4101553 Métodos Numéricos aplicados a la Ingenieria Civil

Departamento de Ingeniería Civil

Universidad Nacional de Colombia

Sede Manizales

Docente: Juan Nicolás Ramírez Giraldo (jnramirezg@unal.edu.co (mailto:jnramirezg@unal.edu.co))

"*Cum cogitaveris quot te antecedant, respice quot sequantur*" **Séneca**

Repositorio de la asignatura (https://github.com/jnramirezg/metodos_numericos_ingenieria_civil/)

Unidad 1: Sistemas de ecuaciones lineales

1.6. Soluciones de sistemas en Python

Se presentan los comandos y funciones más importantes que se relacionan con solución de sistemas de ecuaciones lineales y álgebra lineal. Para ello, se usan las librerías `numpy` y `sympy` , además, de una nueva librería con alto uso en ámbitos científicos: `scipy` .

Se importan las librerías:

In [1]:
```
import numpy as np
import sympy as sp

```

1.6.1. En `numpy`

- `np.linalg.solve(a, b)`
- `np.linalg.cholesky(a)`
- `np.linalg.norm(v)`
- `np.linalg.eig(a)` y `np.linalg.eigvals(a)`

En ocasiones, y por facilidad, se le suele hacer un llamado al submódulo `linalg` de `numpy` así:

In [2]:
```
from numpy import linalg as LA

```

`LA.solve(a, b)`

Ejemplo 1:

In [3]:
```
A1 = [[-1, -1,  6,  9],
      [-5,  5, -3,  6],
      [ 7, -3,  5, -6],
      [ 3, -3, -2,  3]]

B1 = [-29, -54, 38, 41]

```

In [4]:
```
LA.solve(A1, B1)

```

Out[4]:
```
array([ 4., -8., -4., -1.])
```

Se anota que los argumentos de la función pueden ser pasados como lista de listas y lista, o también como `np.array`.

Ejemplo 2: sistema singular no detectado

In [5]:
```
A2 = [[ 2,  1,  4, -1],
      [ 3, -2,  1,  0],
      [ 5,  1, -3,  2],
      [-1,  3,  3, -1]]

B2 =  [1, -1, 4, 3]
```

In [6]:

```
LA.solve(A2, B2)
```

Out[6]:

```
array([ 7.70352568e+14,  3.55547339e+14, -1.59996303e+15, -4.50359963
e+15])
```

Debido a que se trata de un resultado muy sospechoso, se verifica el valor del determinante:

In [7]:

```
LA.det(A2)
```

Out[7]:

```
-1.6875389974302392e-14
```

Claramente `A2` es una matriz singular que, genera un sistema singular, sin embargo, la función no lo detectó por la representación numérica. Por lo tanto, es pertinente poner la advertencia en un programa donde se use la función `LA.solve()` .

Ejemplo 3:

In [8]:

```
A3 = [[ 1, -1,  1],
      [-2,  2, -2],
      [ 2,  4,  2]]

B3 = [1, 2, 1]
```

In [9]:

```
LA.solve(A3, B3)
```

```
------------------------------------------------------------------------
-----
LinAlgError                               Traceback (most recent call
last)
<ipython-input-9-2ff44672f187> in <module>
----> 1 LA.solve(A3, B3)
```

Se verifica el valor del determinante:

In [10]:

```
LA.det(A3)
```

Out[10]:

```
0.0
```

La causa del error es `LinAlgError: Singular matrix` , lo cual es completamente coherente con el resultado del determinante.

Ejemplo 4: valores muy cercanos a cero en la diagonal.

In [11]:

```
A4 = [[0.0000000000000000000000003, 3.0000],
      [1.0000,                       1.0000]]
B4 = [ 2.0000000000000000000000001, 1.0000]
```

In [12]:

```
LA.solve(A4, B4)
```

Out[12]:

```
array([0.33333333, 0.66666667])
```

No se evidencia ningún incoveniente como sí ocurría en otros métodos.

Ejemplo 5:

In [13]:

```
A5 = [[-9,  7,  2,  5,  7],
      [ 5,  3, -2,  2, -6],
      [ 2, -6, -5, -7, -8],
      [-2,  4, -2, -2, -6]]

B5 = [1, -1, 4, 3]
```

In [14]:

```
LA.solve(A5, B5)
```

```
-----------------------------------------------------------------------
-----
LinAlgError                               Traceback (most recent call
last)
<ipython-input-14-7eb12165d5b2> in <module>
----> 1 LA.solve(A5, B5)

<__array_function__ internals> in solve(*args, **kwargs)

~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in solve(a, b)
    378     a, _ = _makearray(a)
    379     _assert_stacked_2d(a)
--> 380     _assert_stacked_square(a)
    381     b, wrap = _makearray(b)
    382     t, result_t = _commonType(a, b)

~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _assert_stacke
d_square(*arrays)
    201         m, n = a.shape[-2:]
    202         if m != n:
--> 203             raise LinAlgError('Last 2 dimensions of the array
must be square')
    204
    205 def _assert_finite(*arrays):
```

En este caso la fuente de error es `LinAlgError: Last 2 dimensions of the array must be square` , es decir, la matriz no es cuadrada.

Ejemplo 6:

In [15]:
```
A6 = [[-1,  2],
      [ 3, -1]]

B6 = [[1, 4, 10],
      [2, 1,  3]]

```

In [16]:
```
LA.solve(A6, B6)

```

Out[16]:
```
array([[1. , 1.2, 3.2],
       [1. , 2.6, 6.6]])
```

De hecho, tiene mucho sentido pues sigue la siguiente estructura algebraica:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$$

En donde se tiene 3 sistemas de ecuaciones de $2x2$, pero cuyos coeficientes constantes son iguales, cambiando únicamente sus constantes. De hecho, hubiéramos podido programar nuestros métodos previos de esta manera.

Es equivalente a resolver:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_{12} \\ x_{22} \end{bmatrix} = \begin{bmatrix} b_{12} \\ b_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_{13} \\ x_{23} \end{bmatrix} = \begin{bmatrix} b_{13} \\ b_{23} \end{bmatrix}$$

`LA.cholesky(a)`

Ejemplo 1: matriz definida positiva y simétrica

In [17]:
```
A1 = np.array([[ 6,   3,   4,   8],
               [ 3,   6,   5,   1],
               [ 4,   5,  10,   7],
               [ 8,   1,   7,  25]])

```

In [18]:
```
LA.cholesky(A1)

```

Out[18]:
```
array([[ 2.44948974,  0.        ,  0.        ,  0.        ],
       [ 1.22474487,  2.12132034,  0.        ,  0.        ],
       [ 1.63299316,  1.41421356,  2.30940108,  0.        ],
       [ 3.26598632, -1.41421356,  1.58771324,  3.13249102]])
```

In [19]:
```
L = LA.cholesky(A1)

```

In [20]:
```
L@L.T

```

Out[20]:
```
array([[ 6.,  3.,  4.,  8.],
       [ 3.,  6.,  5.,  1.],
       [ 4.,  5., 10.,  7.],
       [ 8.,  1.,  7., 25.]])
```

Ejemplo 2: cuando no se cumple que la matriz esté definida positiva y sea simétrica.

In [21]:
```
A2 = [[-1, -1,   6,   9],
      [-5,   5, -3,   6],
```

```
      [ 7, -3,  5, -6],
      [ 3, -3, -2,  3]]
```

In [22]:

```
LA.cholesky(A2)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-22-70355d2581b3> in <module>
----> 1 LA.cholesky(A2)

<__array_function__ internals> in cholesky(*args, **kwargs)

~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in cholesky(a)
    761     t, result_t = _commonType(a)
    762     signature = 'D->D' if isComplexType(t) else 'd->d'
--> 763     r = gufunc(a, signature=signature, extobj=extobj)
    764     return wrap(r.astype(result_t, copy=False))
    765

~\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _raise_linalgerror_nonposdef(err, flag)
     89
     90 def _raise_linalgerror_nonposdef(err, flag):
---> 91     raise LinAlgError("Matrix is not positive definite")
     92
     93 def _raise_linalgerror_eigenvalues_nonconvergence(err, flag):

LinAlgError: Matrix is not positive definite
```

LA.norm(v)

La norma de un vector en $\mathbb{R}^n$

$\underline{V} = (v_0, v_1, v_2, \dots . v_n)$ es:

$||\underline{V}|| = \sqrt{v_0^2 + v_1^2 + v_2^2, \dots, v_n^2}$

In [23]:

```
V = [-1, 2, -2]
```

In [24]:

```
LA.norm(V)
```

Out[24]:

```
3.0
```

`LA.eig(a)` y `LA.eigvals(a)`

Los valores de λ se obtienen de la solución del polinomio característico que surge de: $|\underline{\underline{A}} - \lambda \underline{\underline{I}}| = 0$.

Y los vectores propios son aquellos que satisfacen para cada valor de λ la ecuación $(\underline{\underline{A}} - \lambda \underline{\underline{I}})\underline{X} = \underline{0}$.

Por cada λ hay un vector propio.

In [25]:
```
A = [[1, 2],
     [3, 2]]

```

In [26]:
```
LA.eigvals(A)

```

Out[26]:
```
array([-1.,  4.])
```

In [27]:
```
LA.eig(A)

```

Out[27]:
```
(array([-1.,  4.]),
 array([[-0.70710678, -0.5547002 ],
        [ 0.70710678, -0.83205029]]))
```

In [28]:
```
e, w = LA.eig(A)

```

La primera parte del resultados es equivalente a usar la función `LA.eig(A)` .

In [29]:
```
e

```

Out[29]:
```
array([-1.,  4.])
```

En la segunda parte del resultado, el primer vector propio es la primera columna y el segundo vector propio es la segunda columna (los vectores ya están normalizados a 1).

In [30]:
```
w

```

Out[30]:
```
array([[-0.70710678, -0.5547002 ],
       [ 0.70710678, -0.83205029]])
```

In [31]:
```
w1 = w[:, 0]
w2 = w[:, 1]

```

In [32]:
```
w1

```

Out[32]: `array([-0.70710678,  0.70710678])`

In [33]:
```
LA.norm(w1)

```

Out[33]: `0.9999999999999999`

In [34]:
```
w2

```

Out[34]: `array([-0.5547002 , -0.83205029])`

In [35]:
```
LA.norm(w2)

```

Out[35]: `1.0`

1.6.2. En `sympy`

- `A.is_positive_definite`
- `A.cholesky()`
- `A.LUdecomposition()`
- `A.LUsolve(B)`
- `V.norm()`
- `A.eigenvals()` y `A.eigenvects()`
- `sp.solve(ecu, X)`

`A.is_positive_definite`

In [36]:
```
A = sp.Matrix([[25, 15, -5],
               [15, 18,  0],
               [-5,  0, 11]])

```

In [37]:
```
A
```
Out[37]:

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix}$$

Y por si parece extraño, mire esta otra forma de definir la misma matriz. Dependiendo de la aplicación que se desarrolle, puede ser útil una u otra.

In [38]:
```
# Los dos primeros argumentos indican el tamaño.
sp.Matrix(3, 3, (25, 15, -5, 15, 18,  0, -5,  0, 11))
```
Out[38]:

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix}$$

In [39]:
```
A.is_positive_definite
```
Out[39]:
```
True
```

`A.cholesky()`

In [40]:
```
A.cholesky()
```
Out[40]:

$$\begin{bmatrix} 5 & 0 & 0 \\ 3 & 3 & 0 \\ -1 & 1 & 3 \end{bmatrix}$$

In [41]:
```
L = A.cholesky()
```
In [42]:
```
L*L.T - A
```
Out[42]:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

A.LUdecomposition()

In [43]:
```
A = sp.Matrix([[4, 3],
               [6, 3]])
```

In [44]:
```
A.LUdecomposition()
```

Out[44]:
```
(Matrix([
 [  1, 0],
 [3/2, 1]]),
 Matrix([
 [4,    3],
 [0, -3/2]]),
 [])
```

In [45]:
```
L, U, _ = A.LUdecomposition()
```

In [46]:
```
L
```

Out[46]:

$$\begin{bmatrix} 1 & 0 \\ \frac{3}{2} & 1 \end{bmatrix}$$

In [47]:
```
U
```

Out[47]:

$$\begin{bmatrix} 4 & 3 \\ 0 & -\frac{3}{2} \end{bmatrix}$$

In [48]:
```
L*U - A
```

Out[48]:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

A.LUsolve(B)

Ejemplo 1:

In [49]:
```
A = sp.Matrix([[-1, -1,  6,  9],
               [-5,  5, -3,  6],
               [ 7, -3,  5, -6],
               [ 3, -3, -2,  3]])
```

```
B = sp.Matrix([-29, -54, 38, 41])

```

In [50]:
```
A.LUsolve(B)

```

Out[50]:

$$\begin{bmatrix} 4 \\ -8 \\ -4 \\ -1 \end{bmatrix}$$

Ejemplo 2:

In [51]:
```
A = sp.Matrix([[-1,  2],
               [ 3, -1]])

B = sp.Matrix([[1, 4, 10],
               [2, 1,  3]])

```

In [52]:
```
A.LUsolve(B)

```

Out[52]:

$$\begin{bmatrix} 1 & \frac{6}{5} & \frac{16}{5} \\ 1 & \frac{13}{5} & \frac{33}{5} \end{bmatrix}$$

V.norm()

In [53]:
```
V = sp.Matrix([-1, 2, -2])

```

In [54]:
```
V.norm()

```

Out[54]: 3

A.eigenvals() y A.eigenvects()

In [55]:
```
A = sp.Matrix([[1, 2],
               [3, 2]])

```

In [56]:
```python
A.eigenvals()   # Sus valores propios son 4 y -1

```
Out[56]:
```
{4: 1, -1: 1}
```

In [57]:
```python
A.eigenvects()

```
Out[57]:
```
[(-1,
  1,
  [Matrix([
   [-1],
   [ 1]])]),
 (4,
  1,
  [Matrix([
   [2/3],
   [  1]])])]
```

In [58]:
```python
A.eigenvects()[0][2][0]   # Vector propio 1.

```
Out[58]:

$$\begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

In [59]:
```python
A.eigenvects()[1][2][0] # Vector propio 2.

```
Out[59]:

$$\begin{bmatrix} \frac{2}{3} \\ 1 \end{bmatrix}$$

`sp.solve(ecu, X)`

Se definen las variables simbólicas de las incógnitas para un sistema $3x3$.

In [60]:
```python
x0, x1, x2 = sp.symbols('x_0, x_1, x_2')

```

Ejemplo 1:

Se define `ecu` : ecuaciones e `inc` : incógnitas.

In [61]:
```python
ecu = [5*x0 - 2*x1 + 1*x2 - 24,
        2*x0 + 5*x1 - 2*x2 + 14,
```

```
       1*x0 - 4*x1 + 3*x2 - 26]

inc = [x0, x1, x2]

```

In [62]:
```
sp.solve(ecu, inc)

```

Out[62]: `{x_0: 3, x_1: -2, x_2: 5}`

Ejemplo 2:

In [63]:
```
A = sp.Matrix([[5, -2, 1],
               [2, 5, -2],
               [1, -4, 3]])

B = sp.Matrix([[24],
               [-14],
               [26]])

X = sp.Matrix([[x0],
               [x1],
               [x2]])

```

In [64]:
```
sp.solve(A*X-B, X)

```

Out[64]: `{x_0: 3, x_1: -2, x_2: 5}`

Se le puede indicar qué método usar con el argumento opcional `method` . Por ejemplo, cuando se tienen matrices simétricas definidas positivas se puede poner `method='CH'` .

In [65]:
```
sp.solve(A*X-B, X, method='LU')

```

Out[65]: `{x_0: 3, x_1: -2, x_2: 5}`

In [66]:
```
X = sp.solve(A*X-B, X)

```

In [67]:
```
X[x0]

```

Out[67]: 3

In [68]:
```
X[x1]

```

Out[68]: -2

In [69]:
```
X[x2]
```
Out[69]: 5

Ejemplo 3: un sistema $3x3$ muy simbólico.

In [70]:
```
a0, a1, a2 = sp.symbols('a_0 a_1 a_2')
a3, a4, a5 = sp.symbols('a_3 a_4 a_5')
a6, a7, a8 = sp.symbols('a_6 a_7 a_8')

b0, b1, b2 = sp.symbols('b_0 b_1 b_2')
```

In [71]:
```
A = sp.Matrix([[a0, a1, a2],
               [a3, a4, a5],
               [a6, a7, a8]])

B = sp.Matrix([[b0],
               [b1],
               [b2]])

X = sp.Matrix([[x0],
               [x1],
               [x2]])
```

In [72]:
```
sol = sp.solve(A*X - B, X)
```

In [73]:
```
sol[x0]
```
Out[73]:
$$\frac{a_1 a_5 b_2 - a_1 a_8 b_1 - a_2 a_4 b_2 + a_2 a_7 b_1 + a_4 a_8 b_0 - a_5 a_7 b_0}{a_0 a_4 a_8 - a_0 a_5 a_7 - a_1 a_3 a_8 + a_1 a_5 a_6 + a_2 a_3 a_7 - a_2 a_4 a_6}$$

In [74]:
```
sol[x1]
```
Out[74]:
$$\frac{-a_0 a_5 b_2 + a_0 a_8 b_1 + a_2 a_3 b_2 - a_2 a_6 b_1 - a_3 a_8 b_0 + a_5 a_6 b_0}{a_0 a_4 a_8 - a_0 a_5 a_7 - a_1 a_3 a_8 + a_1 a_5 a_6 + a_2 a_3 a_7 - a_2 a_4 a_6}$$

In [75]:
```
sol[x2]
```
Out[75]:
$$\frac{a_0 a_4 b_2 - a_0 a_7 b_1 - a_1 a_3 b_2 + a_1 a_6 b_1 + a_3 a_7 b_0 - a_4 a_6 b_0}{a_0 a_4 a_8 - a_0 a_5 a_7 - a_1 a_3 a_8 + a_1 a_5 a_6 + a_2 a_3 a_7 - a_2 a_4 a_6}$$

1.6.3. En `scipy`

- scipy.linalg.solve(a, b)
- scipy.linalg.lu(a)

In [76]:
```python
from scipy.linalg import lu, solve

```

In [77]:
```python
A = [[-1, -1,  6,  9],
     [-5,  5, -3,  6],
     [ 7, -3,  5, -6],
     [ 3, -3, -2,  3]]

B = [-29, -54, 38, 41]

```

In [78]:
```python
lu(A)

```

Out[78]:
```
(array([[0., 0., 1., 0.],
        [0., 1., 0., 0.],
        [1., 0., 0., 0.],
        [0., 0., 0., 1.]]),
 array([[ 1.        ,  0.        ,  0.        ,  0.        ],
        [-0.71428571,  1.        ,  0.        ,  0.        ],
        [-0.14285714, -0.5       ,  1.        ,  0.        ],
        [ 0.42857143, -0.6       , -0.54285714,  1.        ]]),
 array([[ 7.        , -3.        ,  5.        , -6.        ],
        [ 0.        ,  2.85714286,  0.57142857,  1.71428571],
        [ 0.        ,  0.        ,  7.        ,  9.        ],
        [ 0.        ,  0.        ,  0.        , 11.48571429]]))
```

In [79]:
```python
solve(A, B)

```

Out[79]:
```
array([ 4., -8., -4., -1.])
```

Revisemos qué tipos de dato se arrojan:

In [80]:
```python
type(lu(A)[0])

```

Out[80]:
```
numpy.ndarray
```

In [81]:

```python
type(solve(A, B))

```

Out[81]:

```
numpy.ndarray
```

Comandos adicionales para el tratamiento de expresiones en `Sympy`:

- sp.collect()
- sp.simplify()
- sp.expand()