

# Librería en Python para Matemáticas II (Álgebra Lineal)

<https://github.com/mbujosab/LibreriaDePythonParaMates2>

Marcos Bujosa

December 17, 2019

# Índice

Declaración de intenciones . . . . .	3
<b>1 Código principal de la librería. Las clases Vector, Matrix y T</b>	<b>4</b>
1.1 La clase <b>Vector</b> . . . . .	4
1.1.1 Implementación de los vectores en la clase <b>Vector</b> . . . . .	5
1.2 La clase <b>Matrix</b> . . . . .	8
1.2.1 Implementación de las matrices en la clase <b>Matrix</b> . . . . .	9
1.3 Operadores selectores . . . . .	11
1.3.1 Operador selector por la derecha para la clase <b>Vector</b> . . . . .	11
1.3.2 Operador selector por la izquierda para la clase <b>Vector</b> . . . . .	13
1.3.3 Operador selector por la derecha para la clase <b>Matrix</b> . . . . .	13
1.3.4 Operador transposición de una <b>Matrix</b> . . . . .	14
1.3.5 Operador selector por la izquierda para la clase <b>Matrix</b> . . . . .	16
1.4 Operaciones con vectores y matrices . . . . .	18
1.4.1 Suma de <b>Vectores</b> . . . . .	18
1.4.2 Producto de un <b>Vector</b> por un escalar a su izquierda . . . . .	18
1.4.3 Producto de un <b>Vector</b> por un escalar, un <b>Vector</b> o una <b>Matrix</b> a su derecha . . . . .	19
1.4.4 Igualdad entre vectores . . . . .	21
1.4.5 Suma de matrices . . . . .	21
1.4.6 Producto de una <b>Matrix</b> por un escalar a su izquierda . . . . .	22
1.4.7 Implementación . . . . .	23
1.4.8 Producto de una <b>Matrix</b> por un escalar, un <b>Vector</b> o una <b>Matrix</b> a su derecha . . . . .	23
1.4.9 Implementación . . . . .	25
1.5 La clase transformación elemental <b>T</b> . . . . .	27
1.5.1 Implementación . . . . .	29
1.5.2 Transposición de transformaciones elementales . . . . .	31
1.5.3 Inversión de transformaciones elementales . . . . .	32
1.6 Transformaciones elementales de una <b>Matrix</b> . . . . .	34
1.6.1 Transformaciones elementales de las columnas de una <b>Matrix</b> . . . . .	34
1.6.2 Transformaciones elementales de las filas de una <b>Matrix</b> . . . . .	35
1.7 Librería completa . . . . .	37
<b>2 Algoritmos del curso</b>	<b>39</b>
2.1 Escalonamiento de una matriz por eliminación Gaussiana . . . . .	39
2.1.1 Variantes que guardan los pasos dados sobre la matriz original . . . . .	42
2.2 Inversión de una matriz por eliminación Gaussiana . . . . .	49
<b>3 Otros trozos de código</b>	<b>53</b>
3.1 Métodos de representación para el entorno Jupyter . . . . .	53
3.2 Completando la clase <b>Vector</b> . . . . .	54
3.2.1 Representación de la clase <b>Vector</b> . . . . .	54
3.2.2 Otros métodos para la clase <b>Vector</b> . . . . .	55
3.3 Completando la clase <b>Matrix</b> . . . . .	55
3.3.1 Otras formas de instanciar una <b>Matrix</b> . . . . .	55
3.3.2 Códigos que verifican que los argumentos son correctos . . . . .	56
3.3.3 Representación de la clase <b>Matrix</b> . . . . .	56
3.3.4 Otros métodos para la clase <b>Matrix</b> . . . . .	57

3.4	Completando la clase <code>T</code> . . . . .	57
3.4.1	Otras formas de instanciar una <code>T</code> . . . . .	57
3.4.2	Representación de la clase <code>T</code> . . . . .	58
3.5	Vectores y Matrices especiales . . . . .	59
3.6	La clase <code>BlockMatrix</code> . Matrices particionadas . . . . .	60
3.6.1	Particionado de matrices . . . . .	61
3.6.2	Representación de la clase <code>BlockMatrix</code> . . . . .	65
<b>A</b>	<b>Sobre este documento</b>	<b>67</b>
A.1	Secciones de código . . . . .	68

## Declaración de intenciones

Uno de los objetivos que me he propuesto para el curso Matemáticas II (Álgebra Lineal) es mostrar que escribir matemáticas y usar un lenguaje de programación son prácticamente la misma cosa. Este modo de proceder debería ser un ejercicio muy didáctico ya que:

*Un PC es muy torpe y se limita a ejecutar literalmente lo que se le indica (un PC no interpreta interpolando para intentar dar sentido a lo que se le dice... eso lo hacemos las personas, pero no los ordenadores).*

*Por tanto, este ejercicio nos impone una disciplina a la que en general no estamos acostumbrados: el ordenador hará lo que queremos solo si las expresiones tienen sentido e indican correctamente lo que queremos. Si el ordenador no hace lo que queremos, será porque que hemos escrito las ordenes de manera incorrecta (lo que supone que también hemos escrito incorrectamente las expresiones matemáticas).*

Con esta idea en mente:

1. La notación de las notas de clase pretende ser operativa, en el sentido de que su uso se pueda traducir en operaciones que debe realizar el ordenador.
2. Muchas demostraciones son algorítmicas (al menos las que tienen que ver con el método de Gauss), de manera que dichas demostraciones describen literalmente la programación en Python de los correspondientes algoritmos.

### Una librería de Python específica para la asignatura

Aunque Python tiene librerías que permiten operar con vectores y matrices, *aquí escribimos nuestra propia librería*. Con ello lograremos que la notación empleada en las notas de clase y las expresiones que usemos en Python se parezcan lo más posible.

ESTE DOCUMENTO DESCRIBE TANTO EL USO DE LA LIBRERÍA COMO EL MODO EN EL QUE ESTÁ PROGRAMADA;  
PERO NO ES UN CURSO DE PYTHON.

No obstante, y pese a la nota de anterior, he escrito unos notebooks de Jupyter que ofrecen unas breves nociones de programación en Python (muy incompletas). Tenga en cuenta que hay muchos cursos y material disponible en la web para aprender Python y que mi labor es enseñar Álgebra Lineal (no Python).

Para hacer más evidente el paralelismo entre las definiciones de las **notas de la asignatura** y el código de nuestra librería, las partes del código menos didácticas se relegan al final<sup>1</sup> (véase la sección *Literate programming* en la [Página 67](#)). Destacar algunas partes del código permitirá apreciar que las definiciones de las notas de la asignatura son implementadas de manera literal en nuestra librería de Python.

#### Tutorial previo en un Jupyter notebook

Antes de seguir, repase el Notebook **“Listas y tuplas”** en la carpeta **“TutorialPython”** en <https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Y recuerde que **¡hacer matemáticas y programar son prácticamente la misma cosa!**

<sup>1</sup>aquellas que tienen que ver con la comprobación de que los inputs de las funciones son adecuados, con otras formas alternativas de instanciar clases, con la representación de objetos en Jupyter usando código L<sup>A</sup>T<sub>E</sub>X, etc.

# Capítulo 1

## Código principal de la librería. Las clases Vector, Matrix y T

### Tutorial previo en un Jupyter notebook

Antes de seguir, mírese el Notebook referente a “Clases” en la carpeta “TutorialPython” en <https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Usando lo mostrado en el Notebook anterior, definiremos una *clase* en Python para los *vectores*, otra para las *matrices*, otra para las *transformaciones elementales* y otra para las *matrices por bloques* (o matrices particionadas).

### 1.1 La clase Vector

En las notas de la asignatura se dice que

Un *vector* de  $\mathbb{R}^n$  es un “sistema” de  $n$  números reales;

y dicho sistema se muestra entre paréntesis, bien en forma de fila

$$\mathbf{v} = (v_1, \dots, v_n)$$

o bien en forma de columna

$$\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

Python no posee objetos que sean “vectores”. Necesitamos crearlos definiendo una nueva *clase*. El texto de ayuda de la clase `Vector` es auto-explicativo y será lo que Python nos muestre cuando tecleemos `help(Vector)`:

```
4 <Texto de ayuda de la clase Vector 4>≡
    """Clase Vector

    Un Vector es una secuencia finita (sistema) de números. Los Vectores se
    pueden construir con una lista o tupla de números. Cuando el argumento
    es un Vector, se crea una copia del mismo. El atributo 'rpr' indica al
    entorno Jupyter si el vector debe ser escrito como fila o como columna.

    Parámetros:
        sis (list, tuple, Vector) : Sistema de números. Debe ser una lista o
        tupla de números, o bien otro Vector.
        rpr (str) : Representación en Jupyter ('columna' por defecto).
        Indica la forma de representar el Vector en Jupyter. Si
```

```

        rpr='fila' se representa en forma de fila. En caso contrario se
        representa en forma de columna.

Atributos:
    lista (list): sistema de números almacenado.
    n      (int) : número de elementos de la lista.
    rpr    (str) : modo de representación en Jupyter.

Ejemplos:
>>> # Crear un Vector a partir de una lista (o tupla) de números
>>> Vector( [1,2,3] )    # con lista
>>> Vector( (1,2,3) )    # con tupla

Vector([1,2,3])

>>> # Crear un Vector a partir de otro Vector
>>> Vector( Vector([1,2,3]) )

Vector([1,2,3])
"""
This code is used in chunk 6b.
Uses Vector 6b.

```

### 1.1.1 Implementación de los vectores en la clase Vector

Tanto las listas (`list`) como las tuplas (`tuple`) de Python son “sistemas” (secuencias finitas de objetos). Así que usaremos las listas (o tuplas) de números para instanciar un `Vector`. El sistema de números contenido en la lista (o tupla) será guardado en el atributo `lista` del `Vector` como una lista (`list`). Veamos cómo:

**Método de inicialización** Comenzamos la clase con el método de inicio: `def __init__(self, ... )`.

- La clase `Vector` usará dos argumentos (o parámetros). Al primero lo llamaremos `sis` y podrá ser una lista o tupla de números, o bien, otro `Vector`. El segundo argumento (`rpr`) nos permitirá indicar si queremos que el entorno `Jupyter Notebook` represente el vector en forma horizontal o en vertical. Si no se indica nada, se asumirá que la representación del vector es en vertical (`rpr='columna'`).
- Añadimos un breve texto de ayuda sobre el método `__init__` que Python mostrará con: `help Vector.__init__`
- Por último se definen tres atributos para la clase `Vector`: los atributos `lista`, `rpr` y `n`.  
(El modo de generar el atributo `lista` depende de qué tipo de objeto es `sis`).

- Cuando `sis` es una `list` o `tuple`, en el atributo “`lista`” se guarda el correspondiente sistema en forma de una `list` de Python: `self.lista = list(sis)`
- Cuando `sis` es un `Vector`, sencillamente se copia su atributo `lista`: `self.lista = sis.lista.copy()`

De esta manera el atributo `self.lista` contendrá la lista ordenada de números que constituye el vector... por tanto *¡ya hemos traducido al lenguaje Python la definición de vector!*

- Cuando `sis` no es ni lista, ni tupla, ni `Vector`, se muestra un mensaje de error.
- Por conveniencia definimos un par de atributos más. El atributo `self.n` guarda el número de elementos de la lista. El atributo `self.rpr` indica si el vector ha de ser representado en el entorno `Jupyter` como fila o como columna (por defecto la representación es en forma de columna).

6a *<Iniciación de la clase Vector 6a>≡*

```
def __init__(self, sis, rpr='columna'):
    """Inicializa un Vector con una lista, tupla, u otro Vector"""

    if isinstance(sis, (list,tuple)):
        self.lista = list(sis)

    elif isinstance(sis, Vector):
        self.lista = sis.lista.copy()

    else:
        raise ValueError('¡el argumento: debe ser una lista, tupla o Vector!')

    self.rpr = rpr
    self.n = len (self.lista)
```

This code is used in chunk 6b.  
Uses Vector 6b.

La clase **Vector** junto con el listado de sus métodos aparece en el siguiente recuadro:

6b *<Definición de la clase Vector 6b>≡*

```
class Vector:
    <Texto de ayuda de la clase Vector 4>
    <Iniciación de la clase Vector 6a>
    <Operador selector por la derecha para la clase Vector 12>
    <Operador selector por la izquierda para la clase Vector 13b>
    <Suma de Vectores 18b>
    <Producto de un Vector por un escalar a su izquierda 19b>
    <Producto de un Vector por un escalar, Vector, o Matrix a su derecha 21a>
    <Definición de la igualdad entre Vectores 21b>
    <Representación de la clase Vector 54b>
    <Reverso de un Vector 55a>
```

This code is used in chunk 37.

Defines:

Vector, used in chunks 4, 6a, 8, 9, 11–13, 15–22, 24, 25a, 39, 51b, 55, 56c, and 59b.

En esta sección hemos visto el texto de ayuda y el método de inicialización. El resto de métodos se describen en secciones posteriores (detrás del nombre de cada trozo de código aparece el número de página donde encontrarlo).

## Resumen

Los **vectores** son sistemas de números. La clase **Vector** almacena una **list** de números en su atributo **Vector.lista**:

1. Cuando se instancia un **Vector** con una lista, dicha lista se almacena en el atributo **lista**.
2. Cuando se instancia un **Vector** con otro **Vector**, se copia el atributo **lista** de dicho **Vector**.

3. Asociados a los `Vectores` hay una serie de métodos que veremos más adelante.



## 1.2 La clase Matrix

En las notas de la asignatura usamos la siguiente definición

Llamamos *matriz* de  $\mathbb{R}^{m \times n}$  a un sistema de  $n$  vectores de  $\mathbb{R}^m$ .

Cuando representamos las matrices, las encerramos entre corchetes

$\mathbf{A} = [\mathbf{v}_1, \dots, \mathbf{v}_n]$ , donde las  $n$  columnas  $\mathbf{v}_i$  son vectores de  $\mathbb{R}^m$ .

En nuestra implementación crearemos un objeto, `Matrix`, que almacene en uno de sus atributos una lista de `Vectores` (todos con el mismo número de componentes). Dicha lista será la lista de “columnas” de la matriz. El texto de ayuda de nuestra clase `Matrix` es auto-explicativo y Python lo mostrará si se teclea `help(Matrix)`.

```
8 <Texto de ayuda de la clase Matrix 8>≡
    """Clase Matrix

    Una Matrix es una secuencia finita (sistema) de Vectores con el mismo
    número de componentes. Una Matrix se puede construir con una lista o
    tupla de Vectores con el mismo número de componentes (serán las columnas
    de la matriz); una lista (o tupla) de listas o tuplas con el mismo
    número de componentes (serán las filas de la matriz); una Matrix (el
    valor devuelto será una copia de la Matrix); una BlockMatrix (el valor
    devuelto es la Matrix que resulta de unir todos los bloques)

    Parámetros:
        sis (list, tuple, Matrix, BlockMatrix): Lista (o tupla) de Vectores
        con el mismo núm. de componentes (columnas de la matriz); o
        lista (o tupla) de listas o tuplas con el mismo núm. de
        componentes (filas de la matriz); u otra Matrix; o una
        BlockMatrix (matriz particionada por bloques).

    Atributos:
        lista (list): sistema de Vectores almacenado
        m      (int) : número de filas de la matriz
        n      (int) : número de columnas de la matriz

    Ejemplos:
    >>> # Crea una Matrix a partir de una lista de Vectores
    >>> a = Vector( [1,2] )
    >>> b = Vector( [1,0] )
    >>> c = Vector( [9,2] )
    >>> Matrix( [a,b,c] )

    Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])

    >>> # Crea una Matrix a partir de una lista de listas de números
    >>> A = Matrix( [ [1,1,9], [2,0,2] ] )
    >>> A

    Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])

    >>> # Crea una Matrix a partir de otra Matrix
    >>> Matrix( A )

    Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])
```

```
>>> # Crea una Matrix a partir de una BlockMatrix
>>> Matrix( {1}|A|{2} )

Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])
"""
This code is used in chunk 10.
Uses BlockMatrix 61b, Matrix 10, and Vector 6b.
```

### 1.2.1 Implementación de las matrices en la clase Matrix

**Método de inicialización** Comenzamos la clase con el método de inicio: `def __init__(self, sis)`.

- Una `Matrix` se instancia con el argumento `sis`, que podrá ser una lista de `Vectores` con el mismo número de componentes (serán sus columnas); pero que también se podrá ser una lista (o tupla) de listas o tuplas con el mismo número de componentes (serán sus filas), o una `BlockMatrix`, o bien otra `Matrix`.
  - Añadimos un breve texto de ayuda del método `__init__`
  - Por último se definen tres atributos para la clase `Matrix`. Los atributos: `lista`, `m` y `n`.  
(El modo de generar el atributo `lista` depende de qué tipo de objeto es `sis`. En el recuadro de más abajo solo se muestra el caso en que `sis` es una lista de `Vectores`).
- El atributo `self.lista` guarda una lista de `Vectores` (que corresponden a la lista de columnas de la matriz). El modo de elaborar dicha lista difiere en función de qué tipo de objeto es el argumento `sis`. Si es
    - \* `list` (o `tuple`) de `Vectores`: entonces la `self.lista` es `list(sis)` (la lista de `Vectores` introducidos).
    - \* `list` (o `tuple`) de `lists` o `tuples`: entonces se interpreta que `sis` es la “lista de filas” de una matriz y se reconstruye la lista de columnas correspondiente a dicha matriz.
    - \* `Matrix`: entonces `self.lista` es una copia de la lista de `sis` (`self.lista = sis.lista.copy()`).
    - \* `BlockMatrix`: se guarda la lista de la `Matrix` resultante de unificar los bloques en una única matriz.
 De esta manera el atributo `self.lista` contendrá la lista ordenada de `Vectores` columna que constituye la matriz... por tanto *y ya hemos traducido al lenguaje Python la definición de matriz!*
  - Por conveniencia definimos un par de atributos más. El atributo `self.m` guarda el número de filas de la matriz, y `self.n` guarda el número de columnas.

```
9 <Iniciación de la clase Matrix 9>≡
def __init__(self, sis):
    """Inicializa una Matrix"""
    <Creación del atributo lista cuando sis no es una lista (o tupla) de Vectores 55b>

    elif isinstance(sis[0], Vector):
        <Verificación de que todas las columnas de la matriz tienen la misma longitud 56c>
        self.lista = list(sis)

    self.m = self.lista[0].n
    self.n = len(self.lista)
```

This code is used in chunk 10.  
Uses Matrix 10 and Vector 6b.

La clase `Matrix` junto con el listado de sus métodos aparece en el siguiente recuadro:

```
10 <Definición de la clase Matrix 10>≡
    class Matrix:
        <Texto de ayuda de la clase Matrix 8>
        <Inicialización de la clase Matrix 9>
        <Operador selector por la derecha para la clase Matrix 14>
        <Operador transposición para la clase Matrix 15b>
        <Operador selector por la izquierda para la clase Matrix 17>
        <Suma de Matrix 22b>
        <Producto de una Matrix por un escalar a su izquierda 23b>
        <Producto de una Matrix por un escalar, un vector o una matriz a su derecha 25a>
        <Definición de la igualdad entre dos Matrix 26>
        <Transformaciones elementales de las columnas de una Matrix 35a>
        <Transformaciones elementales de las filas de una Matrix 36>
        <Potencia de una Matrix 25b>
        <Representación de la clase Matrix 56d>
        <Reverso de una Matrix 57a>

    This code is used in chunk 37.
    Defines:
        Matrix, used in chunks 8, 9, 13–17, 20–25, 28, 30c, 31, 34, 35, 40–51, 55–57, 60, 64c, and 65a.
```

En esta sección hemos visto el texto de ayuda y el método de inicialización. El resto de métodos se describen en secciones posteriores.

## Resumen

Las **matrices** son sistemas de vectores (dichos vectores son sus columnas). La clase `Matrix` almacena una `list` de `Vectores` en el atributo `lista` de cuatro modos distintos (el código de los tres últimos se puede consultar en el Capítulo 3 de este documento):

1. Cuando se instancia con una lista de `Vectores`, dicha lista se almacena en el atributo `lista`. *Esta es la forma de crear una matriz a partir de sus columnas.*
2. Por comodidad, cuando se instancia una `Matrix` con una lista (o tupla) de listas o tuplas, se interpreta que dicha lista (o tupla) son las filas de la matriz. Consecuentemente, se dan los pasos para describir dicha matriz como una lista de columnas, que se almacena en el atributo `lista`. (Esta forma de instanciar una `Matrix` se usará para programar la **transposición** en la Página 14).
3. Cuando se instancia con otra `Matrix`, se copia el atributo `lista` de dicha `Matrix`.
4. Cuando se instancia con una `BlockMatrix`, se unifican los bloques en una sola matriz, cuya lista de columnas es copiada en el atributo `lista`.
5. Asociados a las `Matrix` hay una serie de métodos que veremos más adelante.

Así pues,

- `Vector` guarda un sistema de números en su atributo `lista`
- `Matrix` guarda una sistema de `Vectores` en su atributo `lista`; por tanto:

**¡Ya hemos implementado en Python los vectores y matrices tal y como se definen en las notas de la asignatura!**

... vamos con el operador selector que nos permitirá definir las operaciones de suma, producto, etc...

### 1.3 Operadores selectores

#### Notación en Mates 2

- Si  $\mathbf{v} = (v_1, \dots, v_n)$  entonces  ${}_i|\mathbf{v} = \mathbf{v}|_i = v_i$  para todo  $i \in \{1, \dots, n\}$ .
- Si  $\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$  entonces  $\begin{cases} \mathbf{A}|_j = \begin{pmatrix} a_{1j} \\ \vdots \\ a_{nj} \end{pmatrix} \text{ para todo } j \in \{1, \dots, m\} \\ {}_i|\mathbf{A} = (a_{i1}, \dots, a_{im}) \text{ para todo } i \in \{1, \dots, n\} \end{cases}$ .

Pero puestos a seleccionar, aprovechemos la notación para seleccionar más de un elemento:

#### Notación en Mates 2

- $(i_1, \dots, i_r)|\mathbf{v} = (\mathbf{v}_{i_1}, \dots, \mathbf{v}_{i_r}) = \mathbf{v}|_{(i_1, \dots, i_r)}$  (es un vector formado por elementos de  $\mathbf{v}$ )
- $(i_1, \dots, i_r)|\mathbf{A} = \begin{bmatrix} {}_{i_1}|\mathbf{A}, \dots, {}_{i_r}|\mathbf{A} \end{bmatrix}^\top$  (es una matriz cuyas filas son filas de  $\mathbf{A}$ )
- $\mathbf{A}|_{(j_1, \dots, j_r)} = \begin{bmatrix} \mathbf{A}|_{j_1}, \dots, \mathbf{A}|_{j_r} \end{bmatrix}$  (es una matriz formada por columnas de  $\mathbf{A}$ )

Queremos manejar una notación similar en Python, así que tenemos que definir el operador selector. Y queremos hacerlo con un método de Python que tenga asociado un símbolo que permita invocar el método de selección

#### Tutorial previo en un Jupyter notebook

Si no recuerda a qué me estoy refiriendo con los símbolos asociados a métodos, repase de nuevo la sección “Métodos especiales con símbolos asociados” del Notebook referente a “Clases” en la carpeta “TutorialPython” en

<https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Como los métodos `--or--` y `--ror--` tienen asociados la barra vertical a derecha e izquierda, usaremos el siguiente convenio:

Mates II	Python
$\mathbf{v} _i$	<code>v i</code>
${}_i \mathbf{v}$	<code>i v</code>
$\mathbf{A} _j$	<code>A j</code>
${}_i \mathbf{A}$	<code>i A</code>

#### 1.3.1 Operador selector por la derecha para la clase Vector.

```
11 <Texto de ayuda para el operador selector por la derecha para la clase Vector 11>≡
    """Selector por la derecha

    Extrae la i-ésima componente del Vector, o genera un nuevo vector con
    las componentes indicadas en una lista o tupla (los índices comienzan
    por la posición 1).
```

```

Parámetros:
    i (int, list, tuple): Índice (o lista de índices) de los elementos
                        a seleccionar.

Resultado:
    número: Cuando i es int, devuelve el componente i-ésimo del Vector.
    Vector: Cuando i es list o tuple, devuelve el Vector formado por los
            componentes indicados en la lista o tupla de índices.

Ejemplos:
>>> # Selección de una componente
>>> Vector([10,20,30]) | 2

20

>>> # Creación de un sub-vector a partir de una lista o tupla de índices
>>> Vector([10,20,30]) | [2,1,2]
>>> Vector([10,20,30]) | (2,1,2)

Vector([20, 10, 20])
"""
This code is used in chunk 12.
Uses Vector 6b.

```

### Implementación del operador selector por la derecha para la clase Vector.

Cuando el argumento *i* es un número entero (*int*), seleccionamos el correspondiente elemento del atributo *lista* del *Vector* (recuerde que en Python los índices de objetos iterables comienzan en cero, por lo que para seleccionar el elemento *i*-ésimo de *lista*, escribimos *lista[i-1]*; así *a|1* debe seleccionar el primer elemento del atributo *lista*, es decir *a.lista[0]*).

Una vez hemos definido el operador “|” cuando el argumento *i* es un entero (*int*), podemos usar el método (*self|a*) para definir el operador cuando el argumento *i* es una lista o tupla (*list,tuple*) de índices (y así generar un *Vector* con la lista de componentes indicadas).

```

12  <Operador selector por la derecha para la clase Vector 12>≡
    def __or__(self,i):
        <Texto de ayuda para el operador selector por la derecha para la clase Vector 11>
        if isinstance(i,int):
            return self.lista[i-1]

        elif isinstance(i, (list,tuple) ):
            return Vector ([ (self|a) for a in i ])

This code is used in chunk 6b.
Uses Vector 6b.

```

### 1.3.2 Operador selector por la izquierda para la clase Vector.

13a `<Texto de ayuda para el operador selector por la izquierda para la clase Vector 13a>≡`  
`"""Selector por la izquierda`  
  
`Hace lo mismo que el método __or__ solo que operando por la izquierda`  
`"""`  
 This code is used in chunk 13b.

#### Implementación del operador selector por la derecha para la clase Vector.

Como hace lo mismo que el selector por la derecha, basta con llamar al selector por la derecha: `self|i`

13b `<Operador selector por la izquierda para la clase Vector 13b>≡`  
`def __ror__(self,i):`  
 `<Texto de ayuda para el operador selector por la izquierda para la clase Vector 13a>`  
 `return self | i`  
  
 This code is used in chunk 6b.

### 1.3.3 Operador selector por la derecha para la clase Matrix.

13c `<Texto de ayuda para el operador selector por la derecha para la clase Matrix 13c>≡`  
`"""`  
`Extrae la i-ésima columna de Matrix; o crea una Matrix con las columnas`  
`indicadas; o crea una BlockMatrix particionando una Matrix por las`  
`columnas indicadas (los índices comienzan por la posición 1)`  
  
`Parámetros:`  
 `j (int, list, tuple): Índice (o lista de índices) de las columnas a`  
 `seleccionar`  
 `(set): Conjunto de índices de las columnas por donde particionar`  
  
`Resultado:`  
 `Vector: Cuando j es int, devuelve la columna j-ésima de Matrix.`  
 `Matrix: Cuando j es list o tuple, devuelve la Matrix formada por las`  
 `columnas indicadas en la lista o tupla de índices.`  
 `BlockMatrix: Si j es un set, devuelve la BlockMatrix resultante de`  
 `particionar la matriz por las columnas indicadas en el conjunto`  
  
`Ejemplos:`  
`>>> # Extrae la j-ésima columna la matriz`  
`>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | 2`

```

Vector([0,2])

>>> # Matrix formada por Vectores columna indicados en la lista (o tupla)
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | [2,1]
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | (2,1)

Matrix( [Vector([0,2]), Vector([1,0])] )

>>> # BlockMatrix correspondiente a la partición por la segunda columna
>>> Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | {2}

BlockMatrix( [ [ Matrix([Vector([1, 0]), Vector([0, 2])]),
                  Matrix([Vector([3, 0])]) ] ] )

"""
This code is used in chunk 14.
Uses BlockMatrix 61b, Matrix 10, and Vector 6b.

```

### Implementación del operador selector por la derecha para la clase Matrix

Como el objeto `Matrix` es una lista de `Vectores`, el código para el selector por la derecha es casi idéntico al de la clase `Vector`. Como antes, una vez definido el operador “|” por la derecha que selecciona una única columna, usaremos repetidamente dicho procedimiento (`self|a`) para crear una `Matrix` formada por las columnas indicadas en una lista (o tupla) de índices.

(la partición en bloques de columnas de matrices se verá más adelante, en la sección de la clase `BlockMatrix`).

```

14  <Operador selector por la derecha para la clase Matrix 14>≡
    def __or__(self,j):
        <Texto de ayuda para el operador selector por la derecha para la clase Matrix 13c>
        if isinstance(j,int):
            return self.lista[j-1]

        elif isinstance(j, (list,tuple)):
            return Matrix ([ self|a for a in j ])

        <Partición de una matriz por columnas de bloques 63a>

This code is used in chunk 10.
Uses Matrix 10.

```

### 1.3.4 Operador transposición de una Matrix.

Implementar el operador selector por la izquierda es algo más complicado que en el caso de los `Vectores`, pues ahora no es lo mismo operar por la derecha que por la izquierda. Como paso intermedio definiremos el operador transposición, que después usaremos para definir el operador selector por la izquierda (selección de filas).

#### Notación en Mates 2

Denotamos la *transpuesta* de  $\mathbf{A}$  con:  $\mathbf{A}^\top$ ; y es la matriz tal que  $(\mathbf{A}^\top)_{|j} = {}_j\mathbf{A}; \quad j = 1 : n.$

15a *<Texto de ayuda para el operador transposición de la clase Matrix 15a>≡*

```

"""
Devuelve la traspuesta de una matriz

Ejemplo:
>>> ~Matrix([Vector([1]), Vector([2]), Vector([3])])

Matrix([Vector([1, 2, 3])])
"""
This code is used in chunk 15b.
Uses Matrix 10 and Vector 6b.

```

### Implementación del operador transposición.

Desgraciadamente Python no dispone del símbolo “ $\tau$ ”. Así que hemos de usar un símbolo distinto para indicar transposición. Y además no tenemos muchas opciones ya que el conjunto de símbolos asociados a métodos especiales es muy limitado.

#### Tutorial previo en un Jupyter notebook

Si no recuerda a qué me estoy refiriendo con los símbolos asociados a métodos, repase de nuevo la sección “Métodos especiales con símbolos asociados” del Notebook referente a “Clases” en la carpeta “TutorialPython” en

<https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Para implementar la transposición haremos uso del método `__invert__`, que tiene asociado el símbolo del la tilde “ $\sim$ ”, símbolo que además deberemos colocar a la izquierda de la matriz.

Mates II	Python
$\mathbf{A}^\tau$	$\sim \mathbf{A}$

Recuerde que con la segunda forma de instanciar una `Matrix` (véase el resumen de la página 10), creamos una matriz a partir de la lista de sus filas. Así podemos construir fácilmente el operador trasposición. Basta instanciar `Matrix` con la lista de los  $n$  atributos “`lista`” correspondientes a los consecutivos  $n$  `Vectores` columna.

(Recuerde que `range(1,self.m+1)` recorre los números:  $1, 2, \dots, m$ ).

15b *<Operador transposición para la clase Matrix 15b>≡*

```

def __invert__(self):
    <Texto de ayuda para el operador transposición de la clase Matrix 15a>
    return Matrix([ (self[j]).lista for j in range(1,self.n+1) ])

This code is used in chunk 10.
Uses Matrix 10.

```



### 1.3.5 Operador selector por la izquierda para la clase Matrix.

```

16 <Texto de ayuda para el operador selector por la izquierda para la clase Matrix 16>≡
    """Operador selector por la izquierda

    Extrae la i-ésima fila de Matrix; o crea una Matrix con las filas
    indicadas; o crea una BlockMatrix particionando una Matrix por las filas
    indicadas (los índices comienzan por la posición 1)

    Parámetros:
        i (int, list, tuple): Índice (o lista de índices) de las filas a
            seleccionar
        (set): Conjunto de índices de las filas por donde particionar

    Resultado:
        Vector: Cuando i es int, devuelve la fila i-ésima de Matrix.
        Matrix: Cuando i es list o tuple, devuelve la Matrix cuyas filas son
            las indicadas en la lista de índices.
        BlockMatrix: Cuando i es un set, devuelve la BlockMatrix resultante
            de particionar la matriz por las filas indicadas en el conjunto

    Ejemplos:
    >>> # Extrae la j-ésima fila de la matriz
    >>> 2 | Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])])

    Vector([0, 2, 0])

    >>> # Matrix formada por Vectores fila indicados en la lista (o tupla)
    >>> [1,1] | Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])])
    >>> (1,1) | Matrix([Vector([1,0]), Vector([0,2]), Vector([3,0])])

    Matrix([Vector([1, 1]), Vector([0, 0]), Vector([3, 3])])

    >>> # BlockMatrix correspondiente a la partición por la primera fila
    >>> {1} | Matrix([Vector([1,0]), Vector([0,2])])

    BlockMatrix( [ [Matrix([Vector([1]),Vector([0])]),
                    [Matrix([Vector([0]),Vector([2])])]] ] )

    """
    This code is used in chunk 17.
    Uses BlockMatrix 61b, Matrix 10, and Vector 6b.

```

#### Implementación del operador por la izquierda para la clase Matrix.

Usando el operador selector de columnas y la transposición, es inmediato definir un operador selector de *filas*... ¡que son las columnas de la matriz transpuesta!

```
(~self)|j
```

(para recordar que se ha obtenido una fila de la matriz, representamos el `Vector` en horizontal: `rpr='fila'`)

Una vez definido el operador por la izquierda, podemos usarlo repetidas veces el procedimiento `(a|self)` para crear una `Matrix` con las filas indicadas en una lista o tupla de índices.

(la partición en bloques de filas de matrices se verá más adelante, en la sección de la clase `BlockMatrix`).

```

17  <Operador selector por la izquierda para la clase Matrix 17>≡
    def __ror__(self,i):
        <Texto de ayuda para el operador selector por la izquierda para la clase Matrix 16>
        if isinstance(i,int):
            return Vector ( (~self)|i, rpr='fila' )

        elif isinstance(i, (list,tuple)):
            return Matrix ( [ (a|self).lista for a in i ] )

        <Partición de una matriz por filas de bloques 62b>

    This code is used in chunk 10.
    Uses Matrix 10 and Vector 6b.

```

## Resumen

¡Ahora también hemos implementado en Python el operador “|” (tanto por la derecha como por la izquierda tal) y como se define en las notas de la asignatura!

Ya estamos listos para definir el resto de operaciones con vectores y matrices...

## 1.4 Operaciones con vectores y matrices

Una vez definidas las clases `Vector` y `Matrix` junto con los respectivos operadores selectores “|”, ya podemos definir las operaciones de suma y producto. Fíjese que las definiciones de las operaciones en Python (usando el operador “|”) son idénticas a las empleadas en las notas de la asignatura:

### 1.4.1 Suma de Vectores

En las notas de la asignatura hemos definido la suma de dos vectores de  $\mathbb{R}^n$  como el vector tal que

$$(a + b)_{|i} = a_{|i} + b_{|i} \quad \text{para } i = 1 : n.$$

Usando el operador selector podemos “literalmente” transcribir esta definición

```
Vector ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
```

donde `self` es el vector, `other` es otro vector y `range(1,self.n+1)` es el rango de valores:  $1 : n$ .

18a *<Texto de ayuda para el operador suma en la clase Vector 18a>*≡

```
"""Devuelve el Vector resultante de sumar dos Vectores

Parámetros:
    other (Vector): Otro vector con el mismo número de elementos

Ejemplo
>>> Vector([10, 20, 30]) + Vector([-1, 1, 1])

Vector([9, 21, 31])
"""
```

This code is used in chunk 18b.  
Uses Vector 6b.

### Implementación

18b *<Suma de Vectores 18b>*≡

```
def __add__(self, other):
    <Texto de ayuda para el operador suma en la clase Vector 18a>
    if isinstance(other, Vector):
        if self.n != other.n:
            raise ValueError('Vectores con distinto número de componentes')
        return Vector ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
```

This code is used in chunk 6b.  
Uses Vector 6b.

### 1.4.2 Producto de un Vector por un escalar a su izquierda

En las notas hemos definido el producto de  $\mathbf{a}$  por un escalar  $x$  a su izquierda como el vector tal que

$$(x\mathbf{a})_{|i} = x(\mathbf{a}_{|i}) \quad \text{para } i = 1 : n.$$

cuya transcripción será

```
Vector ([ x*(self[i] for i in range(1,self.n+1) ])
```

donde **self** es el vector y **x** es un número entero, de coma flotante o una fracción (**int**, **float**, **Fraction**).

19a *<Texto de ayuda para el operador producto por la izquierda en la clase Vector 19a>*≡

```
"""Multiplica un Vector por un número a su izquierda

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica

Resultado:
    Vector: Cuando x es int, float o Fraction, devuelve el Vector que
           resulta de multiplicar cada componente por x

Ejemplo:
>>> 3 * Vector([10, 20, 30])

Vector([30, 60, 90])
"""
```

This code is used in chunk 19b.  
Uses Vector 6b.

## Implementación

19b *<Producto de un Vector por un escalar a su izquierda 19b>*≡

```
def __rmul__(self, x):
    <Texto de ayuda para el operador producto por la izquierda en la clase Vector 19a>
    if isinstance(x, (int, float, Fraction)):
        return Vector ([ x*(self[i] for i in range(1,self.n+1) ])
```

This code is used in chunk 6b.  
Uses Vector 6b.

### 1.4.3 Producto de un Vector por un escalar, un Vector o una Matrix a su derecha

- En las notas se acepta que el producto de un vector **a** por un escalar es conmutativo. Por tanto,

$$\mathbf{a}x = x\mathbf{a}$$

cuya transcripción será

```
x * self
```

donde **self** es el vector y **x** es un número entero, o de coma flotante o una fracción (**int**, **float**, **Fraction**).

- El *producto punto* (o producto escalar usual en  $\mathbb{R}^n$ ) de dos vectores  $\mathbf{a}$  y  $\mathbf{x}$  en  $\mathbb{R}^n$  es

$$\mathbf{a} \cdot \mathbf{x} = a_1x_1 + \cdots + a_nx_n = \sum_{i=1}^n a_ix_i \quad \text{para } i = 1 : n.$$

cuya transcripción será

```
sum([ (self|i)*(x|i) for i in range(1,self.n+1) ])
```

donde `self` es el vector  $\mathbf{a}$  y `x` es otro vector (`Vector`).

- El producto de un vector  $\mathbf{a}$  de  $\mathbb{R}^n$  por una matriz  $\mathbf{X}$  con  $n$  filas es

$$\mathbf{aX} = \mathbf{X}^T \mathbf{a}$$

cuya transcripción será

```
(~x) * self
```

donde `self` es el vector y `x` es una matriz (`Matrix`). Para recordar que es una combinación lineal de las filas, su representación es en forma de fila.

(la definición del producto de una `Matrix` por un `Vector` a su derecha se verá más adelante.)

20

```
<Texto de ayuda para el operador producto por la derecha en la clase Vector 20>≡
"""Multiplica un Vector por un número, Matrix o Vector a su derecha.

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica
    (Matrix): Matrix con tantas filas como componentes tiene el Vector
    (Vector): Vector con el mismo número de componentes.

Resultado:
    Vector: Cuando x es int, float o Fraction, devuelve el Vector que
            resulta de multiplicar cada componente por x
            Cuando x es Matrix, devuelve el Vector combinación lineal de
            las filas de Matrix (los componentes del Vector son los
            coeficientes de la combinación lineal)
    Número: Cuando x es Vector, devuelve el producto punto entre
            vectores (producto escalar usual en R^n)

Ejemplos:
>>> Vector([10, 20, 30]) * 3

Vector([30, 60, 90])

>>> a = Vector([1, 1])
>>> B = Matrix([Vector([1, 2]), Vector([1, 0]), Vector([9, 2])])
>>> a * B

Vector([3, 1, 11])

>>> Vector([1, 1, 1]) * Vector([10, 20, 30])

60
"""
```

This code is used in chunk 21a.

Uses Matrix 10 and Vector 6b.

## Implementación

21a *⟨Producto de un Vector por un escalar, Vector, o Matrix a su derecha 21a⟩≡*

```
def __mul__(self, x):
    ⟨Texto de ayuda para el operador producto por la derecha en la clase Vector 20⟩
    if isinstance(x, (int, float, Fraction)):
        return x*self

    elif isinstance(x, Matrix):
        if self.n != x.m:
            raise ValueError('Vector y Matrix incompatibles')
        return Vector( (~x)*self, rpr='fila' )

    elif isinstance(x, Vector):
        if self.n != x.n:
            raise ValueError('Vectores con distinto número de componentes')
        return sum([ (self|i)*(x|i) for i in range(1,self.n+1) ])

This code is used in chunk 6b.
Uses Matrix 10 and Vector 6b.
```

### 1.4.4 Igualdad entre vectores

Dos vectores son iguales cuando lo son los sistemas de números correspondientes a ambos vectores.

21b *⟨Definición de la igualdad entre Vectores 21b⟩≡*

```
def __eq__(self, other):
    """Indica si es cierto que dos vectores son iguales"""
    return self.lista == other.lista

This code is used in chunk 6b.
```

### 1.4.5 Suma de matrices

En las notas de la asignatura hemos definido la suma de matrices como la matriz tal que

$$(\mathbf{A} + \mathbf{B})_{|j} = \mathbf{A}_{|j} + \mathbf{B}_{|j} \quad \text{para } i = 1 : n.$$

de nuevo, usando el operador selector podemos transcribir literalmente esta definición

```
Matrix ([ (self|i) + (other|i) for i in range(1,self.n+1) ])
```

donde **self** es la matriz y **other** es otra matriz.

22a *<Texto de ayuda para el operador suma en la clase Matrix 22a>*≡

```

"""Devuelve la Matrix resultante de sumar dos Matrices

Parámetros:
    other (Matrix): Otra Matrix con el mismo número de filas y columnas

Ejemplo:
>>> A = Matrix( [Vector([1,0]), Vector([0,1])] )
>>> B = Matrix( [Vector([0,2]), Vector([2,0])] )
>>> A + B

Matrix( [Vector([1,2]), Vector([2,1])] )
"""

```

This code is used in chunk 22b.  
Uses Matrix 10 and Vector 6b.

## Implementación

22b *<Suma de Matrix 22b>*≡

```

def __add__(self, other):
    <Texto de ayuda para el operador suma en la clase Matrix 22a>
    if not isinstance(other, Matrix):
        raise ValueError('A una Matrix solo se le puede sumar otra Matrix')
    if (self.m, self.n) != (other.m, other.n):
        raise ValueError('Matrices con distinto orden')
    return Matrix ([ (self|i) + (other|i) for i in range(1, self.n+1) ])

```

This code is used in chunk 10.  
Uses Matrix 10.

### 1.4.6 Producto de una Matrix por un escalar a su izquierda

En las notas hemos definido

- El producto de  $\mathbf{A}$  por un escalar  $x$  a su izquierda como la matriz tal que

$$\boxed{(x\mathbf{A})_{|j} = x(\mathbf{A}_{|j})} \quad \text{para } i = 1 : n.$$

cuya transcripción será:

```
Matrix ([ x*(self|i) for i in range(1, self.n+1) ])
```

donde `self` es la matriz y `x` es un número entero, de coma flotante o una fracción (`int`, `float`, `Fraction`).

23a *<Texto de ayuda para el operador producto por la izquierda en la clase Matrix 23a>*≡

```

"""Multiplica una Matrix por un número a su izquierda.

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica

Resultado:
    Matrix: Devuelve el múltiplo de la Matrix

Ejemplo:
>>> 10 * Matrix([[1,2],[3,4]])

Matrix([[10,20],[30,40]])
"""
This code is used in chunk 23b.
Uses Matrix 10.

```

### 1.4.7 Implementación

23b *<Producto de una Matrix por un escalar a su izquierda 23b>*≡

```

def __rmul__(self,x):
    <Texto de ayuda para el operador producto por la izquierda en la clase Matrix 23a>
    if isinstance(x, (int, float, Fraction)):
        return Matrix ([ x*(self[i] for i in range(1,self.n+1) ])

This code is used in chunk 10.
Uses Matrix 10.

```

### 1.4.8 Producto de una Matrix por un escalar, un Vector o una Matrix a su derecha

- En las notas se acepta que el producto de una `Matrix` por un escalar es conmutativo. Por tanto,

$$\mathbf{A}x = x\mathbf{A}$$

cuya transcripción será

```
x * self
```

donde `self` es la matriz y `x` es un número entero, o de coma flotante o una fracción (`int`, `float`, `Fraction`).

- El producto de  $\mathbf{A}_{m \times n}$  por un vector  $\mathbf{x}$  de  $\mathbb{R}^n$  a su derecha se define como

$$\mathbf{A}\mathbf{x} = x_1\mathbf{A}_{|1} + \cdots + x_n\mathbf{A}_{|n} = \sum_{j=1}^n x_j\mathbf{A}_{|j} \quad \text{para } j = 1 : n.$$

cuya transcripción será



```
sum([ (x|j)*(self|j) for j in range(1,self.n+1) ])
```

donde `self` es el vector y `x` es otro vector (`Vector`).

- El producto de  $\mathbf{A}$  por otra matriz  $\mathbf{X}$  de  $\mathbb{R}^n$  a su derecha se define como la matriz tal que

$$(\mathbf{AX})_{|j} = \mathbf{A}(\mathbf{X}_{|j}) \quad \text{para } j = 1 : n.$$

cuya transcripción será

```
Matrix( [ self*(x|j) for j in range(1,x.n+1)] )
```

donde `self` es la matriz y `x` es otra matriz (`Matrix`).

24

```
<Texto de ayuda para el operador producto por la derecha en la clase Matrix 24>≡
"""Multiplica una Matrix por un número, Vector o una Matrix a su derecha

Parámetros:
    x (int, float o Fraction): Número por el que se multiplica
    (Vector): Vector con tantos componentes como columnas tiene Matrix
    (Matrix): con tantas filas como columnas tiene la Matrix

Resultado:
    Matrix: Si x es int, float o Fraction, devuelve la Matrix que
            resulta de multiplicar cada columna por x
    Vector: Si x es Vector, devuelve el Vector combinación lineal de las
            columnas de Matrix (los componentes del Vector son los
            coeficientes de la combinación)
    Matrix: Si x es Matrix, devuelve el producto entre las matrices

Ejemplos:
>>> # Producto por un número
>>> Matrix([[1,2],[3,4]]) * 10

Matrix([[10,20],[30,40]])

>>> # Producto por un Vector
>>> Matrix([Vector([1, 3]), Vector([2, 4])]) * Vector([1, 1])

Vector([3, 7])

>>> # Producto por otra Matrix
>>> Matrix([Vector([1, 3]), Vector([2, 4])]) * Matrix([Vector([1,1])])

Matrix([Vector([3, 7])])
"""
```

This code is used in chunk 25a.  
Uses Matrix 10 and Vector 6b.

### 1.4.9 Implementación

25a *⟨Producto de una Matrix por un escalar, un vector o una matriz a su derecha 25a⟩≡*

```
def __mul__(self,x):
    ⟨Texto de ayuda para el operador producto por la derecha en la clase Matrix 24⟩
    if isinstance(x, (int, float, Fraction)):
        return x*self

    elif isinstance(x, Vector):
        if self.n != x.n:
            raise ValueError('Vector y Matrix incompatibles')
        return sum([(x|j)*(self|j) for j in range(1,self.n+1)], V0(self.m))

    elif isinstance(x, Matrix):
        if self.n != x.m:
            raise ValueError('matrices incompatibles')
        return Matrix( [ self*(x|j) for j in range(1,x.n+1)] )
```

This code is used in chunk 10.  
Uses Matrix 10, V0 59b, and Vector 6b.

25b *⟨Potencia de una Matrix 25b⟩≡*

```
def __pow__(self,n):
    """Calcula potencias de una Matrix (incluida la inversa)"""
    ⟨Método auxiliar que calcula la inversa de una Matrix 49a⟩

    if self.m != self.n:
        raise ValueError('Matrix no cuadrada')
    if not isinstance(n,int):
        raise ValueError('La potencia no es un entero')

    M = self

    for i in range(1,abs(n)):
        M = M * self

    if n < 0:
        M = MatrixInversa(M)

    return M
```

This code is used in chunk 10.  
Uses Matrix 10.

### Igualdad entre matrices

*Dos matrices son iguales solo cuando lo son las listas correspondientes a ambas.*

26 *⟨Definición de la igualdad entre dos Matrix 26⟩≡*

```
def __eq__(self, other):  
    """Indica si es cierto que dos matrices son iguales"""  
    return self.lista == other.lista
```

This code is used in chunk 10.

## 1.5 La clase transformación elemental T

### Notación en Mates 2

Si  $\mathbf{A}$  es una matriz, consideramos las siguientes transformaciones:

**Tipo I:**  $\mathbf{A}_{[(\lambda)i+j]}$  suma  $\lambda$  veces la fila  $i$  a la fila  $j$  ( $i \neq j$ );  $\mathbf{A}_{[(\lambda)i+j]}^T$  lo mismo con las columnas.

**Tipo II:**  $\mathbf{A}_{[(\lambda)i]}$  multiplica la fila  $i$  por  $\lambda \neq 0$ ; y  $\mathbf{A}_{[(\lambda)j]}^T$  multiplica la columna  $j$  por  $\lambda$ .

**Intercambio:**  $\mathbf{A}_{[i \rightleftharpoons j]}$  intercambia las filas  $i$  y  $j$ ; y  $\mathbf{A}_{[i \rightleftharpoons j]}^T$  intercambia las columnas.

**Comentario sobre la notación.** Como una transformación elemental es el resultado del producto con una matriz elemental, esta notación busca el parecido con la notación del producto matricial:

Al poner la *abreviatura* “ $\tau$ ” de la transformación elemental a derecha es como si multiplicáramos la matriz  $\mathbf{A}_{m \times n}$  por la derecha por la correspondiente matriz elemental

$$\mathbf{A}_{\tau_1} = \mathbf{A} \mathbf{I}_{\tau_1} = \mathbf{A} \mathbf{E}_1 \quad \text{donde} \quad \mathbf{E}_1 = \mathbf{I}_{\tau_1} \quad \text{y donde la matriz } \mathbf{I} \text{ es de orden } n.$$

De manera similar, al poner la *abreviatura* “ $\tau$ ” de la transformación elemental a izquierda, es como si multiplicáramos la matriz  $\mathbf{A}_{m \times n}$  por la izquierda por la correspondiente matriz elemental

$$\tau_2 \mathbf{A} = \tau_2 \mathbf{I} \mathbf{A} = \mathbf{E}_2 \mathbf{A} \quad \text{donde} \quad \mathbf{E}_2 = \tau_2 \mathbf{I} \quad \text{y donde la matriz } \mathbf{I} \text{ es de orden } m.$$

Con ello se gana, entre otras cosas, que la notación sea asociativa. Pero entonces se plantea ¿qué ventaja tiene introducir en el discurso las transformaciones elementales en lugar de utilizar simplemente matrices elementales?

1. Una matriz cuadrada es un objeto muy pesado...  $n^2$  coeficientes para una matriz de orden  $n$ . Afortunadamente una matriz elemental es casi una matriz identidad salvo por uno de sus elementos; por tanto, para describir completamente una matriz elemental basta indicar su orden  $n$  y el componente que no coincide con los de la matriz  $\mathbf{I}$  de orden  $n$ .<sup>1</sup>
2. La ventaja es que las transformaciones elementales omiten el orden  $n$ .

Vamos a definir la siguiente traducción a Python de esta notación:

Mates II	Python	Mates II	Python
$\mathbf{A}_{[i \rightleftharpoons j]}^T$	<code>A &amp; T( {i,j} )</code>	$\mathbf{A}_{[i \rightleftharpoons j]}^T$	<code>T( {i,j} ) &amp; A</code>
$\mathbf{A}_{[(a)j]}^T$	<code>A &amp; T( (a,j) )</code>	$\mathbf{A}_{[(a)i]}^T$	<code>T( (a,i) ) &amp; A</code>
$\mathbf{A}_{[(a)i+j]}^T$	<code>A &amp; T( (a,i,j) )</code>	$\mathbf{A}_{[(a)i+j]}^T$	<code>T( (a,i,j) ) &amp; A</code>

Vemos que:

1. Representar el intercambio con un conjunto, permite admitir la repetición del índice  $\{i, i\} = \{i\}$  como un caso especial en el que la matriz no cambia. Esto simplificará el método de Gauss.
2. Tanto para los pares  $(a, i)$  como las ternas  $(a, i, j)$ 
  - (a) La columna (fila) que cambia es la del índice que aparece en última posición.
  - (b) El escalar aparece en la primera posición y multiplica a la columna (fila) del siguiente índice.

<sup>1</sup>Fíjese que la notación usada en las notas de la asignatura para las matrices elementales  $\mathbf{E}$ , no las describe completamente (se deja al lector la deducción de cuál es el orden adecuado para poder realizar el producto  $\mathbf{A}\mathbf{E}$  o  $\mathbf{E}\mathbf{A}$ )

Empleando listas de abreviaturas extendemos la notación para expresar secuencias de transformaciones elementales, es decir,  $\tau_1 \cdots \tau_k$ . Así logramos la siguiente equivalencia entre expresiones

$$T(t_1) \& T(t_2) \& \cdots \& T(t_k) = T([t_1, t_2, \dots, t_k])$$

De esta manera

$$\mathbf{A}_{\tau_1 \cdots \tau_k} : \quad \mathbf{A} \& T(t_1) \& T(t_2) \& \cdots \& T(t_k) = \mathbf{A} \& T([t_1, t_2, \dots, t_k])$$

$$\tau_1 \cdots \tau_k \mathbf{A} : \quad T(t_1) \& T(t_2) \& \cdots \& T(t_k) \& \mathbf{A} = T([t_1, t_2, \dots, t_k]) \& \mathbf{A}.$$

Si  $\mathbf{A}$  es de orden  $m \times n$ , el primer caso es equivalente a escribir el producto de matrices

$$\mathbf{A}_{\tau_1 \cdots \tau_k} = \mathbf{A} \mathbf{E}_1 \mathbf{E}_2 \cdots \mathbf{E}_k \quad \text{donde } \mathbf{E}_j = \mathbf{I}_{\tau_j} \text{ y donde } \mathbf{I} \text{ es de orden } n.$$

Y el segundo caso es equivalente a escribir el producto de matrices

$$\tau_1 \cdots \tau_k \mathbf{A} = \mathbf{E}_1 \mathbf{E}_2 \cdots \mathbf{E}_k \mathbf{A} \quad \text{donde } \mathbf{E}_i = \tau_i \mathbf{I} \text{ y donde } \mathbf{I} \text{ es de orden } m.$$

28 *<Texto de ayuda de la clase T (Transformación Elemental) 28>≡*

"""Clase T

T ("Transformación elemental") guarda en su atributo 't' una abreviatura (o una secuencia de abreviaturas) de transformaciones elementales. Con el método `__and__` actúa sobre otra T para crear una T que es composición de transformaciones elementales (la lista de abreviaturas), o bien actúa sobre una *Matrix* (para transformar sus filas)

Atributos:

t (set) : {índice, índice}. Abrev. de un intercambio entre los vectores correspondientes a dichos índices  
 (tuple): (índice, número). Abrev. transf. Tipo II que multiplica el vector correspondiente al índice por el número  
 : (índice1, índice2, número). Abrev. transformación Tipo I que suma al vector correspondiente al índice1 el vector correspondiente al índice2 multiplicado por el número  
 (list) : Lista de conjuntos y tuplas. Secuencia de abrev. de transformaciones como las anteriores.  
 (T) : Transformación elemental. Genera una T cuyo atributo t es una copia del atributo t de la transformación dada  
 (list) : Lista de transformaciones elementales. Genera una T cuyo atributo es la concatenación de todas las abreviaturas

Ejemplos:

```
>>> # Intercambio entre vectores
```

```
>>> T( {1,2} )
```

```
>>> # Transformación Tipo II (multiplica por 5 el segundo vector)
```

```
>>> T( (5,2) )
```

```
>>> # Transformación Tipo I (resta el tercer vector al primero)
```

```
>>> T( (-1,3,1) )
```

```
>>> # Secuencia de las tres transformaciones anteriores
```

```
>>> T( [{1,2}, (5,2), (-1,3,1)] )
```

```
>>> # T de una T
```

```
>>> T( T( (5,5) ) )
```

```

T( (5,2) )

>>> # T de una lista de T's
>>> T( [T([(-8, 2), (2, 1, 2)]), T([(-8, 3), (3, 1, 3)]) ] )

T( [(-8, 2), (2, 1, 2), (-8, 3), (3, 1, 3)] )
"""
This code is used in chunk 33b.
Uses Matrix 10 and T 33b.

```

### 1.5.1 Implementación

Python ejecuta las órdenes de izquierda a derecha. Fijándonos en la expresión

$$\mathbf{A} \ \& \ T(t_1) \ \& \ T(t_2) \ \& \cdots \ \& \ T(t_k)$$

podríamos pensar que podemos implementar la transformación elemental como un método de la clase `Matrix`. Así, al definir el método `__and__` por la derecha de la matriz podemos indicar que  $\mathbf{A} \ \& \ T(t_1)$  es una nueva matriz con las columnas modificadas. Python no tiene problema en ejecutar  $\mathbf{A} \ \& \ T(t_1) \ \& \ T(t_2) \ \& \cdots \ \& \ T(t_k)$  pues ejecutar de izquierda a derecha, es lo mismo que ejecutar  $\left[ \left[ \left[ \mathbf{A} \ \& \ T(t_1) \right] \ \& \ T(t_2) \right] \ \& \cdots \right] \ \& \ T(t_k)$  donde la expresión dentro de cada corchete es una `Matrix`, por lo que las operaciones están definidas. La dificultad aparece con

$$T(t_1) \ \& \ T(t_2) \ \& \cdots \ \& \ T(t_k) \ \& \ \mathbf{A}$$

Lo primero que Python tratara de ejecutar es  $T(t_1) \ \& \ T(t_2)$ , pero ni  $T(t_1)$  ni  $T(t_2)$  son matrices, por lo que esto no puede ser programado como un método de la clase `Matrix`.

Por tanto, necesitamos definir una nueva clase que almacene las *abreviaturas* “ $t_i$ ” de las operaciones elementales, de manera que podamos definir  $T(t_i) \ \& \ T(t_j)$ , como un método que “compone” dos transformaciones elementales para formar una secuencia de abreviaturas (de operaciones a ejecutar sobre una `Matrix`).

Así pues, definimos un nuevo tipo de objeto: `T` (“transformación elemental”) que nos permitirá encadenar transformaciones elementales (es decir, almacenar una lista de abreviaturas). El siguiente código inicializa la clase. El atributo `t` almacenará la abreviatura (o lista de abreviaturas) dada al instanciar `T` o bien creará la lista de abreviaturas a partir de otra `T` (o de una lista de `Ts`) empleada para instanciar.

```

29  <Iniciación de la clase T (Transformación Elemental) 29>≡
    def __init__(self, t):
        """Inicializa una transformación elemental"""
        <Método auxiliar CreaLista que devuelve listas de abreviaturas 30b>
        <Creación del atributo t cuando se instancia con otra T o lista de Ts 57b>
        else:
            self.t = t
            <Verificación de que las abreviaturas corresponden a transformaciones elementales 30a>

This code is used in chunk 33b.

```

30a *<Verificación de que las abreviaturas corresponden a transformaciones elementales 30a>≡*

```

for j in CreaLista(self.t):
    if isinstance(j,tuple) and (len(j) == 2) and j[0]==0:
        raise ValueError('T( (0, i) ) no es una transformación elemental')
    if isinstance(j,tuple) and (len(j) == 3) and (j[1] == j[2]):
        raise ValueError('T( (a, i, i) ) no es una transformación elemental')
    if isinstance(j,set) and (len(j) > 2) or not j:
        raise ValueError \
            ('El conjunto debe tener uno o dos índices para ser transformación elemental')

```

This code is used in chunk 29.  
 Uses CreaLista 30b.

Con algunas operaciones, como la composición de transformaciones elementales requeriremos operar con listas de abreviaturas. El siguiente procedimiento *crea la lista* [t] que contiene a t (cuando t no es una lista), si t es una lista, el procedimiento no hace nada. Se usa al instanciar T con una lista de Ts y también al componer transformaciones elementales.

30b *<Método auxiliar CreaLista que devuelve listas de abreviaturas 30b>≡*

```

def CreaLista(t):
    """Devuelve t si t es una lista; si no devuelve la lista [t]"""
    return t if isinstance(t, list) else [t]

```

This code is used in chunks 29, 31, and 33a.  
 Defines:  
 CreaLista, used in chunks 30a, 31, 33a, and 57b.

**Composición de Transf. element. o llamada al método de transformación de filas de una Matrix**

30c *<Texto de ayuda para la composición de Transformaciones Elementales T 30c>≡*

```

"""Composición de transformaciones elementales (o transformación filas)

Crea una T con una lista de abreviaturas de transformaciones elementales
(o llama al método que modifica las filas de una Matrix)

Parámetros:
    (T): Crea la abreviatura de la composición de transformaciones, es
        decir, una lista de abreviaturas
    (Matrix): Llama al método de la clase Matrix que modifica las filas
        de Matrix

Ejemplos:
>>> # Composición de dos Transformaciones elementales
>>> T( {1, 2} ) & T( (2, 4) )

```

```

T( [{1,2}, (2,4)] )

>>> # Composición de dos Transformaciones elementales
>>> T( {1, 2} ) & T( [(2, 4), (2, 1), {3, 1}] )

T( [{1, 2}, (2, 4), (2, 1), {3, 1}] )

>>> # Transformación de las filas de una Matrix
>>> T( [{1,2}, (4,2)] ) & A # multiplica por 4 la segunda fila de A y
                             # luego intercambia las dos primeras filas

"""
This code is used in chunk 31.
Uses Matrix 10 and T 33b.

```

Describimos la composición de transformaciones  $T(t_1) \& T(t_2)$  creando una lista de abreviaturas  $[t_1, t_2]$  (mediante la concatenación de listas)<sup>2</sup>. Si el atributo del método `__and__` de la clase `T` es una `Matrix`, llama al método `__rand__` de la clase `Matrix` (que transforma las filas de la matriz y que veremos un poco más abajo).

```

31  <Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix 31>≡
    def __and__(self, other):
        <Texto de ayuda para la composición de Transformaciones Elementales T 30c>
        <Método auxiliar CreaLista que devuelve listas de abreviaturas 30b>
        if isinstance(other, T):
            return T(CreaLista(self.t) + CreaLista(other.t))

        if isinstance(other, Matrix):
            return other.__rand__(self)

    This code is used in chunk 33b.
    Uses CreaLista 30b, Matrix 10, and T 33b.

```

### 1.5.2 Transposición de transformaciones elementales

Puesto que  $\mathbf{I}_{\tau_1 \dots \tau_k} = (\mathbf{E}_1 \dots \mathbf{E}_k)$  y puesto que el producto de matrices es asociativo, deducimos que la transpuesta de  $\mathbf{I}_{\tau_1 \tau_2 \dots \tau_k}$  es

$$(\mathbf{I}_{\tau_1 \tau_2 \dots \tau_k})^T = (\mathbf{I} \mathbf{E}_1 \dots \mathbf{E}_k)^T = \mathbf{E}_k^T \dots \mathbf{E}_1^T \mathbf{I} = \mathbf{I}_{\tau_k \dots \tau_2 \tau_1}$$

Nótese cómo al transponer no solo cambiamos de lado los subíndices, sino también invertimos el orden de la secuencia de transformaciones (de la misma manera que también cambia el orden en el que se multiplican las matrices elementales). Esto sugiere denotar a la operación de invertir el orden de las transformaciones como una transposición:

$$(\tau_1 \dots \tau_k)^T = \tau_k \dots \tau_1;$$

así

$$(\mathbf{A}_{\tau_1 \dots \tau_k})^T = (\tau_1 \dots \tau_k)^T \mathbf{A}^T = \tau_k \dots \tau_1 \mathbf{A}^T$$

El siguiente procedimiento invierte el orden de la lista cuando `t` es una lista de abreviaturas. Cuando `t` es una única abreviatura, no hace nada.

<sup>2</sup>Recuerde que la suma de listas (`list + list`) concatena las listas



32a *<Operador transposición para la clase T 32a>*≡

```
def __invert__(self):
    """Transpone la lista de abreviaturas (invierte su orden)"""
    return T( list(reversed(self.t)) ) if isinstance(self.t, list) else self
```

This code is used in chunk 33b.  
Uses T 33b.

### 1.5.3 Inversión de transformaciones elementales

Cualquier matriz de la forma  $\mathbf{I}_{\tau_1 \dots \tau_k}$  o de la forma  $\tau_k \dots \tau_1 \mathbf{I}$  es invertible por ser producto de matrices elementales, en particular,

$$\left( \mathbf{I}_{\tau_1 \dots \tau_k} \right) \left( \mathbf{I}_{\tau_k^{-1} \dots \tau_1^{-1}} \right) = \mathbf{E}_1 \dots \mathbf{E}_k \cdot \mathbf{E}_k^{-1} \dots \mathbf{E}_1^{-1} = \mathbf{I}_{\tau_1 \dots \tau_k \cdot \tau_k^{-1} \dots \tau_1^{-1}} = \mathbf{I};$$

por lo que podemos denotar por  $\left( \tau_1 \dots \tau_k \right)^{-1}$  a la sucesión de transformaciones  $\tau_k^{-1} \dots \tau_1^{-1}$ . De este modo

$$\left( \mathbf{I}_{\tau_1 \dots \tau_k} \right)^{-1} = \mathbf{I}_{(\tau_1 \dots \tau_k)^{-1}}.$$

El siguiente método devuelve la potencia n-esima de una transformación elemental. Si n es -1, calcula la inversa. Por ejemplo,

T([ (1,2,3), (2,3), {1,2} ]) \*\*(-1)

es

T([ {1,2}, (Fraction(1,2),3), (-1,2,3) ])

32b *<Potencia de una T 32b>*≡

```
def __pow__(self,n):
    """Calcula potencias de una T (incluida la inversa)"""
    <Método auxiliar que calcula la inversa de una Transformación elemental 33a>
    if not isinstance(n,int):
        raise ValueError('La potencia no es un entero')
    t = self

    for i in range(1,abs(n)):
        t = t & self

    if n < 0:
        t = Tinversa(t)

    return t
```

This code is used in chunk 33b.  
Uses T 33b.

33a *<Método auxiliar que calcula la inversa de una Transformación elemental 33a>≡*

```
def Tinversa ( self ):
    """Calculo de la inversa de una transformación elemental"""
    <Método auxiliar CreaLista que devuelve listas de abreviaturas 30b>

    listaT = [ ( -j[0], j[1], j[2] )    if (isinstance(j,tuple) and len(j)==3) else \
               (Fraction(1,j[0]), j[1]) if (isinstance(j,tuple) and len(j)==2) else \
               j                        for j in CreaLista(self.t) ]

    return ~T( listaT )
```

This code is used in chunk 32b.  
Uses CreaLista 30b and T 33b.

La clase T junto con el listado de sus métodos aparece en el siguiente recuadro:

33b *<Definición de la clase T (Transformación Elemental) 33b>≡*

```
class T:
    <Texto de ayuda de la clase T (Transformación Elemental) 28>
    <Inicialización de la clase T (Transformación Elemental) 29>
    <Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix 31>
    <Operador transposición para la clase T 32a>
    <Potencia de una T 32b>
    <Representación de la clase T 59a>
```

This code is used in chunk 37.  
Defines:  
T, used in chunks 28, 30–36, 40, 41a, 43–51, 57b, and 59a.

## 1.6 Transformaciones elementales de una Matrix

### 1.6.1 Transformaciones elementales de las columnas de una Matrix

34 *<Texto de ayuda de las transformaciones elementales de las columnas de una Matrix 34>≡*

```

"""Transforma las columnas de una Matrix

Atributos:
    t (T): transformaciones a aplicar sobre las columnas de Matrix

Ejemplos:
>>> A & T({1,3})           # Intercambia las columnas 1 y 3
>>> A & T((5,1))           # Multiplica la columna 1 por 5
>>> A & T((5,2,1))         # suma 5 veces la col. 2 a la col. 1
>>> A & T([1,3],(5,1),(5,2,1))# Aplica la secuencia de transformac.
                               # sobre las columnas de A y en el mismo orden de la lista

"""

This code is used in chunk 35a.
Uses Matrix 10 and T 33b.
```

**Implementación** de la aplicación de las transformaciones elementales sobre las columnas de una `Matrix` (nótese que hemos incluido el intercambio, aunque usted ya sabe que es una composición de los otros dos tipos de transf.).

35a *<Transformaciones elementales de las columnas de una Matrix 35a>≡*

```
def __and__(self,t):
    <Texto de ayuda de las transformaciones elementales de las columnas de una Matrix 34>

    if isinstance(t.t,set):
        self.lista = Matrix( [(self|max(t.t)) if k==min(t.t) else \
                               (self|min(t.t)) if k==max(t.t) else \
                               (self|k) for k in range(1,self.n+1)] ).lista.copy()

    elif isinstance(t.t,tuple) and (len(t.t) == 2):
        self.lista = Matrix([ t.t[0]*(self|k) if k==t.t[1] else \
                               (self|k) for k in range(1,self.n+1)] ).lista.copy()

    elif isinstance(t.t,tuple) and (len(t.t) == 3):
        self.lista = Matrix([ t.t[0]*(self|t.t[1]) + (self|k) if k==t.t[2] else \
                               (self|k) for k in range(1,self.n+1)] ).lista.copy()

    elif isinstance(t.t,list):
        for k in t.t:
            self & T(k)

    return self
```

This code is used in chunk 10.  
Uses Matrix 10 and T 33b.

*Observación 1.* Al actuar sobre `self.lista`, las transformaciones elementales modifican la matriz.

## 1.6.2 Transformaciones elementales de las filas de una Matrix

35b *<Texto de ayuda de las transformaciones elementales de las filas de una Matrix 35b>≡*

```
"""Transforma las filas de una Matrix

Atributos:
    t (T): transformaciones a aplicar sobre las filas de Matrix

Ejemplos:
>>> T({1,3}) & A           # Intercambia las filas 1 y 3
>>> T((5,1)) & A           # Multiplica por 5 la fila 1
>>> T((5,2,1)) & A         # Suma 5 veces la fila 2 a la fila 1
>>> T([(5,2,1),(5,1),{1,3}]) & A # Aplica la secuencia de transformac.
                               # sobre las filas de A y en el orden inverso al de la lista
"""
```

This code is used in chunk 36.  
Uses Matrix 10 and T 33b.

Para implementar las transformaciones elementales de las filas usamos el truco de aplicar las operaciones sobre las columnas de la transpuesta y de nuevo transponer el resultado: `~(~self & t)`. Pero hay que recordar que las

transformaciones más próximas a la matriz se ejecutan primero, puesto que  $\tau_1 \dots \tau_k \mathbf{A} = \mathbf{E}_1 \mathbf{E}_2 \dots \mathbf{E}_k \mathbf{A}$ . Con la función `reversed` aplicamos la sucesión de transformaciones en el orden inverso a como aparecen en la lista:

$$T([t_1, t_2, \dots, t_k]) \& \mathbf{A} = T(t_1) \& \dots \& T(t_{k-1}) \& T(t_k) \& \mathbf{A}$$

```

36 <Transformaciones elementales de las filas de una Matrix 36>≡
    def __rand__(self,t):
        <Texto de ayuda de las transformaciones elementales de las filas de una Matrix 35b>
        if isinstance(t.t,set) | isinstance(t.t,tuple):
            self.lista = (~(self & t)).lista.copy()

        elif isinstance(t.t,list):
            for k in reversed(t.t):
                T(k) & self

        return self

```

This code is used in chunk 10.  
Uses T 33b.

*Observación 2.* Al actuar sobre `self.lista`, las transformaciones elementales modifican la matriz.

## 1.7 Librería completa

Finalmente creamos la librería `notacion.py` concatenando los trozos de código que se describen en este fichero de documentación (recuerde que el número que aparece detrás de nombre de cada trozo de código indica la página donde encontrar el código en este documento).

Pero antes de todo, y para que los vectores funcionen como un espacio vectorial, es esencial importar la clase `Fraction` de la librería `fractions`<sup>3</sup>. Así pues, antes de nada, importamos la clase `Fraction` de números fraccionarios con el código:

```
from fractions import Fraction
```

```
37 <notacion.py 37>≡
    # coding=utf8

    from fractions import Fraction

    <Método html general 53a>
    <Método latex general 53b>
    <Métodos html y latex para fracciones 54a>

    <Definición de la clase Vector 6b>
    <Definición de la clase Matrix 10>
    <Definición de la clase T (Transformación Elemental) 33b>
    <Definición de la clase BlockMatrix 61b>

    <Definición del método particion 62a>
    <Definición del procedimiento de generación del conjunto clave para particionar 64b>

    <Definición de vector nulo: VO 59b>
    <Definición de matriz nula: MO 60a>
    <Definición de la matriz identidad: I 60b>

    <Método pivote calcula el índice del primer coef. no nulo de un Vector a partir de cierta posición 39>
    <Escalonamiento de una matriz mediante eliminación por columnas 40>
    <Escalonamiento de una matriz mediante eliminación por filas 41b>
    <Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>
    <Normalizando la diagonal principal para que sus componentes no nulos sean unos 48b>
    <Variantes del escalonamiento por eliminación por filas que guardan los pasos dados 47a>
    <Invirtiendo una matriz 49b>
    <Resolviendo un sistema homogéneo 51b>
    <normal 50c>
    <sistema 51a>

    Root chunk (not used in this document).
```

Consulte el Notebook sobre el **uso de nuestra librería para Mates 2** en la carpeta  
“TutorialPython” en

<sup>3</sup>el tipo de datos `float` no tiene estructura de cuerpo. Por ello vamos a emplear números fraccionarios del tipo  $\frac{a}{b}$ , donde  $a$  y  $b$  son enteros. Así pues, en el fondo estaremos trabajando con vectores de  $\mathbb{Q}^n$  (en lugar de vectores de  $\mathbb{R}^n$ ). Más adelante intentaré emplear un cuerpo de números más grande, que incluya números reales tales como  $\sqrt{2} \dots$  y también el cuerpo de fracciones de polinomios (para obtener el polinomio característico vía eliminación gaussiana, en lugar de vía determinantes).

<https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Para empaquetar esta librería en el futuro: <https://packaging.python.org/tutorials/packaging-projects/>

## Capítulo 2

# Algoritmos del curso

### 2.1 Escalonamiento de una matriz por eliminación Gaussiana

En las notas de clase llamamos *pivote* (de una columna no nula) a su primer componente no nulo; y *posición de pivote* al índice de la fila en la que está el pivote. Vamos a generalizar esta definición y decir sencillamente que llamamos *pivote* de un **Vector** (no nulo) a su primer componente no nulo; y *posición de pivote* al índice de dicho componente. Así podremos usar la definición de pivote tanto si programamos el método de eliminación por filas como por columnas.

Por conveniencia, el método `pivote` nos indicará el primer índice mayor que `k` de un componente no nulo del **Vector** (como por defecto `k=0`, si no especificamos el valor de `k`, entonces nos devuelve la posición de pivote de un **Vector**).

```
39  <Método pivote calcula el índice del primer coef. no nulo de un Vector a partir de cierta posición 39>≡
    def pivote(v, k=0):
        """
        Devuelve el primer índice(i) mayor que k de un coeficiente(c) no
        nulo del Vector v. En caso de no existir devuelve 0
        """
        return ( [i for i,c in enumerate(v.lista, 1) if (c!=0 and i>k)] + [0] )[0]

    This code is used in chunk 37.
    Defines:
        pivote, used in chunks 40–42, 44–48, and 50c.
    Uses Vector 6b.
```

En las notas de la asignatura decimos que la matriz **L** es *escalonada*, si toda columna que precede a una no nula  $L_{|k}$  no es nula y su posición de pivote es anterior a la posición de pivote de  $L_{|k}$ . Por tanto, en una matriz escalonada por columnas el primer pivote corresponde a la primera columna no nula, el segundo pivote a la segunda columna no nula, etc. El Teorema del final de las secciones de referencia de la Lección 4 demuestra que toda matriz es escalonable. Y demás nos indica el procedimiento para programar el método.

El procedimiento es iterativo. La variable `r` es un contador de pivotes encontrados en la matriz escalonada (inicialmente cero). Vamos proceder iterativamente comenzando por la primera fila. Primero buscamos la posición de pivote de la fila en que vamos a eliminar componentes.

- Si `p` es cero, quiere decir que todos los componentes son cero y hemos acabado con la eliminación para esta fila y pasamos a la siguiente.
- Si `p` es positivo, quiere decir que en dicha fila al menos hay un componente distinto de cero en una columna de índice mayor que `r`. Entonces hemos encontrado una nueva columna no nula y consecuentemente hemos encontrado un nuevo pivote. Así que debemos proceder a eliminar de izquierda a derecha hasta anular los componentes de la fila que están más a la izquierda de la posición `r`.



- El contador de pivotes aumenta en una unidad:  $r+=1$
- Como en el cuarto caso de la demostración por inducción del teorema, *intercambiamos la posición de las columnas p-ésima y r-ésima*:  $A \& T(\{p, r\})$ . Nótese que la posición de pivote  $p$  es necesariamente mayor o igual al número de pivotes encontrado  $r$  (cuando  $p=r$  este paso no hace nada, véase la definición de intercambio).
- Como en el tercer caso de la demostración por inducción del teorema, una vez reordenadas las columnas, aplicamos la sucesión de transformaciones elementales Tipo I,  $(\text{Fraction}(-(i|A|j), (i|A|r)), r, j)$ :

$$\left[ \begin{pmatrix} \tau \\ -a_{ij} \\ a_{ir} \end{pmatrix} r+j \right], \quad \text{con } j = r : n.$$

Tenga en cuenta que en el teorema sólo se indica el paso para la primera fila no nula (es decir, para  $r=1$ ).<sup>1</sup> Pero en este algoritmo se continúa fila a fila con el mismo procedimiento hasta escalonar toda la matriz.

40  $\langle \text{Escalonamiento de una matriz mediante eliminación por columnas 40} \rangle \equiv$

```

class GaussCL(Matrix):
    def __init__(self, data):
        """Escalona una Matrix con eliminación por columnas (transf. Gauss)"""
        A = Matrix(data)
        r = 0
        for i in range(1,A.m+1):
            p = pivote((i|A),r)
            if p > 0:
                r += 1
                A & T( {p, r} )
                A & T( [(Fraction(-(i|A|j), (i|A|r)), r, j) for j in range(r+1,A.n+1)] )

        super(self.__class__, self).__init__(A.lista)

This definition is continued in chunks 41a and 42a.
This code is used in chunk 37.
Uses Matrix 10, pivote 39, and T 33b.

```

Este procedimiento da lugar a operaciones con fracciones, ya que para eliminar el número  $b$  usando  $a$ , la estrategia seguida es:

$$b - \left(\frac{b}{a}\right)a = 0;$$

así, en un solo paso se elimina  $b$  restándole un múltiplo de  $a$ . Pero para escalonar una matriz con componentes enteros no es necesario trabajar con fracciones. Podemos eliminar el número  $b$  usando  $a$  encadenando dos operaciones:

$$(-a)b + ba = 0.$$

Es decir, multiplicamos  $b$  por  $a$  y luego restamos  $ba$ . Del modo análogo, podemos aplicar la sucesión de pares de transformaciones elementales Tipo II y Tipo I:

$$T([(-(i|A|r), j), ((i|A|j), r, j)]) \longrightarrow \left[ \begin{pmatrix} \tau \\ (-a_{ir})j \end{pmatrix} \begin{pmatrix} \tau \\ (a_{ij})r+j \end{pmatrix} \right], \quad \text{con } j = r : n.$$

El problema de esta solución tan simple, es que si  $a = 3$  y  $b = 3$ , en realidad basta con restar  $b - a$ , pero la solución de arriba calcularía  $(3 \times 3) - (3 \times 3)$ , por lo que los números de la matriz crecerían sin que ello fuera estrictamente necesario. Y si por ejemplo,  $a = 6$  y  $b = 4$ , entonces  $\frac{b}{a} = \frac{2}{3}$ , así que para eliminar  $b$  basta con la operación  $b \times 3 - a \times 2$ ; es decir, basta con multiplicar  $b$  por el denominador de la fracción simplificada y  $a$  por el numerador. El siguiente código usa esta idea además del hecho de que si  $n$  es un número del tipo `Fraction`, es decir, de la forma  $\frac{a}{b}$ , entonces `n.numerator` nos da el numerador de la fracción simplificada y `n.denominator` es el denominador.

<sup>1</sup>pues por hipótesis de inducción se supone que la submatriz  $B$  es escalonable

41a *⟨Escalonamiento de una matriz mediante eliminación por columnas 40⟩+≡*

```

class GaussCLsd(Matrix):
    def __init__(self, data):
        """Escalona una Matrix con eliminación por columnas (sin div.)"""
        A = Matrix(data)
        r = 0
        for i in range(1,A.m+1):
            p = pivote((i|A),r)
            if p > 0:
                r += 1
                A & T( {p, r} )
                A & T([ T([ Fraction((i|A|j),(i|A|r)).denominator, j), \
                        (-Fraction((i|A|j),(i|A|r)).numerator, r, j))] \
                      for j in range(r+1,A.n+1)])

        super(self.__class__ ,self).__init__(A.lista)

```

This code is used in chunk 37.

Uses Matrix 10, pivote 39, and T 33b.

En la mayoría de los libros de Álgebra Lineal, la eliminación gaussiana consiste en lograr una matriz triangular superior mediante operaciones con las *filas* siguiendo el esquema (NO→SE); es decir, colocando el primer pivote en la primera fila y haciendo ceros por debajo. Pero esto lo podemos lograr con el algoritmo anterior. Basta transponer la matriz, escalonar operando con las columnas, y volver a transponer el resultado:

41b *⟨Escalonamiento de una matriz mediante eliminación por filas 41b⟩≡*

```

class GaussFU(Matrix):
    def __init__(self, data):
        """Escalona una Matrix con eliminación por filas (transf. Gauss)"""
        A = ~GaussCL(~Matrix(data))
        super(self.__class__ ,self).__init__(A.lista)

```

This code is used in chunk 37.

Uses Matrix 10.

También es arbitrario el orden en que se triangula la matriz. Por ejemplo, podemos obtener una matriz triangular superior mediante eliminación por columnas si en lugar de “ir de arriba a abajo” y de “izquierda a derecha” (escalando desde la esquina Noroeste a la esquina Sureste: NO→SE), comenzamos por las filas inferiores y realizando la eliminación de derecha a izquierda (SE→NO).

Si aplicamos el método `reversed` de la clase `Matrix` sobre `A` obtenemos una matriz cuyas columnas son las de `A` pero en orden inverso (primero la última, luego la penúltima, etc.). Así que si aplicamos el método `reversed` sobre la transpuesta de `A` y volvemos a transponer, obtenemos una matriz cuyas filas aparecen en el orden inverso de las de `A`. Si aplicamos de nuevo `reversed` habremos invertido el orden tanto de las filas como de las columnas.

Por tanto, si invertimos el orden de las filas y columnas, escalonamos con `GaussCL` y volvemos a invertir el orden de las filas y las columnas, obtenemos el citado escalonamiento (SE→NO) que arroja una matriz triangular superior:

42a *⟨Escalonamiento de una matriz mediante eliminación por columnas 40⟩+≡*

```
class GaussCU(Matrix):
    def __init__(self, data):
        """Escalona una Matrix con eliminación por columnas (transf. Gauss)"""
        A = reversed(~reversed(~ GaussCL( reversed(~reversed(~ Matrix(data) ))) ))
        super(self.__class__, self).__init__(A.lista)
```

This code is used in chunk 37.  
Uses Matrix 10.

### 2.1.1 Variantes que guardan los pasos dados sobre la matriz original

Las variantes de más abajo tienen un código con la misma estructura: con dos trozos de código comunes a todas las variantes (uno al principio, *⟨Definición del método PasosYEscritura 43a⟩* y otro al final *⟨Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b⟩* que se describen un poco más abajo). Entre ambos trozos aparece el código que aplica las secuencias de transformaciones elementales necesarias en cada variante.

42b *⟨Estructura de las variantes de aplicación del método de Gauss 42b⟩≡*

```
class NombreMetodo(Matrix):
    def __init__(self, data, rep=0):
        """Descripción de lo que hace la variante particular"""

        ⟨Definición del método PasosYEscritura 43a⟩

        A = Matrix(data); pasos = []; r = 0

        for i in (indices_en_el_orden_en_el_que_se_recorren_las_filas_de_A):

            p = pivote(fila_en_el_orden_en_el_que_se_recorren_las_columnas, r)

            if p > 0:
                # si hay nuevo pivote en la fila
                r += 1
                # el contador cuenta un nuevo pivote y ...

                Se_definen_las_transf_necesarias,
                se_almacenan_en_la_variable_pasos,
                y_se_aplican_sobre_A

        ⟨Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b⟩
```

Root chunk (not used in this document).  
Uses Matrix 10 and pivote 39.

Si queremos mostrar los pasos, es más legible mostrar solo los que realmente cambian la matriz (omitiendo sustituciones de una columna por ella misma, productos de una columna por 1, o sumas de un vector nulo a una columna).

Además, si hemos encadenado varios procedimientos de eliminación, deberíamos poder ver los pasos desde el principio hasta el final; añadiendo a la escritura en L<sup>A</sup>T<sub>E</sub>X de dicho pasos previos (TexPasosPrev), los pasos dados como argumento. Si TexPasosPrev es vacío, la escritura comienza con la representación de data. Todo esto es lo que hace el método PasosYEscritura:

43a *<Definición del método PasosYEScritura 43a>≡*

```
def PasosYEScritura(data,pasos,TeXPasosPrev=[]):
    """Escribe en LaTeX los pasos efectivos dados"""
    A = Matrix(data); p = [[],[]]
    tex = latex(data) if len(TeXPasosPrev)==0 else TeXPasosPrev
    for l in range(0,2):
        p[l] = [ T([j for j in pasos[l][i].t if (isinstance(j,set) and len(j)>1) \
            or (isinstance(j,tuple) and len(j)==3 and j[0]!=0) \
            or (isinstance(j,tuple) and len(j)==2 and j[0]!=1) )) \
            for i in range(0,len(pasos[l]))]
        p[l] = [ t for t in p[l] if len(t.t)!=0] # quitamos abreviaturas vacías
        if l==0:
            for i in reversed(range(0,len(p[l]))):
                tex += '\\xrightarrow{' + latex(p[l][i]) + '}'
                if isinstance (data, Matrix):
                    tex += latex( p[l][i] & A )
                elif isinstance (data, BlockMatrix):
                    tex += latex( key(data.lm)|(p[l][i] & A)|key(data.ln) )
            if l==1:
                for i in range(0,len(p[l])):
                    tex += '\\xrightarrow{' + latex(p[l][i]) + '}'
                    if isinstance (data, Matrix):
                        tex += latex( A & p[l][i] )
                    elif isinstance (data, BlockMatrix):
                        tex += latex( key(data.lm)|(A & p[l][i])|key(data.ln) )
        return tex
```

This code is used in chunks 42b and 44–51.  
Uses BlockMatrix 61b, Matrix 10, and T 33b.

Al representar los pasos de eliminación, consideramos si `data` fue obtenido mediante un proceso previo de eliminación, para mostrar el proceso completo. El atributo `tex` guarda el código  $\text{\LaTeX}$  que muestra el proceso completo. El atributo `rank` guarda el rango y `pasos` las listas de transformaciones elementales empleadas.

43b *<Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>≡*

```
pasosPrevios = data.pasos if hasattr(data, 'pasos') and data.pasos else [[],[]]
TeXPasosPrev = data.tex if hasattr(data, 'tex') and data.tex else []
self.tex = PasosYEScritura(data, pasos, TeXPasosPrev)
if rep:
    from IPython.display import display, Math
    display(Math(self.tex))
self.rank = r
pasos[0] = pasos[0] + pasosPrevios[0]
pasos[1] = pasosPrevios[1] + pasos[1]
self.pasos = pasos
super(self.__class__,self).__init__(A.lista)
```

This code is used in chunks 42b and 44–48.

La primera variante hace lo mismo que GaussCL pero guarda los pasos dados en el atributo `pasos` y en el atributo `tex` el código L<sup>A</sup>T<sub>E</sub>X que permite representar el procedimiento de eliminación gaussiana paso a paso.

44a *<Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>≡*

```
class ECL(Matrix):
    def __init__(self, data, rep=0):
        """Escalona una Matrix con eliminación por columnas (transf. Gauss)"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for i in range(1,A.m+1):
            p = pivote((i|A),r)
            if p > 0:
                r += 1
                Tr = T([ {p, r} ])
                pasos[1] += [Tr]
                A & T( Tr )
                Tr = T([(Fraction(-(i|A|j),(i|A|r)), r, j) for j in range(r+1,A.n+1)])
                pasos[1] += [Tr] if Tr.t else []
                A & T( Tr )
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>
```

This definition is continued in chunks 44–46.  
 This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante evita las divisiones

44b *<Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>+≡*

```
class ECLsd(Matrix):
    def __init__(self, data, rep=0):
        """Escalona por eliminación por columnas (sin divisiones)"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for i in range(1,A.m+1):
            p = pivote((i|A),r)
            if p > 0:
                r += 1
                Tr = T( [ {p, r} ] )
                pasos[1] += [Tr]
                A & T( Tr )
                Tr = T( [ T( [ ( Fraction((i|A|j),(i|A|r)).denominator, j), \
                    (-Fraction((i|A|j),(i|A|r)).numerator, r, j) ] ) \
                    for j in range(r+1,A.n+1) ] )
                pasos[1] += [Tr] if Tr.t else []
                A & T( Tr )
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>
```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante (parecida a la primera) asegura que los pivotes sean todos iguales a uno

45a *<Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>+≡*

```

class ECLN(Matrix):
    def __init__(self, data, rep=0):
        """Escalona por eliminación por columnas haciendo pivotes unitarios"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for i in range(1,A.m+1):
            p = pivote((i|A),r)
            if p > 0:
                r += 1
                Tr = T( [ {p, r} ] )
                pasos[1] += [Tr]
                A & T( Tr )
                Tr = T( [ (Fraction(1,(i|A|r)), r) ] )
                pasos[1] += [Tr]
                A & T( Tr )
                Tr = T( [(-i|A|j), r, j) for j in range(r+1,A.n+1)] )
                pasos[1] += [Tr] if Tr.t else []
                A & T( Tr )
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>

```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante logra una matriz triangular superior eliminando de derecha a izquierda y de abajo a arriba.

45b *<Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>+≡*

```

class ECU(Matrix):
    def __init__(self, data, rep=0):
        """Escalona una Matrix con eliminación por columnas (transf. Gauss)"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for i in reversed(range(1,A.m+1)):
            p = pivote(reversed(i|A), r)
            if p > 0:
                r += 1
                Tr = T( [ {A.n-p+1, A.n-r+1} ] )
                pasos[1] += [Tr]
                A & T( Tr );
                Tr = T([ (Fraction(-(i|A|j), (i|A|(A.n-r+1)))), A.n-r+1, j) \
                        for j in reversed(range(1,A.n-r+1)) ] )
                pasos[1] += [Tr] if Tr.t else []
                A & T( Tr )
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>

```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante hace lo mismo que la anterior, pero asegurandose de que los pivotes sean unos.

46a *<Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>+≡*

```

class ECUN(Matrix):
    def __init__(self, data, rep=0):
        """Escalona una Matrix con eliminación por columnas (transf. Gauss)"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for i in reversed(range(1,A.m+1)):
            p = pivote(reversed(i|A), r)
            if p > 0:
                r += 1
                Tr = T( [ {A.n-p+1, A.n-r+1} ] )
                pasos[1] += [Tr]
                A & T( Tr )
                Tr = T( [ (Fraction(1,(i|A|(A.n-r+1))), A.n-r+1 ) ] )
                pasos[1] += [Tr]
                A & T( Tr )
                Tr = T([ -(i|A|j), A.n-r+1, j) for j in reversed(range(1,A.n-r+1)) ] )
                pasos[1] += [Tr] if Tr.t else []
                A & T( Tr )
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>

```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante también obtiene una forma escalonada triangular superior, pero evitando realizar divisiones.

46b *<Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a>+≡*

```

class ECUsd(Matrix):
    def __init__(self, data, rep=0):
        """Escalona una Matrix con eliminación por columnas (transf. Gauss)"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for i in reversed(range(1,A.m+1)):
            p = pivote(reversed(i|A), r)
            if p > 0:
                r += 1; Tr = T( [ {A.n-p+1, A.n-r+1} ] ); pasos[1] += [Tr]
                A & T( Tr )
                Tr = T( [ T( \
                    [ ( Fraction((i|A|j),(i|A|(A.n-r+1))).denominator, j), \
                      (-Fraction((i|A|j),(i|A|(A.n-r+1))).numerator, A.n-r+1, j) ] \
                    ) for j in reversed(range(1,A.n-r+1)) ] )
                pasos[1] += [Tr] if Tr.t else []
                A & T( Tr )
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>

```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

Al operar con las filas, el método `PasosYEscritura` cambia, pues hay que tener en cuenta que las transformaciones elementales actúan por la izquierda.

47a *<Variantes del escalonamiento por eliminación por filas que guardan los pasos dados 47a>≡*

```

class EFU(Matrix):
    def __init__(self, data, rep=0):
        """Escalona una Matrix con eliminación por filas (transf. Gauss)"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for j in range(1,A.n+1):
            p = pivote((A|j),r)
            if p > 0:
                r += 1
                Tr = T( [ {p, r} ] ); pasos[0] += [Tr]
                T( Tr ) & A
                Tr = T( [(Fraction(-(i|A|j),(r|A|r)), r, i) for i in range(r+1,A.m+1)] )
                pasos[0] += list(reversed([Tr]))
                T( Tr ) & A
            pasos[0]=list(reversed(pasos[0]))
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>

```

This definition is continued in chunks 47b and 48a.  
 This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante hace lo mismo que la anterior, pero asegurándose de que los pivotes sean unos.

47b *<Variantes del escalonamiento por eliminación por filas que guardan los pasos dados 47a>+≡*

```

class EFUN(Matrix):
    def __init__(self, data, rep=0):
        """Escalona con eliminación por filas haciendo pivotes unitarios"""
        <Definición del método PasosYEscritura 43a>
        A = Matrix(data); pasos = [[],[]]; r = 0
        for j in range(1,A.n+1):
            p = pivote((A|j),r)
            if p > 0:
                r += 1
                Tr = T( [ {p, r} ] ); pasos[0] += [Tr]
                T( Tr ) & A
                Tr = T( [ (Fraction(1,(j|A|r)), r) ] ); pasos[0] += [Tr]
                T( Tr ) & A
                Tr = T( [(-(i|A|j), r, i) for i in range(r+1,A.m+1)] )
                pasos[0] += list(reversed([Tr])) if Tr.t else []
                T( Tr ) & A
            pasos[0]=list(reversed(pasos[0]))
        <Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b>

```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

La siguiente variante obtiene una matriz triangulas inferior operando con las filas de “abajo a arriba” y de “derecha a izquierda”



48a  $\langle$ Variantes del escalonamiento por eliminación por filas que guardan los pasos dados 47a $\rangle \equiv$

```

class EFL(Matrix):
    def __init__(self, data, rep=0):
        """Escalona una Matrix con eliminación por filas (L)"""
         $\langle$ Definición del método PasosYEScritura 43a $\rangle$ 
        A = Matrix(data); pasos = [[],[]]; r = 0
        for j in reversed(range(1,A.n+1)):
            p = pivote(reversed(A[j]),r)
            if p > 0:
                r += 1
                Tr = T( [ {A.m-p+1, A.m-r+1} ] );          pasos[0] += [Tr]
                T( Tr ) & A
                Tr = T([ (Fraction(-(i|A[j]), ((A.m-r+1)|A[j])), A.m-r+1, i) \
                        for i in (range(1,A.m-r+1)) ] )
                pasos[0] += [Tr] if Tr.t else []
                T( Tr ) & A
        pasos[0]=list(reversed(pasos[0]))
         $\langle$ Se almacenan los atributos tex, pasos y rank; y se devuelve el resultado 43b $\rangle$ 

```

This code is used in chunk 37.  
 Uses Matrix 10, pivote 39, and T 33b.

Si hemos diagonalizado y solo nos falta que la diagonal esté compuesta por unos, lo podemos hacer dividiendo cada columna por el valor de elemento de la diagonal (si no es nulo).

48b  $\langle$ Normalizando la diagonal principal para que sus componentes no nulos sean unos 48b $\rangle \equiv$

```

class NormDiag(Matrix):
    def __init__(self, data, rep=0):
        """Normaliza a uno los componentes no nulos de la diagonal principal"""
         $\langle$ Definición del método PasosYEScritura 43a $\rangle$ 
        A = Matrix(data); pasos = [[],[]];
        Tr = T([ (Fraction(1,(j|A[j])), j) for j in range(1,A.n+1) if (j|A[j])!=0 ] )
        pasos[1] = [Tr] if Tr.t else []
        A & T( [Tr] )
        PPrevios = data.pasos if hasattr(data, 'pasos') and data.pasos else [[],[]]
        TexPPrev = data.tex if hasattr(data, 'tex') and data.tex else []
        self.tex = PasosYEScritura(data, pasos, TexPPrev)
        if rep:
            from IPython.display import display, Math
            display(Math(self.tex))
        self.rank = data.rank if hasattr(data, 'rank') and data.rank else []
        pasos[1] = PPrevios[1] + pasos[1]
        self.pasos = pasos
        super(self.__class__,self).__init__(A.lista)

```

This code is used in chunk 37.  
 Uses Matrix 10 and T 33b.

## 2.2 Inversión de una matriz por eliminación Gaussiana

49a *⟨Método auxiliar que calcula la inversa de una Matrix 49a⟩≡*

```
def MatrixInversa( self ):
    """Calculo de la inversa de una matriz"""
    L = ECL(self)

    if L.rank < L.n:
        raise ArithmeticError('Matrix singular')

    return Matrix( I(L.n) & T(ECUN(L).pasos[1]) )
```

This code is used in chunk 25b.  
Uses Matrix 10 and T 33b.

El siguiente código obtiene la inversa de una matriz cuadrada operando sobre las columnas, y muestra los pasos dados hasta llegar a ella, o hasta llegar a una matriz singular

49b *⟨Invirtiendo una matriz 49b⟩≡*

```
class InvMat(Matrix):
    def __init__(self, data, rep=0):
        """Devuelve la matriz inversa y los pasos dados sobre las columnas"""
        ⟨Definición del método PasosYEscritura 43a⟩
        A = Matrix(data)

        if A.m != A.n:
            raise ValueError('Matrix no cuadrada')

        L = ECL(A)
        if L.rank < A.n:
            raise ArithmeticError('Matrix singular')

        M = ECUN(L)
        stack = BlockMatrix([[A],[I(A.n)]])
        self.tex = PasosYEscritura(stack, M.pasos)
        if rep:
            from IPython.display import display, Math
            display(Math(self.tex))

        Inv = I(A.n) & T(M.pasos[1])
        self.pasos = M.pasos
        super(self.__class__,self).__init__(Inv.lista)
```

This definition is continued in chunk 50.  
This code is used in chunk 37.  
Uses BlockMatrix 61b, Matrix 10, and T 33b.

El siguiente código obtiene la inversa de una matriz operando sobre las filas, y muestra los pasos dados hasta llegar a ella

50a

```

<Invirtiendo una matriz 49b>+=
class InvMatF(Matrix):
    def __init__(self, data, rep=0):
        """Devuelve la matriz inversa y los pasos dados sobre las filas"""
        <Definición del método PasosYEscritura 43a>
        A      = Matrix(data)
        Id      = EFUN(EFL(A))
        stack = BlockMatrix([[A,I(A.m)]])
        self.tex = PasosYEscritura(stack, Id.pasos)
        if rep:
            from IPython.display import display, Math
            display(Math(self.tex))

        Inv      = T(Id.pasos[0]) & I(A.n)
        self.pasos = Id.pasos
        super(self.__class__,self).__init__(Inv.lista)

```

This code is used in chunk 37.

Uses BlockMatrix 61b, Matrix 10, and T 33b.

El siguiente código obtiene la inversa de una matriz operando primero sobre las filas hasta obtener una matriz triangular superior con pivotes iguales a uno y luego operando sobre las columnas hasta obtener la identidad.

50b

```

<Invirtiendo una matriz 49b>+=
class InvMatFC(Matrix):
    def __init__(self, data, rep=0):
        """Devuelve la matriz inversa y los pasos dados sobre las filas y columnas"""
        <Definición del método PasosYEscritura 43a>
        A      = Matrix(data)
        Id      = ECLN(EFU(A))
        stack = BlockMatrix([[A,I(A.m)], [I(A.n),M0(A.m,A.n)]])
        self.tex = PasosYEscritura(stack, Id.pasos)
        if rep:
            from IPython.display import display, Math
            display(Math(self.tex))

        Inv      = ( I(A.n) & T(Id.pasos[1]) ) * ( T(Id.pasos[0]) & I(A.n) )
        self.pasos = Id.pasos
        super(self.__class__,self).__init__(Inv.lista)

```

This code is used in chunk 37.

Uses BlockMatrix 61b, Matrix 10, and T 33b.

50c

```

<normal 50c>=
class Normal(Matrix):

```

```

def __init__(self, data):
    """Escalona por Gauss obteniendo una matriz cuyos pivotes son unos"""
    A = Matrix(data); r = 0
    self.rank = []
    for i in range(1,A.n+1):
        p = pivote((i|A),r)
        if p > 0:
            r += 1
            A & T( {p, r} )
            A & T( (1/Fraction(i|A|r), r) )
            A & T( [ -(i|A|k), r, k) for k in range(r+1,A.n+1)] )

        self.rank+= [r]

    super(self.__class__ ,self).__init__(A.lista)

```

This code is used in chunk 37.

Uses Matrix 10, pivote 39, and T 33b.

Con este código ya podemos hacer muchísimas cosas. Por ejemplo, eliminación gaussiana para encontrar el espacio nulo de una matriz!

51a

*<sistema 51a>*≡

```

def homogenea(A):
    """Devuelve una BlockMatriz con la solución del problema homogéneo"""
    stack=Matrix(BlockMatrix([[A],[I(A.n)]]))
    soluc=Normal(stack)
    col=soluc.rank[A.m-1]
    return {A.m} | soluc | {col}

```

This code is used in chunk 37.

Uses BlockMatrix 61b and Matrix 10.

El siguiente código obtiene una base del espacio nulo de una matriz

51b

*<Resolviendo un sistema homogéneo 51b>*≡

```

class EspacioNulo:
    def __init__(self, data, rep=0):
        """Describe el espacio nulo de una matriz y los pasos para encontrarlo"""
        <Definición del método PasosYEScritura 43a>

        A      = Matrix(data)
        L      = ECL(A)
        E      = I(A.n) & T(L.pasos[1])

        self.base = list([Vector(E|j) for j in range(L.rank+1, L.n+1)])

        stack = BlockMatrix([[A],[I(A.n)]])
        self.tex  = PasosYEScritura(stack, L.pasos)
        if rep:
            from IPython.display import display, Math
            display(Math(self.tex))

    def __repr__(self):
        """ Muestra una matriz en su representación python """
        return 'Combinaciones lineales de {' + repr(self.base) + '}'

```

```
def _repr_html_(self):
    """ Construye la representación para el entorno jupyter notebook """
    return html(self.latex())

def latex(self):
    """ Construye el comando LaTeX """
    return '\\text{Conjunto de combinaciones lineales de }\\left\\{' + \
        '\\;''.join([latex(self.base[i]) for i in range(0,len(self.base))]) + \
        '\\right\\}'
```

This code is used in chunk 37.

Uses [BlockMatrix 61b](#), [Matrix 10](#), [T 33b](#), and [Vector 6b](#).

## Capítulo 3

# Otros trozos de código

### 3.1 Métodos de representación para el entorno Jupyter

El método `html`, escribe el inicio y el final de un párrafo en `html` y en medio del párrafo escribirá la cadena `TeX`; que contendrá el código `LATEX` de las expresiones matemáticas que queremos que se muestren en pantalla cuando usamos **Jupyter Notebook**. En el navegador, la librería **MathJax** de Javascript se encargará de convertir la expresión `LATEX` en la gráfica correspondiente.

53a *⟨Método html general 53a⟩*≡

```
def html(TeX):  
    """ Plantilla HTML para insertar comandos LaTeX """  
    return "<p style=\"text-align:center;\">$" + TeX + "$</p>"
```

This code is used in chunk 37.

El método `latex`, convertirá en cadena de caracteres los inputs de tipo `str`, `float` o `int`<sup>1</sup>, y en el resto de casos llamara al método `latex` de la clase desde la que se invocó a este método (es un truqui recursivo para que trate de manera parecida la expresiones en `LATEX` y los tipos de datos que corresponden a números `int` o `float`).

53b *⟨Método latex general 53b⟩*≡

```
def latex(a):  
    if isinstance(a,float) | isinstance(a,int) | isinstance(a,str):  
        return str(a)  
    else:  
        return a.latex()
```

This code is used in chunk 37.

Si el objeto `a` representar no es un número de coma flotante (`float`) ni tampoco un entero (`int`), el método general `latex` llamará el método `latex` de la clase correspondiente. Por tanto, si `a` es un `Vector`, una `Matrix`, o una transformación elemental (`T`), se llama al método `a.latex` definido en la clase correspondiente a dicho objeto `a`.<sup>2</sup> Sin

<sup>1</sup>resulta que para los tipos de datos `int` y `float` no es posible definir un nuevo método de representación. Afortunadamente la cadena de caracteres que representa el número nos vale perfectamente en ambos casos (tanto para los números enteros, `int`, como los números con decimales, `float`).

<sup>2</sup>más adelante tendré que incluir métodos de representación para otros tipos de datos, por ejemplo para poder representar vectores o matrices con polinomios.

embargo, la clase `Fraction` no tiene definidos los métodos de representación `html` o `latex`. Así pues, para representar fracciones necesitamos incorporar estos métodos en la preexistente clase `Fraction` que hemos importado desde la librería `fractions`. Primero definimos el método `_repr_html_fraction` (que sencillamente llamara al método `latex`) y luego definimos el método `latex_fraction` (en el nombre de ambos métodos he incluido la coletilla `fraction` para recordar que son los métodos que usaremos para la clase `fraction`). Si en  $\text{\LaTeX}$  queremos representar la fracción  $\frac{a}{b}$  escribimos el código: `\frac{a}{b}`. Pero cuando el denominador es  $b = 1$ , no nos gusta escribir  $\frac{a}{1}$ , preferimos mostrar solamente el numerador  $a$ . Esto es precisamente lo que hace el método `latex_fraction` de más abajo.

Finalmente, con la función `setattr`, añadimos a la clase `Fraction` un método que se llamará `'_repr_html_'` (y que hace lo que hemos indicado al definir `_repr_html_fraction`), y un método que se llamará `'latex'` (y que hace los que hemos indicado al definir `latex_fraction`).

54a *⟨Métodos html y latex para fracciones 54a⟩≡*

```
def _repr_html_fraction(self):
    return html(self.latex())

def latex_fraction(self):
    if self.denominator == 1:
        return repr(self.numerator)
    else:
        return "\\frac{" + repr(self.numerator) + "}{" + repr(self.denominator) + "}"

setattr(Fraction, '_repr_html_', _repr_html_fraction)
setattr(Fraction, 'latex', latex_fraction)
```

This code is used in chunk 37.

## 3.2 Completando la clase Vector

### 3.2.1 Representación de la clase Vector

Ahora necesitamos indicar a Python cómo representar los objetos de tipo `Vector`.

Los vectores, son secuencias finitas de números que representaremos con paréntesis, bien en forma de fila

$$\mathbf{v} = (v_1, \dots, v_n)$$

o bien en forma de columna

$$\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

Definimos tres representaciones distintas. Una para la línea de comandos de Python de manera que escriba `Vector` y a continuación encierre la representación de `self.lista` (el sistema de números), entre paréntesis. Por ejemplo, si la lista es `[a,b,c]`, Python nos mostrará en la línea de comandos: `Vector([a,b,c])`.

La representación en  $\text{\LaTeX}$  encierra un vector (en forma de fila o de columna) entre paréntesis; y es usada a su vez por la representación html usada por el entorno Jupyter.

54b *⟨Representación de la clase Vector 54b⟩≡*

```
def __repr__(self):
    """ Muestra el vector en su representación python """
    return 'Vector(' + repr(self.lista) + ')'
```

```

def _repr_html_(self):
    """ Construye la representación para el entorno jupyter notebook """
    return html(self.latex())

def latex(self):
    """ Construye el comando LaTeX """
    if self.rpr == 'fila':
        return '\\begin{pmatrix}' + \
            ',&'.join([latex(self[i] for i in range(1,self.n+1))] + \
            '\\end{pmatrix}'
    else:
        return '\\begin{pmatrix}' + \
            '\\\\'.join([latex(self[i] for i in range(1,self.n+1))] + \
            '\\end{pmatrix}'

```

This code is used in chunk 6b.

### 3.2.2 Otros métodos para la clase Vector

Para jugar con el método de Gauss nos vendrá bien poder hacer el “reverso” de un **Vector**, es decir, obtener el **Vector** cuyas componentes están ordenadas en sentido inverso al original: la primera componente es la última, la segunda es la penúltima, etc.

55a *<Reverso de un Vector 55a>≡*

```

def __reversed__(self):
    """Devuelve el reverso de un Vector"""
    return Vector(self.lista[::-1])

```

This code is used in chunk 6b.  
Uses **Vector** 6b.

## 3.3 Completando la clase Matrix

### 3.3.1 Otras formas de instanciar una Matrix

Si se introduce una lista (tupla) de listas o tuplas, creamos una matriz fila a fila. Si se introduce una **Matrix** creamos una copia de la matriz. Si se introduce una **BlockMatrix** se elimina el particionado y que crea una única matriz. Si el argumento no es correcto se informa con un error.

55b *<Creación del atributo lista cuando sis no es una lista (o tupla) de Vectores 55b>≡*

```

if isinstance(sis, Matrix):
    self.lista = sis.lista.copy()

elif isinstance(sis, BlockMatrix):
    self.lista = [Vector([ sis.lista[i][j]|k|s \
                           for i in range(sis.m) for s in range(1,(sis.lm[i])+1) ]) \
                  for j in range(sis.n) for k in range(1,(sis.ln[j])+1) ]

```



```

<Verificación de que al instanciar Matrix el argumento sis es indexable 56a>

elif isinstance(sis[0], (list, tuple)):
    <Verificación de que todas las filas de la matriz tendrán la misma longitud 56b>
    self.lista = [ Vector([ sis[i][j] for i in range(len(sis  )) ]) \
                  for j in range(len(sis[0])) ]

```

This code is used in chunk 9.

Uses BlockMatrix 61b, Matrix 10, and Vector 6b.

### 3.3.2 Códigos que verifican que los argumentos son correctos

```

56a <Verificación de que al instanciar Matrix el argumento sis es indexable 56a>≡
    elif not isinstance(sis, (list, tuple)):
        raise ValueError(\
            ',argumento: list (tuple) de Vectores (lists o tuples);  BlockMatrix; o Matrix!')

```

This code is used in chunk 55b.

Uses BlockMatrix 61b and Matrix 10.

```

56b <Verificación de que todas las filas de la matriz tendrán la misma longitud 56b>≡
    if not all ( (type(sis[0])==type(v)) and (len(sis[0])==len(v)) for v in iter(sis) ):
        raise ValueError('no todas son listas o no tienen la misma longitud!')

```

This code is used in chunk 55b.

```

56c <Verificación de que todas las columnas de la matriz tienen la misma longitud 56c>≡
    if not all ( isinstance(v, Vector) and (sis[0].n == v.n) for v in iter(sis)):
        raise ValueError('no todos son vectores, o no tienen la misma longitud!')

```

This code is used in chunk 9.

Uses Vector 6b.

### 3.3.3 Representación de la clase Matrix

Y como en el caso de los vectores, construimos los dos métodos de presentación. Uno para la consola de comandos que escribe **Matrix** y entre paréntesis la lista de listas (es decir la lista de filas); y otro para el entorno Jupyter (que a su vez usa la representación  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  que representa las matrices entre corchetes como en las notas de la asignatura). Si `self.lista` es una lista vacía, se representa una matriz vacía.

```

56d <Representación de la clase Matrix 56d>≡
    def __repr__(self):
        """ Muestra una matriz en su representación python """
        return 'Matrix(' + repr(self.lista) + ')'

    def _repr_html_(self):

```

```

        """ Construye la representación para el entorno jupyter notebook """
        return html(self.latex())

    def latex(self):
        """ Construye el comando LaTeX """
        return '\\begin{bmatrix}' + \
            '\\\\'.join(['&'.join([latex(i|self|j) for j in range(1,self.n+1) ]) \
                           for i in range(1,self.m+1) ]) + \
            '\\end{bmatrix}'

This code is used in chunk 10.

```

### 3.3.4 Otros métodos para la clase Matrix

Para jugar con el método de Gauss nos vendrá bien poder hacer el “reverso” de una `Matrix`, es decir, obtener la `Matrix` cuyas columnas están ordenadas en sentido inverso al original: la columna es la última, la segunda es la penúltima, etc.

57a *<Reverso de una Matrix 57a>≡*

```

def __reversed__(self):
    """Devuelve el reverso de una Matrix"""
    return Matrix(self.lista[::-1])

This code is used in chunk 10.
Uses Matrix 10.

```

## 3.4 Completando la clase T

### 3.4.1 Otras formas de instanciar una T

Si se instancia `T` usando otra Transformación elemental, sencillamente se copia el atributo `t`. Si se instancia `T` usando una lista (no vacía) de Transformaciones elementales, el atributo `t` será la lista de abreviaturas resultante de concatenar las abreviaturas de todas las Transformaciones elementales de la lista empleada en la instanciación.

57b *<Creación del atributo t cuando se instancia con otra T o lista de Ts 57b>≡*

```

if isinstance(t, T):
    self.t = t.t

elif isinstance(t, list) and t and isinstance(t[0], T):
    self.t = [val for sublist in [x.t for x in t] for val in CreaLista(sublist)]

This code is used in chunk 29.
Uses CreaLista 30b and T 33b.

```

### 3.4.2 Representación de la clase T

De nuevo construimos los dos métodos de presentación. Uno para la consola de comandos que escribe T y entre paréntesis la abreviatura (una tupla o un conjunto) que representa la transformación. Así,

- $T(\{1, 5\})$  : intercambio entre los vectores primero y quinto.
- $T(6, 2)$  : multiplica por seis el segundo vector.
- $T(-1, 2, 3)$  : resta el segundo vector al tercero.

La otra representación es para el entorno Jupyter y replica la notación usada en los apuntes de la asignatura:

Python	Representación en Jupyter
$T(\{1, 5\})$	$\tau_{[1 \rightleftharpoons 5]}$
$T(6, 2)$	$\tau_{[(6)2]}$
$T(-1, 2, 3)$	$\tau_{[(-1)2+3]}$

Los apuntes de la asignatura usan una notación matricial, y por tanto es una notación que discrimina entre operaciones sobre las filas o las columnas, situando los operadores a la izquierda o a la derecha de la matriz. En este sentido, nuestra notación en Python hace lo mismo. Así, en la siguiente tabla, la columna de la izquierda corresponde a operaciones sobre las filas, y la columna de la derecha a las operaciones sobre las columnas:

Mates II	Python	Mates II	Python
$\tau_{[i \rightleftharpoons j]} \mathbf{A}$	$T(\{i, j\}) \& \mathbf{A}$	$\mathbf{A} \tau_{[i \rightleftharpoons j]}$	$\mathbf{A} \& T(\{i, j\})$
$\tau_{[(a)i]} \mathbf{A}$	$T(a, i) \& \mathbf{A}$	$\mathbf{A} \tau_{[(a)i]}$	$\mathbf{A} \& T(a, i)$
$\tau_{[(a)i+j]} \mathbf{A}$	$T(a, i, j) \& \mathbf{A}$	$\mathbf{A} \tau_{[(a)i+j]}$	$\mathbf{A} \& T(a, i, j)$

### Secuencias de transformaciones

Considere las siguientes transformaciones

- multiplicar por 2 el primer vector, cuya abreviatura es:  $(2, 1)$
- intercambiar el tercer vector por cuarto, cuya abreviatura es:  $\{3, 4\}$

Para indicar una secuencia que contiene ambas transformaciones, usaremos una lista de abreviaturas:  $[(2, 1), \{3, 4\}]$ . De esta manera, cuando componemos ambas operaciones:  $T(2, 1) \& T(\{3, 4\})$ , nuestra librería nos devuelve la transformación composición de las dos operaciones **en el orden en el que han sido escritas**:

al escribir  $T(2, 1) \& T(\{3, 4\})$  Python nos devuelve  $T([(2, 1), \{3, 4\}])$

Por tanto, si queremos realizar dichas operaciones sobre las columnas de la matriz  $\mathbf{A}$ , podemos hacerlo de dos formas:

- $\mathbf{A} \& T(2, 1) \& T(\{3, 4\})$  (indicando las transformaciones de una en una)
- $\mathbf{A} \& T([(2, 1), \{3, 4\}])$  (usando la transformación composición de todas ellas)

y si queremos operar sobre la filas hacemos exactamente igual, pero a la izquierda de la matriz

- $T(2, 1) \& T(\{3, 4\}) \& \mathbf{A}$
- $T([(2, 1), \{3, 4\}]) \& \mathbf{A}$

### Representación de una secuencia de transformaciones.

Representación en la consola de Python	Representación en Jupyter
$T([(2, 1), (1, 3, 2)])$	$\tau_{[(2)1] \atop [(1)3+2]}$

59a *<Representación de la clase T 59a>*≡

```
def __repr__(self):
    """ Muestra T en su representación python """
    return 'T(' + repr(self.t) + ')'

def _repr_html_(self):
    """ Construye la representación para el entorno jupyter notebook """
    return html(self.latex())

def latex(self):
    """ Construye el comando LaTeX """
    def simbolo(t):
        """Escribe el símbolo que denota una transformación elemental particular"""
        if isinstance(t, set):
            return '\\left[\\mathbf{' + latex(min(t)) + \
                '\\rightleftharpoons\\mathbf{' + latex(max(t)) + '}\\right]'
        if isinstance(t, tuple) and len(t) == 2:
            return '\\left[\\left(' + \
                latex(t[0]) + '\\right)\\mathbf{' + latex(t[1]) + '\\right]'
        if isinstance(t, tuple) and len(t) == 3:
            return '\\left[\\left(' + latex(t[0]) + '\\right)\\mathbf{' + \
                latex(t[1]) + '\\right)\\mathbf{' + latex(t[2]) + '\\right]'

        if isinstance(self.t, (set, tuple)):
            return '\\underset{' + simbolo(self.t) + '}{\\mathbf{\\tau}}'

        elif isinstance(self.t, list):
            return '\\underset{\\begin{subarray}{c} ' + \
                '\\\\'.join([simbolo(i) for i in self.t]) + \
                '\\end{subarray}}{\\mathbf{\\tau}}'
```

This code is used in chunk 33b.  
Uses T 33b.

### 3.5 Vectores y Matrices especiales

#### Notación en Mates 2

Los vectores cero **0** y las matrices cero **0** se pueden implementar como subclases de la clase **Vector** y **Matrix** (pero tenga en cuenta que Python necesita conocer el número de componentes del vector y el orden de la matriz):

**V0** es una subclase de **Vector** (por tanto hereda los atributos de la clase **Vector**), pero el código inicia (y devuelve) un objeto de su superclase, es decir, inicia y devuelve un **Vector**.

59b *<Definición de vector nulo: V0 59b>*≡

```
class V0(Vector):
    def __init__(self, n, rpr = 'columna'):
```

```
""" Inicializa el vector nulo de n componentes"""
super(self.__class__,self).__init__([0 for i in range(n)], rpr)
```

This code is used in chunk 37.

Defines:

V0, used in chunks 25a and 60a.

Uses Vector 6b.

Y lo mismo hacemos para matrices

60a *<Definición de matriz nula: MO 60a>≡*

```
class MO(Matrix):
    def __init__(self, m, n=None):
        """ Inicializa una matriz nula de orden n """
        n = m if n is None else n

        super(self.__class__,self).__init__([ V0(m) for j in range(n)])
```

This code is used in chunk 37.

Uses Matrix 10 and V0 59b.

También debemos definir la matriz identidad de orden  $n$  (y sus filas y columnas). En los apuntes de clase no solemos indicar expresamente el orden de la matriz identidad (pues normalmente se sobrentiende por el contexto). Pero esta habitual imprecisión no nos la podemos permitir con el ordenador.

### Notación en Mates 2

- $\mathbf{I}$  (de orden  $n$ ) es la matriz tal que  $_{ij}\mathbf{I}_{lj} = \begin{cases} 1 & \text{si } j = i \\ 0 & \text{si } j \neq i \end{cases}$ .

60b *<Definición de la matriz identidad: I 60b>≡*

```
class I(Matrix):
    def __init__(self, n):
        """ Inicializa la matriz identidad de tamaño n """
        super(self.__class__,self).__init__(\
            [(i==j)*1 for i in range(n)] for j in range(n))
```

This code is used in chunk 37.

Uses Matrix 10.

## 3.6 La clase BlockMatrix. Matrices particionadas

Las matrices particionadas no son tan importantes para seguir el curso, aunque si se usan en esta librería. Piense que cuando invierte una matriz o resuelve un sistema de ecuaciones, usa una matriz particionada (con dos bloques:

una matriz arriba, y la matriz identidad con idéntico número de columnas debajo). Como esta librería replica lo que se ve en clase, es necesario definir las matrices particionadas. Si quiere, **puede saltarse esta sección**: el modo de particionar una matriz es sencillo y se puede aprender rápidamente con el siguiente Notebook

#### Tutorial previo en un Jupyter notebook

Este Notebook es un ejemplo sobre el **uso de nuestra librería para Mates 2** en la carpeta “TutorialPython” en <https://mybinder.org/v2/gh/mbujosab/LibreriaDePythonParaMates2/master>

Las matrices por bloques o cajas **A** son tablas de matrices de modo que todas las matrices de una misma fila comparten el mismo número de filas, y todas las matrices de una misma columna comparten el mismo número de columnas. Por ello al “pegar” todas ellas obtenemos una gran matriz.

El argumento de inicialización **sis** es una lista (o tupla) de listas de matrices, cada una de las listas de matrices es una fila de bloques (o submatrices con el mismo número de filas).

El atributo **self.m** contiene el número de filas (de bloques o submatrices) y **self.n** contiene el número de columnas (de bloques o submatrices). Añadimos el atributo **self.ln**, que es una lista con el número de filas que tienen las submatrices de cada fila, y **self.lm** con el número de columnas de las submatrices de cada columna.

```
61a <Iniciación de la clase BlockMatrix 61a>≡
def __init__(self, sis):
    """Inicializa una BlockMatrix con una lista de listas de matrices"""
    self.lista = list(sis)
    self.m      = len(sis)
    self.n      = len(sis[0])
    self.lm     = [fila[0].m for fila in sis]
    self.ln     = [c.n for c in sis[0]]

This code is used in chunk 61b.
Uses BlockMatrix 61b.
```

La clase **BlockMatrix** junto con el listado de sus métodos aparece en el siguiente recuadro:

```
61b <Definición de la clase BlockMatrix 61b>≡
class BlockMatrix:
    <Iniciación de la clase BlockMatrix 61a>
    <Repartición de las columnas de una BlockMatrix 63b>
    <Repartición de las filas de una BlockMatrix 64a>
    <Representación de la clase BlockMatrix 65b>

This code is used in chunk 37.
Defines:
    BlockMatrix, used in chunks 8, 13c, 16, 43a, 49–51, 55b, 56a, and 61–64.
```

### 3.6.1 Particionado de matrices

Vamos a completar las capacidades de los operadores “**i|**” y “**|j**” sobre matrices. Hasta ahora, si los argumentos **i** o **j** eran *enteros* (int), se seleccionaba una fila o una columna respectivamente; y si los argumentos **i** o **j** eran *listas* o

tuplas de índices, se generaba una submatriz con las filas o las columnas indicadas.

Aquí, si los argumentos  $i$  o  $j$  son conjuntos de enteros, asumimos que dicho números enteros indican las filas o columnas por las que se debe particionar una `Matrix` según el siguiente cuadro explicativo:

### Notación en Mates 2

- Si  $p \leq q \in \mathbb{N}$  denotaremos con  $(p : q)$  a la secuencia  $p, p+1, \dots, q$ , (es decir, a la lista ordenada de los números de  $\{k \in \mathbb{N} | p \leq k \leq q\}$ ).
- Si  $i_1, \dots, i_r \in \mathbb{N}$  con  $i_1 < \dots < i_r \leq m$  donde  $m$  es el número de filas de  $\mathbf{A}$ , entonces  $\{i_1, \dots, i_r\} \mathbf{A}$  es la matriz de bloques

$$\{i_1, \dots, i_r\} \mathbf{A} = \left[ \begin{array}{c} (1:i_1) \mathbf{A} \\ (i_1+1:i_2) \mathbf{A} \\ \vdots \\ (i_{r-1}+1:i_r) \mathbf{A} \end{array} \right]$$

- Si  $j_1, \dots, j_s \in \mathbb{N}$  con  $j_1 < \dots < j_s \leq n$  donde  $n$  es el número de columnas de  $\mathbf{A}$ , entonces  $\mathbf{A}_{\{j_1, \dots, j_s\}}$  es la matriz de bloques

$$\mathbf{A}_{\{j_1, \dots, j_s\}} = \left[ \begin{array}{c|c|c|c} \mathbf{A}_{(1:j_1)} & \mathbf{A}_{(j_1+1:j_2)} & \cdots & \mathbf{A}_{(j_{s-1}+1:j_s)} \end{array} \right]$$

Comencemos por la partición de índices a partir de un conjunto y un número (correspondiente al último índice).

62a

```
<Definición del método particion 62a>≡
def particion(s,n):
    """ genera la lista de particionamiento a partir de un conjunto y un número
    >>> particion({1,3,5},7)

    [[1], [2, 3], [4, 5], [6, 7]]
    """
    p = list(s | set([0,n]))
    return [ list(range(p[k]+1,p[k+1]+1)) for k in range(len(p)-1) ]
```

This code is used in chunk 37.

y ahora el método de partición por filas y por columnas resulta inmediato:

62b

```
<Partición de una matriz por filas de bloques 62b>≡
elif isinstance(i,set):
    return BlockMatrix ([ [a|self] for a in particion(i,self.m) ])
```

This code is used in chunk 17.  
Uses `BlockMatrix` 61b.

63a *⟨Partición de una matriz por columnas de bloques 63a⟩*≡

```

elif isinstance(j,set):
    return BlockMatrix ([ [self|a for a in particion(j,self.n)] ])

```

This code is used in chunk 14.  
Uses `BlockMatrix` 61b.

Pero aún nos falta algo:

### Notación en Mates 2

- Si  $i_1, \dots, i_r \in \mathbb{N}$  con  $i_1 < \dots < i_r \leq m$  donde  $m$  es el número de filas de  $\mathbf{A}$  y  $j_1, \dots, j_s \in \mathbb{N}$  con  $j_1 < \dots < j_s \leq n$  donde  $n$  es el número de columnas de  $\mathbf{A}$  entonces

$$\{i_1, \dots, i_r\} | \mathbf{A} | \{j_1, \dots, j_s\} = \left[ \begin{array}{c|c|c|c} (1:i_1) | \mathbf{A} | (1:j_1) & (1:i_1) | \mathbf{A} | (j_1+1:j_2) & \cdots & (1:i_1) | \mathbf{A} | (j_s+1:n) \\ \hline (i_1+1:i_2) | \mathbf{A} | (1:j_1) & (i_1+1:i_2) | \mathbf{A} | (j_1+1:j_2) & \cdots & (i_1+1:i_2) | \mathbf{A} | (j_s+1:n) \\ \hline \vdots & \vdots & \cdots & \vdots \\ \hline (i_k+1:m) | \mathbf{A} | (1:j_1) & (i_k+1:m) | \mathbf{A} | (j_1+1:j_2) & \cdots & (i_k+1:m) | \mathbf{A} | (j_s+1:n) \end{array} \right]$$

es decir, queremos poder particionar una `BlockMatrix`. Los casos interesantes son cuando particionamos por el lado contrario por el que se particionó la matriz de partida, es decir,

$$\{i_1, \dots, i_r\} | \left( \mathbf{A} | \{j_1, \dots, j_s\} \right) \quad \text{y} \quad \left( \{i_1, \dots, i_r\} | \mathbf{A} \right) | \{j_1, \dots, j_s\}$$

que, por supuesto, debe dar el mismo resultado. Para dichos casos, programamos el siguiente código que particiona una `BlockMatrix`. Primero el procedimiento para particionar por columnas cuando hay una única columna de matrices (`self.n == 1`). El caso general se verá más tarde:

63b *⟨Repartición de las columnas de una BlockMatrix 63b⟩*≡

```

def __or__(self,j):
    """ Reparticiona por columna una matriz por cajas """
    if isinstance(j,set):
        if self.n == 1:
            return BlockMatrix([ [ self.lista[i][0]|a \
                                    for a in particion(j,self.lista[0][0].n)] \
                                   for i in range(self.m) ])

```

*⟨Caso general de repartición por columnas 64c⟩*

This code is used in chunk 61b.  
Uses `BlockMatrix` 61b.

y hacemos lo mismo para particionar por filas cuando `self.m == 1` (la matriz por bloques tiene una única fila):



64a  $\langle$ Repartición de las filas de una `BlockMatrix` 64a $\rangle \equiv$

```
def __ror__(self,i):
    """ Reparticiona por filas una matriz por cajas """
    if isinstance(i,set):
        if self.m == 1:
            return BlockMatrix([[ a|self.lista[0][j] \
                                   for j in range(self.n) ] \
                                   for a in particion(i,self.lista[0][0].m)])

     $\langle$ Caso general de repartición por filas 65a $\rangle$ 
```

This code is used in chunk 61b.  
Uses `BlockMatrix` 61b.

Falta implementar el caso general. Debemos decidir el significado de reparticionar una matriz por el mismo lado por el que ya ha sido particionada. Seguiremos un criterio práctico... eliminar el anterior particionado y aplicar el nuevo:

$$\begin{aligned} \{i'_1, \dots, i'_r\} | \left( \{i_1, \dots, i_k\} | \mathbf{A} | \{j_1, \dots, j_s\} \right) &= \{i'_1, \dots, i'_r\} | \mathbf{A} | \{j_1, \dots, j_s\} \\ \left( \{i_1, \dots, i_k\} | \mathbf{A} | \{j_1, \dots, j_s\} \right) | \{j'_1, \dots, j'_r\} &= \{i_1, \dots, i_k\} | \mathbf{A} | \{j'_1, \dots, j'_r\} \end{aligned}$$

Para ello nos viene bien extraer el conjunto selector a partir del resultado:

64b  $\langle$ Definición del procedimiento de generación del conjunto clave para particionar 64b $\rangle \equiv$

```
def key(L):
    """Genera el conjunto clave a partir de una secuencia de tamaños
    número
    >>> key([1,2,1])

    {1, 3, 4}
    """
    return set([ sum(L[0:i]) for i in range(1,len(L)+1) ])
```

This code is used in chunk 37.

Así, los casos generales consisten en reparticionar de nuevo:

64c  $\langle$ Caso general de repartición por columnas 64c $\rangle \equiv$

```
elif self.n > 1:
    return (key(self.lm) | Matrix(self)) | j
```

This code is used in chunk 63b.  
Uses `Matrix` 10.

65a  $\langle$ Caso general de repartición por filas 65a $\rangle \equiv$

```

elif self.m > 1:
    return i | (Matrix(self) | key(self.ln))

```

This code is used in chunk 64a.  
Uses `Matrix` 10.

*Observación 3.* El método `__or__` está definido para conjuntos ... realiza la unión. Por tanto si  $A$  es una matriz, la orden  $\{1,2\} | (\{3\} | A)$  no da igual que  $(\{1,2\} | \{3\}) | A$ . La primera es igual da  $\{1,2\} | A$ , mientras que la segunda da  $\{1,2,3\} | A$ .

### 3.6.2 Representación de la clase `BlockMatrix`

A continuación definimos las reglas de representación para las matrices por bloques. `Matrix` y `BlockMatrix` son objetos distintos. Los bloques se separan con líneas verticales y horizontales; pero si hay un único bloque, no habrá ninguna línea vertical u horizontal por medio de la representación de la `BlockMatrix`. Así, si una matriz por bloques tienen un único bloque, pintaremos una caja alrededor para distinguirla de una matriz ordinaria.

65b  $\langle$ Representación de la clase `BlockMatrix` 65b $\rangle \equiv$

```

def __repr__(self):
    """ Muestra una matriz en su representación Python """
    return 'BlockMatrix(' + repr(self.lista) + ')'

def _repr_html_(self):
    """ Construye la representación para el entorno Jupyter Notebook """
    return html(self.latex())

def latex(self):
    """ Escribe el código de LaTeX """
    if self.m == self.n == 1:
        return \
            '\\begin{array}{|c|}' + \
            '\\hline ' + \
            '\\\\ \\hline '.join( \
                ['\\\\'.join( \
                    ['&'.join( \
                        [latex(self.lista[0][0]) ]) ]) ] + \
            '\\\\ \\hline ' + \
            '\\end{array}'
    else:
        return \
            '\\left[' + \
            '\\begin{array}{'+ '|'.join([n*'c' for n in self.ln]) + '}' + \
            '\\\\ \\hline '.join( \
                ['\\\\'.join( \
                    ['&'.join( \
                        [latex(self.lista[i][j]|k|s) \
                          for j in range(self.n) for k in range(1,self.ln[j]+1) ]) \
                        for i in range(1,self.lm[i]+1) ]) for i in range(self.m) ]) + \
            '\\\\' + \

```

```
'\\end{array}' + \  
'\\right]'
```

This code is used in chunk 61b.

# Appendix A

## Sobre este documento

Con ánimo de que esta documentación sea más didáctica, en el Capítulo 1 muestro las partes más didácticas del código, y relego las otras al Capítulo 3. Así puedo destacar cómo la librería de Python es una implementación literal de las definiciones dadas en mis notas de la asignatura de Mates II. Para lograr presentar el código en un orden distinto del que realmente tiene en la librería uso la herramienta `noweb`. Una breve explicación aparece en la siguiente sección...

### Literate programming con `noweb`

Este documento está escrito usando `noweb`. Es una herramienta que permite escribir a la vez tanto código como su documentación. El código se escribe a trozos o “chunks” como por ejemplo este:

```
67a <Chunk de ejemplo que define la lista a 67a>≡  
    a = ["Matemáticas II es mi asignatura preferida", "Python mola", 1492, "Noweb"]  
This code is used in chunk 67c.
```

y este otro chunk:

```
67b <Segundo chunk de ejemplo que cambia el último elemento de la lista a 67b>≡  
    a[-1] = 10  
This code is used in chunk 67c.
```

Cada chunk recibe un nombre (que yo uso para describir lo que hace el código dentro del chunk). Lo maravilloso de este modo de programar es que dentro de un chunk se pueden insertar otros chunks. Así, podemos programar el siguiente guión de Python (`EjemploLiterateProgramming.py`) que enumera los elementos de una tupla y después hace unas sumas:

```
67c <EjemploLiterateProgramming.py 67c>≡  
    <Chunk de ejemplo que define la lista a 67a>  
    <Segundo chunk de ejemplo que cambia el último elemento de la lista a 67b>  
  
    for indice, item in enumerate(a, 1):  
        print (indice, item)
```

*<Chunk final que indica qué tipo de objeto es `a` y hace unas sumas 70b>*  
 Root chunk (not used in this document).

Este modo de escribir el código permite destacar unas partes y pasar por alto otras. Por ejemplo, *del chunk del recuadro de arriba me interesa que se vea el código del [bucle que permite enumerar los elementos de una lista](#)*. Lo demás es accesorio y se puede consultar en los correspondientes chunks. Como el nombre de dichos chunks es auto-explicativo, mirando el recuadro anterior es fácil hacerse una idea de que hace el programa “EjemploLiterateProgramming.py” en su conjunto.

Fíjese que el número al final del nombre de cada chunk corresponde a la página donde se puede consultar su código. Por ejemplo, el último chunk de este ejemplo se encuentra en la [Página 70](#) de este documento.

El código completo del ejemplo usado para explicar cómo funciona el “Literate Programming” queda así:

```
a = ["Matemáticas II es mi asignatura preferida", "Python mola", 1492, "Noweb"]
a[-1] = 10

for indice, item in enumerate(a, 1):
    print (indice, item)

type(a)
2+2
3+20
```

## A.1 Secciones de código

*<Caso general de repartición por columnas 64c>* [63b](#), [64c](#)  
*<Caso general de repartición por filas 65a>* [64a](#), [65a](#)  
*<Chunk de ejemplo que define la lista `a` 67a>* [67a](#), [67c](#)  
*<Chunk final que indica qué tipo de objeto es `a` y hace unas sumas 70b>* [67c](#), [70b](#)  
*<Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix 31>* [31](#), [33b](#)  
*<Copyright y licencia GPL 70a>* [70a](#)  
*<Creación del atributo `lista` cuando `sis` no es una lista (o tupla) de Vectores 55b>* [9](#), [55b](#)  
*<Creación del atributo `t` cuando se instancia con otra `T` o lista de `Ts` 57b>* [29](#), [57b](#)  
*<Definición de la clase `BlockMatrix` 61b>* [37](#), [61b](#)  
*<Definición de la clase `Matrix` 10>* [10](#), [37](#)  
*<Definición de la clase `T` (Transformación Elemental) 33b>* [33b](#), [37](#)  
*<Definición de la clase `Vector` 6b>* [6b](#), [37](#)  
*<Definición de la igualdad entre Vectores 21b>* [6b](#), [21b](#)  
*<Definición de la igualdad entre dos Matrix 26>* [10](#), [26](#)  
*<Definición de la matriz identidad: `I` 60b>* [37](#), [60b](#)  
*<Definición de matriz nula: `MO` 60a>* [37](#), [60a](#)  
*<Definición de vector nulo: `VO` 59b>* [37](#), [59b](#)  
*<Definición del método `particion` 62a>* [37](#), [62a](#)  
*<Definición del método `PasosYEscritura` 43a>* [42b](#), [43a](#), [44a](#), [44b](#), [45a](#), [45b](#), [46a](#), [46b](#), [47a](#), [47b](#), [48a](#), [48b](#), [49b](#), [50a](#), [50b](#), [51b](#)  
*<Definición del procedimiento de generación del conjunto clave para particionar 64b>* [37](#), [64b](#)  
*<EjemploLiterateProgramming.py 67c>* [67c](#)  
*<Escalonamiento de una matriz mediante eliminación por columnas 40>* [37](#), [40](#), [41a](#), [42a](#)  
*<Escalonamiento de una matriz mediante eliminación por filas 41b>* [37](#), [41b](#)  
*<Estructura de las variantes de aplicación del método de Gauss 42b>* [42b](#)  
*<Inicialización de la clase `BlockMatrix` 61a>* [61a](#), [61b](#)

<Iniciación de la clase **Matrix** 9> [9](#), [10](#)  
 <Iniciación de la clase **T** (Transformación Elemental) 29> [29](#), [33b](#)  
 <Iniciación de la clase **Vector** 6a> [6a](#), [6b](#)  
 <Invirtiendo una matriz 49b> [37](#), [49b](#), [50a](#), [50b](#)  
 <Método **pivote** calcula el índice del primer coef. no nulo de un **Vector** a partir de cierta posición 39> [37](#), [39](#)  
 <Método auxiliar **CreaLista** que devuelve listas de abreviaturas 30b> [29](#), [30b](#), [31](#), [33a](#)  
 <Método auxiliar que calcula la inversa de una Transformación elemental 33a> [32b](#), [33a](#)  
 <Método auxiliar que calcula la inversa de una **Matrix** 49a> [25b](#), [49a](#)  
 <Método **html** general 53a> [37](#), [53a](#)  
 <Método **latex** general 53b> [37](#), [53b](#)  
 <Métodos **html** y **latex** para fracciones 54a> [37](#), [54a](#)  
 <normal 50c> [37](#), [50c](#)  
 <Normalizando la diagonal principal para que sus componentes no nulos sean unos 48b> [37](#), [48b](#)  
 <notacion.py 37> [37](#)  
 <Operador selector por la derecha para la clase **Matrix** 14> [10](#), [14](#)  
 <Operador selector por la derecha para la clase **Vector** 12> [6b](#), [12](#)  
 <Operador selector por la izquierda para la clase **Matrix** 17> [10](#), [17](#)  
 <Operador selector por la izquierda para la clase **Vector** 13b> [6b](#), [13b](#)  
 <Operador transposición para la clase **Matrix** 15b> [10](#), [15b](#)  
 <Operador transposición para la clase **T** 32a> [32a](#), [33b](#)  
 <Partición de una matriz por columnas de bloques 63a> [14](#), [63a](#)  
 <Partición de una matriz por filas de bloques 62b> [17](#), [62b](#)  
 <Potencia de una **Matrix** 25b> [10](#), [25b](#)  
 <Potencia de una **T** 32b> [32b](#), [33b](#)  
 <Producto de un **Vector** por un escalar a su izquierda 19b> [6b](#), [19b](#)  
 <Producto de un **Vector** por un escalar, **Vector**, o **Matrix** a su derecha 21a> [6b](#), [21a](#)  
 <Producto de una **Matrix** por un escalar a su izquierda 23b> [10](#), [23b](#)  
 <Producto de una **Matrix** por un escalar, un vector o una matriz a su derecha 25a> [10](#), [25a](#)  
 <Repartición de las columnas de una **BlockMatrix** 63b> [61b](#), [63b](#)  
 <Repartición de las filas de una **BlockMatrix** 64a> [61b](#), [64a](#)  
 <Representación de la clase **BlockMatrix** 65b> [61b](#), [65b](#)  
 <Representación de la clase **Matrix** 56d> [10](#), [56d](#)  
 <Representación de la clase **T** 59a> [33b](#), [59a](#)  
 <Representación de la clase **Vector** 54b> [6b](#), [54b](#)  
 <Resolviendo un sistema homogéneo 51b> [37](#), [51b](#)  
 <Reverso de un **Vector** 55a> [6b](#), [55a](#)  
 <Reverso de una **Matrix** 57a> [10](#), [57a](#)  
 <Se almacenan los atributos **tex**, **pasos** y **rank**; y se devuelve el resultado 43b> [42b](#), [43b](#), [44a](#), [44b](#), [45a](#), [45b](#), [46a](#), [46b](#), [47a](#), [47b](#), [48a](#)  
 <Segundo chunk de ejemplo que cambia el último elemento de la lista **a** 67b> [67b](#), [67c](#)  
 <sistema 51a> [37](#), [51a](#)  
 <Suma de **Matrix** 22b> [10](#), [22b](#)  
 <Suma de **Vectores** 18b> [6b](#), [18b](#)  
 <Texto de ayuda de la clase **Matrix** 8> [8](#), [10](#)  
 <Texto de ayuda de la clase **T** (Transformación Elemental) 28> [28](#), [33b](#)  
 <Texto de ayuda de la clase **Vector** 4> [4](#), [6b](#)  
 <Texto de ayuda de las transformaciones elementales de las columnas de una **Matrix** 34> [34](#), [35a](#)  
 <Texto de ayuda de las transformaciones elementales de las filas de una **Matrix** 35b> [35b](#), [36](#)  
 <Texto de ayuda para el operador producto por la derecha en la clase **Matrix** 24> [24](#), [25a](#)  
 <Texto de ayuda para el operador producto por la derecha en la clase **Vector** 20> [20](#), [21a](#)  
 <Texto de ayuda para el operador producto por la izquierda en la clase **Matrix** 23a> [23a](#), [23b](#)  
 <Texto de ayuda para el operador producto por la izquierda en la clase **Vector** 19a> [19a](#), [19b](#)  
 <Texto de ayuda para el operador selector por la derecha para la clase **Matrix** 13c> [13c](#), [14](#)  
 <Texto de ayuda para el operador selector por la derecha para la clase **Vector** 11> [11](#), [12](#)  
 <Texto de ayuda para el operador selector por la izquierda para la clase **Matrix** 16> [16](#), [17](#)  
 <Texto de ayuda para el operador selector por la izquierda para la clase **Vector** 13a> [13a](#), [13b](#)  
 <Texto de ayuda para el operador suma en la clase **Matrix** 22a> [22a](#), [22b](#)

<Texto de ayuda para el operador suma en la clase `Vector` 18a> [18a](#), [18b](#)  
 <Texto de ayuda para el operador transposición de la clase `Matrix` 15a> [15a](#), [15b](#)  
 <Texto de ayuda para la composición de Transformaciones Elementales `T` 30c> [30c](#), [31](#)  
 <Transformaciones elementales de las columnas de una `Matrix` 35a> [10](#), [35a](#)  
 <Transformaciones elementales de las filas de una `Matrix` 36> [10](#), [36](#)  
 <Variantes del escalonamiento por eliminación por columnas que guardan los pasos dados 44a> [37](#), [44a](#), [44b](#), [45a](#), [45b](#),  
[46a](#), [46b](#)  
 <Variantes del escalonamiento por eliminación por filas que guardan los pasos dados 47a> [37](#), [47a](#), [47b](#), [48a](#)  
 <Verificación de que al instanciar `Matrix` el argumento `sis` es indexable 56a> [55b](#), [56a](#)  
 <Verificación de que las abreviaturas corresponden a transformaciones elementales 30a> [29](#), [30a](#)  
 <Verificación de que todas las columnas de la matriz tienen la misma longitud 56c> [9](#), [56c](#)  
 <Verificación de que todas las filas de la matriz tendrán la misma longitud 56b> [55b](#), [56b](#)

## Licencia

70a <Copyright y licencia GPL 70a>≡  
 # Copyright (C) 2019 Marcos Bujosa

```
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
```

```
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
```

```
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <https://www.gnu.org/licenses/
```

Root chunk (not used in this document).

## Último chunk del ejemplo de Literate Programming

Este es uno de los trozos de código del ejemplo.

70b <Chunk final que indica qué tipo de objeto es `a` y hace unas sumas 70b>≡

```
type(a)
2+2
3+20
```

This code is used in chunk 67c.