Solving Linear Systems With C

Matrix and Vector

By
Ismael Mosquera Rivera

To my wife.

Abstract

The subject covered in this article is, mainly, addressed to students interested in how to learn good programming practices using the C programming language. Anyway, it can be also useful to experimented C programmers, since the content is about an implementation suitable to solve linear systems in several different ways. In addition, the most common operations applyed to matrices and vectors are also presented.

This implementation does not use any propietary libraries, so, the code can be compiled under any platform since it uses only the standard headers which come with any C compiler, and the rest is pure C code.

The Vector and Matrix types are defined first; then, they are employed in the algorithms used to solve linear systems. The Vector and Matrix types also implement support to load and store Vectors and Matrices, resize them, and get chunks from them.

So, we thought that it was worth to write this paper for an interested reader who wants to learn some good C programming practices meanwhile learning also how to implement a code library covering Vector and Matrix operations and efficient algorithms to solve systems of linear equations. We hope that the audience interested in this subject can take benefit from this development.

Content

1 Overview **7**
2 Development **11**
3 Appendix **22**

1 Overview

In this chapter, we introduce some concepts about computer programming languages. We will not explain how to use them, but the basics to understand how and why they work. At the early days of computers, the first language which humans could understand was the assembler; this language is the closer to the machine, since it uses mnemonics to refer the instructions implemented in a microprocessor. Eben the assembler language was a very important advance on how to program computers, in most cases it is tedious and error prone; so, languages with higher level of abstraction were needed. Among them, the C programming language developed by Dennis Ritchie was broadly accepted as a high level programming language, and also a language being close to the machine; in fact, it is the language used in O.S. implementations.

Today, there are a large number of programming languages, most of them also broadly accepted to develop computational systems. So, we can think: Why do not use an unique multipurpose programming language instead learning about many ones?

The answer to that question is because each one can be better than the rest, depending on what we need to do.

If what we need is to perform system administration tasks, probably, we will use shell scripts; a good language to process regular expressions is Perl; to build a relational database and make queries to it, maybe the preferred language is SQL (Structured Query Language). Internet browsers normally integrate a JavaScript interpreter. Object oriented languages offer a great level of abstraction and a way to protect code to unwanted modifications; C++ is an OOP language which implements the tools needed to build large systems applying artifacts like Software Patterns suitable to solve complex problems in an elegant fashion. You can compile C code using a C++ compiler, but the inverse is not possible, that is, you cannot compile C++ code using a C compiler; that means that you can take benefit from C code in your C++ developments.

Java is also an OOP language which can be executed in any platform having the JVM (Java Virtual Machine) installed. Java language is compiled into an intermediate code called ‘bytecode’ placed in *.class files; then, that intermediate code can be executed by the particular JVM installed in a particular platform; that means that the bytecode generated by the fron- end of the Java compiler is suitable to run under any platform with its adequate back-end. The Java Development Kit (JDK) comes with the Java compiler, the Java Runtime Environment (JRE) among other tools; it also offers a really huge API (Application Program Interface) with a lot of Java Classes which implement functionality to make almost every task (I/O, Networking, UI, Generics, SQL, LDAP, RMI, SOUND, Streams, Internationalization, XML, Drawing ...) ready to use. Actually, Java is an interpreted language, so, it is not as fast as compiled languages like C and C++. Anyway, to fast execution, Java introduced a technology called Java Native Interface (JNI), with this technology, we can implement Java methods in C native code. To do that, we declare a method with the native reserved word and, with a tool named ‘javap’ a C header file is generated; this header file contains the proper signature for our C function implementation, afterwards, we compile a shared library and call that code from a native method. To illustrate this concept, following a method to compute the square root of a value is presented.

Java class with a native method:

```
public final class JMath
{
public native static double sqrt(double x);

static
{
System.loadLibrary("jmath");
}

private JMath(){}
}

// END
```

We can see that only the prototype of the native method is declared but not implemented. The static code block is executed just once and the loaded library has no extension, so, under windows it maps to jmath.dll and if we are under Linux it maps to jmath.so.

Now that we have the Jmath.java we can use the 'javap' tool to generate the C header file:

```
javac JMath.java
javap –s –p JMath.class

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JMath */

#ifndef _Included_JMath
#define _Included_JMath
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JMath
 * Method:    sqrt
 * Signature: (D)D
 */
JNIEXPORT jdouble JNICALL Java_JMath_sqrt
  (JNIEnv *, jclass, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

The C implementation file:

```
#include <jni.h>
#include <math.h>
#include "JMath.h"

JNIEXPORT jdouble JNICALL Java_JMath_sqrt
(JNIEnv *env, jclass cls, jdouble x)
{
return (jdouble)sqrt((double)x);
}
```

Now, we build a shared library, in this case under Microsoft Windows with the following batch file:

```
@echo off
echo Setting flags...
set FLAGS=-Wl,--add-stdcall-alias -Ic:\jni\include -shared -o
echo Building library...
gcc %FLAGS% jmath.dll JMath.c
echo done!
```

Now we can test our native library:

```
public class TestJMath
{
public static void main(String[] args)
{
System.out.println(JMath.sqrt(25.0));
}
}

// END
```

We compile the class for testing:

```
javac TestJMath.java
```

And execute it:

```
java TestJMath
```

output:

```
5.0
```

In fact, all what need a quiqck response is implemented in Java using JNI technology, for example the java.lang package to manage strins, I/O, networking, etc.

Another useful feature that Java implements is a ‘garbage collector’, that is a daemon running as an independent thread inside the JVM, which work is to release memory previously allocated for objects getting out of scope.

Java has a lot of functionality not mentioned here, Java Server Pages (JSP) and Servlets to program the web, where we can incrust Java code in HTML pages, and connectors to access databases, what they call Java Open DataBase Connection (JODBC technology). Since the first Java implementation, it has grown considerably, adding new features and functionality to the language, as can be Generics, what in C++ is called parametric classes or templates, support for functional programming through the use of Lambda expressions, Method invocation and Streams among others.

Python is also a really good interpreted language which we can do an extensive variety of applications, from system administration tasks to web programming, with capabilities to access databases and suitable to rapid prototypes implementation. It can be pre-compiled in a kind of bytecode as in Java (*.pyc files) or be interpreted statement by statement where the extension is then *.py.

There are much more programming languages and tools to use them as could be Integrated Development Environments (IDE) which simplifies the work in projects development. Our intent is not to mention all of them here, but we can say is that all of them have something in common: a grammar which describes them and, eventually, a mechanism to translate sentences from the grammar to machine instructions to execute under a concrete computer platform.

In this concrete implementation, our choice is the C programming language; we could tell that it is a strongly typed language and a weakly checked one; so, C programmers should take special care on memory release, array bounds checking, I/O or error management among other things, in order to avoid a fatal crash in their applications.

Following, we recommend some readings to learn about some programming languages.

Kernighan, Ritchie: The C programming Language 2nd ed, prentice Hall
Kyle Loudon: Mastering Algorithms With C, O'Reilly
Bjarne Stroustrup: The C++ Programming Language 4th ed, Prentice Hall

F. Codd: The Relational Model for Database Management, Addison-Wesley
Alan Beaulieu: Learning SQL 2nd ed, O'Reilly

Horstmann, Cornell: Core Java, Prentice Hall
Urma, Fusco, Mycroft: Java 8 in Action: Lambdas, Streams and Functional-Style Programming, Manning
Falkner, Servlets and JavaServer Pages: the J2EE Technology Web, Addison-Wesley

Flanagan: JavaScript: The Definitive Guide 6th ed, O'Reilly
Beazley: Python, Essential Reference 4th ed, Addison-Wesley
Freeman: The Definitive Guide to HTML 5, Apress
Skonnard, Gudgin: Essential XML, Addison-Wesley

2 Development

In this chapter, we are going to describe the implementation to solve linear equation systems by using different techniques. We also provide functions to do the most common operations applied to matrices and vectors; in addition, we added some functionality to load and store matrices and vectors and functions to resize and get chunks from vectors and matrices. Our intent is to present a good structured way to implement such a library, which can be also view as an example to know how to make another kind of implementation following the concepts explained here.

We first define the Vector and Matrix types, then, we use them as data types in the algorithms to solve linear systems. There is a book titled: Data Structures + Algorithms = Programs. In this project, that say will be confirmed.

To add support to load and store Vectors and Matrices from and to a file, we define a header: “numio.h” which declares some functions to help on these tasks; we could do that directly without the help from this header, but we, finally, decided to do it in that way. The other C headers for this implementation are:

vector.h
matrix.h
lu.h
lenearsys.h

In the following sections, we discuss their implementation taken in account that you also have the source code in order to follow the explanations introduced here. Intuitively, we can deduce what we are going to find in each of these headers, anyway, we introduce it briefly.

vector.h:
Functions to create, destroy, clone, load, store, resize, get chunks, print, get and set values, and some operations commonly applied to vectors: add, sub, mul and div by a value, module, normalize, dot and cross produc and some useful macros.

matrix.h:
Functions to create, destroy, clone, load, store, resize, get chunks, print, get and set values, and the most common operations applied to matrices: add, sub, mul, transpose, get identity, get a rotation matrix, get the determinant of a matrix, compute the inverse of a matrix and some useful macros.

lu.h:
This header has a function to compute the LU decomposition of a matrix and a function to print the LU to the console; it also define some useful macros.

linearsys.h:
Functions to solve systems of linear equations applying:

Cramer’s rule.
Gaussian elimination.
LU decomposition.

Vector Type

In our implementation, a vector is, basically, an array of double. There are several ways to declare it with C. We can declare a static array using a symbolic constant or a literal:

```
#define SIZE 10 /* symbolic constant */

double v[SIZE];
double x[10]; /* using a literal */
```

In most cases, we do not know the array size, but we cannot declare an array in C using a variable, for example:

```
int n = get_array_size();
double v[n]; /* not allowed */
```

To declare an array which size is not known yet, we can do it dynamically using the functions offered by the C compiler in the stdlib.h header; so, we can dinamically reserve memory:

```
int size = get_array_size();
double* v = (double*)malloc(size * sizeof(double));
```

and initialize it:

```
int i;
for(i = 0; i < size; i++) v[i] = 0.0;
```

In this way, we have our vector of doubles, but we lose its size unless we track the declared int constantly, which is really a nuissance. On the other hand,our vector is an identifier named ‘v’ which has been declared as a pointer to double, that give us not too much information about that, actually, it is a vector. So, we better define a Vector type:

```
typedef struct
{
int _size;
double* _data;
}Vector;
```

Now, we can implement a function to create Vector types:

```
Vector* create_vector(int size)
{
     int i;
Vector* v = (Vector*)malloc(sizeof(Vector));
v->_size = size;
v->_data = (double*)malloc(v->_size * sizeof(double));
for(i = 0; i < v->_size; i++) v->_data[i] = 0.0;
return v;
}
```

With this implementation, it is clear that we are working with vectors, at the time we keep the vector size in our declared Vector variables.

```
int n = get_vector_size();
Vector* v = create_vector(n);
printf("size = %d\n", v->_size);
```

Since an array identifier points to its first memory position, we also could access its elements using the notation:

```
int i;
for(i = 0; i < v->_size; i++) *(v->_data + i) = 0.0;
```

but we prefer to use v->_data[i], because we think it is clearer than the other one. There is another function to release previously allocated memory for a Vector:

```
void destroy_vector(Vector* v);
```

Which uses the free() function in its body. Although you can access vector elements directly using the notation mentioned above, there are functions to do that:

```
double get_vector(const Vector* v, int pos);
void set_vector(Vector* v, double value, int pos);
```

As we told before, functions to clone, resize, get a chunk from a vector, and functions to perform the most common operations with vectors are also available; please, consult the source code to know about them.

To load and store vectors we provide a well defined format to manage such a files:

```
#elements
x0 x1 x2 .. x#-1
```

Such a file could be the following content:

```
3
2.00 -1.25 5.00
```

So, to load a vector from a file, we first read the integer value and, afterwards, the double values which are the elements of the vector. To store a vector to a file, we first write the integer value, that is, the number of elements and, afterwards, the vector elements separated by spaces. Here are the function prototypes:

```
Vector* load_vector(const char* filename);
void store_vector(const Vector* v, const char* filename);
```

We also have a function to print a vector to the console in a well defined format; in this case, the vector values have a precision of just 2 decimals. For example, the vector loaded from the above archive will be printed as follows:

```
[2.00, -1.25, 5.00]
```

Every header file has comments explaining what does each function, so, we encouraged the reader to look at the header files to get more information.

Matrix Type

We can define a matrix in C as a bidimensional array of double; in that case, it is a static declaration, so, we need symbolic constants or literals, as we did in the case of a static vector declaration.

```
#define ROWS 3 /* symbolic constants for rows and columns */
#define COLUMNS 4

double m[ROWS][COLUMNS];
double m[3][4]; /* using literals */
```

and we can access its elements in a really intuitive way:

```
int i, j;
for(i= 0; i < ROWS; i++)
{
for(j = 0; j < COLUMNS; j++)
{
m[i][j] = 0.0;
}
}
```

but in most cases, we do not know, in advance, the number of rows and columns, then, we must define the array dynamically as we did with vectors:

```
int rows = 3;
int columns = 4;
int dimension = rows * columns;
double* m = (double*)malloc(dimension * sizeof(double)) ;
```

but in case of a matrix, we need to access its elements in a different fashion to be in the correct position; following we show that:

```
*(m + i*columns + j) = ...
m[i*columns+j] = ...
```

remember that the array identifier, in this case ‘m’, points to the first memory position in it. Anyway, we also lose the number of rows and columns unless we track their variables constantly, which can be really a nuissance. To solve that issue, we do it in the same way as in the case of the vector implementation, by a new type definition:

```
typedef struct
{
int _rows;
int _columns;
double* _data;
}Matrix;
```

Now, we can declare a function to create matrices without loosing any information related to them:

```
Matrix* create_matrix(int r, int c)
{
int i, j, dim;
Matrix* m = (Matrix*)malloc(sizeof(Matrix));
m->_rows = r;
m->_columns = c;
dim = m->_rows * m->_columns;
m->_data = (double*)malloc(dim*sizeof(double));
for(i = 0; i < m->_rows; i++)
{
for(j = 0; j < m->_columns; j++)
{
     *(m->_data + i*m->_columns + j) = 0.0;
}
}
return m;
}
```

Note that in this implementation, we used the notation:

```
*(m->_data + i*m->_columns + j)
```

but, normally we will use:

```
m->_data[i*m->_columns+j]
```

The data could be accessed directly but, there are functions to do that, and some macros too; Their prototypes are:

```
double get_matrix(const Matrix* m, int i, int j);
void set_matrix(Matrix* m, double value, int i, int j);

#define rows_matrix(matrix) ((matrix)->_rows)
#define columns_matrix(matrix) ((matrix)->_columns)
#define dimension_matrix(matrix) (rows_matrix(matrix) *
columns_matrix(matrix))
```

The header declares some functions, as in the case of vector, to destroy, clone, resize, resize only rows, resize only columns, get chunks, and functions to perform the most common operations applied to matrices: add, sub, mul, identity, transpose, determinant, inverse, rotation matrix; and also support to load and store matrices from archives with a well defined format, similar to the used with vectors. Following, we show the load_matrix prototype that you can find in the header file:

```
/*
* Loads a Matrix from a file.
* The format of the stored Matrix is well defined.
*
* #r #c
* a00 a01 a02 ... a0n
* a10 a11 a12 ... a1n
* ...
* am0 am1 am2 ... amn
*
* Where:
* #r => number of roes.
* #c => number of columns.
* a00..a0n
* a10..a1n
```

```
* ...
* am0..amn
* are the components of the Matrix.
* m => r-1
* n => c-1
*
* param: const char* filename => file where the Matrix is
stored.
*
* returns:
* A pointer to the loaded Matrix.
*/
Matrix* load_matrix(const char* filename);
```

As you can see, the functions are documented with comments explaining what they do, and they are well commented; that is why encouraged the reader to look at the source code.

The content of such a file could be, for instance:

```
3 4
2.0 1.0 1.0 5.0
4.0 -6.0 0.0 -2.0
-2.0 7.0 2.0 9.0
```

where the two integers in the first row are the number of rows and columns of the matrix, respectively, and the three followed rows, each one with four columns, are the components of the matrix.

To store a matrix, there is a function which writes a particular matrix in the presented format. To print a matrix to the console, the format is also similar than with vectors; the content presented above will be printed to the console in the form:

```
[2.00, 1.00, 1.00, 5.00]
[4.00, -6.00, 0.00, -2.00]
[-2.00, 7.00, 2.00, 9.00]
```

We can, for example, solve overdetermined systems, multiplying by the transposed matrix the left and right side to form an equivalent system, but with the same number of rows and columns for the coefficient matrix. To know the implementation details for each function, consult the code in the src folder.

LU Type

When we apply a LU decomposition to a matrix, we get a L (lower) matrix, and a U (upper) matrix. The lower and upper matrices have the same number of rows and columns than the original, which must be a square matrix. The lower matrix has 0s up its diagonal and the upper matrix has 0s down its diagonal. We also get what is called a ‘permutation vector’ to manage possible row permutations during a LU decomposition.

To illustrate it, we are going to show how to perform a LU decomposition of a matrix previously loaded from a file. Suppose that you have a matrix in a file which content is:

```
3 3
2.0 1.0 1.0
4.0 -6.0 0.0
-2.0 7.0 2.0
```

and you want to get its LU decomposition; if the name of this file is, for instance, mymatrix.dat, we can do it using the functions declared in the lu.h header:

```
Matrix* m = load_matrix("mymatrix.dat");
LU* lu = lu_decomposition(m);
```

and print the result to check if all was alright:

```
print_lu(lu);
```

output:

```
lower:
[1.00, 0.00, 0.00]
[0.50, 1.00, 0.00]
[-0.50, 1.00, 1.00]

upper:
[4.00, -6.00, 0.00]
[0.00, 4.00, 1.00]
[0.00, 0.00, 1.00]

permutation:
[1, 0, 2]
```

Notice that in the LU decomposition process, the rows 0 and 1 were permuted. This numerical analisys skill is used to solve NxN linear systems, and to do that, we need to know if there were some permutations in order to compute the solution of the concrete system of equations. We will retrieve that in the section devoted to solve linear systems.

We have also a function to destroy a LU:

```
void destroy_lu(LU* lu);
```

as you might already have guessed, our LU type definition is as follows:

```
typedef struct
{
int* _permutation;
Matrix* _lower;
Matrix* _upper;
}LU;
```

and some helper macros to access its members:

```
#define lu_permutation(lu) ((lu)->_permutation)
#define lu_lower(lu) ((lu)->_lower)
#define lu_upper(lu) ((lu)->_upper)
```

Solve Linear Systems

In this section, three different techniques to solve linear systems are presented:

```
Cramer's rule
Gaussian elimination
LU decomposition
```

Cramer's rule: This method uses determinants to get the solution vector from a concrete equation's system. We know that there is a function to compute the determinant of a matrix declared in the matrix.h header, and implemented in the matrix.c file. So, we can apply it to solve a linear system by using the Cramer's rule. Suppose that we have a linear system:

```
a00 a01 a02 ... a0n = c0
a10 a11 a12 ... a1n = c1
a20 a21 a22 ... a2n = c2
...
...
an0 an1 an2 ... ann = cn
```

To solve such a system, we first compute the determinant of the coefficient's matrix on the left side; if det(A) != 0, then we can get a solution; otherwise, the system cannot be solved, since the determinant equals to 0.

```
D = det(A)
```

once we have the value for that determinant, what we have to do is to substitute the column on the right side (c0 c1 c2 ... cn) on each coefficient's matrix column, solve the current determinant after sustitution and divide it by the determinant D which we computed first. If we name:

```
d0 d1 d2 ... dn
```

the values for every deterninant, then the X solution vector will be:

```
x0 = d0 / D
x1 = d1 / D
x2 = d2 /D
...
...
xn = dn /D
```

The Cramer's rule is ok to solve small systems, that is, composed by a small number of equations; to solve large systems it can be computationally expensive; in that case, it is best to apply another method as we will see.

Gaussian elimination: This method consists on finding 0s in the lower triangle of the coefficient's matrix extended with the vector of independent terms.

To find 0s in the lower triangle of the matrix:
the value of the i-th row, j-th column corresponding to the pivot is always multiplied by the value of the coefficient of the i-th row+1 divided by the value of the

coefficient of the i-th row changed sign by adding then this ith row to the ith row+1. This process is repeated until the corresponding Upper matrix of coefficients and the corresponding extension of independent terms are obtained. During that process, row permutations may be performed, what is called ‘pivoting’. Suppose that we have a system matrix as follows:

```
a00 a01 a02 ... a0n c0
a10 a11 a12 ... a1n c1
a20 a21 a22 ... a2n c2
...
...
an0 an1 an2 ... ann cn
```

after applying Gaussian elimination, we obtain the following triangular system:

```
a’00 a’01 a’02 ... a’0n c’0
0.00 a’11 a’12 ... a’1n c’1
0.00 0.00 a’22 ... a’2n c’2
...
...
0.00 0.00 0.00 ... a’nn c’n
```

once we have such a triangular system, its solution is straightforward; just begin computing the xn value:

```
xn = c’n / a’nn
```

and continue up until x0 substituing the x-th corresponding computed value in the i-th iteration. So, we eventually get the solution vector x0 x1 x2 ... xn.

LU decomposition: As we have seen, Gaussian elimination is an efficient way to solve a NxN system but, there are situations where another method is even more efficient than it. Suppose that we need to solve N systems of linear equations where the coefficient’s matrix is always the same, and only the independent terms change. In that scenary, using Gaussian elimination we must compute the triangular system for every solution. There is a skill suitable to improve this issue: LU decomposition. Suppose that we have a square matrix:

```
a00 a01 a02 ... a0n
a10 a11 a12 ... a1n
a20 a21 a22 ... a2n
...
...
an0 an1 an2 ... ann
```

after applying LU decomposition on it, we get:

```
Lower:
1.00 0.00 0.00 ... 0.00
a’01 1.00 0.00 ... 0.00
a’20 a’21 1.00 ... 0.0
...
...
a’n0 a’n1 a’n2 ... 1.00
```

notice that the lower matrix has always all of 1s in its diagonal.

```
Upper:
a’00 a’01 a’02 ... a’0n
0.00 a’11 a’12 ... a’1n
0.00 0.00 a’22 ... a’2n
...
...
0.00 0.00 0.00 ... a’nn

Permutation’s vector:
P0 p1 p2 ... pn
```

now that we have a LU decomposition, to solve a system for a concrete vector of independent terms, first we check if there was any permutation, in case it was some, we have to handle it in order to put the values for the independent terms in the correct position; then, we solve the lower triangular system for the handled vector of independent terms, and the resulting vector is used to solve the upper triangular system; so, we finally get the vector of solutions.

To illustrate that in practice, we are going to apply this skill to compute the inverse matrix for a square one of order 3.

```
Matrix of coefficients:
[2.00, 1.00, 1.00]
[4.00, -6.00, 0.00]
[-2.00, 7.00, 2.00]
```

after LU decomposition:

```
lower:
[1.00, 0.00, 0.00]
[0.50, 1.00, 0.00]
[-0.50, 1.00, 1.00]

upper:
[4.00, -6.00, 0.00]
[0.00, 4.00, 1.00]
[0.00, 0.00, 1.00]

permutation:
[1, 0, 2]
```

to compute the inverse, we have to solve 3 systems, one for each vector of independent terms which are:

```
1.0 0.0 0.0 | 0.0 1.0 0.0 | 0.0 0.0 1.0
```

each solution vector is the i-th column of the inverse matrix; so, after we have solved all of three, we get the matrix:

```
[0.75, -0.31, -0.38]
[0.50, -0.38, -0.25]
[-1.00, 1.00, 1.00]
```

which is the computed inverse matrix; to check if it actually is correct, we multiply the original matrix by its inverse to get the identity.

```
Matrix* iden = mul_matrix(m, m_inv) ;
print_matrix(iden) ;
```

output :

```
 [1.00, 0.00, 0.00]
[0.00, 1.00, 0.00]
[0.00, 0.00, 1.00]
```

3 Appendix

In this brief appendix, we are going to explain how to build the linearsys.dll library and the example to test it. In this development, we use the MinGW (Minimalist GNU for Windows) compiler and the GNU make utility to build the code, but since we do not use any third party code, just the standard headers which come with any C compiler, you can adapt the building process to compile the code under any platform having a C compiler.

If you are under Linux, just change linearsys.dll to linearsys.so in the Makefile used to build the dll placed in the first directory level. To compile the shared library just type:

```
make
```

and the linearsys.dll library will be placed inside the 'lib' folder; once you have successfully build that library, you can compile the example; go to the example folder and type:

```
make
```

To compile the example which will be moved to the bin folder. To execute the example program, use the run.bat script if you are in a Microsoft Windows command line, or run.sh if you are using a Linux like shell. When you run the example program, a file named 'out.txt' will be created in the bin folder, that is the output generated by the example program. We have to say that the example program to test the library only check for solving linear systems, but not for the functionallity offered by the vector.h and the matrix.h headers; anyway, all functions were tested successfully, you have to believe it!

About the author

Ismael Mosquera Rivera has a degree in Computer Science by Pompeu Fabra University http://www.upf.edu, Barcelona – Spain, 2004. He also was involved in the Clam project, http://www.clam-project.org (C++ Library for Audio and Music); unfortunately, he is affected by a severe eye disease (Retinitis Pigmenthosa), which caused him to get totally blind a number of years ago. Anyway, he finally decided to get back to the computer world, so, he installed a screen reader in his old computer. The one he installed is only available for Windows OS, but he got the MinGW compiler which has also a command line shell from where you can have a little Linux in your Windows system. So, all the textual information in a computer can be read, and he began to do research. Among the things he has done until that, the development of this little package encouraged him to write a paper explaining how it was done, and this is the result. He apologizes the reader to the lack of graphics, diagrams and identation in the source code. Anyway, his expectation is that an interested reader can take benefit from this article.