# tAltris

v1.0

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# Data Structure Index

## 1.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 2

# File Index

## 2.1 File List

Here is a list of all files with brief descriptions:

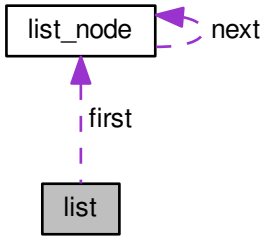
# Chapter 3

# Data Structure Documentation

## 3.1 list Struct Reference

`#include <list.h>`

Collaboration diagram for list:




### Data Fields

- size_t **length**
- struct **list_node** * **first**

### 3.1.1 Detailed Description

Head of a singly-linked list.

### 3.1.2 Field Documentation

#### 3.1.2.1 first

```
struct list_node* first
```

First node.

#### 3.1.2.2 length

```
size_t length
```

List length.

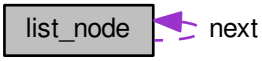
The documentation for this struct was generated from the following file:

- src/utils/ **list.h**

## 3.2 list_node Struct Reference

```
#include <list.h>
```

Collaboration diagram for list_node:




### Data Fields

- struct **list_node** * **next**

### 3.2.1 Detailed Description

A node of a singly-linked list.

### 3.2.2 Field Documentation

#### 3.2.2.1 next

```
struct list_node* next
```

Next node.

The documentation for this struct was generated from the following file:

- src/utils/ **list.h**

## 3.3 matrix Struct Reference

```
#include <matrix.h>
```

### Data Fields

- size_t **rows**
- size_t **cols**
- double * **data**

### 3.3.1 Detailed Description

Matrix structure

### 3.3.2 Field Documentation

#### 3.3.2.1 cols

```
size_t cols
```

Columns

#### 3.3.2.2 data

```
double* data
```

Values

#### 3.3.2.3 rows

```
size_t rows
```

Rows

The documentation for this struct was generated from the following file:

- src/utils/ **matrix.h**

# Chapter 4

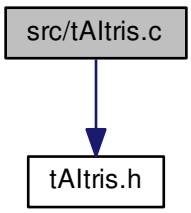
# File Documentation

## 4.1 src/tAItris.c File Reference

Main file.

```
#include "tAItris.h"
```

Include dependency graph for tAItris.c:




### Functions

- int **main** ()

### 4.1.1 Detailed Description

Main file.

**Author**

S4MasterRace

**Version**

1.0

### 4.1.2 Function Documentation

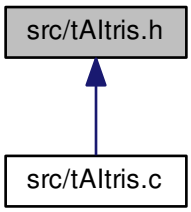
#### 4.1.2.1 main()

```
int main ( )
```

## 4.2 src/tAltris.h File Reference

Main file.

This graph shows which files directly or indirectly include this file:




### 4.2.1 Detailed Description

Main file.

**Author**

S4MasterRace

**Version**

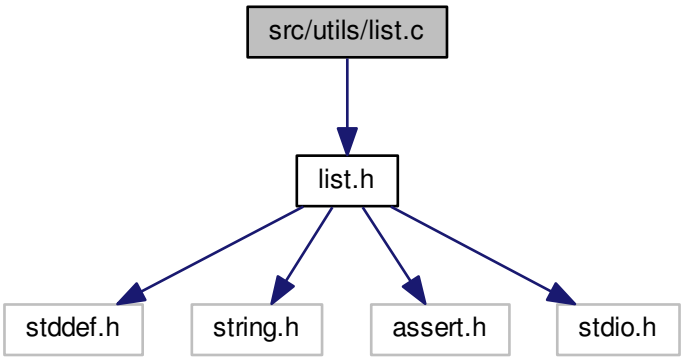
1.0

## 4.3 src/utils/list.c File Reference

Intrusive list implement.

```
#include "list.h"
```

Include dependency graph for list.c:




### Functions

- void **list_init** (struct **list** * **list**)
- size_t **list_length** (const struct **list** * **list**)
- struct **list_node** * **list_first** (const struct **list** * **list**)
- struct **list_node** * **list_last** (const struct **list** * **list**)
- struct **list_node** * **list_next** (const struct **list_node** *node)
- struct **list_node** * **list_advance** (struct **list_node** *node, size_t distance)
- struct **list_node** * **list_at** (const struct **list** * **list**, size_t pos)
- void **list_reverse** (struct **list** * **list**)
- void **list_swap** (struct **list** *l1, struct **list** *l2)
- void **list_split_at** (struct **list** * **list**, size_t pos, struct **list** *right)
- void **list_concat** (struct **list** *l1, struct **list** *l2)
- void **list_sort** (struct **list** * **list**, int(*cmp)(struct **list_node** *, struct **list_node** *))
- int **list_is_empty** (const struct **list** * **list**)
- void **list_add** (struct **list** * **list**, struct **list_node** *node)
- void **list_append** (struct **list** * **list**, struct **list_node** *node)
- void **list_insert_after** (struct **list** * **list**, struct **list_node** *curr, struct **list_node** *node)
- void **list_insert_at** (struct **list** * **list**, struct **list_node** *node, size_t pos)
- void **list_del** (struct **list** * **list**)
- void **list_del_after** (struct **list** * **list**, struct **list_node** *node)
- void **list_del_at** (struct **list** * **list**, size_t pos)
- void **list_print** (const struct **list** * **list**)

### 4.3.1 Detailed Description

Intrusive list implement.

**Author**

S4MasterRace

**Version**

1.0

### 4.3.2 Function Documentation

#### 4.3.2.1 list_add()

```
void list_add (
            struct  list * list,
            struct  list_node * node )  [inline]
```

Adds `node` in the front of `list`

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | the new node. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(1)

#### 4.3.2.2 list_advance()

```
struct  list_node* list_advance (
            struct  list_node * node,
            size_t distance )
```

Returns the nth-node after the current one.

**Parameters**

| | |
|---|---|
| *node* | a node. |
| *distance* | distance to move on. |

**Returns**

the nth-node after `node`.

**Precondition**

`node` must be not NULL.

**Remarks**

Complexity: O(n)

#### 4.3.2.3 list_append()

```
void list_append (
            struct  list * list,
            struct  list_node * node )  [inline]
```

Adds `node` at the end of `list`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | the new node. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(n)

#### 4.3.2.4 list_at()

```
struct  list_node* list_at (
             const struct  list * list,
             size_t pos )
```

Returns node at the position `pos`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *pos* | position (0-based) of the node. |

**Returns**

the node at the position `pos`.

**Precondition**

`list` must be not NULL.
`list` must be not empty.
`pos` must be in [0; list_length(list)[.

**Remarks**

Complexity: O(N)

#### 4.3.2.5 list_concat()

```
void list_concat (
            struct  list * l1,
            struct  list * l2 )  [inline]
```

Concatenates two lists.

**Parameters**

| | |
|---|---|
| *l1* | list 1. |
| *l2* | list 2. |

**Precondition**

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

**Postcondition**

`l2` is reset to an empty list.

**Remarks**

Complexity: O(N)

#### 4.3.2.6 list_del()

```
void list_del (
            struct  list * list )  [inline]
```

Deletes the first node.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Precondition**

`list` must be not NULL.
`list` must be not empty.

**Postcondition**

List size decreases by 1.

**Remarks**

Complexity: O(1)

#### 4.3.2.7 list_del_after()

```
void list_del_after (
            struct  list * list,
            struct  list_node * node )  [inline]
```

Deletes the node at after the node `curr`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | a node of `list`. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.
`list` must be not empty.
`node` must a node of `list`.

**Postcondition**

List size decreases by 1.

**Remarks**

Complexity: O(1)

#### 4.3.2.8 list_del_at()

```
void list_del_at (
            struct  list * list,
            size_t pos )  [inline]
```

Deletes the node at the position `pos`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *pos* | index (0-based) of the node to delete. |

**Precondition**

`list` must be not NULL.
`list` must be not empty.
`pos` must be in [0; list_length(list)[.

**Postcondition**

List size decreases by 1.

**Remarks**

Complexity: O(n)

#### 4.3.2.9 list_first()

```
struct  list_node* list_first (
            const struct  list * list )
```

Returns the first node.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

the first node.

**Precondition**

`list` must be not NULL.
`list` must be not empty.

**Remarks**

Complexity: O(1)

#### 4.3.2.10 list_init()

```
void list_init (
            struct  list * list )  [inline]
```

Initializes the list.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Precondition**

`list` must be not NULL.

**Postcondition**

`list` is empty.
`list` has a size of 0.

**Remarks**

Complexity: O(1)

#### 4.3.2.11 list_insert_after()

```
void list_insert_after (
            struct  list * list,
            struct  list_node * curr,
            struct  list_node * node )  [inline]
```

Inserts `node` at after the node `curr`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *curr* | a node of `list`. |
| *node* | new node. |

**Precondition**

`list` must be not NULL.
`curr` must be not NULL.
`curr` must a node of `list`.
`node` must be not NULL.

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(1)

#### 4.3.2.12 list_insert_at()

```
void list_insert_at (
            struct  list * list,
            struct  list_node * node,
            size_t pos )  [inline]
```

Inserts `node` at the position `pos` in `list`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | new node. |
| *pos* | position (0-based) where to insert the new node. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.
`pos` must be in [0; list_length(list)].

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(n)

#### 4.3.2.13 list_is_empty()

```
int list_is_empty (
            const struct  list * list )  [inline]
```

Tests if a list is empty.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

1 if the list is empty, otherwise 0.

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.3.2.14 list_last()

```
struct  list_node* list_last (
            const struct  list * list )
```

Returns the last node.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

the last node.

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.3.2.15 list_length()

```
size_t list_length (
            const struct  list * list )  [inline]
```

Returns the size of the list.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

the length of the list.

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.3.2.16 list_next()

```
struct  list_node* list_next (
            const struct  list_node * node )
```

Returns the next node.

**Parameters**

| | |
|---|---|
| *node* | a node. |

**Returns**

the next node.

**Precondition**

`node` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.3.2.17 list_print()

```
void list_print (
            const struct  list * list )
```

Print the list

**Parameters**

| | |
|---|---|
| *list* | a list |

#### 4.3.2.18 list_reverse()

```
void list_reverse (
            struct  list * list )  [inline]
```

Reverses the order of the elements in the list.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.3.2.19 list_sort()

```
void list_sort (
            struct  list * list,
            int(*)(struct  list_node *, struct  list_node *) cmp )  [inline]
```

Sort a list using a comparison function.

The contents of the list are sorted in ascending order according to a comparison function which is called with two arguments that point to the node being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members compare as equal, their order in the sorted list is preserved.

**Parameters**

| | |
|---|---|
| *list* | list to sort. |
| *cmp* | comparison function to use. |

**Precondition**

`list` must be not NULL.
`cmp` must be not NULL.

**Remarks**

The sort is stable.
Complexity: O(N log N)
Space complexity: O(1)

#### 4.3.2.20 list_split_at()

```
void list_split_at (
            struct  list * list,
            size_t pos,
            struct  list * right )  [inline]
```

Splits a list in two parts at the position `pos`.

After the split:

- `list` contains nodes in [0, pos[
- `right` contains nodes in [pos,length(list)[

Examples:

```
list = [1, 2, 3]
list_split_at(list, 0, right) => ([],[1,2,3])
list_split_at(list, 1, right) => ([1],[2,3])
list_split_at(list, 2, right) => ([1,2],[3])
list_split_at(list, 3, right) => ([1,2,3],[])
list = []
list_split_at(list, 0, right) => ([],[])
```

**Parameters**

| | |
|---|---|
| *list* | list to split. |
| *pos* | position (0-based) where to split the list. |
| *right* | an empty list to receive the part after `pos` |

**Precondition**

`list` must be not NULL.
`right` must be not NULL.
`right` must be empty.
`list` must be different of `right`.

**Remarks**

Complexity: O(N)

#### 4.3.2.21 list_swap()

```
void list_swap (
            struct  list * l1,
            struct  list * l2 )  [inline]
```

Swaps two lists.

**Parameters**

| | |
|---|---|
| *l1* | list 1. |
| *l2* | list 2. |

**Precondition**

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

**Remarks**

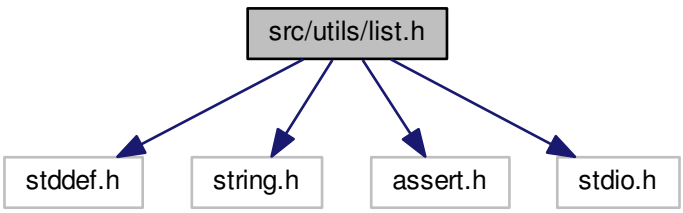
Complexity: O(1)

## 4.4 src/utils/list.h File Reference

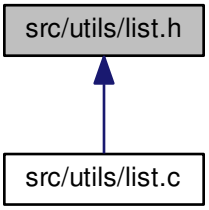
Intrusive list implement.

```
#include <stddef.h>
#include <string.h>
#include <assert.h>
#include <stdio.h>
```

Include dependency graph for list.h:



This graph shows which files directly or indirectly include this file:




## Data Structures

- struct **list**
- struct **list_node**

## Macros

- #define **list_elt**(node, type, fieldname) ((type∗)((char∗)(node) - offsetof(type, fieldname)))
- #define **list_foreach**( **list**, curr) for (curr = **list_first**( **list**); curr != NULL; curr = **list_next**(curr))
- #define **list_foreach_elt**( **list**, curr, type, fieldname)
- #define **list_foreach_safe**( **list**, curr, tmp)
- #define **list_foreach_elt_safe**( **list**, curr, tmp, type, fieldname)

## Functions

- void **list_init** (struct **list** ∗ **list**)
- size_t **list_length** (const struct **list** ∗ **list**)
- struct **list_node** ∗ **list_first** (const struct **list** ∗ **list**)
- struct **list_node** ∗ **list_last** (const struct **list** ∗ **list**)
- struct **list_node** ∗ **list_next** (const struct **list_node** ∗node)
- struct **list_node** ∗ **list_advance** (struct **list_node** ∗node, size_t distance)
- struct **list_node** ∗ **list_at** (const struct **list** ∗ **list**, size_t pos)
- void **list_reverse** (struct **list** ∗ **list**)
- void **list_swap** (struct **list** ∗l1, struct **list** ∗l2)
- void **list_split_at** (struct **list** ∗ **list**, size_t pos, struct **list** ∗right)
- void **list_concat** (struct **list** ∗l1, struct **list** ∗l2)
- void **list_sort** (struct **list** ∗ **list**, int(∗cmp)(struct **list_node** ∗, struct **list_node** ∗))
- int **list_is_empty** (const struct **list** ∗ **list**)
- void **list_add** (struct **list** ∗ **list**, struct **list_node** ∗node)
- void **list_append** (struct **list** ∗ **list**, struct **list_node** ∗node)
- void **list_insert_after** (struct **list** ∗ **list**, struct **list_node** ∗curr, struct **list_node** ∗node)
- void **list_insert_at** (struct **list** ∗ **list**, struct **list_node** ∗node, size_t pos)
- void **list_del** (struct **list** ∗ **list**)
- void **list_del_after** (struct **list** ∗ **list**, struct **list_node** ∗node)
- void **list_del_at** (struct **list** ∗ **list**, size_t pos)
- void **list_print** (const struct **list** ∗ **list**)

### 4.4.1 Detailed Description

Intrusive list implement.

**Author**

S4MasterRace

**Version**

1.0

### 4.4.2 Macro Definition Documentation

#### 4.4.2.1 list_elt

```
#define list_elt(
              node,
              type,
              fieldname ) ((type*)((char*)(node) - offsetof(type, fieldname)))
```

Returns a pointer to the structure which contains the node.

**Parameters**

| | |
|---|---|
| *node* | a list node (struct list_node*). |
| *type* | type of the structure which contains the node. |
| *fieldname* | name of the node (field name) in the structure. |

**Precondition**

`node` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.4.2.2 list_foreach

```
#define list_foreach(
               list,
              curr ) for (curr =  list_first( list); curr != NULL; curr =  list_next(curr))
```

Iterates over list (nodes).

**Parameters**

| | |
|---|---|
| *list* | a list (struct list∗). |
| *curr* | a struct list_node∗ used to hold the current element. |

**Precondition**

`list` must be not NULL.
`curr` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.4.2.3 list_foreach_elt

```
#define list_foreach_elt(
                list,
               curr,
               type,
               fieldname )
```

**Value:**

```
for (curr = list_elt(list_first(list), type, fieldname);       \
         curr != NULL;                                          \
         curr = curr->fieldname.next == NULL ? NULL :           \
         list_elt(list_next(&(curr->fieldname)), type, fieldname))
```

Iterates over list (elements)

**Parameters**

| | |
|---|---|
| *list* | a list (struct list∗). |
| *curr* | pointer (type∗) used to hold the current element. |
| *type* | type of the structure which contains the node. |
| *fieldname* | name of the node (field name) in the structure. |

**Precondition**

`list` must be not NULL.
`list` must be not empty.
`curr` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.4.2.4 list_foreach_elt_safe

```
#define list_foreach_elt_safe(
                list,
               curr,
               tmp,
               type,
               fieldname )
```

**Value:**

```
for (curr = list_elt(list_first(list), type, fieldname),      \
         tmp  = list_next(&(curr->fieldname));              \
         curr != NULL;                                      \
         curr = tmp == NULL ? NULL : list_elt(tmp, type, fieldname), \
         tmp  = tmp == NULL ? NULL : list_next(tmp))          \
```

Iterates over list (elements), allows deletion of the current element.

**Parameters**

| | |
|---|---|
| *list* | a list (struct list∗). |
| *curr* | pointer (type∗) used to hold the current element. |
| *tmp* | a struct list_node∗ used as temporary storage. |
| *type* | type of the structure which contains the node. |
| *fieldname* | name of the node (field name) in the structure. |

**Precondition**

`list` must be not NULL.
`list` must be not empty.
`curr` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.4.2.5 list_foreach_safe

```
#define list_foreach_safe(
                list,
               curr,
               tmp )
```

**Value:**

```
for (curr = list_first(list), tmp = list_next(curr);    \
         curr != NULL;                                  \
         curr = tmp, tmp = tmp == NULL ? NULL : list_next(tmp))
```

Iterates over list (nodes), allows deletion of the current node.

**Parameters**

| | |
|---|---|
| *list* | a list (struct list∗). |
| *curr* | a struct list_node∗ used to hold the current element. |
| *tmp* | a struct list_node∗ used as temporary storage. |

**Precondition**

`list` must be not NULL.
`curr` must be not NULL.
`tmp` must be not NULL.

**Remarks**

Complexity: O(N)

### 4.4.3 Function Documentation

#### 4.4.3.1 list_add()

```
void list_add (
            struct  list * list,
            struct  list_node * node )  [inline]
```

Adds `node` in the front of `list`

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | the new node. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(1)

#### 4.4.3.2 list_advance()

```
struct  list_node* list_advance (
             struct  list_node * node,
             size_t distance )
```

Returns the nth-node after the current one.

**Parameters**

| | |
|---|---|
| *node* | a node. |
| *distance* | distance to move on. |

**Returns**

the nth-node after `node`.

**Precondition**

`node` must be not NULL.

**Remarks**

Complexity: O(n)

#### 4.4.3.3 list_append()

```
void list_append (
             struct  list * list,
             struct  list_node * node )  [inline]
```

Adds `node` at the end of `list`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | the new node. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(n)

#### 4.4.3.4 list_at()

```
struct  list_node* list_at (
            const struct  list * list,
            size_t pos )
```

Returns node at the position `pos`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *pos* | position (0-based) of the node. |

**Returns**

the node at the position `pos`.

**Precondition**

`list` must be not NULL.
`list` must be not empty.
`pos` must be in [0; list_length(list)[.

**Remarks**

Complexity: O(N)

#### 4.4.3.5 list_concat()

```
void list_concat (
            struct  list * l1,
            struct  list * l2 )  [inline]
```

Concatenates two lists.

**Parameters**

| | |
|---|---|
| *l1* | list 1. |
| *l2* | list 2. |

**Precondition**

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

**Postcondition**

`l2` is reset to an empty list.

**Remarks**

Complexity: O(N)

#### 4.4.3.6 list_del()

```
void list_del (
            struct  list * list )  [inline]
```

Deletes the first node.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Precondition**

`list` must be not NULL.
`list` must be not empty.

**Postcondition**

List size decreases by 1.

**Remarks**

Complexity: O(1)

#### 4.4.3.7 list_del_after()

```
void list_del_after (
            struct  list * list,
            struct  list_node * node )  [inline]
```

Deletes the node at after the node `curr`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | a node of `list`. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.
`list` must be not empty.
`node` must a node of `list`.

**Postcondition**

List size decreases by 1.

**Remarks**

Complexity: O(1)

#### 4.4.3.8 list_del_at()

```
void list_del_at (
            struct  list * list,
            size_t pos )  [inline]
```

Deletes the node at the position `pos`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *pos* | index (0-based) of the node to delete. |

**Precondition**

`list` must be not NULL.
`list` must be not empty.
`pos` must be in [0; list_length(list)[.

**Postcondition**

List size decreases by 1.

**Remarks**

Complexity: O(n)

#### 4.4.3.9 list_first()

```
struct  list_node* list_first (
            const struct  list * list )
```

Returns the first node.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

the first node.

**Precondition**

`list` must be not NULL.
`list` must be not empty.

**Remarks**

Complexity: O(1)

#### 4.4.3.10 list_init()

```
void list_init (
            struct  list * list )  [inline]
```

Initializes the list.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Precondition**

`list` must be not NULL.

**Postcondition**

`list` is empty.
`list` has a size of 0.

**Remarks**

Complexity: O(1)

#### 4.4.3.11 list_insert_after()

```
void list_insert_after (
            struct  list * list,
            struct  list_node * curr,
            struct  list_node * node )  [inline]
```

Inserts `node` at after the node `curr`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *curr* | a node of `list`. |
| *node* | new node. |

**Precondition**

`list` must be not NULL.
`curr` must be not NULL.
`curr` must a node of `list`.
`node` must be not NULL.

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(1)

#### 4.4.3.12 list_insert_at()

```
void list_insert_at (
            struct  list * list,
            struct  list_node * node,
            size_t pos )  [inline]
```

Inserts `node` at the position `pos` in `list`.

**Parameters**

| | |
|---|---|
| *list* | a list. |
| *node* | new node. |
| *pos* | position (0-based) where to insert the new node. |

**Precondition**

`list` must be not NULL.
`node` must be not NULL.
`pos` must be in [0; list_length(list)].

**Postcondition**

List size increases by 1.

**Remarks**

Complexity: O(n)

#### 4.4.3.13 list_is_empty()

```
int list_is_empty (
            const struct  list * list )  [inline]
```

Tests if a list is empty.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

1 if the list is empty, otherwise 0.

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.4.3.14 list_last()

```
struct  list_node* list_last (
            const struct  list * list )
```

Returns the last node.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

the last node.

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.4.3.15 list_length()

```
size_t list_length (
            const struct  list * list )  [inline]
```

Returns the size of the list.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Returns**

the length of the list.

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.4.3.16 list_next()

```
struct  list_node* list_next (
            const struct  list_node * node )
```

Returns the next node.

**Parameters**

| | |
|---|---|
| *node* | a node. |

**Returns**

the next node.

**Precondition**

`node` must be not NULL.

**Remarks**

Complexity: O(1)

#### 4.4.3.17 list_print()

```
void list_print (
            const struct  list * list )
```

Print the list

**Parameters**

| | |
|---|---|
| *list* | a list |

#### 4.4.3.18 list_reverse()

```
void list_reverse (
            struct  list * list )  [inline]
```

Reverses the order of the elements in the list.

**Parameters**

| | |
|---|---|
| *list* | a list. |

**Precondition**

`list` must be not NULL.

**Remarks**

Complexity: O(N)

#### 4.4.3.19 list_sort()

```
void list_sort (
            struct  list * list,
            int(*)(struct  list_node *, struct  list_node *) cmp )  [inline]
```

Sort a list using a comparison function.

The contents of the list are sorted in ascending order according to a comparison function which is called with two arguments that point to the node being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members compare as equal, their order in the sorted list is preserved.

**Parameters**

| | |
|---|---|
| *list* | list to sort. |
| *cmp* | comparison function to use. |

**Precondition**

`list` must be not NULL.
`cmp` must be not NULL.

**Remarks**

The sort is stable.
Complexity: O(N log N)
Space complexity: O(1)

#### 4.4.3.20 list_split_at()

```
void list_split_at (
            struct  list * list,
            size_t pos,
            struct  list * right )  [inline]
```

Splits a list in two parts at the position `pos`.

After the split:

- `list` contains nodes in [0, pos[
- `right` contains nodes in [pos,length(list)[

Examples:

```
list = [1, 2, 3]
list_split_at(list, 0, right) => ([],[1,2,3])
list_split_at(list, 1, right) => ([1],[2,3])
list_split_at(list, 2, right) => ([1,2],[3])
list_split_at(list, 3, right) => ([1,2,3],[])
list = []
list_split_at(list, 0, right) => ([],[])
```

**Parameters**

| | |
|---|---|
| *list* | list to split. |
| *pos* | position (0-based) where to split the list. |
| *right* | an empty list to receive the part after `pos` |

**Precondition**

`list` must be not NULL.
`right` must be not NULL.
`right` must be empty.
`list` must be different of `right`.

**Remarks**

Complexity: O(N)

#### 4.4.3.21 list_swap()

```
void list_swap (
            struct  list * l1,
            struct  list * l2 )  [inline]
```

Swaps two lists.

**Parameters**

| | |
|---|---|
| *l1* | list 1. |
| *l2* | list 2. |

**Precondition**

`l1` must be not NULL.
`l2` must be not NULL.
`l1` must be different of `l2`.

**Remarks**

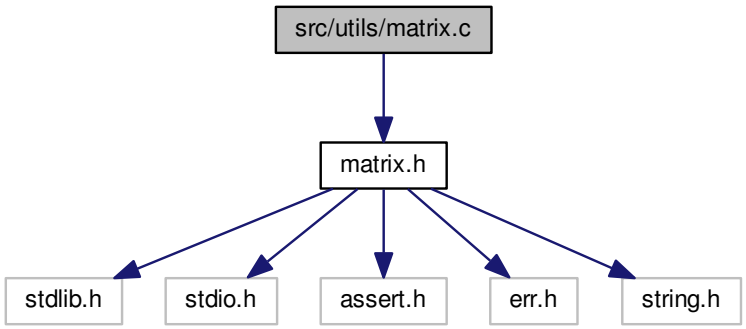
Complexity: O(1)

## 4.5 src/utils/matrix.c File Reference

Matrix implement.

```
#include "matrix.h"
```

Include dependency graph for matrix.c:




### Functions

- struct **matrix** * **matrix_create** (size_t rows, size_t cols)
- struct **matrix** * **matrix_create_from_array** (size_t rows, size_t cols, const double values[ ])
- void **matrix_free** (struct **matrix** *mx)
- size_t **matrix_rows** (const struct **matrix** *mx)
- size_t **matrix_cols** (const struct **matrix** *mx)
- double **matrix_at** (const struct **matrix** *mx, size_t rows, size_t cols)
- void **matrix_set** (struct **matrix** *mx, size_t rows, size_t cols, double value)
- struct **matrix** * **matrix_copy** (const struct **matrix** *mx)
- void **matrix_transpose** (const struct **matrix** *mx, struct **matrix** *tmx)
- void **matrix_sum** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *sum)
- void **matrix_hadamard_product** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *prod)
- void **matrix_product** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *prod)
- void **matrix_scale** (const struct **matrix** *mx, double scale, struct **matrix** *smx)
- double **matrix_dot_product** (const struct **matrix** *v1, const struct **matrix** *v2)
- void **matrix_identity** (struct **matrix** *mx)
- int **matrix_is_square** (const struct **matrix** *mx)
- int **matrix_is_diagonal** (const struct **matrix** *mx)
- int **matrix_is_upper_triangulared** (const struct **matrix** *mx)
- void **matrix_diagonal** (const struct **matrix** *v, struct **matrix** *mx)
- void **matrix_print** (const struct **matrix** *mx)

### 4.5.1 Detailed Description

Matrix implement.

**Author**

S4MasterRace

**Version**

1.0

### 4.5.2 Function Documentation

#### 4.5.2.1 matrix_at()

```
double matrix_at (
            const struct  matrix * mx,
            size_t rows,
            size_t cols )  [inline]
```

Get value at `rows` rows and `cols` columns of `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *rows* | rows |
| *cols* | columns |

**Returns**

the value at `rows` rows and `cols` columns of `mx`

**Precondition**

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx)[`
`cols` must be between `[0, matrix_cols(mx)[`

**Remarks**

Complexity: O(1)

#### 4.5.2.2 matrix_cols()

```
size_t matrix_cols (
            const struct  matrix * mx )  [inline]
```

Get the number of columns of `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

the number of columns `mx`

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.5.2.3 matrix_copy()

```
struct  matrix* matrix_copy (
            const struct  matrix * mx )
```

Copy the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

the copy of `mx`

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.5.2.4 matrix_create()

```
struct  matrix* matrix_create (
            size_t rows,
            size_t cols )
```

Create a matrix of size `rows` rows and `cols` columns

**Parameters**

| | |
|---|---|
| *rows* | number of rows |
| *cols* | number of columns |

**Returns**

the initialized matrix of size `rows` rows and `cols` columns

**Precondition**

`rows` must be greater than zero
`cols` must be greater than zero

**Remarks**

Complexity: O(1)

#### 4.5.2.5 matrix_create_from_array()

```
struct  matrix* matrix_create_from_array (
              size_t rows,
              size_t cols,
              const double values[] )
```

Create a matrix of size `rows` rows and `cols` columns from `values`

**Parameters**

| | |
|---|---|
| *rows* | number of rows |
| *cols* | number of columns |
| *values* | an array |

**Returns**

the initialized matrix of size `rows` rows and `cols` columns from `values`

**Precondition**

`rows` must be greater than zero
`cols` must be greater than zero
`values` must be not NULL

**Remarks**

Complexity: O(N)

#### 4.5.2.6 matrix_diagonal()

```
void matrix_diagonal (
            const struct  matrix * v,
            struct  matrix * mx )
```

Take a vector `v` and put it to the diagonal of `mx`

**Parameters**

| | |
|---|---|
| *v* | a vector |
| *mx* | a matrix |

**Precondition**

`v` must be not NULL
`mx` must be not NULL
`matrix_cols(v)` must be equal to one
`matrix_rows(v)` must be equel to `matrix_rows(mx)`
`matrix_rows(mx)` must be equel to `matrix_cols(mx)`

**Postcondition**

The diagonal of `mx` is `v`

#### 4.5.2.7 matrix_dot_product()

```
double matrix_dot_product (
            const struct  matrix * v1,
            const struct  matrix * v2 )
```

Do the dot product of vector `v1` with vector `v2`

**Parameters**

| | |
|---|---|
| *v1* | a vector |
| *v2* | a vector |

**Returns**

the dot product of vector `v1` with vector `v2`

**Precondition**

`v1` must be not NULL
`v2` must be not NULL
`matrix_cols(v1)` and `matrix_cols(v2)` must be equal to one
`matrix_rows(v1)` must be equal to `matrix_rows(v2)`

**Remarks**

Complexity: O(N)

#### 4.5.2.8 matrix_free()

```
void matrix_free (
            struct  matrix * mx )  [inline]
```

Free the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Precondition**

`mx` must be not NULL

**Postcondition**

`mx` is freed

**Remarks**

Complexity: O(1)

#### 4.5.2.9 matrix_hadamard_product()

```
void matrix_hadamard_product (
            const struct  matrix * mx1,
            const struct  matrix * mx2,
            struct  matrix * prod )
```

Do the hadamard product of matrix `mx1` with `mx2`

**Parameters**

| | |
|---|---|
| *mx1* | a matrix |
| *mx2* | a matrix |
| *prod* | a matrix |

**Precondition**

`mx1` must be not NULL
`mx2` must be not NULL
`prod` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`
`matrix_rows(prod)` must be equal to `matrix_rows(mx1)`
`matrix_cols(prod)` must be equal to `matrix_cols(mx1)`

**Postcondition**

`prod` is the hadamard product of matrix `mx1` with `mx2`

**Remarks**

Complexity: O(N)

#### 4.5.2.10 matrix_identity()

```
void matrix_identity (
            struct  matrix * mx )
```

Set the matrix `mx` to an identity matrix

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Precondition**

`mx` must be not NULL
`matrix_rows(mx)` must be equal to `matrix_cols(mx)`

**Postcondition**

`mx` is an identity matrix

**Remarks**

Complexity: O(N)

#### 4.5.2.11 matrix_is_diagonal()

```
int matrix_is_diagonal (
            const struct  matrix * mx )  [inline]
```

Check if the matrix `mx` is diagonaled

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

`1` if the matrix is diagonaled, `0` otherwise

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(N)

#### 4.5.2.12 matrix_is_square()

```
int matrix_is_square (
            const struct  matrix * mx )  [inline]
```

Check if the matrix `mx` is squared

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

`1` if the matrix is squared, `0` otherwise

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.5.2.13 matrix_is_upper_triangulared()

```
int matrix_is_upper_triangulared (
            const struct  matrix * mx )  [inline]
```

Check if the matrix `mx` is upper triangulared

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

`1` if the matrix is upper triangulared, `0` otherwise

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(N)

#### 4.5.2.14 matrix_print()

```
void matrix_print (
            const struct  matrix * mx )
```

Print the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Precondition**

`mx` must be not NULL

**Postcondition**

Print the matrix `mx`

#### 4.5.2.15 matrix_product()

```
void matrix_product (
            const struct  matrix * mx1,
            const struct  matrix * mx2,
            struct  matrix * prod )
```

Multiply the matrix `mx1` with `mx2`

**Parameters**

| | |
|---|---|
| *mx1* | a matrix |
| *mx2* | a matrix |
| *prod* | a matrix |

**Precondition**

`mx1` must be not NULL
`mx2` must be not NULL
`prod` must be not NULL
`prod` must be not equal to `mx1`
`prod` must be not equal to `mx2`
`matrix_cols(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_rows(prod)` must be equal to `matrix_rows(mx1)`
`matrix_cols(prod)` must be equal to `matrix_cols(mx2)`

**Postcondition**

`prod` is the product of `mx1` with `mx2`

**Remarks**

Complexity: O(nmp)

#### 4.5.2.16 matrix_rows()

```
size_t matrix_rows (
            const struct  matrix * mx )  [inline]
```

Get the number of rows `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

the number of rows of `mx`

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.5.2.17 matrix_scale()

```
void matrix_scale (
            const struct  matrix * mx,
            double scale,
            struct  matrix * smx )
```

Scale the matrix `mx` with `scale`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *scale* | the scale factor |
| *smx* | a matrix |

**Precondition**

`mx` must be not NULL
`smx` must be not NULL
`matrix_rows(smx)` must be equal to `matrix_rows(mx)`
`matrix_cols(smx)` must be equal to `matrix_cols(mx)`

**Postcondition**

`smx` is the `scale` scaled matrix of `mx`

**Remarks**

Complexity: O(N)

#### 4.5.2.18 matrix_set()

```
void matrix_set (
            struct  matrix * mx,
            size_t rows,
            size_t cols,
            double value )  [inline]
```

Set the value at `rows` rows and `cols` columns with `value` of `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *rows* | rows |
| *cols* | columns |
| *value* | a value |

**Precondition**

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx)[`
`cols` must be between `[0, matrix_cols(mx)[`

**Postcondition**

the value at `rows` rows and `cols` columns is `value`

**Remarks**

Complexity: O(1)

#### 4.5.2.19 matrix_sum()

```
void matrix_sum (
            const struct  matrix * mx1,
            const struct  matrix * mx2,
            struct  matrix * sum )
```

Sum the matrix `mx1` with `mx2`

**Parameters**

| | |
|---|---|
| *mx1* | a matrix |
| *mx2* | a matrix |
| *sum* | a matrix |

**Precondition**

`mx1` must be not NULL
`mx2` must be not NULL
`sum` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`
`matrix_rows(sum)` must be equal to `matrix_rows(mx1)`
`matrix_cols(sum)` must be equal to `matrix_cols(mx1)`

**Postcondition**

`sum` is the sum of matrix `mx1` with `mx2`

**Remarks**

Complexity: O(N)

#### 4.5.2.20 matrix_transpose()

```
void matrix_transpose (
            const struct  matrix * mx,
            struct  matrix * tmx )
```

Transpose the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *tmx* | a matrix |

**Precondition**

`mx` must be not NULL
`tmx` must be not NULL
`tmx` must be not equal to `mx`
`matrix_rows(tmx)` must be equal to `matrix_cols(mx)`
`matrix_cols(tmx)` must be equal to `matrix_rows(mx)`

**Postcondition**

`tmx` is the transposed matrix of `mx`

**Remarks**

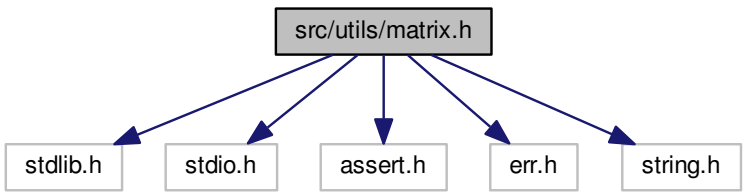
Complexity: O(N)

## 4.6 src/utils/matrix.h File Reference

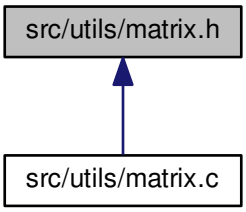
Matrix implement.

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <err.h>
#include <string.h>
```

Include dependency graph for matrix.h:



This graph shows which files directly or indirectly include this file:




## Data Structures

- struct **matrix**

## Functions

- struct **matrix** * **matrix_create** (size_t rows, size_t cols)
- struct **matrix** * **matrix_create_from_array** (size_t rows, size_t cols, const double values[ ])
- void **matrix_free** (struct **matrix** *mx)
- size_t **matrix_rows** (const struct **matrix** *mx)
- size_t **matrix_cols** (const struct **matrix** *mx)
- double **matrix_at** (const struct **matrix** *mx, size_t rows, size_t cols)
- void **matrix_set** (struct **matrix** *mx, size_t rows, size_t cols, double value)
- struct **matrix** * **matrix_copy** (const struct **matrix** *mx)
- void **matrix_transpose** (const struct **matrix** *mx, struct **matrix** *tmx)
- void **matrix_sum** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *sum)
- void **matrix_hadamard_product** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *prod)
- void **matrix_product** (const struct **matrix** *mx1, const struct **matrix** *mx2, struct **matrix** *prod)
- void **matrix_scale** (const struct **matrix** *mx, double scale, struct **matrix** *smx)
- double **matrix_dot_product** (const struct **matrix** *v1, const struct **matrix** *v2)
- void **matrix_identity** (struct **matrix** *mx)
- int **matrix_is_square** (const struct **matrix** *mx)
- int **matrix_is_diagonal** (const struct **matrix** *mx)
- int **matrix_is_upper_triangulared** (const struct **matrix** *mx)
- void **matrix_diagonal** (const struct **matrix** *v, struct **matrix** *mx)
- void **matrix_print** (const struct **matrix** *mx)

### 4.6.1 Detailed Description

Matrix implement.

**Author**

S4MasterRace

**Version**

1.0

### 4.6.2 Function Documentation

#### 4.6.2.1 matrix_at()

```
double matrix_at (
            const struct  matrix * mx,
            size_t rows,
            size_t cols )  [inline]
```

Get value at `rows` rows and `cols` columns of `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *rows* | rows |
| *cols* | columns |

**Returns**

the value at `rows` rows and `cols` columns of `mx`

**Precondition**

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx)[`
`cols` must be between `[0, matrix_cols(mx)[`

**Remarks**

Complexity: O(1)

#### 4.6.2.2 matrix_cols()

```
size_t matrix_cols (
            const struct  matrix * mx )  [inline]
```

Get the number of columns of `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

the number of columns `mx`

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.6.2.3 matrix_copy()

```
struct  matrix* matrix_copy (
            const struct  matrix * mx )
```

Copy the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

the copy of `mx`

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.6.2.4 matrix_create()

```
struct  matrix* matrix_create (
            size_t rows,
            size_t cols )
```

Create a matrix of size `rows` rows and `cols` columns

**Parameters**

| | |
|---|---|
| *rows* | number of rows |
| *cols* | number of columns |

**Returns**

the initialized matrix of size `rows` rows and `cols` columns

**Precondition**

`rows` must be greater than zero
`cols` must be greater than zero

**Remarks**

Complexity: O(1)

#### 4.6.2.5 matrix_create_from_array()

```
struct  matrix* matrix_create_from_array (
             size_t rows,
             size_t cols,
             const double values[] )
```

Create a matrix of size `rows` rows and `cols` columns from `values`

**Parameters**

| | |
|---|---|
| *rows* | number of rows |
| *cols* | number of columns |
| *values* | an array |

**Returns**

the initialized matrix of size `rows` rows and `cols` columns from `values`

**Precondition**

`rows` must be greater than zero
`cols` must be greater than zero
`values` must be not NULL

**Remarks**

Complexity: O(N)

#### 4.6.2.6 matrix_diagonal()

```
void matrix_diagonal (
             const struct  matrix * v,
             struct  matrix * mx )
```

Take a vector `v` and put it to the diagonal of `mx`

**Parameters**

| | |
|---|---|
| *v* | a vector |
| *mx* | a matrix |

**Precondition**

`v` must be not NULL
`mx` must be not NULL
`matrix_cols(v)` must be equal to one
`matrix_rows(v)` must be equel to `matrix_rows(mx)`
`matrix_rows(mx)` must be equel to `matrix_cols(mx)`

**Postcondition**

The diagonal of `mx` is `v`

#### 4.6.2.7 matrix_dot_product()

```
double matrix_dot_product (
            const struct  matrix * v1,
            const struct  matrix * v2 )
```

Do the dot product of vector `v1` with vector `v2`

**Parameters**

| | |
|---|---|
| *v1* | a vector |
| *v2* | a vector |

**Returns**

the dot product of vector `v1` with vector `v2`

**Precondition**

`v1` must be not NULL
`v2` must be not NULL
`matrix_cols(v1)` and `matrix_cols(v2)` must be equal to one
`matrix_rows(v1)` must be equal to `matrix_rows(v2)`

**Remarks**

Complexity: O(N)

#### 4.6.2.8 matrix_free()

```
void matrix_free (
            struct  matrix * mx )  [inline]
```

Free the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Precondition**

`mx` must be not NULL

**Postcondition**

`mx` is freed

**Remarks**

Complexity: O(1)

#### 4.6.2.9 matrix_hadamard_product()

```
void matrix_hadamard_product (
            const struct  matrix * mx1,
            const struct  matrix * mx2,
            struct  matrix * prod )
```

Do the hadamard product of matrix `mx1` with `mx2`

**Parameters**

| | |
|---|---|
| *mx1* | a matrix |
| *mx2* | a matrix |
| *prod* | a matrix |

**Precondition**

`mx1` must be not NULL
`mx2` must be not NULL
`prod` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`
`matrix_rows(prod)` must be equal to `matrix_rows(mx1)`
`matrix_cols(prod)` must be equal to `matrix_cols(mx1)`

**Postcondition**

`prod` is the hadamard product of matrix `mx1` with `mx2`

**Remarks**

Complexity: O(N)

#### 4.6.2.10 matrix_identity()

```
void matrix_identity (
            struct  matrix * mx )
```

Set the matrix `mx` to an identity matrix

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Precondition**

`mx` must be not NULL
`matrix_rows(mx)` must be equal to `matrix_cols(mx)`

**Postcondition**

`mx` is an identity matrix

**Remarks**

Complexity: O(N)

#### 4.6.2.11 matrix_is_diagonal()

```
int matrix_is_diagonal (
            const struct  matrix * mx )  [inline]
```

Check if the matrix `mx` is diagonaled

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

`1` if the matrix is diagonaled, `0` otherwise

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(N)

#### 4.6.2.12 matrix_is_square()

```
int matrix_is_square (
            const struct  matrix * mx )  [inline]
```

Check if the matrix `mx` is squared

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

`1` if the matrix is squared, `0` otherwise

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.6.2.13 matrix_is_upper_triangulared()

```
int matrix_is_upper_triangulared (
            const struct  matrix * mx )  [inline]
```

Check if the matrix `mx` is upper triangulared

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

`1` if the matrix is upper triangulared, `0` otherwise

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(N)

#### 4.6.2.14 matrix_print()

```
void matrix_print (
            const struct  matrix * mx )
```

Print the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Precondition**

`mx` must be not NULL

**Postcondition**

Print the matrix `mx`

#### 4.6.2.15 matrix_product()

```
void matrix_product (
            const struct  matrix * mx1,
            const struct  matrix * mx2,
            struct  matrix * prod )
```

Multiply the matrix `mx1` with `mx2`

**Parameters**

| | |
|---|---|
| *mx1* | a matrix |
| *mx2* | a matrix |
| *prod* | a matrix |

**Precondition**

`mx1` must be not NULL
`mx2` must be not NULL
`prod` must be not NULL
`prod` must be not equal to `mx1`
`prod` must be not equal to `mx2`
`matrix_cols(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_rows(prod)` must be equal to `matrix_rows(mx1)`
`matrix_cols(prod)` must be equal to `matrix_cols(mx2)`

**Postcondition**

`prod` is the product of `mx1` with `mx2`

**Remarks**

Complexity: O(nmp)

#### 4.6.2.16 matrix_rows()

```
size_t matrix_rows (
            const struct  matrix * mx )  [inline]
```

Get the number of rows `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |

**Returns**

the number of rows of `mx`

**Precondition**

`mx` must be not NULL

**Remarks**

Complexity: O(1)

#### 4.6.2.17 matrix_scale()

```
void matrix_scale (
            const struct  matrix * mx,
            double scale,
            struct  matrix * smx )
```

Scale the matrix `mx` with `scale`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *scale* | the scale factor |
| *smx* | a matrix |

**Precondition**

`mx` must be not NULL
`smx` must be not NULL
`matrix_rows(smx)` must be equal to `matrix_rows(mx)`
`matrix_cols(smx)` must be equal to `matrix_cols(mx)`

**Postcondition**

`smx` is the `scale` scaled matrix of `mx`

**Remarks**

Complexity: O(N)

#### 4.6.2.18 matrix_set()

```
void matrix_set (
            struct  matrix * mx,
            size_t rows,
            size_t cols,
            double value )  [inline]
```

Set the value at `rows` rows and `cols` columns with `value` of `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *rows* | rows |
| *cols* | columns |
| *value* | a value |

**Precondition**

`mx` must be not NULL
`rows` must be between `[0, matrix_rows(mx)[`
`cols` must be between `[0, matrix_cols(mx)[`

**Postcondition**

the value at `rows` rows and `cols` columns is `value`

**Remarks**

Complexity: O(1)

#### 4.6.2.19 matrix_sum()

```
void matrix_sum (
            const struct  matrix * mx1,
            const struct  matrix * mx2,
            struct  matrix * sum )
```

Sum the matrix `mx1` with `mx2`

**Parameters**

| | |
|---|---|
| *mx1* | a matrix |
| *mx2* | a matrix |
| *sum* | a matrix |

**Precondition**

`mx1` must be not NULL
`mx2` must be not NULL
`sum` must be not NULL
`matrix_rows(mx1)` must be equal to `matrix_rows(mx2)`
`matrix_cols(mx1)` must be equal to `matrix_cols(mx2)`
`matrix_rows(sum)` must be equal to `matrix_rows(mx1)`
`matrix_cols(sum)` must be equal to `matrix_cols(mx1)`

**Postcondition**

`sum` is the sum of matrix `mx1` with `mx2`

**Remarks**

Complexity: O(N)

#### 4.6.2.20 matrix_transpose()

```
void matrix_transpose (
            const struct  matrix * mx,
            struct  matrix * tmx )
```

Transpose the matrix `mx`

**Parameters**

| | |
|---|---|
| *mx* | a matrix |
| *tmx* | a matrix |

**Precondition**

`mx` must be not NULL
`tmx` must be not NULL
`tmx` must be not equal to `mx`
`matrix_rows(tmx)` must be equal to `matrix_cols(mx)`
`matrix_cols(tmx)` must be equal to `matrix_rows(mx)`

**Postcondition**

`tmx` is the transposed matrix of `mx`

**Remarks**

Complexity: O(N)

# Index