Writing your own linear algebra matrix library in C

Andrei Ciobanu

January 20, 2021

1 The library

Now, let's say you are one of the few Engineering/Computer Science students that are passionate about *linear algebra*, *numerical analysis* and writing code in a low-level language. Or you are just curious about what's behind the `lu(A)` method in Matlab. Or you are passionate about A.I., and you know you cannot learn A.I. algorithms without a good foundation in linear algebra.

I believe the best exercise you can do is to try to write your (own) Matrix library in a low-level programming language (like C, C++ or even D).

This tutorial is precisely this, a step-by-step explanation of writing a C Matrix library that implements the "basic" and "not-so-basic" numerical analysis algorithms that will allow us in the end to solve linear systems of equations.

All code in this tutorial is available on GitHub in the repository called `neat-matrix-library`.

To clone it (using GitHub CLI):

```
~$ gh repo clone nomemory/neat-matrix-library
```

The code is not meant to be "efficient", but relatively easy to follow and understand.

The tutorial assumes that the reader can write C code, understand pointers and dynamic memory allocation, and is familiar with the C standard library.

2 The data: `nml_matrix`

The first step is to model the data our library will work with: "matrices". So we are going to define our first struct, called `nml_mat` which models a matrix:

```
typedef struct nml_mat_s {
  unsigned int num_rows;
  unsigned int num_cols;
  double **data;
  int is_square;
} nml_mat;
```

The properties of this `struct` have self-explanatory names:

- `unsigned int num_rows` – represents the number of rows of the matrix. 0 is not an acceptable value
- `unsigned int num_cols` – represents the number of columns of the matrix. 0 is not an acceptable value
- `double **data` – is "multi-dimensional array" that stores data in rows and columns
- `int is_square` – is a "boolean" value that determines if the matrix is square (has the same number of rows and columns) or not.

From a performance perspective, it's better to keep the matrix elements in a `double*` using the conversion:

```
data[i][j] <=> array[i * m + j]
```

To better understand how to store multi-dimensional arrays in linear storage please refer to *this StackOverflow question*, or *read the wikipedia article* on the topic.

Even if it might sound like a "controversial" decision, for the sake of simplicity, we will use the double ** multi-dimensional storage.

3 Allocating/deallocating memory for the `nml_mat` matrix

Unlike "higher-level" programming languages (Java, Python, etc), that manage memory allocation for you, in C, you need to explicitly ask for memory and explicitly free the memory once you no longer need it.

In this regard, the next step is to create "constructor-like" and "destructor-like" functions for the `nml_mat struct` defined above. There's an unwritten rule that says: "Every `malloc()` has its personal `free()`".

```
// Constructor-like
// Allocates memory for a new matrix
// All elements in the matrix are 0.0
nml_mat *nml_mat_new(unsigned int num_rows, unsigned int num_cols);
```

```
// Destructor-like
// De-allocates the memory for the matrix
void nml_mat_free(nml_mat *matrix);
```

Implementing the `nml_mat_new()` is quite straightforward:

```
nml_mat *nml_mat_new(unsigned int num_rows, unsigned int num_cols) {
  if (num_rows == 0) {
    NML_ERROR(INVALID_ROWS);
    return NULL;
  }
  if (num_cols == 0) {
    NML_ERROR(INVALID_COLS);
    return NULL;
  }
  nml_mat *m = calloc(1, sizeof(*m));
  NP_CHECK(m);
  m->num_rows = num_rows;
  m->num_cols = num_cols;
  m->is_square = (num_rows == num_cols) ? 1 : 0;
  m->data = calloc(m->num_rows, sizeof(*m->data));
  NP_CHECK(m->data);
  int i;
  for(i = 0; i < m->num_rows; ++i) {
    m->data[i] = calloc(m->num_cols, sizeof(**m->data));
    NP_CHECK(m->data[i]);
  }
  return m;
}
```

Notes:

- `NML_ERROR`, `NP_CHECK` are macros defined in `nml_util.h`.
- `NML_ERROR()` or `NML_FERROR()` are logging utils, that helps us print error message on stderr;
- `NP_CHECK` checks if the newly allocated memory chunk is not `NULL`. In case it's `NULL` it aborts the program.

Explanation:

1. First step is to check if `num_rows == 0` or `num_cols == 0`. If they are, we consider the input as invalid, and we print on stderr an error. Afterwards `NULL` is returned;
2. This line: `nml_mat *m = calloc(1, sizeof(*m))` allocates memory for 1 (one) `nml_mat structure`;

3. For a multi-dimensional array (`double**`), we allocate memory in two steps:
   - `m->data = calloc(m->num_rows, sizeof(*m->data))` - this allocates memory for the column array;
   - Then, we allocate memory for each row. By using `calloc()` the data is initialized with 0.0.

Freeing the matrix is even more straightforward. The implementation for `nml_mat_free()`:

```
void nml_mat_free(nml_mat *matrix) {
  int i;
  for(i = 0; i < matrix->num_rows; ++i) {
    free(matrix->data[i]);
  }
  free(matrix->data);
  free(matrix);
}
```

It's important to note, that given the multidimensional nature of `double**` data, we need to:

- de-allocate each row individually: `free(matrix->data[i])`;
- then the column vector: `free(matrix->data)`;
- and lastly the actual struct: `free(matrix)`.

At this point, it's a good idea to add more methods to help the potential use of the library to create various `nml_mat structs`, with various properties.

Method	Description
`nml_mat_rnd()`	A method to create a random matrix.
`nml_mat_sqr()`	A method to create a square matrix with elements 0.0.
`nml_mat_eye()`	A method to create an identity matrix.
`nml_mat_cp()`	A method to copy the content of a matrix into another matrix.
`nml_mat_fromfile()`	A method to read the matrix from a FILE.

4 Creating a random matrix

Writing a method like `nml_mat_rnd()` it's easy, once we have `nml_mat_new()` in place:

```
nml_mat *nml_mat_rnd(
  unsigned int num_rows,
  unsigned int num_cols,
  double min,
  double max
  )
```

```
{
  nml_mat *r = nml_mat_new(num_rows, num_cols);
  int i, j;
  for(i = 0; i < num_rows; i++) {
    for(j = 0; j < num_cols; j++) {
      r->data[i][j] = nml_rand_interval(min, max);
    }
  }
  return r;
}
```

The input params min and max represent the interval boundaries in which the random numbers are being generated.
The `nml_rand_interval(min, max)`, the method responsible for generating random values, looks like this:

```
#define RAND_MAX 0x7fffffff

double nml_rand_interval(double min, double max) {
  double d;
  d = (double) rand() / ((double) RAND_MAX + 1);
  return (min + d * (max - min));
}
```

5 Creating a square matrix

A square matrix has the same number of columns and rows.

For example, A is square 3x3 matrix:

$$A = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{pmatrix}$$

but, B is not a square matrix:

$$B = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \end{pmatrix}$$

Implementing this is as simple as calling the existing `nml_mat_new()` function with `rows=cols`:

```
nml_mat *nml_mat_sqr(unsigned int size) {
  return nml_mat_new(size, size);
}
```

Similarly, you can write a method to produce a random square matrix:

```
nml_mat *nml_mat_sqr_rnd(unsigned int size, double min, double max) {
  return nml_mat_rnd(size, size, min, max);
}
```

EXAMPLE. Create a matrix `A` of 3×3, and a random matrix `B` of 4×4.

```
#include "nml.h"

nml_mat *A = nml_mat_sqr(3);
nml_mat *B = nml_mat_sqr_rnd(4);
```

---

6 Creating an identity matrix

An identity matrix is a square ($N \times N$) matrix that has 1.0 on the first diagonal, and 0.0 elsewhere:

$$I_n = \underbrace{\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}}_{n \text{ columns}} \left.\vphantom{\begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}}\right\} n \text{ rows}$$

A matrix multiplied with its inverse is equal to the identity matrix: $A^{-1} \times A = A \times A^{-1} = I$.

From a programming perspective, the first diagonal represents the series of matrix elements for which the indexes `i` and `j` are equal (`i==j`).

`i` represents the row index, while `j` represents the column index.

Having said this, our method looks like this:

```
nml_mat *nml_mat_eye(unsigned int size) {
  nml_mat *r = nml_mat_new(size, size);
  int i;
  for(i = 0; i < r->num_rows; i++) {
    r->data[i][i] = 1.0;
  }
  return r;
}
```

To find out the reasons why the identity method is named `eye()` please read *this math exchange post*

7 Creating a matrix from a FILE

Instead of having to write something like the code bellow to set the elements of the matrix:

```
nml_mat *m = ...
m->data[0][0] = 1.0;
m->data[1][0] = 2.0;
m->data[2][0] = 4.0;
// etc.
```

It's more convenient to allow the user of our library to be able to read the matrix elements from an input text file.
The input file should be formatted in a certain way, e.g.:

```
matrix01.data
----------------
4 5
0.0     1.0     2.0     5.0     3.0
3.0     8.0     9.0     1.0     4.0
2.0     3.0     7.0     1.0     1.0
0.0     0.0     4.0     3.0     8.0
```

- The first row of the file 4 5 represents the numbers of rows (=4) and columns (=5);
- The rest of the rows are the elements (20) of the matrix.

The C code that is able to read this file is:

```
nml_mat *nml_mat_fromfilef(FILE *f) {
  int i, j;
  unsigned int num_rows = 0, num_cols = 0;
  fscanf(f, "%d", &num_rows);
  fscanf(f, "%d", &num_cols);
  nml_mat *r = nml_mat_new(num_rows, num_cols);
  for(i = 0; i < r->num_rows; i++) {
    for(j = 0; j < num_cols; j++) {
      fscanf(f, "%lf\t", &r->data[i][j]);
    }
  }
  return r;
}
```

Where:

- `fscanf(f, "%d", &num_rows)` and `fscanf(f, "%d", &num_cols)` read the first line;
- The `fscanf(f, "%lf\t", &r->data[i][j])` line inside the for loops read the remaining elements of the matrix.

This method can also be used to read user input from the keyboard, by calling the method like this:

```
nml_mat_fromfilef(stdin);
```

8 Matrix methods

8.1 Checking for equality

It will be nice for the users of our library to be able to compare two matrices and see if they are equal, meaning they have the same number of rows and columns and identical elements.
In practice, it's good to be able to check if two matrices are "almost equal", meaning that their elements are "almost equal" within an accpetable tolerance.
Writing a method like this is trivial. We basically have to iterate over all the elements and check for their equality:

```
// Checks if two matrices have the same dimensions
int nml_mat_eqdim(nml_mat *m1, nml_mat *m2) {
  return (m1->num_cols == m2->num_cols) &&
          (m1->num_rows == m2->num_rows);
}

// Checks if two matrices have the same dimensions, and the elements
// are all equal to each other with a given tolerance;
// For exact equality use tolerance = 0.0
int nml_mat_eq(nml_mat *m1, nml_mat *m2, double tolerance) {
  if (!nml_mat_eqdim(m1, m2)) {
    return 0;
  }
  int i, j;
  for(i = 0; i < m1->num_rows; i++) {
    for(j = 0; j < m1->num_cols; j++) {
      if (fabs(m1->data[i][j] - m2->data[i][j]) > tolerance) {
        return 0;
      }
    }
  }
  return 1;
}
```

`fabs(x)` returns the absolute value of `x` → `|x|`.

8.2 Printing the matrix

Printing the matrix is trivial. We just need to iterate through all the elements and use `fprintf()` to show the matrix in `stdout`:

```
void nml_mat_print(nml_mat *matrix) {
  nml_mat_printf(matrix, "%lf\t\t");
}

// Prints the matrix on the stdout (with a custom formatting for elements)
void nml_mat_printf(nml_mat *matrix, const char *d_fmt) {
  int i, j;
  fprintf(stdout, "\n");
  for(i = 0; i < matrix->num_rows; ++i) {
    for(j = 0; j < matrix->num_cols; ++j) {
      fprintf(stdout, d_fmt, matrix->data[i][j]);
    }
    fprintf(stdout, "\n");
  }
  fprintf(stdout, "\n");
}
```

8.3 Retrieving / Selecting a column

Some advanced numerical analysis algorithms (e.g.: QR decomposition) are working extensively on columns, so it's a good idea to be pro-active about it and create a method that selects/retrieves a column from a given matrix.

We will define this method as:

```
nml_mat *nml_mat_col_get(nml_mat *m, unsigned int col)
```

From a mathematical perspective calling our method on given matrix retrieves another column matrix:

$$\texttt{nml_mat_col_get}\left(\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}, 1\right) = \begin{bmatrix} 2.0 \\ 2.0 \\ 1.0 \end{bmatrix}$$

The C code for `nml_mat_col_get` looks like this:

```
nml_mat *nml_mat_col_get(nml_mat *m, unsigned int col) {
  if (col >= m->num_cols) {
    NML_FERROR(CANNOT_GET_COLUMN, col, m->num_cols);
    return NULL;
  }
  nml_mat *r = nml_mat_new(m->num_rows, 1);
  int j;
  for(j = 0; j < r->num_rows; j++) {
    r->data[j][0] = m->data[j][col];
  }
```

```
  return r;
}
```

Observations:

- The resulting matrix has only one column: `nml_mat *r = nml_mat_new(m->num_rows, 1);`
- We copy all elements from column `[col]` to the only column of the resulting matrix `[0]`: `r->data[j][0] = m->data[j][col]`.

8.4 Retrieving / Selecting a row

To keep the API "consistent" we can write a similar method for retrieving a row:

```
  nml_mat *nml_mat_row_get(nml_mat *m, unsigned int row)
```

This will work similar to the `nml_mat_col_get(...)`, but instead of retrieving a column we will retrieve a row:

$$\texttt{nml_mat_row_get(} \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix} \texttt{,1) =} \begin{bmatrix} 0.0 & 2.0 & 3.0 \end{bmatrix}$$

The C implementation for this method looks like this:

```
nml_mat *nml_mat_row_get(nml_mat *m, unsigned int row) {
  if (row >= m->num_rows) {
    NML_FERROR(CANNOT_GET_ROW, row, m->num_rows);
    return NULL;
  }
  nml_mat *r = nml_mat_new(1, m->num_cols);
  memcpy(r->data[0], m->data[row], m->num_cols * sizeof(*r->data[0]));
  return r;
}
```

This time we write the code differently. Given the fact the memory per row is contiguous we can make use of `memcpy()`.

No loops are needed this time. This line of code is enough to achieve what we want:

```
  memcpy(r->data[0], m->data[row], m->num_cols * sizeof(*r->data[0]))
```

At this point we've created a new row matrix from the initial one `(m)`.

8.5 Setting values

To set the element of the matrix to a given value, we can simply access the data field of the `nml_mat*` struct:

```
nml_mat *m = ...
m->data[i][j] = 2.0;
```

In addition, we can write helper methods to:

- Set all the elements to a given value: `void nml_mat_all_set(nml_mat *matrix, double value)`
- Set all the elements of the first diagonal to a given value: `int nml_mat_diag_set(nml_mat *m, double value)`

The C code for those two is somewhat trivial:

```
// Sets all elements of a matrix to a given value
void nml_mat_all_set(nml_mat *matrix, double value) {
  int i, j;
  for(i = 0; i < matrix->num_rows; i++) {
    for(j = 0; j < matrix->num_cols; j++) {
      matrix->data[i][j] = value;
    }
  }
}

// Sets all elements of the matrix to given value
int nml_mat_diag_set(nml_mat *m, double value) {
  if (!m->is_square) {
    NML_FERROR(CANNOT_SET_DIAG, value);
    return 0;
  }
  int i;
  for(i = 0; i < m->num_rows; i++) {
    m->data[i][i] = value;
  }
  return 1;
}
```

8.6 Multiplying a row with a scalar

Multiplying a row in the matrix (`nml_mat`) can be useful when implementing more numerical analysis advanced algorithms.

The idea is simple, we will define a method with the following signature:

```
int nml_mat_row_mult_r(nml_mat *m, unsigned int row, double num);
```

This method will work directly through reference on the matrix m. That's where the _r stands for.

From a mathematical perspective this method will do the following:

$$\texttt{nml_mat_row_mult_r(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{,1, 2.0) =}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ \mathbf{0.0} & \mathbf{4.0} & \mathbf{6.0} \\ 2.0 & 1.0 & 9.0 \end{bmatrix}$$

The code implementation looks like this:

```
int nml_mat_row_mult_r(nml_mat *m, unsigned int row, double num) {
  if (row>= m->num_rows) {
    NML_FERROR(CANNOT_ROW_MULTIPLY, row, m->num_rows);
    return 0;
  }
  int i;
  for(i=0; i < m->num_cols; i++) {
    m->data[row][i] *= num;
  }
  return 1;
}
```

Notice how we select the row: `m->data[row][i] *= num`, where `i = 0 .. m->num_cols`.

An alternative method, that instead of referencing m will retrieve a new nml_mat *r, can be written like this:

```
nml_mat *nml_mat_row_mult(nml_mat *m, unsigned int row, double num) {
  nml_mat *r = nml_mat_cp(m);
  if (!nml_mat_row_mult_r(r, row, num)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}
```

Notice how the _r ending has dropped. This is a common pattern in C.

8.7 Multiplying a column with a scalar

Multiplying a column is also quite similar with what we described above.
We are going to end-up with two methods:

- `int nml_mat_col_mult_r(nml_mat *m, unsigned int col, double num)`
  - This will modify the matrix m through reference;
- `nml_mat *nml_mat_col_mult(nml_mat *m, unsigned int col, double num)`
  - This will return a `new nml_mat *r matrix`

From a math perspective:

$$\texttt{nml_mat_col_mult_r(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{,0, 2.0) =}\begin{bmatrix} \mathbf{1.0} & 2.0 & 3.0 \\ \mathbf{0.0} & 2.0 & 3.0 \\ \mathbf{4.0} & 1.0 & 9.0 \end{bmatrix}$$

The C code for both of the methods looks like this:

```
nml_mat *nml_mat_col_mult(nml_mat *m, unsigned int col, double num) {
  nml_mat *r = nml_mat_cp(m);
  if (!nml_mat_col_mult_r(r, col, num)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}

int nml_mat_col_mult_r(nml_mat *m, unsigned int col, double num) {
  if (col>=m->num_cols) {
    NML_FERROR(CANNOT_COL_MULTIPLY, col, m->num_cols);
    return 0;
  }
  int i;
  for(i = 0; i < m->num_rows; i++) {
    m->data[i][col] *= num;
  }
  return 1;
}
```

Notice how we select the column: `m->data[i][col] *= num` , where `i = 0 .. m->num_rows`.

8.8 Adding two rows

The ability to add one row to another, is an important method used later for the implementation of more advanced algorithms: LUP Decomposition, Row Echelon Form, Reduced Row Echelon Form, etc.

In addition, before adding one row to another we should also offer the possibility to multiply the row with a given scalar.

We define the following method(s):

```
// We add all elements from row 'row' to row 'where'.
// The elements from row 'row' are muliplied using the 'multiplier'
//
// This one works through reference, modifying the `m` matrix;
int nml_mat_row_addrow_r(nml_mat *m, unsigned int where,
unsigned int row, double multiplier);

// We add all elements from row 'row' to row 'where'.
// The elements from row 'row' are muliplied using the 'multiplier'
//
// This one returns a new matrix, `nml_mat *r`, after the row addition is performed
* nml_mat *nml_mat_row_addrow(nml_mat *m, unsigned int where, unsigned int row,
double multiplier);
```

To better visualise:

$$\texttt{nml_mat_row_addrow_r(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{,0, 1, 0.5) =}$$

$$\begin{bmatrix} 1.0 + 0 \times 0.5 & 2.0 + 2 \times 0.5 & 3.0 + 4.0 \times 0.5 \\ 0.0 & 2.0 & 3.0 \\ 4.0 & 1.0 & 9.0 \end{bmatrix} = \begin{bmatrix} 1.0 & 2.0 & 5.0 \\ 0.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}$$

The corresponding C code for the two methods is:

```
nml_mat *nml_mat_row_addrow(nml_mat *m, unsigned int where,
unsigned int row, double multiplier) {
  nml_mat *r = nml_mat_cp(m);
  if (!nml_mat_row_addrow_r(m, where, row, multiplier)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}

int nml_mat_row_addrow_r(nml_mat *m, unsigned int where,
unsigned int row, double multiplier) {

  if (where >= m->num_rows || row >= m->num_rows) {
    NML_FERROR(CANNOT_ADD_TO_ROW, multiplier, row, where, m->num_rows);
    return 0;
  }
  int i = 0;
  for(i = 0; i < m->num_cols; i++) {
```

```
    m->data[where][i] += multiplier * m->data[row][i];
  }
  return 1;
}
```

Notice how we are selecting the rows: `m->data[where][i] += multiplier * m->data[row][i]`, where `i = 0 .. i < m->num_cols`.

In case it's not obvious, if the user simply wants to add two rows, without any multiplication, the multiplier should be kept as 1.0.

8.9 Multiplying the matrix by a scalar

The mathematical formula for multiplying the matrix with a scalar is simple:

$$s * \begin{bmatrix} a_{01} & a_{02} & \dots & a_{0n} \\ a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} = \begin{bmatrix} s * a_{01} & s * a_{02} & \dots & s * a_{0n} \\ s * a_{11} & s * a_{12} & \dots & s * a_{1n} \\ \vdots & & & \vdots \\ s * a_{m1} & s * a_{m2} & \dots & s * a_{mn} \end{bmatrix}$$

So, just like the formula, the code equivalent is simple:

```
nml_mat *nml_mat_smult(nml_mat *m, double num) {
  nml_mat *r = nml_mat_cp(m);
  nml_mat_smult_r(r, num);
  return r;
}

int nml_mat_smult_r(nml_mat *m, double num) {
  int i, j;
  for(i = 0; i < m->num_rows; i++) {
    for(j = 0; j < m->num_cols; j++) {
      m->data[i][j] *= num;
    }
  }
  return 1;
}
```

For each element `m->data[i][j]` we perform the multiplication with the scalar num: `m->data[i][j] *= num` where `i = 0 .. num_rows` and `j = 0 .. num_cols`.

9 Modifying the `nml_mat` internal structure

The next set of functionalities we can write to help our potential library users to modify the `nml_mat` matrix structure are:

- Remove a columns and rows and return a new matrix;
- Swap rows inside a given matrix;
- Swap columns inside a given matrix;
- Concatenate vertically and horizontally two matrices;

9.1 Removing a column

Removing a column from aM[n x m] matrix, involves the creation of a new `[n x (m-1)]` matrix.

The method signature looks like this:

```
nml_mat *nml_mat_col_rem(nml_mat *m, unsigned int column);
```

For this particular use-case it would be overkill to try to create a "by-reference" `(_r)` version of the method.

Calling the `nml_mat_col_rem` on a matrix yields the following results:

$$\texttt{nml_mat_col_rem(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{,1) =}\begin{bmatrix} 1.0 & 3.0 \\ 0.0 & 4.0 \\ 2.0 & 9.0 \end{bmatrix}$$

The code implementation:

```
nml_mat *nml_mat_col_rem(nml_mat *m, unsigned int column) {
  if(column >= m->num_cols) {
    NML_FERROR(CANNOT_REMOVE_COLUMN, column, m->num_cols);
    return NULL;
  }
  nml_mat *r = nml_mat_new(m->num_rows, m->num_cols-1);
  int i, j, k;
  for(i = 0; i < m->num_rows; i++) {
    for(j = 0, k=0; j < m->num_cols; j++) {
      if (column!=j) {
        r->data[i][k++] = m->data[i][j];
      }
    }
  }
  return r;
}
```

Observations:

- The resulting r matrix has the number of columns `m->num_cols-1`;

- We keep a separate column index for the `r` matrix that we name `k`;
- When copying the elements from `m` to `r` we skip the column column by adding this condition `(column!=j)`:
  - Then we increment `k`, using `k++` inside the `r->data[i][k++]` statement;
  - From this moment onwards `k-j == 1`, meaning `k` and `j` are no longer in sync, because we've skipped the column.

9.2 Removing a row

Removing a row from a `M[n x m]` matrix, involves the creation of a new `[(n-1) x m]` matrix.

The method signature looks like this:

```
nml_mat *nml_mat_row_rem(nml_mat *m, unsigned int row);
```

Calling the `nml_mat_row_rem` on a matrix yields the following results:

$$\texttt{nml_mat_row_rem(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{,1) =}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}$$

The code implementation:

```
nml_mat *nml_mat_row_rem(nml_mat *m, unsigned int row) {
  if (row >= m->num_rows) {
    NML_FERROR(CANNOT_REMOVE_ROW, row, m->num_rows);
    return NULL;
  }
  nml_mat *r = nml_mat_new(m->num_rows-1, m->num_cols);
  int i, j, k;
  for(i = 0, k = 0; i < m->num_rows; i++) {
    if (row!=i) {
      for(j = 0; j < m->num_cols; j++) {
        r->data[k][j] = m->data[i][j];
      }
      k++;
    }
  }
  return r;
}
```

Observations:

- The resulting matrix `r` has the same number of columns as `m` (i.e., `m->num_cols`), but a smaller number of rows (`r->num_rows`);

- We keep a separate row index `k` for the resulting matrix `'r'`;
- Initially `k` is in sync with `i`, as long as (`row!=i`);
- When `row == i`, `k` is no longer incremented, so the sync is lost and `i - k == 1`. This is how the row gets skipped.

9.3 Swapping Rows

This functionality will prove useful later when we re going to implement the Row Echelon Form and LU Decomposition algorithms.

In this case we can define two methods:

```
// Returns a new matrix with row1 and row2 swapped
nml_mat *nml_mat_row_swap(nml_mat *m, unsigned int row1, unsigned int row2);

// Modifies the existing matrix m, by swapping the two rows row1 and row2
int nml_mat_row_swap_r(nml_mat *m, unsigned int row1, unsigned int row2);
```

Visually, the method works like this:

$$\texttt{nml_mat_row_swap_r(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{0,1) =}\begin{bmatrix} 0.0 & 2.0 & 4.0 \\ 1.0 & 2.0 & 3.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}$$

The C implementation makes use of the fact that rows are contiguous memory blocks can be swapped without having to access each element of the rows in particular:

```
int nml_mat_row_swap_r(nml_mat *m, unsigned int row1, unsigned int row2) {
  if (row1 >= m->num_rows || row2 >= m->num_rows) {
    NML_FERROR(CANNOT_SWAP_ROWS, row1, row2, m->num_rows);
    return 0;
  }
  double *tmp = m->data[row2];
  m->data[row2] = m->data[row1];
  m->data[row1] = tmp;
  return 1;
}
```

As for the `nml_mat_row_swap(..)` this can be written by re-using `nml_mat_row_swap_r(...)`:

```
nml_mat *nml_mat_row_swap(nml_mat *m, unsigned int row1, unsigned int row2) {
  nml_mat *r = nml_mat_cp(m);
  if (!nml_mat_row_swap_r(r, row1, row2)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}
```

9.4 Swapping columns

This functionality might not be as useful as the previous one `nml_mat_row_swap(...)`, but for sake of having a robust API for our potential users, we will implement it.

We define again two methods, one that is returning a new `nml_mat` matrix, and one that operates on the given on:

```
nml_mat *nml_mat_col_swap(nml_mat *m, unsigned int col1, unsigned int col2);
int nml_mat_col_swap_r(nml_mat *m, unsigned int col1, unsigned int col2);
```

Visually the two methods are working like this:

$$\texttt{nml_mat_row_swap_r(}\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}\texttt{0,1) =}\begin{bmatrix} 2.0 & 1.0 & 4.0 \\ 2.0 & 0.0 & 3.0 \\ 1.0 & 2.0 & 9.0 \end{bmatrix}$$

Compared to the previous two functions (for swapping rows) the code is slightly different. Columns are not contiguous blocks of memory, so we will need to swap each element one by one:

```
int nml_mat_col_swap_r(nml_mat *m, unsigned int col1, unsigned int col2) {
  if (col1 >= m->num_cols || col2 >= m->num_rows) {
    NML_FERROR(CANNOT_SWAP_ROWS, col1, col2, m->num_cols);
    return 0;
  }
  double tmp;
  int j;
  for(j = 0; j < m->num_rows; j++) {
    tmp = m->data[j][col1];
    m->data[j][col1] = m->data[j][col2];
    m->data[j][col2] = tmp;
  }
  return 1;
}
```

Writing the `nml_mat_col_swap(...)` version of the method will simply re-use the previous "`_r`" one:

```
nml_mat *nml_mat_col_swap(nml_mat *m, unsigned int col1, unsigned int col2) {
  nml_mat *r = nml_mat_cp(m);
  if (!nml_mat_col_swap_r(r, col1, col2)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}
```

9.5 Horizontal Concatenation of two matrices

This functionality is probably not very useful from a "scientific" point of view, but it's a nice exercise we can solve, and a neat "utility" we can add to the library.

We would like to write a function, that takes a variable number of matrices (`nml_mat**`) and returns a new matrix that represents the horizontal concatenation of those matrices.

It's important that all the input matrices have the same number of columns, otherwise the horizontal concatenation won't work.

Our C function will have the following signature:

```
nml_mat *nml_mat_cath(unsigned int mnum, nml_mat **marr);
```

Where:

- `unsigned int mnum` – represents the total number of matrices we want to (horizontally) concatenate;
- `nml_mat **marr` – are the matrices we want to (horizontally) concatenate;

Visually, the function works on the following way. If:

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 4.0 & 0.0 & 9.0 \end{bmatrix}$$

$$C = \begin{bmatrix} 3.0 & -1.0 & 1.0 \\ 2.0 & 0.0 & -5.0 \end{bmatrix}$$

Calling the method `nml_mat_cath(3, **[A, B, C])` will yield the following result:

$$C = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \\ 2.0 & 1.0 & 9.0 \\ 4.0 & 0.0 & 9.0 \\ 3.0 & -1.0 & 1.0 \\ 2.0 & 0.0 & -5.0 \end{bmatrix}$$

The corresponding C code for this implementation looks like this:

```
nml_mat *nml_mat_cath(unsigned int mnum, nml_mat **marr) {
  if (0==mnum) {
    // No matrices, nothing to return
    return NULL;
```

```
  }
  if (1==mnum) {
    // We just return the one matrix supplied as the first param
    // no need for additional logic
    return nml_mat_cp(marr[0]);
  }
  // We calculate the total number of columns to know how to allocate memory
  // for the resulting matrix
  int i,j,k,offset;
  unsigned int lrow, ncols;
  lrow = marr[0]->num_rows;
  ncols = marr[0]->num_cols;
  for(k = 1; k < mnum; k++) {
    if (NULL == marr[k]) {
      NML_FERROR(INCONSITENT_ARRAY, k, mnum);
      return NULL;
    }
    if (lrow != marr[k]->num_rows) {
      NML_FERROR(CANNOT_CONCATENATE_H, lrow, marr[k]->num_rows);
      return NULL;
    }
    ncols+=marr[k]->num_cols;
  }
  // At this point we know how the resulting matrix looks like,
  // we allocate memory for it accordingly
  nml_mat *r = nml_mat_new(lrow, ncols);
  for(i = 0; i < r->num_rows; i++) {
    k = 0;
    offset = 0;
    for(j = 0; j < r->num_cols; j++) {
      // If the column index of marr[k] overflows
      if (j-offset == marr[k]->num_cols) {
        offset += marr[k]->num_cols;
        // We jump to the next matrix in the array
        k++;
      }
      r->data[i][j] = marr[k]->data[i][j - offset];
    }
  }
  return r;
}
```

Observations:

- `i`, `j` are used to iterate over the resulting matrix (tt r);

- `k` is the index of the current we are concatenating;
- `offset` is useful to determine we need to jump to next matrix that needs concatenation.

9.6 Vertical concatenation

Just like the horizontal concatenation, this functionality is not very useful for the more complex algorithms we are going to implement later in this tutorial. But, for the sake of our potential library users, and because it's a nice exercise we will implement it.

The main idea is to write a function, that takes a variable number of matrices (nml_mat**) and returns a new matrix that represents the vertical concatenation of those matrices.

It's important that all the input matrices have the same number of rows, otherwise the vertical concatenation won't work.

The method signature looks like:

```
nml_mat *nml_mat_catv(unsigned int mnum, nml_mat **marr);
```

Where:

- `unsigned int mnum` – represents the total number of matrices we want to (horizontally) concatenate;
- `nml_mat **marr` – are the matrices we want to (horizontally) concatenate;

Visually the method works like this:

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 0.0 & 2.0 & 4.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 4.0 & 0.0 & 9.0 \\ 2.0 & 1.0 & 9.0 \end{bmatrix}$$

Calling `nml_mat_catv(2, **[A, B])` will return the following result:

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 9.0 \\ 0.0 & 2.0 & 4.0 & 2.0 & 1.0 & 9.0 \end{bmatrix}$$

The code implementation looks like this:

```
// Concatenates a variable number of matrices into one.
// The concentation is done vertically this means the matrices need to have
// the same number of columns, while the number of rows is allowed to
// be variable
nml_mat *nml_mat_catv(unsigned int mnum, nml_mat **marr) {
  if (0 == mnum) {
```

```
    return NULL;
  }
  if (1 == mnum) {
    return nml_mat_cp(marr[0]);
  }
  // We check to see if the matrices have the same number of columns
  int lcol, i, j, k, offset;
  unsigned int numrows;
  nml_mat *r;
  lcol = marr[0]->num_cols;
  numrows = 0;
  for(i = 0; i < mnum; i++) {
    if (NULL==marr[i]) {
      NML_FERROR(INCONSITENT_ARRAY, i, mnum);
      return NULL;
    }
    if (lcol != marr[i]->num_cols) {
      NML_FERROR(CANNOT_CONCATENATE_V,lcol,marr[i]->num_cols);
      return NULL;
    }
    // In the same time we calculate the resulting matrix number of rows
    numrows+=marr[i]->num_rows;
  }
  // At this point we know the dimensions of the resulting Matrix
  r = nml_mat_new(numrows, lcol);
  // We start copying the values one by one
  for(j = 0; j < r->num_cols; j++) {
    offset = 0;
    k = 0;
    for(i = 0; i < r->num_rows; i++) {
      if (i - offset == marr[k]->num_rows) {
        offset += marr[k]->num_rows;
        k++;
      }
      r->data[i][j] = marr[k]->data[i-offset][j];
    }
  }
  nml_mat_print(r);
  return r;
}
```

Observations:

- `i`, `j` are used to iterate over the resulting matrix (`r`);
- Compared to our previous method (`nml_mat_cath(..)`) this time we start by iterating

though the columns;

- k is the index of the current matrix we are concatenating;
- `offset` is useful to determine we need to jump to next matrix that needs concatenation

10 Basic Matrix Operations

10.1 Add two matrices

From a mathematical perspective the formula for adding two matrices A and B is quit simple:

$$\begin{bmatrix} a_{01} & a_{02} & \dots & a_{0n} \\ a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{01} & b_{02} & \dots & b_{0n} \\ b_{11} & b_{12} & \dots & b_{1n} \\ \vdots & & & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}$$

$$\begin{bmatrix} a_{01}+b_{01} & a_{02}+b_{02} & \dots & a_{0n}+b_{0n} \\ a_{11}+b_{11} & a_{12}+b_{12} & \dots & a_{1n}+b_{1n} \\ \vdots & & & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \dots & a_{mn}+b_{mn} \end{bmatrix}$$

Basically each element from the first matrix gets added with the corresponding element from the second matrix.

The corresponding C code is straightforward:

```
nml_mat *nml_mat_add(nml_mat *m1, nml_mat *m2) {
  nml_mat *r = nml_mat_cp(m1);
  if (!nml_mat_add_r(r, m2)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}

int nml_mat_add_r(nml_mat *m1, nml_mat *m2) {
  if (!nml_mat_eqdim(m1, m2)) {
    NML_ERROR(CANNOT_ADD);
    return 0;
  }
  int i, j;
  for(i = 0; i < m1->num_rows; i++) {
    for(j = 0; j < m2->num_rows; j++) {
      m1->data[i][j] += m2->data[i][j];
    }
```

```
  }
  return 1;
}
```

10.2 Substracting two matrices

This is very similar with to the addition, except this time each element from m2 is subtracted from the corresponding element from m1:

$$\begin{bmatrix} a_{01} & a_{02} & \dots & a_{0n} \\ a_{11} & a_{12} & \dots & a_{1n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} - \begin{bmatrix} b_{01} & b_{02} & \dots & b_{0n} \\ b_{11} & b_{12} & \dots & b_{1n} \\ \vdots & & & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}$$

$$\begin{bmatrix} a_{01} - b_{01} & a_{02} - b_{02} & \dots & a_{0n} - b_{0n} \\ a_{11} - b_{11} & a_{12} - b_{12} & \dots & a_{1n} - b_{1n} \\ \vdots & & & \vdots \\ a_{m1} - b_{m1} & a_{m2} - b_{m2} & \dots & a_{mn} - b_{mn} \end{bmatrix}$$

The corresponding C code to perform this operation is:

```
nml_mat *nml_mat_sub(nml_mat *m1, nml_mat *m2) {
  nml_mat *r = nml_mat_cp(m2);
  if (!nml_mat_sub_r(r, m2)) {
    nml_mat_free(r);
    return NULL;
  }
  return r;
}

int nml_mat_sub_r(nml_mat *m1, nml_mat *m2) {
  if (!nml_mat_eqdim(m1, m2)) {
    NML_ERROR(CANNOT_SUBTRACT);
    return 0;
  }
  int i, j;
  for(i = 0; i < m1->num_rows; i++) {
    for(j = 0; j < m2->num_cols; j++) {
      m1->data[i][j] -= m2->data[i][j];
    }
  }
  return 1;
}
```

10.3 Multiplying two matrices

Having a matrix $A[m \times x]$, and a matrix $B[n \times p]$:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & & & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{bmatrix}$$

We define the product $A \times B$, as the matrix $A[m \times p]$:

$$C = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & & & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{bmatrix}$$

Where:

$$c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots + a_{in} \cdot b_{nj} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj},$$

for $i = 1 \,..\, m$, $j = 1 \,..\, p$. The product $A \times B$ is defined if and only if the number of columns of A equals the number of rows in B, which is n.

The resulting product matrix will then "inherit" the number of rows from A, and the number of columns from B.

The formula will be easier to digest if we go through an example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 4 \end{bmatrix}, \qquad B = \begin{bmatrix} 2 & 3 \\ 2 & 1 \\ 1 & 5 \end{bmatrix}$$

$A \times B$ exists because $A[2 \times 3]$ and $B[3 \times 2]$. The resulting matrix C will be 2×2.

$$C = \begin{bmatrix} 1 \cdot 2 + 2 \cdot 2 + 3 \cdot 1 & 1 \cdot 3 + 2 \cdot 1 + 3 \cdot 5 \\ 0 \cdot 2 + 0 \cdot 2 + 4 \cdot 1 & 0 \cdot 3 + 0 \cdot 1 + 4 \cdot 5 \end{bmatrix} = \begin{bmatrix} 9 & 20 \\ 4 & 20 \end{bmatrix}$$

The naive implementation for this algorithm looks like:

```
ml_mat *nml_mat_dot(nml_mat *m1, nml_mat *m2) {
  if (!(m1->num_cols == m2->num_rows)) {
    NML_ERROR(CANNOT_MULITPLY);
    return NULL;
  }
  int i, j, k;
  nml_mat *r = nml_mat_new(m1->num_rows, m2->num_cols);
  for(i = 0; i < r->num_rows; i++) {
    for(j = 0; j < r->num_cols; j++) {
```

```
      for(k = 0; k < m1->num_cols; k++) {
        r->data[i][j] += m1->data[i][k] * m2->data[k][j];
      }
    }
  }
  return r;
}
```

Better algorithms exists for matrix multiplications, if you want to find out more please check this wikipedia *article*.

11 Row Echelon Form

A matrix A is in *Row Echelon Form* if it has the shape resulting from a Gaussian Elimination.

Additionally, the matrix A is in Row Echelon form if:

- The first non-zero element for each row is exactly 1.0;
- Rows with all 0.0 elements are bellow rows that have at least one non-zero element.
- Each leading entry (pivot) is in a column to the right of the leading entry in the previous row.

All the bellow matrices are examples of matrices that have been "morphed" into Row Echelon Form:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} C = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Every matrix can be transformed into a Row Echelon, by using elementary row operations:

- Interchanging (swapping) two rows. See `nml_mat_row_swap_r(...)` implemented before;
- Multiply each element in a row by a non-zero number (scalar multiplication of rows). See `nml_mat_row_mult_r(...)` implemented before;
- Multiply a row by a non-zero number and add the result to another row (row addition). See `nml_mat_row_addrow_r(...)` implemented before;

The algorithm to transform the matrix in a Row Echelon Form is as follows:

1. Find the "pivot", the first non-zero entry from the first column of the matrix;
    - If the column has only zero elements, jump to the next column;
2. Interchange rows, moving the pivot row to become the first row;

3. Multiply each element in the pivot by the inverse of the pivot $1/pivot$ so that the pivot equals 1.0;
4. Add multiplies of the pivot row to each of the pivot rows, so every element in the pivot column will equal 0.0.
5. Continue the process until there are no more pivots to process.

Note: A matrix can have multiple Row Echelon Forms, but you will see in the next chapter, there's only one Reduced Row Echelon Form.

11.1 Example

Let's take for example the following matrix, $A[3\times3]$. `REF`(A) is also a $3x3$ matrix. The transitions are:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 7 & 8 \end{bmatrix} \rightarrow A_1 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 2 & 7 & 8 \end{bmatrix} \rightarrow A_2 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 3 & 6 \end{bmatrix} \rightarrow A_{ref} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

$A \rightarrow A_1$: We found out that the first non-zero element of the first `column[0]` is 1 on `row[1]` so we've swapped `row[0]` with `row[1]`. Using our code this means:

$$\texttt{nml_mat_row_swap_r(} \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 7 & 8 \end{bmatrix} \texttt{, 0, 1)=} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 2 & 7 & 8 \end{bmatrix}$$

$A_1 \rightarrow A_2$: For A_1 we've multiplied each element of `row[0]` with -2 and added the result to `row[2]`. Using our code this means:

$$\texttt{nml_mat_row_addrow_r(} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 2 & 7 & 8 \end{bmatrix} \texttt{, 2, 0, -2.0)=} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 3 & 6 \end{bmatrix}$$

$A_2 \rightarrow A_{ref}$: For A_2 we've multiplied `row[1]` with -3 and added the result to `row[2]`. Using the code this means:

$$\texttt{nml_mat_row_addrow_r(} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 3 & 6 \end{bmatrix} \texttt{, 2, 1, -3.0)=} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

11.2 Code implementation

Most of the bits and pieces for assembling the algorithm, were already implemented before: `nml_mat_row_swap_r(...)`, `nml_mat_row_mult_r(...)`, `nml_mat_row_addrow_r(...)`.

What we are missing at this moment to assemble the algorithm is to write the "pivot function". We are going to call this: `_nml_mat_pivotidx(...)`:

```
// Finds the first non-zero element on the col column, under the row row.
// This is used to determine the pivot in gauss Elimination
// If not pivot is found, returns -1
int _nml_mat_pivotidx(nml_mat *m, unsigned int col, unsigned int row) {
  // No validations are made, this is an API Method
  int i;
  for(i = row; i < m->num_rows; i++) {
    if (fabs(m->data[i][col]) > NML_MIN_COEF) {
      return i;
    }
  }
  return -1;
}
```

At this point the algorithm is straight-forward to implement:

```
// Retrieves the matrix in Row Echelon form using Gauss Elimination
nml_mat *nml_mat_ref(nml_mat *m) {
  nml_mat *r = nml_mat_cp(m);
  int i, j, k, pivot;
  j = 0, i = 0;
  // We iterate until we exhaust the columns and the rows
  while(j < r->num_cols && i < r->num_cols) {
    // Find the pivot - the first non-zero entry in the first column of the matrix
    pivot = _nml_mat_pivotidx(r, j, i);
    if (pivot<0) {
      // All elements on the column are zeros
      // We move to the next column without doing anything
      j++;
      continue;
    }
    // We interchange rows moving the pivot to the first row that doesn't have
    // already a pivot in place
    if (pivot!=i) {
      nml_mat_row_swap_r(r, i, pivot);
    }
    // Multiply each element in the pivot row by the inverse of the pivot
    nml_mat_row_mult_r(r, i, 1/r->data[i][j]);
    // We add multiplies of the pivot so every element on the column equals 0
    for(k = i+1; k < r->num_rows; k++) {
      if (fabs(r->data[k][j]) > NML_MIN_COEF) {
        nml_mat_row_addrow_r(r, k, i, -(r->data[k][j]));
      }
    }
    i++;
```

```
    j++;
  }
  return r;
}
```

`1/r->data[i][j]` might pose a risk. If `r->data[i][j]` becomes very small, (like, $0.0000\ldots01$), we might overflow when multiplying wihg `1/r->data[i][j]`. In this regard I've introduced a "guard" value called `NML_MIN_COEF`.

We consider every number smaller than `NML_MIN_COEF` to be 0.0. That's why we perform this additional check: `if (fabs(r->data[k][j]) > NML_MIN_COEF)` in our algorithm.

12 Reduces Row Echelon Form

A matrix A is in *R*educed *R*ow *E*chelon *F*orm, A_{rref} if all the conditions of being in Row Echelon Form are satisfied, and the leading entry in each row is the only non-zero entry in this column.

For example the following matrices are in Reduced Row Echelon Form (RREF):

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Compared to the previous algorithm, an additional step is performed:

1. We identify the last row having a pivot equal to 1;
2. We mark this as the pivot row;
3. We add multiplies of the pivot row to each of it's upper rows, until every element above it remains 0.0;
4. We repeat the process from bottom-up.

Note: A matrix has only RREF form, but can have many REF forms.

12.1 Code Implementation

To make the algorithm more stable from a "computational" perspective we will change the "pivoting method" used above. We introduce a new one:

```
// Find the max element from the column "col" under the row "row"
// This is needed to pivot in Gauss-Jordan elimination
// If pivot is not found, return -1
int _nml_mat_pivotmaxidx(nml_mat *m, unsigned int col, unsigned int row) {
  int i, maxi;
```

```
  double micol;
  double max = fabs(m->data[row][col]);
  maxi = row;
  for(i = row; i < m->num_rows; i++) {
    micol = fabs(m->data[i][col]);
    if (micol>max) {
      max = micol;
      maxi = i;
    }
  }
  return (max < NML_MIN_COEF) ? -1 : maxi;
}
```

Compared to the previous one, this one will return the biggest element on the column under row row. This will be picked as pivot.

The C code:

```
// Retrieves the matrix in Row Echelon form using Gauss Elimination
nml_mat *nml_mat_ref(nml_mat *m) {
  nml_mat *r = nml_mat_cp(m);
  int i, j, k, pivot;
  j = 0, i = 0;
  while(j < r->num_cols && i < r->num_cols) {
    // Find the pivot - the first non-zero entry in the first column of the matrix
    pivot = _nml_mat_pivotidx(r, j, i);
    if (pivot<0) {
      // All elements on the column are zeros
      // We move to the next column without doing anything
      j++;
      continue;
    }
    // We interchange rows moving the pivot to the first row that doesn't have
    // already a pivot in place
    if (pivot!=i) {
      nml_mat_row_swap_r(r, i, pivot);
    }
    // Multiply each element in the pivot row by the inverse of the pivot
    nml_mat_row_mult_r(r, i, 1/r->data[i][j]);
    // We add multiplies of the pivot so every element on the column equals 0
    for(k = i+1; k < r->num_rows; k++) {
      if (fabs(r->data[k][j]) > NML_MIN_COEF) {
        nml_mat_row_addrow_r(r, k, i, -(r->data[k][j]));
      }
    }
```

```
    i++;
    j++;
  }
  return r;
}
```

13 LU(P) Decomposition

LU decomposition, also named LU factorisation refers to the factorisation of a matrix A, into two factors L and U.

Normally the factorisation looks like this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} * \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

In practice, however, this type of factorisation might fail to materialise without swapping various rows of A during the computation. In this case, we need to introduce into the equation a new matrix P where we keep track of all the row changes that are happening during the LU process.

Thus, the decomposition is called LU factorisation with partial pivoting, and the new equation becomes:

$$PA = LU$$

Where:

- P represents any valid (row) permutation of the identity I matrix, and it's computed during the process;
- L is a lower diagonal matrix, with all the elements of the first diagonal `==1`;
- U is an upper diagonal matrix.

There's another factorisation where not only the rows are pivoted, but also columns, this is called LU factorisation with full pivoting but we are not going to implement this.

If the A matrix is square (nxn), it can always be decomposed like $PA = LU$

To compute the LU(P) decomposition we will need to basically implement a modified version of the Gauss Elimination algorithm (see Row Echelon Form). This is probably the most popular implementation, and it requires around $\frac{2}{3}n^3$ floating point operations.

Other algorithms involve direct recursion or randomization. We are not going to implement those versions.

Computing the $PA = LU$ decomposition of matrix A is instrumental for computing the determinant of matrix A, the inverse of matrix A and solving linear systems of equations.

13.1 The LU(P) algorithm as an example

LU(P) factorisation (or decomposition) can be obtained by adjusting the idea of Gaussian Elimination (see Row Echelon Form and Reduced Row Echelon Form).

The algorithm starts like this:

- We allocate memory for the L, U, P matrices
  - L starts as zero matrix;
  - P is the identity matrix;
  - U is an exact copy of A;
- We start iterating the matrix U by columns
  - For each column we look for the pivot value (the biggest value of the column in absolute)
    - If needed we swap the corresponding rows in U, L and P, so that the pivot is position on the first diagonal;
    - If no swap is needed we start creating zeroes on the column by the means of row addition. $Row_x + multiplier * Row_y$.
    - We record the multiplier in matrix L
  - We repeat for every column until U has only zero elements under the first diagonal.

Let's look at the decomposition for a matrix $A[3 \times 3]$:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad A = \begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix}$$

• **Step 1:** Because $4 > 2$, we swap Row_0 with Row_1. After this row operation we have:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 4 & -4 \\ 2 & 1 & 5 \\ 1 & 3 & 1 \end{bmatrix}$$

• **Step 2:** We want to start creating zeroes on the first column. So we apply the following operation, $Row_1 - (\frac{1}{2})Row_0$. We record the multiplier $1/2$ in `L[1][0]`, and we compute the basic row operation on U:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 4 & -4 \\ 0 & -1 & 7 \\ 1 & 3 & 1 \end{bmatrix}$$

• **Step 3:** We continue to create zeroes on the first column , by applying: $Row_2 - \frac{1}{4}Row_0$. We record the multiplier $\frac{1}{4}$ in `L[2][0]`, and we compute the row operation on U:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{4} & 0 & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 4 & -4 \\ 0 & -1 & 7 \\ 0 & 2 & 2 \end{bmatrix}$$

• **Step 4:** We'ew finished with the first column, we skip to the next onw. Because $-1 < 2$ we swap Row_1 with Row_2. The idea is to always have the biggest pivot. P, L and U are affected by this swap:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 4 & -4 \\ 0 & 2 & 2 \\ 0 & -1 & 7 \end{bmatrix}$$

• **Step 5:** We want to create the last `0.0` on the seconds column. In this regard we apply $Row_2 - \left(-\frac{1}{2}\right) Row_1$:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{4} & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 4 & -4 \\ 0 & 2 & 2 \\ 0 & 0 & 8 \end{bmatrix}$$

• **Step 6:** We modify L by adding 1's on the first diagonal.

In conclusion, the $PA = LU$ factorization of A looks like:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & 1 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix} * \begin{bmatrix} 4 & 4 & -4 \\ 0 & 2 & 2 \\ 0 & 0 & 8 \end{bmatrix}$$

There's also a video with this example, link *here*

13.2 Code implementation

The best way to model the results of the `LU(P)` computation is to create a struct called `nml_mat_lup` containing references to all three resulting matrices: `L`, `U`, `P`.

```
typedef struct nml_mat_lup_s {
  nml_mat *L;
  nml_mat *U;
  nml_mat *P;
  unsigned int num_permutations;
} nml_mat_lup;
```

The property `num_permutations` records the number of row permutations we've done during the factorization process. This value it's useful when computing the determinant of the matrix, so it's better to track it now.

To reduce memory consumption, the two matrices `L` and `U` can be kept in single matrix `LU` that looks like this:

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{bmatrix}$$

In our implementation, for simplicity and readability, we will keep them separated.

Following the same recipe as for `nml_mat` we are going to write "constructor-like"/"destructor-like" methods for managing the memory allocation for a `nml_mat_lup` structure.

```
nml_mat_lup *nml_mat_lup_new(nml_mat *L, nml_mat *U, nml_mat *P, unsigned int num_permutatio
  nml_mat_lup *r = malloc(sizeof(*r));
  NP_CHECK(r);
  r->L = L;
  r->U = U;
  r->P = P;
  r->num_permutations = num_permutations;
  return r;
}

void nml_mat_lup_free(nml_mat_lup* lu) {
  nml_mat_free(lu->P);
  nml_mat_free(lu->L);
  nml_mat_free(lu->U);
  free(lu);
}
```

The code that is performing the factorization:

```
nml_mat_lup *nml_mat_lup_solve(nml_mat *m) {
  if (!m->is_square) {
    NML_FERROR(CANNOT_LU_MATRIX_SQUARE, m->num_rows, m-> num_cols);
    return NULL;
  }
  nml_mat *L = nml_mat_new(m->num_rows, m->num_rows);
  nml_mat *U = nml_mat_cp(m);
  nml_mat *P = nml_mat_eye(m->num_rows);

  int j,i, pivot;
  unsigned int num_permutations = 0;
  double mult;

  for(j = 0; j < U->num_cols; j++) {
    // Retrieves the row with the biggest element for column (j)
```

```
    pivot = _nml_mat_absmaxr(U, j);
    if (fabs(U->data[pivot][j]) < NML_MIN_COEF) {
      NML_ERROR(CANNOT_LU_MATRIX_DEGENERATE);
      return NULL;
    }
    if (pivot!=j) {
      // Pivots LU and P accordingly to the rule
      nml_mat_row_swap_r(U, j, pivot);
      nml_mat_row_swap_r(L, j, pivot);
      nml_mat_row_swap_r(P, j, pivot);
      // Keep the number of permutations to easily calculate the
      // determinant sign afterwards
      num_permutations++;
    }
    for(i = j+1; i < U->num_rows; i++) {
      mult = U->data[i][j] / U->data[j][j];
      // Building the U upper rows
      nml_mat_row_addrow_r(U, i, j, -mult);
      // Store the multiplier in L
      L->data[i][j] = mult;
    }
  }
  nml_mat_diag_set(L, 1.0);

  return nml_mat_lup_new(L, U, P, num_permutations);
}
```

14 Solving linear systems of equations

14.1 Forward substitution