"Numerl: Efficient Vector and Matrix Computation for Erlang "

Losseau, Tanguy

ABSTRACT

Though Erlang is a robust language offering an efficient concurrent environment, it comes short in not having a high performance for native computations being executed in a virtual machine (BEAM). Even so, Erlang offers the possibility to write Native Integrated Functions (NIFs), which provide an interface to functions written in C and in result extends the language with lower level operations. This master thesis goal is to write matrix operations for Erlang in NIFs. Considering a small set of matrix operations, its implementations in pure Erlang and NIFs are compared. The NIF versions are observed to run from five to seventeen times faster then their Erlang equivalents.

CITE THIS VERSION

UCLouvain

École polytechnique de Louvain

Numerl: Efficient Vector and Matrix Computation for Erlang

Authors : **Tanguy Losseau**
Supervisors : Peter Van Roy
Readers : **Kim Mens**, **Peter Van Roy**, **Peer Stritzinger**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Numerl: Efficient Vector and Matrix Computation for Erlang

UCL

Losseau Tanguy

05/12/2021

Abstract

Though Erlang is a robust language offering an efficient concurrent environment, it comes short in not having a high performance for native computations being executed in a virtual machine (BEAM). Even so, Erlang offers the possibility to write Native Integrated Functions (NIFs), which provide an interface to functions written in C and in result extends the language with lower level operations. This master thesis goal is to write matrix operations for Erlang in NIFs. Considering a small set of matrix operations, its implementations in pure Erlang and NIFs are compared. The NIF versions are observed to run from five to seventeen times faster than their Erlang equivalents.

Acknowledgements

This master thesis is my work; however, it was only possible thanks to the support of many.

Firstly, I would like to thank my supervisor Professor Peter Van Roy for the advice/help he provided throughout the whole work. His supportive feedback kept me on the right path to finish this master thesis.

I am equally thankful towards Peer Stritzinger, for taking at numerous occasions the time to discuss the wonders of the Erlang programming language.

I also thank Professor Kim Mens for accepting to be part of the jury; I hope you will enjoy this work.

Last and not least, I would like to thank my family, for the energy/encouragement you provided throughout the year.

Contents

Chapter 1

Introduction

1.1 Goal

The main goal of this master thesis is to obtain an easy to use Erlang library, offering the best performances possible for matrix operations. A complete library not being feasible in the time-frame of a single master thesis, my principle input aims towards a small but easily expandable platform: easy to install, work around and upon it, with freedom to add new functions.

1.2 Contributions

The resulting library **numerl** can be found at **github.com/tanguyl/numerl**. In order to be used, it first requires being built using the **make** command, resulting in the files **numerl.beam** and **numerl_NIF.so**. These two files need to be placed next to each other in order to link/use the resulting library. It should be noted that the building step requires the additional libraries **libgslcblas0** and **liblapacke-dev**.

This library supports operations for matrix addition, multiplication, transposition and inversion. It can use a small number of BLAS functions (**ddot, daxpy, dgemv, dgemm, dgesv**). More BLAS functions can be added easily.

Each function was tested using **eunit** to make sure it behaves as expected for various inputs; no memory leaks were found.

NIFs executions' time compared to their pure Erlang equivalent was:

- 12 times faster for addition of matrices
- 6 times faster for multiplication of a matrix by a number
- 12 times faster for multiplication of matrices
- 8 times faster for transposition of matrices
- 17 times faster for inversion of matrices

1.3 Roadmap

This paper is divided into three main sections.

Chapter 2 introduces Erlang, and proceeds with the study of a basic matrix library implementation written in this language.

Chapter 3 presents Native Integrated Functions (NIFs), and transcribes of the aforementioned library into NIFs. It continues by presenting the BLAS library, and its possible usage..

In Chapter 4, a performance comparison of each implemented function is proposed.

Chapter 2

Matrices in Erlang

The starting point of this thesis rests on a small library, 'mat.erl' (B). It serves as a backbone for the thesis "Sensor fusion at the extreme edge of an internet of things network" [15], and provides a small array of functions of matrices (addition, inversion...) allowing a Kalman Filter implementation.

This chapter is structured as such:

1. the presentation of Erlang's basics
2. the implementation of **mat.erl**

2.1 Introduction to Erlang

Only the bare minimum of the Erlang language, required to understand **mat.erl**'s implementation, is presented; a more thorough guide can be found in the book [11].

2.1.1 Setting up Erlang

The easiest way to install Erlang on Ubuntu is to run command:

```
sudo apt-get update
sudo apt-get install erlang-dev
```

Erlang's shell is started by entering the command **erl**, which can be quit using **q()**..

2.1.2 Variables and types

An Erlang program can be considered as a sequence of expressions terminated with either a period or a comma, followed by a whitespace. For example:

```
1> 1 + 3.
4
2> 1 + 5 / 2 .
3.5
3> X = 1 + 2,
    Y = X + 3.
8
```

As such, Erlang supports integers and doubles; both types are supported for arithmetic operations.

It is also a functional language, and thus has immutable variables.

```
1> One.
*1: variable 'One' is unbound
2> One = 1.
1
3> Two = One + 1.
2
4> One = 1.
1
5> One = 2.
** exception error: no match of right hand side value 2
6> 1 = 2.
** exception error: no match of right hand side value 2
```

Variables begin with an uppercase letter, and can be only bound once to a value. The = operator compares values and throws an error if it fails. Otherwise, it returns the compared value. If its left-hand term is an unbound variable, the right-hand value is bound to it and their comparison is successful.

A specific variable exists: _ . This variable never binds to a value, and it is always accepted by the = operator.

```
1> _ = 1.
1
2> _ = 2.
2
3> _.
 *1: variable '_' is unbound
```

Boolean algebra is supported via two special atoms: **true** and **false**. Operators such as **and**, **or**, **not** can be used on them:

```
1> true and not false.
true
2> not true.
false
```

Equality tests are done with ==, inequality with /=.
Orders are tested with ¡ (less than), ¿(greater than), =¡(less or equal), and finally ¿= (greater or equal).

```
1> 1 == 1.0.
true
2> 1 \= 0.
true
3> 1 =< 2.
true
```

2.1.3 Lists

Lists contain consecutive elements. A list of N elements can be noted as [**Elem1, Elem2, ..., ElemN**]. The Nth item is at position N, starting from position 1.

```
1> [1, 2, 3.0]
[1,2,3.0]
```

The first element of a list is its Head; the rest of the list is in its Tail.

```
1> Tail = [2,3].
[2,3]
2> List = [1|Tail].
[1,2,3]
3> [ListHead | ListTail ] = List.
[1,2,3]
4> ListHead.
1
5> ListTail.
[2,3]
```

The line 2 of the preceding code also shows how to add a new element to a matrix: **List** = [**NewHead** — **OldList**] creates the list [**NewHead, OldListElem1, OldListElem2,...**]. Line 3 shows how to extract Head and Tail from a list.
Lists can also be created using list comprehension:

```
1> [2*N || N <- [1,2,3]].
[2,4,6]
```

Values are extracted from the right side of the pipes (||), and are assembled in a new list after being transformed on the pipes' left side.

2.1.4 Functions of list

Erlang provides functions to interact with lists. The function **lists:nth(N, L)** returns the nth element of a list L; while the function **list:seq(Start, End)** generates a list of successive numbers from Start until End included.

```
1> lists:nth(1, [2, 4, 5]).
2
2> lists:seq(1,3).
[1,2,3]
```

2.2 Creating a matrix library

2.2.1 Matrices definition

A matrix is defined as list of list of numbers.

```
1> M1 = [[1,2.0, 3], [4,5,6.0]].
[[1,2.0, 3], [4,5,6.0]]
```

The matrices are defined in row major format; as such, M1 has two rows and three columns. Its element at position (1,1) is one; its first row is [1,2.0, 3] and its first column is [1,4].

Setting up an empty matrix is straightforward using list comprehension:

```
1> NCols = 2, NRows = 3.
3
2> Zero = [[0 || _ <- lists:seq(1,NCols)]
               || _ <- lists:seq(1, NRows)]].
```

By creating a list of rows, (**[... || _ <- lists:seq(1, NRows)**), for each of which we generate a list containing Ncols zeros (**[0 || _ <- lists:seq(1,NCols)]**), a matrix of dimensions NRows, NCols is obtained.

The identity matrix is a square matrix that has ones on its diagonal, and is filled with zeros for the rest of it.

```
1> N = 2.
2
2> [[ if RowId == ColId -> 1; true -> 0 end
           || ColId <- lists:seq(1, N)]
               || RowId <- lists:seq(1, N)].
```

First, a list of consecutive numbers identifying row positions (**RowId <- lists:seq(1,N)**) is constructed; secondly, for each row, a list identifying column is generated (**ColId <- lists:seq(1, N)**). Finally, each matrix element is associated with its row and column number; values can be generated accordingly (**if RowId == ColId -> 1; true -> 0 end**).

2.2.2 Accessing a matrix content

Creating a new matrix containing the Nth row of matrix M is just a matter of returning its Nth list within a list:

```
1> NthRow = [lists:nth(N,M)].
```

Getting a matrix's M content at coordinates (RowId, ColId) can be done as such:

```
1> lists:nth(ColId, lists:nth(RowId, M)).
```

After reaching the RowId'th row of M(**lists:nth(M,RowId)**), its ColId'th element is returned (**lists:nth(ColId, ...)**).

Finally, extracting into a new matrix the Nth column of matrix M can be done as such:

```
1> NthColumn = [[lists:nth(N, Row)] || Row <- M].
```

Iterating each row from M (**Row** $<-$ **M**), we associate it to a list containing its Nth element ([**lists:nth(N, Row)**]).

2.2.3 Matrix * Number

Once again, multiplying a matrix by a number can be done using a list comprehension.

```
1> N = 2,
    M = [[1,2],[3,4]],
    [[N*X || X <- Row] || Row <- M].
[[2,4],[6,8]].
```

We multiply by N each element X, extracted from the matrix's rows, which are themselves extracted from the matrix.

2.2.4 Operator + for matrices

The addition of two matrices is defined as their element-wise addition resulting, and results in a new matrix. The combination of two matrices can be done as such:

```
1                element_wise_op(Op, M1, M2) ->
2                    ZipRow = fun(Row1,Row2) -> lists:zipwith(Op,
                         Row1, Row2) end,
3                    lists:zipwith(ZipRow, M1, M2).
```

We zip each row of M1 and M2 in a new list (**lists:zipwith(ZipRow, M1, M2)**); the zip operation is defined by function Op (**fun(Row1,Row2) − > lists:zipwith(Op, Row1, Row2) end)**).

2.2.5 Matrix * Matrix

Depending whether a matrix is multiplied by a number or another matrix, the '*' operator must behave differently:

```
1                '*'(N, M) when is_number(N) ->
2                    [[N*X|| X <- Row] || Row <- M];
3                '*'(M1, M2) ->
4                    '*tr'(M1, tr(M2)).
```

The multiplication of a matrix by a number is exemplified above.
Multiplying two matrices A and B result into a new matrix, which elements are resulting in the dot product of each row of A by each column of B. However, as shown earlier, accessing consecutive elements of a row is much easier than accessing successive elements of a column, since we stay in the same list. Making the dot product of two rows is quicker than making the dot product of a row by a column. For this reason we use the multiply transpose (***'(A,B)**), which multiply A by the transpose of B, making the dot product of rows.

```
1                '*tr'(A, TB) ->
2                    [[lists:sum(lists:zipwith(fun erlang:'*'/2, Rb,
                         Ra))
3                      || Rb <- TB]
4                     || Ra <- A].
```

For each row in A (|| **Ra** < − **A**), we extract each row in the transpose of B (([**Rb** < − **B**]); and combine them using the sum of their product (

(**lists:sum(lists:zipwith(fun erlang:'*'/2, Rb, Ra)**), aka their dot product..

2.2.6 Inversion of matrices

The Matrix inversion operation is implemented as follows:

```
1  inv(M) ->
2      N = length(M),
3      A = lists:zipwith(fun lists:append/2, M, eye(N)),
4      Gj = gauss_jordan(A, N, 0, 1),
5      [lists:nthtail(N, Row) || Row <- Gj].
```

The **inv** function takes as parameter a square matrix **M** of dimensions **N**x**N**. Creating matrix **A** resulting from **M**'s right-sided juxtaposition to an identity matrix ([**lists:zipwith(fun lists:append/2, M, eye(N))**, a matrix **Gj** is obtained using the function **gauss_jordan**. The inverse of **M** is obtained by extracting **Gj**'s right-side **N**x**N** sub-matrix([**lists:nthtail(N, Row) || Row ← Gj**]).

The **gauss_jordan** function was implemented using Gauss Jordan's algorithm ([7]). Operating on matrix **A** it can be summarised as such:

```
1  A   /*Input matrix, resulting in the juxtaposition of a square
       matrix M with an identity matrix I, each of size NxN
2  R=0 /*Precedent row iterated on*/
3  J=1 /*Current column iterated on*/
4
5  while J < N:
6      K     = max(abs(A[R+1->N, J]))
7      if K is 0:
8          increment R
9      else:
10         Pivot = A[K,J]
11         swap rows R+1 and K of A
12         divide row R+1 of A by Pivot
13         for all rows I from R+1 to N:
14             f := A[i,k] / A[k,k]
15             for all elements A at line i:
16                 E(i,j) -= E(i,j)*f
17         increment J,R
```

This algorithm iterates on the columns and rows of **A**.

Each iteration can be broken down in three steps:

1. a pivot value is searched (lines 6 and 10)
2. the pivot's row is swapped with the current row (line 11)
3. the pivot's row is normalised, the next ones are reduced (lines 12 to 16)

Each described step iterates on lists: they can be written as their own standalone functions.

Finding a pivot's value is done through the **pivot** function:

```
1 %% find the gauss jordan pivot of a column
2 pivot([], _, Pivot) ->
3     Pivot;
4 pivot([[H]|T], I, {_, V}) when abs(H) >= abs(V) ->
5     pivot(T, I+1, {I, H});
6 pivot([_|T], I, Pivot) ->
7     pivot(T, I+1, Pivot).
```

Listing 2.1: Finding a pivot value

This is a terminal-recursive function, selecting the best value **Pivot** from an input list **L**.
The **Pivot** is the **I**'th value **H** of **L**, such that the absolute value of **H** is the largest of **L**.

Swapping two lines of a matrix is equivalent to swapping two elements of a list **List** at positions **A** and **B** :

```
1 %% swap two indexes of a list
2 %% taken from https://stackoverflow.com/a/64024907
3 swap(A, A, List) ->
4     List;
5 swap(A, B, List) ->
6     {P1, P2} = {min(A,B), max(A,B)},
7     {L1, [Elem1 | T1]} = lists:split(P1-1, List),
8     {L2, [Elem2 | L3]} = lists:split(P2-P1-1, T1),
9     lists:append([L1, [Elem2], L2, [Elem1], L3]).
```

Swapping the same two lines requires no modifications. However, swapping two elements at ordered positions **P1**, **P2** requires extracting the list's elements **Element1**, **Element2** of matching position, and splitting the input list into three lists containing **List**'s elements:

- preceding position **P1** (**L1**)
- between positions **P1** and **P2** (**L2**)
- following position **P2** (**L3**).

Two calls to **lists:split** are required. The list can then be reassembled in a new order with a call to **list:append**, swapping the requested elements: **lists:append([L1, [Elem2], L2, [Elem1], L3])**.

Finally, normalisation/reduction of the rows are done with a call to the tail recursive function **gauss_jordan_aux(A, {R, J}, L, I, Acc)**:

```
1 gauss_jordan_aux([], _, _, _, Acc) ->
2     lists:reverse(Acc);
3 gauss_jordan_aux([_|Rows], {I, J}, L, I, Acc)->
4     gauss_jordan_aux(Rows, {I, J}, L, I+1, [L|Acc]);
5 gauss_jordan_aux([Row|Rows], {R, J}, L, I, Acc) ->
6     F = lists:nth(J, Row),
7     NewRow = lists:zipwith(fun(A, B) -> A-F*B end, Row, L),
8     gauss_jordan_aux(Rows, {R, J}, L, I+1, [NewRow|Acc]).
```

Listing 2.2: Third step of Gauss Jordan

It has the following arguments:

- **A**: the matrix that needs to be transformed.
- {**I**,**J**}: **I** is the pivot's row, **J** is the pivot's column.
- **L**: the **I**'th row normalized by pivot's value.
- **I**: the row currently modified.
- **Acc**: the resulting matrix accumulator.

For $\mathbf{I} < \mathbf{R}$, the row is already constructed: **F = lists:nth(J, Row)** is null. Hence, the row is unmodified (**NewRow = lists:zipwith(fun(A, B)**

→ **A end, Row, L)**) and accumulated as is ([**NewRow**‖**Acc**]).
If **I** == **R**, the pivot's line is replaced with it's normalised version **L**.
If **I** > **R**, the row **I** is reduced.
Finally, when **A** is fully consumed, **Acc**'s reverse is returned (since this list was built in a tail-recursion).

Finally, we obtain the following **gauss_jordan** implementation:

```
1  %% Gauss-Jordan method from
2  %% https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-
       Jordan#Pseudocode
3  gauss_jordan(A, N, _, J) when J > N ->
4      A;
5  gauss_jordan(A, N, R, J) ->
6      case pivot(col(J, lists:nthtail(R, A)), R+1, {0, 0}) of
7          {_, 0} ->
8              gauss_jordan(A, N, R, J+1);
9          {K, Pivot} ->
10             A2 = swap(K, R+1, A),
11             [Row] = row(R+1, A2),
12             Norm = lists:map(fun(E) -> E/Pivot end, Row),
13             A3 = gauss_jordan_aux(A2, {R+1, J}, Norm, 1, []),
14             gauss_jordan(A3, N, R+1, J+1)
15     end.
```

Listing 2.3: **gauss_jordan** function

If a pivot is found, matrix **A2** is constructed by swapping the pivot's row to the current row position.
Before the **gauss_jordan_aux** can be called, the normalised pivot's row needs to be constructed: after extracting it with **row(R+1, A2)**, it is transformed through **lists:map(fun(E) → E/Pivot end, Row)**.

2.2.7 Transpose of a matrix

The transpose of a matrix A is the matrix B such that: **A(i,j) = B(j,i)**: in the transposed matrices, columns and rows mirror each other. This can be performed as such in Erlang:

```
1              tr(M) ->
2                  tr(M, []).
3
4              %% transpose matrix with accumulator
```

```
5               tr([[]|_], Rows) ->
6                   lists:reverse(Rows);
7               tr(M, Rows) ->
8                   {Row, Cols} = tr(M, [], []),
9                   tr(Cols, [Row|Rows]).
10
11              %% transpose the first row of a matrix with
                    accumulators
12              tr([], Col, Cols) ->
13                  {lists:reverse(Col), lists:reverse(Cols)};
14              tr([[H|T]|Rows], Col, Cols) ->
15                  tr(Rows, [H|Col], [T|Cols]).
```

This algorithm extracts the first element of each row into a **Col** list, saving all remaining row elements in the **Cols** list.
Once the iteration is finished, the resulting lists need to be reversed (as seen above), since they were built through a terminal-recursive function.
When the input matrix is exhausted, **Rows** is also reversed for the aforementioned reason in order to obtain the matrix's transpose.

2.3 Summary

This chapter introduces Erlang and shows how to implement a basic matrix library in Erlang with addition, multiplication, inverse, and transpose functions.
Matrices are represented as lists of lists, which are iterated/created by the mentioned functions using:

- list comprehension.
- built-in Erlang functions (**lists:nth**,**lists:split**,...).
- tail recursive functions.

In the next chapter, we present how these functions can be rewritten into C and integrated in Erlang.

Chapter 3

Matrices in C

The Erlang code is compiled to BEAM instructions, that are executed in a virtual machine, resulting in an overhead.
Erlang offers Native Integrated Functions (NIFs) allowing functions to be written in C. They are executed natively, and can be called as any normal Erlang function.

In this chapter, the following themes are presented:

1. speed performance of C and Erlang
2. issues of NIF usage
3. a matrix library written as NIFs
4. testing of the previously built library
5. the possibility of extending this library with BLAS.

3.1 Comparing Erlang and C

As a preliminary step, I propose to compare the speed difference between C and Erlang.

The goal of this section is not to make an exhaustive comparison of Erlang and C for matrix operations; rather it is only to show that there exists

a serious problem that can be solved by the techniques of this master's thesis.

To observe this speed difference, I implemented a max function in both languages, returning the maximum of a List (Erlang) or an array (C).

```
1  lmax([],M)->M;
2  lmax([H|T],M)-> lmax(T,max(H,M)).
3
4  kmax(L,_)->
5          lmax(L,0), lmax(L,0), lmax(L,0),
6          % a couple more lmax...
7          listMax(L,_)->
8          lmax(L,0).
```

```
1  int max(int* a, int len){
2   int m = 0;
3
4   for(int i = 0; i<len; i++){
5    m = m>i? m:i;
6   }
7
8   return m;
9  }
10
11 void kmax(int*a, int len){
12  max(a,len); max(a,len);
13  //a couple more max...
14 }
```

Each implementation operates fairly quickly, at less than 1 micro second for a list/array size beneath 100. To obtain a more precise measure, each function is called multiple times in a **kmax** function, respectively 10_000 for the Erlang and 5_000 times for the C version. This due because Erlang's **timer:tc** only measures time at the micro second level, whilst C is more precise.

The benchmarks can be found in the annex A.

As it can be seen in figure 3.1, Erlang is around four times slower than C for this basic iteration.

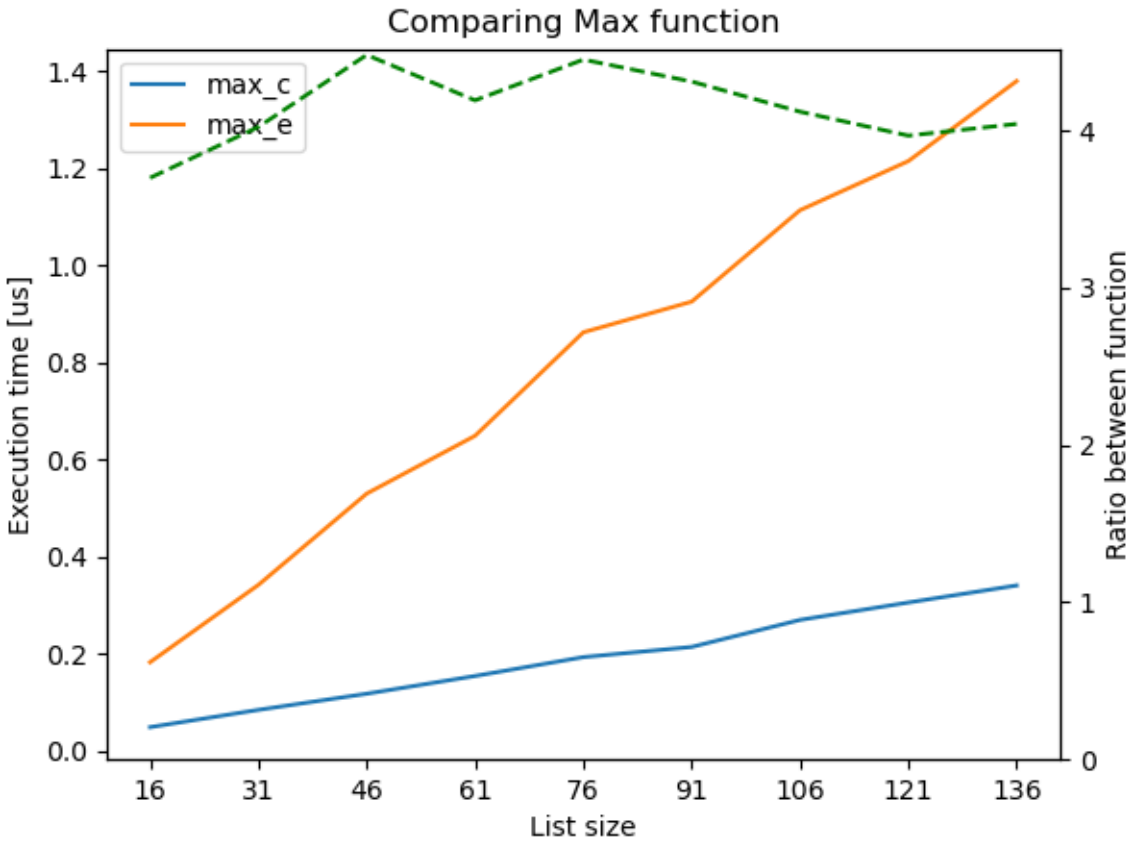

Figure 3.1: Finding maximum value within a N sized list

3.2 Native Integrated Functions

For performance gain, some Erlang functions are not ran in the BEAM virtual machine, but executed natively. They are the Built In Functions (BIF); such is the case of **lists:nth** presented in section 2.2.2. However, this language also leaves the freedom to declare a function and implement it in C: these are Natively Integrated Functions (NIF). They can then be loaded dynamically when required (see section 3.6 for more details). Nevertheless, making a C library interact with Erlang presents a number of challenges! After presenting the main risks that go together with Nifs (and how to avoid them), I will discuss how Nifs can interact with Erlang's datatype.

3.2.1 NIF implementation issues

When referring to the official Erlang manual page for NIF ([**NIF**]), the following message is written in red: "Use this functionality with extreme care". Since Nifs don't execute in the BEAM, they don't benefit from Erlang's safety... A NIF mishandling/crash causes the entire BEAM virtual machine to crash! Extreme care is advised (looking at you, **sigsegv** faults).

One of Erlang's main selling point is its 'soft real time' feature: any process waiting for a sent message will receive it within a reasonable delay. This is performed via a system of schedulers (one per system thread), each associated to its own queue of processes. Those queues are balanced at run time and active processes are swhiched upon frequently using preemptive scheduling ([6]). To do so, each Erlang process can be paused/resumed; this is not the case with Nifs. A native C function running for too long will cause issues such as bad load balance between schedulers.

To avoid this issue, Nifs should not run for more than 1ms ([**NIF**]).

If this cannot be avoided, the easiest solution is to use a dirty scheduler. Such scheduler doesn't have time limitation and will not mess up the overall load balance. However, launching a dirty scheduler requires halting a 'sane' one, with the cost associated.

A third option is to break down the NIF into multiple smaller functions. This can be done both at the NIF level (NIF functions scheduling other Nifs

to continue the work), or from the Erlang's side (calling multiple Nifs).

This is quite a complex issue. An implementation using the latter idea, executing the matrices functions over chunks of submatrices will be investigated by other authors in a future thesis [5].

3.2.2 Dealing with errors

A very important rule is that a NIF should never crash (unless you want to see Erlang burn). Hence, the need to flag errors and unexpected situations.

A NIF can raise a **badarg** error with **enif_make_badarg** if it receives an invalid argument, halting the caller's execution.

Another option is to raise an exception using **enif_raise_exception**. This leaves the possibility to create an atom containing additional information, which can be retrieved on the Erlangs side.

3.2.3 NIF definition

Using a NIF function requires preparation.

An Erlang module can make use of Nifs this way:

```
-module(numerl).
-export([matrix/1]).

%  This function is executed when the module is loaded.
init()->
    erlang:load_nif("./numerl_nif", 0).

matrix(_)->
    nif_not_loaded.
```

The **numerl** module loads the dynamic library **numerl_nif** when itself is loaded. The function matrix is defined, and by default returns the atom **NIF_not_loaded**. Once initialisation is done, this function is overwritten

by the one defined in **numerl_nif.so**.

This shared library is written in a C file and must be written in the following key structure:

```
//This include Erlang library for writing Nifs
#include <erl_nif.h>

//A NIF declaration
ERL_NIF_TERM nif_matrix(ErlNifEnv* env, int argc, ERL_NIF_TERM argv[]){
    //some code..
    return enif_make_atom(env, "a_matrix");
}


//This array is used to make an arity based correspondence
//between an Erlang function, and its NIF version.
ErlNifFunc nif_funcs[] = {
    {"matrix", 1, nif_matrix}
}

//Macro defining the NIF
ERL_NIF_INIT(numerl, nif_functs, NULL, NULL, NULL, NULL)
```

The first step is to include **erl_nif.h** header, which defines key functions and macro.

Now, NIF functions can be declared. They all share the same return type (**ERL_NIF_TERM**), an opaque type representing all Erlangs' types. They also have the same tree arguments:

1. **ErlNifEnv* env**, which is used for memory management;
2. **int argc**, the number of arguments sent to this function by Erlang;
3. **ERL_NIF_TERM argv**[], an array of arguments sent by Erlang.

The NIF **NIF_matrix** returns the atom **a_matrix**. The real implementation of this function will be presented in section 3.3.1.

Each NIF needs to be matched to an Erlang function; this correspondence is established in the **NIF_funcs** array.

A call to the macro **ERL_NIF_INIT** sets the library up.

Finally, when loading the **numerl.beam** in an Erlang shell, the **numerl_nif.so** has to be placed next to it.

For additional information on any part of this structure, more details can be found at [**NIF**].

3.2.4 Nifs overhead

Erlang variables come with a lot of constraints: they are immutable and their life span is controlled by a Garbage Collector. This is a choice of conception, which allows Erlang to be robust in highly concurrent programs.

In opposition, C gives full memory control: memory is allocated/freed at will and variables can be reassigned.

This is quite a clash of perspective! To make things safer, all Erlang variables are accessed from C via the opaque type **ERL_NIF_TERM**. The Erlang API provides functions to retrieve their type and extract their content to their C representation, which can then be modified.

Memory wise, **ERL_NIF_TERM**s' life-cycle is controlled by Erlang's garbage collector. However, to control memory within the NIF itself **enif_alloc** and **enif_free** are advised, instead of **malloc** and **free**. This gives Erlang's virtual machine more memory control.

Consequently, any Erlang variable sent to a NIF requires extraction from an **ERL_NIF_TERM**: any value returned by a NIF requires to be translated into such a term, whilst memory used within the NIF can be manually handled.

Translating Erlangs' and C's datatype results in an overhead, which can be observed quite easily. Let's consider three different implementations of a **max** function, which returns the biggest element of a list/array of integers.

One implemented using an Erlang tail recursive function (3.1), a second one written as a NIF iterating over an Erlang list (3.2), and finally a third one written in C using arrays (3.3).

The Erlang version could have been written using the **lists:max** function. Being a highly optimised BIF executing natively, it would beat the purpose of showing the performance of a 'pure' Erlang code. Instead, I propose a tail recursive function iterating over its input list, which is the exact same algorithm implemented in the NIF (minus translation).
The NIF version has to iterate element by element in its input list, translating each **ERL_NIF_TERM** into an **int**.
Finally, the C version simply iterates over the N first elements of its input array.

```
1  Max = fun F([],M)->M; F([H|T],M)-> F(T,max(H,M))
       end.
```

Listing 3.1: Erlang version

```
1  ERL_NIF_TERM nif_max_list(ErlNifEnv * env, int
       argc, const ERL_NIF_TERM argv[]){
2      ERL_NIF_TERM head = argv[0];
3
4      ERL_NIF_TERM elem;
5      ERL_NIF_TERM max_e;
6
7      int current, max = 0;
8
9      for(int i = 0; enif_get_list_cell(env, head,
           &elem, &head);){
10         enif_get_int(env, elem, &current);
11         if(max < current){
12             max = current;
13             max_e = elem;
14         }
15     }
16
17     return max_e;
18 }
```

Listing 3.2: NIF version

```
1  int max(int* a, int len){
```

```
2                   int m = 0;
3
4                   for(int i = 0; i<len; i++){
5                    m = m>i? m:i;
6                   }
7
8                   return m;
9                  }
```

Listing 3.3: C version

The benchmark (see annex A) gave the following execution time for an input array of size a 1000:

Implementation	Erlang (3.1)	C(3.3	NIF(3.2)
Execution time(μs)	9.29	2.51	7.37

Though the NIF version still outperforms its pure Erlang counterpart, it lags significantly behind the implementation written in C. The overheads produced by using a NIF, namely the cost of swhiching to a native function and the cost of argument translation, can add up significantly. In the precedent example, those delays waste more time than the standalone C algorithm.

3.2.5 Memory blocks

Not much can be done to reduce the cost of calling a NIF. However, Erlang gives Nifs two types of containers to store their custom structures in **Binaries** and **Resources**. With them, a NIF can send back and forth to Erlang chunks of data, removing the necessity of translating each value.

The first container is Binary, a contiguous block of memory. On the C side, the binary struct is defined as such:

```
typedef struct {
    void* data;
    int size;
} ErlNifBinary;
```

Binary can be allocated using **enif_alloc_binary** ([**NIF**]), and released through **enif_release_binary**. Alternatively, it can be translated to an **ERL_NIF_TERM** using **enif_make_binary**; it will later be garbage collected by Erlang once needed.

They can also be derived from an **ERL_NIF_TERM**; however, the resulting binary cannot be modified.

Binaries can store any C struct and be sent to/red from Erlang. Setting the **data** field to a Matrix's content would allow a NIF to access a complete matrix, reading a single **ERL_NIF_TERM**, instead of reading each value from a list of lists.

The other option is to use a **Resource**, a pointer to a C structure which can be modified.
Before it can be used, each **Resource** type needs to be defined via a call to **enif_init_resource_type**, assigning to it a destructor function and a name. They can be created using **enif_alloc_resource**; their memory cycle is managed by the garbage collector via a reference counting that can be increased/decreased on the NIF side via calls to **enif_keep_resource** and **enif_release_resource**.

3.3 Matrix library in C

Having clarified up how NIF functions, I can now present the next part of this research: rewriting **mat.erl** in a NIF. You can find in chapter 2 their equivalent Erlang implementation.

3.3.1 Matrix implementation

Matrices could be either represented by a **Binary** or a **Resource**. However, the latter requires risky manual memory management; to stay within safer boundaries, **numerl** is using **Binary** type to represent its matrices.

```
typedef struct{
    int n_rows;
    int n_cols;

    //Content of the matrix, in row major format.
    double* content;
```

```
    //Erlang binary containing the matrix.
    ErlNifBinary bin;
} Matrix;
```

The **n_rows** and **n_cols** fields specify the matrix dimension; content is an array of double, containing the matrix data. The **bin** field is used to store the **ErlNifBinary** representation of the matrix; this is done as a commodity. Functions related to the memory management of matrices (reading from an Erlang term, allocation...) require an **ErlNifBinary** term, and operations on matrices need a Matrix; hence, they are grouped together. I provide the following functions for matrix handling:

- **matrix_alloc**: allocates a binary and builds/returns the associated Matrix struct.
- **matrix_dup**: duplicates a matrix. Multiple functions need to modify a Matrix given as input; duplicating those into a new modifiable matrix is often required.
- **matrix_free**: matrices not returned to Erlang have to be released.
- **matrix_to_erl**: create an **ERL_NIF_TERM** representation of a Matrix, which can be sent back to Erlang.

This latter associates in a record the matrix's binary to the atom **matrix**; Numerl expects such an association when decoding an input Matrix. This is done for safety; it ensures that decoded binaries are indeed Matrices (unless a user sends such a self made record with an invalid binary). This is mandatory, for reading an invalid binary as a Matrix would likely produce a segmentation fault, and a BEAM crash.

At listing (3.4) you can find the NIF presented at (3.2), rewritten using the Matrix type.

```
1 ERL_NIF_TERM nif_max_matrix(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
2     Matrix L;
3     enif_get_matrix(env, argv[0], &L);
4     int max = 0;
5
```

```
6       for(int i = 0; i<L.n_cols * L.n_rows; i++){
7           max = L.content[i] > max? L.content[i]:max;
8       }
9
10      return enif_make_int(env, max);
11  }
```

Listing 3.4: NIF Matrix version

With an input vector of size 1000, this version ran in 5μs. It is an improvement over the NIF version iterating through a List, which required 7.37 μs!

3.3.2 Some helper functions

All the NIF functions I had to write for this thesis required primarily translating all Erlang terms received into their C representation. I streamlined this operation with the following functions: **int enif_get(ErlNifEnv* env, const ERL_NIF_TERM* erl_terms, const char* format, ...)**.

It has for first argument an **env** field, which contains the **ErlNifEnv** responsible of translated **ERL_NIF_TERMs**.
Follows an array of **ERL_NIF_TERM**s to be translated, then a string defining each expected term type, and finally pointers to C struct in which they are rewritten.

For example, a NIF expecting to receive as arguments a number, two matrices another number and finally another matrix, can be started as follows:

```
1  Matrix A,x,y;
2  double alpha, beta;
3
4  if(!enif_get(env, argv, "nmmnm", &alpha, &A, &x, &beta, &y)){
5      enif_make_badarg(env);
6  }
```

Each number (**int** or **double**) correspond to the char **n** (number), each matrix is denoted with **m** (matrix).
The **enif_get** function returns whether all received Erlang terms were of the expected type and were successfully translated to their C representation.

3.3.3 NIF zeros

This NIF generates a matrix filled with zeros. Taking as argument integers indicating a matrix dimension (number of rows, number of columns), it returns a matrix of the indicated size.

```
1  ERL_NIF_TERM nif_zero(ErlNifEnv *env, int argc, const
       ERL_NIF_TERM argv[]){
2      int m,n;
3      if(!enif_get(env, argv, "ii", &m, &n))
4          return enif_make_badarg(env);
5
6      Matrix a = matrix_alloc(m,n);
7      memset(a.content, 0, sizeof(double)*m*n);
8      return matrix_to_erl(env, a);
9  }
```

Listing 3.5: NIF generating empty matrix

After allocating a matrix of requested size, its content is set to zero using **memset**.

3.3.4 NIF '+'

This NIF sums two matrices element wise into a new one.

```
1  ERL_NIF_TERM nif_plus(ErlNifEnv *env, int argc, const
       ERL_NIF_TERM argv[]){
2
3      Matrix a,b;
4      if(!enif_get(env, argv, "mm", &a, &b))
5          return enif_make_badarg(env);
6
7      if ((a.n_cols != b.n_cols || a.n_rows != b.n_rows)){
8          return enif_make_badarg(env);
9      }
10
11     Matrix result = matrix_alloc(a.n_rows, a.n_cols);
12
13     for(int i = 0; i < a.n_cols*a.n_rows; i++){
14             result.content[i] = a.content[i] + b.content[i];
15     }
16
17     return matrix_to_erl(env, result);
18 }
```

Listing 3.6: Addition of matrices in a NIF

After translating the two matrix terms (and checking their dimension compatibility), this function allocates a Matrix which will contain the result. We then loop for each row/column. The matrices are stored in a row major format; the rows are located next to one another. Hence, iterating their entire content access them element by element, row by row. This exploits spacial locality ([14]): when a memory space is accessed, it is (as well as the contiguous blocks) stored in a cache for faster access (until another block replaces it).

3.3.5 NIF tr

This NIF transposes a matrix; it changes the content ordering so that rows become columns (and vice versa).

```
1  Matrix tr(Matrix a){
2      Matrix result = matrix_alloc(a.n_cols ,a.n_rows);
3
4      for(int j = 0; j < a.n_rows; j++){
5          for(int i = 0; i < a.n_cols; i++){
6              result.content[i*result.n_cols+j] = a.content[j*a.
                   n_cols+i];
7          }
8      }
9      return result;
10 }
11
12 //@arg0: Matrix.
13 ERL_NIF_TERM nif_tr(ErlNifEnv *env, int argc, const ERL_NIF_TERM
       argv[]){
14
15     Matrix a;
16     if(!enif_get_matrix(env, argv[0], &a))
17         return enif_make_badarg(env);
18
19     return matrix_to_erl(env, tr(a));
20 }
```

Listing 3.7: NIF generating the transpose of a matrix

Once again, to make use of spacial locality, we access successive elements of the input matrix **a**.

3.3.6 NIF '*'

A multiplication can be done either between a number and a matrix, or between two matrices; each of which is done by a different NIF. The Erlang module picks the correct version as such:

```
1  '*'(A,B) when is_number(A) -> '*_num'(A,B);
2  '*'(A,B) -> '*_matrix'(A,B).
3
4  '*_num'(_,_)->
5      nif_not_loaded.
6
7  '*_matrix'(_, _)->
8      nif_not_loaded.
```

Listing 3.8: Module picking NIF to call

Calling **numerl:'*'(2, Matrix)** would result in a call to **'*_num'**. Instead, using **numerl:'*'(Matrix_1, Matrix_2)** would result with ***_matrix**.

```
1  //arg 0: double or int
2  //arg 1: Matrix
3  //@return the result of multiplying each matrix element by arg
       0.
4  ERL_NIF_TERM nif_mult_num(ErlNifEnv *env, int argc, const
       ERL_NIF_TERM argv[]){
5
6      double a;
7      Matrix b;
8      if(!enif_get(env, argv, "nm", &a, &b))
9          return enif_make_badarg(env);
10
11     Matrix c = matrix_alloc(b.n_rows, b.n_cols);
12
13     for(int i = 0; i<b.n_cols*b.n_rows; i++){
14         c.content[i] = a * b.content[i];
15     }
16
17     return matrix_to_erl(env, c);
18 }
```

Listing 3.9: NIF multiplying a matrix by a number

This NIF is quite similar to the **NIF_plus** (3.3.4) seen earlier on: we iterate over the rows of input matrix b to build a new matrix c, multiplying each

value of b by the constant a.

```
1
2  ERL_NIF_TERM _nif_mult_matrix(ErlNifEnv *env, int argc, const
       ERL_NIF_TERM argv[]){
3
4      Matrix a,b;
5      if(!enif_get(env, argv, "mm", &a, &b))
6          return enif_make_badarg(env);
7
8      int n_rows = a.n_rows;
9      int n_cols = b.n_cols;
10
11     if(a.n_cols != b.n_rows)
12         return atom_nok;
13
14     Matrix result = matrix_alloc(n_rows, n_cols);
15     memset(result.content, 0.0, n_rows*n_cols * sizeof(double));
16
17     for(int i = 0; i < n_rows; i++){
18         for(int k = 0; k<a.n_cols; k++){
19             double val = a.content[i*a.n_cols + k];
20             for(int j = 0; j < n_cols; j++){
21                 result.content[j+i*result.n_cols] += val * b.
                       content[k*b.n_cols + j];
22             }
23         }
24     }
25
26     return matrix_to_erl(env, result);
27 }
```

After reading input matrices **a**, **b** and checking their dimension compatibility for a multiplication, a new empty matrix C is created.
Once again, we want to benefit from caching. To do so, we iterate the multiplied matrices through their rows.

3.3.7 NIF inv

This NIF implements once again the Gauss Jordan algorithm discussed at 2.2.6. As a reminder, this algorithm proceeds multiple iteration over the following steps: find a pivot value/row, swap it to the current row, normalise

the pivot's row and normalise the remaining ones.

Finding the **r** pivot's value and row within a matrix **gj** is done as such:

```
1  int pivot_row = -1;
2  for(int cur_row=r; cur_row<a.n_rows; cur_row++){
3      if(pivot_row<0 || fabs(gj[cur_row*n_cols + j]) > fabs(gj[
           pivot_row*n_cols+j])){
4          pivot_row = cur_row;
5      }
6  }
7  double pivot_value = gj[pivot_row*n_cols+j];
```

The first column of the current submatrix is iterated, and the row starting with the highest absolute value is selected.

Swapping two rows is done as such:

```
1  for(int i = 0; i<n_cols; i++){
2          double cpy = gj[pivot_row*n_cols+i];
3          gj[pivot_row*n_cols+i]= gj[r*n_cols+i];
4          gj[r*n_cols+i] = cpy;
5  }
```

Instead of the Erlang version which required the creation of five lists in order to swap two columns, the C version can do the transformation 'in place', needing only a variable **cpy**.

Finally, the rows normalisation/reduction is done as such:

```
1  //Normalise the current pivot row
2  for(int cur_col=0; cur_col<n_cols; cur_col++){
3      gj[cur_col+pivot_row*n_cols] /= pivot_value;
4  }
5
6  //Reduce remaining rows
7  gj[pivot_row*n_cols + j] = 1.0; //make up for rounding errors
8      for(int i=0; i<a.n_rows; i++){
9      if(i!=r){
10         double factor = gj[i*n_cols+j];
11         for(int col=0; col<n_cols; col++){
12             gj[col+i*n_cols] -= gj[col+r*n_cols]*factor;
13         }
```

```
14          gj[i*n_cols+j] = 0.0;     //make up for rounding errors
15      }
16  }
```

Each operation is done 'in place' unlike the Erlang version, which had to create new lists element by element (and reverse them). Lastly, you can find below the complete NIF function for this operation:

```
1
2   ERL_NIF_TERM nif_inv(ErlNifEnv *env, int argc, const
        ERL_NIF_TERM argv[]){
3
4       Matrix a;
5       if(!enif_get_matrix(env, argv[0], &a))
6           return enif_make_badarg(env);
7
8       if(a.n_cols != a.n_rows){
9           return atom_nok;
10      }
11      int n_cols = 2*a.n_cols;
12
13      double* gj = (double*) enif_alloc(n_cols*a.n_rows*sizeof(
            double));
14      for(int i=0; i<a.n_rows; i++){
15          memcpy(gj+i*n_cols, a.content+i*a.n_cols, sizeof(double)
                *a.n_cols);
16          memset(gj+i*n_cols + a.n_cols, 0, sizeof(double)*a.
                n_cols);
17          gj[i*n_cols + a.n_cols + i] = 1.0;
18      }
19
20      //Elimination de Gauss Jordan:
21      //https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-
            Jordan
22
23      //Row of last found pivot
24      int r = -1;
25      //j for all indexes of column
26      for(int j=0; j<a.n_cols; j++){
27
28          //Find the row of the maximum in column j
29          int pivot_row = -1;
30          for(int cur_row=r; cur_row<a.n_rows; cur_row++){
31              if(pivot_row<0 || fabs(gj[cur_row*n_cols + j]) >
                    fabs(gj[pivot_row*n_cols+j])){
32                  pivot_row = cur_row;
```

```
33              }
34          }
35          double pivot_value = gj[pivot_row*n_cols+j];
36
37          if(pivot_value != 0){
38              r++;
39              for(int cur_col=0; cur_col<n_cols; cur_col++){
40                  gj[cur_col+pivot_row*n_cols] /= pivot_value;
41              }
42              gj[pivot_row*n_cols + j] = 1.0; //make up for
                    rounding errors
43
44              //Do we need to swap?
45              if(pivot_row != r){
46                  for(int i = 0; i<n_cols; i++){
47                      double cpy = gj[pivot_row*n_cols+i];
48                      gj[pivot_row*n_cols+i]= gj[r*n_cols+i];
49                      gj[r*n_cols+i] = cpy;
50                  }
51              }
52
53              //We can simplify all the rows
54              for(int i=0; i<a.n_rows; i++){
55                  if(i!=r){
56                      double factor = gj[i*n_cols+j];
57                      for(int col=0; col<n_cols; col++){
58                          gj[col+i*n_cols] -= gj[col+r*n_cols]*
                                factor;
59                      }
60                      gj[i*n_cols+j] = 0.0;     //make up for
                            rounding errors
61                  }
62              }
63          }
64
65      }
66
67      Matrix inv = matrix_alloc(a.n_rows, a.n_cols);
68      for(int l=0; l<inv.n_rows; l++){
69          int line_start = l*n_cols + a.n_cols;
70          memcpy(inv.content + inv.n_cols*l, gj + line_start,
                sizeof(double)*inv.n_cols);
71      }
72
73      enif_free(gj);
```

```
74      return matrix_to_erl(env, inv);
75  }
```

3.4 Tests

3.4.1 Rounding errors

Doubles cannot represent infinite-precision values; as such, doing successive operations on them can lead up to slight value error. This is particularly true for operations such as matrix multiplication and inversion. For this reason, matrices should not be compared using the default = operator, but using **numerl:'=='**. It is implemented as follows:

```
1  //Equal all doubles
2  //Compares whether all doubles are approximately the same.
3  int equal_ad(double* a, double* b, int size){
4      for(int i = 0; i<size; i++){
5          if(fabs(a[i] - b[i])> 1e-6)
6              return 0;
7      }
8      return 1;
9  }
10
11 ERL_NIF_TERM nif_eq(ErlNifEnv *env, int argc, const ERL_NIF_TERM
       argv[]){
12     Matrix a,b;
13     if(!enif_get(env, argv, "mm", &a, &b))
14         return enif_make_badarg(env);
15
16     //Compare number of columns and rows
17     if((a.n_cols != b.n_cols || a.n_rows != b.n_rows))
18         return atom_false;
19
20     //Compare content of arrays
21     if(!equal_ad(a.content, b.content, a.n_cols*a.n_rows))
22         return atom_false;
23
24     return atom_true;
25 }
```

Input matrices are compared value by value using **equal_ad**, and considered the same if their delta is less than 10^{-6}.

3.4.2 EUnit tests

My code was tested using E-unit tests. For each function, I verified it behaved correctly for a set of inputs. Each test function name must end with **_test** and must cause a crash in case of a failure. Using the = operator between the expected result and the one obtained, the following example can be written:

```
1  mult_matrix_test() ->
2      CM0 = numerl:eye(2),
3      CM1 = numerl:matrix([[1.0, 2.0], [3.0, 4.0]]),
4      CM3 = numerl:matrix([[1.0], [2.0]]),
5      CM5 = numerl:matrix([[5.0], [11.0]]),
6      CM4 = numerl:matrix([[1.0, 2.0]]),
7      CM6 = numerl:matrix([[7.0, 10.0]]),
8      true = numerl:'=='(CM1, numerl:'*'(CM1, CM0)),
9      true = numerl:'=='(CM5, numerl:'*'(CM1, CM3)),
10     true = numerl:'=='(CM6 ,numerl:'*'(CM4, CM1)).
```

In the case of matrix inversion, an additional test done for a random matrix of order 500 was done. It was implemented as such:

```
1  %Creates a random matrix of size NxN
2  rnd_erl_matrix(N)->
3      numerl:matrix([ [rand:uniform(100) || _ <- lists:seq(1, N)]
           || _ <- lists:seq(1,N)]).
4  rnd_matrix(N)->
5      [ [rand:uniform(100) || _ <- lists:seq(1, N)] || _ <- lists:
           seq(1,N)].
6
7  %Creates an inversible matrix: use DGESV to see if the matrix
       defines a solvable system.
8  rnd_inv_matrix(N) when N > 1->
9      EM = rnd_erl_matrix(N),
10     R = numerl:dgesv(EM,EM),
11     if is_atom(R) ->
12         rnd_inv_matrix(N);
13     true ->
14         EM
15     end;
16 rnd_inv_matrix(1)->
17     rnd_inv_matrix(1).
18
19 inv_test() ->
```

```
20      M = numerl:matrix([[2.0, -1.0, 0.0], [-1.0, 2.0, -1.0], [
            0.0, -1.0, 2.0]]),
21      M_inv = numerl:inv(M),
22      %io:fwrite(numerl:print(M_inv),[]),
23      true = numerl:'=='(numerl:'*'(M, M_inv), numerl:eye(3)),
24      BM = rnd_inv_matrix(500),
25      true = numerl:'=='(numerl:'*'(numerl:inv(BM), BM), numerl:
            eye(500)).
```

The functions **rnd_inv_matrix** generates a random matrix **BM** of order 500, whose invertibility is tested using **dgesv** (presented in section 3.5.1). The **numerl:inv** function works if **in(BM)*BM** results in an identity matrix.

3.4.3 Memory leak test

The absence of memory leaks was tested. I did not find a profiling tool designed to check the absence of memory leaks in NIFs. The test I finally came up with was to run in a loop **numerl**'s functions for some time over predefined arguments, and checking in parallel that the Erlang process was not using increasingly more memory.

```
1   run_all_fcts(M,N)->
2       _ = numerl:eye(N),
3       _ = numerl:zeros(N, N),
4       _ = numerl:'+'(M,M),
5       _ = numerl:'-'(M,N),
6       _ = numerl:'*'(N,M),
7       _ = numerl:'*'(M,M),
8       _ = numerl:tr(M),
9       _ = numerl:inv(M).
10
11  loop_fct_until(Fct, Time)->
12      CurTime = erlang:system_time(second),
13      if CurTime > Time -> ok;
14      true ->
15          Fct(),
16          loop_fct_until(Fct, Time)
17      end.
18
19  loop_fct_for_s(Fct, Seconds)->
20      CurTime = erlang:system_time(second),
21      loop_fct_until(Fct, CurTime + Seconds).
```

It is then possible to run the loop as such:

```
1           M = numerl:matrix(benchmark:rnd_matrix(20)),
2           N = 20,
3           loop_fct_for_s(fun()->benchmark:run_all_fcts(M,N) end,
                1200).
```

Doing this test on a Ubuntu distribution, using the **top** command in a terminal to monitor the overall resource usage, I could verify whether the BEAM virtual machine was not increasing its memory usage.

3.5 Basic Linear Algebra Subprograms

As mentioned before, my first step in this thesis was to rewrite the **mat.erl** file as a NIF library written in C. Subsequently, key functions were optimised making use of a highly efficient library: BLAS.
The latter was first released as a Fortran library in 1979, and has since grown into a standard maintained by the BLAS Technical Forum ([4]). Multiple implementations, optimised for various hardware in multiple languages currently exist. Though a reference Fortran implementation can be found at the netlibt website ([12]), both Intel ([9]) and AMD ([2]) wrote their own C/Fortran implementation tuned for their own architecture.

3.5.1 The BLAS library

For this part of my research, I used the BLAS interface provided by GNU's CBLAS ([8]).

The BLAS operations are divided in three levels:

1. Vector operations, such as $y = \alpha x + y$
2. Matrix-vector operations, such as $y = \alpha Ax + \beta y$
3. Matrix-matrix operations, such as $C = \alpha AB + \beta C$

Each operation is defined for four precisions: single (S), double (D), single complex (C) and double complex (Z). Since my library uses the double format, only the D precision can be supported out of the box.

Various types of matrices are supported by BLAS: general (GE), symmetric (SY), etc,...

Each operation name combines a precision, a matrix type (if required), and a function name. Some examples of supported operations:

- ddot: d(ouble) precision, dot operation (scalar product) of vectors.
- daxpy: d(ouble) precision, a times x plus y operation.
- dgemm: d(ouble) precision, ge(neral) matrix, matrix * matrix operation.

3.5.2 LAPACKE library

LAPACKE ([10]) is a C library built atop BLAS which provides a wider range of operations over matrices, such as solvers for systems of equations and matrix decomposition. In particular, it provides the solver **dgesv** for the following problem: find x such that $Ax = B$; this can efficiently replace the **inv** function.

3.5.3 Wrapped nifs

The following BLAS-LAPACKE functions have been wrapped in my code.

- **daxpy(N, Alpha, X, Y)**: returns Alpha*X+Y for the first N values of X,Y.
- **ddot(N,X,Y)**: returns X*Y, considering the first N values of X,Y.
- **dgemv(Alpha, A, X, Beta, Y)**: returns Alpha*A*X + Beta*Y.
- **dgemm(Alpha, A, B, Beta, C)**: returns Alpha*A*B + Beta*C.
- **dgesv(A,B)**: returns X such that AX = B.

They use the following convention:

- A,B,C matrices
- N a non null integer

- X and Y vectors

Those wrappers give an interface as close as possible to their BLAS-LAPACKE equivalent, removing access to values which users cannot control(row or column major, stride between matrix/vector elements...).

3.6 Adding a new BLAS function

Adding a BLAS or LAPACKE function to Numerl requires multiple steps. I will show step by step how the **ddot** (double precision vector dot) function was added: after analysing the BLAS function (which can be found at [8]), one needs to decide how he can wrap it. The wrapper function can then be declared in the numerl.erl file, implemented in numerl.c, and tested in test_numer.erl.

3.6.1 Deciding on an interface

The **ddot** function uses the following arguments:

```
double precision function ddot ( integer  N,
        double precision, dimension(*)  DX,
        integer  INCX,
        double precision, dimension(*)  DY,
        integer  INCY )
```

N is the number of elements that needs to be considered in the multiplication. DX and DY are pointers to the vectors content; INCX and INCY are the steps between each vector value in the given array.
As vectors can be considered as matrices containing a single column/row, my Matrix implementation can be used to represent vectors. The INCX and INCY parameters are always 1 and can be discarded. Finally, the N parameter remains unchanged.
The **ddot** wrapper will have the following signature:

```
double ddot(int N, Matrix A, Matrix B)
```

Knowing this, the required modifications can be performed to the mentioned files.

3.6.2 Modifications to numerl.erl

At the start of the file, the function and it's arity need to be added to the export statement:

```
-export([ eye/1, zeros/2, ... , ddot/3]).
```

Followed by the function declaration:

```
ddot(_,_,_) -> nif_not_loaded.
```

So much for the numerl.erl modifications.

3.6.3 Modifications to numerl.c

Add the **NIF_dot** implementation:

```
ERL_NIF_TERM nif_ddot(ErlNifEnv *env, int argc, const ERL_NIF_TERM argv[]){
    Matrix x,y;
    int n;

    if(!enif_get(env, argv, "imm", &n, &x, &y)
        || fmin(x.n_rows, x.n_cols) * fmin(y.n_rows, y.n_cols) != 1){
        return enif_make_badarg(env);
    }

    double result = cblas_ddot(n, x.content, 1, y.content, 1);

    return enif_make_double(env, result);
}
```

At the files end, add a line to the **NIF_fun** array:

```
ErlNifFunc nif_funcs[] = {
    ...,
    {"ddot", 3, nif_ddot}
};
```

This line contains successively: the function name on the Erlang side, its arity, and its matching C function.

3.6.4 Modifications to test_numerl.erl

To test_numerl.erl, declare a function finishing with **_test** and do test cases which crash on failure (for example, do pattern matching with =).

```
ddot_test() ->
    Incs = numerl:matrix([[1, 2, 3, 4]]),
    Ones = numerl:matrix([[1], [1], [1], [1]]),
    10.0 = numerl:ddot(4, Incs, Ones),
    30.0 = numerl:ddot(4, Incs, Incs),
    4.0 = numerl:ddot(4, Ones, Ones),
    1.0 = numerl:ddot(1, Incs, Ones).
```

Chapter 4

Results

Now that I have re-implemented **mat.erl** as **numerl.erl**, we can compare their speed difference.

4.1 Case study: Zero matrix generation

In figure 4.2, I compared the execution time required both by a pure Erlang function (**zero_e**) and its C NIF equivalent (**zero_c**), to generate a NxN sized matrix filled with zeros. As mentioned at 3.2.4, using a NIF comes with some costs: the NIF needs to be launched, and its arguments need to be translated to their C representation. This takes some time and as illustrated in the graph 4.2, creating a 1x1 matrix is faster in Erlang than in a NIF.

The NIF implementation operates in two steps: allocate a matrix (**enif_alloc_binary**), set its content to zero (**memset**), transfer it to an Erlang term and return it. For the studied range $n \in [1, 10]$, the resulting function appears to execute in a constant time. However, on the greater range $n \in [10, 5000]$ displayed in figure 4.1, this function behaves with an $O(n^2)$ time complexity (for a matrix containing n elements).

Erlangs implementation is done through list comprehension. It has to create $n+1$ lists, of which each first n'th are filled with n elements (0). This algorithm executes in $O(n^2)$ time complexity, as displayed in figure 4.2.

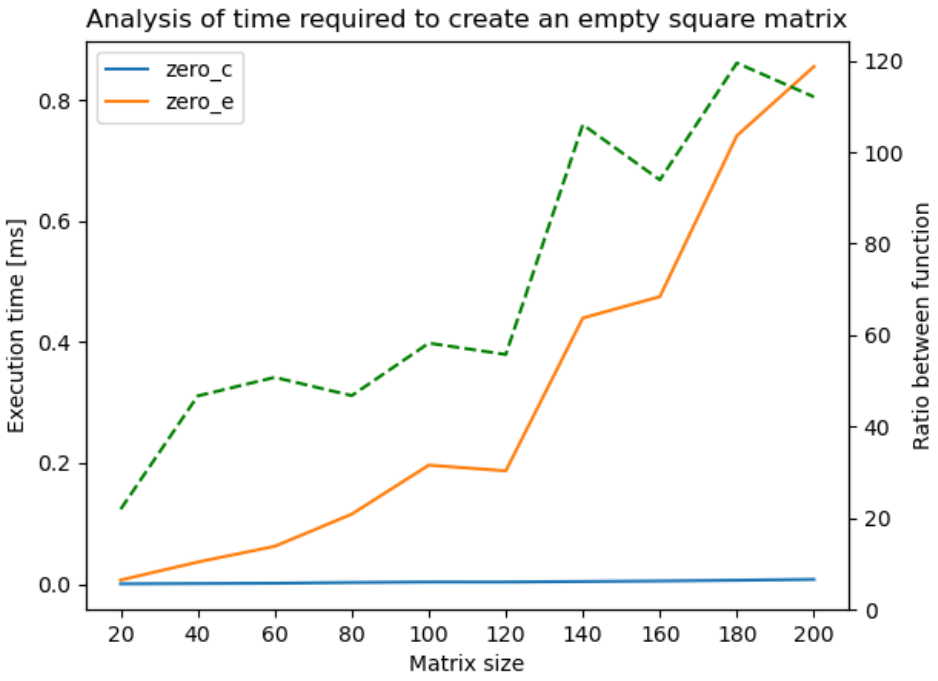

Figure 4.1: Creation of NxN empty matrices

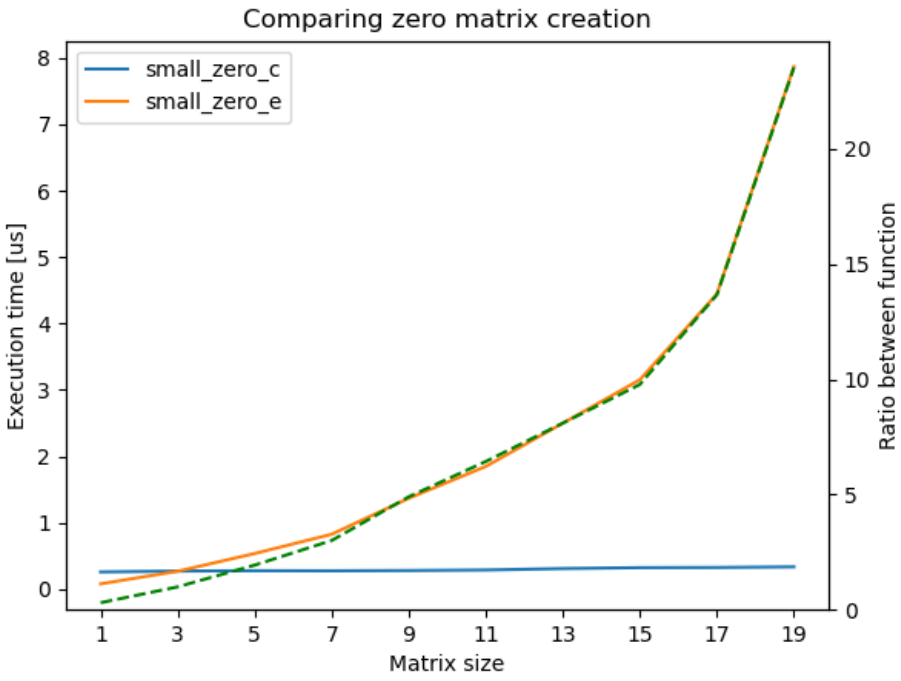

Figure 4.2: Creation of a small NxN empty matrices

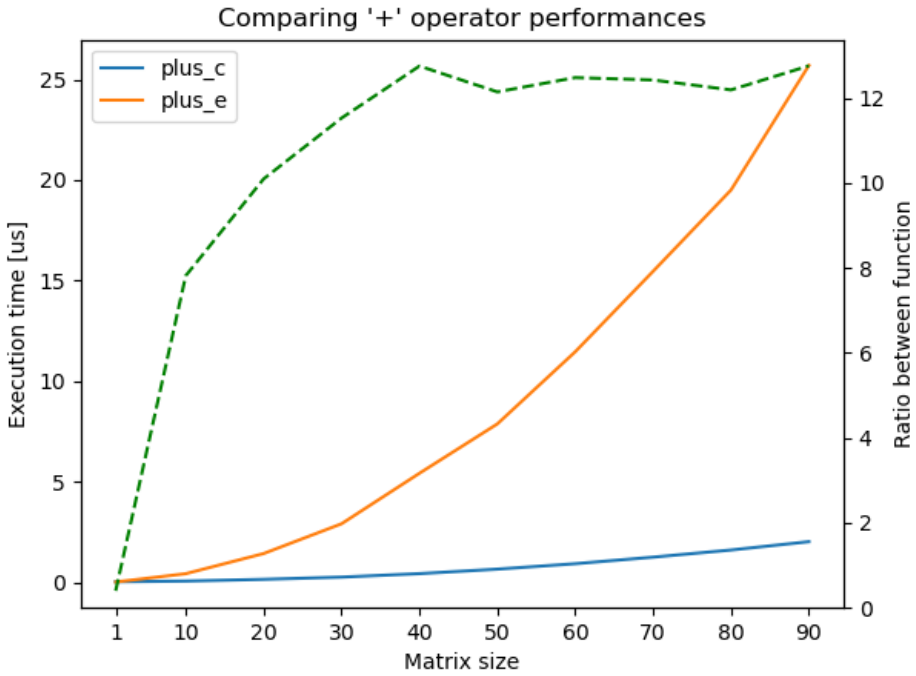

Figure 4.3: Comparison of addition

Aside mentioned overhead, Erlangs version takes considerably more time to run than a NIF for n > 2.

4.2 Remaining functions

Each implemented function behaves similarly to the **zeros** mentioned earlier, as it can be seen in the following graphs.

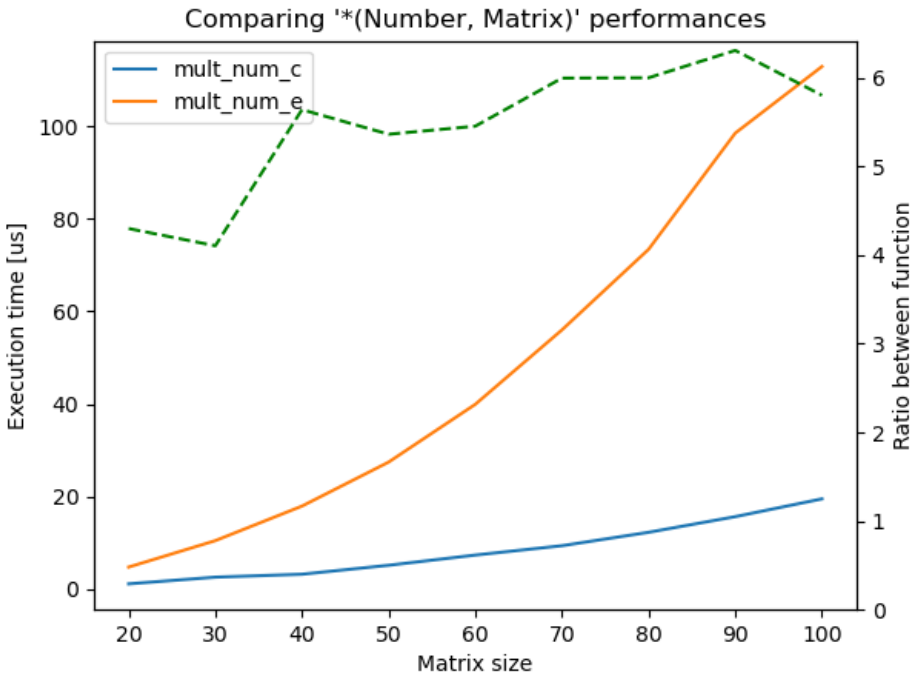

Figure 4.4: Comparison of number-matrix multiplication

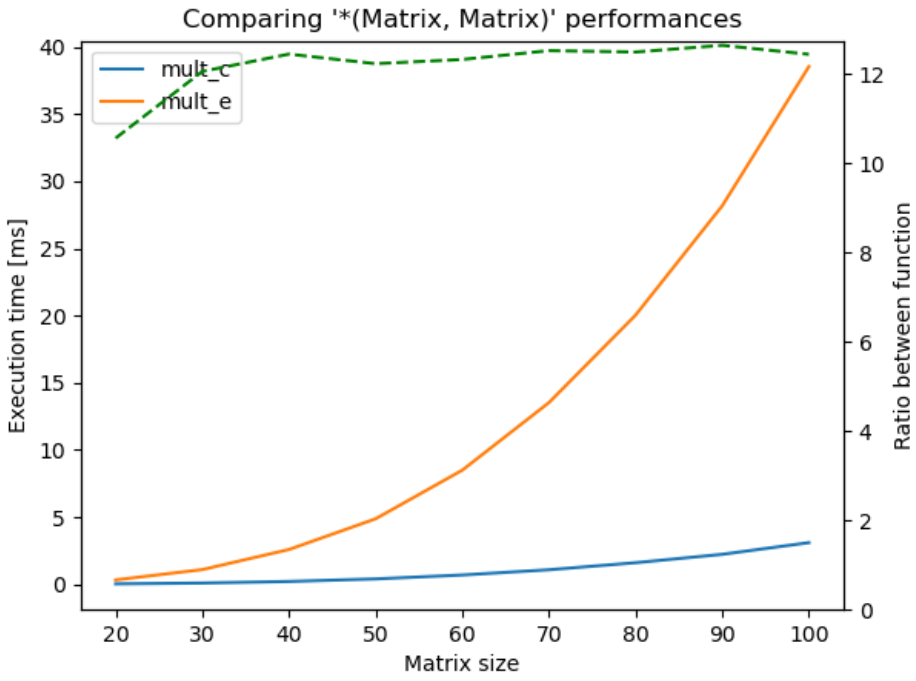

Figure 4.5: Comparison of matrix-matrix multiplication

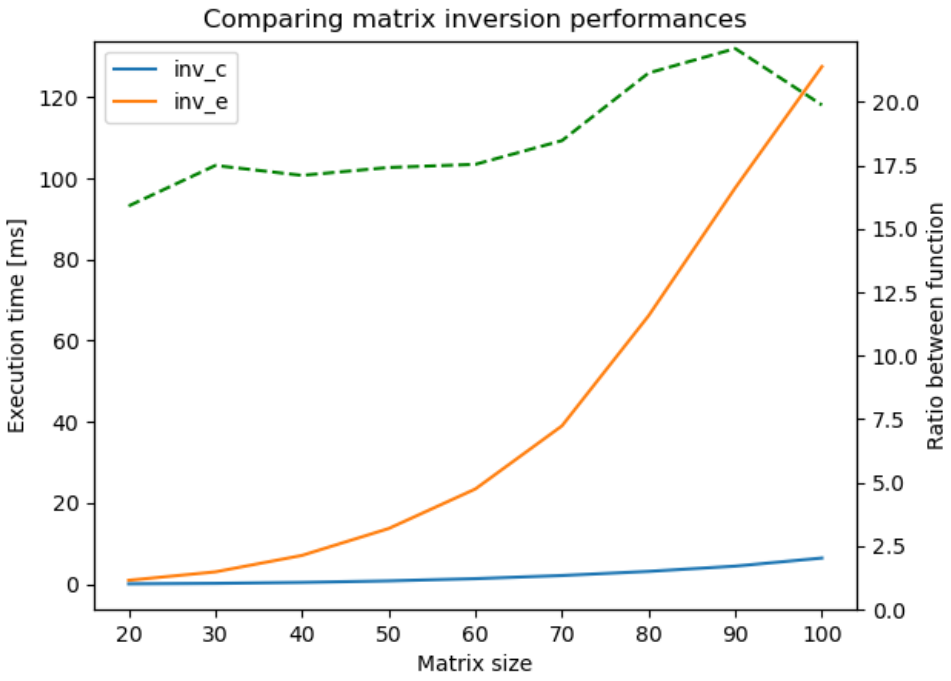

Figure 4.6: Comparison of matrix inversion

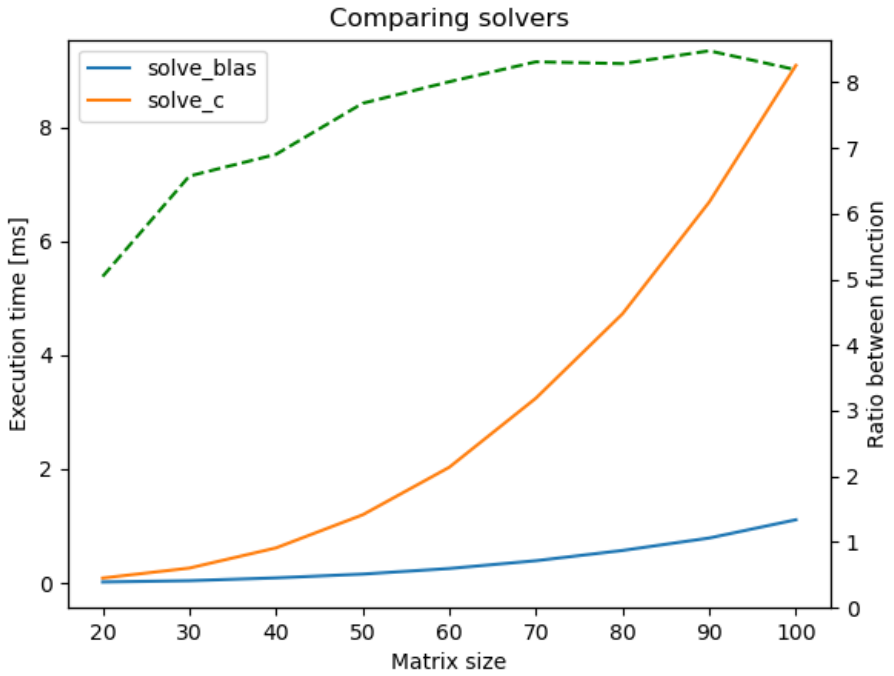

Figure 4.7: Comparison of linear system resolution

4.3 Solution of a linear system of equations

In this section, I study the time required to solve a linear system of equation $Ax = B$, by LAPACKE and Numerl.
The numerl algorithm is based on the Gauss Jordan matrix inversion:

```
1        I = numerl:inverse(A),
2        numerl:'*'(A,B).
```

Instead, LAPACKE proposes the solver **dgesv**, of which the wrapping is detailed in figure 3.5.3. This solver is based on a LU decomposition of its input matrix.

Though they perform similarly for small matrices, the LAPACKE version quickly outperforms its Numerl equivalent.

Chapter 5

Conclusion

This master thesis goal was to explore how an efficient matrix library could be written for Erlang.

Using the work of [15], I demonstrated that Erlang is slower than C at performing matrix operations. Then, I explored the possibility of using the latter as an Erlang NIF library.

The handling of NIFs presents multiple challenges; they cannot crash, and they have to interact with Erlang's primitive types, which greatly differ with the ones used in C. Even so, the resulting library is much faster; for example, twelve times for the multiplication of matrices, six times for the multiplication of a matrix by a number, and twenty times for the inversion of matrices.

Having solved those challenges, I explored how I could integrate in the resulting library BLAS functions, which are the most recognised standards for matrix handling in C.

The resulting library vastly outperforms what could be done with a pure Erlang implementation.

5.1 Future Work

This library is far from a final product, and further work is required.

BLAS functions could be directly integrated within the NIFs of addition,multiplication and inversion, resulting in faster functions.

N dimensional arrays could be implemented, allowing the representation of vectors, matrices, and tensors.

Performing a higher number of tests over random matrices. In addition, checking that NIFs throw **badarg** and **errors** as expected would be a great addition. Ultimately, automated tests for memory leaks should be added.

Finally, a major issue was left: **numerl**'s NIFs can exceed the 1ms barrier. Further work will be done in "The best of both worlds" [5] to resolve this. This research will explore the possibility of tiling matrices, running the NIFs over the tiled sub-matrices. This should also provide a performance boost, using parallelism at Erlang level to run multiple NIFs at once on those smaller matrices.

Bibliography

[1] *A BEAM brief introduction*. URL: https://www.erlang.org/blog/a-brief-beam-primer/. (19/12/2021).

[2] *AMD's BLIS library*. URL: https://developer.amd.com/amd-aocl/blas-library/. (accessed:18/12/2021).

[3] *BLAS and LAPACK reference*. URL: https://www.netlib.org/lapack/explore-html/. (accessed: 18/12/2021).

[4] *BLAS Technical Forum*. URL: http://www.netlib.org/blas/blast-forum/. (accessed: 18/12/2021).

[5] Basile Couplet and Lylian Brunet. *The best of both world*. Expected June 2022.

[6] *Erlangs scheduling*. URL: https://hamidreza-s.github.io/erlang/scheduling/real-time/preemptive/migration/2016/02/09/erlang-scheduler-details.html. 22/12/2021.

[7] *Gauss Jordan's algorithm*. URL: https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-Jordan#Pseudocode. 20/12/2021.

[8] *GNU's GSL CBLAS library*. URL: https://www.gnu.org/software/gsl/doc/html/cblas.html. (accessed: 18/12/2021).

[9] *Intel's Math Kernel Library*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html. (accessed: 18/12/2021).

[10] *LAPACKE*. URL: https://www.netlib.org/lapack/. (accessed: 18/12/2021).

[11] *Learn You Some Erlang for Great Good*. URL: https://learnyousomeerlang.com/. (accessed: 18/12/2021).

[12] *Netlib's BLAS reference*. URL: https://www.netlib.org/blas/index.html. (accessed: 18/12/2021).

[13] *Nif man page*. URL: https://www.erlang.org/doc/man/erl_nif.html. 22/12/2021.

[14] *Optimizing cache access*. URL: https://stackoverflow.com/questions/34189896/cache-performance-concerning-loops-in-c. 23/12/2021.

[15] Kalbusch Sébastien and Verpoten Vincent. "The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking". MA thesis. UCLouvain, 2020. URL: https://dial.uclouvain.be/memoire/ucl/object/thesis:30740.

Appendix A

Benchmarks

The benchmark of C code was done as such:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <stdio.h>
6
7
8
9  int n_runs = 100;
10
11 int max(int* a, int len){
12  int m = 0;
13
14  for(int i = 0; i<len; i++){
15   m = m>i? m:i;
16  }
17
18  return m;
19 }
20
21 void kmax(int*a, int len){
22  max(a,len); max(a,len); max(a,len); max(a, len); max(a,len);
        max(a,len); max(a,len); max(a,len); max(a, len); max(a,len);
23  //For clarity, duplicates of this line were removed; it should
        be present 500 times.
24 }
25
```

```
26 int main(){
27  int n_max = 151;
28  size_t s = sizeof(int)*n_max;
29  int* a = malloc(s);
30
31
32  for(int n = 15; n<n_max; n+= 15){
33   //A little warmup
34   for(int i = 0; i<n_runs/2; i++)
35    kmax(a, n);
36
37
38
39   //Actual benchmark
40   double time_taken = 0;
41   struct timespec start, end;
42
43   for(int i = 0; i<n_runs; i++){
44    int res;
45    clock_gettime(CLOCK_REALTIME, &start);
46    kmax(a,n);
47    clock_gettime(CLOCK_REALTIME, &end);
48    time_taken += ((end.tv_sec - start.tv_sec)*1e6 + (end.tv_nsec
          - start.tv_nsec)/1e3) /(n_runs*5000);
49   }
50
51   printf("Execution time: %f us for size %d \n", time_taken, n);
52  }
53
54  free(a);
55 }
```

Erlang's and NIF's functions were tested with this library:

```
1 -module(benchmark).
2 -compile(export_all).
3 -import(math, [log10/1]).
4 -import(timer, [sleep/1, tc/3]).
5
6
7 %Used to measure the average run time required for a function to
      run.
8 %Measure in micro seconds.
9 bench(_,_,_,0,R)->
10     R;
```

```
11
12 bench(F, A,N,I,R)->
13     {Time, _} = timer:tc(F, A),
14     bench(F, A, N,I-1, R+(Time/float(N))).
15
16 %Run the function N time to get its average performance.
17 bench(F ,Args ,N) ->
18     Argv = Args(N),
19     %Do a FULL preheat
20     _ = bench(F, Argv, N, N, 0.0),
21     bench(F, Argv, N, N, 0.0).
22
23 %Run the function F a number of times.
24 bench(F, Args)->
25     bench(F, Args, 5).
26
27
28
29
30 %Creates a random matrix containing a single row.
31 rnd_row(N)->
32     [[rand:uniform(100) || _ <- lists:seq(1,N)]].
33
34 %Creates a random matrix of size NxN
35 rnd_matrix(N)->
36     [ [rand:uniform(100) || _ <- lists:seq(1, N)] || _ <- lists:
          seq(1,N)].
37
38 %Creates a random int
39 rnd() ->
40     rand:uniform(100).
41
42
43 %Creates an inversible matrix: use DGESV to see if the matrix
       defines a solvable system.
44 inv_matrix(N) when N > 1->
45     M = rnd_matrix(N),
46     EM = numerl:matrix(M),
47     R = numerl:dgesv(EM,EM),
48     if is_atom(R) ->
49         inv_matrix(N);
50     true ->
51         M
52     end;
53 inv_matrix(1)->
```

```
54      rnd_matrix(1).
55
56  %Prints the given results.
57  show_results(Name, T_e, T_n)->
58      io:format("~nTesting ~w\nErlang native:  ~f\nNif: ~f\nFactor
            :~f~n", [Name,T_e, T_n, T_e/T_n]).
59
60  %Save a function
61  write_to_file(Name, Intervals, Values)->
62      FName = string:concat(string:concat("../benchRes/", Name), "
            .txt"),
63      {ok, File} = file:open(FName, [write]),
64      io:fwrite(File, "~p~n~p", [Intervals, Values]).
65
66  bench_mat_creation(N)->
67      io:format("Benching mat creation~n"),
68      List = [ [ rand:uniform(100) || _ <-lists:seq(0,N)] ],
69      Time = bench(fun numerl:matrix/1, fun(_) -> [List] end,
            10000),
70      io:format("Result:~f~n", [float(Time)]).
71
72  bench_plus(N)->
73      io:format("Benching plus~n"),
74      Time = bench(fun numerl:'+'/2, fun(_) -> [numerl:matrix(
            rnd_matrix(N)), numerl:matrix(rnd_matrix(N))] end, 1000),
75      io:format("Result:~f~n", [float(Time)]).
76
77  bench_mult(N)->
78      io:format("Benching multiplication of matrices~n"),
79      Time = bench(fun numerl:'*'/2, fun(_) -> [numerl:matrix(
            rnd_matrix(N)), numerl:matrix(rnd_matrix(N))] end, 1000),
80      io:format("Result:~f~n", [float(Time)]).
81
82  bench_mult_tr(N)->
83      io:format("Benching multiplication of matrices tr~n"),
84      Time = bench(fun numerl:'*tr'/2, fun(_) -> [numerl:matrix(
            rnd_matrix(N)), numerl:matrix(rnd_matrix(N))] end, 1000),
85      io:format("Result:~f~n", [float(Time)]).
86
87  bench_mult_erl(N)->
88      io:format("Benching multiplication of matrices tr~n"),
89      Time = bench(fun mat:'*'/2, fun(_) -> [rnd_matrix(N),
            rnd_matrix(N)] end, 1000),
90      io:format("Result:~f~n", [float(Time)]).
91
```

```
92  bench_tr(N)->
93      io:format("Benching tr of square matrix~n"),
94      Time = bench(fun numerl:'tr'/1, fun(_) -> [numerl:matrix(
            rnd_matrix(N))] end, 1000),
95      io:format("Result:~f~n", [float(Time)]).
96
97  bench_dot(N)->
98      io:format("Benching dot time execution~n"),
99      Steps = lists:seq(0, N),
100     A = [1 || _ <- Steps],
101     B = [2 || _ <- Steps],
102     Fct = fun(X,Y) -> lists:sum(lists:zipwith(fun erlang:'*'/2 ,
            X,Y)) end,
103     Time = bench(Fct, fun(_) -> [A,B] end, 1000),
104     io:format("Result:~f~n", [float(Time)]).
105
106 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
107 %%% Compare performances
108 %%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
109
110
111 %Used to measure the average run time required for a function to
        run.
112 %Measure in milli seconds.
113 b(_,_,_,0,R)->
114     R;
115
116 b(F, A,N,I,R)->
117     {Time, _} = timer:tc(F, A),
118     b(F, A, N,I-1, R+(Time/float(N))).
119
120 %Run the function N time to get its average performance.
121 b(F ,Args ,N) ->
122     %Do a preheat
123     _ = b(F, Args, 500, 500, 0.0),
124     b(F, Args, N, N, 0.0).
125
126 bench_fcts(_,_,[],[],[])->
127     io:format("Finished.");
128
129 bench_fcts(NRuns, Steps, [F|Fcts], [Arg|Args], [File|Files])->
130     io:format("Running benchmark ~s", [File]),
131     Result = [b(F, Arg(N), NRuns) || N<-Steps],
132     write_to_file(File, Steps, Result),
133     io:format ("; finished in ~f s~n", [lists:sum(Result)*(NRuns
```

```
          +500)/1000000]),
134       bench_fcts(NRuns, Steps, Fcts, Args, Files).
135
136
137   bench_zeros(NRuns, Steps)->
138       Fcts = [fun numerl:zeros/2, fun mat:zeros/2],
139       Args = [fun (N)->[N,N] end, fun (N)->[N,N] end],
140       Files = ["zero_c", "zero_e"],
141       bench_fcts(NRuns, Steps, Fcts, Args, Files).
142   bench_zeros()->
143       bench_zeros(500, lists:seq(20, 200, 20)).
144
145
146   bench_mult_num(NRuns, Steps)->
147       io:format("Benching mult num operator.~n"),
148       Fcts = [fun mat:'*'/2, fun numerl:'*'/2],
149       Args = [fun (N) -> [rnd(), rnd_matrix(N)] end,
150               fun (N) -> [rnd(), numerl:matrix(rnd_matrix(N))] end
                    ],
151       Files = ["mult_num_e", "mult_num_c"],
152       bench_fcts(NRuns, Steps, Fcts, Args, Files).
153   bench_mult_num()->
154       bench_mult_num(1500, lists:seq(20,100,10)).
155
156
157   bench_mult(NRuns, Steps)->
158       io:format("Benching mult operator.~n"),
159       Fcts = [fun mat:'*'/2, fun numerl:'*'/2],
160       Args = [fun (N) -> [rnd_matrix(N), rnd_matrix(N)] end,
161               fun (N) -> [numerl:matrix(rnd_matrix(N)), numerl:
                      matrix(rnd_matrix(N))] end],
162       Files = ["mult_e", "mult_c"],
163       bench_fcts(NRuns, Steps, Fcts, Args, Files).
164   bench_mult()->
165       bench_mult(500, lists:seq(20,100,10)).
166
167
168   bench_inv(NRuns, Steps)->
169       io:format("Benching inv operator.~n"),
170       Fcts = [fun mat:inv/1, fun numerl:inv/1],
171       Args = [fun (N) -> [inv_matrix(N)] end,
172               fun (N) -> [numerl:matrix(inv_matrix(N))] end],
173       Files = ["inv_e", "inv_c"],
174       bench_fcts(NRuns, Steps, Fcts, Args, Files).
175   bench_inv()->
```

```
176     bench_inv(500, lists:seq(20,100,10)).
177
178
179 bench_multb(NRuns, Steps)->
180     io:format("Benching BLAS agains na ve multiplication.~n"),
181     Fcts = [ fun numerl:dgemm/5,  fun numerl:'*'/2],
182     Args = [
183             fun (N) -> [1,numerl:matrix(rnd_matrix(N)), numerl:
                    matrix(rnd_matrix(N)), 0, numerl:matrix(
                    rnd_matrix(N))] end,
184             fun (N) -> [numerl:matrix(rnd_matrix(N)), numerl:
                    matrix(rnd_matrix(N))] end],
185     Files = [ "mult_blas", "mult_nc"],
186     bench_fcts(NRuns, Steps, Fcts, Args, Files).
187 bench_multb()->
188     bench_multb(500, lists:seq(20,100,10)).
189
190 bench_solve(NRuns, Steps)->
191     io:format("Benching inverse solver.~n"),
192     Fcts = [ fun(A,B)-> I = numerl:inv(A), numerl:'*'(I, B) end,
            fun numerl:dgesv/2],
193     Args = [
194             fun (N) -> [numerl:matrix(inv_matrix(N)), numerl:
                    matrix(rnd_matrix(N))] end,
195             fun (N) -> [numerl:matrix(inv_matrix(N)), numerl:
                    matrix(rnd_matrix(N))] end],
196     Files = [ "solve_c", "solve_blas"],
197     bench_fcts(NRuns, Steps, Fcts, Args, Files).
198 bench_solve()->
199     bench_solve(500, lists:seq(20,100,10)).
200
201 bench_tr(NRuns, Steps)->
202     io:format("Benching tr operator.~n"),
203     Fcts = [fun mat:tr/1, fun numerl:tr/1],
204     Args = [fun (N) -> [rnd_matrix(N)] end,
205             fun (N) -> [numerl:matrix(rnd_matrix(N))] end],
206     Files = ["tr_e", "tr_c"],
207     bench_fcts(NRuns, Steps, Fcts, Args, Files).
208 bench_tr()->
209     bench_tr(2000, lists:seq(20,200,20)).
210
211
212 bench()->
213     bench_zeros(),
214     bench_mult_num(),
```

```
    bench_mult(),
    bench_inv(),
    bench_multb(),
    bench_solve(),
    bench_tr().

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Benchmark rapid functions: call them Factor times instead of 1 time
%%% to get more accurate measures

bench_small_fcts(_,_,_, [],[],[])->
    io:format("Finished.");

bench_small_fcts(NRuns, Steps, Factor, [F|Fcts], [Arg|Args], [File|Files])->
    io:format("Running benchmark  ~s", [File]),
    Result = [b(F, Arg(N), NRuns) / Factor|| N<-Steps],
    write_to_file(File, Steps, Result),
    io:format ("; finished in ~f ms~n", [lists:sum(Result)*NRuns*Factor/1000]),
    bench_small_fcts(NRuns, Steps, Factor, Fcts, Args, Files).



bench_plus(NRuns, Steps)->
    io:format("Benching plus operator.~n"),
    Fcts = [fun benchk:matplus/2, fun benchk:numerlplus/2],
    Args = [fun (N) -> [rnd_matrix(N), rnd_matrix(N)] end,
            fun (N) -> [numerl:matrix(rnd_matrix(N)), numerl:matrix(rnd_matrix(N))] end],
    Files = ["plus_e", "plus_c"],
    bench_small_fcts(NRuns, Steps, 10000, Fcts, Args, Files).
bench_plus()->
    bench_plus(100, lists:seq(1,100,10)).

rnd_vec_max(N)->
    [ float(rnd()) || _ <- lists:seq(1,N)].

bench_emax(NRuns, Steps)->
    Fcts = [fun benchk:listMax/2],
    Args = [fun (I) -> [rnd_vec_max(I), 0] end],
    File = ["max_e"],
    bench_small_fcts(NRuns, Steps, 10000, Fcts, Args, File).
```

```
256 bench_max(NRuns, Steps)->
257     Fcts = [fun benchk:listMax/2, fun benchk:nif_max/2, fun
            benchk:nif_matrix_max/2],
258     Args = [fun (I) -> [rnd_vec_max(I)] end,
259             fun (I) -> [rnd_vec_max(I)] end,
260             fun (I) -> [numerl:matrix([rnd_vec_max(I)])] end],
261
262     Files = ["max_list_e", "max_list_c", "max_mat_c"],
263     bench_small_fcts(NRuns, Steps, 1000, Fcts, Args, Files).
264 bench_max()->
265     bench_max(20000, lists:seq(1,10)).
266
267 bench_small_zero(NRuns, Steps)->
268     io:format("Benching small zero.~n"),
269     Fcts = [fun benchk:mat_zeros/2, fun benchk:numerl_zeros/2],
270     Args = [fun (N) -> [N,N] end,
271             fun (N) -> [N,N] end],
272     Files = ["small_zero_e", "small_zero_c"],
273     bench_small_fcts(NRuns, Steps, 1000, Fcts, Args, Files).
274 bench_small_zero()->
275     bench_small_zero(1000, lists:seq(1,50,5)).
276
277 bench_small_mult(NRuns, Steps)->
278     io:format("Benching small mult.~n"),
279     Fcts = [fun benchk:mat_mult/2, fun benchk:numerl_mult/2],
280     Args = [fun (N) -> [rnd_matrix(N),rnd_matrix(N)] end,
281             fun (N) -> [numerl:matrix(rnd_matrix(N)),numerl:
                    matrix(rnd_matrix(N))] end],
282     Files = ["small_mult_e", "small_mult_c"],
283     bench_small_fcts(NRuns, Steps, 500, Fcts, Args, Files).
284 bench_small_mult()->
285     bench_small_mult(500, lists:seq(1,10,1)).
286
287 bench_small_inv(NRuns, Steps)->
288     io:format("Benching small mult.~n"),
289     Fcts = [fun benchk:mat_inv/1, fun benchk:numerl_inv/1],
290     Args = [fun (N) -> [rnd_matrix(N)] end,
291             fun (N) -> [numerl:matrix(rnd_matrix(N))] end],
292     Files = ["small_inv_e", "small_inv_c"],
293     bench_small_fcts(NRuns, Steps, 1000, Fcts, Args, Files).
294
295
296 run_all_fcts(M,N)->
297     _ = numerl:eye(N),
298     _ = numerl:zeros(N, N),
```

```
299       _ = numerl:'+'(M,M),
300       _ = numerl:'-'(M,N),
301       _ = numerl:'*'(N,M),
302       _ = numerl:'*'(M,M),
303       _ = numerl:tr(M),
304       _ = numerl:inv(M).
305
306
307   loop_fct_until(Fct, Time)->
308       CurTime = erlang:system_time(second),
309       if CurTime > Time -> ok;
310       true ->
311           Fct(),
312           loop_fct_until(Fct, Time)
313       end.
314
315   loop_fct_for_s(Fct, Seconds)->
316       CurTime = erlang:system_time(second),
317       loop_fct_until(Fct, CurTime + Seconds).
318
319
320   reduce_col([], C, Rm)->
321       {lists:reverse(C), lists:reverse(Rm)};
322   reduce_col([[Hr|Tr]|Rows], C, Rm)->
323       reduce_col(Rows, [Hr|C], [Tr|Rm]).
324
325   btr([[]|_], Tr)->
326       lists:reverse(Tr);
327   btr(M, Tr)->
328       {Col, Rst} = reduce_col(M, [], []),
329       btr(Rst, [Col|Tr]).
330   btr(M)->
331       btr(M,[]).
332
333
334   %btr(Matrix, AccCols, AccCol, NewRows).
335
336   %Job finished!
337   btrf([[]|_], [], [], NewRows)->
338       lists:reverse(NewRows);
339   %Finished extracting a column
340   btrf([], AccCols, AccCol, NewRows)->
341       btrf(lists:reverse(AccCols), [], [], [lists:reverse(AccCol)|
              NewRows]);
342   %Currently extracting a column
```

```
343 btrf([[Rh|Rt]|Rows], AccCols, AccCol, NewRows)->
344     btrf(Rows, [Rt|AccCols], [Rh|AccCol], NewRows).
345 %Simpler function.
346 btrf(M)->
347     btrf(M, [], [], []).
348
349
350
351 %btrf(M)->
352 %     btrf().
353
354
355
356 split(0,_)->
357     [];
358 split(N,[H|T])->
359     [H|split(N-1,T)].
360
361 split_tr(0, _,R)->
362     lists:reverse(R,[]);
363 split_tr(N,[H|T], R)->
364     split_tr(N-1, T, [H|R]).
365
366 m_tr([[]|_]) -> [];
367 m_tr(L)->
368     [[ C || [C|_] <- L] | m_tr([T || [_|T] <- L])].
```

Appendix B

mat.erl

```
-module(mat).

-export([tr/1, inv/1]).
-export(['+'/2, '-'/2, '=='/2, '*'/2, '*  '/2]).
-export([row/2, col/2, get/3]).
-export([zeros/2, eye/1]).
-export([eval/1, list_to_matrix/2]).

-export_type([matrix/0]).

-type matrix() :: [[number(), ...], ...].

%% convert a list into a NxM matrix with N = length(List)/M
-spec list_to_matrix(List, M) -> Matrix when
    List :: list(),
    M :: pos_integer(),
    Matrix :: matrix().

list_to_matrix(List, M) ->
    list_to_matrix(List, M, []).


%% transpose matrix
-spec tr(M) -> Transposed when
    M :: matrix(),
    Transposed :: matrix().

tr(M) ->
    tr(M, []).
```

```
%% matrix addition (M3 = M1 + M2)
-spec '+'(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3:: matrix().

'+'(M1, M2) ->
    element_wise_op(fun erlang:'+'/2, M1, M2).


%% matrix subtraction (M3 = M1 - M2)
-spec '-'(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'-'(M1, M2) ->
    element_wise_op(fun erlang:'-'/2, M1, M2).


%% matrix multiplication (M3 = Op1 * M2)
-spec '*'(Op1, M2) -> M3 when
    Op1 :: number() | matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'*'(N, M) when is_number(N) ->
    [[N*X|| X <- Row] || Row <- M];
'*'(M1, M2) ->
    '*  '(M1, tr(M2)).


%% transposed matrix multiplication (M3 = M1 * tr(M2))
-spec '*  '(M1, M2) -> M3 when
    M1 :: matrix(),
    M2 :: matrix(),
    M3 :: matrix().

'*  '(M1, M2) ->
    [[lists:sum(lists:zipwith(fun erlang:'*'/2, Li, Cj))
        || Cj <- M2]
        || Li <- M1].

%% return true if M1 equals M2 using 1e-6 precision
```

```
-spec '=='(M1, M2) -> boolean() when
    M1 :: matrix(),
    M2 :: matrix().

'=='(M1, M2) ->
    case length(M1) == length(M2) of
        true when length(hd(M1)) == length(hd(M2)) ->
            RoundFloat = fun(F) -> round(F*1000000)/1000000 end,
            CmpFloat = fun(F1, F2) -> RoundFloat(F1) ==
                RoundFloat(F2) end,
            Eq = element_wise_op(CmpFloat, M1, M2),
            lists:all(fun(Row) -> lists:all(fun(B) -> B end, Row
                ) end, Eq);
        false -> false
    end.


%% get the row I of M
-spec row(I, M) -> Row when
    I :: pos_integer(),
    M :: matrix(),
    Row :: matrix().

row(I, M) ->
    [lists:nth(I, M)].


%% get the column J of M
-spec col(J, M) -> Col when
    J :: pos_integer(),
    M :: matrix(),
    Col :: matrix().

col(J, M) ->
    [[lists:nth(J, Row)] || Row <- M].


%% get the element a index (I,J) in M
-spec get(I, J, M) -> Elem when
    I :: pos_integer(),
    J :: pos_integer(),
    M :: matrix(),
    Elem :: number().

get(I, J, M) ->
```

```
    lists:nth(J, lists:nth(I, M)).


%% build a null matrix of size NxM
-spec zeros(N, M) -> Zeros when
    N :: pos_integer(),
    M :: pos_integer(),
    Zeros :: matrix().

zeros(N, M) ->
    [[0 || _ <- lists:seq(1, M)] || _ <- lists:seq(1, N)].


%% build an identity matrix of size NxN
-spec eye(N) -> Identity when
    N :: pos_integer(),
    Identity :: matrix().

eye(N) ->
    [[ if I =:= J -> 1; true -> 0 end
        || J <- lists:seq(1, N)]
        || I <- lists:seq(1, N)].


%% compute the inverse of a square matrix
-spec inv(M) -> Invert when
    M :: matrix(),
    Invert :: matrix().

inv(M) ->
    N = length(M),
    A = lists:zipwith(fun lists:append/2, M, eye(N)),
    Gj = gauss_jordan(A, N, 0, 1),
    [lists:nthtail(N, Row) || Row <- Gj].


%% evaluate a list of matrix operations
-spec eval(Expr) -> Result when
    Expr :: [T],
    T :: matrix() | '+' | '-' | '*' | '*  ',
    Result :: matrix().

eval([L|[O|[R|T]]]) ->
    F = fun mat:O/2,
    eval([F(L, R)|T]);
```

```
eval([Res]) ->
    Res.



%% transpose matrix with accumulator
tr([[]|_], Rows) ->
    lists:reverse(Rows);
tr(M, Rows) ->
    {Row, Cols} = tr(M, [], []),
    tr(Cols, [Row|Rows]).

%% transpose the first row of a matrix with accumulators
tr([], Col, Cols) ->
    {lists:reverse(Col), lists:reverse(Cols)};
tr([[H|T]|Rows], Col, Cols) ->
    tr(Rows, [H|Col], [T|Cols]).

%% apply Op element wise on matrices M1 and M2
element_wise_op(Op, M1, M2) ->
    lists:zipwith(fun(L1, L2) -> lists:zipwith(Op, L1, L2) end,
        M1, M2).

%% Gauss-Jordan method from
%% https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-
    Jordan#Pseudocode
gauss_jordan(A, N, _, J) when J > N ->
    A;
gauss_jordan(A, N, R, J) ->
    case pivot(col(J, lists:nthtail(R, A)), R+1, {0, 0}) of
        {_, 0} ->
            gauss_jordan(A, N, R, J+1);
        {K, Pivot} ->
            A2 = swap(K, R+1, A),
            [Row] = row(R+1, A2),
            Norm = lists:map(fun(E) -> E/Pivot end, Row),
            A3 = gauss_jordan_aux(A2, {R+1, J}, Norm, 1, []),
            gauss_jordan(A3, N, R+1, J+1)
    end.

%% Matrix(i, :) -= Matrix(i, j)/Pivot * Matrix(R, :) forall i\{R
    }
%% Matrix(R, :) *= 1/Pivot
%% with Pivot = Matrix(R, j)
```

```
gauss_jordan_aux([], _, _, _, Acc) ->
    lists:reverse(Acc);
gauss_jordan_aux([_|Rows], {I, J}, L, I, Acc)->
    gauss_jordan_aux(Rows, {I, J}, L, I+1, [L|Acc]);
gauss_jordan_aux([Row|Rows], {R, J}, L, I, Acc) ->
    F = lists:nth(J, Row),
    NewRow = lists:zipwith(fun(A, B) -> A-F*B end, Row, L),
    gauss_jordan_aux(Rows, {R, J}, L, I+1, [NewRow|Acc]).

%% find the gauss jordan pivot of a column
pivot([], _, Pivot) ->
    Pivot;
pivot([[H]|T], I, {_, V}) when abs(H) >= abs(V) ->
    pivot(T, I+1, {I, H});
pivot([_|T], I, Pivot) ->
    pivot(T, I+1, Pivot).

%% swap two indexes of a list
%% taken from https://stackoverflow.com/a/64024907
swap(A, A, List) ->
    List;
swap(A, B, List) ->
    {P1, P2} = {min(A,B), max(A,B)},
    {L1, [Elem1 | T1]} = lists:split(P1-1, List),
    {L2, [Elem2 | L3]} = lists:split(P2-P1-1, T1),
    lists:append([L1, [Elem2], L2, [Elem1], L3]).

list_to_matrix([], _, Acc) ->
    lists:reverse(Acc);
list_to_matrix(List, M, Acc) ->
    {Row, Rest} = lists:split(M, List),
    list_to_matrix(Rest, M, [Row|Acc]).
```

Appendix C

Numerl.erl

```
-module(numerl).
-on_load(init/0).
-export([ eye/1, zeros/2, '==''/2, '+'/2, '-'/2,'*'/2, matrix/1,
    get/3, row/2, col/2, tr/1, inv/1, print/1, ddot/3, daxpy/4,
    dgemv/5,  dgemm/5, dgesv/2]).

%Matrices are represented as such:
%-record(matrix, {n_rows, n_cols, bin}).

%%  Load nif.
init()->
    erlang:load_nif("./numerl_nif", 0).


%%Creates a matrix.
%List: List of doubles, of length N.
%Return: a matrix of dimension MxN, containing the data.
matrix(_) ->
    nif_not_loaded.

%%Returns a value from a matrix.
get(_,_,_) ->
    nif_not_loaded.

%%Returns requested row.
row(_,_) ->
    nif_not_loaded.


%%Returns requested col.
```

```
col(_,_) ->
    nif_not_loaded.


%%Equality test between matrixes.
'=='(_, _) ->
    nif_not_loaded.


%%Addition of matrix.

'+'(_, _) ->
    nif_not_loaded.


%%Substraction of matrix.
'-'(_, _) ->
    nif_not_loaded.


%% Matrix multiplication.
'*'(A,B) when is_number(A) -> '*_num'(A,B);
'*'(A,B) -> '*_matrix'(A,B).

'*_num'(_,_)->
    nif_not_loaded.

'*_matrix'(_, _)->
    nif_not_loaded.


%% build a null matrix of size NxM
zeros(_, _) ->
    nif_not_loaded.

%%Returns an Identity matrix NxN.
eye(_)->
    nif_not_loaded.

%%Print in stdout the matrix's content.
print(_)->
    nif_not_loaded.

%Returns the transpose of the given square matrix.
tr(_)->
```

```
    nif_not_loaded.

%Returns the inverse of asked square matrix.
inv(_)->
    nif_not_loaded.


%-------CBLAS---------

% ddot: dot product of two vectors
% Arguments: int n, vector x, vector y.
%   n is the number of coordinates we should take into account (
    n < min(len(x), len(y)).
%   x and y are matrices, with one of their dimension equal to
    1.
% Returns the result of the dot product of the first N
    coordinates of x, y.
% The returned vector is in the same dimension of y (column or
    row).
ddot(_,_, _)->
    nif_not_loaded.

% daxpy: alpha*x + y
% Arguments: int n, number alpha, vector x, vector y.
%   n is the number of coordinates we should take into account (
    n < min(len(x), len(y)).
%   alpha is a number, converted to a double.
%   x and y are matrices, with one of their dimension equal to
    1.
% Returns the result of the operation alpha*x + y.
% The returned vector is in the same dimension of y (column or
    row).
daxpy(_,_,_,_)->
    nif_not_loaded.

% dgemv: alpha*A*x + beta*y
% Arguments: number alpha, Matrix A, vector x, number beta,
    vector y.
%   alpha and beta re a numbers, converted to doubles.
%   A is a Matrix.
%   x and y are matrices, with one of their dimension equal to
    1.
% Returns the result of the operation alpha*A*x + beta*y.
% The returned vector is in the same dimension of y (column or
    row).
```

```
dgemv(_,_,_,_,_,_)->
    nif_not_loaded.

% dgemm: alpha * A * B + beta * C.
% Arguments: number alpha, Matrix A, Matrix B, number beta,
    matrix C.
%   alpha, beta: numbers (float or ints) used as doubles.
%   A,B,C: matrices.
% Returns the matrice resulting of the operations alpha * A * B
    + beta * C.
dgemm(_,_,_,_,_)->
    nif_not_loaded.

%dgesv: A*x = b.
dgesv(_,_)->
    nif_not_loaded.
```

Appendix D

Numerl.c

```
#include <erl_nif.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <gsl/gsl_cblas.h>
#include <lapacke.h>


ERL_NIF_TERM atom_nok;
ERL_NIF_TERM atom_true;
ERL_NIF_TERM atom_false;
ERL_NIF_TERM atom_matrix;

ErlNifResourceType *MULT_YIELDING_ARG = NULL;

int load(ErlNifEnv* env, void** priv_data, ERL_NIF_TERM
    load_info){
    atom_nok = enif_make_atom(env, "nok\0");
    atom_true = enif_make_atom(env, "true\0");
    atom_false = enif_make_atom(env, "false\0");
    atom_matrix = enif_make_atom(env, "matrix\0");
    return 0;
}

int upgrade(ErlNifEnv* caller_env, void** priv_data, void**
    old_priv_data, ERL_NIF_TERM load_info){

}

//Gives easier access to an ErlangBinary containing a matrix.
```

```
typedef struct{
    int n_rows;
    int n_cols;

    //Content of the matrix, in row major format.
    double* content;

    //Erlang binary containing the matrix.
    ErlNifBinary bin;
} Matrix;


//Access asked coordinates of matrix
double* matrix_at(int row, int col, Matrix m){
    return &m.content[row*m.n_cols + col];
}

//Allocates memory space of for matrix of dimensions n_rows,
    n_cols.
//The matrix_content can be modified, until a call to
    array_to_erl.
//Matrix content is stored in row major format.
Matrix matrix_alloc(int n_rows, int n_cols){
    ErlNifBinary bin;

    enif_alloc_binary(sizeof(double)*n_rows*n_cols, &bin);

    Matrix matrix;
    matrix.n_cols = n_cols;
    matrix.n_rows = n_rows;
    matrix.bin = bin;
    matrix.content = (double*) bin.data;

    return matrix;
}

//Creates a duplicate of a matrix.
//This duplicate can be modified untill uploaded.
Matrix matrix_dup(Matrix m){
    Matrix d = matrix_alloc(m.n_rows, m.n_cols);
    memcpy(d.content, m.content, d.bin.size);
    return d;
}

//Free an allocated matrix that was not sent back to Erlang.
```

```
void free_matrix(Matrix m){
    enif_release_binary(&m.bin);
}

//Constructs a matrix erlang term.
//No modifications can be made afterwards to the matrix.
ERL_NIF_TERM matrix_to_erl(ErlNifEnv* env, Matrix m){
    ERL_NIF_TERM term = enif_make_binary(env, &m.bin);
    return enif_make_tuple4(env, atom_matrix, enif_make_int(env,
        m.n_rows), enif_make_int(env,m.n_cols), term);
}

//Reads an erlang term as a matrix.
//As such, no modifications can be made to the red matrix.
//Returns true if it was possible to read a matrix, false
    otherwise
int enif_get_matrix(ErlNifEnv* env, ERL_NIF_TERM term, Matrix *
    dest){

    int arity;
    const ERL_NIF_TERM* content;

    enif_get_tuple(env, term, &arity, &content);
    if(arity != 4)
        return 0;

    if(content[0] != atom_matrix
        || !enif_get_int(env, content[1], &dest->n_rows)
        || !enif_get_int(env, content[2], &dest->n_cols)
        || !enif_inspect_binary(env, content[3], &dest->bin))
    {
        return 0;
    }

    dest->content = (double*) (dest->bin.data);
    return 1;
}

//Equal all doubles
//Compares wether all doubles are approximatively the same.
int equal_ad(double* a, double* b, int size){
    for(int i = 0; i<size; i++){
        if(fabs(a[i] - b[i])> 1e-6)
            return 0;
    }
```

```
    return 1;
}

int read_choice(ErlNifEnv* env, ERL_NIF_TERM term, char* option1
    , char* option2, int* dest){
    unsigned len;
    char buffer[30];


    if(!enif_get_atom_length(env, term, &len, ERL_NIF_LATIN1)
            || len > 30
            || ! enif_get_atom(env, term, buffer, 30,
               ERL_NIF_LATIN1))
        return 0;

    if(!strcmp(buffer, option1))
        *dest = 1;
    else if(!strcmp(buffer, option2))
        *dest = 0;
    else
        return 0;
    return 1;
}

//Used to translate at once a number of ERL_NIF_TERM.
//Data types are infered via content of format string:
//  n: number (int or double) translated to double.
//  m: matrix
//  i: int
int enif_get(ErlNifEnv* env, const ERL_NIF_TERM* erl_terms,
    const char* format, ...){
    va_list valist;
    va_start(valist, format);
    int valid = 1;

    while(valid && *format != '\0'){
        switch(*format++){
            case 'n':
                //Read a number as a double.
                ;
                double *d = va_arg(valist, double*);
                int i;
                if(!enif_get_double(env, *erl_terms, d))
                    if(enif_get_int(env, *erl_terms, &i))
                        *d = (double) i;
```

```
        else valid = 0;
    break;

case 'm':
    //Reads a matrix.
    valid = enif_get_matrix(env, *erl_terms, va_arg(
        valist, Matrix*));
    break;

case 'i':
    //Reads an int.
    valid = enif_get_int(env, *erl_terms, va_arg(
        valist, int*));
    break;

case 'v':
    //Read vertical axis: triUpper, triLower.
    //Set value to 1 if upper; 0 if lower; otherwise
        returns invalid.
    valid = read_choice(env, *erl_terms, "triUpper",
        "triLower", (va_arg(valist, int*)));
    break;


case 'h':
//Read horizontal axis: left, right.
//Set value to 1 left 0 if right; otherwise returns
    invalid.
    valid = read_choice(env, *erl_terms, "left", "
        right", (va_arg(valist, int*)));
    break;

case 'u':
    //Read unit: unitDiag, nonUnitDiag.
    //Set value to 1 if unitDiag; 0 if nonUnitDiag;
        otherwise returns invalid.
    valid = read_choice(env, *erl_terms, "unitDiag",
        "nonUnitDiag", (va_arg(valist, int*)));
    break;


default:
    //Unknown type... give an error.
    valid = 0;
    break;
```

```
        }
        erl_terms ++;
    }

    va_end(valist);
    return valid;
}

//Benchmark
ERL_NIF_TERM nif_max_list(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
    ERL_NIF_TERM head = argv[0];

    ERL_NIF_TERM elem;

    int current, max = 0;

    for(int i = 0; enif_get_list_cell(env, head, &elem, &head);)
        {
        enif_get_int(env, elem, &current);
        if(max < current){
            max = current;
        }
    }

    return enif_make_int(env, max);
}

ERL_NIF_TERM nif_max_matrix(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
    Matrix L;
    enif_get_matrix(env, argv[0], &L);
    int max = 0;

    for(int i = 0; i<L.n_cols * L.n_rows; i++){
        max = L.content[i] > max? L.content[i]:max;
    }

    return enif_make_int(env, max);
}

ERL_NIF_TERM nif_nok(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
    if(argc > 0)
        return enif_make_badarg(env);
```

```
    return atom_nok;
}


//@arg 0: List of Lists of numbers.
//@return: Matrix of dimension
ERL_NIF_TERM nif_matrix(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    unsigned n_rows, line_length, dest = 0, n_cols = -1;
    ERL_NIF_TERM list = argv[0], row, elem;
    Matrix m;

    //Reading incoming matrix.
    if(!enif_get_list_length(env, list, &n_rows) && n_rows > 0)
        return enif_make_badarg(env);

    for(int i = 0; enif_get_list_cell(env, list, &row, &list); i
        ++){
        if(!enif_get_list_length(env, row, &line_length))
            return enif_make_badarg(env);

        if(n_cols == -1){
            //Allocate binary, make matrix accessor.
            n_cols = line_length;
            m = matrix_alloc(n_rows,n_cols);
        }

        if(n_cols != line_length)
            return enif_make_badarg(env);

        for(int j = 0; enif_get_list_cell(env, row, &elem, &row)
            ; j++){
            if(!enif_get_double(env, elem, &m.content[dest])){
                int i;
                if(enif_get_int(env, elem, &i)){
                    m.content[dest] = (double) i;
                }
                else{
                    return enif_make_badarg(env);
                }
            }
            dest++;
        }
    }
```

```
    return matrix_to_erl(env, m);
}

#define PRECISION 10
//Arg0: a valid matrix.
//Returns: an atom representation of the input matrix.
ERL_NIF_TERM matrix_to_atom(ErlNifEnv *env, Matrix m){
    char *content = enif_alloc(sizeof(char)*((2*m.n_cols-1)*m.
        n_rows*PRECISION + m.n_rows*2 + 3));
    content[0] = '[';
    content[1] = '\0';
    char converted[PRECISION];

    for(int i=0; i<m.n_rows; i++){
        strcat(content, "[");
        for(int j = 0; j<m.n_cols; j++){
            snprintf(converted, PRECISION-1, "%.5lf", m.content[
                i*m.n_cols+j]);
            strcat(content, converted);
            if(j != m.n_cols-1)
                strcat(content, " ");
        }
        strcat(content, "]");
    }
    strcat(content, "]");

    ERL_NIF_TERM result = enif_make_atom(env, content);
    enif_free(content);
    return result;
}

//@arg 0: Matrix.
//@return Nothing.
ERL_NIF_TERM nif_matrix_print(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){

    Matrix m;
    if(!enif_get(env, argv, "m", &m))
        return enif_make_badarg(env);
    return matrix_to_atom(env, m);
}

//@arg 0: matrix.
//@arg 1: int, coord m: row
```

```
//@arg 2: int, coord n: col
//@return: double, at cord Matrix(m,n).
ERL_NIF_TERM nif_get(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
     ErlNifBinary bin;
     int m,n;
     Matrix matrix;

     if(!enif_get(env, argv, "iim", &m, &n, &matrix))
         return enif_make_badarg(env);
     m--, n--;

     if(m < 0 || m >= matrix.n_rows || n < 0 || n >= matrix.
         n_cols)
         return enif_make_badarg(env);

     int index = m*matrix.n_cols+n;
     return enif_make_double(env, matrix.content[index]);
}


//@arg 0: Array.
//@arg 1: Array.
//@return: true if arrays share content, false if they have
    different content || size..
ERL_NIF_TERM nif_eq(ErlNifEnv *env, int argc, const ERL_NIF_TERM
     argv[]){
     Matrix a,b;
     if(!enif_get(env, argv, "mm", &a, &b))
         return enif_make_badarg(env);

     //Compare number of columns and rows
     if((a.n_cols != b.n_cols || a.n_rows != b.n_rows))
         return atom_false;

     //Compare content of arrays
     if(!equal_ad(a.content, b.content, a.n_cols*a.n_rows))
         return atom_false;

     return atom_true;
}


//@arg 0: int.
//@arg 1: Array.
```

```
//@return: returns an array, containing requested row.
ERL_NIF_TERM nif_row(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
    int row_req;
    Matrix matrix;
    if(!enif_get(env, argv, "im", &row_req, &matrix))
        return enif_make_badarg(env);

    row_req--;
    if(row_req<0 || row_req >= matrix.n_rows)
        return enif_make_badarg(env);

    Matrix row = matrix_alloc(1, matrix.n_cols);
    memcpy(row.content, matrix.content + (row_req * matrix.
        n_cols), matrix.n_cols*sizeof(double));

    return matrix_to_erl(env, row);
}


//@arg 0: int.
//@arg 1: Array.
//@return: returns an array, containing requested col.
ERL_NIF_TERM nif_col(ErlNifEnv * env, int argc, const
    ERL_NIF_TERM argv[]){
    int col_req;
    Matrix matrix;
    if(!enif_get(env, argv, "im", &col_req, &matrix))
        return enif_make_badarg(env);

    col_req--;
    if(col_req<0 || col_req >= matrix.n_cols)
        return enif_make_badarg(env);


    Matrix col = matrix_alloc(matrix.n_rows, 1);

    for(int i = 0; i < matrix.n_rows; i++){
        col.content[i] = matrix.content[i * matrix.n_rows +
            col_req];
    }

    return matrix_to_erl(env, col);
}
```

```
//@arg 0: Matrix.
//@arg 1: Matrix.
//@return Matrix resulting of element wise + operation.
ERL_NIF_TERM nif_plus(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){

    Matrix a,b;
    if(!enif_get(env, argv, "mm", &a, &b))
        return enif_make_badarg(env);

    if ((a.n_cols != b.n_cols || a.n_rows != b.n_rows)){
        return enif_make_badarg(env);
    }

    Matrix result = matrix_alloc(a.n_rows, a.n_cols);

    for(int i = 0; i < a.n_cols*a.n_rows; i++){
            result.content[i] = a.content[i] + b.content[i];
    }

    return matrix_to_erl(env, result);
}

//@arg 0: Matrix.
//@arg 1: Matrix.
//@return Matrix resulting of element wise - operation.
ERL_NIF_TERM nif_minus(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){

    Matrix a,b;
    if(!enif_get(env, argv, "mm", &a, &b))
        return enif_make_badarg(env);

    if (a.n_cols != b.n_cols || a.n_rows != b.n_rows){
        return enif_make_badarg(env);
    }

    Matrix result = matrix_alloc(a.n_rows, a.n_cols);

    for(int i = 0; i < a.n_cols*a.n_rows; i++){
        result.content[i] = a.content[i] - b.content[i];
    }

    return matrix_to_erl(env, result);
```

```
}

//@arg 0: int.
//@arg 1: int.
//@return: empty Matrix of requested dimension..
ERL_NIF_TERM nif_zero(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    int m,n;
    if(!enif_get(env, argv, "ii", &m, &n))
        return enif_make_badarg(env);

    Matrix a = matrix_alloc(m,n);
    memset(a.content, 0, sizeof(double)*m*n);
    return matrix_to_erl(env, a);
}

//@arg 0: int.
//@arg 1: int.
//@return: empty matrix of dimension [arg 0, arg 1]..
ERL_NIF_TERM nif_eye(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    int m;
    if(!enif_get_int(env, argv[0], &m))
        return enif_make_badarg(env);

    if(m <= 0)
        return enif_make_badarg(env);

    Matrix a = matrix_alloc(m,m);
    memset(a.content, 0, sizeof(double)*m*m);
    for(int i = 0; i<m; i++){
        a.content[i*m+i] = 1.0;
    }
    return matrix_to_erl(env, a);
}

//Transpose of a matrix.
Matrix tr(Matrix a){
    Matrix result = matrix_alloc(a.n_cols ,a.n_rows);

    for(int j = 0; j < a.n_rows; j++){
        for(int i = 0; i < a.n_cols; i++){
            result.content[i*result.n_cols+j] = a.content[j*a.
                n_cols+i];
        }
```

```
    }
    return result;
}

//arg 0: double or int
//arg 1: Matrix
//@return the result of multiplying each matrix element by arg
    0.
ERL_NIF_TERM nif_mult_num(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){

    double a;
    Matrix b;
    if(!enif_get(env, argv, "nm", &a, &b))
        return enif_make_badarg(env);

    Matrix c = matrix_alloc(b.n_rows, b.n_cols);

    for(int i = 0; i<b.n_cols*b.n_rows; i++){
        c.content[i] = a * b.content[i];
    }

    return matrix_to_erl(env, c);
}

//@arg 0: Matrix.
//@arg 1: Matrix.
//@return Matrix resulting of multiplication.
ERL_NIF_TERM _nif_mult_matrix_tr(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){

    Matrix a,b;
    if(!enif_get(env, argv, "mm", &a, &b))
        return enif_make_badarg(env);

    int n_rows = a.n_rows;
    int n_cols = b.n_cols;

    if(a.n_cols != b.n_rows)
        return atom_nok;

    Matrix result = matrix_alloc(n_rows, n_cols);
    memset(result.content, 0.0, n_rows*n_cols * sizeof(double));
```

```
    Matrix b_tr = tr(b);

    for(int i = 0; i < n_rows; i++){
        for(int j = 0; j < n_cols; j++){
            for(int k = 0; k<a.n_cols; k++){
                result.content[j+i*result.n_cols] += a.content[k+
                    i*a.n_cols] * b_tr.content[k+j*b_tr.n_cols];
            }
        }
    }

    free_matrix(b_tr);

    return matrix_to_erl(env, result);
}


//@arg 0: Matrix.
//@arg 1: Matrix.
//@return Matrix resulting of multiplication.
ERL_NIF_TERM _nif_mult_matrix(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){

    Matrix a,b;
    if(!enif_get(env, argv, "mm", &a, &b))
        return enif_make_badarg(env);

    int n_rows = a.n_rows;
    int n_cols = b.n_cols;

    if(a.n_cols != b.n_rows)
        return atom_nok;

    Matrix result = matrix_alloc(n_rows, n_cols);
    memset(result.content, 0.0, n_rows*n_cols * sizeof(double));

    for(int i = 0; i < n_rows; i++){
        for(int k = 0; k<a.n_cols; k++){
            double val = a.content[i*a.n_cols + k];
            for(int j = 0; j < n_cols; j++){
                result.content[j+i*result.n_cols] += val * b.
                    content[k*b.n_cols + j];
            }
        }
    }
```

```
    return matrix_to_erl(env, result);
}

//@arg0: Matrix.
ERL_NIF_TERM nif_tr(ErlNifEnv *env, int argc, const ERL_NIF_TERM
    argv[]){

    Matrix a;
    if(!enif_get_matrix(env, argv[0], &a))
        return enif_make_badarg(env);

    return matrix_to_erl(env, tr(a));
}


//arg0: Matrix.
ERL_NIF_TERM nif_inv(ErlNifEnv *env, int argc, const
   ERL_NIF_TERM argv[]){

    Matrix a;
    if(!enif_get_matrix(env, argv[0], &a))
        return enif_make_badarg(env);

    if(a.n_cols != a.n_rows){
        return atom_nok;
    }
    int n_cols = 2*a.n_cols;

    double* gj = (double*) enif_alloc(n_cols*a.n_rows*sizeof(
       double));
    for(int i=0; i<a.n_rows; i++){
        memcpy(gj+i*n_cols, a.content+i*a.n_cols, sizeof(double)
            *a.n_cols);
        memset(gj+i*n_cols + a.n_cols, 0, sizeof(double)*a.
            n_cols);
        gj[i*n_cols + a.n_cols + i] = 1.0;
    }

    //Elimination de Gauss Jordan:
    //https://fr.wikipedia.org/wiki/%C3%89limination_de_Gauss-
       Jordan

    //Row of last found pivot
    int r = -1;
```

```
//j for all indexes of column
for(int j=0; j<a.n_cols; j++){

    //Find the row of the maximum in column j
    int pivot_row = -1;
    for(int cur_row=r; cur_row<a.n_rows; cur_row++){
        if(pivot_row<0 || fabs(gj[cur_row*n_cols + j]) >
            fabs(gj[pivot_row*n_cols+j])){
            pivot_row = cur_row;
        }
    }
    double pivot_value = gj[pivot_row*n_cols+j];

    if(pivot_value != 0){
        r++;
        for(int cur_col=0; cur_col<n_cols; cur_col++){
            gj[cur_col+pivot_row*n_cols] /= pivot_value;
        }
        gj[pivot_row*n_cols + j] = 1.0; //make up for
            rounding errors

        //Do we need to swap?
        if(pivot_row != r){
            for(int i = 0; i<n_cols; i++){
                double cpy = gj[pivot_row*n_cols+i];
                gj[pivot_row*n_cols+i]= gj[r*n_cols+i];
                gj[r*n_cols+i] = cpy;
            }
        }

        //We can simplify all the rows
        for(int i=0; i<a.n_rows; i++){
            if(i!=r){
                double factor = gj[i*n_cols+j];
                for(int col=0; col<n_cols; col++){
                    gj[col+i*n_cols] -= gj[col+r*n_cols]*
                        factor;
                }
                gj[i*n_cols+j] = 0.0;     //make up for
                    rounding errors
            }
        }
    }

}
```

```
    Matrix inv = matrix_alloc(a.n_rows, a.n_cols);
    for(int l=0; l<inv.n_rows; l++){
        int line_start = l*n_cols + a.n_cols;
        memcpy(inv.content + inv.n_cols*l, gj + line_start,
            sizeof(double)*inv.n_cols);
    }

    enif_free(gj);
    return matrix_to_erl(env, inv);
}

//Some CBLAS wrappers.


//Performs blas_ddot
//Input: two vectors (matrices containing either one row or one
    column).
ERL_NIF_TERM nif_ddot(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    Matrix x,y;
    int n;

    if(!enif_get(env, argv, "imm", &n, &x, &y)){
        return enif_make_badarg(env);
    }

    if(fmin(x.n_rows, x.n_cols) * fmin(y.n_rows, y.n_cols) != 1)
        {
        //We are not using vectors...
        return enif_make_badarg(env);
    }

    double result = cblas_ddot(n, x.content, 1, y.content, 1);

    return enif_make_double(env, result);
}


//Performs blas_daxpy
//Input: a number, vectors X and Y
//Output: a vector of same dimension as Y, containing alpha X +
    Y
ERL_NIF_TERM nif_daxpy(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
```

```
    Matrix x,y;
    int n;
    double alpha;

    if(!enif_get(env, argv, "inmm", &n, &alpha, &x, &y)){
        return enif_make_badarg(env);
    }

    if(fmin(x.n_rows, x.n_cols) * fmin(y.n_rows, y.n_cols) != 1)
        {
        //We are not using vectors...
        return enif_make_badarg(env);
    }

    Matrix ny = matrix_dup(y);

    cblas_daxpy(n, alpha, x.content, 1, ny.content, 1);

    return matrix_to_erl(env, ny);
}

// Arguments: alpha, A, x, beta, y
// Performs alpha*A*x + beta*y.
// alpha, beta are numbers
// A, x, y are matrices (x and y being vectors)
// x and y are expected to have a length of A.n_cols
ERL_NIF_TERM nif_dgemv(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    Matrix A,x,y;
    double alpha, beta;

    if(!enif_get(env, argv, "nmmmm", &alpha, &A, &x, &beta, &y))
        {
        enif_make_badarg(env);
    }

    //Check dimensions compatibility
    int vec_length = fmin(fmax(x.n_cols, x.n_rows), fmax(y.
        n_cols, y.n_rows));
    if(vec_length < A.n_cols || fmin(x.n_cols, x.n_rows) != 1 ||
        fmin(y.n_cols, y.n_rows) != 1){
        enif_make_badarg(env);
    }

    Matrix ny = matrix_dup(y);
```

```
    cblas_dgemv(CblasRowMajor, CblasNoTrans, A.n_rows, A.n_cols,
        alpha, A.content, A.n_rows, x.content, 1,  beta, ny.
       content, 1);

    return matrix_to_erl(env, ny);
}



//Arguments: double alpha, matrix A, matrix B, double beta,
    matrix C
ERL_NIF_TERM nif_dgemm(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    Matrix A, B, C;
    double alpha;
    double beta;

    if(!enif_get(env, argv, "nmmmm", &alpha, &A, &B, &beta, &C)
            || A.n_rows != C.n_rows
            || B.n_cols != C.n_cols
            || A.n_cols != B.n_rows){

        return enif_make_badarg(env);
    }

    Matrix nC = matrix_dup(C);

    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
    A.n_rows, B.n_cols, A.n_cols,
    alpha, A.content, A.n_cols, B.content, B.n_rows,
    beta, nC.content, nC.n_cols);

    return matrix_to_erl(env, nC);
}



//Arguments: A,B.
//Finds matrix X such that A*X = B.
ERL_NIF_TERM nif_dgesv(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    Matrix A,B;

    if(!enif_get(env, argv, "mm", &A, &B)
```

```
            || A.n_rows != A.n_cols
            || B.n_rows != A.n_rows){
        return enif_make_badarg(env);
    }

    int n = A.n_rows;
    Matrix nA = matrix_dup(A);
    Matrix nB = matrix_dup(B);
    int* ipiv = enif_alloc(sizeof(int)*n);

    int error = LAPACKE_dgesv(LAPACK_ROW_MAJOR, n, nB.n_cols, nA
        .content, n, ipiv, nB.content, nB.n_rows);

    if(!error){
        return matrix_to_erl(env, nB);
    }
    else if (error < 0){
        return enif_make_badarg(env);
    }
    else{
        return enif_make_atom(env, "Invalid LAPACKE argument.\n"
            );
    }
}

ERL_NIF_TERM nif_memleak(ErlNifEnv *env, int argc, const
    ERL_NIF_TERM argv[]){
    enif_alloc(sizeof(double));
    return atom_nok;
}

ErlNifFunc nif_funcs[] = {
    //Benchmark
    {"nif_max", 1, nif_max_list},
    {"nif_max_matrix", 1, nif_max_matrix},
    {"nif_nok", 0, nif_nok},

    //Memory leak
    {"memleak", 0, nif_memleak},

    //Actual library
    {"matrix", 1, nif_matrix},
    {"print", 1, nif_matrix_print},
    {"get", 3, nif_get},
    {"==", 2, nif_eq},
```

```
    {"row", 2, nif_row},
    {"col", 2, nif_col},
    {"+", 2, nif_plus},
    {"-", 2, nif_minus},
    {"zeros", 2, nif_zero},
    {"eye", 1, nif_eye},
    {"*_matrix", 2, _nif_mult_matrix},
    {"*tr", 2, _nif_mult_matrix_tr},
    {"*_num", 2, nif_mult_num},
    {"tr", 1, nif_tr},
    {"inv", 1, nif_inv},

    //--- BLAS----------
    {"ddot", 3, nif_ddot},
    {"daxpy", 4, nif_daxpy},
    {"dgemv", 5, nif_dgemv},
    {"dgemm", 5, nif_dgemm},
    {"dgesv", 2, nif_dgesv}
};

ERL_NIF_INIT(numerl, nif_funcs, load, NULL, NULL, NULL)
```

Appendix E

test_numerl.erl

```
-module(numerl_test).
-include_lib("eunit/include/eunit.hrl").

matrix_test() ->
    M0 = [[1.0, 0.0], [0.0, 1.0]],
    _ = numerl:matrix(M0).

print_test() ->
    M0 = numerl:matrix([[1.0/3.0, 0.0], [0.0, 1.0/3.0]]),
    numerl:print(M0).


get_test() ->
    %Testing access on square matrix
    M0 = [[1.0, 2.0], [3.0, 4.0]],
    CM0 = numerl:matrix(M0),
    V00 = mat:get(1,1,M0),
    V00 = numerl:get(1,1,CM0),

    V11 = mat:get(2,2,M0),
    V11 = numerl:get(2,2,CM0),

    V01 = mat:get(1,2,M0),
    V01 = numerl:get(1,2,CM0),

    V10 = mat:get(2,1,M0),
    V10 = numerl:get(2,1, CM0),

    M1 = [[1.0], [2.0]],
    CM1 = numerl:matrix(M1),
```

```
    W00 = mat:get(1, 1, M1),
    W00 = numerl:get(1, 1, CM1),
    W01 = mat:get(2, 1, M1),
    W01 = numerl:get(2, 1, CM1),

    M2 = [[1.0, 2.0]],
    CM2 = numerl:matrix(M2),
    X00 = mat:get(1,2,M2),
    X00 = numerl:get(1,2,CM2).


equal_test() ->
    M0 = [[1.0, 2.0], [3.0, 4.0]],
    M1 = [[1.0, 2.0]],
    M2 = [[1.0], [2.0]],
    CM0 = numerl:matrix(M0),
    CM1 = numerl:matrix(M1),
    CM2 = numerl:matrix(M2),
    true = numerl:'=='(CM1, CM1),
    true = numerl:'=='(CM0, CM0),
    false = numerl:'=='(CM1, CM2),
    false = numerl:'=='(CM0, CM2),
    false = numerl:'=='(CM0, CM1).

row_test() ->
    M0 = [[1.0, 2.0], [3.0, 4.0]],
    R0 = mat:row(2, M0),
    CM0 = numerl:matrix(M0),
    CR0 = numerl:matrix(R0),
    true = numerl:'=='(CR0, numerl:row(2, CM0)).


col_test() ->
    M0 = [[1.0, 2.0], [3.0, 4.0]],
    C0 = mat:col(2, M0),
    CM0 = numerl:matrix(M0),
    CC0 = numerl:matrix(C0),
    true = numerl:'=='(CC0, numerl:col(2, CM0)).

plus_test()->
     M0 = [[1.0, 2.0], [3.0, 4.0]],
     M1 = [[2.0, 4.0], [6.0, 8.0]],
     CM0 = numerl:matrix(M0),
     CM1 = numerl:matrix(M1),
     CM0p = numerl:'+'(CM0, CM0),
```

```
      true = numerl:'=='(CM1, CM0p).

minus_test()->
      M0 = [[1, 2], [3, 4]],
      M1 = [[0, 0], [0, 0]],
      CM0 = numerl:matrix(M0),
      CM1 = numerl:matrix(M1),
      CM0p = numerl:'-'(CM0, CM0),
      true = numerl:'=='(CM1, CM0p).

zero_test() ->
    M0 = mat:zeros(1,5),
    CM0 = numerl:zeros(1,5),
    G0 = float(mat:get(1, 5, M0)),
    G0 = numerl:get(1, 5, CM0).

eye_test() ->
    CM0 = numerl:eye(5),
    M0 = numerl:get(5,5,CM0),
    M0 = 1.0,
    M0 = numerl:get(1,1,CM0),
    M0 = numerl:get(2,2,CM0),
    M0 = numerl:get(4,4,CM0).

mult_num_test()->
    M0 = numerl:matrix([[1.0, 2.0]]),
    true = numerl:'=='(numerl:matrix([[2,4]]), numerl:'*'(2, M0)
        ),
    true = numerl:'=='(numerl:matrix([[0,0]]), numerl:'*'(0, M0)
        ),
    true = numerl:'=='(numerl:matrix([[-1, -2]]), numerl:'*'(-1,
         M0)).

mult_matrix_test() ->
    CM0 = numerl:eye(2),
    CM1 = numerl:matrix([[1.0, 2.0], [3.0, 4.0]]),
    CM3 = numerl:matrix([[1.0], [2.0]]),
    CM5 = numerl:matrix([[5.0], [11.0]]),
    CM4 = numerl:matrix([[1.0, 2.0]]),
    CM6 = numerl:matrix([[7.0, 10.0]]),
    true = numerl:'=='(CM1, numerl:'*'(CM1, CM0)),
    true = numerl:'=='(CM5, numerl:'*'(CM1, CM3)),
    true = numerl:'=='(CM6 ,numerl:'*'(CM4, CM1)).

mult_matrix_num_test() ->
```

```
    M0 = [[1.0, 2.0, 3.0]],
    CR0 = numerl:matrix(mat:'*'(2.0, M0)),
    CR = numerl:'*'(2.0, numerl:matrix(M0)),
    true = numerl:'=='(CR0, CR).



tr_test() ->
    CM0 = numerl:eye(2),
    CM0 = numerl:tr(CM0),
    CM1 = numerl:matrix([[1.0, 2.0],[3.0, 4.0]]),
    CM2 = numerl:matrix([[1.0, 3.0], [2.0, 4.0]]),
    CM3 = numerl:matrix([[1.0, 2.0]]),
    true = numerl:'=='(CM3, numerl:tr(numerl:matrix([[1.0], [2.0
        ]]))),
    CM1 = numerl:tr(CM2).

inv_test() ->
    M = numerl:matrix([[2.0, -1.0, 0.0], [-1.0, 2.0, -1.0], [
        0.0, -1.0, 2.0]]),
    M_inv = numerl:inv(M),
    io:fwrite(numerl:print(M_inv),[]),
    true = numerl:'=='(numerl:'*'(M, M_inv), numerl:eye(3)).

ddot_test() ->
    Incs = numerl:matrix([[1, 2, 3, 4]]),
    Ones = numerl:matrix([[1], [1], [1], [1]]),
    10.0 = numerl:ddot(4, Incs, Ones),
    30.0 = numerl:ddot(4, Incs, Incs),
    4.0 = numerl:ddot(4, Ones, Ones),
    1.0 = numerl:ddot(1, Incs, Ones).

daxpy_test()->
    Ones = numerl:matrix([[1, 1, 1, 1]]),
    Incs = numerl:matrix([[1, 2, 3, 4]]),
    true = numerl:'=='(numerl:matrix([[3, 4, 5, 6]]), numerl:
        daxpy(4, 2, Ones, Incs)).

dgemv_test()->
    V10 = numerl:matrix([[1,2]]),
    V01 = numerl:matrix([[0], [1]]),
    M = numerl:matrix([[1,2],[3,4]]),
    true = numerl:'=='(numerl:matrix([[10],[26]]), numerl:dgemv
        (2,M,V10, 4, V01)).
```

```
dgemm_test()->
    A = numerl:matrix([[1,2]]),
    B = numerl:matrix([[3,4], [5,6]]),
    C = numerl:matrix([[10, 12]]),
    true = numerl:'=='(numerl:matrix([[31, 38]]), numerl:dgemm
        (2,A,B,0.5,C)).

dgesv_test()->
    A = numerl:matrix([[1,2],[0,3]]),
    B = numerl:matrix([[2,4],[0,6]]),
    X = numerl:matrix([[2,0], [0,2]]),
    true = numerl:'=='(X, numerl:dgesv(A, B)).
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**