**LABORATORIO #3**

* [Introducción](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#Introduccion)
* [Matrices de prueba](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#MATRICES-DE-PRUEBA)
* [Los sistemas lineales](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#SISTEMA-LINEAL)
* [Determinante](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#DETERMINANTE)
* [La matriz inversa](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#MATRIZ-INVERSA)
* [Eliminación Gaussiana](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#ELIMINACION-GAUSS)
* [Factores PLU](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#FACTORES-DE-PLU)
* [Solución por PLU](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#SOLUCION-PLU)
* [Multiplicación Matricial](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#MULTIPLICACION-MATRICIAL)
* [Para hacer en el laboratorio](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-3#ASGNACION)

**Introducción**

Se comenzará el tópico fundamental en análisis numérico, que es álgebra lineal numérica. Se asume que está familiarizado con los conceptos de vectores y matrices y además, sus operaciones: determinantes, inversas y teoría para la solución de sistemas de ecuaciones.

Hay tres tipos de matrices de prueba que se utilizarán con los algoritmos.  Se establecerá en problema de sistemas lineales y se procederá a resolverlos por tres métodos: utilizando el determinante, la inversa o la factorización de Gauss.

**Matriz de prueba**

Ahora se considerará un ejemplo sencillo para estudiar el comportamiento de los algoritmos que se utilizan para resolver un sistema lineal de ecuaciones. Con estos problemas se verificará si se ha programado bien, y aún más importante, se los utilizará para estudiar la exactitud de los algoritmos. Se utilizarán problemas de cualeuier tamaño, en los cuales la solución exacta del determinante, inversa y/o valores característicos sevayan a determinar.

Estas son las tres matrices escogidas:

* Matriz de las segundas diferencias,
* Matriz de Frank,
* Matriz de Hilbert.

La matriz de la segunda siferencia se obtiene cuando se calcula la aproximación de la derivada de datos igualmente espaciados. La fórmula es:

* A(I,I) = -2
* A(I,I-1) = A(I,I+1) = 1, donde lestas columnas existan.
* A(I,J) = 0, de otra manera.

Ejemplo:

-2 1 0 0 0  
 1 -2 1 0 0  
 0 1 -2 1 0  
 0 0 1 -2 1  
 0 0 0 1 -2

La matriz es definida positivamente, simétrica, y tridiagonal. El determinante es más o menos **N+1**.

La fila I de la matriz Frank tiene como fórmula:

* A(I,J) = 0, para J = 1, ..., I-2;
* A(I,J) = N+1-I para J = I-1;
* A(I,J) = N+1-J para J = I, I+1, ..., N.

Ejemplo:

5 4 3 2 1  
 4 4 3 2 1  
 . 3 3 2 1  
 . . 2 2 1  
 . . . 1 1

El determinante de la matriz de Frank es 1, pero se hace dificil de calcular correctamente.

La matriz de Hilbert viene de realizar la aproximanción polinomial en el intervalo [0,1]. La fórmula es:

* A(I,J) = 1 / ( I + J - 1 )

Ejemplo:

1/1 1/2 1/3 1/4 1/5  
 1/2 1/3 1/4 1/5 1/6  
 1/3 1/4 1/5 1/6 1/7  
 1/4 1/5 1/6 1/7 1/8  
 1/5 1/6 1/7 1/8 1/9

Los coeficientes polinomiales **c** que representan a la función **f** se encuentran al calcular la solución del sistemas de ecuaciones definidas por los coeficientes de **A** con el vector de valores independientes **b**. El valor (I,J) en la matriz es la integral del polinomio **xi+j-2** mientras que el lado derecho **b(i)** es la integral de **f(x)\*xi-1**. Teóricamente, es perfectamente razonable asumir esto, pero llevaría  a un error.

**Archivos M** - Copie los archivos *dif2.m*, *frank.m* y *hilbert.m* Genere en cada archivo matrices de orden 5 para utilizar como prueba.

**Los sistemas lineales**

El problema de sistemas lineales se puede definir como sigue:

*Dada una matriz* ***M*** *por* ***N******llamada A*** *y un vector columna* ***M*** *llamado* ***b****,*   
*encuentre el vector* ***columna N******llamado x*** *para que* ***A \* x = b****.*

Si se va a resolver el problema de forma regular, se disponen de varios algoritmos, incluyendo la utilización de determinantes, la inversa de la matriz, *la eliminación de Gauss-Jordan y la factorización de factorization*. A continuación se verán estos y se descubrirá que hay buenas razones para preferir ela factorización de Gauss.

Se tomarán en cuenta los siguientes aspectos:

* *Eficiencia*: ¿Qué algoritmos producen los resultados con menor trabajo?
* Exactitud: ¿Qué algoritmos producen la respuesta que wsea más cercana?
* *Dificultad*: ¿Qué hace de un problema imposible o dificil de resolver?
* *Casos especiales*: ¿Cómo se resuelven problemas muy grandes? (Simétricos, de banda, singular, rectangular?

**El determinante**

Un resultado clasico del álgebra lineal establece:

Un sistema de ecuaciones lineales tiene solución si el   
determinante de la matriz de coeficientes no es cero. En este caso  
la solución para **x** se puede expresar en función de una fórmula  
en donde aparezcan los determinantes de las matrices relacionadas con  
**A**.

**Ejercicio**: copie el archivo *determinante.m*  de la página web. Esta función regresa el determinante de la matriz, utilizando la definición. Utilice el comando tic y toc de MATLAB para contabilizar el tiempo de cálculo.

tic  
**determinante(dif2(N))**   
**toc**

Para calcular el trabajo, tome la diferencia de tiempo para cada operación.

N DETERMINANTE(DIF2(N)) TRABAJO   
 (tiempo)  
  
 10 \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 100 \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 1000 \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 5000 \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

A medida que **N** se incrementa, como se incrementa el valor de **WORK**? Si esta toma un segundo para obtener el **determinante(dif2(10))**, entonces cuanto le tomaría realizar el cálculo de **determinant(dif2(20))**? 100 segundos? 10,000 segundos? 30,000,000 segundos (= 1 año)?

**Ejercicio**: MATLAB tiene un comando llamado **det** el cual computa la aproximación numérica del determinante de una matriz. Repita el ejercicio utilizando este comando.

N DETERMINANTE TRABAJO  
 (DIF2(N)) (operaciones)  
 1 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 2 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 4 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 8 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 16 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 32 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 64 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Los valores obtenidos con **det** y **determinant** concuerdan? Si duplicamos el valor de **N**, qué pasa con la cantidad de trabajo a realizar? Suponga que toma un segundo realizar un determinante de **N=10**. Cuanto tiempo Ud. cree que tomaría realizar esta operación para un determinante de **N=20**? 100 segundos? 10,000 segundos? 30,000,000 segundos?

The results from the **determinant** function should convince you that we can't work with the determinant as defined. You might think that we could use the MATLAB **det** function instead. But MATLAB uses Gauss factorization to get the determinant, and that's where we're actually going to end up.

We won't even *begin* to consider the nasty formulas that are required to solve a linear system using determinants. Instead, we will now move on to the next classic (and flawed) method.

**La matriz inversa**

Another classic result from linear algebra is the following:

The linear system problem is uniquely solvable if and only if the inverse  
matrix **A-1** exists. In this case the solution **x** can be expressed as  **x=A-1\*b**.

This result is even nicer than the one involving determinants. Consider that once we have computed the inverse matrix, we can solve as many linear systems involving the matrix **A** as we want, simply by multiplying. If we had used determinants, we'd have to start our computation all over again.

But what's the catch? Well, in particular, the way you are usually taught to find an inverse is *by computing determinants*. And that's not an option. However, let's suppose that we actually had a formula for the inverse matrix. (This is a completely unrealistic idea!). In that case, we could go ahead and solve the linear system. For our three test matrices, we happen to to have such a formula.

**Exercise** - Copy the M files *dif2\_inv.m*, *frank\_inv.m* and *hilbert\_inv.m*. These compute the inverse of each matrix, *exactly*. Test the use of the inverse to solve a linear system by setting up a 10 by 10 matrix, multiplying it by the vector [1,2,...,10] to get the right hand side. I've written the right hand sides of the statements you will need, to get you started:

**A = \_\_\_\_\_\_\_\_\_\_  
 x = \_\_\_\_\_\_\_\_\_\_  
 b = \_\_\_\_\_\_\_\_\_\_  
 Ainv = \_\_\_\_\_\_\_\_\_\_  
 x\_sol = \_\_\_\_\_\_\_\_\_\_  
 max\_err = \_\_\_\_\_\_\_\_\_\_**

Repeat for a larger **N** and for the Frank and Hilbert matrices.

Matrix N Maximum Absolute Error  
  
 DIF2 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_  
 FRANK 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_  
 HILBERT 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_

These results look pretty good, until we get to the Hilbert matrix, which we will come to see as the "bad boy" of linear algebra. You can pretty much expect every linear solution algorithm to fail on this matrix unless you are extremely careful. So we won't criticize the inverse matrix technique on the basis of the bad results for the Hilbert matrix.

But in the real world, we are very unlikely to have an exact formula for the inverse of a matrix. It is possible to compute the inverse, or more properly, to approximate it. However, this exposes us to an additional source of errors. (We already know that the Hilbert matrix was hopeless when we had the exact inverse, so we'll worry now about what happens with the other two.) Presumably, the errors we make in computing the inverse will then be reflected as additional errors in the solution of our linear system. Let's get a feeling for how worse this might make things.

**Exercise** - the MATLAB command **inv(A)** computes an approximation for the inverse of a matrix **A**. Let's repeat the previous experiment, but now using the approximate inverse instead of the exact one.

Matrix N Maximum Absolute Error  
  
 DIF2 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_  
 FRANK 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_  
 HILBERT 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_

Particularly for the larger problems, you should start to see significant errors cropping up. Watch out for cases where MATLAB prints a vector using scientific notation. A large multiplier like **1.0e+03** may appear in front of a vector of numbers that look small by themselves!

**Eliminación Gaussiana**

It is now time to talk about the standard algorithm for solving linear systems, usually called "Gauss elimination". I really don't want to belabor this idea. The details are excruciatingly boring, although it is actually a very tricky algorithm to code correctly. Yet we all learned it in high school when our minds were surely on other things!

Just as an extremely simple example, think about solving the problem

[ 1 9 ] [ x1 ] [ 19 ]  
 [ 2 4 ] [ x2 ] = [ 10 ]

First, we will prefer to think of the process as having two separable parts, the *factorization* and *back substitution* steps. The important things to recall include:

* in the **K-th** step of factorization, one of the 'free' equations is chosen to become the *pivot equation*, to be associated with **xK**.
* the pivot equation is used to eliminate references to **xK** in the remaining free equations.
* after **N-1** steps of factorization, back substitution begins. The last equation involves a single variable, and can be solved for **xN**. But then equation **N-1** can be solved for **xN-1**, because **xN** is now known, and similarly, the other equations are solved in backwards order.

The simplest version of Gauss factorization is called *Gauss factorization with no pivoting*, because it has a very simple way of choosing the pivot equation. The first equation is always associated with the first variable, and so on. If on step **K**, the coefficient of **xK** is zero in equation **K**, this method will fail, even though a solution may be exist.

The method of *Gauss factorization with partial pivoting* chooses the pivot equation as the free equation with largest coefficient for **xK**. To keep the calculation orderly, this pivot row is actually moved into the **K-th** row of the matrix. In this case, the matrix gradually is transformed into upper triangular shape. For exact arithmetic, if there is a solution, this method is guaranteed to reach it. Moreover, for inexact arithmetic, this method has better accuracy properties.

On step **K** we actually can choose not only which equation we want to work with, but which variable we want to eliminate. The method of *Gauss factorization with complete pivoting* chooses the coefficient **AI,J** of largest absolute magnitude in the free equations, associates equation **I** with variable **J** and then eliminates **xJ** from the other free equations. This method has more bookkeeping than the partial pivoting method, and doesn't produce much more improvement, so it is little used.

**Exercise** - to start with, let's treat Gauss factorization and back substitution as a complete "black box". In that case, to solve the linear system **A\*x=b**, we simply type the MATLAB command

**x = A \ b**

Try setting up and solving the difference, Frank and Hilbert systems using this approach.

Matrix N Maximum Absolute Error  
  
 DIF2 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_  
 FRANK 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_  
 HILBERT 10 \_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_

This is the third time we've solved the problem. We probably can't get a smaller error than with the exact inverse. But did we do significantly better or worse than when we used the *approximate* inverse?

*(I'd expect you to do a little better this time, actually. MATLAB computes the approximate inverse by PLU factorization and solution; then we multiply the inverse times the right hand side. It's usually faster and more accurate to compute the solution directly!)*

**Factores PLU**

The first step of Gauss elimination actually can be viewed as factoring the original matrix **A** into the form:

**A = P \* L \* U**

where

* **P** is a permutation matrix;
* **L** is a unit lower triangular matrix;
* **U** is an upper triangular matrix;

Then backward substitution step of Gauss elimination can be thought of as using this factor information to "peel away" the multiplication a step at a time:

**P \* L \* U \* x = b  
 L \* U \* x = P-1 \* b  
 U \* x = L-1 \* P-1 \* b  
 x = U-1 \* L-1 \* P-1 \* b**

However, instead of explicitly computing the inverses of the matrices, we use special facts about their form. It's not the inverse matrix itself that we want, but the inverse of the right hand side.

**Hand calculation**: For our extremely simple example, the PLU factorization is:

[ 1 9 ] [ 0 1 ] [ 1 0 ] [ 2 4 ]  
 [ 2 4 ] = [ 1 0 ] \* [ 1/2 1 ] \* [ 0 7 ]

Look over the right hand side and verify that the three factors have the right form. Multiply P\*L\*U and verify that the result is **A**. *(Usually, of course, the factors are much uglier than this!)*

**Exercise**: Copy the file *ge\_pp.m* from the web page. It computes a factorization of a matrix by carrying out Gauss elimination with partial pivoting. To try the code out, carry out the following steps:

1. Set **A** to be the difference matrix of size 5;
2. Use **ge\_pp** to factor **A** into matrices **P**, **L** and **U**;
3. Look at each of the three matrices and notice their properties;
4. Verify that **A=P\*L\*U**;

You might wonder why this routine stops at factorization. We will see shortly that we can use the PLU factorization for many different tasks, not just for a single linear solve.

**Solución PLU**

We have labored hard to determine the PLU factorization of a matrix **A**. It is time to make this effort pay off. We are going to design and construct an algorithm for solving the linear system problem **A\*x=b** assuming we have already computed the PLU factorization of **A**.

Let's consider the factored linear system once more. Just like in the case of systems of ODE's, it helps to make up some names for variables and look at the problem in a different way:

**P \* ( L \* U \* x ) = b   
 P \* z = b  
  
 L \* ( U \* x ) = P-1 \* b  
 L \* y = z  
  
 U \* x = L-1 \* P-1 \* b  
 U \* x = y  
  
 x = U-1 \* L-1 \* P-1 \* b**

Notice that all I have done is to use parentheses to group factors, and name them. But you should now be able to see what an algorithm for solving this problem might look like.

*PLU Solution Algorithm*: To solve **A\*x=b** given factors **P**, **L**, **U** and right hand side **b**,

1. Solve **P \* z = b**;
2. Solve **L \* y = z**;
3. Solve **U \* x = y**.

Ouch! I promised a simple algorithm, but now instead of having to solve one system, we have to solve three, and we have three different solution vectors running around. But I promise you, things actually have gotten better, because these are really simple systems to solve:

* The solution of **P \* z = b** is **z = P' \* b**;
* It's easy to solve **L \* y = z**, starting with row 1;
* It's almost as easy to solve **U \* x = y**, starting with row **N**.

**Hand calculation**: For our extremely simple example, the PLU factorization is

[ 1 9 ] [ 0 1 ] [ 1 0 ] [ 2 4 ]  
 [ 2 4 ] = [ 1 0 ] \* [ 1/2 1 ] \* [ 0 7 ]

Try going through the steps of solving the system, with right hand side [19, 10]', using this factorization:

Step  
 0 B = [ 19, 10 ]  
 1 Z = [ \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ ]  
 2 Y = [ \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ ]  
 3 X = [ \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ ]

Writing down the solution steps may help you with your program assignment.

**Multiplicación Matricial**

If you're comfortable handling vectors and matrices in programs, you can skip ahead to the assignment. Otherwise, let's pause for a moment and try to write a program to compute the product of a matrix and a vector, to get comfortable with a few ideas.

To start with, our function will be stored in the file *matvec.m* and have the form:

**function b = matvec ( A, x )**

computing the product of **A\*x**

If I'm given a matrix **A**, I can ask MATLAB to tell me its dimensions with the **size** command:

**[ m, n ] = size ( A )**

Our result vector **b** should have **m** rows and and 1 column. To zero it out, we could use a **for** loop, but a better way, which ensures that it has the right shape as well is:

**b = zeros ( m, 1 )**

We think of the index **i** as associated with rows. Each element **b(i)** is defined as the sum over **j** of **A(i,j) \* x(j)**. So this code is straightforward, except I won't actually write the punchline out:

**for i = 1 : m  
 for j = 1 : n  
 b(i) = ?  
 end  
 end**

**Exercise** - Try to put together this routine and compute the product of the Frank matrix with the vector of all 1's. It should be easy to check your work using MATLAB.

If you'd like some more practice, try to write the related routine *matmat.m* which allows you to compute **C=A\*B**, for the matrices **A** and **B** of order **LxM** and **MxN** respectively. Again, it's easy to check your results by using MATLAB.

**Para hacer en el laboratorio**

**Asignación** - Escriba una rutina sol\_plu*.m*, para resolver el sistema lineal **A\*x=b** por medio de PLU. La rutina debe de tener la forma:

**function x = sol\_plu ( P, L, U, b )**

**Sujerencias para el diseño**: - Para comenzar con la codificación, establezca tres seciones del para el código con lo siguiente:

1. % Solución **P\*z=b**
2. % Solución **L\*y=z**
3. % Solución **U\*x=y**

Ahora continúe con la escritura del código para cada sección:

1. Solving the permutation matrix system is easy, because **P** is orthogonal, hence its inverse...is its transpose! So we can use the fact that the solution of a linear system can be computed by multiplying the right hand side by the inverse.
2. To solve **L\*y=z**, write something like:

y = zeros ( n, 1 )  
 for j = 1 : n  
 y(j) = ? ...*Solve equation* **j**  
 for i = j+1 : n  
 z(i) = z(i) - ? ...*Update the right hand sides*  
 end  
 end

You may use the fact that the diagonal entries of **L** are 1.

1. To solve **U\*x=y**, write something like:

x = zeros ( n, 1 )  
 for j = n : -1 : 1  
 x(j) = ? ...*Solve equation* **j**  
 for i = 1 : j-1  
 y(i) = y(i) - ? ...*Update the right hand sides*  
 end  
 end

The diagonal entries of **U** are not 1, by the way.

**Test**: When you think your code is working, carry out the following test:

**A = frank ( 5 )**   
**[P,L,U] = ge\_pp ( A )**   
**x = [1:5]'**   
**b = A \* x**   
**x2 = plu\_solve ( P, L, U, b )**

**Results**: If your code has passed the test I just descrbed, it is probably working correctly. I want to take a look at your code (that's all). Mail your **plu\_solve.m** code to me.