Laboratorio 1 MA-43B: Algebra Lineal Numérica 1

Gonzalo Hernández

UChile - Departamento de Ingeniería Matemática

El objetivo de este laboratorio es aprender:

1) Programación elemental en Matlab: scripts y funciones .m

2) Algebra lineal numérica: Sistemas de ecuaciones lineales (SEL)

3) Aplicaciones del sistemas de ecuaciones lineales: Interpolacion polinomial

1 Programación elemental en Matlab

1.1 Bifurcaciones

En Matlab, al igual que otros lenguajes, se pueden programar bifurcaciones, es decir, se ejecuta un conjunto de comandos si solo si una cierta expresión condicial es verdadera.

La instrucción básica para evaluar una expresión condicional es el if/end:

if (expresión condicional)
..... //Comandos de Matlab
end

Veamos un ejemplo simple:

```
>> x=12; y=-3;
>> if ((x~=13)|(y<0))
z=x*y
end
z =
   -36
```

También se puede hacer una bifurcación if/else/end. En este caso, se ejecuta un conjunto de comandos si la expresión condicional es verdadera, o ejecuta otro conjunto de comandos si la expresión condicional es falsa.

if (expresión condicional)
..... //Comandos de Matlab
else
..... //Comandos de Matlab
end

Veamos un ejemplo simple:

```
>> x=10; y=5;
>> if (x+y)/3 < 5
z=x*y
else
z=x-y
end
z =
     5
```

Por último, es posible realizar la bifurcación if/elseif/else/end. En este caso, se ejecuta un conjunto de comandos si la expresión condicional del if es verdadera, ejecuta otro conjunto de comandos si la expresión condicional del elseif es verdadera y en otro caso ejecuta el conjunto de comandos asociados al else.

if (expresión condicional 1)
..... //Comandos de Matlab
elseif (expresión condicional 2)
..... //Comandos de Matlab
else
..... //Comandos de Matlab
end

Veamos un ejemplo simple:

```
>> if ((x+y)/3 < 5 & (x+y)/3 >= 3)
w=x*y*z
elseif ((x+y)/3 <= 10 & (x+y)/3 >= 5)
w=x+y+z
else
w=x^2 - y^2 - z^2
end
w =
    22
```

1.2 Ciclos

Como en otros lenguajes de programación, en Matlab existen ciclos o loops: while, for. La estructura de un ciclo while/end está dada por:

while (expresión condicional)
..... //Comandos de Matlab
end

Observaciones:

1) El valor de verdad de la expresión del while se revisa sólo al comienzo de cada ciclo.

2) En el caso que el valor de verdad de la expresión es falso se terminará de ejecutar todas las intrucciones.

Veamos un ejemplo simple:

```
>> x=0;
n=0;
while x<3
x=x*x+1
n=n+1
end
x =
     1
n =
     1
x =
     2
n =
     2
```

La estructura de un ciclo for/end está dada por:

for k = r : s : t
..... //Comandos de Matlab
end

k : nombre de la variable indicadora del ciclo.
r : valor de la variable k en la primera iteración.
s : Valor que se le incrementa a la variable k al final de cada iteración
t : Valor de la variable k en la última iteración

Observaciones:

1) El incremento s puede ser negativo, por ejemplo k = 25:-5:10 produce cuatro iteraciones con k = 25,20,15,10.

2) Si el incremento s es omitido, por defecto vale 1, es decir k = r:t $\Leftrightarrow$ k = r:1:t. Si r = t, el ciclo es ejecutado una vez.

3) Si los valores de r, s y t no hacen posible que k sea igual a t, entonces:

   (a) Si s es positivo, la última iteración es cuando k vale el mayor número menor que t, por ejemplo k=8:10:50 produce cinco iteraciones con k=8,18,28,38,48.

   (b) Si s es negativo, la última iteración es cuando k vale el menor número mayor que t, por ejemplo k=10:-3:2 produce tres iteraciones con k=10,7,4.

Veamos un par de ejemplos.
El siguiente código imprime los números primos entre 1 y 30, asignándolos al vector p.

```
>> p=[];
>> for k=1:30
if (isprime(k))
p=[p k];
end
end
```

```
p
p =
     2     3     5     7    11    13    17    19    23    29
```

La matriz de Van der Monde V de un vector x se utiliza para calcular el polinomio de interpolación que pasa por los puntos definidos por los vectores x, y. Está definida como:

$$\begin{aligned} V &= [V_{ij}]_{i,j=1}^{n} \\ V_{ij} &= x_i^{(n-j)} \end{aligned} \tag{1}$$

El siguiente código calcula esta matriz para $x = [1\ 2\ 3\ 4]$.

```
>> x=[1:4];
>> n=length(x);
M=[];
if (n>=1)
    for i=1:n
        m=[];
        for j=1:n
            m=[m x(i)^(n-j)];
        end
        M = [M;m];
    end
else
    error('Tamano de vector incorrecto')
end
disp(M)
     1     1     1     1
     8     4     2     1
    27     9     3     1
    64    16     4     1
```

El código anterior calcula correctamente V pero no lo hace eficientemente. Una forma de mejorar la eficiencia del código es reemplazar los ciclos for por instrucciones matriciales o vectoriales:

```
>> x=[1:4];
>> n=length(x);
j=[(n-1):-1:0];
M=[];
if (n>=1)
    for i=1:n
        m=x(i).^j;
        M = [M;m];
    end
else
    error('Tamano de vector incorrecto')
end
```

```
disp(M)
     1     1     1     1
     8     4     2     1
    27     9     3     1
    64    16     4     1
```

El comando:

$$m = x(i).\hat{}j; \tag{2}$$

es una operación vectorizada. Permite calcular simultánemente $x(i)\hat{}j(1)$ $x(i)\hat{}j(2)\ldots x(i)\hat{}j(n)$, ahorrando el segundo ciclo for.

Las siguientes recomendaciones son útiles para mejorar el rendimiento de programas Matlab, ver ref. [2]:

E1) Vectorización: Matlab es un lenguaje matricial/vectorial. La gran mayoría de las operaciones aritméticas, matemáticas, lógicas y relacionales se realizan internamente mediante código muy eficiente. Los ciclos while y for son la instrucciones menos eficiente de Matlab, especialmente cuando el código adentro de alguno de estos ciclos es complejo de evaluar matemática o lógicamente. Por esta razón, la recomendación principal para mejorar el rendimiento es utilizar la menor cantidad de ciclos, reemplazándolos por operaciones matriciales/vectoriales. Veremos más adelante un ejemplo de optimización por vectorización.

E2) Pre-alocación de arreglos: En Matlab es posible utilizar arreglos sin haberlos declarados previamente. esta flexibilidad origina ineficiencias debido que para arreglos o matrices no declaradas se reservan espacios de memoria más grandes de lo necesario. Pre-asignar el tamaño de memoria tiene la ventaja adicional de reducir la fragmentación de la ram, ocasionada por el manejo de memoria dinámico.

E3) El exceso de bifurcaciones if/elseif/else provoca también bajo rendmiento. Es posible evitar bifurcaciones utilizando operadores relacionales. Para ejemplificar este punto, supongamos que se necesita evaluar la función $f(x)$ en un vector x :

$$f(x) = \begin{cases} \sin(x) & x < 0 \\ x & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases} \tag{3}$$

Es posible realizar este cálculo de la siguiente forma:

```
>> x=[-2:0.5:2];
>> y=sin(x).*(x<0) + x.*(x>=0 & x<=1) + 1.*(x>1)
y =
  Columns 1 through 9

   -0.9093   -0.9975   -0.8415   -0.4794         0    0.5000    1.0000    1.0000    1.0000
```

O equivalentemente:

```
>> x=[-2:0.5:2];
>> y=ones(size(x));
>> k=(x<0); y(k)=sin(x(k));
>> k=(x>=0 & x<=1); y(k)=x(k);
>> y
```

```
y =
  Columns 1 through 9

   -0.9093   -0.9975   -0.8415   -0.4794         0    0.5000    1.0000    1.0000    1.0000
```

En el último código se ha utilizado sub-indexación unidimensional, es decir se define el índice del vector y en función de los valores del vector x.

ACT1: Un script de Matlab es una secuencia de comandos de la forma:

```
function ejemplo()
x=[-2:0.25:2];
n=length(x);
out=zeros(2,n);
y=ones(size(x));
k=(x<0); y(k)=sin(x(k));
k=(x>=0 & x<=1); y(k)=x(k);
out=[x;y];
disp(out);
plot(x,y);
end
```

El script ejemplo.m es una función .m que no tiene argumentos de entrada y su salida son los valores de x, y y el gráfico de las variables x, y. La principal diferencia de un script con una función .m es que la última sí tiene argumentos de entrada y salida.

El objetivo de esta actividad es programar un script de Matlab que permita calcular los parámetros de una regresión lineal en base a datos conocidos. El método de Regresión Lineal permite calcular la recta que mejor representa un conjunto de puntos conocidos $(x_1, y_1), ..., (x_n, y_n)$, en el sentido de los mínimos cuadrados discretos:

$$\min_{a_0, a_1} \varepsilon_{MC} = \sum_{i=1}^{n} \left(y_i - (a_0 + a_1 x_i)\right)^2 \tag{4}$$

Dado que la función ε_{MC} es cuadrática y convexa con respecto a los coeficientes a_0, a_1 para calcularlos basta aplicar las condiciones de optimalidad de primer orden:

$$\begin{aligned} \frac{\partial \varepsilon_{MC}}{\partial a_0} &= 0 \\ \frac{\partial \varepsilon_{MC}}{\partial a_1} &= 0 \end{aligned} \tag{5}$$

cuya solución está dada por las ecuaciones normales:

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{n} y_i \\ \sum_{i=0}^{n} x_i y_i \end{bmatrix} \tag{6}$$

Programe un script de Matlab en la ventana "Editor" que:

1) Construya 1000 pares de puntos (x_i, y_i) que tengan una relación aproximadamente lineal.

2) Para los puntos anteriores calcule los parámetros a_0, a_1 que son solución de la ecuación (6). Haga este cálculo utilizando la regla de Cramer. En la programación del cálculo anterior utilice las recomendaciones E1 - E3 para optimizar el rendimiento del programa.

3) Imprima los valores de a_0, a_1, el grafico con los puntos y la recta calculada, el coeficiente de correlación y el error total de aproximación.

   Incluya en el informe del laboratorio:

   i) El codigo del script.

   ii) Explicación de las optimizaciones de rendimiento utilizadas.

   iii) Los resultados obtenidos: $a_{0,}a_1$, coeficiente de correlación, error total de aproximación.

   iv) Los graficos de los puntos y la recta calculada.

2 Sistemas de ecuaciones lineales

La mayor capacidad de Matlab son los comandos y funciones disponibles para realizar cálculo matricial:

1) Sistemas de ecuaciones lineales

2) Factorización de matrices: $PA = LU, A = LU$ (Crout), $A = RR^T$ (Cholesky), $A = QR$ (Householder), $A = USV$ (Descomposición en valores singulares).

3) Métodos para valores y vectores propios

4) Métodos iterativos para SEL: Jacobi, Gauss - Seidel, Sobre-Relajación, Gradiente Conjugado.

Los métodos implementados en Matlab para Algebra Lineal provienen de la librería LAPACK, ver ref. [1]. A continuación aprenderemos los comandos básicos de Matlab que permiten resolver SEL. Inicialmente trabajaremos con el sistema:

$$\begin{bmatrix} 100 & 1 & 10 \\ 101 & 1 & 9 \\ 102 & 1 & 9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 1 \\ 0 \end{bmatrix} \tag{7}$$

Cuya solución exacta es:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -1 \\ 30 \\ 8 \end{bmatrix} \tag{8}$$

Veamos como ingresamos este sistema a Matlab:

2.1 Creación de un vector

$b = [a_1\ a_2\ ...\ a_n];$ $\longrightarrow$ Si b es un vector fila
$b = [a_1; a_2;\ ...\ ; a_n];$ $\longrightarrow$ Si b es un vector columna

En nuestro ejemplo:

```
>> b=[10;1;0]

b =

    10
     1
     0
```

O de forma equivalente:

```
>> b=[10 1 0]'

b =

    10
     1
     0
```

El comando ' significa transposición.

2.2 Creación de una matriz

$A = [f_1; f_2; f_3; ...; f_n]$; $\longrightarrow$ donde los f_k son filas de la forma $f_k = a_1\ a_2 ...\ a_n$

En nuestro ejemplo:

```
>> A=[100 1 10; 101 1 9; 102 1 9]
A =
   100     1    10
   101     1     9
   102     1     9
```

Es posible también construir una matriz por bloques, por ejemplo:

```
>> A=[rand(3,2) zeros(3);zeros(3,2) eye(3)]
A =
    0.4565    0.4447         0         0         0
    0.0185    0.6154         0         0         0
    0.8214    0.7919         0         0         0
         0         0    1.0000         0         0
         0         0         0    1.0000         0
         0         0         0         0    1.0000
```

Existe una gran cantidad de matrices predefinidas en Matlab, como por ejemplo ($n, m, p, k \in \mathbb{N}$; $x, y \in \mathbb{R}^q$):

```
>> ones(n); zeros(n); eye(n)
>> pascal(n)
>> hilb(n); invhilb(n)
>> magic(n)
>> vander(x)
```

```
>> rand(n,m)
>> compan(v)
>> wilkinson(n)
>> gallery('binomial',n)
>> gallery('cauchy',x,y)
>> gallery('rando',n,k)
```

Se invita al alumno a investigar que significa cada uno de estos comandos. Mas información puede obtener del "Help" de Matlab.

ACT2: Ud. deberá probar en Matlab los siguientes comandos y describir brevemente en el informe el resultado que obtiene. Considere $i, n, m \in \mathbb{N}$.

1) A(n,m); A(:,m); A(n,:); A(i,n:m); A(n:m,i)

2) U=triu(A); L=tril(A); [L,U,P]=lu(A);

3) L+U; L*U; L.*U; A*n; A+n; A^n; A.^n

4) D1=diag(b); D2=diag(A); A./D1; A./D2;

5) det(A); inv(A); x1=A\b; x2=inv(A)*b;

2.3 Aplicación de SEL: Interpolación polinomial

Una aplicación de SEL es determinar el polinomio que interpola $(n+1)$ puntos. Sean $(x_1, y_1), ..., (x_{n+1}, y_{n+1})$ los datos. Vamos a suponer que: $y_i = f(x_i)\ \forall i = 1, ..., (n+1)$ pero que desconocemos esta función $f(x)$. El polinomio de interpolación de Lagrange:

$$p_L(x) = \sum_{k=0}^{n} a_k x^k \tag{9}$$

es el polinomio de menor grado que interpola estos puntos, es decir, que verifica:

$$y_i = p_L(x_i) = \sum_{k=0}^{n} a_k x_i^k \quad \forall i = 1, ..., (n+1) \tag{10}$$

Para calcular $p_L(x)$ basta resolver el sistema de Van der Monde:

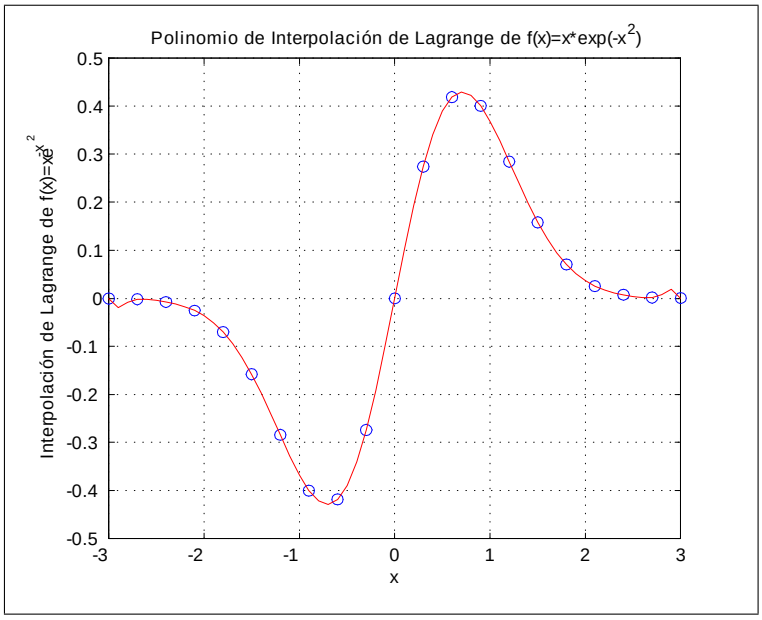
$$\begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n+1}^n & x_{n+1}^{n-1} & \cdots & x_{n+1} & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{bmatrix} \tag{VM}$$

Es decir, el problema de determinar el polinomio de interpolación $p(x)$ de una función $f(x)$ se reduce a un sistema de ecuaciones lineales donde las incógnitas son los coeficientes del polinomio:

$$Xa = y \tag{11}$$

Por ejemplo, para calcular el polinomio de interpolación de Lagrange de la función $f(x) = xe^{-x^2}$ en $[-3, 3]$ utilizando una malla de 21 puntos equi-espaciados a distancia $h = 0.3$, se debe ejecutar la secuencia de comandos:

```
>> x=[-3:0.3:3];
>> y=x.*exp(-x.^2);
>> p=vander(x)\(y)';
>> xx=[-3:0.1:3];
>> yy=polyval(p,xx);
>> plot(x,y,'bo');
>> hold on
>> plot(xx,yy,'-r');
```



ACT3: Para calcular el polinomio de interpolación de Lagrange de $f(x) = e^{\sin(x^3)}$ en $[-2, 2]$:

1) Defina una malla gruesa de puntos $x_1, ..., x_{n+1}$ a distancia $q = 0.2$ y evalúe: $y_i = f(x_i)\ \forall i = 1, ..., (n+1)$.

2) Calcule la matriz de Vandermonde X y la solución del sistema de interpolación (VM).

3) Grafique en un subplot los datos $(x_i, y_i)_{i=1}^{n+1}$ destacándolos y el polinomio de interpolación $p_L(x)$ evaluado en una malla fina a distancia $q = 0.05$. Puede utilizar la función de Matlab: "polyval".

4) Calcule el error cuadrático total de interpolacion en la malla fina:

$$\epsilon_L = \sum_{x_k \in MFk} \left(f(x_x) - p_L(x_k)\right)^2 \tag{12}$$

Grafique en otro subplot el error de interpolacion punto a punto.

Puede concluir que el polinomio de Lagrange aproxima con precisión a la función $f(x)$ en $[-2, 2]$?

Cómo es la calidad de la extrapolación fuera del intervalo anterior ?

Incluya en el informe los resultados que obtuvo, los comandos que utilizó y los gráficos generados.

3 Creación de una función .m

A continuación programaremos una función .m que implementa el algoritmo de Gauss sin pivoteo. La indicaciones más importantes relacionadas a la programación de funciones .m son:

F1) Para crear una función .m se debe ir al menú *File* y ejecutar: $File \rightarrow New \rightarrow M\text{-}File$. Se abrirá una ventana en el "Editor". La primera linea de la función .m debe ser, por ejemplo:

function x = gauss(A,b)

En este caso, la función gauss.m tiene dos entradas o inputs: A, b y una única salida o output, la variable x.

Si se quiere que la función gauss.m tenga más salidas, por ejemplo x, e :

function [x,e] = gauss(A,b)

También se puede crear una afunción .m en la ventana "Editor", que puede ser activada en el menú "Desktop".

F2) La última linea de una función .m debe ser el comando end.

F3) Para guardar la función se debe ir al menú *File* y ejecutar: *File* → *Save As*. Se escoge la carpeta donde se guardará la función y el nombre que tendrá. Por ejemplo, la puede guardar en la carpeta "C:\Laboratorio-MA-43B", con el nombre de gauss.m. Es una buena práctica guardar la función con el nombre que se utilizó en la linea function.

F4) Para poder utilizar funciones que hemos creado, estas se deben visualizar en la ventana "Current Directory". Por ejemplo, si queremos utilizar la funcion gauss.m, se debe cambiar el "Current Directory" a "C:\Laboratorio-MA-43B".

F5) Una vez creadas las variables A y b se puede utilizar la funcion gauss.m de la siguiente forma:

>> x3=gauss(A,b)

Queda entonces guardada en la variable x3 el valor entregado por la funcion gauss.m.

F6) Se puede programar una función que utiliza otra función siempre y cuando ambas estén en el "Current Directory". Por ejemplo:

function [x,n] = gauss2(A,b)

n = length(b); x=gauss(A,b);

end

Los comandos que ejecuta la función gauss.m son los siguientes:

```
function x=gauss(A,b)
%Se guarda en la variable n el tamaño del vector b
n = length(b)
%El ciclo for principal comienza en k=1, en cada iteración se suma 1 a k, y termina cuando k=n-1
for k = 1:(n-1)
    %Comienza un ciclo for anillado al anterior entre i=k+1 y i=n
    for i = k+1:n
        %En la variable lambda se guarda el valor de la división por el pivote A(k,k)
        lambda = A(i,k)/A(k,k);
        %Para ahorrar un ciclo, se realiza la operación elemental en A desde la columna k+1
        %hasta la n, simultáneamente.
        A(i,k+1:n) = A(i,k+1:n) - lambda*A(k,k+1:n);
        %Operación elemental en el vector b
        b(i)= b(i) - lambda*b(k);
    %Finaliza ciclo for del índice i
    end
%Finaliza ciclo for del índice k
end
%Comienza ciclo for para la sustitución backwards, inicializando la variable k=(n-1), en cada iteración
se
```

```
%suma -1 a k, y termina cuando k=1
x=zeros(n,1);
x(n)=b(n)/A(n,n);
for k = (n-1):-1:1
x(k) = (b(k) - A(k,k+1:n)*x((k+1):n))/A(k,k);
end
%Finaliza función gauss.m
end
```

4 Comparación de métodos para resolver SEL

En Matlab se puede resolver un SEL de varias formas. Incluyendo el que programamos, contamos con 3 métodos:

1) x1=A\b

2) x2=inv(A)*b

3) x3=gauss(A,b)

A continuamos veremos tres formas de comparar estos métodos:

- Exactitud o precisión
- Robustez ante perturbaciones de los datos
- Cantidad de operaciones o tiempo de ejecución

La exactitud se mide mediante el error de la solución aproximada $\tilde{x}$ en comparación con la solución exacta x, se define de dos formas:

i) Error absoluto:

$$E_{abs}(x, \tilde{x}) = \|x - \tilde{x}\| \tag{13}$$

ii) Error relativo:

$$E_{rel}(x, \tilde{x}) = \frac{\|x - \tilde{x}\|}{\|x\|} \tag{14}$$

Utilizaremos la norma infinito, definida para una matriz C y un vector z como:

$$\|C\| = \max_{i=1\ldots n} \sum_{j=1}^{n} |c_{ij}| \qquad \|z\| = \max_{i=1\ldots n} \ |c_i| \tag{15}$$

En Matlab, esta norma se escribe como:

```
>> norm(A,inf)
ans =
   112
```

Se invita al alumno a revisar las otras normas que están disponibles en Matlab, para esto busque en el Help información sobre el comando norm.

Ahora bien, como no siempre se sabe a priori la solución exacta x, estos errores se pueden calcular como:

i) Error Absoluto:

$$E_{abs} = \|A\tilde{x} - b\| \tag{16}$$

ii) Error Relativo:

$$E_{rel} = \frac{\|A\tilde{x} - b\|}{\|b\|} \tag{17}$$

En el caso de SEL u otro tipo de ecuaciones, no siempre se tiene acceso a los datos reales. Esto se debe principalmente a errores en la toma de datos o fallas de funcionamiento en los instrumentos que los toman. Es razonable entonces considerar pequeñas fluctuaciones o pertubaciones en sus valores. Por ejemplo, consideremos el siguiente sistema pertubado de (A, b):

$$\begin{bmatrix} 100 & 1 & 10 \\ 101 & \boxed{1.01} & 9 \\ 102 & 1 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_2 \end{bmatrix} = \begin{bmatrix} 10 \\ 1 \\ 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} x_1 \\ x_2 \\ x_2 \end{bmatrix} \approx \begin{bmatrix} -0.863\,636\,363\,636\,363\,636\,36 \\ 13.\,636\,363\,636\,363\,636\,364 \\ 8.\,272\,727\,272\,727\,272\,727\,3 \end{bmatrix} \tag{18}$$

Como puede observar, un pequeño cambio en el coeficiente a_{22} genera un cambio significativo en la solución del SEL. Diremos que un SEL (A, b) es robusto si es estable ante perturbaciones, es decir, si la solución del sistema perturbado es cercana a la solución del sistema original. Si las perturbaciones se realizan en la matriz A, el número de condicionamiento permite saber si tendrán un gran efecto en la solución del sistema. Se define como:

$$cond(A) = \|A\| \left\|A^{-1}\right\| \tag{19}$$

Es posible demostrar que $cond(A) \geq 1 \ \forall A \in \mathbb{R}^{n\times n}$ y que una matriz es robusta si $cond(A)$ es cercano a 1, ver ref. [3]. El comando cond permite calcular el numero de condicionamiento en Matlab.

Desde el punto vista computacional, un método es más eficiente mientras menos operaciones ejecuta. Más aún, se acostumbra clasificar los algoritmos en base a la cantidad de operaciones según: polinomial/no polinomial. La instrucción *cputime* retorna el tiempo total en segundos que Matlab ha utilizado la CPU al ejecutar un comando o función:

```
>> t=cputime; x=inv(A)*b; e=cputime-t
e =
     0
```

Tambien es posible medir el tiempo transcurrido entre 2 instantes mediante el comando *tic/toc* :

```
>> tic; x=inv(A)*b; toc
Elapsed time is 0.000176 seconds.
```

Como el sistema (A, b) es de 3×3, el tiempo necesario para resolverlo es despreciable. Pero si resolvemos un sistema de mayor dimensión el tiempo será considerable.

En Matlab, un cell array es una matriz en donde sus elementos pueden ser constantes, enteros, reales, matrices, cadenas de caracteres, etc. Por ejemplo:

```
>> C={'','cadena','hola', 'mundo';'numeros', [3],[55],[5];'error',[3.764],[10],[pi]}
C =
          ''    'cadena'              'hola'    'mundo'
    'numeros'    [               3]    [  55]    [               5]
    'error'      [3.76400000000000]    [  10]    [3.14159265358979]
```

ACT4: Sistemas de Ecuaciones Lineales. Defina un SEL con $A \in \mathbb{R}^{10\times 10}, b \in \mathbb{R}^{10}$

1) Programe una función en Matlab, llamada errores.m, cuyos inputs sean una matriz A y un vector b, y entregue el error relativo de los 3 métodos al resolver el SEL: $Ax = b$. Incluya la función errores.m en el informe, los resultados de la prueba con el sistema (A, b). Explique cual método es más exacto.

2) Realice una perturbación del $1\%, 2\%, 3\%$ del valor en algún coeficiente de la matriz A. Resuelva cada uno de los SEL pertubados con los 3 métodos que hemos visto en este laboratorio. Incluya en el informe las perturbaciones que realizó y los resultados que obtuvo. ¿Qué puede observar? ¿Cuál método es más robusto?

   En resumen, ¿ Cuál método en general es mejor considerando errores y robustez ?

3) Programe una función en Matlab, llamada tiempo.m, que reciba una matriz M y un vector N, y entregue el tiempo de ejecución de cada uno de los 3 métodos para resolver el SEL. Haga una prueba con un sistema de 500×500, utilizando los comandos 'M=rand(500)' y 'N=rand(500,1)'. Incluya en el informe la función tiempo.m y los resultados que obtuvo. ¿Qué puede observar? ¿Cuál método es más eficiente?

4) Utilizando las funciones "errores.m" y "tiempo.m", programe una funcion llamada "tabla.m" que reciba una matriz M y un vector N, y devuelva un cell array con el error relativo y el tiempo de ejecución de los 3 métodos. Escriba en el informe esta función, y el resultado de la prueba con un sistema de 500×500.

References

[1] Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK User's Guide, Third Edition, SIAM, Philadelphia, 1999. (http://www.netlib.org/lapack/lug/lapack_lug.html)

[2] Highman, D.J., N.J. Highman, Matlab Guide, Second Edition, SIAM, 2005.

[3] Ortega, J.M., Numerical Analysis: A Second Course, SIAM Classics in Applied Mathematics, 1990.