Capítulo 2

Arreglos

En muchas situaciones necesitamos procesar una colección de valores que están relacionados entre sí por algún método, por ejemplo, una series de temperaturas medidas a lo largo del mes, etc. Si usamos datos simples, esto puede llegar a ser muy tediosos y extremadamente complicado, por lo que la mayoría de los lenguajes de programación incluyen características de estructura de datos. La estructura de datos básica que soportan la mayoría de los lenguajes de programación son los arreglos (El concepto análogo será los vectores o matrices).

Un **arreglo** es una secuencia de posiciones de la memoria central a las que se puede acceder directamente, que contiene datos del mismo tipo y pueden ser seleccionados individualmente mediante el uso de subíndices.

Un arreglo es un conjunto finito y ordenado de elementos homogéneos. La propiedad 'ordenado' significa que el elemento primero, segundo, tercero,···, enésimo de un arreglo puede ser identificado. Los elementos de un arreglo son homogéneos, es decir, del mismo tipo de datos. Un arreglo puede estar compuesto de todos sus elementos de tipo 'character', otro puede tener elementos de tipo 'integer', etc.

2.1. Arreglos unidimensionales

El tipo más simple de arreglo es el arreglo unidimensional. El análogo de los arreglos unidimensionales son los vectores o mejor dicho matrices de rango 1. Sea un vector de una dimensión denominado VECTOR, que consta de n elementos se puede representar como:

VECTOR(1)
VECTOR(2)
⋮
VECTOR(I)
⋮
VECTOR(N)

El término en paréntesis denomina el lugar que tendrá en la computadora.

Como ejemplo de un arreglo unidimensional, se puede considerar el vector TEMPERATURA que contiene las temperaturas de cada hora registradas en una ciudad durante un día. Este vector constará de veinticuatro elementos de tipo real.

Cada elemento de un vector se puede procesar como si fuera una variable simple al ocupar una posición de la memoria.

Las operaciones que se pueden utilizar con vectores durante el proceso de resolución de un problema son:

- asignación
- lectura y escritura
- recorrido (acceso secuencial)
- actualizar (añadir, borrar, insertar)
- ordenación
- búsqueda

En general, las operaciones con vectores implican el procesamiento o tratamiento de los elementos individual del vector.

2.1.1. Asignación

La asignación de valores a un elemento del vector se realizará con la instrucción de asignación:

Podemos hacer la asignación de una lista vacia. Está lista se declara sólo usando `a=[ ]`, decimos que es una listaque no tiene ningún elemento.

Una manera de llenar esta lista sería usar la función `append`. La función `append` dice que le agregamos un elemento al arreglo. En realidad, concatenamos un elemento a la lista a la que se use. Digamos que se queremos convertir el arreglo vacio a a un arreglo que tenga como único elemento 1.

```
In [36]: a=[ ]

In [37]: a.append(1)

In [38]: a
Out[38]: [1]

In [39]: a.append(2)

In [40]: a
Out[40]: [1, 2]

In [41]: a.append('1')

In [42]: a
Out[42]: [1, 2, '1']
```

Ejercicios:

- Usando un ciclo for-in, haz un arreglo con 5 elementos y que todos esos elementos sean 5.

```
In [43]: b=[ ]

In [44]: for i in range(4):
   ....:     b.append(5)
   ....:
   ....:

In [45]: b
Out[45]: [5, 5, 5, 5]
```

- Usando un ciclo, haz un arreglo con elementos que el usuario introduzca los elementos.

```
In [46]: c=[ ]

In [47]: for i in range(3):
   ....:     x=int(raw_input('da el valor del elemento %d' %i))
   ....:     c.append(x)
   ....:
   ....:
da el valor del elemento 01
da el valor del elemento 11
da el valor del elemento 21

In [48]: c
Out[48]: [1, 1, 1]
```

Haz un programa que el usuario introduzca valores y que los promedie

```
c=[]
suma=0.0
for i in range(3):
x=float(raw_input('Da los elementos que se promediaran \t'))
c.append(x)
suma+=c[i]
promedio=suma/float(len(c))
print promedio
```

2.1.2. Ejercicio

Elabora un programa, en el cual introduzcas los elementos del arreglo por medio del teclado y te imprima en pantalla el elemento que es máximo.

2.2. Arreglos de varias dimensiones

Se puede definir matrices como arreglos multidimensionales, cuyos elementos se pueden referenciar por dos, tres o más subíndices. Los arreglos no unidimensionales los dividiremos en dos grandes grupos:

- arreglos bidimensionales (2-d)
- arreglos multidimensionales (3-d, o más)

2.2.1. Arreglos bidimensionales

El arreglo bidimensional se puede considerar como un vector de vectores, de manera más intuitiva una matriz. Es por consiguiente, un conjunto de elementos, todos del mismo tipo, en el cual, el orden de los componentes es significativo y en el que se necesita especificar dos índices para poder identificar cada elemento del arreglo.

Sin embargo, un subíndice no es suficiente para especificar un elemento de un arreglo bidimensional: por ejemplo, si el nombre del arreglo es M, no se puede indicar M[3], ya que no sabemos si es el tercer elemento de la primera fila o de la primera columna. Para evitar la ambigüedad, los elementos de un arreglo bidimensional se referirán con dos subíndices: el primer subíndice se refiere a la fila y el segundo subíndice se refiere a la columna. Por consiguiente M[2,3] se refiere al elemento de la segunda fila, tercera columna.

El elemento $B[i, j]$ también se puede representar por B_{ij} . Donde $i = 1, \cdots, M$, o bien $1 \leq i \leq M$ y $j = 1, \cdots, N$ o $1 \leq j \leq N$

En general, se considera que un arreglo bidimensional comienzan sus subíndices en 1, pero puede tener límites seleccionados por el usuario durante la elaboración del algoritmo. En general, el arreglo bidimensional B con su primer subíndice, variando desde un límite inferior a un límite superior. B[L1:U1, L2:U2] = (B[I,J]) donde $L1 \leq i \leq U1$ y $L2 \leq j \leq U2$.

2.3. Cuadrados mágicos

Un cuadrado mágico consiste en una distribución de números en filas y columnas, formando un cuadrado, de forma que los números de cada columna y diagonal suman lo mismo.

Los cuadrados mágicos, tradicionalmente se forman con números naturales consecutivos desde el 1 al n^2. Se pueden formar cuadrados mágicos con cualquier tipo de número, ya sea natural, entero, fraccionario, números complejos, potencia de un número, etc.

Ejemplo

2.6	1.1	1.4
0.5	1.7	2.9
2	2.3	0.8

Los cuadrados mágicos tienen su origen en China, donde eran conocidos varios siglos antes de Cristo. Según cuenta la leyenda, el primer cuadrado mágico fue revelado al emperador Yu a través de una tortuga divina que apareció en el río Luo, que lo llevaba grabado en su caparazón. Este emperador reinó en el siglo XII A.C.

De la China pasaron al Japón, al sudeste asiático, a la India y de allí a Arabia. Al mundo occidental llegó su conocimiento mucho más tarde, a través de los árabes. El primer cuadrado mágico del que se tiene documentación en Europa aparece en el grabado Melancolía de Alberto Durero.

Los cuadrados mágicos se clasifican de acuerdo con el número de celdas que tiene cada fila o columna. Un cuadrado con 3 celdas se dice que es de tercer orden. De 5 celdas será de quinto orden.

Propiedades de los cuadrados mágicos. Se puede sumar un escalar a un cuadro mágico, obteniendo otro cuadro mágico como resultado, teniendo que la nueva N', será $N' = N + c$. Donde c es el escalar que se suma, en el ejemplo siguiente $c = 2$.

$$\begin{bmatrix} 7 & 2 & 3 \\ 0 & 4 & 8 \\ 5 & 6 & 1 \end{bmatrix} + 2 = \begin{bmatrix} 9 & 4 & 5 \\ 2 & 6 & 10 \\ 7 & 8 & 3 \end{bmatrix}$$

Por otro lado, podemos restar un escalar a un cuadrado mágico, obteniendo otro cuadro mágico con N '$= N - c$. En el siguiente ejemplo, $c = 6$

$$\begin{bmatrix} 14 & 6 & 12 \\ 9 & 11 & 13 \\ 10 & 15 & 8 \end{bmatrix} \text{ - } 6 = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

De la misma forma, se puede multiplicar ó dividir un cuadro mágico por un escalar. Para el caso de la multiplicación la nueva N 'es N '$= cN$. En la división se tendrá N '$= \frac{N}{c}$

Otra de las operaciones válidas que se pueden realizar entre cuadros mágicos es la suma y resta de un cuadro mágico a otro, obteniendo otro cuadro mágico.

$$\begin{bmatrix} 7 & 2 & 3 \\ 0 & 4 & 8 \\ 5 & 6 & 1 \end{bmatrix} + \begin{bmatrix} 15 & 0 & 21 \\ 18 & 12 & 6 \\ 3 & 24 & 9 \end{bmatrix} = \begin{bmatrix} 22 & 2 & 24 \\ 18 & 16 & 14 \\ 8 & 30 & 10 \end{bmatrix}$$

Se puede intercambiar entre sí dos filas junto con dos columnas simétricas en bloque. Es decir, todos los números de una fila con todos los números de otra fila, haciendo lo mismo con los números de las filas y columnas que sean simétricas a ellas respecto de los ejes verticales, horizontal y de las diagonales.

Ejemplo

Intercambiamos la primera fila con la cuarta fila, junto con la primera y cuarta columna.

$$\begin{bmatrix} 8 & 4 & 3 & 15 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 6 & 10 & 13 & 1 \end{bmatrix} \begin{bmatrix} 6 & 10 & 13 & 1 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 8 & 4 & 3 & 15 \end{bmatrix} \begin{bmatrix} 6 & 10 & 13 & 1 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 8 & 4 & 3 & 15 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 10 & 13 & 6 \\ 14 & 9 & 2 & 5 \\ 0 & 7 & 12 & 11 \\ 15 & 4 & 3 & 8 \end{bmatrix}$$

Una propiedad muy interesante que hay en los cuadrados mágicos de 4x4, es aquella en donde tenemos que los números de casillas situadas en vértices de rectángulos concéntricos paralelos al cuadrado suman N.

Ejemplo

$$\begin{bmatrix} 8 & 4 & 3 & 15 \\ 5 & 9 & 2 & 14 \\ 11 & 7 & 12 & 0 \\ 6 & 10 & 13 & 1 \end{bmatrix} \begin{bmatrix} 8 & - & - & 15 \\ - & - & - & - \\ - & - & - & - \\ 6 & - & - & 1 \end{bmatrix} \begin{bmatrix} - & - & - & - \\ - & 9 & 2 & - \\ - & 7 & 12 & - \\ - & - & - & - \end{bmatrix}$$

$$\begin{bmatrix} - & - & - & - \\ 5 & - & - & 14 \\ 11 & - & - & 0 \\ - & - & - & - \end{bmatrix} \begin{bmatrix} - & 4 & 3 & - \\ - & - & - & - \\ - & - & - & - \\ - & 10 & 13 & - \end{bmatrix}$$

Una posible forma para construir cuadradados mágicos de orden 3, Sean a,b,c números enteros cualesquiera.

a+b	a-(b+c)	a+c
a-(b-c)	a	a+(b-c)
a-c	a+(b+c)	a-b

No existen métodos generales para construir cuadrados mágicos, sobre todo para los de orden par.

Ejemplos

1. *Construye un cuadrado mágico de 3x3, cuya suma sea 15.*

6	1	8
7	5	3
2	9	4

2. *Del 10 al 18. Halla el número de K, sabiendo que el cuadrado en el cual está inscrito es mágico y se compone de los números de 10 a 18.*

a	b	c
	K	
d	e	f

$$\begin{aligned} a+b+c &= N \\ a+K+f &= N \\ b+K+e &= N \\ c+K+d &= N \\ d+e+f &= N \end{aligned} \tag{2.1}$$

Sumando miembro a miembro de las tres igualdades centrales:

$$\begin{aligned} (a+b+c)+3K+(d+e+f) &= 3N \\ N+3K+N &= 3N \\ 3K &= N \\ K &= \frac{N}{3} \end{aligned} \tag{2.2}$$

En este cuadrado mágico N es la tercera parte de la suma de sus elementos

$$10+11+12+13+14+15+16+17+18 = 126 \Rightarrow N = 42 \tag{2.3}$$

Por lo que $K = 14$.

3. Coloque en cada cuadrado libre un número menor que 10 de tal manera que en cada fila y en cada columna haya un número que se repita exactamente dos veces y que la suma de cada fila y de cada columna sea 17.

2	5	2	**8**
2	**6**	7	2
8	5	2	2
5	1	**6**	5

4. De vuelta al cuadrado anterior y resuélvalo nuevamente

2	9	**3**	3
5	4	3	5
2	2	9	4
8	2	5	5

5. Completa los casilleros que faltan para que resulte mágico el siguiente cuadrado:

7		5
	8	
11		9

6. Construya un cuadrado mágico con los 9 primeros números pares de modo que las filas, columnas y diagonales sumen 30.

7. Construye un cuadrado mágico con los 9 primeros números impares de modo que las filas, columnas y diagonales sumen 27.

8. Construye un cuadrado mágico de 4x4 (suma=34). Los elementos de cada una de las nueve matrices 2x2 que componen el cuadrado también deben sumar 34.

Los posibles programas que podemos realizar serán:

```
from numpy import *
x=zeros(9).reshape(3,3)
print'programa que solo funciona para cuadrados magicos de 3x3'
n=float(raw_input('Introduce el numero, por el cual se introducirá la condición de cuadrado mágico \t'))
x[0][0]=float(raw_input('Da el elemento 1,1 \t'))
#x[1][2]=float(raw_input('Da el elemento 2,3 \t'))
a=n/3.0
b=x[0][0]-a
c=n-a-b
#x[0][0]=a+b
x[0][1]=a-(b+c)
x[0][2]=a+c
x[1][0]=a-(b-c)
x[1][1]=a
x[1][2]=a+(b-c)
x[2][0]=a-c
x[2][1]=a+(b+c)
x[2][2]=a-b
print 'Tu cuadrado magico es: '
for i in range(3):
      print x[i][0],x[i][1],x[i][2]
```

Una manera de crear un cuadrado mágico de 5 x 5 es con el siguiente cuadro.

$a+b_1$	$a+c_1$	$a-(b_1-b_2)$	$a-c_1$	$a+b_2$
a+d	$a+b_3$	$a-(b_2+b_4)$	$a-b_4$	a-d
$a-(b_1-b_2)$	$a-(c_1-c_2)$	a	$a+(c_1-c_2)$	$a+(b_1-b_2)$
a-d	$a-b_4$	$a+(b_3+b_4)$	$a-b_3$	a+d
$a-b_2$	$a-c_2$	$a+(b_1-b_2)$	$a+c_2$	$a-b_1$

Del cuadro anterior se ve que para construir un cuadro mágico de 5x5, por lo menos se necesitan conocer 7 elementos, esto para conocer los 25 restantes. El algoritmo queda:

2.4. Matrices

Ya hemos visto las propiedades importantes de los cuadrados mágicos. Como hemos observado los cuadrados mágicos son matrices cuadradas con propiedades muy bien determinadas.

Dos matrices de mxn, digamos A y B se definen como iguales si y sólo si sus elementos correspondientes son iguales, esto es:

$$A_{ij} = B_{ij}$$

Para $1 \leq i \leq m$ y $1 \leq j \leq n$. Al conjunto de todas las matrices de mxn podemos definir las siguientes operaciones:

$$(A+B)_{ij} = A_{ij} + B_{ij}$$

La suma de dos matrices da como resultado una tercera matriz.

$$(cA_{ij}) = cA_{ij}$$

La multiplicación de un escalar por una matriz da como resultado otra matriz.

A continuación se muestra un código que hace la suma de matrices. Las matrices en este código se ven como lista de listas.

```
# -*- coding: utf-8 -*-
m=int(raw_input('da numero de filas'))
n=int(raw_input('da numero de columnas'))
A=[]
B=[]
for i in range(m):
        A.append([0]*n)
        B.append([0]*n)
print'Introduce los elementos de la primera matriz'
for i in range(m):
        for j in range(n):
                A[i][j]=float(raw_input('da la componente (%d,%d)'%(i,j)))
print'Introduce los elementos de la segunda matriz'
for i in range(m):
        for j in range(n):
                B[i][j]=float(raw_input('da la componente (%d,%d)'%(i,j)))
#Creamos una matriz que guardará el resultado de A+B
C=[]
```

```
for i in range(m):
        C.append([0]*n)
for i in range(m):
        for j in range(n):
                C[i][j]=A[i][j]+B[i][j]
print 'la suma de las matrices que diste es:'
for i in range(m):
        print C[i]
```

Definimos la multiplicación de matrices de la siguiente manera:

Sean A una matriz de mxn y B una matriz de nxp. Definimos el producto de A por B, denotado por AB, como la matriz de mxp tal que:

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Para $1 \leq i \leq m$ y $1 \leq j \leq$. Ejemplo:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 4 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ 3 \end{bmatrix}$$

```
# -*- coding: utf-8 -*-
m=int(raw_input('da numero de filas de la primera matriz'))
n=int(raw_input('da numero de columnas de la primera matriz y numero de renglones de la segunda'))
p=int(raw_input('da numero de columnas de la segunda matriz'))
A=[]
B=[]
for i in range(m):
        A.append([0]*n)
for i in range(n):
        B.append([0]*p)
print'Introduce los elementos de la primera matriz'
for i in range(m):
        for j in range(n):
                A[i][j]=float(raw_input('da la componente (%d,%d)'%(i,j)))
print'Introduce los elementos de la segunda matriz'
for i in range(n):
        for j in range(p):
                B[i][j]=float(raw_input('da la componente (%d,%d)'%(i,j)))
#Creamos una matriz que guardará el resultado de AB
C=[]
for i in range(m):
        C.append([0]*p)
for i in range(m):
        for j in range(p):
                s=0.0
                for k in range(n):
                        s+=A[i][k]*B[k][j]
                        C[i][j]=s
```

```
print 'la multiplicacion de las matrices que diste es:'
for i in range(m):
        print C[i]
```

2.4.1. Numpy

Python tiene la librería Numpy que facilita la creación de arreglos multidimensionales. Acrónimo de "Python numérico" en inglés, esta librería provee una de las herramientas más útiles en el cómputo ciéntifico: los arreglos n-dimensionales. Para definir un arreglo de este tipo se utiliza array:

Acrónimo de "Python numérico" en inglés, esta librería provee una de las herramientas más útiles en el cómputo científico: los arreglos n-dimensionales. Para definir un arreglo de este tipo se utiliza *array*:

```
>>a=array([1.2,2.2,4.5])
```

Cabe destacar que se utilizan tanto los paréntesis como los paréntesis cuadrados, en ese orden, al momento de definir un arreglo entrada por entrada. Si en cambio se requiere de un arreglo de números consecutivos se puede utilizar *arange*:

```
>>> b=arange(0.1,1,0.1)
>>> print b
[ 0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
>>>
```

los argumentos de *arange* son (*origen*, *final*, *tamaño del paso*). En caso de que no se especifíque el tamaño del paso, éste será por default 1. Si no se específica el origen, éste será por default 0.

También se pueden definir arreglos con puros ceros o unos:

```
>>> b=zeros(10).reshape(2,5)
>>> b
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> b=ones(10)
>>> b
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

`reshape(2,5)` da la forma en dos filas y 5 columnas.

Se pueden hacer muchas cosas con los arreglos. Tomando nuestro arreglo `b` del ejemplo anterior,

- `b.ndim`: nos dice el número de dimensiones del arreglo.
- `b.size`: nos dice el número de elementos que contiene el arreglo.
- `b.shape`: nos dice la forma del arreglo en número de filas y columnas. También puede cambiar la forma del arreglo usando `b.shape=2,5` donde 2 y 5 son las filas y columnas respectivamente.
- `b.transpose()`: transpone el arreglo. Si era un arreglo de 2 filas y 5 columnas, pasa a ser de 5 filas y 2 columnas.
- `b.ravel()`: reordena el arreglo en una sola fila

- `b.sum()`: suma todos sus elemetos. Se pueden agregar argumentos a `sum()` para sumar los elementos de las columnas o de las filas usando `axis=0` ó `axis=1`.
- `b.min()`: da el valor mínimo de los elementos. También se pueden usar como argumentos los ejes.
- `b.max()`: lo mismo que el anterior sólo que ahora es el máximo.

Una de las grandes ventajas de **numpy** es que permite un manejo sencillo de los arreglos. De esta manera, si dos arreglos son del mismo tamaño podemos sumarlos o restarlos.

```
>>> a=ones(3)
>>> a
array([ 1.,  1.,  1.])
>>> b=ones(10)
>>> b
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
>>> a=ones(10)
>>> a+b
array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
>>> a-b
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

También se pueden añadir reales a todos los números del arreglo

```
>>> a+sin(pi/2)
array([ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.])
```

Si los arreglos se usan como matrices `dot(a,b)` regresa el producto de las dos matrices con las mismas condiciones que se requieren para un producto de matrices.

Capítulo 3

Análisis Numérico de problemas matriciales

La manipulación matricial asociada a encontrar eigenvalores o a resolver ecuaciones lineales simultáneas frecuentemente es la estructura principal del trabajo para resolver sistemas físicos.

3.1. Sistema de ecuaciones lineales

Un sistema de ecuaciones algebraicas tiene la forma:

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n &= b_1 \\ A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n &= b_2 \\ \vdots &\quad \vdots \\ A_{n1}x_1 + A_{n2}x_2 + \cdots + A_{nn}x_n &= b_n \end{aligned}$$

Donde A_{ij} y b_j son constantes conocidas. El sistema de ecuaciones lineales puede ser representado como:

$$A\vec{x} = \vec{b} \rightarrow (A)_{11}\, A_{12} \cdots A_{1n} A_{21} A_{22} \cdots A_{2n} \vdots \cdots \vdots A_{n1} A_{n2} \cdots A_{nn}\, (x)_1\, x_2 \vdots x_n = (b)_1\, b_2 \vdots b_n \tag{3.1}$$

Se sabe que la solución es única cuando la matriz es no singular, es decir, cuando ningún renglón es linealmente independiente. Si la matriz es singular se tienen dos posibles opciones, que exista una infinidad de soluciones o que no exista ninguna.

Por ejemplo, $x + 3y = 2$ y $4x + 12y = 8$ se ve que cualquier combinación lineal que satisface la primera, satisface la segunda, por lo que se tiene una infinidad de soluciones. Pero si el sistema de ecuaciones cambia $x + 3y = 2$ y $4x + 12y = 0$, se tiene que la segunda ecuación contradice la primera, así que no existe la solución para este sistema de ecuaciones.

La manera más natural de resolver sistema de ecuaciones lineales es haciendo operaciones algebraicas básicas y luego la regla de Cramer, es decir, A^{-1} será proporcional a la matriz de cofactores de A. Para

Metodo	Forma inicial	Forma final
Eliminación Gaussiana	$A\vec{x} = \vec{b}$	$U\vec{x} = \vec{c}$
Gauss-Jordan	$A\vec{x} = \vec{b}$	$I\vec{x} = \vec{c}$

Tabla 3.1: Se usa U como una matriz triangular superior e I como la matriz identidad.

encontrar esta relación se debe calcular N^2 determinantes de las matrices de dimensión $(N-1) \times (N-1)$. Haciendo esto con mucho cuidado encontramos la siguiente relación:

$$det(A) = \sum_{p} (-1)^p A_{1p_1} A_{2p_2} \cdots A_{Np_N} \tag{3.2}$$

Donde P es una de la $N!$ permutaciones de las N columnas, P_i es el i-ésimo elemento de P. Esta fórmula implica que si se tiene una N grande, se tendrán que hacer muchísimas operaciones. Por ejemplo, si se tiene $N = 20$, se tendrán que hacer 2×10^{18} multiplicaciones. Como se puede ver, este no es el método más eficiente.

En la actualidad, existen 2 tipos de métodos de solución para ecuaciones algebraicas:

- Métodos Directos
- Métodos Iterativos

La característica de los métodos directos es que transforman las ecuaciones originales en ecuaciones equivalente (ecuaciones que tienen la misma solución). Las transformaciones siguen las operaciones elementales

1. Multiplicación de renglones por alguna constante.
2. Intercambio de renglones
3. Suma de renglones entre si

Los 2 métodos mas populares para resolver sistemas de ecuaciones lineales son:

3.1.1. Método de Eliminación Gaussiana

Aquí la idea es utilizar varias operaciones elementales, tal que, sin importar el número de operaciones que se haga, se tenga siempre matrices equivalentes a A.

Para llevar a cabo el programa se deben de considerar varias cosas. Por ejemplo, puede ser que se tenga que en algún punto en el procedimiento, el elemento de la diagonal sea cero. En este caso, un intercambio de columnas dará un nuevo valor distinto de cero. A este intercambio se le llama método de pivoteo, si este método no se puede llevar acabo, entonces la matriz es singular. Se debe de considerar que la máquina redondeará las operaciones, por lo que al final se tendrán muchos errores de redondeo y la solución del sistema encontrado puede variar mucho de la solución real. Se debe considerar entonces que todas las entradas deben tener la misma magnitud.

El procedimiento general de eliminación gaussiana que se aplica al sistema lineal es $a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1$
$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2$
$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = b_1$
$\vdots$
$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_1$

De este sistema de ecuaciones podemos ver que la matriz aumentada será:

$$A = (\,a\,)_{11}\, a_{12}a_{13}\cdots a_{1n}b_1a_{21}a_{22}a_{23}\cdots a_{2n}b_2a_{31}a_{32}a_{33}\cdots a_{3n}b_3\vdots a_{n1}a_{n2}a_{n3}\cdots a_{nn}b_n \tag{3.3}$$

Decimos que la fórmula general para crear ceros será:

$$E_j - \left[\frac{a_{j1}}{a_{11}}\right] E_i \to E_j \tag{3.4}$$

Esto sólo si a_{11} es distinto de cero. Se realiza para cada $j = 2, 3, \cdots, n$ para eliminar el coeficiente de x_1 en cada uno de estos renglones. Aunque se espera que los elementos de los renglones $2, 3, \cdots, n$ cambien para facilitar la notación denotamos nuevamente con a_{ij} el elemento del i-ésimo renglón y la j-ésima columna. Teniendo presente esto, aplicamos el procedimiento secuencial para $i = 2, 3, \cdots, n-1$ y realizamos la operación:

$$E_j - \left[\frac{a_{ji}}{a_{ii}}\right] E_i \to E_j \tag{3.5}$$

Para toda $j = i+1, i+2, \cdots, n$ a condición de que $a_{ii} \neq 0$. Con ello se suprime (es decir, se transforma en cero el coeficiente) x_i en cada renglón debajo del i-ésimo para todos los valores de $i = 1, 2, \cdots, n-1$. La matriz resultante tiene la forma de una triangular superior.

$$A = (\,a\,)_{11}\, a_{12}a_{13}\cdots a_{1n}b_10a_{22}a_{23}\cdots a_{2n}b_200a_{33}\cdots a_{3n}b_3\vdots 000\cdots a_{nn}b_n \tag{3.6}$$

De esta manera escalonamos el sistema y llevamos el sistema con forma $A\vec{x} = \vec{b}$ a un sistema de la forma $U\vec{x} = \vec{c}$.

Desarrollo Numérico

Empezaremos por importar el módulo Numpy, el cual nos permite utilizar de manera atractiva y fácil los arreglos y matrices.

```
In [1]: from numpy import *

In [2]: A=matrix([[1,2,3],[4,5,6],[7,8,10]])

In [3]: A
Out[3]:
matrix([[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8, 10]])
```

```
"Convirtiendo el elemento A[1,0] en 0"

In [4]: A[1]-=(A[1,0]/A[0,0])*A[0]

In [5]: A
Out[5]:
matrix([[ 1,  2,  3],
        [ 0, -3, -6],
        [ 7,  8, 10]])

"A[2,0]=0"
In [6]: A[2]-=(A[2,0]/A[0,0])*A[0]

In [7]: A
Out[7]:
matrix([[  1,   2,   3],
        [  0,  -3,  -6],
        [  0,  -6, -11]])

"Para convertir A[2,1]=0"
In [8]: A[2]-=(A[2,1]/A[1,1])*A[1]

In [9]: A
Out[9]:
matrix([[ 1,  2,  3],
        [ 0, -3, -6],
        [ 0,  0,  1]])
```

De esta manera ya escalonamos la matriz y encontramos la matriz triangular superior que dictamina el método de Gauss, debe ser encontrada. Para generalizar el procedimiento seria bueno escribir un pequeño fragmento de programa que haga por si solo la elección de los coeficientes. Algo útil es usar dos for-in (anidados) y que nos fijemos como se recorren los índices, por lo que vemos cual es el índice *lento* y el índice *rápido*.

El cacho de programa queda:

```
In [40]: A=matrix([[1,2,3],[4,5,6],[7,8,10]])

In [41]: for i in range(3):
    ....:     for j in range(3):
    ....:         A[j]-=(A[j,i]/A[i,i])*A[i]
    ....:
    ....:

In [44]: A
Out[44]:
matrix([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

¡Que pasa aquí! Nuestro fragmento de programa hizo la matriz ceros. Lo que ocurre es que esta haciendo el caso $A[0]- = (A[0,0]/A[0,0]) * A[0]$, lo cual hace que toda la matriz se haga cero. Por lo que se tiene que excluir el caso en que $i = j$. Esto se soluciona:

```
In [45]: A=matrix([[1,2,3],[4,5,6],[7,8,10]])

In [46]: for i in range(3):
    for j in range(i+1,3):
        A[j]-=(A[j,i]/A[i,i])*A[i]
   ....:
   ....:

In [49]: A
Out[49]:
matrix([[ 1,  2,  3],
        [ 0, -3, -6],
        [ 0,  0,  1]])
In [50]: A=matrix([[4,-2,1],[-2,4,-2],[1,-2,4]])

In [52]: A
Out[52]:
matrix([[ 4, -2,  1],
        [-2,  4, -2],
        [ 1, -2,  4]])

In [53]: for i in range(3):
    for j in range(i+1,3):
        A[j]-=(A[j,i]/A[i,i])*A[i]
   ....:
   ....:

In [56]: A
Out[56]:
matrix([[ 4, -2,  1],
        [ 2,  2, -1],
        [ 3,  0,  3]])
```

Entonces, si cambiamos a otra matriz tenemos que el metodo no funciona. La razón principal se debe a que hemos hechos matrices enteras, por lo que la división entre enteros solo nos devuelve la parte entera. Para solucionar esto, tenemos que hacer las matrices como reales, o de punto flotante.

```
In [61]: A=matrix([[4,-2,1],[-2,4,-2],[1,-2,4]],float)

In [62]: A
Out[62]:
matrix([[ 4., -2.,  1.],
        [-2.,  4., -2.],
        [ 1., -2.,  4.]])

In [63]: for i in range(3):
    for j in range(i+1,3):
        A[j]-=(A[j,i]/A[i,i])*A[i]
   ....:
```

```
   ....:

In [66]: A
Out[66]:
matrix([[ 4. , -2. ,  1. ],
        [ 0. ,  3. , -1.5],
        [ 0. ,  0. ,  3. ]])
```

Que pasa si consideramos el caso en que se tiene un elemento igual a cero, digamos el elemento a[1,0]. Para hacer mas eficiente nuestro código, excluimos los casos en que los elementos que queremos hacer cero ya sean cero.

```
In [73]: A=matrix([[4,-2,1],[0,4,-2],[1,-2,4]],float)

In [74]: for i in range(3):
   ....:     for j in range(i+1,3):
   ....:         if A[j,i]!=0:
   ....:             A[j]-=(A[j,i]/A[i,i])*A[i]
   ....:
   ....:

In [75]: A
Out[75]:
matrix([[ 4., -2.,  1.],
        [ 0.,  4., -2.],
        [ 0.,  0.,  3.]])
```

Solo falta tener la posibilidad de cambiar renglones. Por ejemplo, tenemos que el primer renglón de la matriz tiene 2 ceros y luego algo distinto de cero. En este caso, quisiéramos cambiar el primer renglón con el tercero. Por lo que usamos el siguiente fragmento de código:

```
In [95]: A
Out[95]:
matrix([[ 0.,  0.,  1.],
        [ 0.,  4., -2.],
        [ 1., -2.,  4.]])

In [96]: s=0

In [97]: for i in range(3):
    if A[0,i]==0:
        s+=1
   ....:
   ....:

In [100]: s
Out[100]: 2

In [101]: A[0],A[s]=A[s].copy(),A[0].copy()

In [102]: A
Out[102]:
```

```
matrix([[ 1., -2.,  4.],
        [ 0.,  4., -2.],
        [ 0.,  0.,  1.]])
```

Hasta este momento solo estamos escalonando la matriz, falta resolver el sistema de ecuaciones lineales, Para esto introducimos el vector $\vec{b}$. Lo que se le haga a la matriz A, se le tendrá que hacer al vector b. Por lo que el código (en conjunto) queda:

```
s=0
for i in range(3):
        if A[0,i]==0:
        s+=1
if A[0,s]==0: A[0],A[s]=A[s].copy(),A[0].copy()
print A
for i in range(3):
        for j in range(i+1,3):
                fact=A[j,i]/A[i,i]
                if A[j,i]!=0:
                        A[j]=A[j]-fact*A[i]
                        b[j]-=fact*b[i]
```

Para obtener explícitamente el valor del $\vec{b}$, necesitamos usar el método de sustitución hacia atrás. Donde la ultima ecuación que se tiene es $A_{nn}x_n = b_n$, de esta resolvemos $x_{nn} = \frac{b_n}{A_n n}$, haciendo lo mismo con la ecuación k-esima $A_{kk}x_k + A_{k,k+1}x_{k+1} + \cdots + A_{kn}x_n = b_k$. La solución es:

$$x_k = (b_k - \sum_{j=k+1}^{n} A_{kj}x_j)\frac{1}{A_{kk}}$$

Utilizando que $\sum A_{kj}x_j = A \cdot \vec{x}$ y haciendo la función `gauss`, tenemos:

```
m=matrix([[4,-2,1],[-2,4,-2],[1,-2,4]],float)
n=matrix([[11],[-16],[17]],float)
def gauss(A,b):
        n=len(b)
        s=0
        for i in range(n):
                if A[0,i]==0:
                        s+=1
        if A[0,s]==0: A[0],A[s+1]=A[s+1].copy(),A[0].copy()
        print A
        for i in range(n):
                for j in range(i+1,n):
                        fact=A[j,i]/A[i,i]
                        if A[j,i]!=0:
                                A[j]=A[j]-fAct*A[i]
                                b[j]-=fact*b[i]

        for i in range(n-1,-1,-1):
                b[i]=(b[i]-dot(A[i,i+1:n],b[i+1:n]))/float(A[i,i])
        return b
```

```
print gauss(m,n)
```

Si quisiéramos realizar la inversa, habrá que hacer la matriz diagonal, por lo que haremos cero a todos los elementos fuera de la diagonal. Este procedimiento se llama *Método general de pivoteo*. Usando este procedimiento encontraremos la inversa.

3.2. Inversión de Matrices

Consideremos el problema de invertir una matriz cuadrada A. Para ello se tiene que resolver el siguiente sistema lineal:

$$Ax = b \tag{3.7}$$

Donde el x es un vector desconocido y b es un vector que si conocemos.

3.3. Ejemplo de inversa

Encontrar la inversa de la siguiente matriz:

$$A = (\,1\,)\,2 - 1210 - 112 \tag{3.8}$$

- Paso 1: i=1, j=2

$$E_2 - \left[\frac{a_{21}}{a_{11}}\right] E_1 \rightarrow E_2 \tag{3.9}$$

$$E_2 - \left[\frac{a_{21}}{a_{11}}\right] E_1 \rightarrow E_2 \tag{3.10}$$

$$E_2 - 2E_1 \rightarrow E_2 \tag{3.11}$$

$$A = (\,1\,)\,2 - 11000 - 32 - 210 - 112001 \tag{3.12}$$

- Paso 2: i=1, j=3

$$E_3 - \left[\frac{a_{31}}{a_{11}}\right] E_1 \rightarrow E_3 \tag{3.13}$$

$$E_3 - \left[\frac{-1}{1}\right] E_1 \rightarrow E_3 \tag{3.14}$$

$$E_3 + E_1 \rightarrow E_3 \tag{3.15}$$

$$A = (\,1\,)\,2 - 11000 - 32 - 210031101 \tag{3.16}$$

- Paso 3: i=2, j=3

$$E_3 - \left[\frac{a_{32}}{a_{22}}\right] E_2 \rightarrow E_3 \tag{3.17}$$

$$E_3 - \left[\frac{3}{-3}\right] E_2 \rightarrow E_3 \tag{3.18}$$

$$E_3 + E_2 \rightarrow E_3 \tag{3.19}$$

$$A = (\,1\,)\,2 - 11000 - 32 - 210003111 \tag{3.20}$$

- Paso 4: i=3, j=2

$$E_2 - \left[\frac{a_{23}}{a_{33}}\right] E_3 \rightarrow E_2 \tag{3.21}$$

$$E_2 - \left[\frac{2}{3}\right] E_3 \rightarrow E_2 \tag{3.22}$$

$$E_2 - \frac{2}{3} E_3 \rightarrow E_2 \tag{3.23}$$

$$A = (\,1\,)\,2 - 1100030\frac{-4}{3}\frac{1}{3}\frac{-2}{3}003 - 111 \tag{3.24}$$

- Paso 5: i=3, j=1

$$E_1 - \left[\frac{a_{13}}{a_{33}}\right] E_3 \rightarrow E_1 \tag{3.25}$$

$$E_1 - \left[\frac{-1}{3}\right] E_3 \rightarrow E_1 \tag{3.26}$$

$$E_1 - \frac{1}{3} E_3 \rightarrow E_1 \tag{3.27}$$

$$A = (\,1\,)\,20\frac{2}{3}\frac{1}{3}\frac{1}{3}030\frac{-4}{3}\frac{1}{3}\frac{-2}{3}003 - 111 \tag{3.28}$$

- Paso 6: i=2, j=1

$$E_1 - \left[\frac{a_{12}}{a_{22}}\right] E_2 \rightarrow E_1 \tag{3.29}$$

$$E_1 - \left[\frac{2}{-3}\right] E_2 \rightarrow E_1 \tag{3.30}$$

$$E_1 + \frac{2}{3} E_2 \rightarrow E_1 \tag{3.31}$$

$$A = (\,1\,)\,00\frac{-2}{9}\frac{5}{9}\frac{-1}{9}030\frac{-4}{3}\frac{1}{3}\frac{-2}{3}003 - 111 \tag{3.32}$$

  De está manera hemos llegado a una matriz diagonal, de aquí falta hacer la matriz original en la identidad. Para hacer la matriz identidad sólo tenemos que dividir los respectivos elementos.

- Paso 8 Multiplicar el renglón 2 y el renglón 3 por $\frac{1}{3}$. De esta manera llegamos a la identidad

$$A = (\,1\,)\,00\frac{-2}{9}\frac{5}{9}\frac{-1}{9}010\frac{4}{9}\frac{-1}{9}\frac{2}{9}001\frac{-1}{3}\frac{1}{3}\frac{1}{3} \tag{3.33}$$

3.4. Algebra matricial con Numpy

Como se ha visto hasta el momento, cuando se utilizan arreglos en Python es conveniente llamar al modulo Numpy. Numpy ya tiene varias funciones básicas de álgebra matricial, listas para usarse. Hagamos un pequeño análisis de estas funciones:

- `numpy.linspace(start, stop, num=50, endpoint=True, retstep=False)` Regresa una muestra de números igualmente espaciados sobre un intervalo específico. El `num` indica el numero de muestras a generarse, el default es 50. El `endpoint` es booleano, es opcional, si es *False* no se incluye el número final y el paso de la secuencia la escoge más uniforme. Por ultimo, `retstep` es de tipo booleano y es opcional, si es verdadero imprime el tamaño del paso.

```
In [5]: linspace(2.0,3.0,num=5)
Out[5]: array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ])

In [7]: linspace(2.0,3.0,num=5, endpoint=False)
Out[7]: array([ 2. ,  2.2,  2.4,  2.6,  2.8])

In [8]: linspace(2.0,3.0,num=5, retstep=True)
Out[8]: (array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ]), 0.25)
```

- `numpy.reshape(a,nueva_forma)` Da una nueva forma a un arreglo sin cambiar su contenido. Depende de los siguientes parametros; `a` indica el arreglo, `nueva_forma` indica cual será la nueva forma del arreglo.

```
In [12]: a=array([[1,2,3],[4,5,6]])

In [21]: a
Out[21]:
array([[1, 2, 3],
       [4, 5, 6]])

In [22]: reshape(a,6)
Out[22]: array([1, 2, 3, 4, 5, 6])

In [25]: reshape(a,(3,-1))
Out[25]:
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Las dos opciones anteriores se pueden combinar dando como resultado lo siguiente.

```
In [29]: t=linspace(1,30,30)

In [30]: t
Out[30]:
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
        12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,  22.,
        23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.])

In [31]: t=linspace(1,30,30).reshape(5,6)

In [32]: t
Out[32]:
array([[  1.,   2.,   3.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,  20.,  21.,  22.,  23.,  24.],
       [ 25.,  26.,  27.,  28.,  29.,  30.]])
```

Si quisiéramos escoger una submatriz de la matriz anterior, podemos hacerlo con la siguiente opción:

```
In [33]: t[1:-1:2,2:]
```

```
Out[33]:
array([[  9.,  10.,  11.,  12.],
       [ 21.,  22.,  23.,  24.]])
```

Del arreglo t se escogen dos renglones, así que decimos cuales y cuantos 1:-1:2

```
In [35]: t[1:-1:2]
Out[35]:
array([[  7.,   8.,   9.,  10.,  11.,  12.],
       [ 19.,  20.,  21.,  22.,  23.,  24.]])
```

Si ponemos

```
In [36]: t[1:-1]
Out[36]:
array([[  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,  20.,  21.,  22.,  23.,  24.]])
```

Selecciona todos los renglones que están entre el renglón 1 y -1. Luego escogemos las columnas. En este caso lo que hacemos es decir de la dos para adelante al poner 2:

Otro ejemplo:

```
In [39]: t[:-2,:-1:2]
Out[39]:
array([[  1.,   3.,   5.],
       [  7.,   9.,  11.],
       [ 13.,  15.,  17.]])
```

Tratemos de hacer vectorial una función constante. Supongamos que se tiene una función constante:

```
In [47]: x=array([[1,2,3,4,5],[1,1,1,1,1],[5,4,3,2,1]])

In [48]: x
Out[48]:
array([[1, 2, 3, 4, 5],
       [1, 1, 1, 1, 1],
       [5, 4, 3, 2, 1]])

In [49]: def f(x):
   ....:     return 2
   ....:

In [50]: x=array([[1,2,3,4,5],[1,1,1,1,1],[5,4,3,2,1]])

In [51]: f(x)
Out[51]: 2
```

Esta función acepta un arreglo como argumento, pero regresará un flotante, mientras que si la función estuviera hecha de forma correcta, se tendría de regreso un arreglo del mismo tamaño de x pero constante. Haciendo una corrección a esto:

```
In [71]: def fv(x):
             return zeros(x.shape,x.dtype) + 2
   ....:

In [73]: fv(x)
Out[73]:
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])
```

Pero la función correcta y más óptima será cuando la función funciona tanto para arreglos como para escalares. Para la cual propondremos usar: `isinstance(objeto, classinfo)`. Este comando es de tipo booleano y regresa *True* si el argumento del objeto corresponde con lo definido en la clase.

```
In [80]: def f(x):
   ....:     if isinstance(x,(float,int)):
   ....:         return 2
   ....:     else:
   ....:         return zeros(x.shape,x.dtype) + 2
   ....:
   ....:

In [81]: f(x)
Out[81]:
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])

In [82]: f(5)
Out[82]: 2
```

- `numpy.dot(a,b)`: Se le dan dos arreglos. Para arreglos de dos dimensiones es equivalente a la multiplicación matricial, mientras que para arreglos en una dimensión realiza la operación de producto punto entre vectores. Para cosas complejas, no hace la operación entre términos conjugados.

  ```
  In [78]: import numpy

  In [80]: numpy.dot([2j,3j],[2j,3j])
  Out[80]: (-13+0j)

  In [81]: a=[[1,0],[0,1]]

  In [82]: b=[[4,1],[2,2]]

  In [83]: numpy.dot(a,b)
  Out[83]:
  array([[4, 1],
         [2, 2]])
  ```

- `numpy.dot(a,b)` Regresa el producto de dos vectores. Si el primer argumento es complejo, se toma el conplejo conjugado del primer término para hacer el producto punto.

  Tiene las mismas cosas que la función anterior.

```
In [86]: a=numpy.array([1+2j,3+4j])

In [87]: b=numpy.array([5+6j,7+8j])

In [88]: numpy.vdot(a,b)
Out[88]: (70-8j)\right]

In [90]: numpy.vdot(b,a)
Out[90]: (70+8j)

In [91]: numpy.vdot(b,b)
Out[91]: (174+0j)
```

- `numpy.linal.inv(a)`: Calcula la inversa de la matriz A (matriz cuadrada). Muestra un mensaje de error si la matriz A es singular o no es cuadrada

```
In [100]: c= [[1.,2.],[3.,4.]]

In [101]: c
Out[101]: [[1.0, 2.0], [3.0, 4.0]]

In [102]: numpy.linalg.inv(c)

Out[102]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [106]: numpy.dot(c,numpy.linalg.inv(c))
Out[106]:
array([[  1.00000000e+00,   1.11022302e-16],
       [  0.00000000e+00,   1.00000000e+00]])
```

- `numpy.linalg.eig(a)`: Calcula los eigenvectores y los eigenvalores de la matriz cuadrada a. Saca un mensaje de error si el cálculo de los eigenvalores no converge.

```
In [115]: numpy.linalg.eig(c)
Out[115]:
(array([-0.37228132,  5.37228132]),
 array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
```