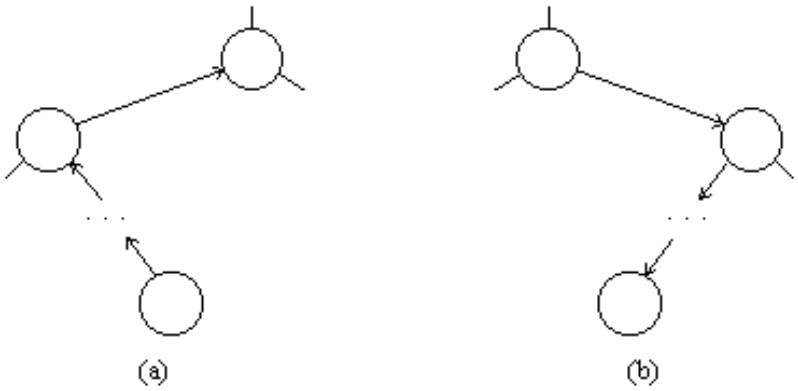

Fig. 2.4. Searching for a node, which follows the given node

2.5 `prev_node` — search for the previous node

Synopsis

```
#include "glpavl.h"
AVLNODE *prev_node(AVLTREE *tree, AVLNODE *node);
```

Description

The `prev_node` routine searches for the previous node of the AVL `tree`, which precedes the given `node`. If the parameter `node` is `NULL`, the routine searches for the last node of the tree. The key of the previous node is always not greater that the key of the given node.

The following statement may be used for visiting all nodes of the tree in the sequence of non-increasing their keys:

```
for (n = prev_node(t, NULL); n != NULL; n = prev_node(t, n)) ...
```

Note that if the tree has nodes with identical keys, these nodes will be also visited only once, however, the order of visiting these indistinguishable nodes is undefined.

This operation is applicable if nodes has no keys. In such case the next node is that one, which has the previous sequential number (see Algorithm).

Complexity

This operation has time complexity $O(\log n)$, where n is the current number of nodes in the tree.

Returns

If the parameter `node` is `NULL`, the `prev_node` routine returns a pointer to the last node of the tree, or `NULL`, if the tree is empty. If the parameter `node` points to the given node, the `prev_node` routine returns a pointer to the previous node or `NULL`, if the previous node does not exist.

Algorithm

Visiting all nodes of AVL tree in the reverse order may be explained by the following recursive procedure:

1) visit all nodes of the right subtree using the same procedure;

2) visit the root;

3) visit all nodes of the left subtree using the same procedure.

However, the `prev_node` routine uses non-recursive version of this procedure. Two cases, which may take a place, are schematically shown on Fig. 2.5.

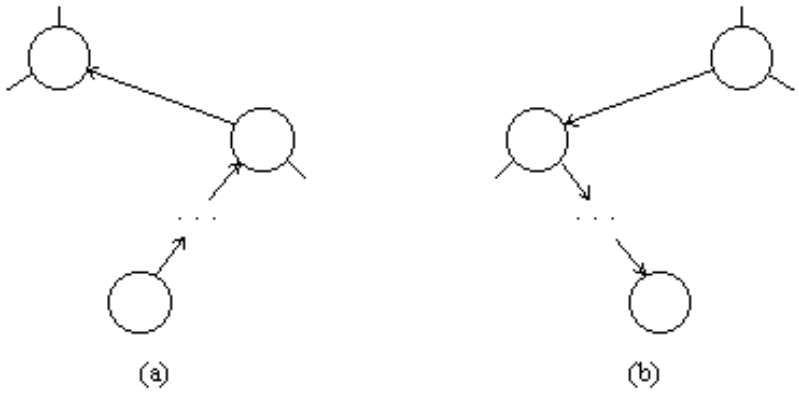

Fig. 2.5. Searching for a node, which precedes the given node

2.6 `find_by_key` — search for the first node with given key

Synopsis

```
#include "glpavl.h"
AVLNODE *find_by_key(AVLTREE *tree, void *key);
```

Description

The `find_by_key` routine searches for the first node in the AVL `tree`, which has the given `key`. (This operation requires key nodes, therefore the key comparing routine should be defined on creating the tree; see Section 2.2.)

In order to find all other nodes with the same key the `next_by_key` routine may be used (see Section 2.7).

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

The `find_by_key` routine returns a pointer to the first node with the given key, or `NULL`, if the tree has no nodes with the given key.

Algorithm

At first the `find_by_key` routine finds arbitrary node with the given key using the same descent on the tree as in the case of inserting new node (see Section 2.8). If a node with the given key has been found, the `find_by_key` routine calls the `prev_node` routine (see Section 2.5) in order to find the first node with the given key.

2.7 `next_by_key` — search for the next node with the same key

Synopsis

```
#include "glpavl.h"
AVLNODE *next_by_key(AVLTREE *tree, void *key);
```

Description

The `next_by_key` routine searches for the next node in the AVL tree, which has the same key as the given node. (This operation requires key nodes, therefore the key comparing routine should be defined on creating the tree; see Section 2.2.)

In order to find the first node with the given key the `find_by_key` routine may be used (see Section 2.6).

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

The `next_by_key` routine returns a pointer to the next node with the same key as the given node, or `NULL`, if the next node with the same key does not exist.

Algorithm

In order to find the next node the `next_by_key` routine calls the `next_node` routine (see Section 2.4). If the next node has the same key as the given node, the routine returns a pointer to the next node, otherwise the routine returns `NULL`.

2.8 `insert_by_pos` — insert new node to given position

Synopsis

```
#include "glpavl.h"
AVLNODE *insert_by_pos(AVLTREE *tree, int pos);
```

Description

The `insert_by_pos` routine creates a new node and inserts it to the given position `pos` of the AVL `tree`. It is necessary that $1 \leq pos \leq n+1$, where $n =$ `tree->size` is total number of nodes in the AVL tree before insertion.

If we renumber all nodes of the AVL tree by numbers $1, 2, \ldots, n$ in that order, in which nodes would be visited by the next_node routine (see Section 2.4), then new node is placed between nodes with numbers `pos`-1 and `pos` (if `pos` $= 1$, new node is placed before the first node, and if `pos` $= n+1$, new node is placed after the last node). Thus, after insertion the new node will have the number `pos`.

On exit the fields `type` and `link` of the structure `AVLNODE`, that represents the inserted node, will contain binary zeros. These fields are available through the pointer returned by the routine and may be used for arbitrary purposes.

It is not recommended to use the `insert_by_pos` routine together with routines, which assume keyed nodes (`insert_by_key`, `find_by_key`, `next_by_key`), because incorrect insertion may invalidate the order of nodes.

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

The `insert_by_pos` routine returns a pointer to the inserted node.

Algorithm

At first the routine searches for the point, where a new should be inserted. In order to do that the routine starts from the root of the tree. If $pos \leq rank$, where $rank$ is the relative position of the current node in its own subtree (number of nodes in the left subtree plus one), the routine goes to the left child of the current node; otherwise, if $pos > rank$, the routine goes to the right child of the current node. Thereto in the latter case the routine decreases pos by the $rank$ value of the current node. The descent is continued until the routine has found an empty left or right subtree, instead which it inserts new node. Then the routine restores the balance of all subtrees in the same way as in case of insertion by keys (see Section 2.3).

2.9 find_by_pos — search for node placed in given position

Synopsis

```
#include "glpavl.h"
AVLNODE *find_by_pos(AVLTREE *tree, int pos);
```

Description

The `find_by_pos` routine searches for a node, which is placed in the given position `pos` of the AVL `tree`.

Do not use this routine for visiting all nodes of the AVL tree, because this is inefficient. It's better to use the `next_node` or the `prev_node` routines (see Sections 2.4 and 2.5).

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

If $1 \leq pos \leq n$, where $n =$ `tree->size` is total number of nodes in the AVL tree, the routine returns a pointer to the node placed in the given position `pos`. Otherwise the routine returns `NULL`.

2.10 delete_node — delete given node

Synopsis

```
#include "glpavl.h"
void delete_node(AVLTREE *tree, AVLNODE *node);
```

Description

The `delete_node` routine deletes the given `node` from the AVL `tree`. It is assumed (without any checking) that the given node belongs to the given tree; otherwise the behavior is undefined. (This operation is applicable independently of whether nodes have keys or not, since the order of nodes is preserved.)

Note that the `delete_node` routine deletes exactly the specified node, therefore all references to other nodes from the user's data structures (if such references exist) will remain valid after deletion.

Actually the routine doesn't free the memory allocated to the deleted node. It just returns that memory to the memory pool, so that memory can be reused in the future for the same tree.

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Algorithm

If both subtrees of the deleted node are not empty, the routine exchanges the deleted node with the next node, the left subtree of which is empty. It is important to note that the routine exchanges nodes themselves (i.e. structures of `AVLNODE` type), but not data placed in nodes. Therefore all references to other nodes from the user's data structures (if such references exist) will remain valid after deletion.

In order to delete node, which has one non-empty subtree, the routine connects that subtree to the parent of the deleted node. If the deleted node has no subtrees, the routine just disconnect the node from its parent.

After deletion the tree may become disbalanced. So, the routine rises from the deleted node up to the root of the tree and restores the balance of each visited node similarly to the case of insertion (see Section 2.3).

Let us assume that the routine visits a node, the left subtree of which was decreased as the result of the deletion. Depending on the difference between heights of subtrees of that node the following three cases can take a place (see Fig. 2.6):

a) before deletion $bal = -1$, after deletion $bal = 0$. Since in this case the height of the subtree, the root of which is the visited node, was decreased, further movement up to the tree should be continued;

b) before deletion $bal = 0$, after deletion bal = +1. Since in this case the height of the subtree, the root of which is the visited node, was not changed, further movement up to the tree is terminated;

c) before deletion bal = +1, after deletion bal = +2. In this case the routine performs the auxiliary transformation called *rotation*. This transformation, keeping the order of nodes, restores the balance (for details see Section 2.11). After rotation the routine analyzes the balance flag of that node, which has replaced the visited node. If $bal = -1$ (the height of subtree was not changed), the movement is terminated. If $bal = 0$ (the height of subtree was decreased), the movement is continued.

Now let us assume that the routine visits a node, the right subtree of which was decreased as the result of the deletion. The corresponding cases are mirror reflections of cases described above (see Fig. 2.7).

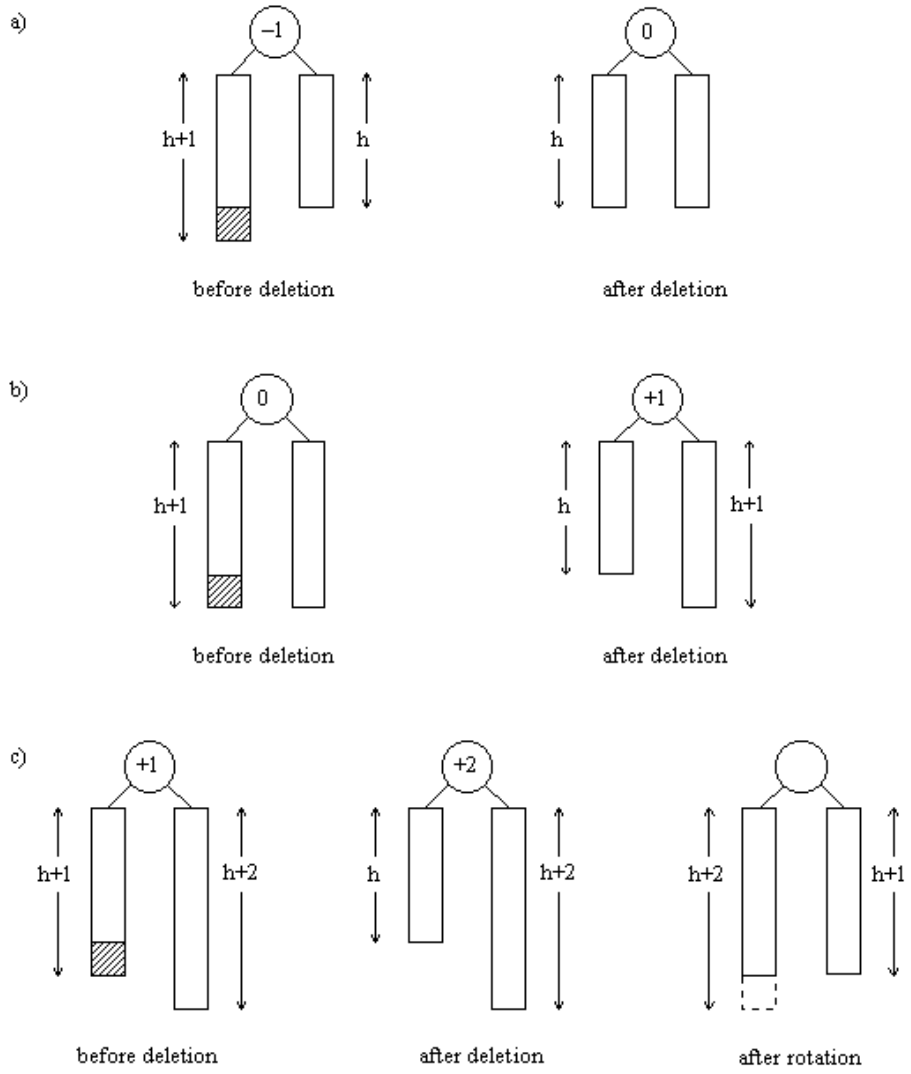

Fig. 2.6. Node was deleted from the left subtree

2.11 `rotate_avl` — restore balance of AVL subtree

Synopsis

```
#include "glpavl.h"
void delete_node(AVLTREE *tree, AVLNODE *node);
```

Description

The `rotate_avl` routine restores balance of the subtree (of the AVL `tree`) specified by its root, which the parameter `node` points to.

The `rotate_avl` routine is auxiliary, it is not intended for independent usage.

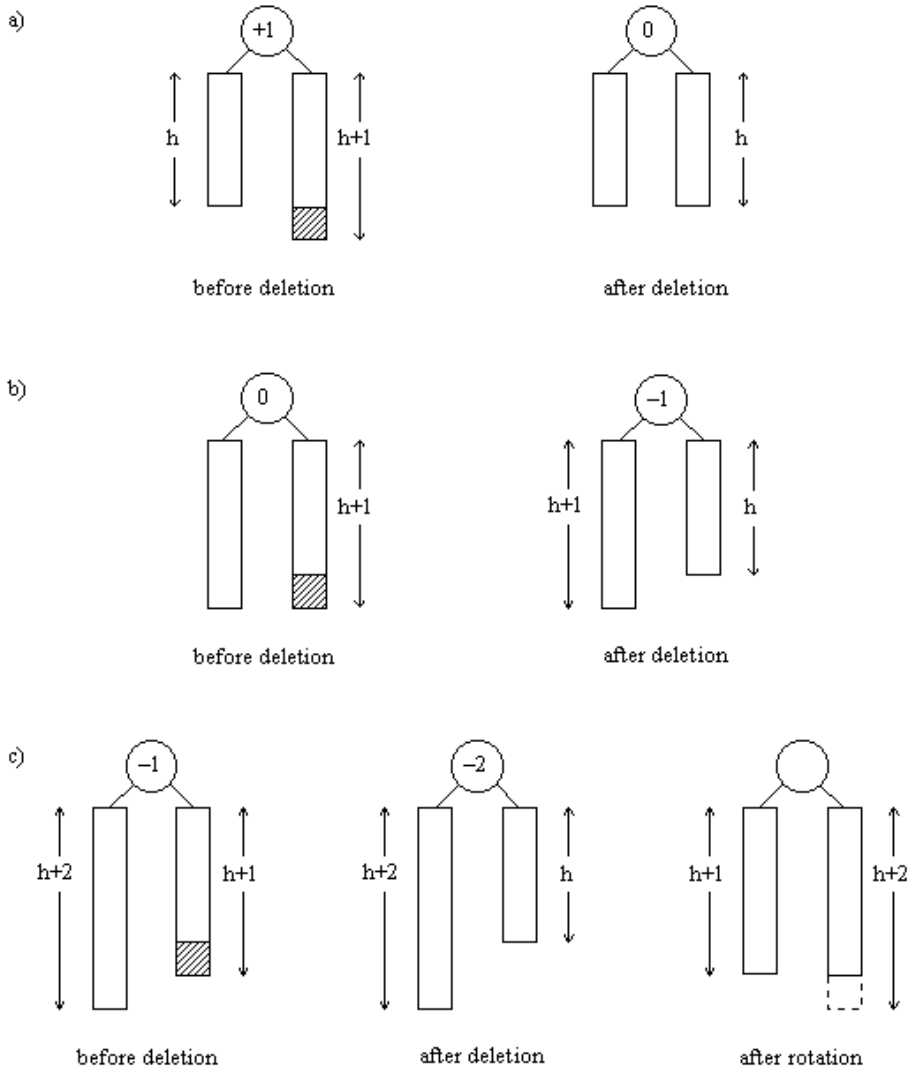

Fig. 2.7. Node was deleted from the right subtree

Complexity

This operation has time complexity $O(\log n)$, where n is current number of nodes in the tree.

Returns

The `rotate_avl` routine returns a pointer to the node, which replaces the root of the specified subtree.

Algorithm

If the height of the left subtree of the given key is higher that the height of the right subtree, the routine performs single or double clockwise rotation depending on cases shown on Fig. 2.8. Analogously, if the height of the left subtree of the given key is lower that the

height of the right subtree, the routine performs single or double counterclockwise rotation depending on cases shown on Fig. 2.9.

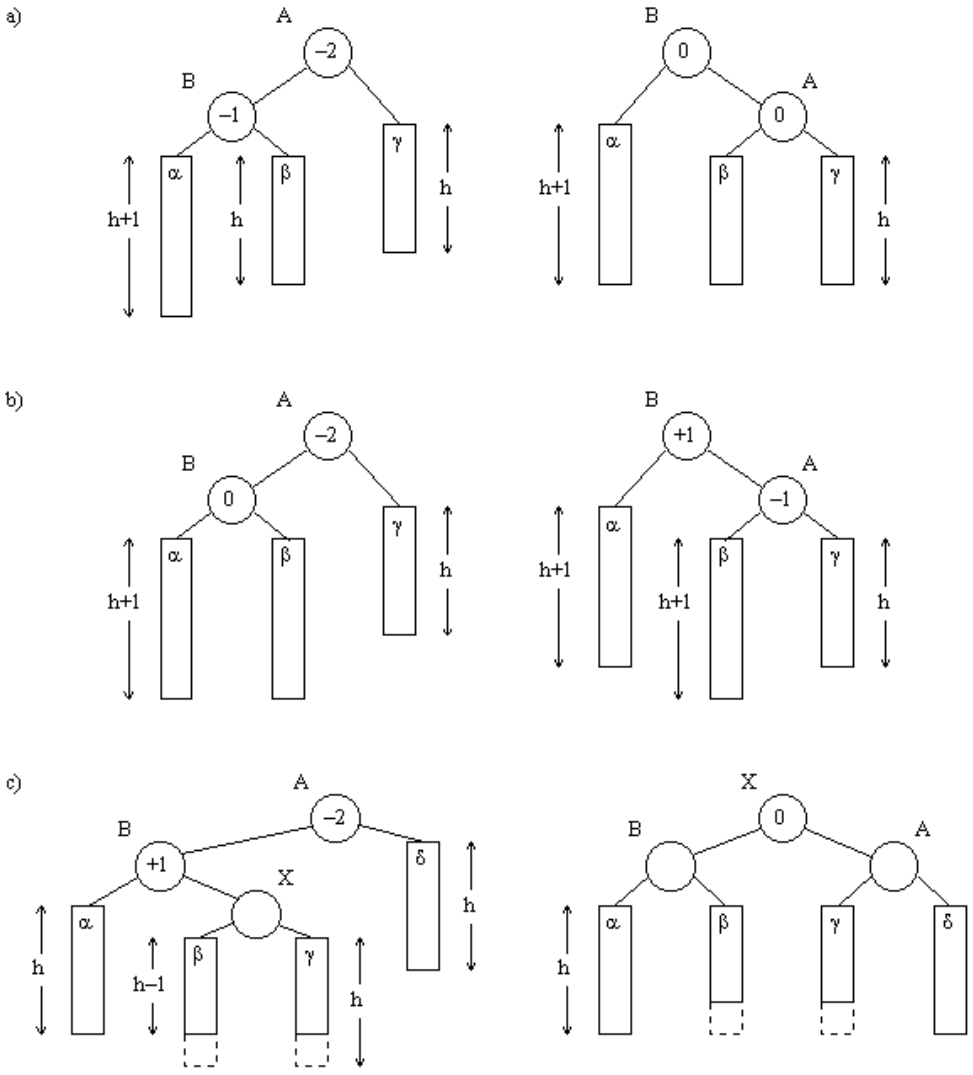

Fig. 2.8. Single (a, b) and double (c) clockwise rotation

2.12 `delete_avl` — delete AVL tree

Synopsis

```
#include "glpavl.h"
void delete_avl(AVLTREE *tree);
```

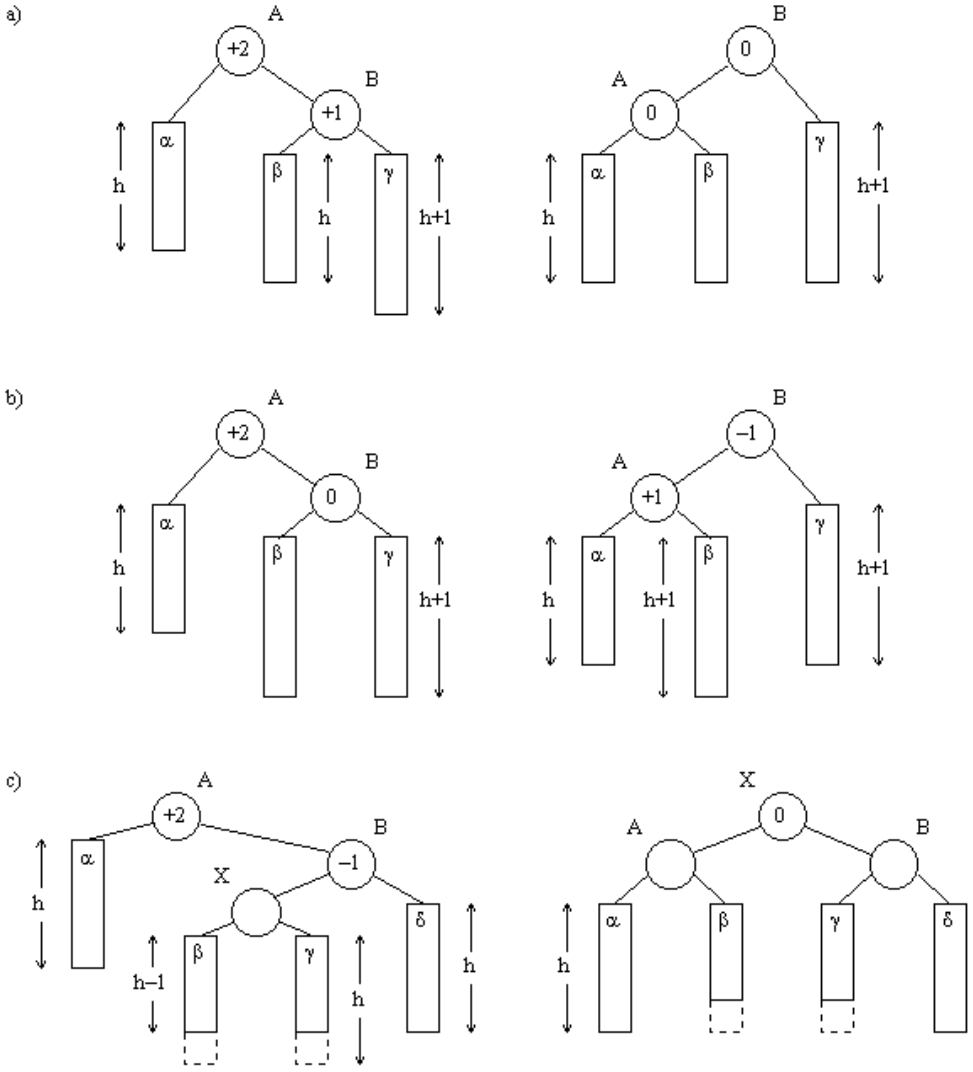

Fig. 2.9. Single (a, b) and double (c) counterclockwise rotation

Description

The `delete_avl` routine deletes the AVL `tree`, freeing all memory allocated to this object. Note that the routine doesn't free any memory allocated to the user's data structures.

References

1. *G. M. Adelson-Velsky, E. M. Landis.* Odin algoritm organizacii informacii. DAN SSSR, 146, 1962, s. 263—66 (in Russian).
2. *Donald E. Knuth.* The Art of Computer Programming. Vol. 3. Sorting and Searching. Addison-Wesley, 1973.

3 GLPDUFF: Duff Search Technique

3.1 Prelude

Let us consider a set of n objects numbered by integers $1, 2, \ldots, n$, where a particular object with the number k can posses the length $\lambda l(k)$. *Duff search technique* allows efficient (i.e. for a fixed time) performing the following two main operations:

determining numbers of all objects that have the given length;

changing the length of an object.

These operations appear mainly in implementations of sparse gaussian elimination based on Markowitz strategy for pivoting. The considered search technique was proposed by I. Duff [1].

This implementation of Duff search technique is based on *Duff scheme*, which is a set of unordered double linked lists. A particular linked list contains numbers of objects that have the same length.

Duff scheme is represented by the DUFF structure:

```
typedef struct DUFF DUFF;
struct DUFF
{     int n;
      int maxlen;
      int len[1:n];
      int head[0:maxlen];
      int prev[1:n];
      int next[1:n];
};
```

where:

`n` total number of objects, n > 0;

`maxlen` maximal length of each object, $maxlen \geq 0$;

`len` array of object lengths. Location `len`[0] is not used. Location `len`[k] contains the length of kth object ($1 \leq k \leq n$) in the range from 0 to `maxlen`. The case `len`[k] < 0 means that kth object is not in the scheme and its length is undefined;

`head` array of initial pointers. Location `head`[len] ($0 \leq len \leq maxlen$) contains number of the first object, which has the length len;

`prev` array of predecessors. Location `prev`[0] is not used. Location `prev`[k] ($1 \leq k \leq n$) contains number of the preceding object, which has the same length as kth object (`prev`[k] = 0 means that kth object is the first in the list);

`next` array of successors. Location `next`[0] is not used. Location `next`[k] ($1 \leq k \leq n$) contains number of the succeeding object, which has the same length as kth object (`next`[k] = 0 means that kth object is the last in the list).

3.2 create_duff — create Duff scheme

Synopsis

```
#include "glpduff.h"
DUFF *create_duff(int n, int maxlen);
```

Description

The `create_duff` routine creates Duff scheme that can contain up to `n` objects, where the length of each object may be in the range from 0 to `maxlen`. Initially the scheme contains no objects.

Returns

The `create_duff` routine returns a pointer to the created scheme.

3.3 include_obj — include object into scheme

Synopsis

```
#include "glpduff.h"
void include_obj(DUFF *duff, int k, int len);
```

Description

The `include_obj` routine includes the object with number `k` ($1 \le k \le n$) into the Duff scheme `duff`. The parameter `len` specifies the object length ($1 \le len \le maxlen$). It is assumed that before inclusion the object doesn't belong the scheme.

Complexity

This operation has time complexity $O(1)$.

3.4 exclude_obj — exclude object from scheme

Synopsis

```
#include "glpduff.h"
void exclude_obj(DUFF *duff, int k);
```

Description

The `exclude_obj` routine excludes the object with number `k` ($1 \le k \le n$) from the Duff scheme `duff`. It is assumed that before exclusion the object does belong to the scheme.

Complexity

This operation has time complexity $O(1)$.

3.5 Determining all objects with given length

In order to determine all objects that have the given length *len* the following for statement may be used:

```
for (k = duff->head[len]; k; k = duff->next[k])
{
   /* processing the object with number k */
}
```

3.6 Determining length of given object

The length of the object with number k can be obtained from the location `duff->len[k]`. The case `duff->len[k]` < 0 means, that the object with number k doesn't belong to the Duff scheme, therefore its length is undefined.

3.7 Changing length of given object

In order to change the length of the object with number k it is necessary to exclude this object from the Duff scheme and to include it back into the scheme using its new length:

```
len = duff->len[k];        /* determine current length */
exclude_obj(duff, k);      /* exclude object from scheme */
len = ...                  /* set new length */
include_obj(duff, k, len); /* include object into scheme */
```

3.8 `reset_duff` — remove all objects from scheme

Synopsis

```
#include "glpduff.h"
void reset_duff(DUFF *duff);
```

Description

The `reset_duff` routine removes all objects from the Duff scheme `duff`, so the scheme becomes empty.

Complexity

This operation has time complexity $O(\max(n, maxlen))$.

3.9 `delete_duff` — delete Duff scheme

Synopsis

```
#include "glpduff.h"
void delete_duff(DUFF *duff);
```

Description

The `delete_duff` routine deletes the Duff scheme `duff`, freeing all the memory allocated to this object.

References

1. *I. S. Duff.* MA28 — a set of FORTRAN subroutines for sparse unsymmetric linear equations. Harwell Report AERE, R. 8730, 1977.

4 GLPEFI: Elimination Form of the Inverse (EFI)

4.1 Prelude

One of the most essential components of any implementation of the revised simplex method for linear programming is the representation of the basis matrix[1], which allow performing the following four main operations:

- building the representation of a given basis matrix B;
- updating the representation if one column of B has been replaced by other column;
- forward transformation of a given vector z by means of solving the system $Bz' = z$;
- backward transformation of a given vector z by means of solving the system $B^T z' = z$.

This chapter describes routines implementing so called *Elimination Form of the Inverse (EFI)*, which is the following representation of the basis matrix:

$$B = B_0 H, \tag{4.1}$$

where B_0 is an initial basis matrix, H is a matrix represented in the form of eta file (see Section 5.1).

Let there is the current basis matrix B of order m, for which the representation (4.1) is known. Let also $\overline{B}$ is the adjacent basis matrix, which differs from B only in one column B_p. Since B is assumed to be non-singular and $B^{-1}B = I$, therefore

$$B^{-1}\overline{B} = \overline{H}, \tag{4.2}$$

where $\overline{H}$ also differs from unity matrix only in one pth column.

Let pth column of $\overline{H}$ is η. Then from (4.2) it follows that

$$B^{-1}B_p = \eta, \tag{4.3}$$

i.e. pth column of $\overline{H}$ may be determined by solving the system $B\eta = B_p$.

It is well known that the matrix $\overline{H}$ can be represented in the product form

$$\overline{H} = H^{(p)}H^{(1)} \ldots H^{(p-1)}H^{(p+1)} \ldots H^{(m)}, \tag{4.4}$$

where each term $H^{(i)}$ on the right hand side has the form of eta term (5.2), and non-trivial elements of $H^{(p)}, H^{(1)}, \ldots, H^{(m)}$ are, respectively, $\eta_p, \eta_1, \ldots, \eta_m$. In other words each element of the column η (that is pth column of the matrix $\overline{H}$) gives a particular eta term $H^{(i)}$, which has η_i placed in ith row and pth column. Note that the eta term $H^{(p)}$, which corresponds to the diagonal element η_p, should follow first, other eta terms $H^{(i)}$ $(i \neq p)$ may follow in arbitrary order[2].

Thus, if (4.1) represents the current basis matrix B, the adjacent basis matrix $\overline{B}$ may be represented in the same form:

$$\overline{B} = B\overline{H} = B_0(H\overline{H}), \tag{4.5}$$

[1] *Basis matrix* or just *basis* is a quadratic non-singular matrix formed by columns of basis variables.

[2] If $\eta_i = 0$, it is understood that eta term $H^{(i)}$ is unity matrix and may be omitted.

i.e. replacing the current matrix B to the adjacent matrix $\overline{B}$ just involves new eta terms to be appended to the eta file H.

The initial basis matrix B_0 is represented in decomposed form using LU factorization (see also Section 6.2):

$$B_0 = PLUQ, \tag{4.6}$$

where L is lower triangular matrix with unit diagonal, U is upper triangular matrix, P and Q are permutation matrices.

If the representation (4.1) becomes inaccurate or the eta file H becomes too large, EFI routines allows reinverting the current basis matrix, i.e. set $B_0 = B$ and $H = I$ and compute LU factorization of B_0. Of course, EFI may be built also for other given basis matrix.

From (4.1) it follows that $B^{-1} = H^{-1}$ and[3] $B^{-T} = B_0^{-T} H^{-T}$ Therefore forward and backward transformations can be efficiently performed by solving systems with matrices B_0 and H.

Note that EFI has a minor disadvantage, because in some cases column η may be quite dense, and there is no freedom for improving its sparsity. On the other hand EFI provides good numerical stability.

EFI is defined by the following structure:

```
typedef struct EFI EFI;
struct EFI
{     int m;
      LU *lu;
      HFILE *eta;
      double col[1+m];
      int flag;
      double work[1+m];
};
```

where:

`m` order of basis matrix B;

`lu` representation of initial basis matrix B_0 in the form of LU factorization;

`eta` representation of matrix H in the form of eta file;

`col` column η, which has to be used on updating EFI for adjacent basis matrix $\overline{B}$;

`flag` column flag (this flag is reset to 0 after each updating EFI):

- `0` column η is not ready yet;
- `1` column η is ready for updating.

`work` working storage.

(It is understood that the column η is transformed column of that non-basis variable, which enters the adjacent basis. In case of the primal simplex method this column is used on choosing basis variable that leaves the current basis, therefore keeping this column after forward transformation allows not to perform this operation on subsequent updating EFI.)

[3] B^{-T} means $(B^T)^{-1}$.

4.2 `create_efi` — create EFI

Synopsis

```
#include "glpefi.h"
EFI *create_efi(int m);
```

Description

The `create_efi` routine creates EFI (4.1) for representing basis matrix of order `m`. Initially EFI corresponds to unity basis matrix, i.e. $B_0 = H = I$.

Returns

The `create_efi` routine returns a pointer to the created EFI.

4.3 `build_efi` — build EFI for given basis matrix

Synopsis

```
#include "glpefi.h"
int build_efi(EFI *efi, MAT *A, int indb[]);
```

Description

The `build_efi` routine builds EFI for the given basis matrix B (see below) in the form (4.1). It sets $B_0 = B$ and $H = I$ (unity matrix) and then computes LU factorization (4.6) by means of the `build_lu` routine (see Subsection 6.2.2). This operation is usually performed if the current EFI becomes inaccurate or too large, or the basis matrix is completely changed.

The basis matrix B is specified implicitly by means of the sparse matrix `A` and the list of basis columns `indb`. The matrix A should have m rows, where m is order of basis matrix. Locations `indb[1]`, `indb[2]`, ..., `indb[m]` should contain numbers of basis columns of the matrix A, which specify the basis matrix B.

Returns

If EFI has been successfully built, the `build_efi` routine returns zero. Otherwise, if the matrix B is singular or ill conditioned, the routine returns non-zero. In the latter case partial LU factorization can be used for further analysis of the matrix B (for details see Chapter 6).

4.4 `update_efi` — update EFI for adjacent basis matrix

Synopsis

```
#include "glpefi.h"
int update_efi(EFI *efi, int p);
```

Description

The `update_efi` routine recomputes EFI, which `efi` points to and which corresponds to the current basis matrix B, in order that it corresponds to the adjacent basis matrix $\overline{B}$ (see Subsection 4.1).

The parameter `p` specifies the number of column of the basis matrix B, which is replaced by new column if the adjacent basis matrix $\overline{B}$.

New basis column is passed implicitly. The `update_efi` routine assumes that new basis column was prepared before by means of the `efi_ftran` routine (see Section 4.5).

Returns

The `update_efi` routine returns one of the following error codes:

0 EFI has been successfully updated;

1 EFI is inaccurate;

2 EFI is too large.

In case of non-zero error code the calling program should reinvert the basis matrix by means of the `build_efi` routine (see Section 4.3).

4.5 `efi_ftran` — perform forward transformation (FTRAN)

Synopsis

```
#include "glpefi.h"
double *efi_ftran(EFI *efi, double z[], int save);
```

Description

The `efi_ftran` routine performs so called *forward transformation (FTRAN)* of a given vector, i.e. it solves the system $Bz' = z$, where B is the current basis matrix represented by EFI, which `efi` points to.

On entry the array `z` should contain elements of the input vector z in locations `z[1]`, `z[2]`, ..., `z[m]`, where m is order of the basis matrix. On exit the array `z` contains elements of the transformed vector z' in the same locations.

The parameter `save` is a flag. If this flag is set, it means that the input vector z is a column corresponding to that non-basis variable, which has been chosen to enter the basis. In this case the `efi_ftran` routine saves the transformed column z' (that is column η; see (4.3)) in order that it could be used later by the `update_efi` routine (see Section 4.4) on updating EFI. It is assumed that the calling program performs at least one call to the `efi_ftran` routine with the `save` flag set.

Returns

The `efi_ftran` routine returns a pointer to the array `z`.

4.6 efi_btran — perform backward transformation (BTRAN)

Synopsis

```
#include "glpefi.h"
double *efi_btran(EFI *efi, double z[]);
```

Description

The `efi_btran` routine performs so called *backward transformation (BTRAN)* of a given vector, i.e. it solves the system $B^T z' = z$, where B is the current basis matrix represented by EFI, which `efi` points to.

On entry the array `z` should contain elements of the input vector z in locations `z[1]`, `z[2]`, ..., `z[m]`, where m is order of the basis matrix. On exit the array `z` contains elements of the transformed vector z' in the same locations.

Returns

The `efi_ftran` routine returns a pointer to the array `z`.

4.7 delete_efi — delete EFI

Synopsis

```
#include "glpefi.h"
void delete_efi(EFI *efi);
```

Description

The `delete_efi` routine deletes EFI, which `efi` points to, freeing all the memory allocated to this object.

5 GLPETA: Implementation of Eta File

5.1 Prelude

Product form of (square) matrix H, which is also called eta file[1], is the representation of this matrix in the following multiplication form:

$$H = H_1 H_2 \ldots H_N = \sum_{k=1}^{N} H_k \tag{5.1}$$

(in case of $n = 0$ it is assumed that H is unity matrix).

Each term H_k has the form:

$$H_k = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & \eta_{ij} & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}, \tag{5.2}$$

i.e. each term of eta file differs from the unity matrix only in one element η_{ij}, which may be placed out of the main diagonal ($i \neq j$) as well as on the main diagonal ($i = j$).

In order to represent eta file on the whole the `HFILE` structure is used:

```
typedef struct HFILE HFILE;
struct HFILE
{     int n;
      POOL *pool;
      HTERM *head;
      HTERM *tail;
};
```

where:

`n` the order of matrix H (all eta terms have the same order), $n > 0$;

`pool` memory pool (see Section 10.3) that holds all eta terms;

`head` pointer to the first eta term H_1;

`tail` pointer to the last eta term H_N.

The representation of a particular eta term is defined by the `HTERM` structure:

[1] The name "eta file" appeared in the beginning of 1960s, when terms of the form (5.1) used to be stored as records on a magnetic tape and the corresponding sequential file was often denoted by the letter H (Greek "eta").

```
typedef struct HTERM HTERM;
struct HTERM
{     int i;
      int j;
      double val;
      HTERM *prev;
      HTERM *next;
};
```

where:

`i` row index of element η_{ij};

`j` column index of element η_{ij};

`val` numerical value of element η_{ij};

`prev` pointer to the previous eta term H_{k-1} or `NULL`, if this eta term is H_1;

`next` pointer to the next eta term H_{k+1} or `NULL`, if this eta term is H_N.

5.2 `create_eta` — create eta file

Synopsis

```
#include "glpeta.h"
HFILE *create_eta(int n);
```

Description

The `create_eta` routine creates eta file (5.1) of order n. Initially the created eta file has no terms and therefore it formally corresponds to unity matrix.

Returns

The `create_eta` routine returns a pointer to the created eta file.

5.3 `app_term` — append new eta term

Synopsis

```
#include "glpeta.h"
HTERM *app_term(HFILE *eta, int i, int j, double val);
```

Description

The `app_term` routine appends new eta term $H_{N+1} = (\eta_{ij})$ to the end of the eta file `eta`, where `i` and `j` are respectively row and column indices of element η_{ij}, `val` is its numerical value.

If $i = j$ and $val = 1.0$ or if $i \neq j$ and $val = 0.0$, the new eta term formally corresponds to unity matrix. Therefore in this two cases the routine actually doesn't append the eta term.

Complexity

This operation has time complexity $O(1)$.

Returns

The `app_term` routine returns a pointer to the new eta term or `NULL`, if the eta term was not appended because of two cases mentioned above.

5.4 `h_solve` — solve system $Hx = b$

Synopsis

```
#include "glpeta.h"
double *h_solve(HFILE *eta, double x[]);
```

Description

The `h_solve` routine solves the system $Hx = b$, where H is the matrix represented by the eta file, which `eta` points to, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain elements of the vector b in locations `x[1]`, `x[2]`, ..., `x[n]`. On exit the array `x` contains elements of the vector x in the same locations.

The matrix H should be non-singular, i.e. all η_{ii} should be non-zero.

Complexity

This operation has time complexity $O(N)$, where N is total number of eta terms.

Returns

The `h_solve` routine returns a pointer to the array `x`.

Method

From (5.1) it follows that

$$x = H^{-1}b = H_N^{-1} \dots H_2^{-1} H_1^{-1} b,$$

therefore solving the system $Hx = b$ involves sequential solving systems with trivial matrices $H_1, H_2, \dots, H_N$ of the form (5.2).

5.5 ht_solve — solve system *H'x = b*

Synopsis

```
#include "glpeta.h"
double *ht_solve(HFILE *eta, double x[]);
```

Description

The `ht_solve` routine solves the system $H^T x = b$, where H is the matrix represented by the eta file, which `eta` points to, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain elements of the vector `b` in locations `x[1]`, `x[2]`, ..., `x[n]`. On exit the array `x` contains elements of the vector x in the same locations.

The matrix H should be non-singular, i.e. all η_{ii} should be non-zero.

Complexity

This operation has time complexity $O(N)$, where N is total number of eta terms.

Returns

The `ht_solve` routine returns a pointer to the array `x`.

Method

From (5.1) it follows that[2]

$$x = H^{-T} b = H_1^{-T} H_2^{-T} \dots H_N - T,$$

therefore solving the system $H^T x = b$ involves sequential solving systems with trivial matrices $H_N^T, \dots, H_2^T, H_1^T$ of the form (5.2).

5.6 reset_eta — reset eta file

Synopsis

```
#include "glpeta.h"
void reset_eta(HFILE *eta);
```

Description

The `reset_eta` routine removes all eta terms from the eta file `eta`. After this operation the eta file formally corresponds to unity matrix.

Actually the routine doesn't free the memory allocated to eta terms. It just returns that memory to the memory pool, so that memory can be reused in the future for the same eta file.

[2] H^{-T} means $(H^T)^{-1}$.

5.7 `delete_eta` — delete eta file

Synopsis

```
#include "glpeta.h"
void delete_eta(HFILE *eta);
```

Description

The `delete_eta` routine deletes the eta file `eta`, freeing all the memory allocated to this object.

6 GLPGEL: Sparse Gaussian Elimination

This chapter describes a set of routines related to sparse gaussian elimination.

6.1 Unsymmetric Gaussian Elimination

This section describes routines, which implement general version of sparse unsymmetric gaussian elimination as well as some specific versions of this technique assuming the special structure of transformed matrix.

6.1.1 `gel` — transform sparse matrix to upper triangular form using gaussian elimination

Synopsis

```
#include "glpgel.h"
int gel(MAT *V, PER *P, PER *Q,
   void (*func)(int i, int k, double f), double tol, double lim,
   int *Unz, double *Umax, double *Ubig, DUFF *rs, DUFF *cs,
   double rmax[], double work[]);
```

Description

The `gel` routine transforms the sparse matrix U, which initially is the given sparse matrix V, to the upper triangular form using gaussian elimination technique. The result of transformation is the upper triangular matrix $U' = PV'Q$, where V' is the resultant matrix placed instead the input matrix V, P and Q are resultant permutation matrices.

The matrix U, which is transformed to the final upper triangular matrix U', has implicit representation, i.e. all operations are actually performed on the matrix V, which differs from U only in the order of rows and columns determined by permutation matrices P and Q. This implicit representation allows not to explicitly permute rows and columns during elimination and thereby allows improving the efficiency of computational process.

The example of transformation of the test matrix $D(100, 30)$ (see Section 7.28) performed by means of the gel routine is shown on Fig. 6.1.

In order to eliminate subdiagonal elements of the matrix $U = PVQ$ the routine applies to the matrix U elementary transformations of the following three kinds: row permutation, column permutation, and subtraction one row multiplied by a number from other row (gaussian transformation). Each time, when gaussian transformation is applied to the matrix U (in order to eliminate one of its subdiagonal elements), the `gel` routine calls the formal routine `func` in order to allow the calling program accumulating information about all performed gaussian transformations (information about row and column permutations is accumulated in the matrices P and Q by the `gel` routine and also available to the calling program). (This feature of the `gel` routine allows accommodating it to any context, which needs transformation to the upper triangular form, unlike other existing gaussian elimination routines, which usually accumulate information about performed gaussian transformations in some particular form: in the form of lower triangular matrix in case of LU factorization, in the form of eta file, etc.)

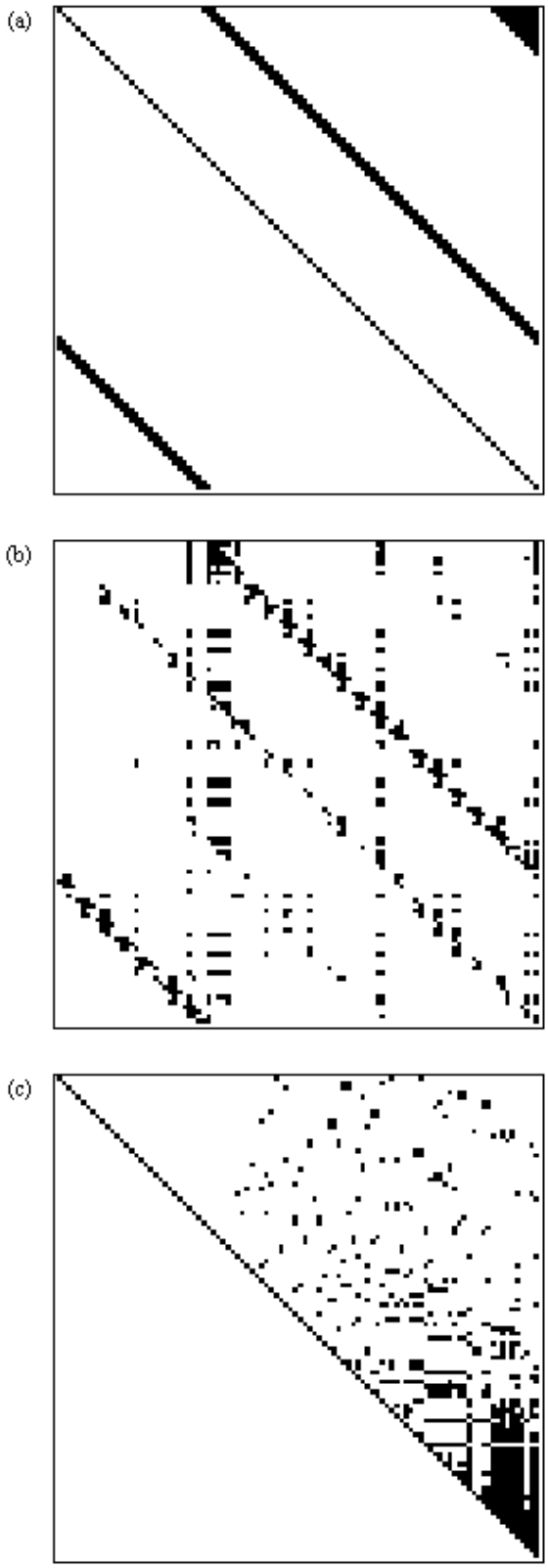

Fig. 6.1. Example of transformation performed by the `gel` routine
(a) input matrix $U = V$; (b) resultant matrix V'; (c) resultant matrix $U' = PV'Q$.

The point, where the `func` routine is called, and the meaning of parameters passed to that routine can be explained with the following model scheme of the elimination process:

```
for (k = 1; k <= n; k++)
{  /* k-th elimination step starts here */
   /* choose the pivot element u[p,q] */
   p = ...; assert(k <= p <= n);
   q = ...; assert(k <= q <= n);
   /* permute rows of the matrix U with numbers k and p */
   ...;
   /* permute columns of the matrix U with numbers k and q */
   ...;
   /* now the pivot element is u[k,k] */
   assert(u[k,k] != 0.0);
   /* eliminate non-zero subdiagonal elements of the matrix U that
      are placed in k-th column */
   for (i = k+1; i <= n; i++)
   {  /* eliminate subdiagional element u[i,k] */
      if (u[i,k] == 0.0) continue;
      /* compute gaussian multiplier */
      f = u[i,k] / u[k,k];
      /* (i-th row of U) := (i-th row of U) - f * (k-th row of U) */
      ...;
      assert(u[i,k] == 0.0);
      /* i-th row of the matrix U has been transformed */
      func(i, k, f);
   }
}
```

Note that row numbers passed to the `func` routine correspond to the matrix U, not to V. The correspondence between row and column numbers of the matrices $U = PVQ$ and V is determined by the row permutation matrix P and the column permutation matrix Q. Thus, if an element $u[i,j]$ of the matrix U corresponds to an element $v[i',j']$ of the matrix V, the following formulae may be used:

```
i' = P->row[i], j' = Q->col[j];
i = P->col[i'], j = Q->row[j'];
```

(these formulae are valid on exit as well as during all the elimination process).

The control parameter *tol* is a relative tolerance used for threshold pivoting. Its value should be in the range $0 <$ *tol* < 1. For more details see the method description below.

The control parameter *lim* is intended for checking numerical stability and specifies maximal allowable growth of elements of the matrix U during elimination process. For more details see the method description below.

On entry the location `Unz` should contain the total number of non-zeros in the matrix $U = V$. On exit this location contains total number of non-zeros in the final matrix $U' = PV'Q$. (If on entry the location `Unz` contains zero, on exit it contains the fill-in count, i.e. the difference between numbers of non-zeros in the matrices U' and U.)

On entry contents of the locations `Umax` and `Ubig` are ignored. On exit the location `Umax` contains maximal of absolute values of elements of the initial matrix $U = V$. The location

`Ubig` contains maximal of absolute values of elements of the matrix U, which appeared in this matrix during entire elimination process (i.e. $Ubig = max|u_{ij}^k|$, where k is number of the elimination step). Thus, the ratio $Ubig/Umax$ may be used to measure the numerical "quality" of transformation.

The parameters `rs` and `cs` should specify Duff schemes (see Chapter 3), which are used by the `gel` routine to keep lists of active rows and columns. On entry contents of these schemes is ignored. In the case of error (see below) on exit these schemes contain active rows and columns of the matrix V (not U!) and define the final active submatrix (this information may be used for further analysis). These schemes should be created by statements `rs = create_duff(n, n)` and `cs = create_duff(n, n)`, where n is order of the matrix V.

The arrays `rmax` and `work` are working storage. They should have at least $1+n$ locations, where n is order of the matrix V.

Returns

The `gel` routine returns one the following error codes:

0	no errors;
$-k$	on kth elimination step all elements of the active submatrix are equal to zero;
$+k$	on kth elimination step the numerical stability condition is violated (see below the method description).

Method

In order to transform the given matrix U to the upper triangular form the `gel` routine uses gaussian elimination with partial pivoting based on Markowitz strategy and Duff search technique (see, for example, [1]).

Before kth elimination step the partially transformed matrix U has the form:

$$U = \begin{pmatrix} u_{11} & \dots & u_{1,k-1} & u_{1k} & u_{1,k+1} & \dots & u_{1n} \\ & \dots & \dots & \dots & \dots & \dots & \dots \\ & & u_{k-1,k-1} & u_{k-1,k} & u_{k-1,k+1} & \dots & u_{k-1,n} \\ & & & u_{kk} & u_{k,k+1} & \dots & u_{kn} \\ & & & \dots & \dots & \dots & \dots \\ & & & u_{nk} & u_{n,k+1} & \dots & u_{nn} \end{pmatrix}, \tag{6.1}$$

where rows and columns with numbers k, k+1, . . . , n form the active submatrix.

The goal of the kth elimination step is to eliminate (nullify) subdiagonal elements of the kth column $u_{k+1,k}, \dots, u_{nk}$.

At first the `gel` routine chooses the pivot element in the active submatrix. Let's assume that the pivot element chosen on the kth elimination step is $u_{pq}, k \le p, q \le n$. So, it is necessary to permute rows of the matrix U with numbers k and p and columns with numbers k and q in order to move the pivot element to the position u_{kk}. Since the matrix U has implicit representation $U = PVQ$, no actual permutations are performed. Instead that the routine actually permutes kth and pth rows of the matrix P, and kth and qth

columns of the matrix Q. Note that the order of rows and columns of the matrix V remain unchanged from the beginning to the end of elimination process.

In order to eliminate a particular subdiagonal element $u_{ik}, k+1 \le i \le n$ the routine applies elementary gaussian transformation to the matrix U, which consists of subtracting kth (pivot) row multiplied by gaussian factor $f_{ik} = u_{ik}/u_{kk}$ from ith (transformed) row. Thus, elements of ith row of the matrix U are transformed using the formula:

$$\overline{u}_{ij} = u_{ij} - f_{ik}u_{kj} = u_{ij} - \frac{u_{ik}u_{kj}}{u_{kk}}, \quad j = 1, \ldots, n, \tag{6.2}$$

that gives $\overline{u}_{ik} = 0$. After last elimination step ($k = n$) the matrix U becomes upper triangular.

The process of gaussian elimination is completely defined, if pivot elements are known on each elimination step. It is well known that pivoting is determined by two main factors:

a) small pivot element may cause numerical instability;

b) pivoting affects on appearance of new non-zeros in the matrix U.

Since the `gel` routine is intended for general case (when a priori numerical properties of the matrix U is unknown), it uses so called dynamic pivoting — the next pivot element is determined directly before the corresponding elimination step.

In order to provide numerical stability the `gel` routine chooses the next pivot element only among those elements of the active submatrix, which are satisfied to the following numerical stability condition:

$$|u_{ij}| \ge tol \cdot \max_l |u_{il}|, \tag{6.3}$$

where $0 < tol < 1$ is a given relative tolerance (see below). This condition guarantees that the pivot element is not very small among other elements in the same row (of the active submatrix).

The choice of the pivot element also affects on appearence of new non-zeros in the matrix U. Locally this affection can be estimated in the following way. Let $\nu_r(i)$ is number of non-zeros in ith row and $\nu_c(j)$ is number of non-zeros in jth column. If the pivot element is u_{ij}, number of non-zeros apperaing in the matrix U after kth elimination step is not greater than

$$c(i,j) = [\nu_r(i) - 1][\nu_c(j) - 1], \tag{6.4}$$

where $c(i,j)$ is so called *Markowitz cost* of the element u_{ij} [1] (see Fig. 6.2).

The `gel` routine uses heuristic rule called *Markowitz strategy* [1]. It chooses that element of the active submatrix, which satisfies to the numerical stability condition (6.3) and has possibly minimal Markowitz cost (6.4).

The relative tolerance *tol* in the condition (6.3) is so called Markowitz parameter. Generally this parameter allows improving numerical stability at the expense of sparsity and vice versa. If *tol* is close to 1, the pivot element is close to the maximal element of the pivot row, that provides good numerical stability but decreases freedom for choosing good pivot element to keep sparsity. On the contrary, if *tol* is close to 0, it is possible to choose pivot element for minimizing fill-in, however, such element may be very small that may involve numerical instability.

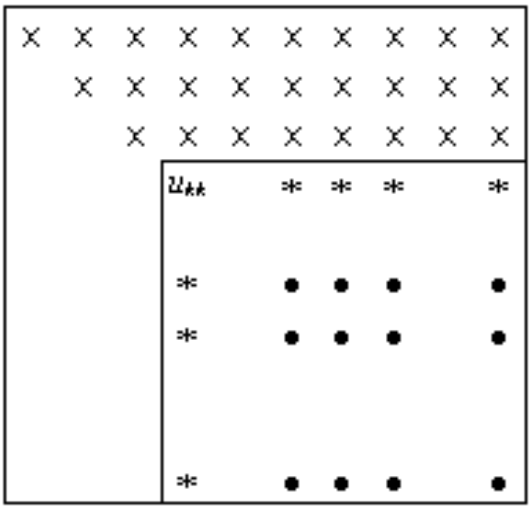

Fig. 6.2. Appearence of new non-zeros in the matrix U on kth elimination step
ukk is the pivot element, '`bullet`' is new non-zero

In practice good results can be obtained using $tol = 0.1$—0.01. However, if the computational process becomes numerically unstable (see below), the tolerance tol should be increased.

In order to look after numerical stability the `gel` routine uses some heuristic test. The computational process is considered as stable, if the following condition is satisfied on each elimination step:

$$Ubig \leq lim \cdot Umax, \tag{6.5}$$

where $Umax$ is maximal of absolute values of elements of the initial matrix U, $Ubig$ is maximal of absolute values of elements of the matrix U, which appeared in this matrix during entire elimination process (i.e. $Ubig = max|u_{ij}^k|$, where k is the number of elimination step), lim is the specified parameter.

If on kth elimination step the condition (6.5) is violated, the `gel` routine returns error code $+k$.

The choice of the parameter lim depends on the working precision of a particular computer. If `DBL_DIG` is approximately 15 decimal places, the recommended value is $lim = 10^{10}$—10^{12}.

Now let's consider some important details of implementation of Markowitz strategy.

In order to find the pivot element, which satisfies to the stability condition (6.3) and has smallest Markowitz cost (6.4) it is necessary to examine all elements of the active submatrix. Although the matrix U is assumed to be sparse, such full pivoting may be very expensive. Therefore the `gel` routine uses partial pivoting based on the search technique proposed by I. Duff [2].

The main idea of *Duff search technique* is based on the reasonable thought that pivot element with smallest Markowitz cost should be probably placed in row and column, which have small number of non-zeros. So, the `gel` routine examines rows and columns of the active submatrix in the order of increasing their counts $\nu_r(i)$ and $\nu_c(i)$. In order to organize efficient examining all active rows and columns the routine splits them into sets (active lists) $R(n) = i|\nu_r(i) = n$ and $C(n) = j|\nu_c(j) = n$, which are implemented as Duff schemes.

The `gel` routine builds these sets[1] before the first elimination step and corrects them each time, when new non-zero is appeared in some row or column or some element becomes zero.

The choice of the pivot element using Duff search technique is performed in the following way. If $C(1) \neq \emptyset$, jth column of the active submatrix, where $j \in C(1)$, has the only non-zero element (column singleton), which the routine immediately chooses as the pivot element. Similarly, if $R(1) \neq \emptyset$, ith row of the active submatrix, where $i \in R(1)$, has the only non-zero element (row singleton), which the routine immediately chooses as the pivot element. (Note that singletons give no fill-in.) Otherwise, if $C(1) = \emptyset$and $R(1) = \emptyset$, the `gel` routine continues examining elements in rows and columns from sets $C(2)$, $R(2)$, $C(3)$, $R(3)$, etc., storing those element, which satisfies to the stability condition (6.3) and has the smallest Markowitz cost among other examined elements. In order not to examine all rows and columns from sets up to $C(n)$ and $R(n)$ the routine uses the following termination heuristics proposed by I. Duff [2]: the examination is terminated if all columns from the set $C(\nu)$ have been examined and $c^* \leq \nu(\nu - 1)$, or all rows from the set $R(\nu)$ have been examined and $c^* \leq (\nu - 1)^2$, where c^* is best Markowitz cost that has been reached for some examined element, which is also satisfied to the stability condition. This heuristics can be easily grounded. In fact, if one of mentioned conditions for c^* is satisfied, it is doubtful that there is other element with better Markowitz cost.

6.1.2 `gel_bg` — transform sparse matrix of special structure to upper triangular form using Bartels and Golub technique

Synopsis

```
#include "glpgel.h"
int gel_bg(PER *P, MAT *V, PER *Q, int k1, int k2,
   void (*func)(int i, int k, double f), double tol, double eps,
   int *Unz, double work[]);
```

Description

The `gel_bg` routine transforms the sparse matrix $U = PVQ$ of special structure (see below), where V is the given sparse matrix, P and Q are given permutation matrices, to the upper triangular form using the version of gaussian elimination proposed by R. H. Bartels and G. H. Golub [3, 4]. The result of transformation is the upper triangular matrix $U' = P'V'Q'$, where P', V', and Q' are resultant matrices placed instead matrices P, V, and Q respectively.

The `gel_bg` routine assumes that the initial matrix $U = PVQ$ has the special structure[2] shown on Fig. 6.3. Thus, only the submatrix of U formed by rows $k_1, k_1 + 1, \ldots, k_2$ and columns $k_1, k_1 + 1, \ldots, n$ should be transformed (elements of this submatrix are marked by $*$, other elements of the initial matrix U, are marked by $\times$).

[1] Actually sets $R(n)$ and $C(n)$ contain numbers of rows and columns of the matrix V, not of the matrix U, in order to avoid explicit permutations of rows and columns.

[2] Such matrices usually appear in the simplex method of linear programming after replacing a column of upper triangular matrix by other column.

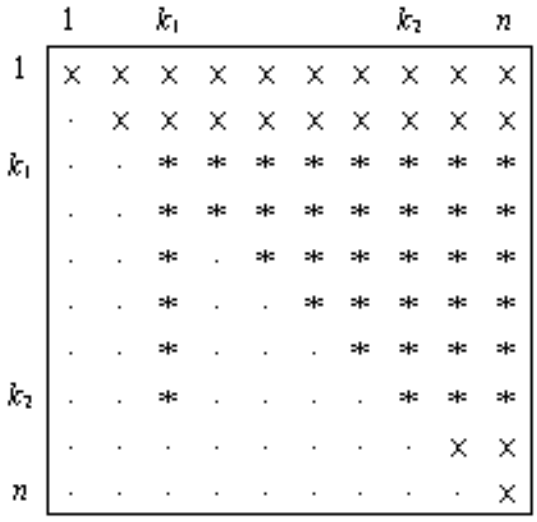

Fig 6.3. The special structure of the initial matrix $U = PVQ$

The matrix U, which is transformed to the final upper triangular matrix U', has implicit representation, i.e. all operations are actually performed on the matrix V, which differs from U only in the order of rows and columns determined by permutation matrices P and Q (cf. also Subsection 6.1.1). This implicit representation allows not to explicitly permute rows and columns during elimination and thereby allows improving the efficiency of computational process.

The example of transformation of a sparse matrix ($n = 97$) performed by means of the *gel_bg* routine is shown on Fig. 6.4.

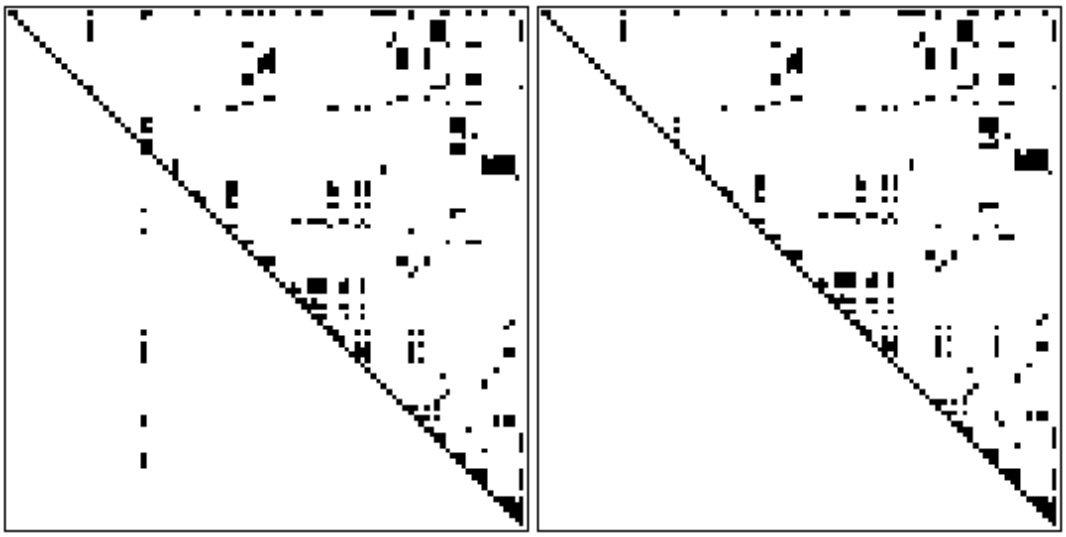

Fig 6.4. Example of transformation performed by the `gel_bg` routine (input matrix $U = PVQ$ and resultant matrix $U' = P'V'Q'$).

In order to eliminate subdiagonal elements of the matrix $U = PVQ$ the routine applies to the matrix U elementary transformations of the following three kinds: row permutation,

column permutation, and subtraction one row multiplied by a number from other row (gaussian transformation). Each time, when gaussian transformation is applied to the matrix U (in order to eliminate one of its subdiagonal elements), the `gel_bg` routine calls the formal routine `func` in order to allow the calling program accumulating information about all performed gaussian transformations (information about row and column permutations is accumulated in the matrices P and Q by the `gel_bg` routine and also available to the calling program).

The point, where the func routine is called, and the meaning of parameters passed to that routine are explained below in the method description.

The control parameter *tol* is a relative tolerance used for threshold pivoting. Its value should be in the range $0 <$ *tol* < 1. For more details see the method description below.

The control parameter *eps* is a relative tolerance for diagonal elements of the matrix U'. For more details see the method description below.

On entry the location `Unz` should contain total number of non-zeros in the initial matrix U. On exit this location contains total number of non-zeros in the resultant matrix U'. (If on entry the location `Unz` contains zero, on exit it contains the fill-in count, i.e. the difference between numbers of non-zeros in the matrices U' and U.)

The array `work` is working storage. It should have at least $1+n$ locations, where n is order of the matrix V.

Returns

The `gel_bg` routine returns one the following error codes:

0	no errors;
$-k$	if $k_1 \leq k < k_2$, on kth elimination step elements u_{kk} and $u_{k+1,k}$ are equal to zero. If $k = k_2$, after last elimination step element u_{k_2,k_2} is equal to zero;
$+k$	if $k_1 \leq k < k_2$, on kth elimination step the numerical stability condition is violated (see below the method description). If $k = k_2$, this condition is violated for element u_{k_2,k_2}.

Method

The `gel_bg` routine is based on the version of gaussian elimination proposed by R. H. Bartels and G. H. Golub [3, 4], which exploits the special structure of the matrix U.

Before elimination the routine shifts columns $k_1 + 1, k_1 + 2, \ldots, k_2$ by one position to the left and also moves the column k_1 to the position k_2 (in the original Bartels & Golub version $k_2 = n$, because $k_2 < n$ reduces the freedom of pivoting and thereby involves worsening numerical stability). Due to these column permutations the matrix U becomes upper Hessenberg matrix shown on Fig. 6.5 (elements that should be eliminated are marked by '`bullet`').

In order to fully exploit the special structure of the matrix U the `gel_bg` routine uses partial pivoting only in the active column. So far as on each elimination step the active column contain the only non-zero subdiagonal element, the choice is limited by two elements: u_{kk} and $u_{k+1,k}$.

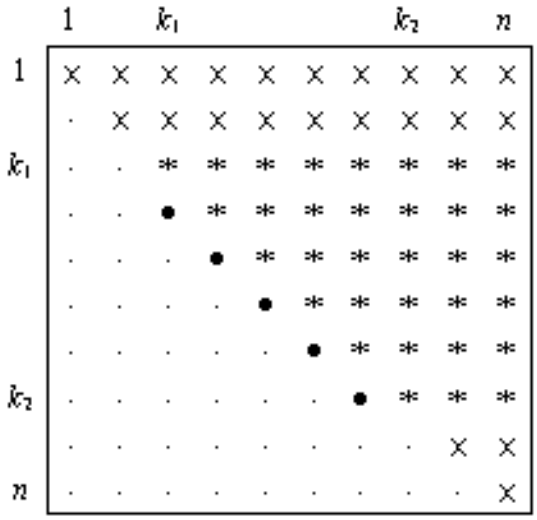

Fig 6.5. The matrix U after column permutations

In the original Bartels & Golub version the largest (in absolute value) element is used as the pivot element, because this guarantees good numerical stability. However, the `gel_bg` routine tries to reduce fill-in at the expense of some worsening numerical stability. The pivoting rule, which is used in the `gel_bg` routine, is based on the following heuristics[3], where the parameter *tol* ($0 < tol < 1$) is the given relative tolerance:

a) if $|u_{k+1,k}| < tol \cdot |u_{kk}|$, the routine chooses the element u_{kk};

b) if $|u_{kk}| < tol \cdot |u_{k+1,k}|$, the routine chooses the element $u_{k+1,k}$;

c) in other cases the routine chooses that element, which is placed in the row with smaller number of non-zeros.

(Note that in the original Bartels & Golub version $tol = 1$.)

Thus, the model scheme of the elimination process is the following:

```
/* permute columns of the matrix U */
...;
for (k = k1; k < k2; k++)
{  /* k-th elimination step starts here */
   /* choose the pivot element u[k,k] or u[k+1,k] */
   ...;
   /* if u[k+1,k] has been chosen, permute rows k and k+1 of the
      matrix U */
   if (...) ...;
   /* now the pivot element is u[k,k] */
   assert(u[k,k] != 0.0);
   /* eliminate non-zero subdiagonal element u[k+1,k] */
   i = k+1;
   if (u[i,k] == 0.0) continue;
   /* compute gaussian multiplier */
   f = u[i,k] / u[k,k];
   /* (i-th row of U) := (i-th row of U) - f * (k-th row of U) */
   ...;
```

[3] This heuristics is Markowitz strategy applied to this particular case.

```
      assert(u[i,k] == 0.0);
      /* i-th row of the matrix U has been transformed */
      func(i, k, f);
   }
   assert(u[k2,k2] != 0.0);
```

Note that row numbers passed to the `func` routine correspond to the matrix U, not to V. The correspondence between row and column numbers of the matrices $U = PVQ$ and V is determined by the row permutation matrix P and the column permutation matrix Q. Thus, if an element $u[i,j]$ of the matrix U corresponds to an element $v[i',j']$ of the matrix V, the following formulae may be used:

```
i' = P->row[i], j' = Q->col[j];
i = P->col[i'], j = Q->row[j'];
```

(these formulae are valid on exit as well as during all the elimination process).

In order to look after numerical stability the `gel_bg` routine checks the following condition on each elimination step, where *eps* is the given relative tolerance:

$$u'_{k,k} \geq eps \cdot \max_j |u'_{k,j}| > 0,$$

i.e. each diagonal element should be non-zero and should not be very small among other elements in the same row. (Note that this condition is checked only for rows $k_1, k_1 + 1, \ldots, k_2$).

Some experimental results show that:

1. Fairly good numerical stability (even in case of bad scaled matrices) can be provided using $tol = 0.2$—0.5. Lower values of *tol* usually involve excessive round-off errors and as a consequence numerical instability.

2. The choice of the parameter *eps* depends on the working precision of a particular computer. If `DBL_DIG` is approximately 15 decimal places, the recommended value is $eps = 10^4$—10^6.

6.1.3 `gel_ft` — transform sparse matrix of special structure to upper triangular form using Forrest and Tomlin technique

Synopsis

```
#include "glpgel.h"
int gel_ft(PER *P, MAT *V, PER *Q, int k1, int k2,
   void (*func)(int i, int k, double f), double eps, int *Unz,
   double work[]);
```

Description

The `gel_ft` routine transforms the sparse matrix $U = PVQ$ of special structure (see below), where V is the given sparse matrix, P and Q are given permutation matrices, to the upper triangular form using the version of gaussian elimination proposed by J. J. H. Forrest and J. A. Tomlin [5]. The result of transformation is the upper triangular

matrix $U' = P'V'Q'$, where P', V', and Q' are resultant matrices placed instead matrices P, V, and Q respectively.

The `gel_ft` routine assumes that the initial matrix $U = PVQ$ has the special structure[4] shown on Fig. 6.6. Thus, only the submatrix of U formed by rows $k_1, k_1 + 1, \ldots, k_2$ and columns $k_1, k_1 + 1, \ldots, n$ should be transformed (elements of this submatrix are marked by $*$, other elements of the initial matrix U, are marked by $\times$).

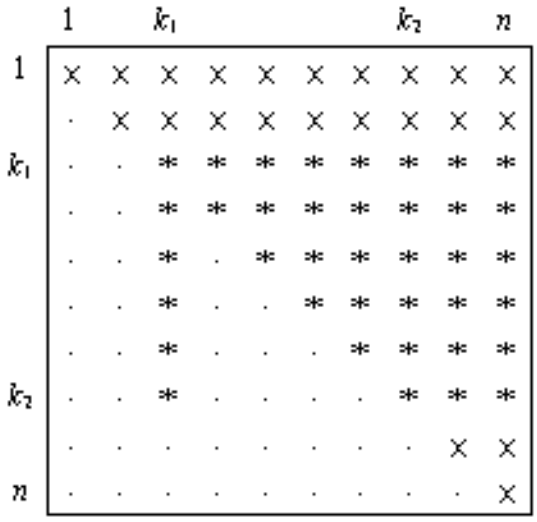

Fig 6.6. The special structure of the initial matrix $U = PVQ$

The matrix U, which is transformed to the final upper triangular matrix U', has implicit representation, i.e. all operations are actually performed on the matrix V, which differs from U only in the order of rows and columns determined by permutation matrices P and Q (cf. also Subsection 6.1.1). This implicit representation allows not to explicitly permute rows and columns during elimination and thereby allows improving the efficiency of computational process.

The example of transformation of a sparse matrix ($n = 97$) performed by means of the `gel_ft` routine is shown on Fig. 6.7.

(Obviously, the `gel_ft` routine solves the same problem as the `gel_bg` routine described in Subsection 6.1.2.)

In order to eliminate subdiagonal elements of the matrix $U = PVQ$ the routine applies to the matrix U elementary transformations of the following three kinds: row permutation, column permutation, and subtraction one row multiplied by a number from other row (gaussian transformation). Each time, when gaussian transformation is applied to the matrix U (in order to eliminate one of its subdiagonal elements), the `gel_ft` routine calls the formal routine `func` in order to allow the calling program accumulating information about all performed gaussian transformations (information about row and column permutations

[4] Such matrices usually appear in the simplex method of linear programming after replacing a column of upper triangular matrix by other column.

is accumulated in the matrices P and Q by the `gel_ft` routine and also available to the calling program).

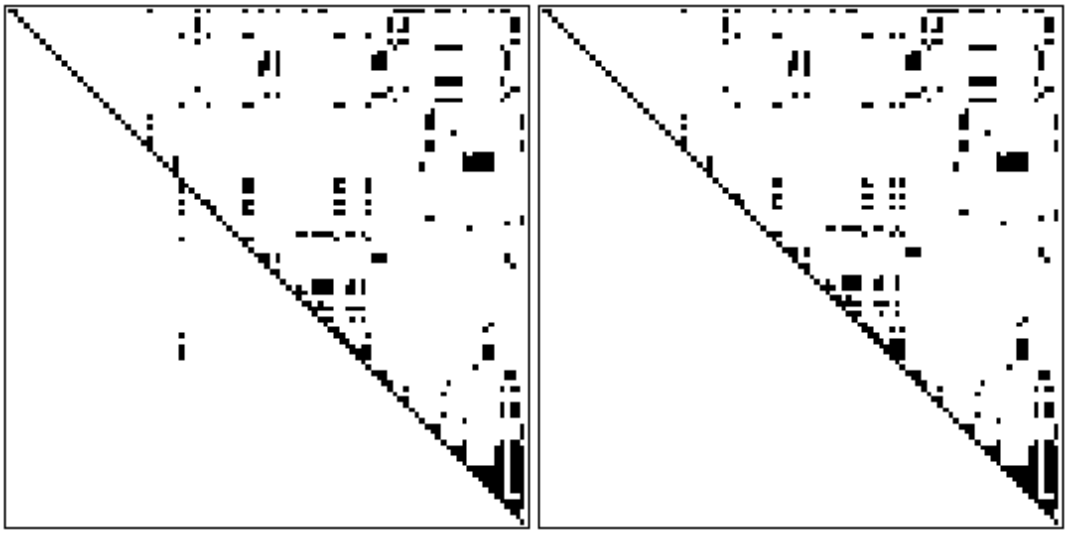

Fig 6.7. Example of transformation performed by the `gel_ft` routine (input matrix $U = PVQ$ and resultant matrix $U' = P'V'Q'$).

The point, where the `func` routine is called, and the meaning of parameters passed to that routine are explained below in the method description.

The control parameter *eps* is a relative tolerance for diagonal elements of the matrix U'. For more details see the method description below.

On entry the location `Unz` should contain total number of non-zeros in the initial matrix U. On exit this location contains total number of non-zeros in the resultant matrix U'. (If on entry the location `Unz` contains zero, on exit it contains the fill-in count, i.e. the difference between numbers of non-zeros in the matrices U' and U.)

The array `work` is working storage. It should have at least $1+n$ locations, where n is order of the matrix V.

Returns

The `gel_ft` routine returns one the following error codes:

0 — no errors;

$-k$ — if $k_1 \le k < k_2$, on kth elimination step elements u_{kk} and $u_{k+1,k}$ are equal to zero. If $k = k_2$, after last elimination step element u_{k_2,k_2} is equal to zero;

$+k$ — if $k_1 \le k < k_2$, on kth elimination step the numerical stability condition is violated (see below the method description). If $k = k_2$, this condition is violated for element u_{k_2,k_2}.

Method

The `gel_ft` routine is based on the version of gaussian elimination proposed by J. J. H. Forrest and J. A. Tomlin [5], which exploits the special structure of the matrix U.

Before elimination the routine shifts columns $k_1+1, k_1+2, \ldots, k_2$ by one position to the left and upwards, and also moves the row and column k_2 to the position k_1. Due to these row and column permutations the matrix U acquires the form shown on Fig. 6.8. Then the routine applies gaussian elimination in order to eliminate (nullify) subdiagonal elements of the matrix U.

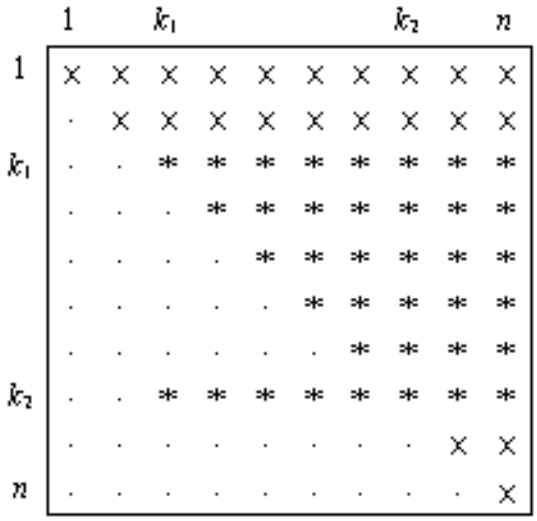

Fig 6.8. The matrix U after row and column permutations

Forrest & Tomlin version of gaussian elimination assumes no pivoting. This means that on kth elimination step ($k = k_1, k_1+1, \ldots, k_2-1$) the pivot element is always u_{kk}, which is used to eliminate the only subdiagonal element $u_{k_2,k}$ (of course, if it is non-zero).

Thus, Forrest—Tomlin technique makes no fill-in of the matrix U. On the other hand this involves worsening numerical stability in contrast to other methods, which use pivoting.

Note that no pivoting is a key idea in Forrest—Tomlin technique, because this allows keeping the special structure of the matrix U during entire elimination process. In fact, if we would provide Forrest—Tomlin technique with, for example, partial pivoting in the active column, it would turn into Bartels—Golub technique (see Subsection 6.1.2). May note that Forrest—Tomlin technique can be considered as a version of Bartels—Golub technique, where rows k and $k+1$ are permuted on each elimination step.

Thus, the model scheme of the elimination process is the following:

```
/* permute symmetrically rows and columns of the matrix U */
...;
for (k = k1; k < k2; k++)
{  /* k-th elimination step starts here */
   /* the pivot element is always u[k,k] */
```

```
      assert(u[k,k] != 0.0);
      /* eliminate non-zero subdiagonal element u[k2,k] */
      i = k2;
      if (u[i,k] == 0.0) continue;
      /* compute gaussian multiplier */
      f = u[i,k] / u[k,k];
      /* (i-th row of U) := (i-th row of U) - f * (k-th row of U) */
      ...;
      assert(u[i,k] == 0.0);
      /* elementary transformation of i-th row of the matrix U has been
         performed */
      func(i, k, f);
   }
   assert(u[k2,k2] != 0.0);
```

Note that row numbers passed to the `func` routine correspond to the matrix U, not to V. The correspondence between row and column numbers of the matrices $U = PVQ$ and V is determined by the row permutation matrix P and the column permutation matrix Q. Thus, if an element $u[i,j]$ of the matrix U corresponds to an element $v[i',j']$ of the matrix V, the following formulae may be used:

```
i' = P->row[i], j' = Q->col[j];
i = P->col[i'], j = Q->row[j'];
```

(these formulae are valid on exit as well as during all the elimination process).

So far as the only row k_2 is being changed in Forrest—Tomlin version, in order to check for numerical stability the `gel_ft` routine uses simplified test. The transformation is considered as satisfactory if the following condition is true after the transformation has been finished:

$$|uk_2, k_2| \geq eps \cdot big,$$

where eps is the given relative tolerance, $big = \max_{k,j} |u^k_{k_2,j}|$ is maximal of absolute values of elements, which appeared in the matrix U during entire elimination process (k is number of the elimination step).

Note that the `gel_ft` routine detects zero diagonal element u_{kk} only in that case if $u_{k_2,k} \neq 0$, i.e. if elementary gaussian transformation is actually needed.

The choice of the tolerance eps depends on the working precision of a particular computer. If `DBL_DIG` is approximately 15 decimal places, the value $eps = 10^{-4}$—10^{-6} may be recommended.

6.1.4 `min_bump` — minimize bump of sparse matrix using Reid technique

Synopsis

```
#include "glpgel.h"
void *min_bump(PER *P, MAT *V, PER *Q, int *k1, int *k2,
   DUFF *rs, DUFF *cs, int prev[], int next[]);
```

Description

The `min_bump` routine minimizes the bump of the sparse matrix $U = PVQ$ of special structure (see below), where V is the given sparse matrix, P and Q are given permutation matrices, implicitly permuting its rows and columns by means of the structural transformation proposed by J. K. Reid [6]. The result of transformation is the matrix $U' = P'VQ'$ with minimal bump, where P' and Q' are resultant permutation matrices placed instead matrices P and Q respectively (the matrix V is not changed on exit).

This transformation may be used before subsequent transforming the matrix $U = PVQ$ using Bartels—Golub or Forrest—Tomlin methods (see Subsections 6.1.2 and 6.1.3) in order to improve sparsity.

The `min_bump` routine assumes that the initial matrix $U = PVQ$ has the special structure shown on Fig. 6.9, where submatrix formed by rows and columns $k_1, k_1+1, ..., k_2$ is so called *bump*. Thus, the matrix U differs from upper triangular matrix only in the column k_1, which is called *spike*. Additionally it is assumed that all diagonal elements of the matrix U (except may be the element u_{k_1,k_1}) are non-zero. The element u_{k_2,k_1} also should be non-zero, otherwise the bump could be decreased. (On Fig. 6.9 elements of bump are marked by $*$, other elements are marked by $\times$.)

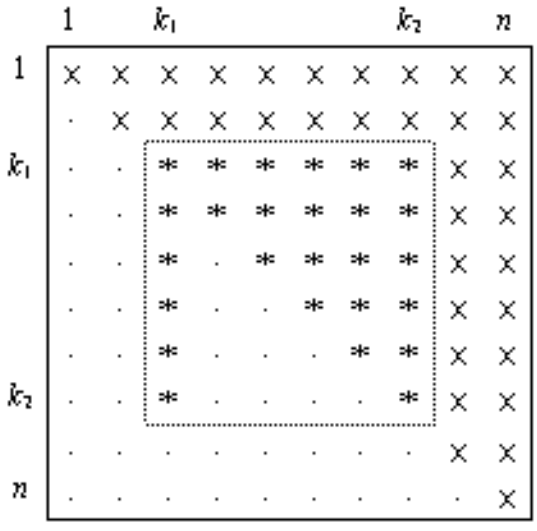

Fig 6.9. The initial matrix $U = PVQ$

The `min_bump` routine performs symmetric row and column permutations in order that the final matrix $U = P'VQ'$ of the same special structure has the bump of minimal size (the routine accumulates these permutations in matrices P and Q, so the matrix V remain unchanged). On exit the routine stores bounds of the final bump k_1 and k_2 to the locations `k1` and `k2` respectively (the case $k_1 = k_2$ means that the final matrix U' became of upper triangular form with non-zero diagonal elements, except may be the element u_{k_1,k_1}, which is the final bump in this case).

The example of structural transformation of a sparse matrix ($n = 97$) performed by means of the `min_bump` routine is shown on Fig. 6.10.

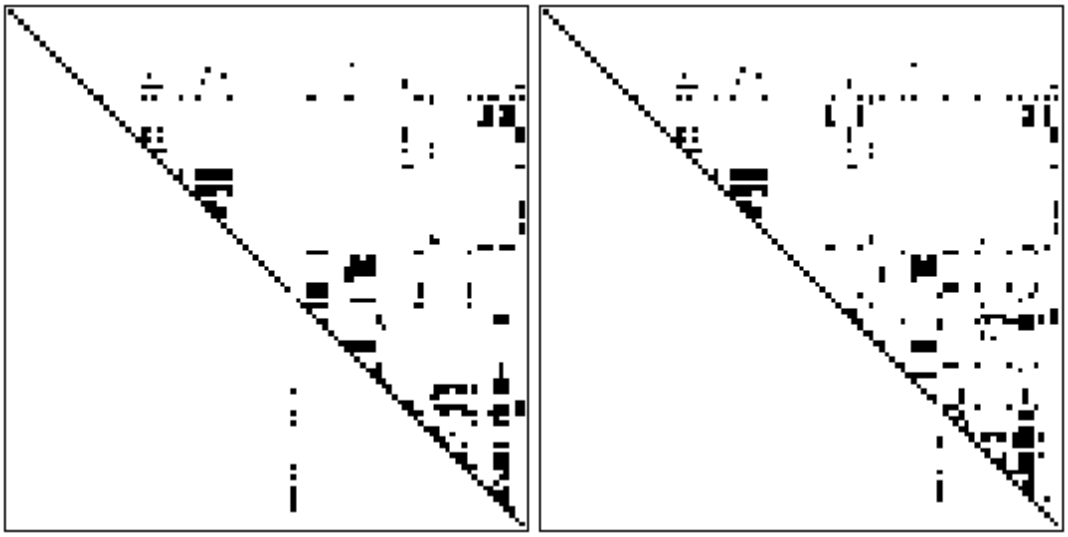

Fig 6.10. Example of structural transformation performed by the `min_bump` routine input matrix $U = PVQ$ and resultant matrix $U' = P'VQ'$.

The parameters `rs` and `cs` should specify Duff schemes (see Chapter 3), which are used by the `min_bump` routine to keep lists of active (i.e. that belong to the bump) rows and columns (see the method description). On entry contents of these schemes is ignored. On exit these schemes contain rows and columns of the matrix U (not V!) that belong to the final bump. These schemes should be created by statements `rs = create_duff(n, n)` and `cs = create_duff(n, n)`, where n is order of the matrix U (which is the same as of the matrix V).

The arrays `prev` and `next` are working storage. They should have at least $1+n$ locations, where n is order of the matrix U.

Method

Let the current bump contains a column, which is not the spike and which has the only non-zero element u_{kk} (column singleton). Obviously, this non-zero element is on the main diagonal. Therefore the `min_bump` routine shifts rows and columns $k_1, k_1 + 1, \ldots, k - 1$ to the right and downward by one position and moves row and column k to the position k_1.

Such symmetric transformation keeps the special structure of the matrix U and decreases its bump size by one (Fig. 6.11).

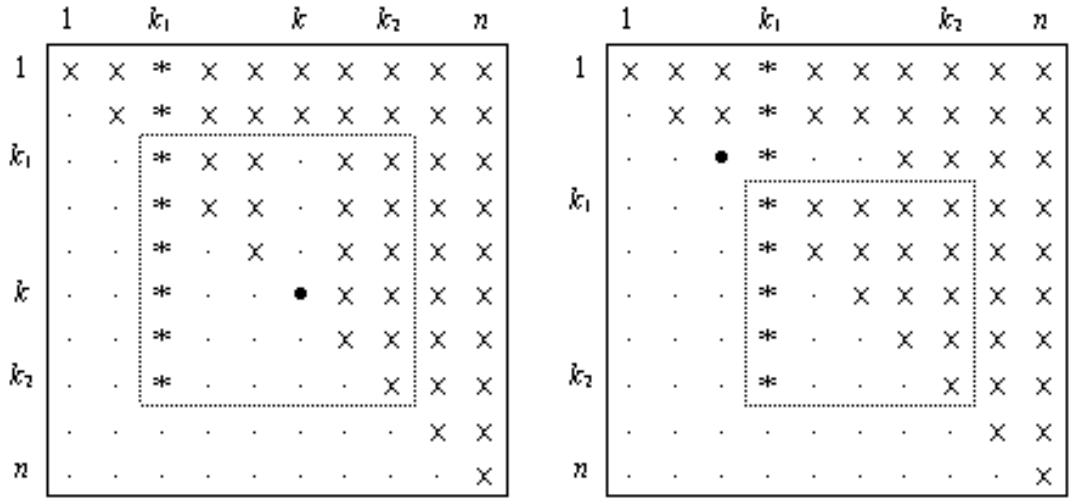

Fig 6.11. Symmetric transformation in case of column singleton

Similarly, if the current bump contains a row, which is not symmetric to the spike and has the only non-zero element u_{kk} (row singleton) on the main diagonal, the `min_bump` routine shifts rows and columns $k+1, k+2, \ldots, k_2$ to the left and upward by one position and moves row and column k to the position k_2. Such symmetric transformation also keeps special structure of the matrix U and again decreases its bump size by one (Fig. 6.12).

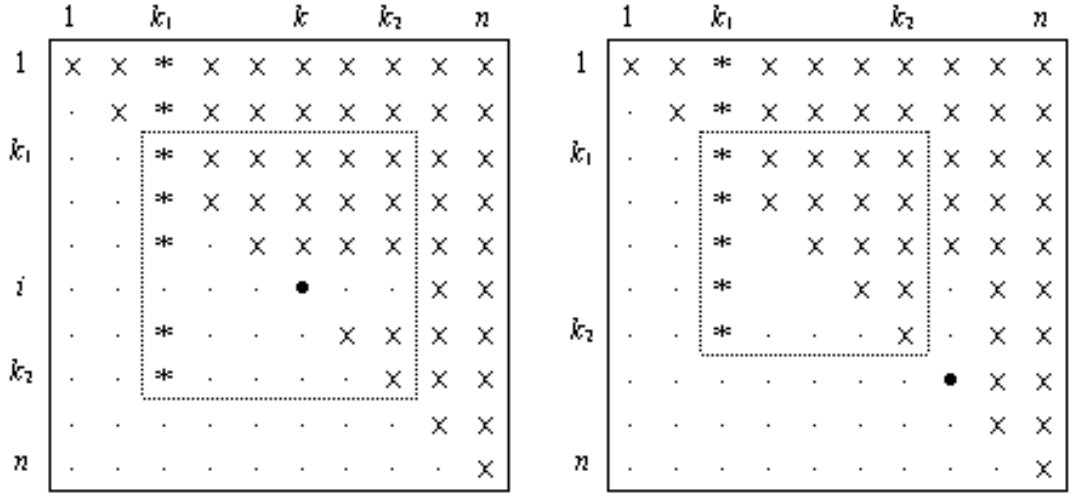

Fig 6.12. Symmetric transformation in case of row singleton

So far as removing row or column from the bump may cause new singletons to appear in the bump, the `min_bump` routine repeats these structural transformations until it is possible. It may be easily shown that in fact the final bump has the smallest possible size.

Now let's consider some important implementation details. Explicit row and columns permutations of the matrix U (which has implicit representation $U = PVQ$) is very expensive, because one shift of row or column requires up to n transpositions, i.e. up to n movements of elements of matrices P and Q. Therefore the `min_bump` routine performs all elementary transformations using other matrix $\overline{U}$, which differs from the matrix U only in the order of rows and columns. For representing the matrix $\overline{U}$ the routine doesn't use permutation matrices (like P and Q); instead that it uses double linked list, elements of which are numbers of rows and columns of the matrix U. In the linked list these elements are placed in the order, which reflects the order of the corresponding rows and columns in the matrix U. (Only one linked list is needed, because all performed structural transformations are symmetric.) Initially $\overline{U}$ is identical to U. After the matrix U has been completely transformed, the `min_bump` routine uses the linked list for reordering rows and columns of the matrix U. This technique allows performing one movement of rows and columns (of the matrix $\overline{U}$) for a fixed time, because one movement is equivalent to one exclusion from and one inclusion to the linked list. This gives time estimation $O(nz)$, where nz is number of non-zeros in the matrix U.

In order to find row or column singleton for a fixed time the routine uses Duff search technique (see Chapter 3).

6.2 LU Factorization

This section describes a set of routines, which are based on sparse gaussian elimination routines (see Section 6.1) and implement the following LU factorization of a given (quadratic) matrix A:

$$A = PLUQ, \tag{6.6}$$

where L is lower triangular matrix with unit main diagonal, U is upper triangular matrix, P and Q are permutation matrices.

Permutation matrices P and Q are introduced in order to allow permuting row and column during factorization for improving numerical stability by means of partial pivoting. Being computed the factorization (6.6) may be considered as if it would be computed for the matrix $\overline{A} = P^{-1}AQ^{-1}$ (which differs from the original matrix A only in the order of rows and columns) with no row and column permutations, because $\overline{A} = LU$.

LU factorization (6.6) is represented by the LU structure:

```
typedef struct LU LU;
struct LU
{     int n;
      PER *P;
      MAT *L;
      MAT *U;
      PER *Q;
};
```

where:

`n` order of matrices P, L, U, and Q ($n > 0$);

`P` left permutation matrix P;

`L` lower triangular matrix L with unit main diagonal;

`U` upper triangular matrix U;

`Q` right permutation matrix Q.

6.2.1 `create_lu` — create LU factorization

Synopsis

```
#include "glpgel.h"
LU *create_lu(int n);
```

Description

The `create_lu` routine creates LU factorization of order $n > 0$. Initially the factorization corresponds to unity matrix, i.e. $P = L = U = Q = I$.

Returns

The `create_lu` routine returns a pointer to the created LU factorization.

6.2.2 `build_lu` — compute LU factorization of given sparse matrix

Synopsis

```
#include "glpgel.h"
int build_lu(LU *lu, MAT *A, double tol, double lim,
   void (*monit)(LU *lu, int kase, int i, int k, double f),
   double *Amax, double *Ubig);
```

Description

The `build_lu` routine computes LU factorization (6.6) of the given sparse matrix A. In order to compute LU factorization the `build_lu` routine uses the `gel` routine, which performs sparse gaussian elimination (see Subsection 6.1.1).

The LU factorization, which `lu` points to, should be created before by the `create_lu` routine (see Subsection 6.2.1) and may be used several times for different matrices of the same order to factorize them by means of the `build_lu` routine.

It is allowed to pass the matrix `lu->U` as the input matrix A. Otherwise the matrix A remains unchanged on exit.

The control parameter *tol* is a relative tolerance (Markowitz parameter) used for threshold pivoting. Its value should be in the range $0 < tol < 1$. An element u_{ij} of the active submatrix fits to be the pivot element (from the point of view of numerical stability) if $u_{ij} \geq tol \cdot \max_l |u_{il}|$ on some elimination step. Decreasing *tol* usually involves improving

sparsity at the expense of accuracy and vice versa. If the input matrix A is expected to be well conditioned, the value $tol = 0.1$—0.01 may be recommended in most cases. For more hard matrices the value of tol should be increased. For details see the description of the `gel` routine (Subsection 6.2.1).

The control parameter *lim* specifies maximal allowable growth of elements of the matrix U during factorization process. The process is considered as numerically stable if on each step the condition $Ubig \leq lim \cdot Amax$ is satisfied ($Amax$ and $Ubig$ are described below). If `DBL_DIG` is approximately 15 decimal places, the recommended value is $lim = 10^{10}$—10^{12}. For details see the description of the `gel` routine (Subsection 6.2.1).

The formal routine `monit` is intended for monitoring purposes. The `build_lu` routine calls the `monit` routine in the following cases (recognized by the parameter `kase`):

`kase = 0` once before the beginning of factorization (parameters `i`, `k`, and `f` should be ignored);

`kase = 1` each time when the next elementary gaussian transformation is applied to the matrix U using the following formula:

$$(i\text{th row of } U) := (i\text{th row of } U) - f \times (k\text{th row of } U),$$

where i is number of transformed row, k is number of pivot row. As a result of this elementary transformation the next non-zero element appears in the matrix L. Note that during factorization matrices `lu->L` and `lu->U` differ from matrices L and U as declared by (6.6) in the order of rows and columns, which is determined by permutation matrices P and Q. This is used to avoid explicit permutations of rows and columns of matrices L and U. For details see the method description;

`kase = 2` once after the end of factorization (parameters `i`, `k`, and `f` should be ignored);

There is standard monitoring routine `monit_lu` (see Subsection 6.2.3), which may be passed to the `build_lu` routine. Every second the `monit_lu` routine prints the following information: n (order of the matrix A), k (number of elimination step), $nz(L)$ (current number of non-zeros in the matrix L), and $nz(U)$ (current number of non-zeros in the matrix U).

If the parameter `monit` is `NULL`, no monitoring is performed.

On entry contents of the locations `Amax` and `Ubig` are ignored. On exit the location `Amax` contains maximal of absolute values of elements of the input matrix A. The location `Ubig` contains maximal of absolute values of elements of the matrix U, which appeared in this matrix during entire elimination process (i.e. $Ubig = \max |u_{ij}^{(k)}|$, where k is number of the elimination step). Thus, the ratio $Ubig/Amax$ may be used to measure the numerical "quality" of computed factorization.

Factorization process may fail if the input matrix A is singular or ill conditioned. In this case matrices L and U correspond to partial factorization and look like shown on Fig. 6.13 (where k is number of factorization step, $*$ marks the active submatrix that causes a problem).

Note that the equality $A = PLUQ$ is valid even in case of error, therefore this partial factorization may be used for performing further analysis of the matrix A.

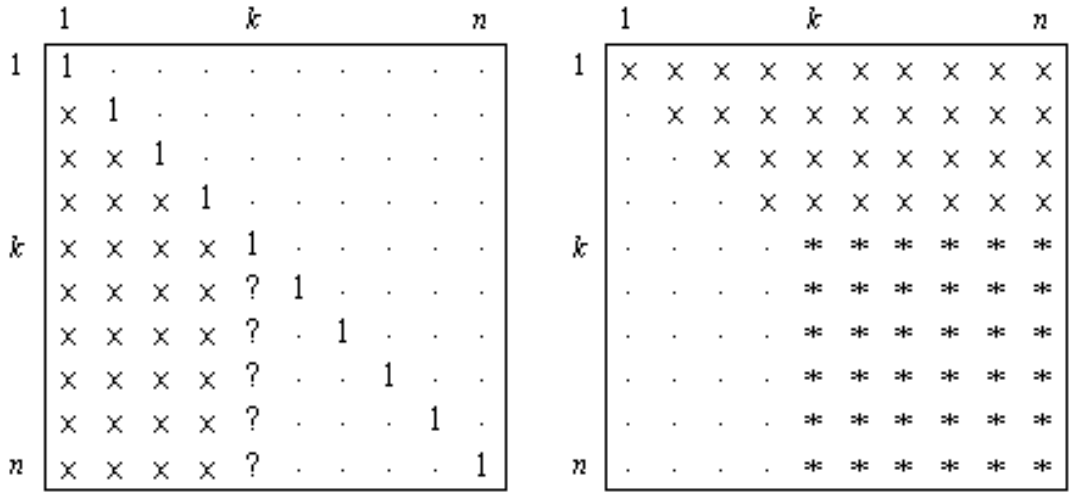

Fig 6.13. Matrices L and U in case of error (partial factorization)

Returns

The `build_lu` routine returns one of the following error codes:

0 no errors, factorization has been successfully computed;

$-k$ on kth elimination step all elements of the active submatrix are equal to zero;

$+k$ on kth elimination step the numerical stability condition is violated.

Method

In order to factorize the input matrix A the `build_lu` routine calls the `gel` routine, which allows transforming matrix to upper triangular form using gaussian elimination (see Subsection 6.1.1).

If the `build_lu` routine would use the form (6.6) directly, it should explicitly permute rows and columns of L each time when the `build_lu` routine (implicitly) permute rows and columns of U. Explicit permutations of rows and columns are undesirable because of low efficiency. Therefore during factorization process the `build_lu` routine instead (6.6) uses the following form:

$$A = MV, \tag{6.7}$$

where $M = PLP^{-1}$ and $V = PUQ$. Thus, (6.7) is equivalent to (6.6), where M and V differ from L and U in the order of rows and columns determined by permutation matrices P and Q. After the factorization (6.7) has been computed the `build_lu` routine replaces matrices M and V by matrices L and U, that gives the final factorization (6.6). Note that during the factorization process matrices $\overline{P} = P^{-1}$ and $\overline{Q} = Q^{-1}$ are actually used instead matrices P and Q, because the `gel` needs the form $U = \overline{P}V\overline{Q}$ instead $V = PUQ$. These permutation matrices are inverted after the factorization is finished.

At first the `build_lu` routine sets $M = I$ and $V = A$, and then calls the `gel` routine in order to transform the matrix $U = \overline{P}V\overline{Q}$ to upper triangular form (the `gel` routine initially sets $\overline{P} = \overline{Q} = I$, so the equation (6.7) is kept). During factorization the `build_lu` routine accumulates all inverted elementary transformation, which the `gel` routine applies to the matrix U (actually to the matrix V, because U has implicit representation), in

the form of the matrix M through the routine `func` passed to the `gel` routine as formal parameter. Finally the `build_lu` routine inverts permutation matrices $\overline{P}$ and $\overline{Q}$ computed by the `gel` routine that gives matrices P and Q, and builds resultant matrices $L = P^{-1}MP$ and $U = P^{-1}VQ^{-1}$

Let's assume that the `gel` routine applies to the matrix `U` elementary gaussian transformation using the formula (6.2) in order to eliminate the subdiagonal element u_{ik}. This elementary transformation may be written as $\overline{U} = \Phi U$, where

$$\Phi = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & -f_{ik} & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}, \tag{6.8}$$

is a transforming matrix, f_{ik} is gaussian multiplier. In order to keep the equality (6.6) it is necessary to apply the corresponding inverted transformation to the matrix L, i.e. $\overline{L} = L\Phi^{-1}$, where

$$\Phi^{-1} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & +f_{ik} & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}, \tag{6.9}$$

It is well known that if gaussian elimination is performed by rows (as the `gel` routine does), the transformation (6.9) involves appearing the only new non-zero element $l_{ik} = f_{ik}$ in the matrix L, i.e. no actual matrix multiplication is needed.

As it was said above the `build_lu` works with matrices M and V, which differ from matrices L and U only in the order of rows and columns. Obviously, transforming L and U using (6.8) and (6.9) is equivalent to transforming M and V using transforming matrices, which has the same structure as (6.8) and (6.9) and differ only in the order of rows and columns. Therefore each time when the `gel` routine applies elementary gaussian transformation, the `build_lu` routine just adds the next non-zero element f_{ik} to the matrix M, determining its position in the matrix M by means of current permutation matrices $\overline{P}$ and $\overline{Q}$.

The example of LU factorization of the test matrix $D(100, 30)$ (see Section 7.28) performed by means of the `build_lu` routine is shown on Fig. 6.14.

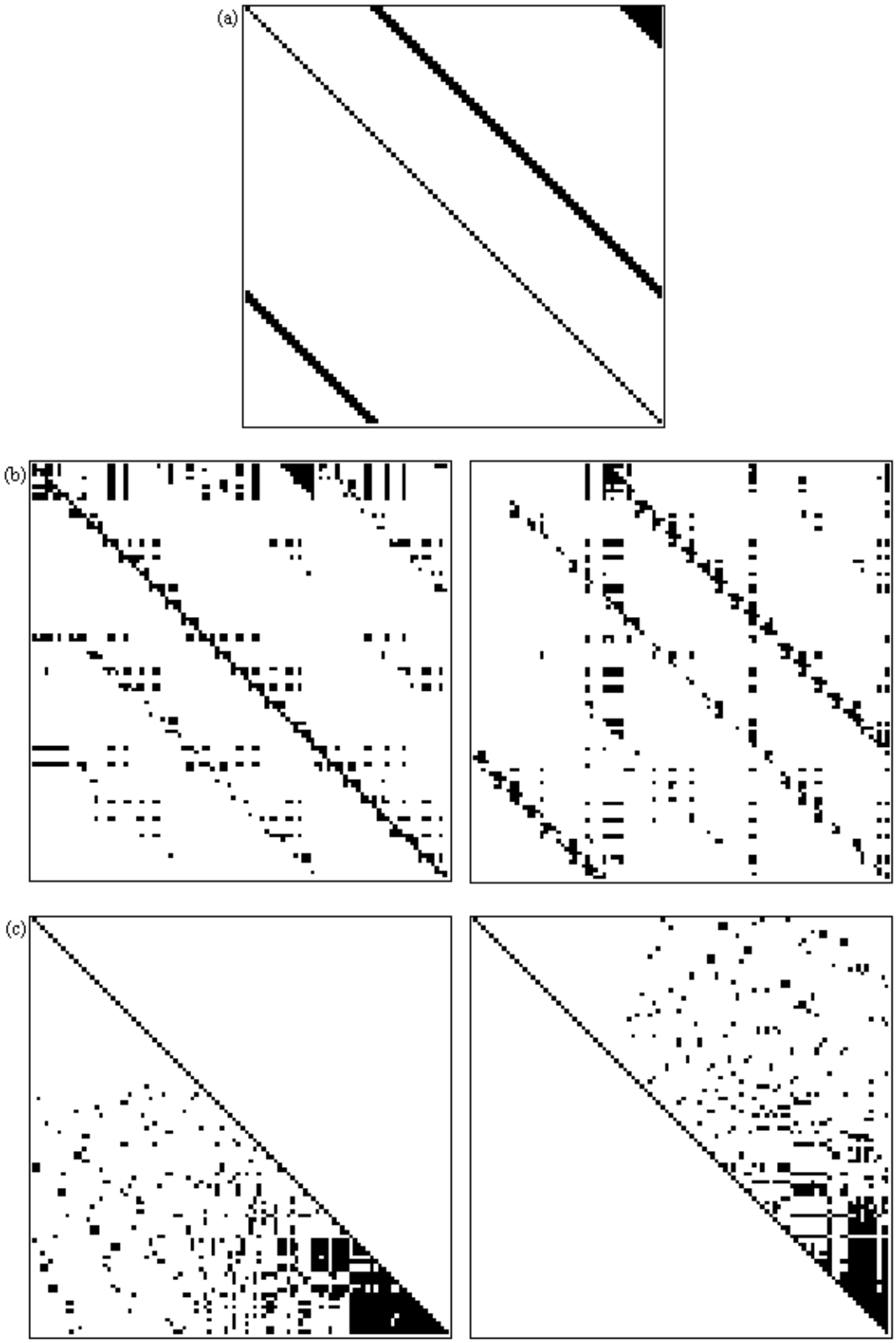

Fig 6.14. Example of LU factorization performed by the `build_lu` routine (a) input matrix A; (b) matrices M and V; (c) matrices L and U.

6.2.3 `monit_lu` — standard monitoring routine

Synopsis

```
#include "glpgel.h"
void monit_lu(LU *lu, int kase, int i, int k, double f);
```

Description

The `monit_lu` routine is the standard monitoring routine, which can be passed to the `build_lu` routine (see Subsection 6.2.2). It also can be used as an example for user-written monitoring routine.

For details see Subsection 6.2.2.

6.2.4 `solve_lu` — solve linear system using LU factorization

Synopsis

```
#include "glpgel.h"
void solve_lu(LU *lu, int flag, double x[], double work[]);
```

Description

The `solve_lu` routine solves either the system $Ax = b$ or the system $A^Tx = b$, where A is given non-singular coefficient matrix, x is dense vector of unknowns, b is dense vector of right hand sides.

The parameter `lu` should specify LU factorization of the coefficient matrix computed by the `build_lu` routine (see Subsection 6.2.2). If the `build_lu` routine reported an error, LU factorization should not be used.

The parameter `flag` specifies what system has to be solved. If `flag` $= 0$, the routine solves the system $Ax = b$. Otherwise the routine solves the transposed system $A^Tx = b$.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector of unknowns x.

The array `work` is auxiliary storage. It should have at least $1+n$ locations, where n is the order of the matrix A.

LU factorization, which `lu` points to, is not changed. Therefore it may be used several times for solving systems with the same coefficient matrix and different right hand sides.

Complexity

The `solve_lu` routine has time complexity $O[nz(L)+nz(U)]$, where $nz(L)$ is number of non-zeros in the matrix L, $nz(U)$ is number of non-zeros in the matrix U.

Method

From (6.6) it follows that

$$A^{-1} = Q^{-1}U^{-1}L^{-1}P^{-1}$$

and[1]

$$A^{-T} = PL^{-T}U^{-T}Q.$$

Therefore solving systems $Ax = b$ and $A^Tx = b$ may be performed through solving systems with triangular matrices L and U and additionally permuting elements of vectors x and b as determined by permutation matrices P and Q.

6.2.5 `delete_lu` — delete LU factorization

Synopsis

```
#include "glpgel.h"
void delete_lu(LU *lu);
```

Description

The `delete_lu` routine deletes the LU factorization `lu`, freeing all the memory allocated to this object.

References

1. *Sergio Pissanetzky.* Sparse Matrix Technology. Academic Press Inc., 1984.
2. *I. S. Duff.* MA28 — a set of FORTRAN subroutines for sparse unsymmetric linear equations. Harwell Report AERE, R. 8730, 1977.
3. *R. H. Bartels.* A Numerical Investigation of the Simplex Method. Tech. Report. Computer Science Department, Stanford University, California, 1968.
4. *R. H. Bartels, G. H. Golub.* The Simplex Method of Linear Programming Using LU-decomposition. Comm. ACM, 12, pp. 266—68, 1969.
5. *J. J. H. Forrest, J. A. Tomlin.* Updating Triangular Factors of the Basis to Maintain Sparsity in the Product Form Simplex Method. Mathematical Programming, 2, pp. 263—78, 1972.
6. *J. K. Reid.* A Sparsity Exploiting Variant of the Bartels—Golub Decomposition for Linear Programming Bases. Mathematical Programming, 24, pp. 55—69, 1982.

[1] A^{-T} means $(A^T)^{-1}$.

7 GLPMAT: Operations on Vectors and Matrices

7.1 Prelude

This chapter describes a set of routines, which operates on vectors and matrices. All routines are given in the alphabetical order.

Vectors

Vectors are stored using dense format. There is no specific data structure for representing vectors. In order to represent a vector $x = (x_1, x_2, \ldots, x_n)$ of length n the ordinary array of `double` type should be used:

```
double x[1+n];
```

Note that the array should have $n+1$ locations, since the location `x[0]` is not used. This convention provides natural correspondence: `x[1]` $= x_1$, `x[2]` $= x_2$, $\ldots$, `x[n]` $= x_n$.

Sparse matrices

Sparse matrix is a matrix, most of elements of which are equal to zero.

Sparse matrices are stored using so called *sparse rowwise/columnwise unordered* format. This means that:

only non-zero elements are explicitly stored[1] ("sparse");

for each row and each column of matrix the corresponding elements are linked into linked lists ("rowwise/columnwise");

as a rule the ordering of elements inside row and column lists is undefined ("unordered").

In order to represent sparse matrix on the whole the `MAT` structure is used:

```
typedef struct MAT MAT;
struct MAT
{     int m;
      int n;
      POOL *pool;
      ELEM row[1+m];
      ELEM col[1+n];
};
```

where:

`m` number of rows, $m > 0$;

`n` number of columns, $n > 0$;

`pool` memory pool (see Section 10.3) that holds all elements of the matrix;

`row` array of row lists. The location `row[0]` is not used. The location `row[i]` is a pointer to the first element placed in ith row ($1 \le i \le m$);

[1] It is allowed to store zero elements. Therefore "non-zero element" means "element, which is explicitly stored in the memory".

`col` array of column lists. The location `col[0]` is not used. The location `col[j]` is a pointer to the first element placed in jth column ($1 \leq j \leq n$).

The representation of a particular element of sparse matrix is defined by the following `ELEM` structure:

```
typedef struct struct ELEM;
struct ELEM
{     int i;
      int j;
      double val;
      ELEM *row;
      ELEM *col;
};
```

where:

`i` row number ($1 \leq i \leq m$);

`j` column number ($1 \leq j \leq n$);

`val` numerical value of element (usually non-zero, but zero is allowed);

`row` pointer to the next element in the same row or `NULL`, if there is no the next element (this linked list is unordered);

`col` pointer to the next element in the same column or `NULL`, if there is no the next element (this linked list is unordered).

Let's consider the following example of sparse matrix:

$$\begin{pmatrix} 0 & 0 & 2.5 & 0 & -1.7 \\ 0 & 0 & 0 & 0 & 0 \\ 3.2 & 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & -4.1 & 0 \end{pmatrix}.$$

The corresponding data structure of this matrix is shown on Fig. 7.1.

Note that the format used for representing sparse matrices allows *multiplets*, i.e. sets of elements, which have identical row and column indices. Above it was also said that this format allows explicit zeros. In some cases it is desirable to keep multiplets and/or explicit zeros (for example, when a matrix describes the structure of directed multigraph, where multiplets represent multiarcs, or when a matrix is the result of symbolic factorization, where explicit zeros determine matrix elements, which might become non-zero for other matrix with the same pattern). However, multiplets and explicit zeros should be considered as special cases. Therefore, it is assumed everywhere that *normally a sparse matrix has neither multiplets nor explicit zeros.* If some routine allows multiplets or/and explicit zeros in a sparse matrix, this will be declared explicitly in the description of the corresponding routine. It is very important, because some matrix routines are not able to work correctly, if input matrices have multiplets and/or explicit zeros. Note also that some matrix routines may produce matrices with multiplets and/or explicit zeros.

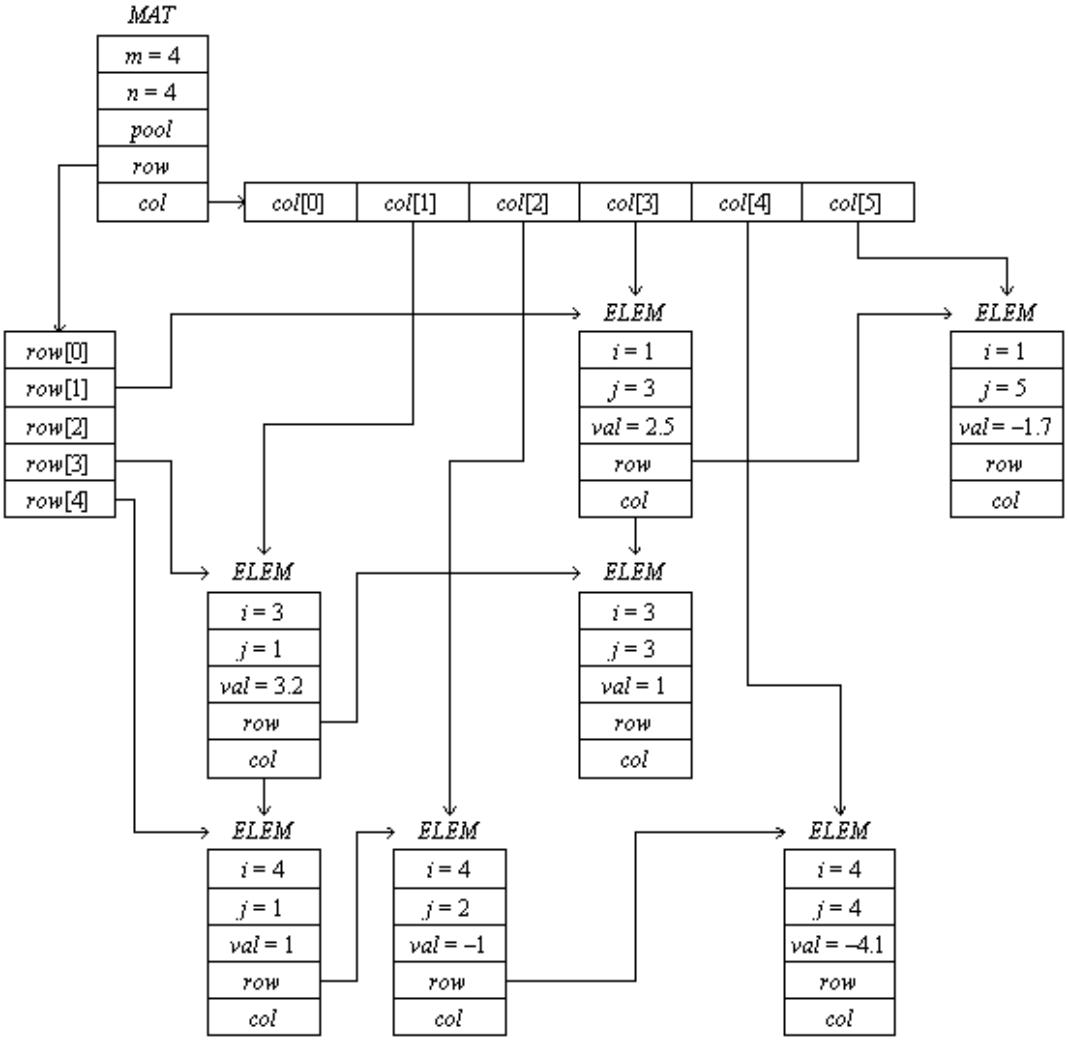

Fig 7.1. Example of sparse matrix representation

Permutation matrices

Permutation matrix is a quadratic matrix, which is a result of permuting columns (or rows) of unity matrix. (Permutation matrix can be also defined as a quadratic matrix, each row and each column of which contains only one non-zero element, which is equal to one.)

Permutation matrices are described by the PER structure:

```
typedef struct PER PER;
struct PER
{     int n;
      int row[1+n];
      int col[1+n];
};
```

where:

`n` order of permutation matrix, $n>0$;

`row` rowwise representation. The location `row[0]` is not used. The location `row[i]` contains column number of unity element, which is placed in ith row, $1 \leq i \leq n$. In other words, `row[i]` $=$ `j` if $p_{ij} = 1$;

`col` columnwise representation. The location `col[0]` is not used. The location `col[j]` contains row number of unity element, which is placed in jth column, $1 \leq j \leq n$. In other words, `col[j] = i` if $p_{ij} = 1$.

Let's consider the following example of permutation matrix:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Representation of this matrix is the following:

```
row[1] = 3, row[2] = 2, row[3] = 4, row[4] = 1,
col[1] = 4, col[2] = 2, col[3] = 1, col[4] = 3.
```

It is seen that the representation of permutation matrices is superfluous, since `row[col[j]]` $\equiv$ `j` and `col[row[i]]` $\equiv$ `i`. However, on practice it is convenient to have both arrays at the same time.

7.2 `check_mat` — check sparse matrix for correctness

Synopsis

```
#include "glpmat.h"
void check_mat(MAT *A);
```

Description

The `check_mat` routine checks representation of the sparse matrix A for correctness. In case of incorrect representation the routine reports an error and abnormally terminates the program.

Sparse matrix is correct if all its elements have correct row and column indices and there are no "lost" elements, i.e. elements which being presented in row (column) linked list are missing from the corresponding column (row) linked list.

The normal return from the `check_mat` routine guarantees that the matrix A is correct. (Note that sparse matrix is considered as correct even if it has multiplets and explicit zeros.)

Complexity

This operation is extremely inefficient and intended only for debugging purposes.

7.3 check_per — check permutation matrix for correctness

Synopsis

```
#include "glpmat.h"
void check_per(PER *P);
```

Description

The `check_per` routine checks the representation of the permutation matrix P for correctness. In case of incorrect representation the routine reports an error and abnormally terminates the program.

The normal return from the `check_per` routine guarantees that the matrix P is correct.

Complexity

This operation has time complexity $O(n)$, where n is the order of permutation matrix.

7.4 clear_line — clear row or column of sparse matrix

Synopsis

```
#include "glpmat.h"
MAT *clear_line(MAT *A, int k);
```

Description

The `clear_line` routine performs one of the following operations:

if $k > 0$, the routine clears kth row of the sparse matrix A;

if $k < 0$, the routine clears kth column of the sparse matrix A;

if $k = 0$, the routine clears the entire sparse matrix A.

To clear row, column, or entire matrix means to remove all elements from the row, column, or matrix (all removed elements are returned to the corresponding memory pool).

The `clear_line` routine works correctly if the matrix A has multiplets and/or explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is total number of non-zero elements of sparse matrix (in all three cases).

Returns

The `clear_line` routine returns a pointer to the matrix A.

7.5 clear_mat — clear sparse matrix ($A := 0$)

Synopsis

```
#include "glpmat.h"
MAT *clear_mat(MAT *A);
```

Description

The `clear_mat` routine clears the sparse matrix A, making its row and column lists be empty.

Note that the routine doesn't free the memory allocated to the matrix A. It just returns all its elements to the corresponding memory pool.

The `clear_mat` routine works correctly if the matrix A has multiplets and/or explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is total number of non-zero elements of sparse matrix.

Returns

The `clear_mat` routine returns a pointer to the matrix A.

7.6 copy_mat — copy sparse matrix ($B := A$)

Synopsis

```
#include "glpmat.h"
MAT *copy_mat(MAT *B, MAT *A);
```

Description

The `copy_mat` routine copies the sparse matrix A to the sparse matrix B. The matrices A and B should be of identical size.

The `copy_mat` performs verbatim copying. Therefore if the matrix A has multiplets and/or explicit zeros, the matrix B inherits exactly the same structure.

If A and B is the same matrix, no copying is performed.

Complexity

This operation has time complexity $O[nz(A), nz(B)]$, where $nz(A)$ and $nz(B)$ are number of non-zero elements of the matrices A and B (before copying) respectively.

Returns

The `copy_mat` routine returns a pointer to the matrix B.

7.7 copy_per — copy permutation matrix ($Q := P$)

Synopsis

```
#include "glpmat.h"
PER *copy_per(PER *Q, PER *P);
```

Description

The `copy_per` routine copies the permutation matrix P to the permutation matrix Q. The matrices P and Q should be of identical order.

If P and Q is the same matrix, no copying is performed.

Complexity

This operation has time complexity $O(n)$, where n is the order of permutation matrices.

Returns

The `copy_per` routine returns a pointer to the matrix Q.

7.8 count_nz — count non-zeros of sparse matrix

Synopsis

```
#include "glpmat.h"
int count_nz(MAT *A, int k);
```

Description

If $k = 0$, the `count_nz` routine returns total number of non-zeros in the entire sparse matrix A.

If $k > 0$, the routine returns number of non-zeros in kth row of the matrix A.

If $k < 0$, the routine returns number of non-zeros in kth column of the matrix A.

Each element is counted once even if it belongs to multiplet (i.e. if there are other elements with the same row and column indices) or if it is explicit zero.

Complexity

This operation has time complexity $O(nz)$, where nz is the returned value.

Returns

The `count_nz` routine returns the number of non-zero elements.

7.9 `create_mat` — create sparse matrix

Synopsis

```
#include "glpmat.h"
MAT *create_mat(int m, int n);
```

Description

The `create_mat` routine creates a sparse matrix with m rows and n columns (m, $n > 0$). Initially the created matrix is null matrix, i.e. its row and column lists are empty.

Returns

The `create_mat` routine returns a pointer to the created matrix.

7.10 `create_per` — create permutation matrix

Synopsis

```
#include "glpmat.h"
PER *create_per(int n);
```

Description

The `create_per` routine creates a permutation matrix of order n ($n > 0$). Initially the created matrix is identity matrix.

Returns

The `create_per` routine returns a pointer to the created matrix.

7.11 `delete_mat` — delete sparse matrix

Synopsis

```
#include "glpmat.h"
void delete_mat(MAT *A);
```

Description

The `delete_mat` routine deletes the sparse matrix A, freeing the memory allocated to this object.

7.12 delete_per — delete permutation matrix

Synopsis

```
#include "glpmat.h"
void delete_per(PER *P);
```

Description

The `delete_per` routine deletes the permutation matrix P, freeing the memory allocated to this object.

7.13 eq_scaling — implicit equilibration scaling

Synopsis

```
#include "glpmat.h"
void eq_scaling(MAT *A, double R[], double S[], int ord);
```

Description

The `eq_scaling` routine performs implicit equilibration scaling of the matrix RAS, where A is the given sparse matrix (remains unchanged on exit), R and S are given diagonal scaling matrices.

The result of scaling is the matrix $R'AS'$, where R' and S' are new scaling matrices computed by the routine.

Diagonal elements of the matrix R should be placed in locations `R[1]`, `R[2]`, ..., `R[m]`, where m is number of rows of the matrix A. Analogously, diagonal elements of the matrix S should be placed in locations `S[1]`, `S[2]`, ..., `S[n]`, where n is number of columns of the matrix A. Diagonal elements of new scaling matrices R' and S' are placed in the same manner on exit.

If the matrix A is not scaled on entry, R and S should be unity matrices, i.e. all their diagonal elements should be set to 1.

The parameter `ord` specifies the order of scaling:

if `ord` = 0, at first the routine scales rows, then columns;

if `ord` = 1, at first the routine scales columns, then rows.

Method

In order to perform equilibration scaling the routine divide all elements of each row (column) of the matrix RAS by the largest absolute value of elements in the corresponding row (column).

7.14 gm_scaling — implicit geometric mean scaling

Synopsis

```
#include "glpmat.h"
void gm_scaling(MAT *A, double R[], double S[], int ord,
   double eps, int itmax);
```

Description

The `gm_scaling` routine performs implicit geometric mean scaling of the matrix RAS, where A is the given sparse matrix (remains unchanged on exit), R and S are given diagonal scaling matrices.

The result of scaling is the matrix $R'AS'$, where R' and S' are new scaling matrices computed by the routine.

Diagonal elements of the matrix R should be placed in locations `R[1]`, `R[2]`, ..., `R[m]`, where m is number of rows of the matrix A. Analogously, diagonal elements of the matrix S should be placed in locations `S[1]`, `S[2]`, ..., `S[n]`, where n is number of columns of the matrix A. Diagonal elements of new scaling matrices R' and S' are placed in the same manner on exit.

If the matrix A is not scaled on entry, R and S should be unity matrices, i.e. all their diagonal elements should be set to 1.

The parameter `ord` specifies the order of scaling:

if `ord` = 0, at first the routine scales rows, then columns;

if `ord` = 1, at first the routine scales columns, then rows.

Description of parameters `eps` and `itmax` are given below (see method description).

Method

In order to perform geometric mean scaling the routine divides all elements of each row (column) by $(\beta/\alpha)^{1/2}$, where α and β are, respectively, smallest and largest absolute values of non-zero elements of the corresponding row (column). For improving the scaling "quality" the routine performs scaling rows and columns several times.

The parameter *eps* specifies a termination criterion. The iterative scaling process is terminated, if $t^{(k-1)} - t^{(k)} < eps \cdot t^{(k-1)}$, where $t^{(k)} = \beta^{(k)}/\alpha^{(k)}$ is the scaling "quality", $\alpha^{(k)}$ and $\beta^{(k)}$ are, respectively, smallest and largest absolute elements of the matrix $R'AS'$ on kth iteration. In most cases the value $eps = 0.01$—0.1 may be recommended.

The parameter *itmax* specifies maximal number of scaling iterations. In most cases the value $itmax = 10$—50 may be recommended.

On each iteration the `gm_scaling` routine prints the scaling "quality" (see above).

7.15 inv_per — invert permutation matrix (*P* := *P*')

Synopsis

```
#include "glpmat.h"
PER *inv_per(PER *P);
```

Description

The `inv_per` routine inverts (transposes) the permutation matrix P.

Complexity

This operation has time complexity $O(1)$.

Returns

The `inv_per` routine returns a pointer to the matrix P.

7.16 iper_vec — permute vector elements in reverse order (*y* := *P*'*x*)

Synopsis

```
#include "glpmat.h"
double *iper_vec(double y[], PER *P, double x[]);
```

Description

The `iper_vec` routine computes the product $y = P^{-1}x$, where P is permutation matrix, x and y are dense vectors.

The arrays `x` and `y` should not overlap. The matrix P and the vector x remain unchanged on exit.

Complexity

This operation has time complexity $O(n)$, where n is order of the matrix P.

Returns

The `iper_vec` routine returns a pointer to the array y.

7.17 `l_solve` — solve lower triangular system $Lx = b$

Synopsis

```
#include "glpmat.h"
double *l_solve(MAT *L, double x[]);
```

Description

The `l_solve` routine solves the system $Lx = b$, where L is non-singular lower triangular matrix, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector of unknowns x.

The matrix L may contain explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of L.

Returns

The `l_solve` routine returns a pointer to the array `x`.

Method

Since ith equation of the system $Lx = b$ has the form:

$$\sum_{j=1}^{i} l_{ij} x_j = l_{i1} x_1 + l_{i2} x_2 + \ldots + l_{i,i-1} x_{i-1} + l_{ii} x_i = b_i,$$

the unknown x_i can be computed using the formula:

$$x_i = (b_i - l_{i1} x_1 - l_{i2} x_2 - \ldots - l_{i,i-1} x_{i-1}) / l_{ii} = \frac{1}{l_{ii}} (b_i - \sum_{j=1}^{i-1} l_{ij} x_j),$$

assuming that $x_1, x_2, \ldots, x_{i-1}$ have been computed yet.

7.18 `lt_solve` — solve transposed lower triangular system $L'x = b$

Synopsis

```
#include "glpmat.h"
double lt_solve(MAT *L, double x[]);
```

Description

The `lt_solve` routine solves the system $L^T x = b$, where L is non-singular lower triangular matrix, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector of unknowns x.

The matrix L may contain explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of L.

Returns

The `lt_solve` routine returns a pointer to the array x.

Method

Since ith equation of the system $L^T x = b$ has the form:

$$\sum_{j=i}^{n} l_{ji} x_j = l_{ii} x_i + l_{i+1,i} x_{i+1} + \ldots + l_{n-1,i} x_{n-1} + l_{ni} x_n = b_i,$$

the unknown x_i can be computed using the formula:

$$x_i = (b_i - l_{i+1,i} x_{i+1} - \ldots - l_{n-1,i} x_{n-1} - l_{ni} x_n)/l_{ii} = \frac{1}{l_{ii}}(b_i - \sum_{j=i+1}^{n} l_{ji} x_j),$$

assuming that $x_{i+1}, \ldots, x_{n-1}, x_n$ have been computed yet.

7.19 `mat_per` — permute matrix columns ($A := AP$)

Synopsis

```
#include "glpmat.h"
MAT *mat_per(MAT *A, PER *P, void *work[]);
```

Description

The `mat_per` routine computes the product $A' = AP$, where A is sparse matrix, P is permutation matrix, and stores the resultant matrix A' instead the input matrix A.

The auxiliary array `work` should have at least $1+n$ locations, where n is number of columns of the matrix A.

Since the `mat_per` routine just permutes columns of the matrix A, the latter may have multiplets and explicit zeros, which are retained.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `mat_per` routine returns a pointer to the matrix A.

7.20 `mat_vec` — multiply sparse matrix on vector ($y := Ax$)

Synopsis

```
#include "glpmat.h"
double *mat_vec(double y[], MAT *A, double x[]);
```

Description

The `mat_vec` routine computes the product $y = Ax$, where A is sparse matrix, x and y are dense vectors.

The arrays `x` and `y` should not overlap. The matrix A and the vector x remain unchanged on exit.

The matrix A may have explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros A.

Returns

The `mat_vec` routine returns a pointer to the array `y`.

7.21 `new_elem` — create new element of sparse matrix

Synopsis

```
#include "glpmat.h"
ELEM *new_elem(MAT *A, int i, int j, double val);
```

Description

The `new_elem` routine creates new element of the sparse matrix A, which is placed in ith row and jth column and has the value `val`.

Note that the `new_elem` routine doesn't check whether such element already exists. It just appends new element to the corresponding row and column lists. Therefore multiple calls to the `new_elem` routine with the same i and j indices involve appearance of multiplets. The routine also allows creating explicit zeros, if `val` is set to 0.

Complexity

This operation has time complexity $O(1)$.

Returns

The `new_elem` routine returns a pointer to the created element.

7.22 per_mat — permute matrix rows ($A := PA$)

Synopsis

```
#include "glpmat.h"
MAT *per_mat(PER *P, MAT *A, void *work[]);
```

Description

The `per_mat` routine computes the product $A' = PA$, where A is sparse matrix, P is permutation matrix, and stores the resultant matrix A' instead the input matrix A.

The auxiliary array work should have at least $1+m$ locations, where m is number of rows of the matrix A.

Since the `per_mat` routine just permutes rows of the matrix A, the latter may have multiplets and explicit zeros, which are retained.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `per_mat` routine returns a pointer to the matrix A.

7.23 per_vec — permute vector elements ($y := Px$)

Synopsis

```
#include "glpmat.h"
double *per_vec(double y[], PER *P, double x[]);
```

Description

The `per_vec` routine computes the product $y = Px$, where P is permutation matrix, x and y are dense vectors.

The arrays `x` and `y` should not overlap. The matrix P and the vector x remain unchanged on exit.

Complexity

This operation has time complexity $O(n)$, where n is order of the matrix P.

Returns

The `per_vec` routine returns a pointer to the array `y`.

7.24 `reset_per` — reset permutation matrix ($P := I$)

Synopsis

```
#include "glpmat.h"
PER *reset_per(PER *P);
```

Description

The `reset_per` routine resets the permutation matrix P to unity matrix.

Complexity

This operation has time complexity $O(n)$, where n is order of the matrix P.

Returns

The `reset_per` routine returns a pointer to the matrix P.

7.25 `scrape_mat` — remove tiny elements from sparse matrix

Synopsis

```
#include "glpmat.h"
int scrape_mat(MAT *A, double eps);
```

Description

The `scrape_mat` routine removes from the sparse matrix A all elements, whose absolute value is not greater than *eps*. If $eps = 0$, the routine removes only explicit zeros.

The matrix A may contain multiplets and explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `scrape_mat` routine returns number of elements, which have been removed.

7.26 sort_mat — sort row and column lists of sparse matrix

Synopsis

```
#include "glpmat.h"
MAT *sort_mat(MAT *A);
```

Description

The `sort_mat` routine sorts row and column lists of the sparse matrix A in order that in row lists all its elements follow in the order of increasing column indices and in column lists all its elements follow in the order of increasing row indices.

The matrix may contain multiplets and explicit zeros, which are retained.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `sort_mat` routine returns a pointer to the matrix A.

7.27 submatrix — copy submatrix of sparse matrix

Synopsis

```
#include "glpmat.h"
MAT *submatrix(MAT *B, MAT *A, int i1, int i2, int j1, int j2);
```

Description

The `submatrix` routine copies a submatrix of the sparse matrix A to the sparse matrix B. The submatrix consists of rows with numbers `i1` to `i2` and columns with numbers `j1` to `j2`. The matrix B should have `i2-i1+1` rows and `j2-j1+1` columns. The matrix A remains unchanged.

If the submatrix has explicit zeros, they are retained in the matrix B.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `submatrix` routine returns a pointer to the matrix B.

7.28 `test_mat_d` — create test sparse matrix of $D(n,c)$ class

Synopsis

```
#include "glpmat.h"
MAT *test_mat_d(int n, int c);
```

Description

The `test_mat_d` routine creates a test sparse matrix of $D(n, c)$ class (for details see [1]).

Matrix of $D(n, c)$ class is a quadratic matrix of order n. It has unit main diagonal, three codiagonals above main diagonal in the distance of c, which are cyclically continued below main diagonal, and a triangle of 10×10 elements in the upper right corner. An example of $D(100, 30)$ test matrix is shown on Fig. 7.2.

It is necessary that $n \geq 14$ and $1 \leq c \leq n - 13$.

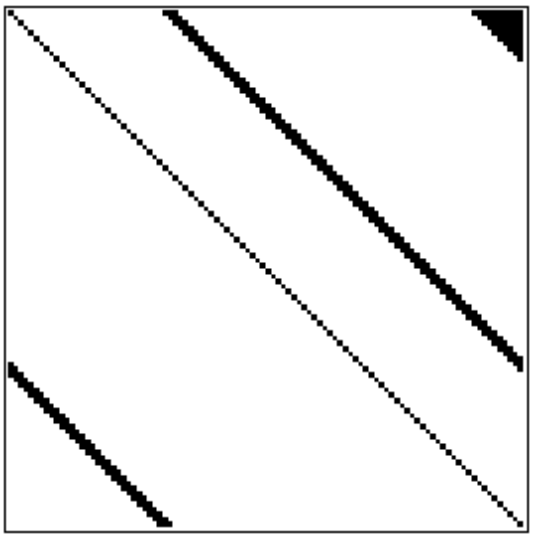

Fig 7.2. The pattern of $D(100, 30)$ test matrix

Returns

The `test_mat_d` routine returns a pointer to the created test matrix.

7.29 `test_mat_e` — create test sparse matrix of $E(n,c)$ class

Synopsis

```
#include "glpmat.h"
MAT *test_mat_e(int n, int c);
```

Description

The `test_mat_e` routine creates a test sparse matrix of $E(n, c)$ class (for details see [1]).

Matrix of $E(n, c)$ class is a symmetric positive definite matrix of order n. It has the number 4 on its main diagonal and the number -1 on its four codiagonals, two of which neighbor with main diagonal and two others are away from main diagonal in the distance c. An example of $E(100, 30)$ test matrix is shown on Fig. 7.3.

It is necessary that $n \geq 3$ and $2 \leq c \leq n - 1$.

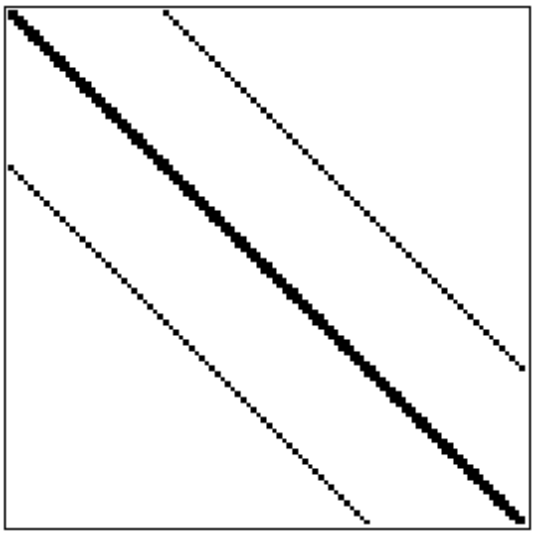

Fig 7.3. The pattern of $E(100, 30)$ test matrix

Returns

The `test_mat_e` routine returns a pointer to the created test matrix.

7.30 `tmat_vec` — multiply transposed sparse matrix on vector ($y := A'x$)

Synopsis

```
#include "glpmat.h"
double *tmat_vec(double y[], MAT *A, double x[]);
```

Description

The `tmat_vec` routine computes the product $y = A^T x$, where A is sparse matrix, x and y are dense vectors.

The arrays `x` and `y` should not overlap. The matrix A and the vector x remain unchanged on exit.

The matrix A may have explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `tmat_vec` routine returns a pointer to the array `y`.

7.31 `trn_mat` — transpose sparse matrix ($A := A'$)

Synopsis

```
#include "glpmat.h"
MAT *trn_mat(MAT *A);
```

Description

The `trn_mat` routine transposes the sparse matrix A.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of A.

Returns

The `trn_mat` routine returns a pointer to the matrix A.

7.32 `u_solve` — solve upper triangular system $Ux = b$

Synopsis

```
#include "glpmat.h"
double *u_solve(MAT *U, double x[]);
```

Description

The `u_solve` routine solves the system $Ux = b$, where U is non-singular upper triangular matrix, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector of unknowns x.

The matrix U may contain explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of U.

Returns

The `u_solve` routine returns a pointer to the array `x`.

Method

Since ith equation of the system $Ux = b$ has the form:

$$\sum_{j=i}^{n} u_{ij}x_j = u_{ii}x_i + u_{i,i+1}x_{i+1} + \ldots + u_{i,n-1}x_{n-1} + u_{in}x_n = b_i,$$

the unknown x_i can be computed using the formula:

$$x_i = (b_i - u_{i,i+1}x_{i+1} - \ldots - u_{i,n-1}x_{n-1} - u_{i,n}x_n)/u_{ii} = \frac{1}{u_{ii}}(b_i - \sum_{j=i+1}^{n} u_{ij}x_j),$$

assuming that $x_{i+1}, \ldots, x_{n-1}, x_n$ have been computed yet.

7.33 `ut_solve` — solve transposed upper triangular system *U'x = b*

Synopsis

```
#include "glpmat.h"
double *u_solve(MAT *U, double x[]);
```

Description

The `ut_solve` routine solves the system $U^T x = b$, where U is non-singular upper triangular matrix, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector of unknowns x.

The matrix U may contain explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of U.

Returns

The `ut_solve` routine returns a pointer to the array `x`.

Method

Since ith equation of the system $U^T x = b$ has the form:

$$\sum_{j=1}^{i} u_{ji} x_j = u_{1i} x_1 + u_{2i} x_2 + \ldots + u_{i-1,i} x_{i-1} + u_{ii} x_i = b_i,$$

the unknown x_i can be computed using the formula:

$$x_i = (b_i - u_{1i} x_1 - u_{2i} x_2 - \ldots - u_{i-1,i} x_{i-1}) / u_{ii} = \frac{1}{u_{ii}} (b_i - \sum_{j=1}^{i-1} u_{ji} x_j),$$

assuming that $x_1, x_2, \ldots, x_{i-1}$ have been computed yet.

7.34 `v_solve` — solve implicit upper triangular system $Vx = b$

Synopsis

```
#include "glpmat.h"
double *v_solve(PER *P, MAT *V, PER *Q, double x[], double work[]);
```

Description

The `v_solve` routine solves the system $Vx = b$, where $U = PVQ$ is implicit non-singular upper triangular matrix, V is the given sparse matrix, P and Q are permutation matrices, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector of unknowns x.

The matrix V may contain explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of V.

Returns

The `v_solve` routine returns a pointer to the array `x`.

Method

The `v_solve` routine uses the same formulae as the `u_solve` routine (see Section 7.32), determining elements of the implicit upper triangular matrix $U = PVQ$ each time, when it is necessary.

7.35 `vt_solve` — solve implicit transposed upper triangular system *V'x = b*

Synopsis

```
#include "glpmat.h"
double *vt_solve(PER *P, MAT *V, PER *Q, double x[], double work[]);
```

Description

The `vt_solve` routine solves the system $V^T x = b$, where $U = PVQ$ is implicit non-singular upper triangular matrix, V is the given sparse matrix, P and Q are permutation matrices, x is dense vector of unknowns, b is dense vector of right hand sides.

On entry the array `x` should contain the vector of right hand sides b. On exit this array contains the vector x.

The matrix V may contain explicit zeros.

Complexity

This operation has time complexity $O(nz)$, where nz is number of non-zeros of V.

Returns

The `vt_solve` routine returns a pointer to the array x.

Method

The `vt_solve` routine uses the same formulae as the `ut_solve` routine (see Section 7.33), determining elements of the implicit upper triangular matrix $U = PVQ$ each time, when it is necessary.

References

1. Ole Østerby, Zahari Zlatev. Direct Methods for Sparse Matrices. Springer-Verlag, 1983.

8 GLPMPS: Processing LP Problems Using MPS Format

8.1 Linear programming (LP) problem

The MPS format[1] is intended for coding LP problem data.

This format assumes the following linear programming (LP) problem formulation:

minimize (or maximize)

$$Z = x_{obj} \tag{8.1}$$

subject to linear constraints

$$\begin{aligned} x_1 &= a_{11}x_{m+1} + a_{12}x_{m+2} + \ldots + a_{1n}x_{m+n} \\ x_2 &= a_{21}x_{m+1} + a_{22}x_{m+2} + \ldots + a_{2n}x_{m+n} \\ &\ldots\ldots\ldots \\ x_m &= a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} \end{aligned} \tag{8.2}$$

and bounds of variables

$$\begin{aligned} l_1 &\le x_1 \le u_1 \\ l_2 &\le x_2 \le u_2 \\ &\ldots\ldots\ldots \\ l_{m+n} &\le x_{m+n} \le u_{m+n} \end{aligned} \tag{8.3}$$

where $x_1, x_2, \ldots, x_m$ are rows or auxiliary variables; $x_{m+1}, x_{m+2}, \ldots, x_{m+n}$ are columns or structural variables; Z is the objective function; obj is number of row, which defines the objective function; $a_{11}, a_{12}, \ldots, a_{mn}$ are constraint coefficients; $l_1, l_2, \ldots, l_{m+n}$ are lower bounds of variables; $u_1, u_2, \ldots, u_{m+n}$ are upper bounds of variables.

Auxiliary variables are called *rows*, because they correspond to rows of the constraint matrix (i.e. matrix formed by constraint coefficients). Note that the value of an auxiliary variable is the value of the corresponding linear form. Analogously, structural variables are called *columns*, because they correspond to columns of the constraint matrix.

Bounds of variables can be finite as well as infinite. Besides, a lower bound can be equal to an upper bound. Thus, the following types of variables are possible:

Bounds of variable	Type of variable
$-\infty < x_k < +\infty$	Free (unbounded) variable
$l_k \le x_k < +\infty$	Variable with lower bound
$-\infty < x_k \le u_k$	Variable with upper bound
$l_k \le x_k \le u_k$	Double-bounded variable
$l_k = x_k = u_k$	Fixed variable

Note that types of variables shown above are applicable to structural as well as to auxiliary variables.

[1] The MPS format was proposed in 1960's by IBM as input format for their mathematical programming system MPS/360. To-day the MPS format became most widely used format, which is understood by most packages that are intended for solving mathematical programming problems. This chapter describes only those features of the MPS format, which are supported by GLPK routines.

8.2 MPS Format Specification

A file in the MPS format is a text file, which contains two types of cards[2]: indicator cards and data cards.

Indicator cards determine a kind of following data. Each indicator card has one word in uppercase letters beginning at the column 1.

Data cards contain LP problem data. Each data card is divided into six fixed fields:

	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
Columns	2—3	5—12	15—22	25—36	40—47	50—61
Contents	Code	Name	Name	Number	Name	Number

On a particular data card some fields may be optional.

Names are used for identifying LP rows, LP columns, and some vectors (see below).

Aligning the indicator code to the left margin of the field 1 is optional.

All names specified in fields 2, 3, and 5 should contain from 1 to 8 arbitrary characters (except control characters). If a name is placed in the field 3 or 5, its first character should not be the dollar sign '`$`'. If a name contains spaces, these spaces are ignored.

All numerical values placed in fields 4 and 6 should be coded in the form *sxx*`E`*syy*, where *s* is the plus '`+`' or the minus '`-`' sign, *xx* is a real number with optional decimal point, *yy* is an integer decimal exponent. Any number should contain up to 12 characters. If the sign *s* is omitted, the plus sign is assumed. The exponent part is optional. If a number contains spaces, these spaces are ignored.

If a card has the asterisk '`*`' in the column 1, this card is considered as a comment and ignored. Besides, if the first character in the field 3 or 5 is the dollar sign '`$`', all characters from the dollar sign to the end of the card are considered as comment and ignored.

MPS file should contain cards in the following order:

- the NAME indicator card;
- the ROWS indicator card;
- data cards specifying LP rows (constraints or auxiliary variables);
- the COLUMNS indicator card;
- data cards specifying LP columns (structural variables) and constraint coefficients;
- the RHS indicator card;
- data cards specifying right-hand sides of constraints;
- the RANGES indicator card;
- data cards specifying double-sided constraints;
- the BOUNDS indicator card;
- data cards specifying types and bounds of LP columns;
- the ENDATA indicator card.

[2] In 1960's MPS file was a deck of 80-column punching cards. The author decided to keep the word "card", which is to be understood as "line of text file".

Section is a group of cards consisting of an indicator card and data cards following that indicator card. For example, the ROWS section consists of the ROWS indicator card and data cards specifying LP rows.

Sections RHS, RANGES, and BOUNDS are optional and may be omitted.

NAME indicator card

The NAME indicator card should be the first card of MPS file (except optional comment cards). This card should contain the word '`NAME`' in columns 1—4 and the problem name in the field 3. The problem name is optional and may be omitted.

ROWS section

The ROWS section should start with the indicator card, which contains the word '`ROWS`' in columns 1—3.

Each data card of the ROWS section specifies one row (constraint) of the LP problem. These data cards have the following format.

The filed 1 specifies the row type (see (8.2), Section 8.1):

'`N`' in the filed 1 means that the row is free (unbounded), i.e. the corresponding constraint has the form:

$$-\infty < x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} < +\infty;$$

'`L`' in the field 1 means that the constraint is of "less than or equal to" type:

$$-\infty < x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} \leq b_i;$$

'`G`' in the field 1 means that the constraint is of "greater than or equal to" type:

$$b_i \leq x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} < +\infty;$$

'`E`' in the field 1 means that the row is the constraint of "equal to" type:

$$x_i = a_{m1}x_{m+1} + a_{m2}x_{m+2} + \ldots + a_{mn}x_{m+n} = b_i.$$

where b_i is the right hand side. Note that each constraint is connected with the corresponding auxiliary variable (x_i above), therefore bounds of the linear form can be understood as bounds of such auxiliary variable.

The field 2 specifies the row name (i.e. the name of the corresponding auxiliary variable).

Fields 3, 4, 5, and 6 are not used and should be empty.

Numerical values of all non-zero right-hand sides b_i should be specified in the RHS section. All double-sided constraints should be specified in the RANGES section.

COLUMNS section

The COLUMNS section should start with the indicator card that contains the word '`COLUMNS`' in columns 1—7.

Each data card of the COLUMNS section specifies one or two constraint coefficients and introduces names of LP columns, i.e. names of structural variables. These data cards have the following format.

The field 1 is not used and should be empty.

The field 2 specifies the column name. If this field is empty, the name from the previous data card is used.

The field 3 specifies the row name defined in the ROWS section.

The field 4 specifies the numerical value of the constraint coefficient a_{ij}, which is placed in the corresponding row and column.

Fields 5 and 6 are optional. If they are used, they should contain the second pair "row name—constraint coefficient" for the same column.

Elements of constraint matrix (i.e. constraint coefficients) should follow in the column wise manner: all elements for the current column should be specified before specifying elements for the next column. However, the order of rows in the COLUMNS section may differ from the order of rows in the ROWS section.

Constraint coefficients not specified in the COLUMNS section, are considered as zeros. Therefore zero coefficients may be omitted, although it's allowed to specify them.

Bounds of columns should be specified in the BOUNDS section.

RHS section

The RHS section should start with the indicator card, which contains the word '`RHS`' in columns 1—3.

Each data card of the RHS section specifies one or two right-hand sides b_i (see the description of the ROWS section).

The field 1 is not used and should be empty.

The field 2 specifies the name of the right-hand side (RHS) vector[3]. If this field is empty, the name from the previous data card is used.

The field 3 specifies the row name defined in the ROWS section.

The field 4 specifies the right-hand side b_i for the row, whose name is specified in the field 3. Depending on the row type b_i defines the lower bound (in case of G type), or the upper bound (in case of L type), or the fixed value (in case of E type).

Fields 5 and 6 are optional. If they are used, they should contain the second pair "row name—right hand side value" for the same RHS vector.

All right hand sides for the current RHS vector should be specified before right hand sides for the next RHS vector. However, the order of rows in the COLUMNS section may

[3] This allows the user to specify several RHS vectors in the same MPS file. However, for solving the problem the user should choose the particular RHS vector.

differ from the order of rows in the ROWS section. Each RHS vector should have unique name.

Right-hand sides not specified in the RHS section are considered as zeros. Therefore zero right hand sides may be omitted, although it's allowed to specify them.

RANGES section

The RANGES section should start with the indicator card, which contains the word '`RANGES`' in columns 1—6.

Each data card of the RANGES section specifies one or two ranges for double-sided constraints, i.e. for constraints that have types L and G at the same time:

$$l_i \le x_i = a_{i1}x_{m+1} + a_{i2}x_{m+2} + \ldots + a_{in}x_{m+n} \le u_i,$$

where x_i is an auxiliary variable, l_i is a lower bound, u_i is an upper bound.

The field 1 is not used and should be empty.

The field 2 specifies the name of the range vector[4]. If this field is empty, the name from the previous data card is used.

The field 3 specifies the row name defined in the ROWS section.

The field 4 specifies the range value ri for the row, whose name is specified in the field 3.

Fields 5 and 6 are optional. If they are used, they should contain the second pair "row name—range value" for the same range vector.

All range values for the current range vector should be specified before range values for the next range vector. However, the order of rows in the RANGES section may differ from the order of rows in the ROWS section. Each range vector should have unique name.

Lower and upper bounds are determined by the following table:

Row type	Sign of r_i	Lower Bound	Upper Bound
G	+ or –	b_i	$b_i + \|r_i\|$
L	+ or –	$b_i - \|r_i\|$	b_i
E	+	b_i	$b_i + \|r_i\|$
E	–	$b_i - \|r_i\|$	b_i

where b_i is the right-hand side specified in the RHS section (if b_i is not specified, it is considered to be equal to zero), r_i is the range value specified in the RANGES section).

BOUNDS section

The BOUNDS section should start with the indicator card, which contains the word '`BOUNDS`' in columns 1—6.

Each data card of the BOUNDS section specifies one bound of one column (structural variable).

The field 1 specifies the bound type:

[4] This allows the user to specify several range vectors in the same MPS file. However, for solving the problem the user should choose the particular range vector.

'LO' lower bound;

'UP' upper bound;

'FX' fixed variable (lower bound equal to upper bound);

'FR' free variable (no bounds);

'MI' no lower bound (lower bound is "minus infinity");

'PL' no upper bound (upper bound is "plus infinity").

The field 2 contains the name of the bound vector[5]. If this field is empty, the name from the previous data card is used.

The field 3 contains the column name defined in the COLUMNS section.

The field 4 specifies the bound value. If the bound type in the field 1 differs from LO, UP, or FX, this value is ignored and may be omitted.

Fields 5 and 6 are not used and should be empty.

All bound values for the current bound vector should be specified before bound values for the next bound vector. However, the order of columns in the BOUNDS section may differ from the order of columns in the COLUMNS section. Each bound vector should have unique name. The specification of the lower bound should precede the specification of the upper bound for the same column (if both they are explictly specified).

By default all columns (structural variables) are considered as non-negative, i.e. with lower bound equal to zero and upper bound equal to $+\infty$. Lower (l_j) or/and upper (u_j) bounds of each column (i.e. structural variable x_j) specified in the BOUNDS section are set in the following way (s_j denotes the corresponding bound value):

'LO' sets l_j to s_j;

'UP' sets u_j to s_j;

'FX' sets both l_j and u_j to s_j;

'FR' sets l_j to $-\infty$ and u_j to $+\infty$;

'MI' sets l_j to $-\infty$;

'PL' sets u_j to $+\infty$.

If only zero upper bound is specified, it is the same as if FX type would be specified.

ENDATA indicator card

The ENDATA indicator card should be the last card of MPS file (except optional comment cards). This card should contain the word 'ENDATA' in columns 1—6.

It is impossible to specify the objective function and the optimization direction explicitly in the MPS file. However, the following rule is used by default in all cases: the first row of N type is considered as the row of the objective function (i.e. the objective function is the same as the corresponding auxiliary variable), which should be minimized.

[5] This allows the user to specify several bound vectors in the same MPS file. However, for solving the problem the user should choose the particular bound vector.

Example of MPS file

In order to illustrate what the MPS format is, let us consider the following example LP problem:

```
   minimize

      Z = COST

   subject to linear constraints

COST = 0.03 MIX1 + 0.08 MIX2 + 0.17 MIX3 + 0.12 MIX4 + 0.15 MIX5 + 0.21 ALUM
     + 0.38 SILC
SUM  =      MIX1 +      MIX2 +      MIX3 +      MIX4 +      MIX5 +      ALUM
     + SILC
FE   = 0.15 MIX1 + 0.04 MIX2 + 0.02 MIX3 + 0.04 MIX4 + 0.02 MIX5 + 0.01 ALUM
     + 0.03 SILC
CU   = 0.03 MIX1 + 0.05 MIX2 + 0.08 MIX3 + 0.02 MIX4 + 0.06 MIX5 + 0.01 ALUM
MN   = 0.02 MIX1 + 0.04 MIX2 + 0.01 MIX3 + 0.02 MIX4 + 0.02 MIX5
MG   = 0.02 MIX1 + 0.03 MIX2 + 0.01 MIX5
AL   = 0.70 MIX1 + 0.75 MIX2 + 0.80 MIX3 + 0.75 MIX4 + 0.80 MIX5 + 0.97 ALUM
SI   = 0.02 MIX1 + 0.06 MIX2 + 0.08 MIX3 + 0.12 MIX4 + 0.02 MIX5 + 0.01 ALUM
     + 0.97 SILC

   and bounds of (auxiliary and structural) variables

      -inf <  COST < +inf        0 <= MIX1 <=  200
              SUM  =  200        0 <= MIX2 <= 2500
      -inf <  FE   <=  60      400 <= MIX3 <=  800
      -inf <  CU   <= 100      100 <= MIX4 <=  700
      -inf <  MN   <=  40        0 <= MIX3 <= 1500
      1500 <= AL   < +inf        0 <= ALUM <  +inf
       250 <= SI   <= 300        0 <= SILC <  +inf
```

The MPS file, which specifies data for this example LP problem, is shown below (two top comment lines show card positions).

```
*000000001111111111222222222233333333334444444445555555566
*234567890123456789012345678901234567890123456789012345678901
NAME          PLAN
ROWS
 N  COST
 E  SUM
 L  FE
 L  CU
 L  MN
 L  MG
 G  AL
 L  SI
```

```
COLUMNS
    MIX1      COST            .03000   SUM            1.00000
              FE              .15000   CU              .03000
              MN              .02000   MG              .02000
              AL              .70000   SI              .02000
    MIX2      COST            .08000   SUM            1.00000
              FE              .04000   CU              .05000
              MN              .04000   MG              .03000
              AL              .75000   SI              .06000
    MIX3      COST            .17000   SUM            1.00000
              FE              .02000   CU              .08000
              MN              .01000   AL              .80000
              SI              .08000
    MIX4      COST            .12000   SUM            1.00000
              FE              .04000   CU              .02000
              MN              .02000   AL              .75000
              SI              .12000
    MIX5      COST            .15000   SUM            1.00000
              FE              .02000   CU              .06000
              MN              .02000   MG              .01000
              AL              .80000   SI              .02000
    ALUM      COST            .21000   SUM            1.00000
              FE              .01000   CU              .01000
              AL              .97000   SI              .01000
    SILC      COST            .38000   SUM            1.00000
              FE              .03000   SI              .97000
RHS
    RHS1      SUM         2000.00000   FE            60.00000
              CU           100.00000   MN            40.00000
              SI           300.00000
              MG            30.00000   AL          1500.00000
RANGES
    RNG1      SI            50.00000
BOUNDS
 UP BND1      MIX1         200.00000
 UP           MIX2        2500.00000
 LO           MIX3         400.00000
 UP           MIX3         800.00000
 LO           MIX4         100.00000
 UP           MIX4         700.00000
 UP           MIX5        1500.00000
ENDATA
```

8.3 MPS data structures

MPS data structures are internal representation of MPS file.

The structure `MPS` is a root structure, which contains initial pointers to other structures:

```
typedef struct MPS MPS;
struct MPS
{     POOL *pool;
      char *name;
      int n_row;
      int n_col;
      int n_rhs;
      int n_rng;
      int n_bnd;
      MPSROW *row[1+n_row];
      MPSCOL *col[1+n_col];
      MPSCOL *rhs[1+n_rhs];
      MPSCOL *rng[1+n_rng];
      MPSBND *bnd[1+n_bnd];
      MPSQFE *quad;
};
```

where:

`pool` memory pool (see Section 10.3) that holds all other data structures;

`name` problem name (1 to 8 characters);

`n_row` number of rows, `n_row` $\geq$ 1;

`n_col` number of columns, `n_col` $\geq$ 1;

`n_rhs` number of right hand side vectors, `n_rhs` $\geq$ 0;

`n_rng` number of range vectors, `n_rng` $\geq$ 0;

`n_bnd` number of bound vectors, `n_bnd` $\geq$ 0;

`row` array of rows. Location `row[0]` is not used. Location `row[`k`]` is a pointer to kth row ($1 \leq k \leq$ `n_row`). The order of rows is the same as in MPS file;

`col` array of columns. Location `col[0]` is not used. Location `col[`k`]` is a pointer to kth column ($1 \leq k \leq$ `n_col`). The order of columns is the same as in MPS file;

`rhs` array of RHS vectors. Location `rhs[0]` is not used. Location `rhs[`k`]` is a pointer to kth RHS vector ($1 \leq k \leq$ `n_rhs`). The order of RHS vectors is the same as in MPS file;

`rng` array of range vectors. Location `rng[0]` is not used. Location `rng[`k`]` is a pointer to kth range vector ($1 \leq k \leq$ `n_rng`). The order of range vectors is the same as in MPS file;

`bnd` array of bound vectors. Location `bnd[0]` is not used. Location `bnd[`k`]` is a pointer to kth bound vector ($1 \leq k \leq$ `n_bnd`). The order of bound vectors is the same as in MPS file;

`quad` reserved for future use.

The structure `MPSROW` defines a particular row (constraint):

```
typedef struct MPSROW MPSROW;
struct MPSROW
{     char *name;
      char type[2+1];
};
```

where:

`name` row name (1 to 8 characters);

`type` row type (left justified):

- 'N' free (unbounded) row;
- 'L' row of "less than or equal to" type;
- 'G' row of "greater than or equal to" type;
- 'E' row of "equal to" type.

The structure `MPSCOL` defines a particular column (structural variable), right hand sides vector, or range vector:

```
typedef struct MPSCOL MPSCOL;
struct MPSCOL
{     char *name;
      int flag;
      MPSCQE *ptr;
};
```

where:

`name` column or vector name (1 to 8 characters);

`flag` reserved for future use;

`ptr` pointer to the linked list of column or vector elements. The order of elements is the same as in MPS file.

The structure `MPSCQE` defines a particular element of column, right hand sides vector, or range vector:

```
typedef struct MPSCQE MPSCQE;
struct MPSCQE
{     int ind;
      double val;
      MPSCQE *next;
};
```

where:

`ind` row number ($1 \leq$ `ind` $\leq$ `n_row`);

`val` numeric value of the element;

`next` pointer to the next element of the same column, RHS vector, or range vector.

The structure `MPSBND` defines a particular bound vector:

```
typedef struct MPSBND MPSBND;
struct MPSBND
{     char *name;
      MPSBQE *ptr;
};
```

where:

`name` vector name (1 to 8 characters);

`ptr` pointer to the linked list of bound vector elements. The order of elements is the same as in MPS file.

The structure `MPSBQE` defines a particular element of bound vector:

```
typedef struct MPSBQE MPSBQE;
struct MPSBQE
{     char type[2+1];
      int ind;
      double val;
      MPSBQE *next;
};
```

where:

`type` bound type:

- 'LO' lower bound;
- 'UP' upper bound;
- 'FX' fixed variable;
- 'FR' free variable;
- 'MI' no lower bound (lower bound is $-\infty$);
- 'PL' no upper bound (upper bound is $+\infty$).

`ind` column number ($1 \leq$ `ind` $\leq$ `n_col`);

`val` numeric value of element;

`ptr` pointer to the next element of the same bound vector.

8.4 `load_mps` — load MPS file

Synopsis

```
#include "glpmps.h"
MPS *load_mps(char *fname);
```

Description

The `load_mps` routine reads linear programming problem data using MPS format from the text file, whose name is the character string `fname`.

Returns

The `load_mps` routine returns a pointer to the object of MPS type, which is internal representation of MPS file. In case of error the `load_mps` routine prints an appropriate error message to `stderr` and returns `NULL`.

8.5 `free_mps` — free MPS file

Synopsis

```
#include "glpmps.h"
void free_mps(MPS *mps);
```

Description

The `free_mps` routine deletes the object of MPS type, which `mps` points to, freeing all memory allocated to this object.

9 GLPRFI: Reid Form of the Inverse (RFI)

9.1 Prelude

One of the most essential components of any implementation of the revised simplex method for linear programming is the representation of the basis matrix[1], which allow performing the following four main operations:

- building the representation of a given basis matrix B;
- updating the representation if one column of B has been replaced by other column;
- forward transformation of a given vector z by means of solving the system $Bz' = z$;
- backward transformation of a given vector z by means of solving the system $B^T z' = z$.

This chapter describes routines implementing *Reid Form of the Inverse (RFI)*, which is the representation of basis matrix proposed by J. K. Reid [1]. (See also Chapter 4, which describes other representation of basis matrix.)

RFI is the following four:

$$B = (H, V, P, Q), \tag{9.1}$$

where B is the current basis matrix, H is a matrix in the form of eta file (see Section 5.1), V is sparse matrix, P and Q are permutation matrices.

Also it is assumed that

$$B = HV \tag{9.2}$$

and

$$PVQ = U, \tag{9.3}$$

where U is upper triangular matrix (U is not stored explicitly).

(It should be understood that (9.1)—(9.3) is some variant of LU factorization. In fact, $B = HV = HP^{-1}UQ^{-1}$, i.e. $PBQ = (PHP^{-1})U$, and it may be shown that PHP^{-1} is lower triangular matrix.)

Let's assume that the matrix U is not upper triangular. In this case we can apply gaussian elimination procedure in order to restore its upper triangular form.

Gaussian elimination procedure includes elementary transformations of the following three kinds: row permutation, column permutation, and subtraction one row multiplied by a number from other row (gaussian transformation). Row and column permutations are used for pivoting, gaussian transformations are used for eliminating subdiagonal elements. (See also Chapter 6.)

Each time when RFI routines apply elementary transformations to the matrix U, they also apply corresponding inverted transformations to the matrices H, V, P, and Q in order to keep main equalities (9.2) and (9.3). Note that the matrix U has implicit representation. This means that all transformations are actually applied to the matrix V, which differs from U only in the order of rows and columns determined by permutation matrices P and Q.

Let's assume that kth and pth rows of the matrix U have been permuted. In order to compensate this permutation and to keep the equality (9.3) we should also permute

[1] *Basis matrix* or just *basis* is a quadratic non-singular matrix formed by columns of basis variables.

kth and pth rows of the permutation matrix P. Similarly, if kth and qth columns of the matrix U have been permuted, kth and qth columns of the permutation matrix Q should be permuted. Note that rows and columns permutations don't affect on the matrix V, i.e. the order of rows and columns of V is not changed.

Now let's assume that in order to eliminate subdiagonal element u_{ki} kth row of the matrix U multiplied by $f = u_{ki}/u_{kk}$ (gaussian multiplier) has been subtracted from its ith row. Because the matrix U has implicit representation, this transformation should be actually applied to the matrix V, i.e. it is necessary to subtract row k' multiplied by f from row i', where row numbers k' and i' can be determined by row numbers k and i through permutation matrices P and Q.

This elementary gaussian transformation of the matrix V is equivalent to multiplying the following elementary matrix Φ from the left on the matrix V:

$$\Phi = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & -f_{i'k'} & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}, \tag{9.4}$$

At the same time in order to keep the equality (9.2) we should multiply the matrix Φ^{-1} from the right on the matrix H, where Φ^{-1} is the following:

$$\Phi^{-1} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & +f_{i'k'} & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}, \tag{9.5}$$

And so far the matrix H has the from of eta file, this multiplication is equivalent to adding new eta term, which Φ^{-1} is (for details see Chapter 5).

If it is required to initially compute RFI for the given basis matrix B, RFI routines just set $H = P = Q = I, V = B$, and apply general gaussian elimination procedure in order to transform the matrix $U = V$ to upper triangular form.

Now let there is the current basis matrix B, for which the RFI representation (9.1)—(9.3) is already known. Let also $\overline{B}$ is the adjacent basis matrix, which differs from B only in pth column B_p. From (9.2) it follows that

$$\overline{B} = H\overline{V}, \tag{9.6}$$

where $\overline{V}$ also differs from V only in one pth column V_p. Moreover, $B_p = HV_p$, therefore

$$V_p = H^{-1}B_p. \tag{9.7}$$

Thus, replacing pth column of the matrix V on new column V_p (9.7) automatically gives the representation (9.1)—(9.3) for the adjacent basis matrix $\overline{B}$. However, it also involves that the matrix U becomes not upper triangular. In order to restore upper triangular form of U the general gaussian elimination procedure could be used. On the other hand, it is obvious that new matrix U has the special structure, because it differs from upper triangular matrix only in one column (in fact, this matrix has the same rows and columns as the matrix V, only in other order). Due to this updating RFI can be performed more efficiently if to use special versions of gaussian elimination like Bartels—Golub or Forrest—Tomlin techniques (see Subsections 6.1.2 and 6.1.3).

Comparing RFI and EFI (see Chapter 4) it may be noted that RFI is generally able to producesparser representation, because it allows pivoting on reinverting as well as on updating the representation unlike EFI, which allows pivoting only on reinverting the representation.

From (9.2)—(9.3) it follows that $B^{-1} = V^{-1}H^{-1}$ and[2] $B^{-T} = H^{-T}V^{-T}$, where $V^{-1} = QU^{-1}P$ and $V^{-T} = P^{-1}U^{-T}Q^{-1}$. Therefore forward and backward transformations can be efficiently performed by solving systems with matrices of easy structures.

RFI is defined by the following structure:

```
typedef struct RFI RFI;
{     int m;
      HFILE *H;
      MAT *V;
      PER *P;
      PER *Q;
      int nzH0;
      int nzV0;
      int reid;
      int tech;
      double col[1+m];
      int flag;
      DUFF *rs, cs;
      double work[1+m];
};
```

where:

`m` order of basis matrix B;

`H` matrix H in the form eta file;

`V` matrix V in the sparse format;

`P` left permutation matrix P;

`Q` right permutation matrix Q;

`nzH0` size of the matrix H (number of eta terms) at the last reinversion;

`nzV0` size of the matrix V (number of non-zeros) at the last reinversion;

`reid` if this flag is set, Reid's technique is used in order to minimize the bump of the matrix $U = PVQ$ on updating the representation (see Subsection 6.1.4);

[2] B^{-T} means $(B^T)^{-1}$

`tech` updating technique flag:

- `RFI_BG` use Bartels—Golub updating technique (see Subsection 6.1.3);
- `RFI_FT` use Forrest—Tomlin updating technique (see Subsection 6.1.4).

`col` partially transformed column V_p (9.7), which has to be used on updating RFI for adjacent basis matrix $\overline{B}$;

`flag` column flag (this flag is reset to 0 after each updating RFI):

- 0 column V_p is not ready yet;
- 1 column V_p is ready for updating.

`rs, cs` Duff schemes used on building the representation by means of the `gel` routine (see Subsection 6.1.2).

`work` working storage.

(The column V_p is an intermediate result, which appears on performing forward transformation of column of that non-basis variable, which enters the adjacent basis. In case of the primal simplex method that transformed column is used on choosing basis variables that leaves the current basis, therefore keeping V_p after forward transformation allows not to compute this column on subsequent updating RFI.)

9.2 `create_rfi` — create RFI

Synopsis

```
#include "glprfi.h"
RFI *create_rfi(int m);
```

Description

The `create_rfi` routine creates RFI (9.1)—(9.3) for representing basis matrix of order m. Initially RFI corresponds to unity matrix, i.e. $H = V = P = Q = I$.

Returns

The `create_rfi` routine returns a pointer to the created RFI.

9.3 `build_rfi` — build RFI for given basis matrix

Synopsis

```
#include "glprfi.h"
int build_rfi(RFI *rfi, MAT *A, indb[]);
```

Description

The `build_rfi` routine builds RFI for the given basis matrix B (see below) in the form (9.1)—(9.3). It sets $H = P = Q = I$ (unity matrix) and $V = B$ and then calls the `gel` routine (see Subsection 6.1.1) in order to transform the matrix $U = PVQ$ to upper triangular form by means of gaussian elimination. This operation is usually performed if the current RFI becomes inaccurate or too large, or the basis matrix is completely changed.

The basis matrix B is specified implicitly by means of the sparse matrix A and the list of basis columns `indb`. The matrix A should have m rows, where m is order of basis matrix. Locations `indb[1]`, `indb[2]`, ..., `indb[m]` should contain numbers of basis columns of the matrix A, which specify the basis matrix B.

Returns

If RFI has been successfully built, the `build_rfi` routine returns zero. Otherwise, if the matrix B is singular or ill conditioned, the routine returns non-zero. In the latter case partial LU factorization can be used for further analysis of the matrix B (for details see Chapter 6).

9.4 `update_rfi` — update RFI for adjacent basis matrix

Synopsis

```
#include "glprfi.h"
int update_rfi(RFI *rfi, int p);
```

Description

The `update_rfi` routine recomputes RFI, which `rfi` points to and which corresponds to the current basis matrix B, in order that it corresponds to the adjacent basis matrix $\overline{B}$ (see Subsection 9.1).

The parameter `p` specifies the number of column of the basis matrix B, which is replaced by new column of the adjacent basis matrix $\overline{B}$.

New basis column is passed implicitly. The `update_rfi` routine assumes that new basis column was prepared before by means of the `rfi_ftran` routine (see Section 4.5).

Returns

The `update_rfi` routine returns one of the following error codes:

0 RFI has been successfully updated;

1 RFI is inaccurate;

2 RFI is too large.

In case of non-zero error code the calling program should reinvert the basis matrix by means of the `build_rfi` routine (see Section 9.3).

9.5 rfi_ftran — perform forward transformation (FTRAN)

Synopsis

```
#include "glprfi.h"
double *rfi_ftran(RFI *rfi, double z[], int save);
```

Description

The `rfi_ftran` routine performs so called *forward transformation (FTRAN)* of a given vector, i.e. it solves the system $Bz' = z$, where B is the current basis matrix represented by RFI, which `rfi` points to.

On entry the array `z` should contain elements of the input vector z in locations `z[1]`, `z[2]`, ..., `z[m]`, where m is order of the basis matrix. On exit the array `z` contains elements of the transformed vector z' in the same locations.

The parameter `save` is a flag. If this flag is set, it means that the input vector z is a column corresponding to non-basis variable, which has been chosen to enter the basis. In this case the `rfi_ftran` routine saves the partially transformed column V_p (see (9.7)) in order that it could be used later by the `update_rfi` routine (see Section 9.4) on updating RFI. It is assumed that the calling program performs at least one call to the `rfi_ftran` routine with the `save` flag set.

Returns

The `rfi_ftran` routine returns a pointer to the array `z`.

9.6 rfi_btran — perform backward transformation (BTRAN)

Synopsis

```
#include "glprfi.h"
double *rfi_btran(RFI *rfi, double z[]);
```

Description

The `rfi_btran` routine performs so called *backward transformation (BTRAN)* of a given vector, i.e. it solves the system $B^T z' = z$, where B is the current basis matrix represented by RFI, which `rfi` points to.

On entry the array `z` should contain elements of the input vector z in locations `z[1]`, `z[2]`, ..., `z[m]`, where m is order of the basis matrix. On exit the array `z` contains elements of the transformed vector z' in the same locations.

Returns

The `rfi_ftran` routine returns a pointer to the array z.

9.7 `delete_rfi` — delete RFI

Synopsis

```
#include "glprfi.h"
void delete_rfi(EFI *rfi);
```

Description

The `delete_rfi` routine deletes RFI, which `rfi` points to, freeing all the memory allocated to this object.

References

1. *J. K. Reid.* A Sparsity Exploiting Variant of the Bartels—Golub Decomposition for Linear Programming Bases. Mathematical Programming, 24, pp. 55—69, 1982.

10 GLPSET: Auxiliary Routines

This chapter describes auxiliary routines, which are used by most other GLPK routines. However, these routines may be useful from the user's point of view.

10.1 Communicative Routines

GLPK includes three communicative routines, which are used by all other GLPK routines. The communicative routines were introduced to the package mainly in order to provide its portability on other platforms that differ from the standard C environment. For example, all GLPK routines report error messages by means of the `error` routine, which in case of the standard C environment sends an error message to `stderr`. Therefore the user can easily change such behavior of GLPK routines replacing the `error` routine by its own version, which, for example, sends a message to the error window instead `stderr`. Of course, if the user makes such replacement, it should keep the external behavior of the original communicative routines.

10.1.1 `print` — print informative message

Synopsis

```
#include "glpset.h"
void print(char *fmt, ...);
```

Description

The `print` routine sends an informative message specified by the format control string `fmt` and variable parameter list to the standard output (`stdout`). Note that the `print` routine automatically appends `LF` control character to the end of output.

10.1.2 `error` — print error message and continue program execution

Synopsis

```
#include "glpset.h"
void error(char *fmt, ...);
```

Description

The `error` routine sends an error message specified by the format control string `fmt` and variable parameter list to `stderr` and returns to the calling program. Note that the error routine automatically appends `LF` control character to the end of output.

10.1.3 `fault` — print error message and terminate program execution

Synopsis

```
#include "glpset.h"
void fault(char *fmt, ...);
```

Description

The `fault` routine sends an error message specified by the format control string `fmt` and variable parameter list to `stderr` and abnormally terminates execution of the program. Note that the `fault` routine automatically appends `LF` control character to the end of output.

10.2 Memory Management Routines

GLPK includes three low level memory management routines `umalloc`, `ucalloc`, and `ufree`, which are used by all other GLPK routines instead the standard C routines `malloc`, `calloc`, and `free` respectively. These routines were introduced to the package in order to: a) simplify error handling (if there is no free memory, `umalloc` and `ucalloc` just report an error and abnormally terminate execution of the program); and b) provide some additional features (see below), which are not supported by the standard C routines.

Each memory block allocated by `umalloc` and `ucalloc` routines includes a prefix, which is called *memory block descriptor*. This descriptor has the following format:

```
struct mem_desc
{     int size;
      int flag;
      /* actual data */
};
```

where:

`size` total size of a memory block (including descriptor) in bytes;

`flag` descriptor flag (a value, which is used by the `ufree` routine for checking whether a pointer to memory block is valid).

Actual data of variable size, a pointer to which is returned by the `umalloc` and `ucalloc` routines, are placed immediately after the memory block descriptor.

There are also several external variables that are related to dynamic memory allocation. These variables are the following:

`int mem_flag;`
a value used as the memory block descriptor flag;

`int mem_limit;`
maximal amount of memory (in bytes), which is available for dynamic allocation. Initially this variable is set to `INT_MAX`. The user may decrease this value, for example, in order to prevent abnormal page swapping on virtual memory

systems if the storage required by the program exceeds amount of the real memory (in this case `mem_limit` should be set to 80-90% of amount of available real memory);

`int mem_total;`
: total amount of memory (in bytes), which is currently allocated. This value is the sum of `size` fields of all memory block descriptors and may be used for statistics;

`int mem_tpeak;`
: peak value of `mem_total` reached during program execution. This value may be used for statistics;

`int mem_count;`
: total number of currently allocated memory block. This value may be used for statistics;

`int mem_cpeak;`
: peak value of `mem_count` reached during program execution. This value may be used for statistics.

The application program should not change these external variables (except `mem_limit`).

10.2.1 `umalloc` — allocate memory block

Synopsis

```
#include "glpset.h"
void *umalloc(int size);
```

Description

The `umalloc` routine allocates a memory block of `size` bytes long and clears this block by binary zeros. Failure of the operation involves abnormal program termination.

Returns

The `umalloc` routine returns a pointer to the allocated memory block. In order to free this block the `ufree` routine (not `free`!) should be used (see Subsection 10.2.3).

10.2.2 `ucalloc` — allocate memory block

Synopsis

```
#include "glpset.h"
void *ucalloc(int nmemb, int size);
```

Description

The `ucalloc` routine allocates a memory block of `nmemb`×`size` bytes long and clears this block by binary zeros. Failure of the operation involves abnormal program termination.

Returns

The `umalloc` routine returns a pointer to the allocated memory block. In order to free this block the `ufree` routine (not `free`!) should be used (see Subsection 10.2.3).

10.2.3 `ufree` — free memory block

Synopsis

```
#include "glpset.h"
void ufree(void *ptr);
```

Description

The `ufree` routine frees the memory block, which `ptr` points to and which was previously allocated by the `umalloc` or the `ucalloc` routine (see Subsections 10.2.1 and 10.2.2).

10.3 Memory Pools

In many applications it is necessary to organize information in the form of linked lists. Such data structures require specific memory allocation, because they usually consist of a lot of elementary data blocks of small size (several tens of bytes or even less). This makes use of the standard routines like `malloc` and `free` to be inefficient, because small size of data blocks involves high overhead expenses and may cause pathological fragmentation of the main memory. Therefore in order to avoid this negative phenomenon many GLPK routines organize data structures using so called memory pools.

Memory pool is an auxiliary data structure, which allows allocating and freeing atoms (i.e. elementary data blocks) of fixed size for a fixed time.

Each memory pool includes three linked lists:

the list of free atoms;

the list of free blocks;

the list of used blocks.

In order to allocate an atom the memory pool routine looks through the list of free atoms. This operation requires a fixed time, because all atoms have the same size, so the first free atom is fitted. If the list of free atoms is empty, the routine pulls the first free block (all blocks have the same large enough size) and splits it into free atoms. Free blocks are drawn from the list of free blocks or, if it is empty, allocated by means of the `umalloc` routine (see Subsection 10.2.1). In both cases the routine pushes the block to the list of used blocks. In order to free an atom the memory pool routine just returns it to the list of free atoms. Note that the total amount of memory allocated to a pool always increases and never decreases (until the pool will be deleted).

Memory pools give the following two main advantages: allocating and freeing atoms require a fixed time, because all atoms have the same size; overhead expenses and memory fragmentation remain low, because all blocks allocated by the control program (operating system) have the same large enough size.

Memory pools also may hold atoms of variable (different) size. However, in this case atoms can't be freed separately; it is allowed freeing only all atoms at the same time. Such pools may be used in that cases, when information is only added to data structures, but not removed.

Memory pools are described by the `POOL` structure:

```
typedef struct POOL POOL;
struct POOL
{     int size;
      void *avail;
      void *link;
      int used;
      void *stock;
      int count;
};
```

where:

`size` if atoms are of fixed size, this field contains the size of atom ($1 \leq size \leq 256$). If atoms are of variable size, this field contains zero;

`avail` pointer to the list of free atoms;

`link` pointer to the list of used blocks (it points to the last recently allocated block, which may have free space for allocating free atoms);

`used` number of bytes used in the block, which `link` points to;

`stock` pointer to the list of free blocks;

`count` total number of allocated atoms (i.e. of atoms, which belong to this pool, but missing from the list of free atoms).

Remarks.

1. The first `sizeof(void *)` bytes of each free atom contain a pointer to the previous free atom.
2. The first `sizeof(void *)` bytes of each used (free) block contain a pointer to the previous used (free) block.
3. All blocks have the same size defined by the preprocessor variable `POOL_SIZE`.

10.3.1 `create_pool` — create memory pool

Synopsis

```
#include "glpset.h"
POOL *create_pool(int size);
```

Description

The `create_pool` routine creates an empty memory pool, which holds no atoms. The parameter `size` specifies size of atoms in bytes. If $1 \leq$ `size` ≤ 256, all atoms are of `size` bytes long. If `size` $= 0$, atoms may have different size.

Returns

The `create_pool` routine returns a pointer to the created memory pool.

10.3.2 `get_atom` — allocate atom of fixed size

Synopsis

```
#include "glpset.h"
void *get_atom(POOL *pool);
```

Description

The `get_atom` routine allocate an atom from the specified memory `pool` and clears the atom by binary zeros. The size of atom should be specified on creating the pool (see Subsection 10.3.1).

Complexity

This operation has time complexity $O(1)$.

Returns

The `get_atom` routine returns a pointer to the allocated atom.

10.3.3 `free_atom` — free atom

Synopsis

```
#include "glpset.h"
void free_atom(POOL *pool, void *ptr);
```

Description

The `free_atom` routine frees the atom, which `ptr` points to, returning it to the list of free atoms of the specified memory `pool`. It is assumed that the atom was allocated from the same memory pool by means of the `get_atom` routine (see Subsection 10.3.2); otherwise the behavior is undefined.

Complexity

This operation has time complexity $O(1)$.

10.3.4 `get_atomv` — allocate atom of variable size

Synopsis

```
#include "glpset.h"
void *get_atomv(POOL *pool, int size);
```

Description

The `get_atomv` routine allocates an atom from the specified memory `pool` and clears the atom by binary zeros. The pool should be created by the `create_pool` routine (see Subsection 10.3.1) with `size` = 0. The actual size of atom in bytes is specified by the parameter `size`, which should be positive and not greater than 256.

Complexity

This operation has time complexity $O(1)$.

Returns

The `get_atomv` routine returns a pointer to the allocated atom.

10.3.5 `clear_pool` — free all atoms

Synopsis

```
#include "glpset.h"
void clear_pool(POOL *pool);
```

Description

The `clear_pool` routine free all atoms that belong to the specified memory `pool` and were allocated by means of the routines `get_atom` or `get_atomv` (see Subsection 10.3.2 and 10.3.4).

Note that the routine performs this operation just moving all used (allocated) blocks to the list of free blocks, hence no memory block are returned to the control program (operating system).

10.3.6 `delete_pool` — delete memory pool

Synopsis

```
#include "glpset.h"
void delete_pool(POOL *pool);
```

Description

The `clear_pool` routine deletes the specified memory `pool`, freeing all the memory allocated to this object.

11 GLPSTR: Operations on Charater Strings

11.1 Ordinary Character Strings

This section describes routines, which perform some operations on ordinary character strings and complement standard C routines.

11.1.1 `str2int` — convert character string to value of integer type

Synopsis

```
#include "glpstr.h"
int str2int(char *str, int *val);
```

Description

The `str2int` routine converts the character string `str` to a value of integer type and stores this value to the location, which `val` points to. In case of error the location remains unchanged.

Returns

The `str2int` routine returns one of the following error codes:

0 no error;

1 value is out of range;

2 character string is syntactically incorrect.

11.1.2 `str2dbl` — convert character string to value of double type

Synopsis

```
#include "glpstr.h"
int str2dbl(char *str, double *val);
```

Description

The `str2dbl` routine converts the character string `str` to a value of double type and stores this value to the location, which `val` points to. In case of error the location remains unchanged.

Returns

The `str2dbl` routine returns one of the following error codes:

0 no error;

1 value is out of range;

2 character string is syntactically incorrect.

11.1.3 strspx — remove all spaces from character string

Synopsis

```
#include "glpstr.h"
char *strspx(char *str);
```

Description

The `strspx` routine removes all space characters from the character string `str`.

Examples

```
strspx("   Errare   humanum   est   ") => "Errarehumanumest"
strspx("      ")                        => ""
```

Returns

The `strspx` routine returns a pointer to the character string `str`.

11.1.4 strtrim — remove trailing spaces from character string

Synopsis

```
#include "glpstr.h"
char *strtrim(char *str);
```

Description

The `strtrim` routine removes trailing space characters from the character string `str`.

Examples

```
strtrim("Errare humanum est   ") => "Errare humanum est"
strtrim("      ")                => ""
```

Returns

The `strtrim` routine returns a pointer to the character string `str`.

11.2 Segmented Character Strings

Segmented character string is a character string, which is represented in the form of linked list, where each element of linked list represents a particular segment of string.

In cases of intensive creating, modifying and deleting character strings during program execution this data structure allows greatly decreasing loadness on the system mechanism of dynamic memory allocation. However, this data structure involves some overhead expenses of main memory, because each linked list element needs a pointer to the next element.

In order to represent segmented character string on the whole the `STR` structure is used:

```
typedef struct STR STR;
struct STR
{     POOL *pool;
      int len;
      SQE *head;
      SQE *last;
};
```

where:

`pool` memory pool (see Section 10.3) that holds all elements of the string. Note that the same pool can be shared between several strings;

`len` current string length, len $\geq$ 0;

`head` pointer to the first string element;

`tail` pointer to the last string element.

A particular element of segmented character string is defined by the structure `SQE`:

```
typedef struct SQE SQE;
struct SQE
{     char data[SQE_SIZE];
      SQE *next;
};
```

where:

`data` characters allocated in this element;

`next` pointer to the next string element.

Remarks.

1. Segmented character string may contain any control characters including `NUL` control character ('`\0`'). This allows representing text data as well as binary data.
2. Number of characters allocated in one string element is defined by the preprocessor variable `SQE_SIZE`.
3. Only the last element of segmented character string can be partially filled, i.e. there are no "holes" or unnecessary elements.
4. All freed elements are returned to the memory pool, which is connected with the corresponding segmented character string.

11.2.1 `clear_str` — clear segmented character string

Synopsis

```
#include "glpstr.h"
STR *clear_str(STR *str);
```

Description

The `clear_str` routine makes the segmented character string `str` be empty.

Returns

The `clear_str` routine returns a pointer to the string `str`.

11.2.2 `compare_str` — compare segmented character strings

Synopsis

```
#include "glpstr.h"
int compare_str(STR *str1, STR *str2);
```

Description

The `compare_str` routine lexicographically compares the segmented character string `str1` and `str2`. If compared strings have different length, shorter string is considered as if it would be padded by binary zeros to the length of longer string.

Returns

The `compare_str` returns one of the following values:

< 0 `str1` lexicographically less than `str2`;

= 0 `str1` and `str2` are identical;

> 0 `str1` lexicographically greater than `str2`.

11.2.3 `create_str` — create segmented character string

Synopsis

```
#include "glpstr.h"
STR *create_str(POOL *pool);
```

Description

The `create_str` routine creates empty segmented character string.

The parameter `pool` should specify a memory pool (see Section 10.3), which is created by the `create_str_pool` routine (see Subsection 11.2.4). The `create_str` routine connects this pool with the created string, therefore this pool will be used in all subsequent operations on this string. Note that several strings may use the same memory pool.

Returns

The `create_str` routine returns a pointer to the created string.

11.2.4 `create_str_pool` — create pool for segmented character strings

Synopsis

```
#include "glpstr.h"
POOL *create_str_pool(void);
```

Description

The `create_str_pool` routine creates a memory pool (see Section 10.3), which may be used to hold elements of segmented character strings.

In order to free all memory allocated to memory pool the `delete_pool` routine should be used (see Subsection 10.3.6). Note that deleting memory pool involves deleting all segmented character string, which refer to this pool.

Returns

The `create_str_pool` routine returns a pointer to the created memory pool.

11.2.5 `delete_str` — delete segmented character string

Synopsis

```
#include "glpstr.h"
void delete_str(STR *str);
```

Description

The `delete_str` routine deletes the segmented character string `str`, returning all memory allocated to this string to the corresponding memory pool.

Note that allocated memory is not returned to the control program (operating system) while the corresponding memory pool exists.

11.2.6 `get_str` — extract value of segmented character string

Synopsis

```
#include "glpstr.h"
char *get_str(char *to, STR *str);
```

Description

The `get_str` routine copies data from the segmented character string `str` to the ordinary character string `to`. The array of `char` type, which `to` points to, should have at least $len+1$ characters, where *len* is the current length of the string `str`.

Returns

The `get_str` routine returns a pointer to the string `to`.

11.2.7 `set_str` — assign value to segmented character string

Synopsis

```
#include "glpstr.h"
STR *set_str(STR *str, char *from);
```

Description

The `set_str` routine copies data from the ordinary character string `from` to the segmented character string `str`.

Returns

The `set_str` routine returns a pointer to the string `str`.

Index

A

app_term ... 32
AVLNODE ... 4
AVLTREE ... 4

B

build_efi ... 27
build_rfi ... 102

C

check_mat ... 66
check_per ... 67
clear_line ... 67
clear_mat ... 68
clear_pool ... 113
clear_str ... 117
compare_str ... 118
copy_mat ... 68
copy_per ... 69
count_nz ... 69
create_avl ... 5
create_duff ... 22
create_efi ... 27
create_eta ... 32
create_lu ... 56
create_mat ... 70
create_per ... 70
create_pool ... 111
create_rfi ... 102
create_str ... 118
create_str_pool ... 119

D

delete_avl ... 18
delete_duff ... 23
delete_efi ... 29
delete_eta ... 35
delete_lu ... 62
delete_mat ... 70
delete_node ... 14
delete_per ... 71
delete_pool ... 113
delete_rfi ... 105
delete_str ... 119
DUFF ... 21

E

EFI ... 26
efi_btran ... 29
efi_ftran ... 28
ELEM ... 64
eq_scaling ... 71
error ... 107
exclude_obj ... 22

F

fault ... 108
find_by_key ... 11
find_by_pos ... 14
free_atom ... 112
free_mps ... 98

G

gel ... 37
gel_bg ... 43
gel_ft ... 47
get_atom ... 112
get_atomv ... 112
get_str ... 119
gm_scaling ... 72

H

h_solve ... 33
HFILE ... 31
ht_solve ... 34
HTERM ... 31

I

include_obj ... 22
insert_by_key ... 6
insert_by_pos ... 13
inv_per ... 73
iper_vec ... 73

L

l_solve ... 74
load_mps ... 97
lt_solve ... 74
LU ... 55

M

MAT ... 63
mat_per ... 75
mat_vec ... 76
mem_count ... 109
mem_cpeak ... 109
mem_desc ... 108
mem_flag ... 108
mem_limit ... 108
mem_total ... 109
mem_tpeak ... 109
min_bump ... 52
monit_lu ... 61
MPS ... 95
MPSBND ... 97
MPSBQE ... 97
MPSCOL ... 96
MPSCQE ... 96
MPSROW ... 96

N

new_elem ... 76
next_by_key ... 12
next_node ... 7

P

PER ... 65
per_mat ... 77
per_vec ... 77
POOL ... 111
prev_node ... 10
print ... 107

R

reset_duff ... 23
reset_eta ... 34
reset_per ... 78
RFI ... 101
rfi_btran ... 104
rfi_ftran ... 104
rotate_avl ... 16

S

scrape_mat ... 78
set_str ... 120
solve_lu ... 61
sort_mat ... 79
SQE ... 117
STR ... 116
str2dbl ... 115
str2int ... 115
strspx ... 116
strtrim ... 116
submatrix ... 79

T

test_mat_d ... 80
test_mat_e ... 80
tmat_vec ... 81
trn_mat ... 82

U

u_solve ... 82
ucalloc ... 109
ufree ... 110
umalloc ... 109
update_efi ... 27
update_rfi ... 103
ut_solve ... 83

V

v_solve ... 84
vt_solve ... 85