Índice general

I La clase `Sistema` y sus subclases `BlockV`, `Vector`, `BlockM` y `Matrix` 3

1. La clase `Sistema` 5

1.1. Implementación de los sistemas (o listas ordenadas) en la clase `Sistema` 6
1.1.1. Texto de ayuda 6
1.1.2. Método de inicialización 6
1.1.3. Métodos similares a los de una `list` 7
1.1.4. Concatenación de `Sistemas` 8
1.1.5. Sustitución y simplificación de expresiones simbólicas en un `Sistema` 10
1.1.6. Otros métodos de la clase `Sistema` 11
1.1.7. Métodos que devuelven `SubEspacios` 14
1.1.8. Método de resolución de sistema de Ecuaciones lineales 14
1.2. Operaciones algebraicas sobre `Sistemas` 15
1.2.1. Implementación del operador selector por la derecha para la clase `Sistema` . 15
1.2.2. Implementación del operador selector por la izquierda para la clase `Sistema` . 16
1.2.3. Suma y diferencia de `Sistemas` 16
1.2.4. Producto de un `Sistema` por un escalar a su izquierda 18
1.2.5. Producto de un `Sistema` por escalar, `Vector` o `Matrix` a su derecha 19
1.3. Transformaciones elementales de un `Sistema` 21
1.3.1. Texto de ayuda transformaciones elementales por la derecha 21
1.3.2. Implementación de las transformaciones elementales por la derecha 22
1.3.3. Implementación de las transformaciones elementales por la izquierda 22
1.4. Eliminación 22
1.4.1. El método `elim` 23
1.4.2. Variantes de eliminación 25
1.4.3. Formas pre-escalonada, escalonada y escalonada reducida 29
1.4.4. Ejemplos de eliminación 29
1.4.5. Representación de los procesos de eliminación Gaussiana 32
1.4.6. Desarrollos partidos en varias líneas, compactos o abreviados 35
1.5. Representación de la clase `Sistema` 37
1.5.1. Ejemplo de representación de un `Sistema` 39
1.5.2. Ejemplo de representación con un `Sistema` más complejo 40
1.6. La clase `Sistema` completa 41

2. La subclase `BlockV` 42

2.1. Implementación 42
2.1.1. Texto de ayuda 42

2.1.2. Método de inicialización: . . . 43
2.2. Métodos de la subclase `BlockV` . . . 44
2.3. Representación de la subclase `BlockV` . . . 44
2.3.1. Ejemplo de representación de un `BlockV` . . . 45
2.4. La clase `BlockV` completa . . . 45

3. La subclase `Vector` 46
3.1. Implementación de los vectores de $\mathbb{R}^n$ en la subclase `Vector` . . . 46
3.1.1. Texto de ayuda . . . 46
3.1.2. Método de inicialización: . . . 47
3.2. Métodos específicos de la subclase `Vector` . . . 47
3.3. Representación de la clase `Vector` . . . 48
3.4. La clase `Vector` completa . . . 48

4. Las subclases `V0` y `V1` 49

5. La subclase `Polinomio` 50
5.1. Implementación . . . 50
5.2. Operaciones . . . 51
5.3. Representación . . . 52
5.4. La clase `Polinomio` completa . . . 53

6. La subclase `BlockM` 54
6.1. Implementación . . . 54
6.1.1. Texto de ayuda . . . 54
6.1.2. Método de inicialización: . . . 55
6.2. Métodos de la clase `BlockM` . . . 55
6.2.1. Transposición de un BlockM . . . 55
6.2.2. Comprobación de que las filas o columnas son de composición homogenea . . . 56
6.2.3. Apilado de `BlockMs` . . . 56
6.2.4. Operador selector por la izquierda . . . 56
6.3. Otros métodos de la clase `BlockM` . . . 57
6.3.1. Extiende una `BlockM` a lo largo de la diagonal con una lista de `BlockMs` . . . 57
6.4. Transformaciones elementales de las filas de un `BlockM` . . . 57
6.4.1. Texto de ayuda . . . 57
6.4.2. Implementación de las transformaciones elementales de las filas . . . 58
6.5. Representación de la clase BlockM . . . 58
6.6. La clase `BlockM` completa . . . 60

7. La subclase `Matrix` 61
7.1. Implementación de las matrices en la subclase `Matrix` . . . 61
7.1.1. Texto de ayuda . . . 61
7.1.2. Método de inicialización: . . . 62
7.2. Métodos específicos de la subclase `Matrix` . . . 62
7.2.1. Disposición de los elementos de una `Matrix` . . . 62
7.2.2. Creación de un `Vector` a partir de la diagonal de una `Matrix` . . . 63
7.2.3. Traza de una `Matrix` . . . 63
7.2.4. Normalizado de las columnas (o filas) de una matriz . . . 63

7.2.5. Potencias de una `Matrix` cuadrada . . . 63
7.2.6. Determinante mediante la expansión de Laplace . . . 64
7.2.7. Método de Gram-Schmidt . . . 64
7.2.8. Otros métodos específicos de las matrices cuadradas que usan eliminación . . . 64
7.2.9. Diagonalización ortogonal de una matriz simétrica . . . 71
7.3. Representación de la clase `Matrix` . . . 72
7.4. La clase `Matrix` completa . . . 72

8. Las subclases `M0`, `M1` e `I` **73**

9. Las subclases de sistemas simbólicos **74**

10. `Sistemas` obtenidos tras algunos procedimientos específicos de eliminación **76**
10.1. Subclases de `Sistema`: `Elim`, `ElimG`, `ElimGJ`, `ElimGF` y `ElimGJF` . . . 76
10.2. Subclases de `Matrix`: `InvMat`, `InvMatF` e `InvMatFC` . . . 77
10.3. Diagonalización de matrices: resumen de los tres métodos . . . 79
10.3.1. Ejemplos de diagonalización . . . 80
10.4. Subclases de `Matrix`: `DiagonalizaS`, `DiagonalizaO` y `DiagonalizaC` . . . 82

II La clase `T` (transformación elemental) **84**

11. La clase `T` **85**
11.1. Introducción . . . 85
11.2. Resumen de uso de la clase `T` . . . 87
11.2.1. Ejemplos de transformaciones elementales . . . 88
11.3. Implementación de las transformaciones elementales . . . 90
11.3.1. Texto de ayuda . . . 90
11.3.2. Método de inicialización . . . 91
11.4. Composición de Transformaciones o transformación de un `Sistema` por la izquierda . . . 92
11.4.1. Texto de ayuda . . . 92
11.4.2. Implementación . . . 93
11.5. Transposición de transformaciones elementales . . . 93
11.6. Potencias e inversa de transformaciones elementales . . . 94
11.7. Transformaciones elementales "espejo" . . . 95
11.8. Sustitución de variables simbólicas . . . 95
11.9. Métodos similares a los de una `list` . . . 96
11.9.1. T es iterable . . . 96
11.9.2. Igualdad entre transformaciones elementales . . . 96
11.10. Representación de la clase `T` . . . 97
11.10.1. Secuencias de transformaciones . . . 97
11.11. La clase `T` completa . . . 99

III Las clases `SubEspacio` y `EAfin` **100**

12. La clase `SubEspacio` (de $\mathbb{R}^m$) **101**
12.1. Ejemplos de subespacios . . . 102
12.2. Implementación de `SubEspacio` . . . 103

12.3. Métodos de la clase `SubEspacio` . . . 105
12.4. Métodos de representación de la clase `SubEspacio` . . . 107
12.5. La clase `SubEspacio` completa . . . 107

13.La clase `EAfin` (de $\mathbb{R}^m$) **108**
13.1. Ejemplos de espacios afines . . . 108
13.2. Implementación de `EAfin` . . . 110
13.3. Métodos de la clase `EAfin` . . . 111
13.4. Métodos de representación de la clase `EAfin` . . . 112
13.5. La clase `EAfin` completa . . . 113

IV Las clases `Homogenea` y `SEL` **114**

14.Resolución de sistemas de ecuaciones homogéneos. La clase `Homogenea` **115**
14.1. Implementación de la clase `Homogenea` . . . 116
14.2. Métodos de representación de la clase `Homogenea` . . . 116
14.3. Ejemplos de sistemas homogéneos . . . 117

15.Resolución de sistemas de ecuaciones lineales. La clase `SEL` **119**
15.1. Implementación . . . 120
15.1.1. Texto de ayuda . . . 120
15.2. Métodos de representación de la clase `SEL` . . . 121
15.3. Ejemplos de sistemas de ecuaciones . . . 122
15.4. Ejemplo: combinaciones lineales de matrices . . . 124
15.5. Ejemplo: combinaciones lineales de polinomios . . . 125

V La clase `Determinante`. Cálculo del determinante por eliminación **127**
15.5.1. Método de inicialización . . . 128
15.6. Ejemplos de determinantes . . . 129

VI La clase `FuncionLineal` **131**
15.7. Implementación de las funciones afines en la clase `FuncionLineal` . . . 132
15.7.1. Método de inicialización . . . 132
15.8. Métodos la clase `FuncionLineal` . . . 132
15.9. Métodos de representación de la clase `FuncionLineal` . . . 133

VII Librería completa **135**

16.Nombres que exporta el módulo **138**

17.Métodos generales **140**
17.1. Números racionales . . . 140
17.2. Métodos de representación para el entorno Jupyter . . . 141
17.3. Idioma de los textos de salida . . . 142
17.4. Presentación de las fracciones . . . 144

17.4.1. Ejemplos de presentación de las fracciones 145
17.5. Simplificación de expresiones simbólicas 145
17.6. Números 146
17.7. Ristras 147
17.8. Métodos `CreaLista` y `CreaSistema` 147
17.9. Métodos `primer_no_nulo` y `ultimo_no_nulo` de un Sistema 147
17.10. Otros métodos auxilares 148
17.11. Cribado de secuencias de transformaciones 148

Notación Asociativa para un Curso de Álgebra Lineal (NAcAL)

Versión: 0.5.8

Marcos Bujosa

27 de noviembre de 2026

Versión: 0.5.8
https://github.com/mbujosab/nacallib

Introducción

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).
>
> Consecuentemente este ejercicio impone una disciplina a la que en general el alumno no está acostumbrado: *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 no hemos escrito las ordenes de manera correcta (lo que supone que también hemos escrito incorrectamente las expresiones matemáticas).

Con esta idea en mente:

1. La notación del Curso de Álgebra Lineal pretende ser operativa; es decir, su uso debe ser directamente traducible a operaciones a realizar por un ordenador. Para lograr una mayor simplificación, la notación explota de manera intensiva la asociatividad.
2. Muchas de las demostraciones del libro son algorítmicas. En particular aquellas las relacionadas con la eliminación. De esta manera las demostraciones describen literalmente la programación de los correspondientes algoritmos.

Un módulo específico para el curso de Álgebra Lineal

Aunque Python dispone de módulos para operar con vectores y matrices, he decidido escribir mi propio módulo. De este modo logro que la notación del libro y las expresiones empleadas con este módulo para Python se parezcan lo más posible. Este documento describe tanto el uso del módulo NAcAL como su código.

Tenga en cuenta que esto no es un tutorial de Python.

Mi labor es enseñar Álgebra Lineal (no Python). Afortunadamente usted dispone de muchos cursos y material en la web para aprender Python. No obstante, he escrito unos Notebooks de Jupyter que ofrecen unas breves nociones de programación en Python (aunque muy incompletas).

Antes de seguir, repase el Notebook **"Listas y tuplas"** en la carpeta "TutorialPython"

Y recuerde que

¡hacer matemáticas y programar son prácticamente la misma cosa!

Cómo leer el código de este documento

Este documento contiene todo el código del módulo: los ficheros de Python se generan a partir de él. El código no aparece de una vez, sino repartido en trozos, y cada trozo se muestra junto al texto

que lo explica. Dentro de un trozo puede aparecer una línea como esta:

```
<<Reducción con un pivote>>
```

Esa línea no es código de Python. Es el nombre de otro trozo, e indica que su código va insertado en ese lugar. El nombre describe lo que hace el trozo, así que basta leerlo para seguir el hilo; el código correspondiente está en el recuadro que lleva ese mismo nombre escrito en su línea superior, normalmente poco después. Así cada recuadro enseña solo lo que interesa en ese momento, y los detalles se consultan donde se explican.

Parte I

La clase `Sistema` y sus subclases `BlockV`, `Vector`, `BlockM` y `Matrix`

El primer capítulo presenta una *clase* para las *listas ordenadas de objetos* denominada `Sistema`.

Los capítulos siguientes definen ciertas subclases (y subclases de subclases) de la clase `Sistema`:

- `BlockV` es una subclase que puede mostrarse verticalmente en su representación LaTeX.
- `Vector` es un `BlockV` formado exclusivamente por números (implementa los vectores de $\mathbb{R}^n$).
  - `V0` y `V1` son `Vectores` que de ceros y de unos respectivamente.
- `BlockM` es una subclase de `Sistema` cuyos elementos son `BlockVs` con el mismo número de elementos. Permite trabajar con arreglos rectangulares de objetos. Por ser arreglos rectangulares podremos definir su transposición y con ello generar `BlockM` cuyas componentes están formadas por los `BlockVs` que contienen las primeras componentes, luego las segundas, etc.
- `Matrix` es un `BlockM` formado exclusivamente por `Vectores` (matrices de $\mathbb{R}^{n\times m}$) .
  - `M0` y `M1` son `Matrices` de ceros y de unos respectivamente. La subclase `I` permite definir fácilmente matrices identidad.
- Por último se definen ciertas subclases de `Sistemas` (en algunos casos son específicamente `Matrices`) que resultan tras la aplicación de algún proceso específico de eliminación. Estos `Sistemas` poseen algunos atributos adicionales tales como los `pasos` de eliminación que se han dado hasta llegar a su obtención.

Capítulo 1

La clase `Sistema`

En el libro del curso se dice que

> Un *sistema* es una lista ordenada de objetos.

Aunque Python posee de manera nativa las `list` (listas), `NAcAL` define una clase específica denominada `Sistema`. De esta manera `NAcAL` puede implementar tanto los métodos específicos descritos en el curso como la notación empleada en el libro, donde los sistemas genéricos se muestran entre *corchetes* y con los elementos de la lista *seguidos de* " `;`".[1] No solo eso, cada uno de los elementos de un `Sistema` se muestra con su propia representación `latex` (si la tiene). Por ejemplo, el `Sistema`

```
Sistema([ Vector([1,2,3]), I(2), T({1,2}) ])
```

contiene un vector de $\mathbb{R}^3$, la matriz identidad 2 por 2 y una transformación intercambio entre las componentes 1 y 2 de un sistema. En un Notebook de Jupyter veremos dicho `Sistema` así

$$\left[\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}; \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \underset{[\mathbf{1} \rightleftharpoons \mathbf{2}]}{\boldsymbol{\tau}}; \right]$$

(es decir, con la misma notación descrita en el curso de Álgebra Lineal).

Aunque los `Sistemas` y las `listas` de Python se diferencian en su representación y en algunos de sus métodos, otros métodos son comunes. De hecho, en algunos aspectos un `Sistema` se comporta exactamente igual que una `list` (por ejemplo, un `Sistema` también es iterable). La mayoría de métodos definidos en `NAcAL` son específicos de los `Sistemas`, y otros métodos se han modificado para reservar los símbolos de algunos *métodos mágicos* de Python para ciertas operaciones algebraicas. Por ejemplo, las `listas` de Python se concatenan con "+", pero los `Sistemas` se concatenan con el método `concatena()`. Así `NAcAL` reserva el símbolo "+" para sumar `Sistemas` componente a componente como se hace en Álgebra Lineal. Con ello se logra que lo que veamos y escribamos en un Notebook de Jupyter sea lo más parecido posible a lo que vemos y escribimos en un curso de Álgebra Lineal.

[1] `NAcAL` también replica la notación de otros sistemas particulares tales como los vectores de $\mathbb{R}^n$ y las matrices.

1.1. Implementación de los sistemas (o listas ordenadas) en la clase `Sistema`

1.1.1. Texto de ayuda

El texto de ayuda de la clase `Sistema` es auto-explicativo y Python lo muestra al teclear `help(Sistema)`:

Texto de ayuda de la clase Sistema

```
"""Clase para listas ordenadas con reprentación latex

Un Sistema es una lista ordenada de objetos. Los Sistemas se instancian
con una lista, tupla u otro Sistema.

Parámetros:
    arg (list, tuple, Sistema): lista, tupla o Sistema de objetos.

Atributos:
    lista       (list): lista de objetos.
    n            (int): número de elementos del sistema.
    corteSistema (set): índices de los elementos tras los que mostrar
                        una separación en la representación LaTeX
Ejemplos:
>>> # Crea un nuevo Sistema a partir de una lista, tupla o Sistema
>>> Sistema( [ 10, 'hola', T({1,2}) ]  )           # con lista
Sistema([10; 'hola'; T([{1, 2}]);])
>>> Sistema( ( 10, 'hola', T({1,2}) )  )           # con tupla
Sistema([10; 'hola'; T([{1, 2}]);])
>>> Sistema( Sistema( [ 10, 'hola', T({1,2}) ] ) ) # con Sistema
Sistema([10; 'hola'; T([{1, 2}]);])

"""
```

1.1.2. Método de inicialización

La clase se inicia con el método: `def__init__(self,...)`.

- Un `Sistema` se instancia con el argumento `arg` (que es una *ristra* de objetos —Sección 17.7).
- Añadimos un breve texto de ayuda que Python mostrará con: `help Sistema.__init__`
- Cuando `arg` es una lista, tupla o `Sistema`, el atributo `self.lista` guarda una lista en forma de `list` (lista de Python) con los elementos contenidos en `arg`.
- Cuando `arg` no es una lista, tupla, o `Sistema` se devuelve un mensaje de error.
- El atributo `.n` contiene el número de elementos del `Sistema`.
- El atributo `.corteSistema` tiene que ver con una modificación de la Representación de la clase `Sistema` para *visualmente* cortar el `Sistema` en sublistas con unas líneas verticales en las posiciones indicadas en este atributo (Véase la Sección 1.5).

Inicialización de la clase Sistema

```
def __init__(self, arg):
    """Inicializa un Sistema con una lista, tupla o Sistema"""
    if es_ristra(arg):
        self.lista = [racional(elemento) for elemento in arg]
    else:
        raise ValueError('El argumento debe ser una lista, tupla, o Sistema.')

    self.n = len(self)
```

```
    self.corteSistema = set()
```

En las siguientes secciones se definen los métodos de la clase `Sistema`, empezando por aquellos que fuerzan a que un `Sistema` se comporte como una `list` de Python en ciertos aspectos.

1.1.3. Métodos similares a los de una `list`

Los siguientes métodos replican en la clase `Sistema` algunos aspectos de la clase `list` de Python.

Para que un `Sistema` sea iterable necesitamos los métodos "mágicos" `__getitem__` (para seleccionar componentes) y `__setitem__` (para modificar componentes). Así, con `A[0]` obtendremos el primer elemento del sistema `A` y con `A[2] = 0` sustituiremos su tercer elemento por cero.[2]

```
def __getitem__(self, i):
    """Devuelve el i-ésimo coeficiente del Sistema"""
    return self.lista[i]

def __setitem__(self, i, nuevo_valor):
    """Modifica el i-ésimo coeficiente del Sistema"""
    self.lista[i] = nuevo_valor
```

Con `len(A)` contamos el número de elementos del `Sistema`.

```
def __len__(self):
    """Número de elementos del Sistema """
    return len(self.lista)
```

Con `copy` podemos hacer una copia, por ejemplo `B = A.copy()` hace una copia del sistema `A` generando un nuevo sistema del mismo *tipo* (de la misma clase o subclase) y con una `lista` de elementos igual que la de `A`.

```
def copy(self):
    """ Genera un Sistema copiando la lista de otro """
    return type(self)(self.lista)
```

Así, si `A = Vector([1,2,3])` entonces `B = A.copy()` es un nuevo `Vector` con idéntica `lista` de elementos que `A`. Sin embargo, `B = Sistema(A)` devuelve un el sistema genérico `Sistema([1,2,3])`.

Además, al disponer del método `__getitem__`, también podemos generar un sistema con la misma `lista` de elementos que otro del sistema haciendo *"slicing"*: `B = Sistema(A[:])`, pues `A[:]` nos devuelve la `lista` de `A`.

Más adelante se implementa otro Método para copiar un Sistema y sus atributos (`fullcopy()`) que no aparece aquí porque no es un método de las `lists` de Python.

[2] ¡Recuerde que en Python los índices comienzan en `0`! Aunque `NAcAL` incorpora este "pythonesco" modo de indexar `Sistemas`, posteriormente (Sección 1.2.1) se añade otro método adicional que implementa el operador selector " `|` " y que, tal como se hace en el libro, emplea el `1` como primer índice.

Comprobamos si `A` y `B` son iguales con `A==B` y si son distintos con `A!=B`.

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

def __ne__(self, other):
    """Indica si es cierto que dos Sistemas son distintos"""
    return not (self == other)
```

Con `A.reverse()` invertimos el orden de los elementos de `A` (por tanto cambiamos el sistema `A`). Con `reversed(A)` obtenemos un nuevo `Sistema` con los elementos de `A` en el orden inverso a como aparecen en `A`.

```
def reverse(self):
    """Da la vuelta al orden de la lista del sistema"""
    self.corteSistema = {len(self)-i for i in self.corteSistema}
    self.lista.reverse()

def __reversed__(self):
    """Reversed(S) devuelve una copia de S con la lista en orden inverso"""
    copia = self.fullcopy()
    copia.corteSistema = {len(self)-i for i in self.corteSistema}
    copia.lista = list(reversed(self.lista))
    return copia
```

1.1.4. Concatenación de `Sistemas`

Concatenamos dos `Sistemas` con el método `concatena()`.

(*Primero definimos un método auxiliar que se usa cuando* `divisionesVisuales` *es* `True`).

Cuando intentamos concatenar un `Sistema` con algo que no lo es obtenemos un mensaje de error.

Cuando el `Sistema` no es vacío el procedimiento arranca con una copia completa de dicho sistema (es decir, una copia que también incluye valor del atributo `.corteSistema`). Pero si el `Sistema` es vacío obtenemos una copia completa del segundo sistema (y con dicha copia el método termina).

`A.concatena(B)` es un nuevo `Sistema` cuya `lista` es la concatenación de la `lista` del sistema `A` seguida de la `lista` del sistema `B`; por tanto, el número `n` de elementos del `Sistema` resultante es la suma del número de elementos de `A` más el de `B`.

Si queremos visualizar cortes o divisiones que separen sublistas del `Sistema`, a las divisiones visuales pre-existentes en ambos `Sistemas`, se añade otra en la posición que separa los sistemas originales. El método `nuevoConjuntoMarcas()` devuelve el conjunto de índices donde representar dichos cortes.

Si los elementos de la lista resultante no tienen la misma longitud (por ejemplo, si concatenamos dos matrices con distinto número de filas), entonces el sistema no puede ser representado como un arreglo rectangular de objetos porque no tiene dicha estructura. En tal caso el tipo de objeto resultante será un `Sistema` genérico.

Método de la clase Sistema para concatenar dos sistemas

```
def concatena(self, other, marcasVisuales=False):
    """Concatena dos Sistemas"""
```

```
    def nuevoConjuntoMarcas(Sistema_A, Sistema_B):
        return Sistema_A.corteSistema.union(
            {len(Sistema_A)},
            {len(Sistema_A)+indice for indice in Sistema_B.corteSistema}
        )

    if not isinstance(other, Sistema):
        raise ValueError('Un Sistema solo se puede concatenar a otro Sistema')

    if self:
        sistemaAmpliado = self.fullcopy()
    else:
        return other.fullcopy()

    sistemaAmpliado.lista = self.lista + other.lista
    sistemaAmpliado.n = len(self) + len(other)

    if marcasVisuales:
        sistemaAmpliado.corteSistema = nuevoConjuntoMarcas(self, other)


    if self.es_arreglo_rectangular():
        return sistemaAmpliado
    elif isinstance(self, Vector) and es_ristra_de_numeros(sistemaAmpliado.lista):
        return sistemaAmpliado     # al concatenar números a un Vector se obtiene otro Vector
    else:
        return Sistema(sistemaAmpliado)
```

El método `junta()` crea el `Sistema` resultante de concatenar una lista de sistemas. Por ejemplo `A.junta([B,C,D])` devuelve el sistema cuya `lista` es la concatenación de las `listas` de los sistemas `A`, `B`, `C` y `D`. Si `marcas` es `True` se muestran los cortes entre los distintos subsistemas.

Método que junta una lista de Sistemas en un único Sistema

```
def junta(self, lista, marcas=False):
    """Junta una lista o tupla de Sistemas en uno solo concatenando las
    correspondientes listas de los distintos Sistemas

    """
    reune = lambda lista,marcas: lista[0] if len(lista)==1 else lista[0].concatena(reune(lista[1:],marcas),
    ↪  marcas)
    return reune([self] + [sistema for sistema in lista], marcas)
```

Método que amplía la lista de un Sistema con nuevos elementos

```
def amplia(self, args, marcas=False):
    """Añade más elementos al final de la lista de un Sistema"""
    A = self.fullcopy()
    return A.concatena(Sistema(CreaLista(args)), marcas)
```

Método que vectoriza un Sistema

```
def vec(self):

    def vectoriza(sist):
        lista = []
        for item in sist:
            if isinstance(item, Sistema):
                lista += vectoriza(item)
            else:
                lista += [item]
        return lista
```

```
    return BlockV(vectoriza(self))
```

1.1.5. Sustitución y simplificación de expresiones simbólicas en un `Sistema`

Sustitución de variables simbólicas.

Sustitución de variables simbólicas

```
def subs(self, reglasDeSustitucion=[]):
    """ Sustitución de variables simbólicas """

    def sustitucion(elemento, regla_de_sustitucion):
        if es_numero(elemento):
            return racional(sympy.S(elemento).subs(CreaLista(regla_de_sustitucion)))
        else:
            return elemento.subs(CreaLista(regla_de_sustitucion))

    reglas = CreaLista(reglasDeSustitucion)
    NuevoSistema = self.fullcopy()
    NuevoSistema.lista = [sustitucion(elemento, reglasDeSustitucion) for elemento in NuevoSistema]
    return NuevoSistema
```

El argumento es una lista de reglas de sustitución formadas por pares `(símbolo, valor)`; por ejemplo `[(a,2), (b,0), (c,a)]`.

```
A = Sistema([ a, b, c ])
B = A.subs([(a,222), (b,sympy.sqrt(5)), (c,a)])
```

El método devuelve un nuevo `Sistema`; el original no cambia.

$$\left[\ 222;\quad \sqrt{5};\quad a;\ \right]$$

Cuando hay una única regla de sustitución, basta escribir como argumento el correspondiente par. Por ejemplo: `A.subs( (a,0) )`.

Métodos para simpificar expresiones simbólicas

Simplificación de las expresiones simbólicas contenidas en la `lista` de un `Sistema`

Simplificación de expresiones simbólicas en la lista de un Sistema

```
def simplify(self):
    """Simplifica las expresiones simbólicas (modifica el Sistema y lo devuelve)"""
    self.lista = [simplify(elemento) for elemento in self.lista]
    return self

def factor(self):
    """Factoriza las expresiones simbólicas (modifica el Sistema y lo devuelve)"""
    self.lista = [factor(elemento) for elemento in self.lista]
    return self

def expand(self):
    """Desarrolla las expresiones simbólicas (modifica el Sistema y lo devuelve)"""
    self.lista = [expand(elemento) for elemento in self.lista]
    return self
```

```
x, y, z = sympy.symbols('x y z')
A = Sistema([(x**3 + x**2 - x - 1)/(x**2 + 2*x + 1), x**2*z + 4*x*y*z + 4*y**2*z])
```

$$\left[\ \frac{x^3+x^2-x-1}{x^2+2x+1};\quad x^2z+4xyz+4y^2z;\ \right]$$

```
A.simplify()
```

$$\left[\ x-1;\quad z\left(x^2+4xy+4y^2\right);\ \right]$$

1.1.6. Otros métodos de la clase `Sistema`

Método para copiar un Sistema con todos sus atributos

```
def fullcopy(self):
    """ Copia la lista de otro Sistema y sus atributos"""
    new_instance = self.copy()
    new_instance.__dict__.update(self.__dict__)
    return new_instance
```

Método para recuperar el `Sistema` genérico de cualquier subclase de `Sistema`

Con el método `sis` obtendremos el `Sistema` correspondiente a cualquier `Sistema` o subclase de `Sistema`. Así, si `A` es una `Matrix`, con `A.sis()` obtenemos el `Sistema` de `Vectores` (columnas) asociado.

```
def sis(self):
    """Devuelve el Sistema en su forma genérica"""
    return Sistema(self.lista)
```

Comprobación de que un `Sistema` es nulo

```
def es_nulo(self, sust=[]):
    """Indica si es cierto que el Sistema es nulo"""
    def nulo(elemento):
        if isinstance(elemento, Sistema):
            return elemento.es_nulo()
        elif es_numero(elemento):
            return es_cero(elemento)
        else:
            try:
                return elemento == 0*elemento
            except Exception:
                return False

    return all(nulo(elemento) for elemento in self.subs(sust))

def no_es_nulo(self, sust=[]):
    """Indica si es cierto que el Sistema no es nulo"""
```

```
    return not self.es_nulo(sust)
```

Comprobación de que un `Sistema` tiene estructura de arreglo rectangular

Un `Sistema` es un arreglo rectangular de objetos si es un `Sistema` de `Sistemas` con idéntica longitud (como en el caso de una matriz, pues todas sus columnas tienen el mismo número de elementos).

```
def es_arreglo_rectangular(self):
    """Indica si el Sistema tiene estructura de arreglo rectangular"""

    def solo_contiene_sistemas(sis):
        return all([isinstance(elemento, Sistema) for elemento in sis])

    def elementos_con_la_misma_logitud(sis):
        primerElemento = sis|1
        return all([len(primerElemento)==len(elemento) for elemento in sis])

    if solo_contiene_sistemas(self) and elementos_con_la_misma_logitud(self):
        return True
    else:
        return False

def no_es_arreglo_rectangular(self):
    """Indica si el Sistema no tiene estructura de arreglo rectangular"""
    return not self.es_arreglo_rectangular()
```

Comprobación de que todos los elementos de un `Sistema` son del mismo tipo

---

```
def es_de_composicion_uniforme(self):
   """Indica si es cierto que todos los elementos son del mismo tipo"""
   if all([es_numero(c) for c in self]):
      return True
   else:
      return all(type(elemento)==type(self|1) for elemento in self)
```

Comprobación de que todos los elementos de un `Sistema` son del mismo tipo y tienen la misma longitud

---

```
def es_de_composicion_y_longitud_uniforme(self):
   """Indica si es cierto que todos los elementos son del mismo tipo y
   longitud

   """
   if self.es_de_composicion_uniforme() and es_numero(self|1):
      return True
   elif all(isinstance(elemento, Polinomio) for elemento in self):
      return True      # polinomios de distinto grado se pueden sumar
   elif self.es_de_composicion_uniforme() and not es_numero(self|1):
      return all(len(elemento)==len(self|1) for elemento in self)
   else:
      return False
```

Búsqueda del primer, o del último, elemento no nulo del `Sistema`

```
def primer_no_nulo(self, reglasDeSustitucion=[]):
    """Devuelve una lista con la posición del primer no nulo o vacía si
    todos los elementos son nulos

    """
    sistema = self.subs(reglasDeSustitucion)
    return next( ([indice] for indice, elemento in enumerate(sistema, 1) if
    ↪  CreaSistema(elemento).no_es_nulo()), [])

def ultimo_no_nulo(self, reglasDeSustitucion=[]):
    """Devuelve una lista con la posición del último no nulo o vacía si
    todos los elementos son nulos

    """
    sistema = reversed(self.copy()).subs(reglasDeSustitucion)
    return next( ([len(self)-indice] for indice,elemento in enumerate(sistema) if
    ↪  CreaSistema(elemento).no_es_nulo()), [])

elementoPivote     = lambda self: self.extractor(self.primer_no_nulo()) if self.primer_no_nulo() else False

elementoAntiPivote = lambda self: self.extractor(self.ultimo_no_nulo()) if self.ultimo_no_nulo() else False
```

Extractor de un elemento dada una lista de indices (coordenadas)

```
def extractor(self, listaDeIndices=[]):
    """Selección consecutiva por la derecha del sistema A empleando la
    lista de enteros de c. Ej.: si c = [5,1,2] devuelve A|5|1|2

    """
    objeto = self
    for indice in listaDeIndices:
        objeto = objeto|indice
    return objeto if listaDeIndices else []
```

Reordena un `Sistema` para generar un `BlockM`

```
def reshape(self, orden=[]):
    "Reordena los elementos de un Sistema para generar un BlockM"
    if not orden or isinstance(orden, int):
        return self
    elif orden[0]*orden[1] == self.n:
        return ~BlockM(list(zip(*(iter(self.lista),)*orden[0])))
    else:
        raise ValueError('Orden incompatible con el número de elementos')
        return None
```

Estructura de componentes de un `Sistema`

```
def estructura(self):
    """Devuelve la estructura de loscomponentes de un Sistema"""
    if isinstance(self, Vector):
        return [(type(self), (self.n,))]
```

```
    elif isinstance(self, Matrix):
        return [(type(self), (self.n, self.m))]
    else:
        return [type(self), [estructura(item) for item in self]]
```

1.1.7. Métodos que devuelven `SubEspacios`

Espacio generado los los elementos del `Sistema`

```
def span(self, sust=[], Rn=[]):
    return SubEspacio(self.sis(), sust, Rn)
```

Espacio Nulo de un `Sistema` de composición y longitud uniforme

```
def espacio_nulo(self, sust=[], Rn=[]):
    if self: Rn = self.n
    K = self.elim(0, False, sust)
    E = I(self.n) & T(K.pasos[1])
    lista = [v for j,v in enumerate(E,1) if (K|j).es_nulo()]
    return SubEspacio(Sistema(lista)) if lista else SubEspacio(Sistema([]), Rn=Rn)
```

1.1.8. Método de resolución de sistema de Ecuaciones lineales

---

```
def sel(self, v, rep=False, sust=[]):
    """Devuelve el conjunto solución con las soluciones x de sistema*x=v

    """
    A = self.copy().amplia(-v,1)
    operaciones = A.elim(0,False,sust).pasos[1]
    testigo     = 0| (I(A.n) & T(operaciones)) |0
    Normaliza   = T([]) if testigo==1 else T([[ fracc(1,testigo), A.n ]])
    pasos       = operaciones+[Normaliza] if Normaliza else operaciones
    K           = A & T(pasos)

    if rep:
        try:
            MA = Matrix(self.copy().amplia(-v)).subs(sust).csis({self.n}).apila(I(self.n+1),1)
            MA.corteElementos.update({self.n+A.m})
        except Exception:
            MA = BlockM([self.copy().amplia(-v)]).subs(sust).csis({self.n}).apila(I(self.n+1),1)
            MA.corteElementos.update({self.n+A.m})

        dispElim(MA, [[],pasos], [], sust)

    if (K|0).no_es_nulo():
        return Sistema([])
    else:
        solP = factor(I(self.n).amplia(V0(self.n)) & T(pasos))|0
        return EAfin(self.espacio_nulo().sgen, solP, 1)
```

1.2. Operaciones algebraicas sobre `Sistemas`

1.2.1. Implementación del operador selector por la derecha para la clase `Sistema`

Esta sección muestra la implementación del operador selector tal como se describe en el curso, es decir, la selección de elementos de un `Sistema` con el operador `|` actuando por la derecha. El siguiente texto de ayuda es auto-explicativo y Python lo muestra al teclear `help(Sistema.__or__)`.

Texto de ayuda para el operador selector por la derecha para la clase Sistema

```
"""Extrae el j-ésimo componente del Sistema; o crea un Sistema con la
tupla de elementos indicados (los índices comienzan por el número 1)

Parámetros:
    j (int, list, tuple, slice): Índice (o lista de índices) del
          elementos (o elementos) a seleccionar

Resultado:
          ?: Si j es int, devuelve el elemento j-ésimo del Sistema.
    Sistema: Si j es list, tuple o slice devuelve el Sistema formado por
          los elementos indicados en la lista, tupla o slice de índices.

Ejemplos:
>>> # Extrae el j-ésimo elemento del Sistema
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | 2
Vector([0, 2])

>>> # Sistema formado por los elementos indicados en la lista (o tupla)
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | [2,1]
Sistema([Vector([0, 2]); Vector([1, 0]);])
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | (2,1)
Sistema([Vector([0, 2]); Vector([1, 0]);])

>>> # Sistema formado por los elementos indicados en el slice
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) | slice(1,3,2)
Sistema([Vector([1, 0]); Vector([3, 0]);])

"""
```

Cuando el argumento `j` es un número entero (`int`), se selecciona el `j`-ésimo elemento del sistema (recuerde que en Python los índices de objetos iterables comienzan en cero; consecuentemente, al seleccionar el `j`-ésimo elemento del sistema `A` con el operador selector (`A|j`), lo que realmente estamos ejecutando es la operación `A.lista[j-1]`).

Se emplea el método (`self|indice`) (siendo `indice` un `int`) para definir el operador selector cuando `j` es una lista o tupla de índices y generar así un sistema con las componentes indicadas. El sistema obtenido será del mismo tipo que `self`, es decir, o un `Sistema` genérico, o un `BlockV`, o un `Vector`, o una `BlockM`, o una `Matrix` dependiendo de a qué objeto se aplica el selector.

Cuando el argumento `j` es de tipo `slice(start,stop,step)`, se crea un `Sistema` con la selección de ciertos componentes; comenzando por aquél cuyo índice es `start`, y seleccionando de `step` en `step` componentes hasta llegar al de índice `stop`. Dicho sistema será del mismo tipo que `self`. Si el primer argumento de `slice` es `None` se seleccionan los componentes empezando por el primero. Si el segundo argumento de `slice` es `None` se recorren todos los índices hasta llegar al último componente. Si se omite el tercer argumento de `slice` (o si el tercer argumento es `None`) entonces `step` es igual a uno. Así, `slice(None,None)` selecciona todos los componentes; `slice(2,None,2)` selecciona los componentes pares hasta el final; y `slice(4,11,3)` selecciona un componente de

cada tres comenzando por el cuarto y hasta llegar al undécimo (es decir, los índices 4, 7 y 10).

Operador selector por la derecha para la clase Sistema

```
def __or__(self,j):
    <<Texto de ayuda para el operador selector por la derecha para la clase Sistema>>
    if isinstance(j, int):
        return self[j-1]

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

    elif isinstance(j, slice):
        start = None if j.start is None else j.start-1
        stop  = None if j.stop  is None else (j.stop if j.stop>0 else j.stop-1)
        step  = j.step  or 1
        return type(self) (self[slice(start,stop,step)])
```

El operador selector por la derecha funciona de la misma manera tanto para la clase `Sistema` como para cualquiera de sus subclases.

1.2.2. Implementación del operador selector por la izquierda para la clase `Sistema`

En el curso de Álgebra Lineal admitimos la selección de elementos por la izquierda, ${}_{i|}\boldsymbol{v} = \boldsymbol{v}_{|i}$.

La implementación de esta operación es inmediata... si el selector por la izquierda hace lo mismo que el selector por la derecha, basta con llamar al selector por la derecha: `self|i`.

Operador selector por la izquierda para la clase Sistema

```
def __ror__(self,i):
    """Hace exactamente lo mismo que el método __or__ por la derecha."""
    return self | i
```

(*Tenga en cuenta que este método cambia en las subclases* `BlockM` *y* `Matrix`*, pues lo usaremos para seleccionar las filas de dichos arreglos rectangulares de objetos.*)

1.2.3. Suma y diferencia de `Sistemas`

Con la definición de la clase `Sistema` y el operador selector `|` por la derecha, ya podemos definir las operaciones de suma de dos sistemas y de producto de un sistema por un escalar. Fíjese que las definiciones de las operaciones en Python (usando el operador `|`) son idénticas a las empleadas en el curso, donde hemos definido la suma de dos vectores de $\mathbb{R}^n$ como el vector tal que

$$\boxed{(\boldsymbol{a}+\boldsymbol{b})_{|i} = \boldsymbol{a}_{|i} + \boldsymbol{b}_{|i}} \quad \text{para} \quad i = 1:n$$

y la suma de matrices como la matriz tal que

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

Ambas son casos particulares de sumas elemento a elemento entre dos sistemas de n elementos:

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

Usando el operador selector podemos "literalmente" transcribir esta definición

```
Sistema ([ (self|i) + (other|i) for i in range(1,len(self)+1) ])
```

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

Hay que tener en cuenta que cuando el `Sistema` es un `Vector` el resultado es un `Vector` y cuando el `Sistema` es una `Matrix` el resultado es una `Matrix`. Es decir, el código debe devolver un objeto del mismo tipo que `self`. Esto lo logramos sustituyendo `Sistema` por `type(self)` en el código anterior. Así, la implementación final es:

```
type(self) ([ (self|i) + (other|i) for i in range(1,len(self)+1) ])
```

Por último, nótese que para que la implementación funcione es necesario que los elementos $A_{|i}$ y $B_{|i}$ sean sumables, es decir, es necesario que la operación

```
(self|i) + (other|i)
```

esté definida para cada `i`. (De manera análoga definimos diferencia entre `Sistemas`).

Python muestra el texto de ayuda para la suma tecleando `help(Sistema.__add__)`.

Texto de ayuda para el operador resta en la clase Sistema

```
"""Devuelve el Sistema resultante de sumar dos Sistemas

Parámetros:
    other (Sistema): Otro sistema del mismo tipo y misma longitud

Ejemplos:
>>> Sistema([10, 20, 30]) + Sistema([-1, 1, 1])

Sistema([9, 21, 31])
>>> Vector([10, 20, 30]) + Vector([-1, 1, 1])

Vector([9, 21, 31])
>>> Matrix([[1,5],[5,1]]) + Matrix([[1,0],[0,1]])

Matrix([Vector([2, 5]); Vector([5, 2])]) """
```

Python muestra el texto de ayuda para la diferencia tecleando `help(Sistema.__sub__)`.

Texto de ayuda para el operador diferencia en la clase Sistema

```
"""Devuelve el Sistema resultante de restar dos Sistemas

Parámetros:
    other (Sistema): Otro sistema del mismo tipo y misma longitud

Ejemplos:
>>> Sistema([10, 20, 30]) - Sistema([1, 1, -1])
Sistema([9; 19; 31;])
>>> Vector([10, 20, 30]) - Vector([1, 1, -1])
Vector([9, 19, 31])
>>> Matrix([[1,5],[5,1]]) - Matrix([[1,0],[0,1]])
Matrix([Vector([0, 5]), Vector([5, 0])])
"""
```

Suma y diferencia de Sistemas

```
def __add__(self, other):
    <<Texto de ayuda para el operador suma en la clase Sistema>>
    if not type(self)==type(other) or not len(self)==len(other):
        raise ValueError ('Solo se suman Sistemas del mismo tipo y misma longitud')
    suma = self.fullcopy()
```

```
    suma.lista = [ (self|i) + (other|i) for i in range(1,len(self)+1) ]
    suma.corteSistema.update(other.corteSistema)
    return factor(suma)

def __sub__(self, other):
    <<Texto de ayuda para el operador diferencia en la clase Sistema>>
    if not type(self)==type(other) or not len(self)==len(other):
        raise ValueError ('Solo se restan Sistemas del mismo tipo y misma longitud')
    diferencia = self.fullcopy()
    diferencia.lista = [ (self|i) - (other|i) for i in range(1,len(self)+1) ]
    diferencia.corteSistema.update(other.corteSistema)
    return factor(diferencia)

def __radd__(self, other):
    """Suma con algo que no es un Sistema a la izquierda (no está definida)"""
    raise ValueError ('Solo se suman Sistemas del mismo tipo y misma longitud. ' + AVISO_DE_PRECEDENCIA)

def __rsub__(self, other):
    """Resta con algo que no es un Sistema a la izquierda (no está definida)"""
    raise ValueError ('Solo se restan Sistemas del mismo tipo y misma longitud. ' + AVISO_DE_PRECEDENCIA)
```

1.2.4. Producto de un `Sistema` por un escalar a su izquierda

El producto de un sistema A por un escalar x a su izquierda es el *sistema*

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

cuya transcripción literal es

`Sistema ( [ x*(self|i) for i in range(1,len(self)+1) ] )`

donde `x` es un número (`int`, `float` o un objeto del módulo Sympy `sympy.Basic`) y `self` es A.

Casos particulares son el producto de un *vector* $\boldsymbol{a}$ por un escalar x a su izquierda, que es el *vector*:

$$\boxed{\left(x\boldsymbol{a}\right)_{|i} = x\left(\boldsymbol{a}_{|i}\right)} \quad \text{para} \quad i = 1:n.$$

Y el producto de una *matriz* $\mathbf{A}$ por un escalar x a su izquierda, que es la *matriz*:

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

Como en los casos particulares se obtienen *sistemas* de tipos particulares (*vectores* en el primer caso y *matrices* en el segundo), debemos sustituir `Sistema` por `type(self)` para obtener sistemas del mismo tipo que `self`:

`type(self) ( [ x*(self|i) for i in range(1,len(self)+1) ] )`

Texto de ayuda para el operador producto por la izquierda en la clase `Sistema`

Texto de ayuda para el operador producto por la izquierda de un Sistema

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

Parámetros:
    x (int, float o sympy.Basic): Escalar por el que se multiplica
Resultado:
    Sistema resultante de multiplicar cada componente por x
```

```
Ejemplo:
>>> 3 * Sistema([10, 20, 30])
Sistema([30; 60; 90;])
"""
```

Producto de un Sistema por un escalar a su izquierda

```
def __rmul__(self, x):
    <<Texto de ayuda para el operador producto por la izquierda de un Sistema>>
    if es_numero(x):
        x = racional(x)
        multiplo = self.fullcopy()
        multiplo.lista = [ x*(self|i) for i in range(1,len(self)+1) ]
        return factor(multiplo)

    return NotImplemented
```

También nos viene viene bien manejar el opuesto de un `Sistema`: $-\mathsf{A} = -1 \cdot \mathsf{A}$.

Opuesto de un Sistema

```
def __neg__(self):
    """Devuelve el opuesto de un Sistema"""
    return -1*self
```

1.2.5. Producto de un `Sistema` por escalar, `Vector` o `Matrix` a su derecha

En el curso se acepta que el producto de un `Sistema` por un escalar es conmutativo. Por tanto,

$$\boxed{\mathsf{A}x = x\mathsf{A}}$$

por tanto también debemos implementar el producto

```
self * x
```

donde `self` es el `Sistema` y `x` es un número (`int`, `float`, `sympy.Basic`).

El producto de A, de n componentes, por un vector $\boldsymbol{x}$ de $\mathbb{R}^n$ a su derecha se define como

$$\boxed{\mathsf{A}\boldsymbol{x} \;=\; \left(\mathsf{A}_{|1}\right)x_1 + \cdots + \left(\mathsf{A}_{|n}\right)x_n \;=\; \sum_{j=1}^{n}\left(\mathsf{A}_{|j}\right)x_j} \qquad \text{para } j = 1:n.$$

cuya transcripción será

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

donde `self` es un `Sistema` y `x` es un (`Vector`).

Fíjese que el *producto punto* (o producto escalar usual en $\mathbb{R}^n$) de dos vectores $\boldsymbol{a}$ y $\boldsymbol{x}$ en $\mathbb{R}^n$ es un caso particular en el que el sistema A es un vector $\boldsymbol{a}$.

El producto del sistema A de p componentes por una matriz $\underset{p\times n}{\mathbf{X}}$ de $\mathbb{R}^n$ a su derecha se define como el sistema tal que

$$\boxed{(\mathsf{A}\mathbf{X})_{|j} = \mathsf{A}(\mathbf{X}_{|j})} \qquad \text{para } j = 1:n.$$

cuya transcripción será

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

donde `self` es el `Sistema` y `x` es una `Matrix`.

Fíjese que el *producto de matrices* es un caso particular en el que el sistema A es una matriz **A**.

Además, sabemos por las notas de la asignatura que en el caso particular de que el sistema A sea un vector, el resultado es una combinación lineal de las filas de la matriz **X** (es decir, el resultado es un vector). Para recordar que el vector resultante es una combinación lineal de las filas, lo representaremos en forma de fila.

Python muestra el siguiente texto de ayuda al teclear `help(Sistema.__mul__)`.

Texto de ayuda para el operador producto por la derecha en la clase Sistema

```
"""Multiplica un Sistema por un número, Vector o una Matrix a su derecha

Parámetros:
    x (int, float o sympy.Basic): Escalar por el que se multiplica
      (Vector): con tantos componentes como el Sistema
      (Matrix): con tantas filas como componentes tiene el Sistema

Resultado:
    Sistema del mismo tipo: Si x es int, float o sympy.Basic, devuelve
       el Sistema que resulta de multiplicar cada componente por x
    Objeto del mismo tipo de los componentes del Sistema: Si x es Vector,
       devuelve una combinación lineal de los componentes del Sistema,
       donde los componentes de x son los coeficientes de la combinación.
    Sistema del mismo tipo: Si x es Matrix, devuelve un Sistema cuyas
       componentes son combinación lineal de las componentes originales.

Ejemplos:
>>> # Producto por un número
>>> Vector([10, 20, 30]) * 3
Vector([30, 60, 90])
>>> Matrix([[1,2],[3,4]]) * 10
Matrix([Vector([10, 30]), Vector([20, 40])])
>>> # Producto por un Vector
>>> Vector([10, 20, 30]) * Vector([1, 1, 1])
60
>>> Matrix([Vector([1, 3]), Vector([2, 4])]) * Vector([1, 1])
Vector([3, 7])
>>> # Producto por una Matrix
>>> Vector([1,1,1])*Matrix( ( [1,1,1], [2,4,8], [3,-1,0] ) )
Vector([6, 4, 9])
>>> Matrix([Vector([1, 3]), Vector([2, 4])]) * Matrix([Vector([1,1])])
Matrix([Vector([3, 7])])

"""
```

Para implementar la operación `Sistema` por número se llama a la operación número por `Sistema`.

Para implementar `Sistema` por `Vector` se usa la función `sum`; que tiene dos argumentos: el primero es la lista de objetos a sumar, y el segundo es un primer objeto al que se suman los de la lista (por defecto es el *número* "0"). Como la suma de 0 y un elemento del `Sistema` puede no tener sentido, se emplea el siguiente truco: ese primer objeto es el primer elemento de la lista multiplicado por 0.

Para implementar `Sistema` por `Matrix` se usa la operación `Sistema` por `Vector` para generar cada uno de los elementos del sistema resultante. Cuando el `Sistema` es un `Vector`, la operación `Sistema` por `Matrix` calcula el producto de un `Vector` por una `Matrix`. Para recordar que el sistema resultante es una combinación lineal de las filas de la matriz, la representación del resultado sera en forma horizontal (`rpr='h'`) si se emplea la representación `latex`.

Producto de un Sistema por un escalar un Vector o una Matrix a su derecha

```
def __mul__(self,x):
```

```
    <<Texto de ayuda para el operador producto por la derecha en la clase Sistema>>
    if es_numero(x):
        return x*self

    elif isinstance(x, Vector):
        if len(self) != x.n:
            raise ValueError(error_de_producto(self, x))
        if self.es_arreglo_rectangular():
            if not all([f.es_de_composicion_y_longitud_uniforme() for f in ~BlockM([BlockV([i]) for i in
            ↪  self])]):
                raise ValueError('El sistema de la derecha debe tener elementos de composicion y longitud
                ↪  uniforme')
        elif not self.es_de_composicion_y_longitud_uniforme():
            raise ValueError('El sistema de la derecha debe tener elementos de composicion y longitud
            ↪  uniforme')

        return factor(sum([(self|j)*(x|j) for j in range(1,len(self)+1)], 0*self|1))

    elif isinstance(x, Matrix):
        if len(self) != x.m:
            raise ValueError(error_de_producto(self, x))
        if isinstance(self, BlockV):
            return factor(BlockV( [ self*(x|j) for j in range(1,(x.n)+1)], rpr='h' ))
        elif isinstance(self, BlockM):
            return factor(BlockM ( [ self*(x|j) for j in range(1,(x.n)+1)] ))
        else:
            return factor(type(self) ( [ self*(x|j) for j in range(1,(x.n)+1)] ))

    return NotImplemented
```

1.3. Transformaciones elementales de un `Sistema`

En el libro del curso se definen las transformaciones elementales de `Sistemas` de vectores como una generalización a las transformaciones elementales de las columnas de una `Matrix`. Puesto que cada `Matrix` es un `Sistema` de `Vectores`, implementamos de manera general las transformaciones elementales sobre `Sistemas` genéricos.

Como el método no verifica si las operaciones son licitas, podría obtener un error si el sistema contiene objetos incompatibles con dichas operaciones; por ejemplo, si el `Sistema` contiene una cadena de caracteres y un número, al intentar sumar un múltiplo de uno de los elementos al otro obtendremos un error (aunque esto no pasará con los intercambios).

1.3.1. Texto de ayuda transformaciones elementales por la derecha

Texto de ayuda de las transformaciones elementales de un Sistema

```
"""Transforma los elementos de un Sistema

    T(abreviaturas): transformaciones a aplicar sobre un Sistema S
Ejemplos:
>>> # Intercambia los elementos 1º y 3º
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) & T({1,3})
Sistema([Vector([3, 0]); Vector([0, 2]); Vector([1, 0]);])

>>> # Multiplica por 5 el primer elemento
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) & T((5,1))
Sistema([Vector([5, 0]); Vector([0, 2]); Vector([3, 0]);])

>>> # Suma 5 veces el 2º elemento al 1º
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) & T((5,2,1))
Sistema([Vector([1, 10]); Vector([0, 2]); Vector([3, 0]);])
```

```
>>> # Aplica la secuencia de transformaciones en el orden de la lista
>>> Sistema([Vector([1,0]), Vector([0,2]), Vector([3,0])]) & T([{1,3},(5,1),(5,2,1)])
Sistema([Vector([15, 10]); Vector([0, 2]); Vector([1, 0]);])

"""
```

1.3.2. Implementación de las transformaciones elementales por la derecha

(*aunque sea una composición de transformaciones elementales, también se incluye el intercambio.*)

Transformaciones elementales de los elementos de un Sistema

```
def __and__(self,operaciones):
    <<Texto de ayuda de las transformaciones elementales de un Sistema>>
    def transformacionDelSistema(abrv):
        if isinstance(abrv,set):
            self.lista = [ (self|max(abrv)) if k==min(abrv) else \
                           (self|min(abrv)) if k==max(abrv) else \
                           (self|k)                 for k in range(1,len(self)+1)].copy()

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

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

    for abrv in operaciones.abreviaturas:
        transformacionDelSistema(abrv)

    return factor(self)
```

Nótese que al actuar sobre `self.lista`, las transformaciones elementales no crean nuevos `Sistemas` sino que modifican el `Sistema` sobre el que actúan.

1.3.3. Implementación de las transformaciones elementales por la izquierda

Hacen lo mismo que por la derecha (como ocurre con el operador selector)

Transformaciones elementales por la izquierda de un Sistema

```
def __rand__(self, operaciones):
    """Hace exactamente lo mismo que el método __and__ por la derecha."""
    return self & operaciones
```

1.4. Eliminación

La eliminación es el procedimiento central del curso. Casi todos los demás algoritmos de este módulo (el rango, la inversa, la resolución de sistemas de ecuaciones, las bases de un subespacio, el determinante o las diagonalizaciones) consisten en aplicar el método `elim` y leer el resultado. El propio libro lo advierte al enunciar el teorema de *Eliminación*: «¡Prácticamente el resto del curso descansa sobre este teorema!».

El método implementa las demostraciones de la Sección 4.2 del libro (*Eliminación por columnas*). Recordemos sus definiciones:

- El *pivote* de una columna no nula es su primer componente no nulo, y su *posición de pivote* es el índice de la fila en la que se encuentra.
- Una matriz es *pre-escalonada* si las componentes a la derecha de cada pivote son nulas.
- Una matriz es *escalonada* si es pre-escalonada, los pivotes describen una escalera descendente y las columnas nulas (si las hay) están a la derecha.
- Una matriz es *escalonada reducida* si es escalonada, cada pivote es un uno y el resto de componentes de la fila de cada pivote son ceros.

El libro demuestra que toda matriz se puede transformar en cada uno de esos tres tipos de matriz mediante una secuencia de transformaciones elementales de sus columnas:

Resultado del libro	Sección	Matriz que se obtiene
Teorema de *Eliminación*	4.2.1	pre-escalonada
Corolario *Eliminación gaussiana*	4.2.2	escalonada
Corolario *Eliminación Gauss-Jordan*	4.2.3	escalonada reducida

Las tres demostraciones son constructivas (indican qué transformaciones hay que aplicar), así que programarlas es traducirlas a Python.

El método `elim` pertenece a la clase `Sistema`, y no a la subclase `Matrix`, porque las transformaciones elementales actúan sobre cualquier sistema de vectores (Sección 9.3). Basta generalizar la idea de pivote: el pivote de un elemento del sistema es su primer componente no nulo, y si dicho componente es a su vez un `Sistema`, se busca dentro de él su primer componente no nulo, y así sucesivamente (por eso los métodos `primer_no_nulo` y `ultimo_no_nulo` devuelven una lista de índices). De este modo el mismo código pre-escalona las columnas de una `Matrix`, un `Sistema` de matrices (véase el ejemplo de la clase `SEL`) o los sistemas «acoplados» de vectores de la Sección 9.5.1.

1.4.1. El método `elim`

El método recorre los elementos del sistema (las columnas, si se trata de una matriz) de izquierda a derecha. Si el elemento visitado no es nulo, se toma su pivote y con él se anulan, mediante transformaciones elementales, los componentes de los elementos *posteriores* que están en esa misma posición de pivote. El bucle `for` del código hace exactamente eso: `indiceXP` es el índice del elemento visitado, `pivote` su pivote y `Reduccion` la secuencia de transformaciones que anula lo que hay a su derecha.

Al terminar el recorrido el sistema está pre-escalonado. En efecto, tras visitar el elemento j-ésimo todos los elementos posteriores tienen un cero en la posición de su pivote; y los pasos siguientes solo suman a esos elementos posteriores múltiplos de otros elementos posteriores, que también tienen un cero en dicha posición. Por tanto, los ceros logrados a la derecha de cada pivote ya no se pierden.

La demostración del libro llega al mismo sitio por inducción sobre el número de filas (pre-escalona las primeras filas y después se ocupa de la última), mientras que `elim` avanza columna a columna. El orden de las operaciones es distinto, pero en ambos casos se usa cada pivote para eliminar lo que tiene a su derecha, y el resultado es una matriz pre-escalonada.

Tras la reducción, y solo si la variante elegida lo pide, se normalizan los pivotes y se reordenan los elementos para escalonar el sistema. El método no modifica el sistema original: opera sobre una copia y devuelve un nuevo sistema del mismo tipo.

Los argumentos del método son:

variante número que indica el modo de proceder (se describe en el siguiente apartado). Por defecto es 0: pre-escalonar.

rep si es verdadero, se muestran en Jupyter todos los pasos del proceso.

sust lista de reglas de sustitución para las variables simbólicas.

repsust si es verdadero, los pasos se muestran con las sustituciones ya hechas.

Eliminación

```
def elim(self, variante=0, rep=False, sust=[], repsust=False):
    """Forma pre-escalonada de un Sistema mediante transformaciones elementales

    Parámetros:
        variante: suma de los números (distintos) que indican el modo de
              proceder (por defecto 0: pre-escalona operando con los
              elementos del sistema, de izquierda a derecha, y evitando
              las fracciones)
               +1 reducción rápida (solo transformaciones Tipo I)
               +2 doble reducción (a ambos lados de cada pivote)
               +4 por filas
              +10 normalización de los pivotes
              +20 escalonamiento
             +100 de atrás hacia delante
        rep:  si es True se muestran los pasos en Jupyter
        sust: lista de reglas de sustitución de variables simbólicas
        repsust: si es True los pasos se muestran tras la sustitución

    Atributos del Sistema devuelto:
        pasos: [pasos por la izquierda (filas), pasos por la derecha]
        TrF, TrC: transformaciones aplicadas por la izquierda y por la derecha
        tex:   código LaTeX con la representación de todo el proceso

    Ejemplos:
    >>> A = Matrix([ [0,2,2], [1,-2,-8], [1,1,1] ])
    >>> A.elim()      # una forma pre-escalonada   (lo mismo que A.K())
    Matrix([Vector([0, 1, 1]), Vector([2, 0, 3]), Vector([0, 0, 6])])
    >>> A.elim(20)    # una forma escalonada       (lo mismo que A.L())
    Matrix([Vector([2, 0, 3]), Vector([0, 1, 1]), Vector([0, 0, 6])])
    >>> A.elim(32)    # forma escalonada reducida  (lo mismo que A.R())
    Matrix([Vector([1, 0, 0]), Vector([0, 1, 0]), Vector([0, 0, 1])])
    >>> A.elim().TrC  # transformaciones aplicadas a las columnas
    T([(2, 1, 2), (8, 1, 3), (-1, 2, 3)])
    """
    <<Métodos auxiliares de la variante de eliminación>>
    <<Reducción con un pivote>>
    <<Normalización de los pivotes>>
    <<Escalonamiento>>
    <<Aplicación de un paso de eliminación y registro de las transformaciones>>
    <<Sistema final con los atributos del proceso de eliminación>>
    <<Análisis de las opciones de eliminación elegidas>>

    if not self:
        return sistemaFinalYPasosDchaIzda(Sistema([]), [T([])] )

    if not self.es_de_composicion_y_longitud_uniforme():
        raise ValueError('Los elementos del sistema deben ser del mismo tipo y longitud')

    ValorAEliminar = lambda indiceVAE: sistema.extractor([indiceVAE]+posicionXPivote(sistema|indiceXP))
    sistema, recorrido, XPivote, posicionXPivote, componentesAmodificar = metodos_auxiliares_de_la(variante)

    pasosAcumulados = []
    for indiceXP,_ in recorrido(sistema):
        pivote = XPivote(sistema|indiceXP)
        if pivote:                                  # reducción
            sistema, pasosAcumulados = transformacionYPasos(sistema,  Reduccion,  pasosAcumulados)
```

```
    if 10 in analisis_opcion_elegida(variante):  # normalización de pivotes
        sistema, pasosAcumulados = transformacionYPasos(sistema,  Normalizacion,  pasosAcumulados)

    if 20 in analisis_opcion_elegida(variante):  # escalonamiento
        sistema, pasosAcumulados = transformacionYPasos(sistema, Escalonamiento,  pasosAcumulados)

    return sistemaFinalYPasosDchaIzda(sistema, pasosAcumulados)
```

El método comienza comprobando dos casos especiales: un sistema vacío ya está pre-escalonado; y si los elementos del sistema no son todos del mismo tipo y longitud no es posible operar con ellos (no se puede sumar un vector de $\mathbb{R}^2$ con uno de $\mathbb{R}^3$), así que se lanza un error.

1.4.2. Variantes de eliminación

Todas las variantes del método comparten el mismo esquema, y se eligen sumando los números de la siguiente tabla:

Número	Efecto
+1	Reducción rápida: solo transformaciones Tipo I (pueden aparecer fracciones)
+2	Doble reducción: con cada pivote se elimina a su derecha y a su izquierda
+4	Por filas: se opera con las filas en lugar de con las columnas
+10	Normalización: se divide cada elemento no nulo por su pivote
+20	Escalonamiento: se reordenan los elementos para que los pivotes formen una escalera
+100	De atrás hacia delante: con el último componente no nulo se elimina a su izquierda

Así, la variante 32 ($= 20 + 10 + 2$) realiza una doble reducción, normaliza los pivotes y escalona el resultado; es decir, obtiene la forma escalonada reducida.

Como cada número de la lista es mayor que la suma de todos los menores que él, la descomposición de `variante` en sumandos es única, y se obtiene restando en cada paso el mayor número posible de la lista. Si al terminar queda un resto, el número no era una suma válida y se lanza un error.

Análisis de las opciones de eliminación elegidas

```
def analisis_opcion_elegida(tipo):
    'Análisis de las opciones de eliminación elegidas'
    lista = [100,20,10,4,2,1]
    opcion = set()
    for t in lista:
        if (tipo - (tipo % t)) in lista:
            opcion.add(tipo - (tipo % t))
            tipo = tipo % t
    if tipo:
        raise ValueError('Variante de eliminación no válida: debe ser una suma de números distintos de la
        ↪  lista [100, 20, 10, 4, 2, 1]')
    return opcion
```

Modo de recorrer el sistema y de localizar los pivotes

Según la variante elegida se definen cinco objetos que usa el bucle principal:

`sistema` la copia del sistema sobre la que se opera (con las sustituciones de **`sust`** ya hechas).

`recorrido` el orden en el que se visitan los elementos.

`XPivote` y `posicionXPivote` el componente que hace de pivote en un elemento, y su posición.

`componentesAmodificar` los índices de los elementos en los que se va a eliminar.

Por defecto, el recorrido es de izquierda a derecha, el pivote es el primer componente no nulo y se elimina en los elementos posteriores. Con la opción +100 todo se hace al revés: el recorrido es de derecha a izquierda, se usa el *último* componente no nulo y se elimina en los elementos anteriores. Con la opción +2 se elimina en todos los elementos distintos del visitado.

La eliminación por filas (opción +4) no necesita código propio. La Sección 4.A del libro muestra que transformar las filas de una matriz es transformar las columnas de su transpuesta y volver a transponer, y eso es literalmente lo que se hace: se opera sobre la transpuesta `~self` y al final se deshace la transposición. Por eso esta opción exige que el sistema sea un arreglo rectangular.

Métodos auxiliares de la variante de eliminación

```
def metodos_auxiliares_de_la(variante):
    """Define los métodos auxiliares y el modo de actuación sobre el sistema
    en función de la variante de eliminación elegida.

    'variante' es la suma de los siguientes números:

       +1 reduccion rápida (solo transformaciones tipo I)
       +2 doble reducción
       +4 por filas
      +10 normalización de los pivotes
      +20 escalonamiento
     +100 de atrás hacia delante

    Por defecto variante = 0 (reducción simple de delante hacia atrás,
    por columnas y evitando fracciones)

    """
    if 100 in analisis_opcion_elegida(variante): # recorrido de atrás hacia delante
        componentesAmodificar     = lambda    sistema:  filter(lambda x:  x < indiceXP,
        ↪  range(1,len(sistema)+1))
        recorrido                 = lambda    sistema:  reversed(list(enumerate(CreaSistema(sistema),1)))
        XPivote                   = lambda componente:  elementoAntiPivote(componente)
        posicionXPivote           = lambda componente:  ultimo_no_nulo(componente)

    else:                                    # recorrido de delante hacia atrás
        componentesAmodificar     = lambda    sistema:  filter(lambda x:  x > indiceXP,
        ↪  range(1,len(sistema)+1))
        recorrido                 = lambda    sistema:  enumerate(CreaSistema(sistema),1)
        XPivote                   = lambda componente:  elementoPivote(componente)
        posicionXPivote           = lambda componente:  primer_no_nulo(componente)

    if 4 in analisis_opcion_elegida(variante):   # reducción de los componentes en arreglos rectangulares
        if (not self.es_arreglo_rectangular()) or (not all([item.es_de_composicion_uniforme() for item in
        ↪  self])):
            raise ValueError('El sistema debe ser un arreglo rectangular con componentes de composición
            ↪  uniforme')
        sistema = ~self.fullcopy().subs(sust);
    else:
        sistema = self.fullcopy().subs(sust);

    if 2 in analisis_opcion_elegida(variante):   # doble reducción (reducción posiciones anteriores y
    ↪  posteriores al pivote)
        componentesAmodificar = lambda    sistema:  filter(lambda x: x != indiceXP, range(1,len(sistema)+1))

    return sistema, recorrido, XPivote, posicionXPivote, componentesAmodificar
```

Reducción con un pivote

La función `Reduccion` construye la secuencia de transformaciones que anula, con el pivote del elemento visitado, los componentes de los demás elementos que están en su posición de pivote. Si p es el pivote y a el componente que queremos anular en el elemento j-ésimo, hay dos modos de hacerlo:

Reducción rápida (opción +1). Es la de la demostración del libro: se resta al elemento j-ésimo el elemento del pivote multiplicado por a/p. Es una única transformación Tipo I, pero si a/p no es entero aparecen fracciones.

Reducción sin fracciones (por defecto). Si n/d es la fracción irreducible igual a a/p, primero se multiplica el elemento j-ésimo por d (transformación Tipo II) y después se le resta el elemento del pivote multiplicado por n. Así, si el sistema solo contiene enteros, todos los pasos se dan con números enteros, que es lo más cómodo cuando se hacen las cuentas con lápiz y papel.

Cuando a es cero ambos modos generan transformaciones que no hacen nada (sumar cero veces un elemento, o multiplicar por uno). La función `filtradopasos` las descarta para que no aparezcan al representar el proceso.

Reducción con un pivote

```
def Reduccion(sistema):
    if 1 in analisis_opcion_elegida(variante):   # reducción rápida (solo transformaciones tipo I)
        operaciones = [ (-fracc(ValorAEliminar(indiceVAE), pivote), indiceXP, indiceVAE)  \
                                                    for indiceVAE in componentesAmodificar(sistema)]
    else:                                        # reducción lenta (evitando fracciones)
        operaciones = [[( denom(ValorAEliminar(indiceVAE), pivote),           indiceVAE), \
                        (-numer(ValorAEliminar(indiceVAE), pivote), indiceXP, indiceVAE)] \
                                                    for indiceVAE in componentesAmodificar(sistema)]
    return filtradopasos(T(operaciones))
```

Normalización de los pivotes y escalonamiento

La normalización (opción +10) multiplica cada elemento no nulo por el inverso de su pivote.

Normalización de los pivotes

```
def Normalizacion(sistema):
    return filtradopasos(T([ (fracc(1, XPivote(sistema|indiceXP)), indiceXP)
                             for indiceXP,_ in recorrido(sistema) if XPivote(sistema|indiceXP)]))
```

El escalonamiento (opción +20) es la demostración del corolario de *Eliminación gaussiana*: «basta pre-escalonar la matriz y reordenar sus columnas para que los pivotes describan una escalera descendente, dejando las columnas nulas a la derecha». Para ello se recorren las filas de arriba a abajo; cada vez que se encuentra una fila que es posición de pivote de alguna columna aún no colocada, dicha columna se intercambia con la que ocupa el primer lugar todavía libre. Como solo se mueven columnas con pivote, las columnas nulas terminan a la derecha. La función devuelve la secuencia de intercambios.

Escalonamiento

```
def Escalonamiento(sistema):
    M = sistema.copy()
    if 100 in analisis_opcion_elegida(variante): # con recorrido de atrás hacia delante
        destino       = lambda     : (M.n)-r+1
        resto         = lambda    r: slice(None, max(M.n-r,1))
        columnaAMover = lambda i, r: posicionXPivote(i|M|resto(r))[0]   if posicionXPivote(i|M|resto(r)) and
        ↪  i==posicionXPivote(M|posicionXPivote(i|M|resto(r))[0]  )[0] else 0
    else:                                        # con recorrido de delante hacia atrás
        destino       = lambda     : r
```

```
        resto         = lambda    r: slice(r+1, None)
        columnaAMover = lambda i, r: posicionXPivote(i|M|resto(r))[0]+r if posicionXPivote(i|M|resto(r)) and
        ↪  i==posicionXPivote(M|posicionXPivote(i|M|resto(r))[0]+r)[0] else 0

    r = 0
    intercambios = []
    for i,_ in recorrido(M|1):
        indiceColumnaPivote = columnaAMover(i,r)
        if indiceColumnaPivote:
            r += 1
            intercambio  = T( {destino(), indiceColumnaPivote} )
            M & intercambio
            intercambios.append(intercambio)

    return filtradopasos(T(intercambios))
```

Nótese que el orden de las operaciones en el corolario de *Eliminación Gauss-Jordan* del libro es escalonar, reducir y normalizar, mientras que `elim(32)` reduce, normaliza y escalona. El resultado es el mismo, pues la forma escalonada reducida de una matriz es única.

Registro de los pasos y sistema resultante

Cada grupo de transformaciones (la reducción con un pivote, la normalización o el escalonamiento) es un *paso*. La función `transformacionYPasos` calcula un paso, lo aplica al sistema y lo añade a la lista de pasos ya dados. Cuando se opera por filas se trabaja con la transpuesta, así que el paso que se guarda es el transpuesto y, como las transformaciones de las filas se escriben a la izquierda de la matriz, se añade al principio de la lista:

$$_{\tau_k\cdots\tau_1}\mathbf{A} \qquad \text{frente a} \qquad \mathbf{A}_{\tau_1\cdots\tau_k}.$$

Aplicación de un paso de eliminación y registro de las transformaciones

```
def transformacionYPasos(sistema, operacion, pasosPrevios):
    pasoDado = operacion(sistema)
    if 4 in analisis_opcion_elegida(variante):    # reducción de los componentes en arreglos rectangulares
        pasosAcumulados = [~pasoDado] + pasosPrevios if pasoDado else pasosPrevios
    else:
        pasosAcumulados = pasosPrevios  + [pasoDado] if pasoDado else pasosPrevios
    sistema & T(pasoDado)
    return factor(sistema.subs(sust)), pasosAcumulados
```

Por último, `sistemaFinalYPasosDchaIzda` deshace la transposición si se ha operado por filas y añade al sistema resultante cuatro atributos que guardan la historia del proceso:

`pasos` lista con dos listas de pasos: los dados por la izquierda (sobre las filas) y los dados por la derecha (sobre las columnas, o sobre los elementos del sistema).

`TrF y TrC` las mismas transformaciones, pero como objetos de la clase `T`; de manera que si `B` es el resultado de aplicar `elim` sobre `A`, entonces `B` es igual a `B.TrF & A & B.TrC`.

`tex` el código LaTeX que representa el proceso completo.

Estos atributos son los que usan el resto de procedimientos del módulo. Por ejemplo, para invertir una matriz no interesa la matriz reducida, sino las transformaciones que la han reducido. Además, si el sistema de partida ya tenía estos atributos (porque es el resultado de una eliminación previa) los nuevos pasos se añaden a los anteriores, así que se pueden encadenar varias eliminaciones sin perder la historia (el método `texYpasos` se describe en el siguiente apartado).

Sistema final con los atributos del proceso de eliminación

```
def sistemaFinalYPasosDchaIzda(sistema,transformaciones):
    if 4 in analisis_opcion_elegida(variante):    # reducción de los componentes en arreglos rectangulares
```

```
        TransformacionesPorLaIzquierda = filtradopasos(transformaciones)
        TransformacionesPorLaDerecha   = []
        if self.es_arreglo_rectangular():
            sistema = ~sistema
    else:
        TransformacionesPorLaDerecha   = filtradopasos(transformaciones)
        TransformacionesPorLaIzquierda = []

    SistemaFinal =  sistema.subs(sust)
    pasos        = [TransformacionesPorLaIzquierda, TransformacionesPorLaDerecha]
    SistemaFinal.tex, SistemaFinal.pasos = texYpasos(self, pasos, rep, sust, repsust)
    SistemaFinal.TrF = T(SistemaFinal.pasos[0])
    SistemaFinal.TrC = T(SistemaFinal.pasos[1])
    return factor(SistemaFinal)
```

1.4.3. Formas pre-escalonada, escalonada y escalonada reducida

Por comodidad se definen métodos con el nombre que el libro da a cada tipo de matriz. Además, por compatibilidad con versiones anteriores del módulo, cada uno tiene asociada una clase que hace lo mismo (se definen más adelante): `Elim(A)` es lo mismo que `A.K()`.

Método	Variante	Clase	Resultado	Libro
`K()`	`elim(0)`	`Elim`	una forma pre-escalonada	4.2.1
`L()`	`elim(20)`	`ElimG`	una forma escalonada	4.2.2
`R()`	`elim(32)`	`ElimGJ`	la forma escalonada reducida	4.2.3
	`elim(4)`	`ElimF`	una forma pre-escalonada por filas	4.A
`U()`	`elim(24)`	`ElimGF`	una forma escalonada por filas	4.A.1
`UR()`	`elim(36)`	`ElimGJF`	la forma escalonada reducida por filas	4.A.1

formas escalonadas

```
def K(self,rep=0, sust=[], repsust=0):
    """Una forma pre-escalonada por columnas (K) de una Matrix"""
    return self.elim(0, rep, sust, repsust)

def L(self,rep=0, sust=[], repsust=0):
    """Una forma escalonada por columnas (L) de una Matrix"""
    return self.elim(20, rep, sust, repsust)

def R(self,rep=0, sust=[], repsust=0):
    """Forma escalonada reducida por columnas (R) de una Matrix"""
    return self.elim(32, rep, sust, repsust)

def U(self,rep=0, sust=[], repsust=0):
    """Una forma escalonada por filas (U) de una Matrix"""
    return self.elim(24, rep, sust, repsust)

def UR(self,rep=0, sust=[], repsust=0):
    """Una forma escalonada reducida por filas (U) de una Matrix"""
    return self.elim(36, rep, sust, repsust)
```

1.4.4. Ejemplos de eliminación

Usaremos la matriz del ejemplo de la Sección 4.2.1 del libro.

```
A = Matrix( [ [0,2,2], [1,-2,-8], [1,1,1] ] )
```

El atributo `tex` de la forma pre-escalonada muestra el proceso. Con el pivote de la primera columna (el 1 de la segunda fila) se anulan los componentes de la segunda fila que están a su derecha, y

después se hace lo mismo con el pivote de la segunda columna (el 2 de la primera fila):

```
print(displaymath( A.K().tex ))
```

$$\begin{bmatrix} 0 & 2 & 2 \\ 1 & -2 & -8 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow[\substack{[(2)\mathbf{1}+\mathbf{2}]\\[(8)\mathbf{1}+\mathbf{3}]}]{\boldsymbol{\tau}} \begin{bmatrix} 0 & 2 & 2 \\ 1 & 0 & 0 \\ 1 & 3 & 9 \end{bmatrix} \xrightarrow[[(-1)\mathbf{2}+\mathbf{3}]]{\boldsymbol{\tau}} \begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 1 & 3 & 6 \end{bmatrix}$$

Cada flecha es un *paso*, y el atributo `pasos` los guarda todos. La primera lista está vacía porque no se ha transformado ninguna fila:

```
K = A.K()
print(dprint( K.pasos ))
```

```
[[], [T([(2, 1, 2), (8, 1, 3)]), T([(-1, 2, 3)])]]
```

El atributo `TrC` es la secuencia completa de transformaciones de las columnas

```
print(dmath( K.TrC ))
```

$$\begin{matrix}\boldsymbol{\tau}\\ [(2)\mathbf{1}+\mathbf{2}]\\ [(8)\mathbf{1}+\mathbf{3}]\\ [(-1)\mathbf{2}+\mathbf{3}]\end{matrix}$$

y al aplicarla sobre `A` se obtiene de nuevo la forma pre-escalonada:

```
print(dprint( Matrix(A) & K.TrC == K ))
```

```
True
```

(como las transformaciones elementales modifican el sistema sobre el que actúan, se han aplicado sobre la copia `Matrix(A)`).

Reducción con y sin fracciones

En la siguiente matriz el pivote de la primera columna es 2 y a su derecha hay un 3 y un 1.

```
D = Matrix( [ [2,3,1], [1,1,1], [4,5,7] ] )
```

Por defecto se evitan las fracciones multiplicando antes las columnas en las que se va a eliminar:

```
print(displaymath( D.elim(0).tex ))
```

$$\begin{bmatrix} 2 & 3 & 1 \\ 1 & 1 & 1 \\ 4 & 5 & 7 \end{bmatrix} \xrightarrow[\substack{[(2)\mathbf{2}]\\[(-3)\mathbf{1}+\mathbf{2}]\\[(2)\mathbf{3}]\\[(-1)\mathbf{1}+\mathbf{3}]}]{\boldsymbol{\tau}} \begin{bmatrix} 2 & 0 & 0 \\ 1 & -1 & 1 \\ 4 & -2 & 10 \end{bmatrix} \xrightarrow[[(1)\mathbf{2}+\mathbf{3}]]{\boldsymbol{\tau}} \begin{bmatrix} 2 & 0 & 0 \\ 1 & -1 & 0 \\ 4 & -2 & 8 \end{bmatrix}$$

Con la reducción rápida (opción +1) solo se usan transformaciones Tipo I, como en la demostración del libro, y aparecen las fracciones:

```
print(displaymath( D.elim(1).tex ))
```

$$\begin{bmatrix} 2 & 3 & 1 \\ 1 & 1 & 1 \\ 4 & 5 & 7 \end{bmatrix} \xrightarrow[\substack{[(-\frac{1}{2})\mathbf{1}+\mathbf{3}]}]{\overset{\boldsymbol{\tau}}{[(-\frac{3}{2})\mathbf{1}+\mathbf{2}]}} \begin{bmatrix} 2 & 0 & 0 \\ 1 & -\frac{1}{2} & \frac{1}{2} \\ 4 & -1 & 5 \end{bmatrix} \xrightarrow{\boldsymbol{\tau}\ [(1)\mathbf{2}+\mathbf{3}]} \begin{bmatrix} 2 & 0 & 0 \\ 1 & -\frac{1}{2} & 0 \\ 4 & -1 & 4 \end{bmatrix}$$

Eliminación gaussiana y Gauss-Jordan

Para escalonar, tras la reducción se intercambian las columnas:

```
print(displaymath( A.L().tex ))
```

$$\begin{bmatrix} 0 & 2 & 2 \\ 1 & -2 & -8 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow[\substack{[(8)\mathbf{1}+\mathbf{3}]}]{\overset{\boldsymbol{\tau}}{[(2)\mathbf{1}+\mathbf{2}]}} \begin{bmatrix} 0 & 2 & 2 \\ 1 & 0 & 0 \\ 1 & 3 & 9 \end{bmatrix} \xrightarrow{\boldsymbol{\tau}\ [(-1)\mathbf{2}+\mathbf{3}]} \begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 1 & 3 & 6 \end{bmatrix} \xrightarrow{\boldsymbol{\tau}\ [\mathbf{1}\rightleftharpoons\mathbf{2}]} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 1 & 6 \end{bmatrix}$$

Para obtener la forma escalonada reducida se elimina también a la izquierda de cada pivote, se normalizan los pivotes y, si hace falta, se intercambian las columnas. En el siguiente ejemplo la matriz tiene rango 2, así que queda una columna nula a la derecha:

```
C = Matrix( [ [1,2,3], [2,4,6], [1,1,1] ] )
print(displaymath( C.R().tex ))
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow[\substack{[(-3)\mathbf{1}+\mathbf{3}]}]{\overset{\boldsymbol{\tau}}{[(-2)\mathbf{1}+\mathbf{2}]}} \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & -1 & -2 \end{bmatrix} \xrightarrow[\substack{[(-2)\mathbf{2}+\mathbf{3}]}]{\overset{\boldsymbol{\tau}}{[(1)\mathbf{2}+\mathbf{1}]}} \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \xrightarrow{\boldsymbol{\tau}\ [(-1)\mathbf{2}]} \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Eliminación por filas

Cuando se opera con las filas las transformaciones se representan bajo las flechas. Como se escriben a la izquierda de la matriz, la primera transformación que se aplica es la última de cada lista (la de más abajo):

```
print(displaymath( A.U().tex ))
```

$$\begin{bmatrix} 0 & 2 & 2 \\ 1 & -2 & -8 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow[\substack{[(-1)\mathbf{1}+\mathbf{3}]\\[(2)\mathbf{3}]\\[(1)\mathbf{1}+\mathbf{2}]}]{\boldsymbol{\tau}} \begin{bmatrix} 0 & 2 & 2 \\ 1 & 0 & -6 \\ 2 & 0 & 0 \end{bmatrix} \xrightarrow[\substack{[(-2)\mathbf{2}+\mathbf{3}]}]{\boldsymbol{\tau}} \begin{bmatrix} 0 & 2 & 2 \\ 1 & 0 & -6 \\ 0 & 0 & 12 \end{bmatrix} \xrightarrow[\substack{[\mathbf{1}\rightleftharpoons\mathbf{2}]}]{\boldsymbol{\tau}} \begin{bmatrix} 1 & 0 & -6 \\ 0 & 2 & 2 \\ 0 & 0 & 12 \end{bmatrix}$$

Eliminaciones encadenadas

Al aplicar un método de eliminación sobre el resultado de otro, los pasos se acumulan. Aquí se pre-escalona por columnas y después se escalona por filas:

```
E  = Matrix( [ [1,2,3], [1,1,1], [2,4,6] ] )
KU = E.K().U()
print(displaymath( KU.tex ))
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \\ 2 & 4 & 6 \end{bmatrix} \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{2}]\\[(-3)\mathbf{1}+\mathbf{3}]}]{\boldsymbol{\tau}} \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & -2 \\ 2 & 0 & 0 \end{bmatrix} \xrightarrow[[(-2)\mathbf{2}+\mathbf{3}]]{\boldsymbol{\tau}} \begin{bmatrix} 1 & 0 & 0 \\ 1 & -1 & 0 \\ 2 & 0 & 0 \end{bmatrix} \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{3}]\\[(-1)\mathbf{1}+\mathbf{2}]}]{\boldsymbol{\tau}} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

El atributo `pasos` guarda las transformaciones de las filas en la primera lista y las de las columnas en la segunda:

```
print(dprint( KU.pasos ))
```

```
[[T([(-2, 1, 3), (-1, 1, 2)])], [T([(-2, 1, 2), (-3, 1, 3)]), T([(-2, 2, 3)])]]
```

Eliminación de atrás hacia delante

Con la opción +100 se recorren las columnas de derecha a izquierda, y con el último componente no nulo de cada una se anula lo que hay a su izquierda:

```
print(displaymath( A.elim(100).tex ))
```

$$\begin{bmatrix} 0 & 2 & 2 \\ 1 & -2 & -8 \\ 1 & 1 & 1 \end{bmatrix} \xrightarrow[\substack{[(-1)\mathbf{3}+\mathbf{1}]\\[(-1)\mathbf{3}+\mathbf{2}]}]{\boldsymbol{\tau}} \begin{bmatrix} -2 & 0 & 2 \\ 9 & 6 & -8 \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow[\substack{[(2)\mathbf{1}]\\[(-3)\mathbf{2}+\mathbf{1}]}]{\boldsymbol{\tau}} \begin{bmatrix} -4 & 0 & 2 \\ 0 & 6 & -8 \\ 0 & 0 & 1 \end{bmatrix}$$

1.4.5. Representación de los procesos de eliminación Gaussiana

Cuando hemos encadenado varios procedimientos de eliminación, deberíamos poder ver los pasos desde el principio hasta el final. Para ello comprobamos si `data` fue obtenido mediante un proceso previo de eliminación. El modo de saberlo es comprobar si `data` posee el atributo `pasos`. El atributo `tex` guarda el código LaTeX que muestra el proceso completo, y se construye aplicando el método `rprElim`. El atributo `pasos` guarda las listas de abreviaturas de las transformaciones elementales empleadas. Por comodidad añadimos dos atributos más: `TrF` es la transformación (de la clase `T`) aplicada a las filas y `TrC` es la aplicada a las columnas.

Método que define los atributos .tex y .pasos y representa los pasos si se pide

```
def texYpasos(data, pasos, rep=0, sust=[], repsust=0):
    pasosPrevios = data.pasos if hasattr(data, 'pasos') and data.pasos else [[],[]]
    TexPasosPrev = data.tex   if hasattr(data, 'tex')   and data.tex   else []
    if repsust:
        tex = rprElim(data, pasos, TexPasosPrev, sust)
    else:
```

```
        tex = rprElim(data, pasos, TexPasosPrev)
    pasos[0] = pasos[0] + pasosPrevios[0]
    pasos[1] = pasosPrevios[1] + pasos[1]

    if rep:
        display(Math(tex))

    return tex, pasos
```

Cuando mostramos los pasos, es más legible mostrar únicamente los que modifican 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).

El atributo `tex` guardará el código LaTeX que muestra el proceso completo. Si ha habido transformaciones previas, la cadena de LaTeX que permite su representación en el entorno Jupyter estará guardada en la variable (`TexPasosPrev`), y a dicha cadena hay que añadir la correspondiente cadena de LaTeX que permita representar los nuevos `pasos` dados como argumento de este método. Si `TexPasosPrev` es vacío, la escritura comienza con la representación de `data`. A la hora de representar los pasos hay que tener en cuenta si se dan sobre las filas (`l==0`) o sobre las columnas (`l==1`).

Las funciones que escriben un desarrollo no devuelven una cadena corriente, sino un `TexDesarrollo`: una cadena (se usa exactamente igual que cualquier otra y su contenido es el código LaTeX de siempre) que además recuerda de qué eslabones está hecha, es decir, la lista `[sistema, flecha, sistema, ..., flecha, sistema]`. Así, la función `desarrollo` puede recomponer el código sin tener que analizarlo. Si el desarrollo continúa un código previo que es una cadena corriente (por ejemplo, uno escrito a mano), sus eslabones se obtienen analizándolo, y solo cuando se piden.

Cadena con el código de un desarrollo que recuerda sus eslabones

```
class TexDesarrollo(str):
    """Código LaTeX de un desarrollo que recuerda sus eslabones

    Es una cadena formada por un código previo seguido de nuevos
    eslabones [flecha, sistema, flecha, sistema, ...]. El atributo
    trozos es la lista [sistema, flecha, sistema, ..., flecha, sistema]
    de todo el desarrollo."""
    def __new__(cls, previo, eslabones=()):
        eslabones = list(eslabones)
        self = super().__new__(cls, previo + ''.join(eslabones))
        self._previo, self._eslabones = previo, eslabones
        return self

    @property
    def trozos(self):
        previo = self._previo
        return (previo.trozos if isinstance(previo, TexDesarrollo)
                else trozos_del_desarrollo(previo)) + self._eslabones
```

Representación de un proceso de eliminación rprElim

```
def rprElim(data, pasos, TexPasosPrev=[], sust=[], metodo=factor):
    """Escribe en LaTeX los pasos efectivos y los sucesivos sistemas"""
    A     = data.fullcopy().subs(sust)
    previo, nuevos = (latex(A) if not TexPasosPrev else TexPasosPrev), []

    simplifica = lambda metodo,expresion: metodo(expresion)

    # transformaciones por la izquierda
```

```
    for  _,pasoDeEliminacion in enumerate(pasos[0][::-1]):
        if data.es_arreglo_rectangular(): # entonces transforman las filas
            nuevos.append('\\xrightarrow[' + latex( pasoDeEliminacion.subs(sust) ) + ']{}')
            nuevos.append(latex( simplifica(metodo, ((pasoDeEliminacion & A).subs(sust)) ) ))
        else:  # hacen lo mismo que por la derecha
            nuevos.append('\\xrightarrow{' + latex( pasoDeEliminacion.subs(sust) ) + '}')
            nuevos.append(latex( simplifica(metodo, ((A & pasoDeEliminacion).subs(sust)) ) ))

    # transformaciones por la derecha
    for  _,pasoDeEliminacion in enumerate(pasos[1]):
        nuevos.append('\\xrightarrow{' + latex( pasoDeEliminacion.subs(sust) ) + '}')
        nuevos.append(latex( simplifica(metodo, ((A & pasoDeEliminacion).subs(sust)) ) ))

    return TexDesarrollo(previo, nuevos)
```

Representación de un proceso de eliminación rprElimCF

```
def rprElimCF(data, pasos, TexPasosPrev=[], sust=[], metodo=factor):
    """Escribe en LaTeX los pasos efectivos y los sucesivos arreglos rectangulares"""
    if not data.es_arreglo_rectangular():
        raise ValueError('El sistema tiene que ser un arreglo rectangular')
    if len(pasos[0])!=len(pasos[1]):
        raise ValueError('Esta representación requiere el mismo número de pasos por la izquierda y la
        ↪  derecha')

    A = data.fullcopy().subs(sust)
    previo, nuevos = (latex(data) if not TexPasosPrev else TexPasosPrev), []

    simplifica = lambda metodo,expresion: metodo(expresion)

    for  i,pasoDeEliminacionFilas in enumerate(pasos[0][::-1]):
        nuevos.append('\\xrightarrow{' + latex( (pasos[1][i]).subs(sust) ) + '}')
        nuevos.append(latex( simplifica(metodo, ((A & pasos[1][i]).subs(sust))) ))
        nuevos.append('\\xrightarrow[' + latex( (pasoDeEliminacionFilas).subs(sust) ) + ']{}')
        nuevos.append(latex( simplifica(metodo, ((pasoDeEliminacionFilas & A).subs(sust))) ))

    return TexDesarrollo(previo, nuevos)
```

Representación de un proceso de eliminación rprElimFyC

```
def rprElimFyC(data, pasos, TexPasosPrev=[], sust=[], metodo=factor):
    """Escribe en LaTeX los pasos efectivos y los sucesivos arreglos rectangulares"""
    if not data.es_arreglo_rectangular():
        raise ValueError('El sistema tiene que ser un arreglo rectangular.')
    if len(pasos[0])!=len(pasos[1]):
        raise ValueError('Esta representación requiere el mismo número de pasos por la izquierda y la
        ↪  derecha')

    A = data.fullcopy().subs(sust)
    previo, nuevos = (latex(data) if not TexPasosPrev else TexPasosPrev), []

    simplifica = lambda metodo,expresion: metodo(expresion)

    for  i,pasoDeEliminacionFilas in enumerate(pasos[0][::-1]):
        nuevos.append('\\xrightarrow' \
                + '[' + latex( (filtradopasos(pasoDeEliminacionFilas)).subs(sust) ) + ']' \
                + '{' + latex( (pasos[1][i]).subs(sust)                            ) + '}')
        nuevos.append(latex( simplifica(metodo, (( pasoDeEliminacionFilas & A & pasos[1][i] )).subs(sust)) ))

    return TexDesarrollo(previo, nuevos)
```

Estos procedimientos son para “mostrar” en los Jupyter notebooks los pasos de eliminación.

Representación de un proceso de eliminación dispElim, dispElimFyC y dispElimCF

```
def dispElim(self, pasos, TexPasosPrev=[], sust=[], metodo=factor):
    display(Math(rprElim(self, pasos, TexPasosPrev, sust, metodo)))

def dispElimFyC(self, pasos, TexPasosPrev=[], sust=[], metodo=factor):
    display(Math(rprElimFyC(self, pasos, TexPasosPrev, sust, metodo)))

def dispElimCF(self, pasos, TexPasosPrev=[], sust=[], metodo=factor):
    display(Math(rprElimCF(self, pasos, TexPasosPrev, sust, metodo)))
```

1.4.6. Desarrollos partidos en varias líneas, compactos o abreviados

El atributo `tex` contiene todo el proceso en una única fórmula, que con frecuencia no cabe en el ancho de una página. La función `desarrollo` devuelve una variante de dicho código LaTeX sin repetir los cálculos, pues se limita a recomponer el código ya escrito. Admite tanto un objeto con atributo `tex` (el resultado de `elim`, `Elim`, `SEL`, `Homogenea`, `InvMat`, `Determinante`, etc.) como directamente una cadena con el código.

Argumento	Efecto
`por_linea`	número de sistemas (matrices) en cada línea
`compacto`	reduce la separación entre columnas (`True`: 2pt; un número: esa separación)
`pasos`	lista de los pasos que se muestran (además del sistema inicial y del final)
`omitidos`	rótulo de la flecha que resume pasos omitidos: `'transformaciones'` o `'puntos'`

Sin argumentos devuelve el atributo `tex` sin modificar. Por ejemplo, en un documento con PythonTeX:

```
\[ \py{desarrollo(SEL(S, B), por_linea=2, compacto=True)} \]
```

Con `por_linea` el resultado es un entorno `aligned` (del paquete `amsmath`); todas las líneas contienen el mismo número de sistemas (salvo quizá la última) y, excepto la primera, comienzan con una flecha. Hasta la versión 0.5.5 `por_linea` contaba pasos (flechas), y la primera línea llevaba un sistema más que las demás. La opción `compacto` usa la longitud `\arraycolsep` de LaTeX, que los cuadernos Jupyter no interpretan (es una opción pensada para documentos). Con `pasos` cada flecha que sustituye a varios pasos omitidos se rotula con la composición de las transformaciones omitidas (los rótulos superiores se yuxtaponen en el orden en que se aplican y los inferiores en orden inverso, pues las transformaciones por la izquierda se componen hacia la izquierda). Si son demasiadas, con `omitidos='puntos'` el rótulo se reduce a unos puntos suspensivos. Los pasos se numeran desde 1 y los índices negativos cuentan desde el final (`-1` es el último paso, que siempre se muestra).

La recomposición requiere separar el código en sus eslabones: los sistemas y las flechas que los unen. Las flechas del desarrollo son las únicas órdenes `\xrightarrow` que no están dentro de unas llaves, así que basta recorrer el código llevando la cuenta de las llaves abiertas.

Separación de un desarrollo en sistemas y flechas

```
def fin_del_argumento(tex, inicio):
    "Posición siguiente al cierre del argumento [...] o {...} que comienza en inicio"
    cierre, nivel, k = {'[':']', '{':'}'}[tex[inicio]], 0, inicio
    while True:
        if   tex[k] == '\\': k += 1
        elif tex[k] == '{':  nivel += 1
        elif tex[k] == '}':  nivel -= 1
        if nivel == 0 and tex[k] == cierre:
            return k + 1
        k += 1
```

```python
def trozos_del_desarrollo(tex):
    """Separa el código LaTeX de un desarrollo en la lista
    [sistema, flecha, sistema, ..., flecha, sistema]"""
    FLECHA = '\\xrightarrow'

    trozos, inicio, nivel, i = [], 0, 0, 0
    while i < len(tex):
        if nivel == 0 and tex.startswith(FLECHA, i):
            j = i + len(FLECHA)
            while j < len(tex) and tex[j] in '[{':       # rótulos de la flecha
                j = fin_del_argumento(tex, j)
            trozos += [tex[inicio:i], tex[i:j]]
            inicio  = i = j
        elif tex[i] == '\\':                             # otras órdenes (y llaves \{ \})
            i += 2
        else:
            nivel += (tex[i] == '{') - (tex[i] == '}')
            i += 1
    return trozos + [tex[inicio:]]

def flecha_resumen(flechas):
    """Flecha rotulada con la composición de las transformaciones de varias flechas"""
    def rotulos(flecha):
        "Rótulos [inferior, superior] de una flecha"
        j, encontrados = len('\\xrightarrow'), {'[':'', '{':''}
        while j < len(flecha):
            fin = fin_del_argumento(flecha, j)
            encontrados[flecha[j]] = flecha[j+1:fin-1]
            j = fin
        return [encontrados['['], encontrados['{']]

    inferiores, superiores = zip(*[rotulos(f) for f in flechas])
    inferior = ''.join(reversed(inferiores))   # por la izquierda: la última es la primera
    superior = ''.join(superiores)
    return '\\xrightarrow' + ('[' + inferior + ']' if inferior else '') + '{' + superior + '}'
```

Recomposición de un desarrollo

```python
def desarrollo(objeto, por_linea=None, compacto=False, pasos=None, omitidos='transformaciones'):
    """Código LaTeX de un proceso de eliminación con otra presentación

    Parámetros:
        objeto:    objeto con atributo tex (resultado de elim, Elim, SEL,
                   Homogenea, InvMat, Determinante...) o cadena con el código
        por_linea: número de sistemas (matrices) en cada línea. Por
                   defecto todo el desarrollo va en una única línea
        compacto:  si es True la separación entre columnas se reduce a 2pt;
                   si es un número, a dicho número de puntos (solo LaTeX)
        pasos:     lista de los pasos que se muestran además del sistema
                   inicial y el final (se numeran desde 1; los negativos
                   cuentan desde el final). Por defecto se muestran todos
        omitidos:  rótulo de la flecha que resume varios pasos omitidos:
                   'transformaciones' (por defecto) para la composición de
                   las transformaciones omitidas o 'puntos' para unos
                   puntos suspensivos

    Sin argumentos opcionales devuelve el código sin modificar.

    Ejemplos:
    >>> A = Matrix([ [0,2,2], [1,-2,-8], [1,1,1] ])
    >>> desarrollo(ElimGJ(A)) == ElimGJ(A).tex
    True
    >>> desarrollo(ElimGJ(A), por_linea=2).count('[1ex]')      # 6 matrices: 3 líneas
    2
    >>> desarrollo(ElimGJ(A), pasos=[]).count('xrightarrow')   # inicial y final
    1
    >>> desarrollo(ElimGJ(A), pasos=[], omitidos='puntos').count('cdots')
```

```
    1
    """
    tex    = objeto if isinstance(objeto, str) else objeto.tex
    trozos = tex.trozos if isinstance(tex, TexDesarrollo) else trozos_del_desarrollo(tex)
    sistemas, flechas = trozos[0::2], trozos[1::2]

    if pasos is not None:
        ultimo   = len(sistemas) - 1
        if any(not -ultimo <= p <= ultimo for p in pasos):
            raise ValueError('El desarrollo solo tiene ' + str(ultimo) + ' pasos')
        elegidos = sorted({0, ultimo} | {p % (ultimo + 1) for p in pasos})
        if omitidos not in ('transformaciones', 'puntos'):
            raise ValueError("omitidos debe ser 'transformaciones' o 'puntos'")
        resumen  = flecha_resumen if omitidos == 'transformaciones' else \
                   lambda flechas: '\\xrightarrow{\\ \\cdots\\ }'
        flechas  = [flechas[p] if q == p+1 else resumen(flechas[p:q])
                    for p, q in zip(elegidos, elegidos[1:])]
        sistemas = [sistemas[p] for p in elegidos]

    eslabones = [sistemas[0]] + [f + s for f, s in zip(flechas, sistemas[1:])]

    if por_linea is not None and por_linea < 1:
        raise ValueError('por_linea debe ser un entero positivo')

    if por_linea and len(eslabones) > por_linea:
        lineas = [''.join(eslabones[k:k+por_linea])
                  for k in range(0, len(eslabones), por_linea)]
        # el grupo vacío {} evita un error del paquete mathtools cuando
        # tras \begin{aligned} van seguidos un & y un entorno array
        tex = '\\begin{aligned}{}' + '\\\\[1ex]'.join('&' + l for l in lineas) + '\\end{aligned}'
    else:
        tex = ''.join(eslabones)

    if compacto:
        separacion = 2 if compacto is True else compacto
        tex = '{\\setlength{\\arraycolsep}{' + str(separacion) + 'pt}' + tex + '}'

    return tex
```

1.5. Representación de la clase `Sistema`

Necesitamos indicar a Python cómo representar los objetos de tipo `Sistema`. Los sistemas, son secuencias finitas de objetos que representaremos con corchetes, separando los elementos por “;”

$$\boldsymbol{v} = [v_1;\ \ldots;\ v_n;]$$

Si la lista es vacía, entonces se pintan unos corchetes `[ ]` sin “;” (por no haber elementos).

Definimos varios tipos de representación.

- La primera se muestra con la función `print()` o la función `str()` y está formada por caracteres ASCII. Es la que se ve en la línea de comandos. Entre corchetes muestra todos los elementos de `self.lista` separados por “puntos y comas” (`;`):

  Si, por ejemplo, el atributo `.corteSistema` (*“corta Sistema”*) indica que se separen las dos primeras componentes del sistema respecto de la última, esta representación pinta una barra vertical detrás de la segunda componente (véase la Sección 1.5.1).

```
def __str__(self):
```

```
    """ Muestra un Sistema en su representación python """
    pc = ';' if len(self.lista) else ''
    ln = [len(n) for n in particion(self.corteSistema,self.n)]
    return '[' + \
             ';|'.join(['; '.join([str(c) for c in s]) \
                      for s in [ self|i for i in particion(self.corteSistema, self.n)] ]) + pc + ']'
```

- La segunda forma de representación se muestra con la función `repr()` y también está formada por caracteres ASCII. Se parece a la anterior, pero indica explícitamente que el objeto es un `Sistema` y no muestra ninguna barra vertical que separe el sistema en sublistas.

```
def __repr__(self):
    """ Muestra un Sistema en su representación python """
    pc = ';' if len(self.lista) else ''
    return 'Sistema([' + '; '.join( repr (e) for e in self ) + pc + '])'
```

- La representación `latex` (LaTeX) es similar a la primera representación (`str`), pues también muestra barras verticales que separan la lista de elementos en sub-listas si el atributo `.corteSistema` así lo indica. La única diferencia es que los elementos aparecen con su representación `latex` (cuando la tienen).

  Es la representación que los Notebooks de Jupyter emplean por defecto (y también es usada por Emacs (Scimax) mediante los dos últimos métodos que aparecen más abajo).

  Llamamos a la representación `latex` con los métodos `display()` y `pinta()`.

```
def latex(self):
    """ Construye el comando LaTeX para representar un Sistema """
    if not self:
        return r'\left[\ \right]'
    else:
        pc = ';' if len(self) else r'\ '
        ln = [len(i) for i in particion(self.corteSistema, len(self))]
        return \
            r'\left[ \begin{array}{' + '|'.join([n*'c' for n in ln])  + '}' + \
            r';& '.join([latex(e) for e in self]) + pc + \
            r'\end{array} \right]'
```

- Jupyter llama al método `__repr_html__` (que a su vez llama al método general `html` (véase 17.2) para mostrar la representación `latex` de los objetos en el navegador.

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

- Es posible trabajar con los Jupyter Notebooks dentro de Emacs con la configuración Scimax. Para poder visualizar la representación LaTeX dentro del editor, es necesario generar las imágenes en ficheros auxiliares `png`. Para ello, definimos un par de representaciones adicionales usadas en los Notebooks con Emacs.

```
def _repr_latex_(self):
    """ Representación para el entorno jupyter en Emacs """
```

```
    return '$'+self.latex()+'$'

def _repr_png_(self):
    """ Representación png para el entorno jupyter en Emacs """
    try:
        expr = '$'+self.latex()+'$'
        workdir = tempfile.mkdtemp()
        with open(join(workdir, 'borrame.png'), 'wb') as outputfile:
            sympy.preview(expr, viewer='BytesIO', outputbuffer=outputfile)
        return open(join(workdir, 'borrame.png'),'rb').read()
    except Exception:
        return '$'+self.latex()+'$'
```

Método para establecer los índices donde poner marcas de corte de un Sistema

Para separar visualmente distintas partes de un sistema es necesario indicar los índices de los componentes tras lo que se mostrará una barra vertical. Para especificar dichos índices se llama al método `csis()` (que usaremos para visualmente *cortar un sistema*, por ejemplo para separar visualmente las columnas de una matriz).

```
def csis(self, conjuntoIndices={}):
    """Modifica el atributo corteSistema para insertar lineas entre
    determinados elementos del sistema

    """
    self.corteSistema = set(conjuntoIndices) if conjuntoIndices else {0}
    return self
```

1.5.1. Ejemplo de representación de un `Sistema`

Veamos la representación `str` de un sistema `A` con tres elementos:

```
A = Sistema([ 2, fracc(a,b), sympy.sqrt(5), ])
print( A )
```

```
[2; a/b; sqrt(5);]
```

Si incluimos una separación visual detrás de la segunda componente de `A` su representación `str` es:

```
A.csis({2})
print( A )
```

```
[2; a/b;|sqrt(5);]
```

Sin embargo, la representación `repr` no muestra la barra vertical de separación:

```
repr( A )
```

```
Sistema([2; a/b; sqrt(5);])
```

La barra vertical de separación visual sí se muestra en la representación `latex` (es la representación empleada por defecto en los Notebooks de Jupyter y en el material del curso de Álgebra Lineal):

```
pinta(A)
```

$$\left[\ 2;\quad \frac{a}{b};\ \middle|\ \sqrt{5};\ \right]$$

1.5.2. Ejemplo de representación con un `Sistema` más complejo

Únicamente cuando un `Sistema` tiene una estructura muy sencilla las tres formas de representación de `Sistemas` son prácticas. Por ejemplo, la representación `str` se ve mal cuando el sistema contiene objetos que son complicados de representar (por ejemplo matrices dentro de otros sistemas). Por otra parte, aunque la representación `repr` indica claramente cuál es el tipo de cada objeto, es difícil ver qué contiene cada uno de los objetos. La representación `latex` es, con diferencia, la más fácil de interpretar de un simple vistazo.

En el siguiente ejemplo, con `.csis({1})` indicamos que el primer elemento del sistema debe estar visualmente separado del resto.

```
a,b,c = sympy.symbols('a b c')
vv = Vector([1,2,3])
Z = Sistema( [ Vector([6,8,10],rpr='h').csis({2}), vv , 1492] )
ZZ = Sistema([ Z, 'Hola', Matrix([Vector([0,sympy.pi,fracc(a,2)]),Vector([0,0,0])])  ]).csis({1})
print(ZZ)
repr(ZZ)
ZZ
```

```
[[(6, 8,|10,); (1, 2, 3,); 1492;];|Hola; | 0 0|\n| pi 0|\n|a/2 0|;]
```

```
Sistema([Sistema([Vector([6, 8, 10]); Vector([1, 2, 3]); 1492;]); 'Hola'; Matrix([Vector([0, pi,
a/2]), Vector([0, 0, 0])]);])
```

$$\left[\ \left[\ \left(\ 6,\quad 8,\ \middle|\ 10,\ \right);\quad \begin{pmatrix}1\\2\\3\end{pmatrix};\quad 1492;\ \right];\ \middle|\ \text{Hola};\quad \begin{bmatrix}0 & 0\\ \pi & 0\\ \frac{a}{2} & 0\end{bmatrix};\ \right]$$

1.6. La clase `Sistema` completa

Definición de la clase Sistema

```
class Sistema:
    <<Texto de ayuda de la clase Sistema>>
    <<Inicialización de la clase Sistema>>
    <<Métodos de la clase Sistema para que actúe como si fuera una list de Python>>
    <<Método de la clase Sistema para concatenar dos sistemas>>
    <<Método que junta una lista de Sistemas en un único Sistema>>
    <<Método que amplía la lista de un Sistema con nuevos elementos>>
    <<Método que vectoriza un Sistema>>
    <<Sustitución de variables simbólicas>>
    <<Simplificación de expresiones simbólicas en la lista de un Sistema>>
    <<Sustitución de un símbolo por otro símbolo o valor en un Sistema>>
    <<Otros métodos de la clase Sistema>>
    <<Métodos que devuelven SubEspacios>>
    <<Método de resolución de sistema de Ecuaciones lineales>>
    <<Operador selector por la derecha para la clase Sistema>>
    <<Operador selector por la izquierda para la clase Sistema>>
    <<Suma y diferencia de Sistemas>>
    <<Producto de un Sistema por un escalar a su izquierda>>
    <<Opuesto de un Sistema>>
    <<Producto de un Sistema por un escalar un Vector o una Matrix a su derecha>>
    <<Transformaciones elementales de los elementos de un Sistema>>
    <<Transformaciones elementales por la izquierda de un Sistema>>
    <<Reducción por eliminacion>>
    <<Eliminación>>
    <<formas escalonadas>>
    <<Métodos de representación de la clase Sistema>>
```

Capítulo 2

La subclase `BlockV`

En el curso de Álgebra Lineal empleamos arreglos rectangulares de objetos (principalmente las matrices). En NAcAL, la clase de los arreglos rectangulares de objetos son los `BlockM` ("Block Matrix"). Son `Sistemas` formados por una lista de `Sistemas` de la misma longitud y que representamos verticalmente para formar las columnas del arreglo rectangular.

En consecuencia, los elementos de un `BlockM` son *sistemas con una representación vertical*. Así pues, en este capítulo se define una primera subclase de la clase `Sistema` cuya representación difiere de la de los `Sistemas` genéricos. A estos subsistemas los denominamos `BlockV`.

Por defecto, los `BlockV` tienen representación `latex` vertical (opcionalmente podremos representarlos horizontalmente). Para distinguirlos de los `Sistemas` genéricos, su lista de componentes está encerrada entre paréntesis (en lugar de corchetes); y si se representan horizontalmente, tras de cada elemento aparece una *coma* (`,`) en lugar de un *punto y coma* (`;`). Así que para instanciar un `BlockV`, además del argumento con la lista de elementos del sistema, disponemos de un segundo argumento opcional (`rpr`) que indica si queremos una representación vertical (por defecto es la que se usará si no se indica nada) u horizontal. En todo lo demás, un `BlockV` es como un `Sistema` genérico.

2.1. Implementación

2.1.1. Texto de ayuda

Texto de ayuda de la subclase BlockV

```
"""BlockV es un Sistema que se puede representar verticalmente.

Se puede instanciar con una lista, tupla o otro Sistema. Si al
instanciar un BlockV la lista, tupla o sistema solo contiene números
el objeto obtenido es un Vector (subclase de BlockV).

El atributo 'rpr' indica si la representación latex debe mostrar el
sistema en disposición vertical (por defecto) u horizontal.

Parámetros:
    sis   (list, tuple, Sistema): Lista, tupla o Sistema de objetos.
    rpr   (str): Para su representación latex (en vertical por defecto).
                 Si rpr='h' se representa en forma horizontal.

Atributos de la subclase:
    rpr   (str): modo de representación en Jupyter.

Atributos heredados de la clase Sistema:
```

```
    lista              (list): list con los elementos.
    n                  (int) : número de elementos de la lista.
    corteSistema (set) : Conjunto de índices donde pintar
                                 separaciones visuales

Ejemplos:
>>> # Instanciación a partir de una lista, tupla o Sistema de números
>>> BlockV( [1,'abc',(2,)] )            # con una lista
BlockV([1, 'abc', (2,)])
>>> BlockV( (1,'abc',(2,)) )            # con una tupla
BlockV([1, 'abc', (2,)])
>>> BlockV( Sistema( [1,'abc',(2,)] ) ) # con un Sistema
BlockV([1, 'abc', (2,)])
>>> BlockV( BlockV ( [1,'abc',(2,)] ) ) # a partir de otro BlockV
BlockV([1, 'abc', (2,)])

>>> BlockV( [1,2,3] )                   # con una lista de números
Vector([1, 2, 3])
"""
```

2.1.2. Método de inicialización:

`BlockV` es una subclase `Sistema`. Se inicia con el método: `def __init__(self, arg, rpr='columna')`.

- La clase `BlockV` se instancia con dos argumentos.
  1. `arg` es obligatorio y debe ser una lista, tupla o `Sistema`.
  2. `rpr` es opcional e indica si queremos que la representación `latex` sea en forma horizontal o en vertical. Por defecto la representación es vertical. Para una disposición horizontal `rpr` de ser la cadena de caracteres `h`, es decir, `rpr='h'`.
- Con `super().__init__(arg)` la subclase `BlockV` hereda los métodos y atributos de la clase `Sistema`. En consecuencia `BlockV` tendrá los atributos `lista`, `n` y `corteSistema` así como todos los métodos definidos para la clase `Sistema`.
- El atributo `rpr` tomará el valor indicado al instanciar la clase (`'columna'` por defecto). Y el atributo `n` será igual al número de elementos del sistema (su longitud).
- Por último, un `BlockV` cuya lista tan solo contiene números es un vector de $\mathbb{R}^n$.

  Consecuentemente, cuando todos los elementos de `arg` son números (Véase la Sección 17.6) el objeto que se crea es un `Vector` (una subclase de `BlockV` que solo contiene números).

Inicialización de la subclase BlockV

```
def __init__(self, arg, rpr='columna'):
    """Inicializa un BlockV con una lista, tupla o Sistema"""
    super().__init__(arg)
    self.rpr  =  rpr
    self.n  = len(self)

    if all([es_numero(e) for e in arg]):
        self.__class__ = Vector
    #if all([isinstance(e, Vector) for e in arg]) and self.es_arreglo_rectangular():
    #    self.__class__ = Matrix
    #    try:
    #        self.m = (self|1).n
    #    except:
    #        self.m = 0
```

```
    #    self.corteElementos = set()
```

2.2. Métodos de la subclase `BlockV`

```
def repre(self, rpr='h'):
    """ Devuelve el BlockV con la representación indicada """
    v = self.fullcopy()
    v.rpr = rpr
    return v
```

2.3. Representación de la subclase `BlockV`

Un `BlockV` es una secuencia finita de objetos; es decir, un `Sistema`. La única diferencia respecto de un `Sistema` genérico es su representación. Por tanto solo necesitamos redefinir las representaciones `str`, `repr` y `latex` de esta subclase particular de `Sistema`.

```
def __repr__(self):
    """ Muestra el BlockV en su representación Python """
    return 'BlockV(' + repr(self.lista) + ')'

def __str__(self):
    """ Muestra el BlockV en su representación Python """
    pc = ',' if len(self.lista) else ''
    ln = [len(n) for n in particion(self.corteSistema,self.n)]
    return '(' + \
        ',|'.join([', '.join([str(c) for c in s]) \
                for s in [ self|i for i in particion(self.corteSistema, self.n)]]) + \
        pc + ')'

def latex(self):
    """ Construye el comando LaTeX para representar un BlockV """
    if bool(self.corteSistema):
        pc = ',' if len(self) else r'\ '
        ln = [len(n) for n in particion(self.corteSistema,self.n)]
        if self.rpr == 'h' or self.n==1:
            return \
                r'\left( \begin{array}{' + '|'.join([n*'c' for n in ln])  + '}' + \
                r',& '.join([latex(e) for e in self]) + pc + \
                r'\\ \end{array} \right)'
        else:
            salto = salto_de_fila(self)
            return \
                r'\left( \begin{array}{c}' + \
                (salto + r' \hline ').join([salto.join([latex(c) for c in e]) \
                    for e in [ self|i for i in particion(self.corteSistema, self.n)]]) + \
                r'\\ \end{array} \right)'
    else:
        if not self:
            return r'\left(\ \right)'

        if self.rpr == 'h' or self.n==1:
            return r'\begin{pmatrix}' + \
                ',& '.join([latex(e) for e in self]) + \
                r',\end{pmatrix}'
        else:
            return r'\begin{pmatrix}' + \
```

```
            (salto_de_fila(self) + ' ').join([latex(e) for e in self]) + \
            r'\end{pmatrix}'
```

2.3.1. Ejemplo de representación de un `BlockV`

Veamos la representación de un `BlockM` cuya lista contiene una matriz dos por tres y tres números.

```
BV = BlockV([Matrix([[1,2,3],[4,5,6]]), 1,0,0])
```

La representación `str` es no es práctica en este caso, pues su elemento `Matrix` necesita de un salto de línea, por lo que la visualización algo deficiente: `(|1 2 3|\n|4 5 6|, 1, 0, 0,)`

La representación `repr` es mejor, pero resulta difícil leer qué objetos son elementos de otros: `BlockV([Matrix([Vector([1, 4]), Vector([2, 5]), Vector([3, 6])]), 1, 0, 0])`

La representación `latex` es la mejor: $\begin{pmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \\ 1 \\ 0 \\ 0 \end{pmatrix}$

y también la podemos usar en horizontal: $\begin{pmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, & 1, & 0, & 0, \end{pmatrix}$

2.4. La clase `BlockV` completa

Definición de la subclase BlockV

```
class BlockV(Sistema):
    <<Texto de ayuda de la subclase BlockV>>
    <<Inicialización de la subclase BlockV>>
    <<Métodos de la subclase BlockV>>
    <<Métodos de representación de la subclase BlockV>>
```

Capítulo 3

La subclase `Vector`

El curso de Álgebra Lineal define un vector de $\mathbb{R}^n$ del siguiente modo

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

y se indica que los vectores se representan entre paréntesis tanto horizontal como verticalmente. Por tanto hay que redefinir la representación de la clase `Vector` para que los `Vectores` no sean representados como `Sistemas` genéricos, sino a la manera de los vectores.

Pero esto ya se hace en el capítulo anterior con los `BlockV`. Como un `Vector` es un `BlockV` que solo contiene números, lo más sencillo es definir la clase `Vector` como una subclase `BlockV` que solo contiene números. Así solo necesitamos redefinir la representación `repr` para que indique específicamente que se trata de un `Vector`, ya que la subclase `Vector` hereda el resto de métodos y atributos de la clase `Sistema` y la subclase `BlockV`.

3.1. Implementación de los vectores de $\mathbb{R}^n$ en la subclase `Vector`

3.1.1. Texto de ayuda

Texto de ayuda de la subclase Vector

```
"""Clase para los Sistemas de números.

Sólo se puede instanciar con una lista, tupla o Sistema de objetos
int, float o sympy.Basic. Si se instancia con un Vector se crea una
copia del mismo.

El atributo 'rpr' indica si, en la representación latex, el vector
debe ser escrito en vertical (por defecto) o en horizontal.

Parámetros:
    sis (list, tuple, Sistema): Lista, tupla o Sistema de objetos
        de tipo int, float o sympy.Basic.
    rpr (str): Para su representación latex (en vertical por defecto).
        Si rpr='h' el vector se representa en horizontal.

Atributos heredados de la subclase BlockV::
    rpr   (str)    : modo de representación en Jupyter.

Atributos heredados de la clase Sistema:
    lista             (list): list con los elementos.
    n                 (int) : número de elementos de la lista.
    corteSistema (set) : Conjunto de índices donde pintar
                              separaciones visuales
```

```
Ejemplos:
>>> # Instanciación a partir de una lista, tupla o Sistema de números
>>> Vector( [1,2,3] )           # con lista
Vector([1, 2, 3])
>>> Vector( (1,2,3) )           # con tupla
Vector([1, 2, 3])
>>> Vector( Sistema( [1,2,3] ) )# con Sistema
Vector([1, 2, 3])
>>> Vector( Vector ( [1,2,3] ) )# a partir de otro Vector
Vector([1, 2, 3])
"""
```

3.1.2. Método de inicialización:

La clase se inicia con el método: `def __init__(self, arg, rpr='columna')`.

- La clase `Vector` emplea dos argumentos. El primero (`arg`) es una lista, tupla o `Sistema` de objetos tipo `int`, `float` o `sympy.Basic`. Cuando `arg` es un `Vector` se obtiene una copia. El segundo argumento (`rpr`) es opcional e indica si queremos que la representación `latex` sea en forma horizontal o en vertical (Véase la subclase `BlockV`).
- Python mostrará el texto de ayuda sobre el método `__init__` con: `help Vector.__init__`
- Se verifica que `arg` es una secuencia de números (Sección 17.7). Si no lo es obtenemos un error.
- Con `super().__init__(arg)` la subclase `Vector` hereda los métodos y atributos de la clase `BlockV` (por tanto, `Vector` tendrá un atributo `lista`, así como el resto de atributos y todos los métodos definidos para la clase `Sistema` y la subclase `BlockV`).

Inicialización de la subclase Vector

```
def __init__(self, arg, rpr='columna'):
    """Inicializa Vector con una lista, tupla o Sistema de números"""
    if not es_ristra_de_numeros(arg):
        raise ValueError('no todos los elementos son números o parámetros!')

    super().__init__(arg, rpr)
```

3.2. Métodos específicos de la subclase `Vector`

Aquí se definen algunos métodos específicos de la subclase `Vector`. El primero calcula la norma euclídea de un `Vector` de $\mathbb{R}^n$, es decir, la raíz cuadrada del producto punto del vector por si mismo.

El segundo usa dicha norma para devolver un múltiplo con norma uno de cualquier un vector no nulo.

Normalización de un Vector

```
def norma(self):
    """Devuelve la norma de un vector"""
    return sympy.sqrt(self*self)

def normalizado(self):
    """Devuelve un múltiplo de norma uno si el vector no es nulo"""
    if self.es_nulo(): raise ValueError('Un vector nulo no se puede normalizar')
```

```
    return self * fracc(1,self.norma())
```

El tercer método devuelve una `Matrix` diagonal cuya diagonal principal es igual a `Vector`.

Creación de una Matrix diagonal a partir de un Vector

```
def diag(self):
    """Crea una Matrix diagonal cuya diagonal es self"""
    return Matrix([a*(I(self.n)|j) for j,a in enumerate(self, 1)])
```

3.3. Representación de la clase `Vector`

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

Los vectores, son secuencias finitas de números que escribimos entre paréntesis en horizontal

$$\boldsymbol{v} = (v_1, \ldots, v_n)$$

o en vertical

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

Esta forma de representación ya se ha establecido para la subclase `BlockV`, por lo que no es necesario volver a programarla. Tan solo re-especificamos la representación de tipo `repr` para que explícitamente indique que el objeto es un `Vector`.

Métodos de representación de la subclase Vector

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

3.4. La clase `Vector` completa

Definición de la subclase Vector

```
class Vector(BlockV):
    <<Texto de ayuda de la subclase Vector>>
    <<Inicialización de la subclase Vector>>
    <<Normalización de un Vector>>
    <<Creación de una Matrix diagonal a partir de un Vector>>
    <<Métodos de representación de la subclase Vector>>
```

Capítulo 4

Las subclases `V0` y `V1`

La clase `V0` corresponde a vectores nulos. La clase `V1` corresponde a vectores constantes cuyas componentes son todas iguales a 1. En ambos casos se instancian con un parámetro que indica el número de componentes y, opcionalmente, el parámetro `rpr` para indicar si la representación `latex` es en forma horizontal (`'h'`) o vertical.

Definición de las subclases V0 y V1

```
class V0(Vector):
    """Clase para los Vectores nulos"""
    def __init__(self, n, rpr = 'columna'):
        """Inicializa un vector nulo de n componentes

        V0 se inicializa con

        1) un entero que indica el  número de componentes nulas

        2) la cadena de texto rpr. Si rpr = 'h' la representación
        es horizontal (en otro caso es vertical)

        """
        super().__init__([0]*n, rpr)
        self.__class__ = Vector

class V1(Vector):
    """Clase para los Vectores constantes 1"""
    def __init__(self, n, rpr = 'columna'):
        """Inicializa un vector uno de n componentes

        V1 se inicializa con

        1) un entero que indica el  número de componentes nulas

        2) la cadena de texto rpr. Si rpr = 'h' la representación
        es horizontal (en otro caso es vertical)

        """
        super().__init__([1]*n, rpr)
        self.__class__ = Vector
```

Capítulo 5

La subclase `Polinomio`

Los polinomios de una variable forman un espacio vectorial: se pueden sumar y multiplicar por números. La subclase `Polinomio` permite que los elementos de un `Sistema` sean polinomios, de modo que la eliminación y las clases `Homogenea` y `SEL` operen con ellos igual que con vectores de $\mathbb{R}^n$ o con matrices.

Un `Polinomio` es un `Vector` cuyos componentes son los coeficientes, ordenados de menor a mayor grado; es decir, sus coordenadas respecto de la base $1,\ x,\ x^2, \ldots$ Lo único que cambia es su representación, que es la habitual de un polinomio (también escrito de menor a mayor grado). Consecuentemente, el *pivote* de un polinomio no nulo (su primer componente no nulo) es el coeficiente de su término de menor grado, que es el primero que aparece escrito.

No basta con usar las expresiones simbólicas de Sympy como elementos de un `Sistema`, pues para este módulo dichas expresiones son *números*: la eliminación tomaría cada polinomio como un escalar por el que se puede dividir. Sympy solo se usa para leer los coeficientes de una expresión y para escribir los monomios.

5.1. Implementación

Se instancia con una expresión de Sympy (o un número), o con la lista de coeficientes. La variable se deduce de la expresión; solo hay que indicarla cuando la expresión contiene más de un símbolo (porque los coeficientes son simbólicos) o cuando, instanciando con una lista de coeficientes, no se quiere usar la variable x.

Texto de ayuda de la clase Polinomio

```
"""Clase para los polinomios de una variable

Un Polinomio es un Vector cuyos componentes son los coeficientes del
polinomio ordenados de menor a mayor grado.

Parámetros:
    arg: expresión de sympy (o número) polinómica en una variable, o
        lista, tupla o Sistema con los coeficientes (de menor a mayor
        grado)
    rpr (str): modo de representación como elemento de un sistema
    variable (sympy.Symbol): variable del polinomio. Por defecto es el
        único símbolo de la expresión; o x si no hay ninguno

Atributos:
    variable (sympy.Symbol): la variable del polinomio
```

```
    lista, n: los coeficientes y su número (heredados de Sistema)

Ejemplos:
>>> x = sympy.symbols('x')
>>> Polinomio( (1+x)**2 )
Polinomio(1 + 2*x + x**2)
>>> Polinomio( [1,2,1] )
Polinomio(1 + 2*x + x**2)
>>> Polinomio( (1+x)**2 ).lista
[1, 2, 1]
>>> 2*Polinomio(1 + x) + Polinomio(x**2)
Polinomio(2 + 2*x + x**2)
>>> a = sympy.symbols('a')
>>> Polinomio( a + x, variable=x )
Polinomio(a + x)
"""
```

Inicialización de la clase Polinomio

```
def __init__(self, arg, rpr='columna', variable=None):
    """Inicializa un Polinomio con una expresión o con sus coeficientes"""
    if isinstance(arg, Polinomio) and variable is None:
        variable = arg.variable

    if es_ristra(arg):
        coeficientes = list(arg)
    elif es_numero(arg):
        expresion = sympy.sympify(racional(arg))
        if variable is None:
            simbolos = sorted(expresion.free_symbols, key=str)
            if len(simbolos) > 1:
                raise ValueError('Hay más de un símbolo: indique cuál es la variable')
            variable = simbolos[0] if simbolos else sympy.Symbol('x')
        if not expresion.is_polynomial(variable):
            raise ValueError('La expresión no es un polinomio en ' + str(variable))
        coeficientes = sympy.Poly(expresion, variable).all_coeffs()[::-1]
    else:
        raise ValueError('El argumento debe ser una expresión o una lista de coeficientes')

    super().__init__(coeficientes, rpr)
    self.__class__ = Polinomio
    self.variable  = variable if variable is not None else sympy.Symbol('x')

def copy(self):
    """Genera un Polinomio copiando los coeficientes y la variable de otro"""
    return Polinomio(self.lista, self.rpr, self.variable)
```

(La penúltima línea de `__init__` es necesaria porque al inicializar un `BlockV` con una lista de números el objeto pasa a ser de la clase `Vector`.)

5.2. Operaciones

La suma, la resta y el producto por un número son las de la clase `Sistema` (operan componente a componente). Pero la clase `Sistema` solo suma sistemas con el mismo número de elementos, y dos polinomios pueden tener distinto grado; así que antes de sumar o restar se completa con ceros la lista de coeficientes del más corto. La igualdad compara los polinomios y no las listas (pues $1 + x$ y $1 + x + 0x^2$ son el mismo polinomio).

El producto de dos polinomios es su producto como polinomios (y no el producto punto de sus listas de coeficientes, que es lo que se heredaría de la clase `Vector`).

Operaciones de la clase Polinomio

```
def expresion(self):
    """Expresión de sympy correspondiente al Polinomio"""
    return sympy.expand(sum(c*self.variable**i for i, c in enumerate(self.lista)))

def grado(self):
    """Grado del polinomio (el del polinomio nulo es menos infinito)"""
    no_nulos = [i for i, c in enumerate(self.lista) if not es_cero(c)]
    return no_nulos[-1] if no_nulos else -sympy.oo

def completado(self, n):
    """Polinomio con n coeficientes (se añaden ceros en los grados altos)"""
    completo       = self.fullcopy()
    completo.lista = self.lista + [sympy.S.Zero]*(n - len(self))
    completo.n     = len(completo.lista)
    return completo

def igualados(self, other):
    """Los dos polinomios con el mismo número de coeficientes"""
    if not isinstance(other, Polinomio) or self.variable != other.variable:
        raise ValueError('Solo se operan Polinomios de la misma variable')
    n = max(len(self), len(other))
    return self.completado(n), other.completado(n)

def __add__(self, other):
    """Suma de dos polinomios"""
    p, q = self.igualados(other)
    return Sistema.__add__(p, q)

def __sub__(self, other):
    """Resta de dos polinomios"""
    p, q = self.igualados(other)
    return Sistema.__sub__(p, q)

def __eq__(self, other):
    """Indica si dos polinomios son iguales"""
    return isinstance(other, Polinomio) and self.variable == other.variable \
        and es_cero(self.expresion() - other.expresion())

def __mul__(self, x):
    """Producto por un número o por otro Polinomio"""
    if isinstance(x, Polinomio):
        if self.variable != x.variable:
            raise ValueError('Solo se operan Polinomios de la misma variable')
        return Polinomio(sympy.expand(self.expresion()*x.expresion()), self.rpr, self.variable)
    if es_numero(x):
        return x*self
    raise ValueError('Un Polinomio solo se multiplica por números o por Polinomios')
```

5.3. Representación

Los monomios se escriben de menor a mayor grado, y se omiten los de coeficiente nulo.

Representación de la clase Polinomio

```
def monomios(self, escribe, por, parentesis):
    """Cadena con la suma de los monomios no nulos, de menor a mayor grado

    escribe es la función que escribe una expresión; por es el signo del
    producto y parentesis los que rodean a un coeficiente que es una suma"""
    cadena = ''
    for i, c in enumerate(self.lista):
        if es_cero(c):
            continue
        c = sympy.sympify(c)
        if c.is_number or i == 0:
```

```
            texto = escribe(c*self.variable**i)
        else:
            coeficiente = parentesis[0] + escribe(c) + parentesis[1] if c.is_Add else escribe(c)
            texto       = coeficiente + por + escribe(self.variable**i)
        if not cadena:
            cadena = texto
        elif texto.startswith('-'):
            cadena += ' - ' + texto[1:].lstrip()
        else:
            cadena += ' + ' + texto
    return cadena if cadena else '0'

def __str__(self):
    """Muestra el Polinomio como texto"""
    return self.monomios(str, '*', '()')

def __repr__(self):
    """Muestra el Polinomio en su representación Python"""
    return 'Polinomio(' + str(self) + ')'

def latex(self):
    """Construye el comando LaTeX para representar un Polinomio"""
    return self.monomios(sympy.latex, ' ', ('\\left(', '\\right)'))
```

5.4. La clase `Polinomio` completa

Definición de la subclase Polinomio

```
class Polinomio(Vector):
    <<Texto de ayuda de la clase Polinomio>>
    <<Inicialización de la clase Polinomio>>
    <<Operaciones de la clase Polinomio>>
    <<Representación de la clase Polinomio>>
```

Capítulo 6

La subclase `BlockM`

Un `BlockM` es un sistema de `BlockVs` de la misma longitud, es decir, es un arreglo rectangular de objetos. A sus elementos los llamaremos *columnas* del `BlockM`.

6.1. Implementación

6.1.1. Texto de ayuda

Texto de ayuda de la clase BlockM

```
"""Clase para arreglos rectangulares de objetos.

Sistema formado por BlockVs con el mismo número de componentes. Se
instancia con: 1) una lista, tupla o Sistema de BlockVs (serán sus
columnas); 2) una lista, tupla o Sistema de listas, tuplas o Sistemas
con la misma longitud (serán sus filas); 3) otro BlockM (se obtendrá
una copia).

Parámetros:
    arg (list, tuple, Sistema): Lista, tupla o Sistema de BlockVs
        (con identica longitud); o de listas, tuplas o Sistemas (con
        identica longitud).

Atributos:
    m              (int) : número de filas
    corteElementos (set) : Conjunto de índices donde pintar
                           separaciones visuales entre filas

Atributos heredados de la clase Sistema:
    lista              (list): list con los elementos.
    n                  (int) : número de elementos de la lista.
    corteSistema (set) : Conjunto de índices donde pintar
                                separaciones visuales

Ejemplos:
>>> # Crear un BlockM a partir de una lista de Vectores o BlockVs:
>>> a = BlockV( ['Hola',2,3] ); b = Vector( [1,2,5] ); c = Vector( [1,2,7] )
>>> BlockM( [a,b,c] )
BlockM([BlockV(['Hola', 2, 3]), Vector([1, 2, 5]), Vector([1, 2, 7])])
>>> # Crear una BlockM a partir de una lista de listas, tuplas o Sistemas
>>> A = BlockM([('Hola',1,1),[2,2,2],Vector([3,5,7])])

BlockM([BlockV(['Hola', 2, 3]), Vector([1, 2, 5]), Vector([1, 2, 7])])

"""
```

6.1.2. Método de inicialización:

Inicialización de la subclase BlockM

```
def __init__(self, arg):
    """Inicializa un BlockM con una

    1) lista, tupla o Sistema de BlockVs con el mismo número de
    elementos,

    2) tupla, lista o Sistema de tuplas, listas o Sistemas con el
    mismo número de elementos,

    3) con otra BlockM.

    """
    listaInicial = Sistema(arg).lista.copy()

    if isinstance(arg, BlockM):
        lista = arg.copy().lista

    elif all([(isinstance(elemento, BlockV) and len(elemento)==len(listaInicial[0])) for elemento in
    ↪  listaInicial]):
        lista = listaInicial.copy()

    elif listaInicial and not listaInicial[0]:
        lista = [BlockV([])]

    elif all([(es_ristra(elemento) and len(elemento)==len(listaInicial[0])) for elemento in listaInicial]):
        lista = (~BlockM([BlockV(i) for i in arg])).lista

    else:
        raise ValueError("""El argumento debe ser una lista de
        BlockVs o una lista, tupla o Sistema de listas, tuplas o
        sistemas con el mismo número de elementos!""")

    listaFinal = [BlockV( elemento, rpr = 'columna' )  for elemento in lista]

    super().__init__(listaFinal)

    try:
        self.m  = (self|1).n
    except Exception:
        self.m  = 0

    self.corteElementos = set()

    if all( [isinstance(e, Vector) for e in self] ): self.__class__ = Matrix
```

6.2. Métodos de la clase `BlockM`

6.2.1. Transposición de un BlockM

Operador transposición

```
def __invert__(self):
    """
    Devuelve la traspuesta de un BlockM.

    Ejemplo:
    >>> ~BlockM([ [1,2,3], [2,4,6] ])
    Matrix([Vector([1, 2, 3]), Vector([2, 4, 6])])
    """
    M = BlockM([ BlockV([v|f for v in self]) for f in range(1,self.m+1)])
    M.corteElementos, M.corteSistema = self.corteSistema, self.corteElementos
```

```
    return M
```

6.2.2. Comprobación de que las filas o columnas son de composición homogenea

Si queremos aplicar transformaciones elementales a las columnas o a las filas de un `BlockM` es necesario que las operaciones entre componentes tengan sentido. Los siguientes dos métodos nos indican si es posible realizar dichas operaciones por columnas (si éstas son homogéneas) o por filas.

Comprobación de que las filas o columnas son de composición homogenea

```
def columnas_homogeneas(self):
    """Indica si las columnas contienen objetos del mismo tipo y longitud"""
    return self.es_de_composicion_y_longitud_uniforme()

def filas_homogeneas(self):
    """Indica si las filas contienen objetos del mismo tipo y longitud"""
    return (~self).es_de_composicion_y_longitud_uniforme()
```

6.2.3. Apilado de `BlockMs`

Creamos un `BlockM` apilando `BlockMs` uno encima de otro si tienen el mismo número de elementos.

Apila una lista de Sistemas con el mismo número de elementos un BlockM

```
def apila(self, lista, marcasVisuales = False):
    """Apila una lista o tupla de BlockMs con el mismo número de elementos
    (columnas) en un BlockM concatenando los respectivos elementos

    """
    apila_dos = lambda x, other, marcasVisuales=False: ~((~x).concatena(~other, marcasVisuales))
    apila = lambda x: x[0] if len(x)==1 else apila_dos( apila(x[0:-1]), x[-1], marcasVisuales)
    BlockM_apilado = apila([self] + [s for s in CreaLista(lista)])
    return BlockM_apilado.cele(BlockM_apilado.corteElementos)
```

6.2.4. Operador selector por la izquierda

Texto de ayuda para el operador selector por la izquierda para la clase BlockM

```
"""Extrae la j-ésima fila de un BlockM en forma de BlockV; o crea un
BlockM cuyas filas corresponden a las filas indicadas en una tupla o
lista de índices (los índices comienzan por el número 1)

Parámetros:
    j (int, list, tuple, slice): Índice (o lista de índices) del
          elementos (o elementos) a seleccionar

Resultado:
          ?: Si j es int, devuelve la j-ésima fila del BlockM.
    Sistema: Si j es list, tuple o slice devuelve el BlockM cuyas
          filas son las filas indicadas en la lista, tupla o slice de
          índices.

Ejemplos:
>>> # Extrae la j-ésima fila del BlockM
>>> 1 | BlockM([['Hola', 2, 3], [1, 2, 5], [1, 2, 7]])
BlockV(['Hola', 2, 3])
>>> # Sistema formado por los elementos indicados en la lista (o tupla)
>>> [2,1] | BlockM([['Hola', 2, 3], [1, 2, 5], [1, 2, 7]])
BlockM([BlockV([1, 'Hola']), Vector([2, 2]), Vector([5, 3])])
```

```
>>> (2,1) | BlockM([['Hola', 2, 3], [1, 2, 5], [1, 2, 7]])
BlockM([BlockV([1, 'Hola']), Vector([2, 2]), Vector([5, 3])])

>>> # Sistema formado por los elementos indicados en el slice
>>> slice(1,3,2) | BlockM([['Hola', 2, 3], [1, 2, 5], [1, 2, 7]])
BlockM([BlockV(['Hola', 1]), Vector([2, 2]), Vector([3, 7])])

"""
```

Operador selector por la izquierda para la clase BlockM

```
def __ror__(self,i):
    <<Texto de ayuda para el operador selector por la izquierda para la clase BlockM>>
    if isinstance(i,int):
        return  BlockV( (~self)|i , rpr='h' )

    elif isinstance(i, (list,tuple,slice)):
        return ~BlockM( (~self)|i )
```

Aviso sobre el orden en que Python calcula los operadores

El selector se escribe con el operador `|` de Python, y Python decide en qué orden se calculan los operadores sin tener en cuenta lo que significan aquí. Dos consecuencias que conviene conocer:

- **Dos selectores seguidos por la izquierda.** En `2|3|A` Python calcula primero `2|3`, que es una operación entre números enteros cuyo resultado es 3. Así que `2|3|A` es `3|A`: la fila 3, sin ningún aviso. Para seleccionar la componente 2 de la fila 3 hay que escribir `2|(3|A)`, o bien `3|A|2`.
- **Sumas y restas.** Python calcula las sumas y las restas antes que el selector: `a|1 + b|1` se calcula como `a|(1 + b)|1`, que intenta sumar un número y un vector y produce un error. Hay que escribir `(a|1) + (b|1)`. Lo mismo ocurre con el producto por un número, que también se calcula antes: `2*A|1` es `(2*A)|1`.

6.3. Otros métodos de la clase `BlockM`

6.3.1. Extiende una `BlockM` a lo largo de la diagonal con una lista de `BlockMs`

Extiende una BlockM a lo largo de la diagonal con una lista de BlockMs

```
def extDiag(self, lista, c=False):
    "Extiende una BlockM a lo largo de la diagonal con una lista de BlockMs"
    lista = CreaLista(lista)
    if not all(isinstance(elemento, BlockM) for elemento in lista):
        raise ValueError('No es una lista de BlockMs')
    Ext_dos = lambda x, y: x.apila(M0(y.m,x.n),c).concatena(M0(x.m,y.n).apila(y,c),c)
    ExtDiag = lambda x: x[0] if len(x)==1 else Ext_dos( ExtDiag(x[0:-1]), x[-1] )
    return ExtDiag([self]+lista)
```

6.4. Transformaciones elementales de las filas de un `BlockM`

6.4.1. Texto de ayuda

Texto de ayuda de las transformaciones elementales de un BlockM

```
"""Transforma las filas de un BlockM
```

```
Atributos:
    operaciones (T): transformaciones a aplicar sobre las filas
                     de un BlockM A
Ejemplos:
>>> # Intercambia las filas 1 y 3
>>> T({1,3}) & Matrix([[1,2],[3,4],[5,6]])
Matrix([Vector([5, 3, 1]), Vector([6, 4, 2])])

>>> # Multiplica por 5 la fila 1
>>> T((5,1)) & Matrix([[1,2],[3,4],[5,6]])
Matrix([Vector([5, 3, 5]), Vector([10, 4, 6])])

>>> # Suma 5 veces la fila 2 a la fila 1
>>> T((5,2,1)) & Matrix([[1,2],[3,4],[5,6]])
Matrix([Vector([16, 3, 5]), Vector([22, 4, 6])])

>>> # Aplica la secuencia de transformaciones en el orden inverso al de la lista
>>> T([(5,2,1),(5,1),{1,3}]) & Matrix([[1,2],[3,4],[5,6]])
Matrix([Vector([40, 3, 1]), Vector([50, 4, 2])])

"""
```

6.4.2. Implementación de las transformaciones elementales de las filas

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 antes y que $\tau_1 \cdots \tau_k \mathbf{A} = \left((_{\tau_1 \cdots \tau_k} \mathbf{A})^\intercal \right)^\intercal = \left((\mathbf{A}^\intercal)_{(\tau_1 \cdots \tau_k)^\intercal} \right)^\intercal = \left((\mathbf{A}^\intercal)_{\tau_k \cdots \tau_1} \right)^\intercal$.

Transformaciones elementales de las filas de un BlockM

```
def __rand__(self, operaciones):
    <<Texto de ayuda de las transformaciones elementales de un BlockM>>
    for item in reversed(operaciones.abreviaturas):
        if isinstance(item, (set, tuple) ):
            self.lista = (~(~self & T(item))).lista.copy()

        elif isinstance(item, list):
            for k in item:
                ~T(k) & self

    return self
```

6.5. Representación de la clase BlockM

Método de representación repr de la subclase BlockM

```
def __repr__(self):
    """ Muestra un BlockM en su representación Python repr """
    return 'BlockM(' + repr(self.lista) + ')'
```

Métodos de representación str y LaTeX de las subclases Matrix y BlockM

```
def __str__(self):
    """ Muestra un BlockM en su representación Python str """
    ln  = [len(n) for n in particion(self.corteSistema,self.n)]
    car = max([len(str(e)) for c in self for e in c])

    def escribeFila(f,d=0):
        parte = lambda f,d=0: str(' '.join([str(e).rjust(d) for e in f]))
```

```
        s = '|'+'|'.join([parte([e for e in c],d) for c in [p|f for p in particion(self.corteSistema, self.n)]
        ↪  ])+'|'
        return s

    num_guiones = len(escribeFila(1|self, car))
    s = ('\n'+ '-'*(num_guiones) + '\n').join(['\n'.join([escribeFila(f,car) for f in ~s]) \
                                            for s in [i|self for i in particion(self.corteElementos,
                                            ↪  self.m)]])
    return s

def latex(self):
    """ Construye el comando LaTeX para representar una BlockM """
    if not self.m:
            return self.sis().latex()

    ln = [len(n) for n in particion(self.corteSistema, self.n)]
    salto = salto_de_fila(e for f in (~self).lista for e in f.lista)
    return \
        '\\left[ \\begin{array}{' + '|'.join([n*'c' for n in ln])  + '}' + \
        (salto + ' \\hline ').join([salto.join(['&'.join([latex(e) for e in f.lista]) \
                                        for f in (~M).lista]) \
          for M in [ i|self for i in particion(self.corteElementos,self.m)]]) + \
         '\\\\ \\end{array} \\right]'
```

Método para definir por puntos de corte en los elementos un BlockM (sus "filas")

Para separar visualmente los elementos de los componentes (de los `BlockV` o columnas) de un `BlockM` es necesario indicar los índices de los puntos de corte de sus elementos (tras los que se mostrará una línea horizontal). Para especificar dichos índices se llama al método `cele()` (pues lo usaremos para visualmente separar algunos elementos en todos los componentes (es decir, en todos los `BlockV` o columnas) del `BlockM` (por ejemplo para *cortar por filas* una matriz).

Método para definir por puntos de corte en los elementos un BlockM

```
def cele(self, conjuntoIndices={}):
    """Modifica el atributo .corteElementos para insertar lineas
    horizontales entre las filas del BlockM

    """
    self.corteElementos = set(conjuntoIndices) if conjuntoIndices else {0}
    self.lista = [v.csis(self.corteElementos) for v in self.lista]
    return self
```

6.6. La clase `BlockM` completa

Definición de la subclase BlockM

```
class BlockM(Sistema):
    <<Texto de ayuda de la clase BlockM>>
    <<Inicialización de la subclase BlockM>>
    <<Operador transposición>>
    <<Comprobación de que las filas o columnas son de composición homogenea>>
    <<Apila una lista de Sistemas con el mismo número de elementos un BlockM>>
    <<Operador selector por la izquierda para la clase BlockM>>
    <<Transformaciones elementales de las filas de un BlockM>>
    <<Método para definir por puntos de corte en los elementos un BlockM>>
    <<Extiende una BlockM a lo largo de la diagonal con una lista de BlockMs>>
    <<Método de representación repr de la subclase BlockM>>
    <<Métodos de representación str y LaTeX de las subclases Matrix y BlockM>>
```

Capítulo 7

La subclase `Matrix`

7.1. Implementación de las matrices en la subclase `Matrix`

7.1.1. Texto de ayuda

Texto de ayuda de la clase Matrix

```
"""Matrix un Sistema de Vectores con el mismo número de componentes.

Una Matrix se puede instanciar con:

 1. una lista, tupla o Sistema de Vectores con el mismo número de
    componentes o longitud (serán las columnas).
 2. una lista, tupla o Sistema de listas, tuplas o Sistemas de núemros
    con la misma longitud (serán las filas de la matriz).

Parámetros:
    arg (list, tuple, Sistema): Lista, tupla o Sistema de Vectores con
        mismo núm. de componentes (sus columnas); o de listas, tuplas
        o Sistemas de números de misma longitud (sus filas).

Atributos heredados de la clase Sistema:
    lista              (list): list con los elementos.
    n                  (int) : número de elementos de la lista.
    corteSistema (set) : Conjunto de índices donde pintar
                                 separaciones visuales

Atributos heredados de la subclase BlockM:
    m              (int) : número de filas
    corteElementos (set) : Conjunto de índices donde pintar
                               separaciones visuales entre filas

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

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])])

"""
```

7.1.2. Método de inicialización:

Inicialización de la subclase Matrix

```
def __init__(self, data):
    """Inicializa una Matrix con una

    1) lista, tupla o Sistema de Vectores con el mismo número de
    elementos,

    2) tupla, lista o Sistema de tuplas, listas o Sistemas de números
    con el mismo número de elementos,

    3) con otra Matrix.

    """

    super().__init__(data)   # BlockM construye la lista de columnas

    # Lo que sigue solo comprueba que el argumento es válido para una Matrix
    lista = Sistema(data).lista

    if all([(isinstance(elemento, Vector) and len(elemento)==len(lista[0])) for elemento in lista]):
        pass

    elif lista and not lista[0]:
        pass

    elif Sistema(lista).es_de_composicion_y_longitud_uniforme() and es_ristra(lista[0]) and
    ↪  es_numero(lista[0][0]):
        # las columnas deben poder ser Vectores (si no, Vector da un error)
        [ Vector([elemento[i] for elemento in lista]) for i in range(len(lista[0])) ]

    elif isinstance(data,Matrix):
        pass

    else:
        raise ValueError("""El argumento debe ser una lista de Vectores o una lista de listas o
        tuplas con el mismo número de elementos!""")

    for v in self.lista:
        v.rpr='columna'
```

7.2. Métodos específicos de la subclase `Matrix`

7.2.1. Disposición de los elementos de una `Matrix`

Información sobre la disposición de los elementos de una Matrix

```
def es_cuadrada(self):
    """Indica si es cierto que la Matrix es cuadrada"""
    return self.m==self.n

def es_simetrica(self):
    """Indica si es cierto que la Matrix es simétrica"""
    return self == ~self

def es_antisimetrica(self):
    """Indica si es cierto que la Matrix es antisimétrica (igual a la
    opuesta de su transpuesta)"""
    return self == -(~self)

def es_triangularSup(self):
    """Indica si es cierto que la Matrix es triangular superior"""
    return not any(sum([[i|self|j for i in range(j+1,self.m+1)]      \
```

```
                                for j in range(1   ,self.n+1)], []))

def es_triangularInf(self):
    """Indica si es cierto que la Matrix es triangular inferior"""
    return (~self).es_triangularSup()

def es_triangular(self):
    """Indica si es cierto que la Matrix es triangular inferior"""
    return self.es_triangularSup() | self.es_triangularInf()

def es_diagonal(self):
    """Indica si es cierto que la Matrix es diagonal"""
    return self.es_triangularSup() & self.es_triangularInf()
```

7.2.2. Creación de un `Vector` a partir de la diagonal de una `Matrix`

Creación de un Vector a partir de la diagonal de una Matrix

```
def diag(self):
    """Crea un Vector a partir de la diagonal de la Matriz"""
    return Vector([ (j|self|j) for j in range(1,min(self.m,self.n)+1)])
```

7.2.3. Traza de una `Matrix`

Cálculo de la traza

```
def traza(self):
    """Devuelve la traza de la Matrix"""
    return sum(self.diag())
```

7.2.4. Normalizado de las columnas (o filas) de una matriz

Normalizado de las columnas o filas de una matriz

```
def normalizada(self, opcion='Columnas'):
    if opcion == 'Columnas':
        if any( vector.es_nulo() for vector in self):
            raise ValueError('algún vector es nulo')
        return Matrix([ vector.normalizado() for vector in self])
    else:
        return ~(~self.normalizada())
```

7.2.5. Potencias de una `Matrix` cuadrada

Ahora podemos calcular la `n`-ésima potencia de una `Matrix`. Cuando `n` es un entero positivo; basta multiplicar la `Matrix` por si misma `n` veces.

Si `n` es un entero negativo, entonces necesitamos calcular la inversa de la `n`-ésima potencia; para ello usará el método de eliminación Gaussiano que se describe en el Capítulo 10.

Potencia de una Matrix

```
def __pow__(self,n):
    """Calcula la n-ésima potencia de una Matrix"""
    if not isinstance(n,int): raise ValueError('La potencia no es un entero')
    if not self.es_cuadrada():  raise ValueError('Matrix no es cuadrada')

    M = self if n else I(self.n)
```

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

    return M.inversa() if n < 0 else M
```

7.2.6. Determinante mediante la expansión de Laplace

Determinante mediante la expansión de Laplace

```
def det(self, sust=[]):
    """Calculo del determinate mediante la expansión de Laplace"""
    if not self.es_cuadrada(): raise ValueError('Matrix no cuadrada')

    def cof(self,f,c):
        """Cofactor de la fila f y columna c"""
        excl = lambda k: tuple(i for i in range(1,self.m+1) if i!=k)
        return (-1)**(f+c)*(excl(f)|self|excl(c)).det()

    if self.m == 1:
        return 1|self|1

    A = Matrix(self.subs(sust))
    # expansión por la 1ª columna
    return sympy.simplify(sum([((f|A|1)*cof(A,f,1)).subs(sust) for f in range(1,A.m+1)]))
```

7.2.7. Método de Gram-Schmidt

Método Gram-Schmidt para ortogonalizar un sistema de Vectores de Rn

```
def GS(self):
    """Devuelve una Matrix equivalente cuyas columnas son ortogonales

    Emplea el método de Gram-Schmidt"""
    A = Matrix(self)
    for n in range(2,A.n+1):
        A & T([ (-fracc((A|n)*(A|j),(A|j)*(A|j)), j, n) \
                for j in range(1,n) if (A|j).no_es_nulo() ])
    return A
```

7.2.8. Otros métodos específicos de las matrices cuadradas que usan eliminación

Rango de una Matrix

El rango de una matriz es el número de columnas no nulas (el número de pivotes) de cualquiera de sus formas pre-escalonadas (Sección 5.2 del libro). Así que basta contar las columnas no nulas de `self.K()`.

Rango de una Matrix

```
def rango(self):
    """Rango de una Matrix"""
    return [v.no_es_nulo() for v in self.K()].count(True)
```

Comprobación de que una Matrix es singular o de que es invertible

Una matriz cuadrada de orden n es invertible si y solo si su rango es n; es decir, si sus formas pre-escalonadas no tienen columnas nulas. En caso contrario se dice que es singular.

Comprobación de que una Matrix es singular o de que es invertible

```
def es_singular(self):
    if not self.es_cuadrada():
        raise ValueError('La matriz no es cuadrada')
    return self.rango()<self.n

def es_invertible(self):
    if not self.es_cuadrada():
        raise ValueError('La matriz no es cuadrada')
    return self.rango()==self.n
```

Cálculo de la matriz inversa

El método `inversa` implementa el *Método para encontrar la inversa de una matriz* de la Sección 5.1.4 del libro. Dicho método se apoya en que, al aplicar una secuencia de transformaciones elementales a las columnas de la matriz que resulta de apilar $\mathbf{A}$ sobre la identidad, en la parte inferior queda registrada la matriz que ha realizado la transformación:

$$\begin{bmatrix}\mathbf{A}\\ \mathbf{I}\end{bmatrix} \xrightarrow{\tau_1\cdots\tau_k} \begin{bmatrix}\mathbf{AE}\\ \mathbf{E}\end{bmatrix}, \qquad \text{donde} \quad \mathbf{E} = \mathbf{I}_{\tau_1\cdots\tau_k}.$$

Si la secuencia de transformaciones conduce a la forma escalonada reducida de $\mathbf{A}$, y esta es la identidad, entonces $\mathbf{AE} = \mathbf{I}$ y, por tanto, $\mathbf{E}$ es la inversa de $\mathbf{A}$. Si por el contrario la forma escalonada reducida tiene columnas nulas, $\mathbf{A}$ es singular.

El código sigue esos pasos:

1. Encadena tres eliminaciones (véase la tabla de variantes): la doble reducción `elim(2)` anula los componentes a ambos lados de cada pivote, `elim(20)` reordena las columnas para escalonar la matriz y `elim(10)` normaliza los pivotes. El resultado es la forma escalonada reducida, y el atributo `pasos` acumula las transformaciones de las tres etapas.
2. Si la matriz es singular se produce un error.
3. En caso contrario la inversa es el resultado de aplicar sobre la identidad las transformaciones de las columnas: `I(self.n) & T(TrC)`.

No es necesario apilar realmente la identidad para hacer los cálculos; la matriz apilada solo se usa para escribir el código LaTeX del atributo `tex`, de modo que la representación muestra el procedimiento tal como aparece en el libro.

Cálculo de la matriz inversa

```
def inversa(self, rep=0, sust=[], repsust=0):
    """Inversa de Matrix

    Con una lista de sustituciones (sust) hace lo mismo que
    InvMat(self, rep, sust, repsust)"""
    if sust:
        return InvMat(self, rep, sust, repsust)
    if not self.es_cuadrada():
        raise ValueError('Matrix no cuadrada')
```

```
    pasos = self.elim(2).elim(20).elim(10).pasos
    TrF = pasos[0]
    TrC = pasos[1]
    tex   = rprElim(self.apila(I(self.n),1),pasos)

    if rep:
        display(Math(tex))

    if self.es_singular():
        raise ValueError('Matrix es singular')

    Inversa = I(self.n) & T(TrC)
    Inversa.pasos = pasos
    Inversa.TrF   = TrF
    Inversa.TrC   = TrC
    Inversa.tex   = tex
    return Inversa
```

Las clases `InvMat`, `InvMatF` e `InvMatFC` (que se definen más adelante) ofrecen tres maneras de llegar a la inversa:

Clase	Transformaciones	Matriz que se transforma	La inversa es
`InvMat(A)`	de las columnas	**A** apilada sobre **I**	la parte inferior
`InvMatF(A)`	de las filas	**A** concatenada con **I**	la parte derecha
`InvMatFC(A)`	de filas y columnas	**A** con **I** a la derecha y debajo	el producto de ambas submatrices

Las tres producen un error si la matriz es singular (tras mostrar los pasos, si se han pedido). `InvMat(A)` es otro modo de llamar a `A.inversa()`. `InvMatF` hace lo mismo con las filas (al transformar las filas, la matriz que realiza la transformación queda registrada a la derecha). `InvMatFC` primero escalona con transformaciones de las filas y después termina de reducir con transformaciones de las columnas; si **F** y **C** son las matrices en que se transforman las identidades de la derecha y de debajo, entonces $\mathbf{FAC} = \mathbf{I}$, y por tanto la inversa de **A** es el producto **CF**.

Ejemplos de matrices inversas

Usaremos la matriz de un ejemplo de la Sección 5.1.4 del libro.

```
A = Matrix( [ [5,0], [2,-2] ] )
```

El atributo `tex` de la inversa muestra el proceso sobre la matriz apilada:

```
print(displaymath( InvMat(A).tex ))
```

$$\left[\begin{array}{cc} 5 & 0 \\ 2 & -2 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \xrightarrow{\overset{\boldsymbol{\tau}}{[(1)\mathbf{2}+\mathbf{1}]}} \left[\begin{array}{cc} 5 & 0 \\ 0 & -2 \\ \hline 1 & 0 \\ 1 & 1 \end{array}\right] \xrightarrow[\left[\left(-\frac{1}{2}\right)\mathbf{2}\right]]{\overset{\boldsymbol{\tau}}{\left[\left(\frac{1}{5}\right)\mathbf{1}\right]}} \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \\ \hline \frac{1}{5} & 0 \\ \frac{1}{5} & -\frac{1}{2} \end{array}\right]$$

La inversa es la matriz que ha quedado en la parte inferior

```
print(dmath( InvMat(A) ))
```

$$\begin{bmatrix} \frac{1}{5} & 0 \\ \frac{1}{5} & -\frac{1}{2} \end{bmatrix}$$

y, como con cualquier resultado de una eliminación, el atributo `pasos` guarda las transformaciones aplicadas (aquí todas son de las columnas):

```
print(dprint( InvMat(A).pasos ))
```

```
[[], [T([(1, 2, 1)]), T([(1/5, 1), (-1/2, 2)])]]
```

Efectivamente, el producto de la matriz por su inversa es la identidad:

```
print(dprint( A*InvMat(A) == I(2) ))
```

```
True
```

Operando con las filas, la inversa aparece a la derecha:

```
print(displaymath( InvMatF(A).tex ))
```

$$\left[\begin{array}{cc|cc} 5 & 0 & 1 & 0 \\ 2 & -2 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{2}]\\ [(5)\mathbf{2}]}]{\boldsymbol{\tau}} \left[\begin{array}{cc|cc} 5 & 0 & 1 & 0 \\ 0 & -10 & -2 & 5 \end{array}\right] \xrightarrow[\substack{[(-\frac{1}{10})\mathbf{2}]\\ [(\frac{1}{5})\mathbf{1}]}]{\boldsymbol{\tau}} \left[\begin{array}{cc|cc} 1 & 0 & \frac{1}{5} & 0 \\ 0 & 1 & \frac{1}{5} & -\frac{1}{2} \end{array}\right]$$

Y operando primero con las filas y después con las columnas, la inversa es el producto de la matriz de debajo por la matriz de la derecha (por ese orden):

```
print(displaymath( InvMatFC(A).tex ))
```

$$\left[\begin{array}{cc|cc} 5 & 0 & 1 & 0 \\ 2 & -2 & 0 & 1 \\ \hline 1 & 0 & & \\ 0 & 1 & & \end{array}\right] \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{2}]\\ [(5)\mathbf{2}]}]{\boldsymbol{\tau}} \left[\begin{array}{cc|cc} 5 & 0 & 1 & 0 \\ 0 & -10 & -2 & 5 \\ \hline 1 & 0 & & \\ 0 & 1 & & \end{array}\right] \xrightarrow[[(-\frac{1}{10})\mathbf{2}]]{\substack{\boldsymbol{\tau}\\ [(\frac{1}{5})\mathbf{1}]}} \left[\begin{array}{cc|cc} 1 & 0 & 1 & 0 \\ 0 & 1 & -2 & 5 \\ \hline \frac{1}{5} & 0 & & \\ 0 & -\frac{1}{10} & & \end{array}\right]$$

```
print(dprint( InvMatFC(A) == InvMat(A) ))
```

```
True
```

Cuando la matriz es singular su rango es menor que su orden, y al intentar invertirla se produce un error:

```
B = Matrix( [ [1,2], [2,4] ] )
print(dprint( (B.rango(), B.es_singular()) ))
```

```
(1, True)
```

```
try:
    B.inversa()
```

```
except ValueError as error:
    print(dprint( error ))
```

```
Matrix es singular
```

Cálculo del determinante

Cálculo del determinante por eliminación

```
def determinante(self, rep = False, sust = []):
    """Calculo del determinate mediante eliminación"""
    if not self.es_cuadrada(): raise ValueError('Matrix no cuadrada')

    return Determinante(self.subs(sust),rep).valor
```

Diagonalización por Semejanza

Diagonalizando por Semejanza en una matriz por bloques triangulares

```
def diagonalizaS(self, espectro, rep=False, sust=[]):
    <<Texto de ayuda para la clase DiagonalizaS>>
    D            = Matrix(self).copy().subs(sust)
    <<Comprobaciones previas a la diagonalización>>
    espectro     = [sympy.S(l).subs(sust) for l in espectro]
    S            = I(D.n)
    Tex          = latex( D.apila(S,1) )
    pasosPrevios = [[],[]]
    selecc       = list(range(1, D.n+1))

    for landa in espectro:
        m = selecc.pop()
        <<Restamos lambda*I>>
        # eliminamos elementos superiores de la columna con elim de izda a dcha
        TrCol = filtradopasos((slice(None,m)|D|slice(None,m)).elim(20).pasos[1])
        <<Aplicación de las transformaciones y sus inversas espejo>>

        if m < D.n: # eliminamos elementos inferiores de la columna con los pivotes de la diagonal
            for i,_ in enumerate(slice(m+1,None)|D|m, m+1):
                TrCol = filtradopasos([ T([(-fracc(i|D|m, i|D|i), i, m)]) ]) if i|D|i else [T([])]
                <<Aplicación de las transformaciones y sus inversas espejo>>
        <<Sumamos lambda*I>>

    if rep: display(Math(Tex))
    D.espectro = espectro[::-1]
    D.tex = Tex
    D.S   = S
    D.TrC = pasosPrevios[1]
    D.TrF = [op.Tinversa().espejo() for op in D.TrC[::-1]]
    D.pasos = [D.TrF, D.TrC]
    return D
```

Texto de ayuda para la clase DiagonalizaS

```
"""Diagonaliza por bloques triangulares una Matrix cuadrada

Encuentra una matriz diagonal semejante mediante transformaciones de
sus columnas y las correspondientes transformaciones inversas espejo
de las filas. Requiere una lista de autovalores (espectro), que deben
aparecer en dicha lista tantas veces como sus respectivas
multiplicidades algebraicas. Los autovalores aparecen en la diagonal
principal de la matriz diagonal. El atributo S de dicha matriz
diagonal es una matriz cuyas columnas son autovectores de los
correspondientes autovalores.  """
```

Comprobaciones previas a la diagonalización

```
def no_son_autovalores(A, L):
    no_son=[l for i,l in enumerate(L) if (D-l*I(D.n)).es_invertible()]
    if no_son:
        print(texto_de_salida('no_autovalores'), no_son)
        return True
    else:
        False

def espectro_correcto(A, L):
    x = sympy.symbols('x')
    monomio = lambda l,x: l-x
    p = (A-x*I(A.n)).det()
    for l in L:
        p, r = sympy.div(p, monomio(l,x), domain ='QQ')
    return False if p!=1 or r!=0 else True

if not D.es_cuadrada():
    raise ValueError('Matrix no es cuadrada')

if not isinstance(espectro, list):
    raise ValueError('espectro no es una lista')

if no_son_autovalores(D, espectro):
    raise ValueError('quite de la lista los valores que no son autovalores')

if not len(espectro)==D.n:
    raise ValueError('el espectro propocionado tiene un número inadecuado de autovalores')

if not espectro_correcto(D, espectro):
    raise ValueError('introduzca una lista correcta de autovalores')
```

Restamos lambda*I

```
D = (D-(landa*I(D.n))).subs(sust)
Tex += r'\xrightarrow[\boxed{' + latex(landa) + r'\mathbf{I}}]{(-)}' + latex((D.apila(S,1)).subs(sust))
```

Sumamos lambda*I

```
D = (D+(landa*I(D.n))).subs(sust)
Tex += r'\xrightarrow[\boxed{' + latex(landa) + r'\mathbf{I}}]{(+)}' + latex((D.apila(S,1)).subs(sust))
```

Aplicación de las transformaciones y sus inversas espejo

```
if T(TrCol):
    Tex             = rprElim( D.apila(S,1),  [[], TrCol], Tex, sust) if TrCol else Tex
    D               = (D & T(TrCol)).subs(sust)
    S               = (S & T(TrCol)).subs(sust)
    pasosPrevios[1] = pasosPrevios[1] + TrCol

    TrFilas         = [ T( [op.Tinversa().espejo() for op in TrCol[::-1]] ) ]

    Tex             = rprElim( D.apila(S,1), [TrFilas, []], Tex, sust) if TrCol else Tex
    D               = (T(TrFilas) & D).subs(sust)
    pasosPrevios[0] = TrFilas + pasosPrevios[0]
```

Diagonalización por Congruencia

Texto de ayuda para la clase DiagonalizaC

```
"""Diagonaliza por congruencia una Matrix simétrica (evitando dividir)
```

```
Encuentra una matriz diagonal por conruencia empleando una matriz B
invertible (evitando fracciones por defecto, si variante=1 entonces no
evita las fracciones) por la derecha y su transpuesta por la
izquierda. No emplea los autovalores. En general los elementos en la
diagonal principal no son autovalores, pero hay tantos elementos
positivos en la diagonal como autovalores positivos, tantos negativos
como autovalores negativos, y tantos ceros como auntovalores nulos.

"""
```

Diagonalizando por Congruencia una matriz simétrica

```
def diagonalizaC(self, rep=False, sust=[], variante=0):
    <<Texto de ayuda para la clase DiagonalizaC>>
    <<Definición del método auxiliar BuscaPrimerNoNuloEnLaDiagonal>>
    if not variante in {0,1}:
        raise ValueError('La variante debe ser 0 ó 1')

    D            = Matrix(self).copy().subs(sust)

    if not D.es_simetrica():
        raise ValueError('La matriz no es simétrica')

    pasosPrevios = [ [], [] ]

    for i in range(1, D.n):
        p = BuscaPrimerNoNuloEnLaDiagonal(D, i)
        j = [k for k,col in enumerate(D|slice(i,None),i) if (i|col and not k|col)]

        if not (i|D|i):
            if p:
                Tr = T( {i, p} )
                p = i
                <<Aplicación de las transformaciones a las columnas y a las filas>>
            elif j:
                Tr = T( (1, j[0], i) )
                p = i
                <<Aplicación de las transformaciones a las columnas y a las filas>>
        if p:
            Tr = T(((i,)|D).elim(variante).pasos[1])
            <<Aplicación de las transformaciones a las columnas y a las filas>>

    D.pasos     = pasosPrevios
    D.tex       = rprElimCF(Matrix(self).subs(sust), D.pasos, [], sust)
    D.TrF       = filtradopasos(T(D.pasos[0]))
    D.TrC       = filtradopasos(T(D.pasos[1]))
    D.B         = I(self.n) & D.TrC

    if rep:
        display(Math(D.tex))

    return D
```

Definición del método auxiliar BuscaPrimerNoNuloEnLaDiagonal

```
def BuscaPrimerNoNuloEnLaDiagonal(self, i=1):
    """Indica el índice de la primera componente no nula de a diagonal
    desde de la posición i en adelante. Si son todas nulas devuelve 0

    """

    d = (slice(i,None)|self|slice(i,None)).diag().sis()
    return next((pos for pos, x in enumerate(d) if x), -i) + i
```

Aplicación de las transformaciones a las columnas y a las filas

```
pasos = [ filtradopasos([~Tr]), filtradopasos([Tr]) ]
pasosPrevios[0] = pasos[0] + pasosPrevios[0]
pasosPrevios[1] = pasosPrevios[1] + pasos[1]
D = (T(pasos[0]) & D & T(pasos[1])).subs(sust)
```

7.2.9. Diagonalización ortogonal de una matriz simétrica

Diagonalizando Ortogonalmente una matriz simétrica

```
def diagonalizaO(self, espectro, sust=[]):
    <<Texto de ayuda para la clase DiagonalizaO>>
    D            = Matrix(self).copy().subs(sust)

    if not D.es_simetrica():
        raise ValueError('La matriz no es simétrica')

    <<Comprobaciones previas a la diagonalización>>
    <<Método auxiliar para creación de una base ortonormal donde \Vect{q} es el último vector>>

    espectro     = [sympy.S(l).subs(sust) for l in espectro]
    S            = I(D.n)
    Tex          = latex( D.apila(S,1) )
    pasosPrevios = [[],[]]
    selecc       = list(range(1,D.n+1))

    for l in espectro:
        D       = (D - l*I(D.n)).subs(sust)
        k       = len(selecc)
        nmenosk = (D.n)-k
        m       = selecc.pop()
        TrCol   = filtradopasos((slice(None,m)|D|slice(None,m)).elim(20, False, sust).pasos[1])
        D       = (D + l*I(D.n)).subs(sust)


        q = ( (I(k) & T(TrCol)).subs(sust) )|0
        q = (sympy.sqrt(q*q)) * q

        Q = BaseOrtNor(q).concatena(M0(k,nmenosk)).apila( \
                M0(nmenosk,k).concatena(I(nmenosk)))  if nmenosk else BaseOrtNor(q)

        S = (S*Q).subs(sust)
        D = (~Q*D*Q).subs(sust)

    D.Q = S
    D.espectro = espectro[::-1]
    return D
```

Texto de ayuda para la clase DiagonalizaO

```
"""Diagonaliza ortogonalmente una Matrix simétrica

Encuentra una matriz diagonal por semejanza empleando una matriz
ortogonal Q a la derecha y su inversa (transpuesta) por la izquierda.
Requiere una lista de autovalores (espectro), que deben aparecer
tantas veces como sus respectivas multiplicidades algebraicas. Los
autovalores aparecen en la diagonal principal de la matriz
diagonal. El atributo Q de la matriz diagonal es la matriz ortogonal
cuyas columnas son autovectores de los correspondientes autovalores.

"""
```

Método auxiliar para creación de una base ortonormal donde q es el último vector

```
def BaseOrtNor(q):
    "Crea una base ortonormal cuyo último vector es 'q'"
```

```
    if not isinstance(q,Vector): raise ValueError('El argumento debe ser un Vector')
    M = Matrix([q]).concatena(I(q.n)).GS()
    l = [ j for j, v in enumerate(M, 1) if v.no_es_nulo() ]
    l = l[1:len(l)]+[l[0]]
    return (M|l).normalizada()
```

7.3. Representación de la clase `Matrix`

Método de representación repr de la subclase Matrix

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

7.4. La clase `Matrix` completa

Definición de la subclase Matrix

```
class Matrix(BlockM):
    <<Texto de ayuda de la clase Matrix>>
    <<Inicialización de la subclase Matrix>>
    <<Información sobre la disposición de los elementos de una Matrix>>
    <<Creación de un Vector a partir de la diagonal de una Matrix>>
    <<Cálculo de la traza>>
    <<Normalizado de las columnas o filas de una matriz>>
    <<Potencia de una Matrix>>
    <<Determinante mediante la expansión de Laplace>>
    <<Método Gram-Schmidt para ortogonalizar un sistema de Vectores de Rn>>
    <<Rango de una Matrix>>
    <<Comprobación de que una Matrix es singular o de que es invertible>>
    <<Cálculo de la matriz inversa>>
    <<Cálculo del determinante por eliminación>>
    <<Diagonalizando por Semejanza en una matriz por bloques triangulares>>
    <<Diagonalizando por Congruencia una matriz simétrica>>
    <<Diagonalizando Ortogonalmente una matriz simétrica>>
    <<Método de representación repr de la subclase Matrix>>
```

Capítulo 8

Las subclases `M0`, `M1` e `I`

Definición de las subclases M0, M1 e I

```
class M0(Matrix):
    def __init__(self, m, n=None):
        """ Inicializa una matriz nula de orden m por n """
        n = m if n is None else n

        super().__init__( [V0(m)]*n )
        self.__class__ = Matrix

class M1(Matrix):
    def __init__(self, m, n=None):
        """ Inicializa una matriz nula de orden m por n """
        n = m if n is None else n

        super().__init__( [V1(m)]*n )
        self.__class__ = Matrix

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

Capítulo 9

Las subclases de sistemas simbólicos

Definición de las subclases de sistemas simbólicos

```
class SistemaSimbol(BlockV):
    def __init__(self, arg, char=' ', rpr='columna'):
        """Inicializa un Sistema Simbólico con una lista, tupla o Sistema"""
        self.sistema = arg if isinstance(arg, Sistema) else Sistema(CreaLista(arg))
        letra, simbolo = self.simbolos(char)
        self.lista   = [self.sistema, simbolo, letra]

        self.n = len(self.sistema)
        self.corteSistema = set()
        self.rpr = rpr

    def simbolos(self, char):
        letra = sympy.Symbol('0', zero=True) if self.es_nulo() else sympy.Symbol(char.upper())
        simbolo_latex = sympy.Symbol(r'\mathsf{0}', zero=True) if self.es_nulo() else
        ↪  sympy.Symbol(r'\mathsf{'+str(letra)+'}')
        return letra, simbolo_latex

    def es_nulo(self, sust=[]):
        """Indica si es cierto que el sistema simbólico es nulo"""
        # tras operar con el sistema simbólico, el sistema al que representa
        # es el primer elemento de la lista (el atributo sistema no se actualiza)
        sistema = self.lista[0] if hasattr(self, 'lista') else self.sistema
        return CreaSistema(sistema).es_nulo(sust)

    def copy(self):
        """ Genera una copia de un SistemaSimbol """
        return type(self)(self.lista[0], str(self.lista[2]))

    def __str__(self):
        """ Muestra un SistemaSimbol en su representación python """
        return str(self.lista[2])

    def __repr__(self):
        """ Muestra un SistemaSimbol en su representación python """
        return 'SistemaSimbol(' + repr(self.lista[0]) + ', "' + repr(self.lista[2]) + '")'

    def latex(self):
        """ Construye el comando LaTeX para representar un SistemaSimbol """
        return latex(self.lista[1])

    def _repr_latex_(self):
        """ Representación para el entorno jupyter en Emacs """
        return '$'+self.latex()+'$'

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

```
        return html(self.latex())

    def elim(self, variante=0, rep=False, sust=[], repsust=False):
        return self.sistema.elim(variante, rep, sust, repsust)

class MatrixSimbol(SistemaSimbol):
    def __init__(self, arg, char=' ', rpr='columna'):
        self.sistema = arg if isinstance(arg, Matrix) else BlockM(CreaLista(arg))
        letra, simbolo = self.simbolos(char)
        self.lista   = [self.sistema, simbolo, letra]

        self.n = len(self.sistema)
        self.corteSistema = set()
        self.rpr = rpr

    def simbolos(self, char):
        letra = sympy.Symbol('0', zero=True) if self.es_nulo() else sympy.Symbol(char.upper())
        simbolo_latex = sympy.Symbol(r'\boldsymbol{\mathsf{0}}', zero=True) if self.es_nulo() else
        ↪  sympy.Symbol(r'\boldsymbol{\mathsf{'+str(letra)+'}}')
        return letra, simbolo_latex

    def __repr__(self):
        """ Muestra un MatrixSimbol en su representación python """
        return 'MatrixSimbol(' + repr(self.lista[0]) + ', "' + repr(self.lista[2]) + '")'

    def latex(self):
        """ Construye el comando LaTeX para representar un MatrixSimbol """
        return latex(self.lista[1])

class VectorSimbol(SistemaSimbol):
    def __init__(self, arg, char=' ', rpr='columna'):
        self.sistema = arg if isinstance(arg, Vector) else BlockV(CreaLista(arg))
        letra, simbolo = self.simbolos(char)
        self.lista   = [self.sistema, simbolo, letra]

        self.n = len(self.sistema)
        self.corteSistema = set()
        self.rpr = rpr

    def simbolos(self, char):
        letra = sympy.Symbol('0', zero=True) if self.es_nulo() else sympy.Symbol(char.lower())
        simbolo_latex = sympy.Symbol(r'\boldsymbol{0}', zero=True) if self.es_nulo() else
        ↪  sympy.Symbol(r'\boldsymbol{'+str(letra)+'}')
        return letra, simbolo_latex

    def __repr__(self):
        """ Muestra un VectorSimbol en su representación python """
        return 'VectorSimbol(' + repr(self.lista[0]) + ', "' + repr(self.lista[2]) + '")'

    def latex(self):
        """ Construye el comando LaTeX para representar un VectorSimbol """
        return latex(self.lista[1])
```

Capítulo 10

`Sistemas` obtenidos tras algunos procedimientos específicos de eliminación

Por comodidad y compatibilidad con la versión anterior de NAcAL creamos las siguientes subclases de Sistema que resultan tras aplicar ciertos pasos de eliminación.

10.1. Subclases de `Sistema`: `Elim`, `ElimG`, `ElimGJ`, `ElimGF` y `ElimGJF`

Son una forma alternativa de llamar a los métodos `.K()`, `.L()`, `.R()`, `.U()` y `.UR()`. Los dos últimos, `.U()` y `.UR()` (es decir, `ElimGF` y `ElimGJF`) solo tienen sentido si el Sistema es un arreglo rectangular y cada uno de sus componentes es de composición uniforme, es decir, es necesario que las operaciones de eliminación estén definidas entre los elementos de cada componente del sistema. Es el método correspondiente a la eliminación por filas en matrices.

```
class Elim(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve una forma pre-escalonada de un sistema

           operando con sus elementos (y evitando operar con
           fracciones).  Si rep es no nulo, se muestran en Jupyter los
           pasos dados

        """
        self.__dict__.update(sistema.K(rep, sust, repsust).__dict__)
        self.__class__ = type(sistema)
```

```
class ElimG(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve una forma escalonada de un sistema

           operando con sus elementos (y evitando operar con
           fracciones).  Si rep es no nulo, se muestran en Jupyter los
           pasos dados

        """
```

```
        self.__dict__.update(sistema.L(rep, sust, repsust).__dict__)
        self.__class__ = type(sistema)
```

```
class ElimGJ(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la forma escalonada reducida de un sistema

           operando con sus elementos (y evitando operar con
           fracciones).  Si rep es no nulo, se muestran en Jupyter los
           pasos dados

        """
        self.__dict__.update(sistema.R(rep, sust, repsust).__dict__)
        self.__class__ = type(sistema)
```

```
class ElimF(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la forma escalonada por filas (si es posible)

           y evitando operar con fracciones.  Si rep es no nulo, se
           muestran en Jupyter los pasos dados

        """
        self.__dict__.update(sistema.elim(4, rep, sust, repsust).__dict__)
        self.__class__ = type(sistema)
```

```
class ElimGF(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la forma escalonada por filas (si es posible)

           y evitando operar con fracciones.  Si rep es no nulo, se
           muestran en Jupyter los pasos dados

        """
        self.__dict__.update(sistema.U(rep, sust, repsust).__dict__)
        self.__class__ = type(sistema)
```

```
class ElimGJF(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la forma escalonada reducida por filas (si es posible)

           y evitando operar con fracciones.  Si rep es no nulo, se
           muestran en Jupyter los pasos dados

        """
        self.__dict__.update(sistema.UR(rep, sust, repsust).__dict__)
        self.__class__ = type(sistema)
```

10.2. Subclases de `Matrix`: `InvMat`, `InvMatF` e `InvMatFC`

Son una forma alternativa de llamar al método `.inversa()`,

```
class InvMat(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la matriz inversa y los pasos dados sobre las columnas

           y evitando operar con fracciones.  Si rep es no nulo, se
           muestran en Jupyter los pasos dados

        """
        A = sistema.subs(sust).inversa()
        A.tex, A.pasos = texYpasos(sistema.apila(I(sistema.n),1), A.pasos, rep, sust, repsust)
        A.TrF = A.pasos[0]
        A.TrC = A.pasos[1]
        self.__dict__.update(A.__dict__)
        self.__class__ = type(sistema)
```

```
class InvMatF(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la matriz inversa y los pasos dados sobre las filas

           y evitando operar con fracciones.  Si rep es no nulo, se
           muestran en Jupyter los pasos dados

        """

        if not sistema.es_cuadrada():
            raise ValueError('Matrix no cuadrada')

        pasos = sistema.elim(26).elim(26).elim(14).pasos
        A = T(pasos[0]) & I(sistema.n)
        A.tex, A.pasos = texYpasos(sistema.concatena(I(sistema.n),1), pasos, rep, sust, repsust)
        if sistema.subs(sust).es_singular():
            raise ValueError('Matrix es singular')
        A.TrF = A.pasos[0]
        A.TrC = A.pasos[1]
        self.__dict__.update(A.__dict__)
        self.__class__ = type(sistema)
```

```
class InvMatFC(Sistema):
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Devuelve la matriz inversa y los pasos dados sobre las filas y columnas

           y evitando operar con fracciones.  Si rep es no nulo, se
           muestran en Jupyter los pasos dados

        """

        if not sistema.es_cuadrada():
            raise ValueError('Matrix no cuadrada')

        pasos = sistema.elim(24).elim(10).pasos
        A = sistema.copy()
        nan = sympy.symbols(r'\ ')
        dummyMatrix = M1(A.n)*nan
        A.tex, A.pasos = texYpasos(A.concatena(I(A.n),1).apila(I(A.n).concatena(dummyMatrix,1),1), pasos, rep,
        ↪  sust, repsust)
        if sistema.subs(sust).es_singular():
            raise ValueError('Matrix es singular')
        A.TrF = A.pasos[0]
        A.TrC = A.pasos[1]
        A.F   = T(A.TrF) & I(A.n)
```

```
A.C   = I(A.n) & T(A.TrC)
A.lista = (A.C*A.F).lista
self.__dict__.update(A.__dict__)
self.__class__ = type(sistema)
```

10.3. Diagonalización de matrices: resumen de los tres métodos

El módulo diagonaliza matrices cuadradas de tres maneras. Las tres devuelven una `Matrix` (la matriz diagonal) que guarda en un atributo la matriz con la que se ha logrado.

Método	Matriz	Resultado `D`	Atributo	Relación
`A.diagonalizaS(espectro)`	cuadrada	diagonal (o triangular por bloques) semejante	`S`	$\mathbf{S}^{-1}\mathbf{AS} = \mathbf{D}$
`A.diagonalizaO(espectro)`	simétrica	diagonal semejante	`Q`	$\mathbf{Q}^{\intercal}\mathbf{AQ} = \mathbf{D}$, con $\mathbf{Q}$ ortogonal
`A.diagonalizaC()`	simétrica	diagonal congruente	`B`	$\mathbf{B}^{\intercal}\mathbf{AB} = \mathbf{D}$

- **Por semejanza** (`diagonalizaS`). Hay que indicar el espectro: una lista con los autovalores, cada uno repetido tantas veces como su multiplicidad algebraica. Para cada autovalor λ se resta $\lambda\mathbf{I}$, se anulan componentes con transformaciones elementales de las columnas, se aplican a las filas las transformaciones inversas «espejo» (para que la matriz siga siendo semejante a la inicial) y se vuelve a sumar $\lambda\mathbf{I}$. Las columnas de `S` son autovectores. Si la matriz no es diagonalizable, el resultado es triangular inferior por bloques, con un bloque por cada autovalor.
- **Ortogonalmente** (`diagonalizaO`). Solo para matrices simétricas, y también requiere el espectro. Las columnas de `Q` son autovectores de norma uno y perpendiculares entre sí. No guarda el desarrollo (no tiene atributo `tex`).
- **Por congruencia** (`diagonalizaC`). Solo para matrices simétricas. No usa los autovalores: aplica a las filas las mismas transformaciones que a las columnas. Los números de la diagonal no son, en general, los autovalores, pero hay tantos positivos, negativos y nulos como autovalores positivos, negativos y nulos. Por defecto evita las fracciones; con `variante=1` no lo hace.

En los dos métodos que usan el espectro, los autovalores aparecen en la diagonal en orden inverso al de la lista indicada, y el atributo `espectro` del resultado los guarda en el orden en que aparecen.

Argumento	Métodos	Significado
`espectro`	`diagonalizaS`, `diagonalizaO`	lista de autovalores (con sus multiplicidades)
`rep`	`diagonalizaS`, `diagonalizaC`	si no es nulo, muestra el proceso en Jupyter
`sust`	los tres	lista de sustituciones de símbolos
`variante`	`diagonalizaC`	si es 1 no evita las fracciones

Como con la eliminación, `diagonalizaS` y `diagonalizaC` guardan el proceso en el atributo `tex` y las transformaciones en `pasos`, `TrF` (filas) y `TrC` (columnas). Las clases `DiagonalizaS`, `DiagonalizaO` y `DiagonalizaC` (descritas más abajo) son otra manera de llamar a estos métodos: `DiagonalizaS(A, espectro)` es lo mismo que `A.diagonalizaS(espectro)`.

Si la lista no contiene los autovalores de la matriz con sus multiplicidades, se produce un error que indica el motivo.

10.3.1. Ejemplos de diagonalización

Una matriz simétrica con autovalores 1 y 3:

```
A = Matrix( [ [2,1], [1,2] ] )
```

Por semejanza, la matriz diagonal y la matriz de autovectores son

```
D = A.diagonalizaS( [1,3] )
```

```
print(dmath( D ))
```

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

```
print(dmath( D.S ))
```

$$\begin{bmatrix} \frac{1}{2} & -1 \\ \frac{1}{2} & 1 \end{bmatrix}$$

y se verifica que $\mathbf{S^{-1}AS = D}$:

```
print(dprint( D.S.inversa() * A * D.S == D ))
```

```
True
```

El atributo `tex` muestra el proceso. Las flechas rotuladas con $(-)$ y $(+)$ indican que se resta o se suma el múltiplo de la identidad que aparece recuadrado; bajo la matriz se va formando `S`:

```
print(displaymath( desarrollo(D, por_linea=5) ))
```

$$\left[\begin{array}{cc} 2 & 1 \\ \hline 1 & 2 \\ 1 & 0 \\ 0 & 1 \end{array}\right] \xrightarrow[\boxed{1\mathbf{I}}]{(-)} \left[\begin{array}{cc} 1 & 1 \\ \hline 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{array}\right] \xrightarrow[[(-1)\mathbf{1}+\mathbf{2}]]{\boldsymbol{\tau}} \left[\begin{array}{cc} 1 & 0 \\ \hline 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{array}\right] \xrightarrow[[(1)\mathbf{2}+\mathbf{1}]]{\boldsymbol{\tau}} \left[\begin{array}{cc} 2 & 0 \\ \hline 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{array}\right] \xrightarrow[\boxed{1\mathbf{I}}]{(+)} \left[\begin{array}{cc} 3 & 0 \\ \hline 1 & 1 \\ 1 & -1 \\ 0 & 1 \end{array}\right]$$

$$\xrightarrow[\boxed{3\mathbf{I}}]{(-)} \left[\begin{array}{cc} 0 & 0 \\ \hline 1 & -2 \\ 1 & -1 \\ 0 & 1 \end{array}\right] \xrightarrow[[(\frac{1}{2})\mathbf{2}+\mathbf{1}]]{\boldsymbol{\tau}} \left[\begin{array}{cc} 0 & 0 \\ \hline 0 & -2 \\ \frac{1}{2} & -1 \\ \frac{1}{2} & 1 \end{array}\right] \xrightarrow[[(-\frac{1}{2})\mathbf{1}+\mathbf{2}]]{\boldsymbol{\tau}} \left[\begin{array}{cc} 0 & 0 \\ \hline 0 & -2 \\ \frac{1}{2} & -1 \\ \frac{1}{2} & 1 \end{array}\right] \xrightarrow[\boxed{3\mathbf{I}}]{(+)} \left[\begin{array}{cc} 3 & 0 \\ \hline 0 & 1 \\ \frac{1}{2} & -1 \\ \frac{1}{2} & 1 \end{array}\right]$$

Ortogonalmente se obtiene la misma matriz diagonal, pero con una matriz ortogonal:

```
O = A.diagonalizaO( [1,3] )
```

```
print(dmath( O.Q ))
```

$$\begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

```
print(dprint( ~O.Q * A * O.Q == O ))
```

```
True
```

Por congruencia no hace falta conocer los autovalores. Los números de la diagonal no son los autovalores, pero ambos son positivos, como los autovalores:

```
C = A.diagonalizaC()
```

```
print(displaymath( C.tex ))
```

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \xrightarrow[[(-1)\mathbf{1}+\mathbf{2}]]{\boldsymbol{\tau}\atop[(2)\mathbf{2}]} \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} \xrightarrow[\boldsymbol{\tau}\atop[(2)\mathbf{2}]]{[(-1)\mathbf{1}+\mathbf{2}]} \begin{bmatrix} 2 & 0 \\ 0 & 6 \end{bmatrix}$$

```
print(dmath( C.B ))
```

$$\begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

```
print(dprint( ~C.B * A * C.B == C ))
```

```
True
```

Con `variante=1` no se evitan las fracciones:

```
print(dmath( A.diagonalizaC(variante=1) ))
```

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

Una matriz que no es diagonalizable: el autovalor 2 tiene multiplicidad algebraica dos, pero solo una dirección de autovectores. El resultado es triangular por bloques:

```
N = Matrix( [ [2,1,0], [0,2,0], [0,0,3] ] )
```

```
print(dmath( N.diagonalizaS( [2,2,3] ) ))
```

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

Si algún número de la lista no es un autovalor, se indica cuál y se produce un error:

```
try:
    A.diagonalizaS( [1,2] )
except ValueError as error:
    print(dprint( error ))
```

los valores de la siguiente lista no son autovalores de la matriz: [2] `quite de la lista los valores que no son autovalores`

Y los métodos para matrices simétricas rechazan las que no lo son:

```
try:
    Matrix( [ [1,2], [3,4] ] ).diagonalizaC()
except ValueError as error:
    print(dprint( error ))
```

`La matriz no es simétrica`

10.4. Subclases de `Matrix`: `DiagonalizaS`, `DiagonalizaO` y `DiagonalizaC`

Es una forma alternativa de llamar al método `.diagonalizaS()`,

```
class DiagonalizaS(Matrix):
    def __init__(self, sistema, espectro, rep=0, repType=0, sust=[]):
        <<Texto de ayuda para la clase DiagonalizaS>>
        A = Matrix(sistema)
        if sust:
            A = A.fullcopy().subs(sust)
            D = A.diagonalizaS(espectro, sust=sust)
        else:
            D = A.diagonalizaS(espectro)

        if rep:
            nan   = sympy.symbols(r'\ ')
            dummyMatrix = M1(A.n)*nan
            if repType==1:
                nan   = sympy.symbols(r'\ ')
                D.tex = rprElimCF(A.apila(I(A.n),1), D.pasos)
            elif repType==2:
                nan   = sympy.symbols(r'\ ')
                D.tex = rprElimCF(A.apila(I(A.n),1).concatena(I(A.m).apila(dummyMatrix,1),1), D.pasos)
            elif repType==3:
                nan   = sympy.symbols(r'\ ')
                D.tex = rprElimFyC(A.apila(I(A.n),1), D.pasos)
            elif repType==4:
                nan   = sympy.symbols(r'\ ')
                D.tex = rprElimFyC(A.apila(I(A.n),1).concatena(I(A.m).apila(dummyMatrix,1),1), D.pasos)

            display(Math(D.tex))

        self.__dict__.update(D.__dict__)
```

```
        self.__class__ = type(sistema)
```

Es una forma alternativa de llamar al método `.diagonalizaO()`,

```
class DiagonalizaO(Matrix):
    def __init__(self, sistema, espectro):
        <<Texto de ayuda para la clase DiagonalizaO>>
        A = Matrix(sistema)
        D = A.diagonalizaO(espectro)

        self.__dict__.update(D.__dict__)
        self.__class__ = type(sistema)
```

Es una forma alternativa de llamar al método `.diagonalizaC()`,

```
class DiagonalizaC(Matrix):
    def __init__(self, sistema, rep=False, sust=[], variante=0):
        <<Texto de ayuda para la clase DiagonalizaC>>
        A = Matrix(sistema)
        D = A.diagonalizaC(rep, sust, variante)

        self.__dict__.update(D.__dict__)
        self.__class__ = type(sistema)
```

Parte II

La clase `T` (transformación elemental)

Capítulo 11

La clase T

11.1. Introducción

Vamos a implementar las transformaciones de un sistema A mediante secuencias de *transformaciones elementales de Tipo I y II* e *intercambios*. Para que funcionen las *transformaciones elementales* debe estar definida tanto la suma de los elementos del sistema como el producto de los elementos por un escalar:

Tipo I: $\mathsf{A}_{\substack{\boldsymbol{\tau} \\ [(\lambda)\boldsymbol{i}+\boldsymbol{j}]}}$ suma λ veces el elemento i al elemento j $(i \neq j)$;

Tipo II: $\mathsf{A}_{\substack{\boldsymbol{\tau} \\ [(\lambda)\boldsymbol{i}]}}$ multiplica el elemento i por $\lambda \neq 0$.

Intercambio: $\mathsf{A}_{\substack{\boldsymbol{\tau} \\ [\boldsymbol{i}=\boldsymbol{j}]}}$ intercambia los elementos i y j.

Comentario sobre la notación.

Cualquier transformación elemental de un sistema se puede lograr con el producto del sistema por una matriz elemental, y *la notación empleada busca parecerse a la notación del producto matricial* (por ello, para ilustrar lo que sigue usaremos como ejemplo de sistema una matriz $\mathbf{A}$):

> Al poner la *abreviatura* "$\boldsymbol{\tau}$" 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
>
> $$\mathbf{A}_{\boldsymbol{\tau}} = \mathbf{A}(\mathbf{I}_{\boldsymbol{\tau}}) \quad \text{donde la matriz } \mathbf{I} \text{ es de orden } n.$$
>
> De manera similar, al poner la *abreviatura* "$\boldsymbol{\tau}$" 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
>
> $$_{\boldsymbol{\tau}}\mathbf{A} = (_{\boldsymbol{\tau}}\mathbf{I})\mathbf{A} \quad \text{donde la matriz } \mathbf{I} \text{ es de orden } m.$$

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

Fíjese que una matriz cuadrada es un objeto muy "pesado"... una matriz de orden n posee n^2 coeficientes. Afortunadamente una matriz elemental es casi una matriz identidad salvo por el cambio de uno de sus elementos. Por tanto, para describir completamente una matriz elemental es necesario indicar su orden n y qué componente ha cambiado.[1] Por este motivo...

La ventaja de las transformaciones elementales es que omiten el orden n.

pues solo necesitan indicar el índice de los elementos que intervienen en la transformación.

Traducción de las transformaciones elementales a Python

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

Curso	Python	Curso	Python
$\underset{[i\rightleftharpoons j]}{\tau}\mathbf{A}$	`T( {i,j } ) & A`	$\mathbf{A}\underset{[i\rightleftharpoons j]}{\tau}$	`A & T( {i,j} )`
$\underset{[(a)j]}{\tau}\mathbf{A}$	`T( (a,j) ) & A`	$\mathbf{A}\underset{[(a)j]}{\tau}$	`A & T( (a,j) )`
$\underset{[(a)i+j]}{\tau}\mathbf{A}$	`T( (a,i,j) ) & A`	$\mathbf{A}\underset{[(a)i+j]}{\tau}$	`A & T( (a,i,j) )`

El código Python de la columna izquierda, donde la transformación se aplica por la izquierda, solo será aplicable a los `BlockM` (solo ellos tienen "filas"). El código Python de la columna derecha será aplicable a los `Sistemas`, pues opera con son elementos.

Vemos que:

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

Empleando listas de abreviaturas extendemos la notación para expresar secuencias de transformaciones elementales, es decir, $\boldsymbol{\tau}_1 \cdots \boldsymbol{\tau}_k$. Así logramos la siguiente equivalencia entre expresiones

$$\texttt{T(}\, t_1\texttt{) \& T(}\, t_2\texttt{) \&}\cdots\texttt{\& T(}\, t_k\texttt{)} \quad = \quad \texttt{T([}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{])}$$

De esta manera

$$\mathbf{A}_{\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k}: \qquad \mathbf{A}\texttt{ \& T(}\, t_1\texttt{) \& T(}\, t_2\texttt{) \&}\cdots\texttt{\& T(}\, t_k\texttt{)} = \mathbf{A}\texttt{ \& T([}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{])}$$

$$\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k\mathbf{A}: \qquad \texttt{T(}\, t_1\texttt{) \& T(}\, t_2\texttt{) \&}\cdots\texttt{\& T(}\, t_k\texttt{)}\ \&\ \mathbf{A} = \texttt{T([}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{])}\ \&\ \mathbf{A}.$$

[1] Fíjese que la notación usada en el libro del curso de Álgebra Lineal para las matrices elementales no las describe completamente; se deja al lector la deducción de cuál es el orden de la matriz elemental necesario para poder realizar el producto $\mathbf{A}\left(\mathbf{I}_{\boldsymbol{\tau}}\right)$ o el producto $\left(_{\boldsymbol{\tau}}\mathbf{I}\right)\mathbf{A}$.

Así, usando abreviaturas y si $\mathbf{A}$ es de orden $m \times n$, el primer caso es equivalente a escribir el producto de matrices

$$\mathbf{A}_{\tau_1 \cdots \tau_k} \;=\; \mathbf{A}(\mathbf{I}_{\tau_1})(\mathbf{I}_{\tau_2}) \cdots (\mathbf{I}_{\tau_k}) \qquad \text{donde } \mathbf{I} \text{ es de orden } n;$$

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

$$_{\tau_1 \cdots \tau_k}\mathbf{A} \;=\; (_{\tau_1}\mathbf{I})(_{\tau_2}\mathbf{I}) \cdots (_{\tau_k}\mathbf{I})\mathbf{A} \qquad \text{donde } \mathbf{I} \text{ es de orden } m...$$

¡Pero gracias a las abreviaturas de las transformaciones elementales no es necesario indicar el orden (el número de filas y columnas) de las matrices elementales en ningún momento!

Necesidad de creación de una nueva clase

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

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

podríamos pensar que podemos implementar la transformación elemental como un método de la clase `Sistema`. Así, al definir el método `__and__` por la derecha del `Sistema` 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 $\Big[\Big[\big[\mathbf{A} \;\texttt{ \& T(}\, t_1\texttt{)}\big]\texttt{\& T(}\, t_2\, \texttt{)}\Big]\texttt{\&}\cdots\Big]\texttt{\& T(}\, t_k\texttt{)}$ donde la expresión dentro de cada corchete es una `Matrix`, por lo que las operaciones están bien definidas. La dificultad aparece con

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

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 subclase `BlockM` (recuerde que al actuar por la izquierda se opera con las filas, y solo los `BlockMs` tiene definidas las filas).

Así pues, definiremos una nueva clase que almacene las *abreviaturas* t_i ("$\boldsymbol{\tau}_i$") de las operaciones elementales, de manera que podamos definir `T(` t_i `) & T(` t_j`)`, como un método que "compone" dos transformaciones elementales ("$\boldsymbol{\tau}_i\boldsymbol{\tau}_j$") para formar una secuencia de abreviaturas (que en última instancia será una secuencia de operaciones a ejecutar sobre un `Sistema`).

El nuevo objeto, `T` ("transformación elemental"), 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 lista de objetos `T`) empleada para instanciar.

11.2. Resumen de uso de la clase `T`

Una transformación elemental se crea con su *abreviatura*, y una secuencia de transformaciones con la lista de sus abreviaturas:

Expresión	Transformación
`T( (a,i,j) )`	Tipo I: suma `a` veces el elemento `i` al elemento `j`
`T( (a,i) )`	Tipo II: multiplica el elemento `i` por `a` (no nulo)
`T( {i,j} )`	intercambio de los elementos `i` y `j`
`T( [t1, t2, ..., tk] )`	secuencia: primero `t1`, luego `t2`, etc.

Se aplican con el operador `&`: a la derecha del sistema actúan sobre sus elementos (las columnas, si es una matriz) y a la izquierda de una matriz actúan sobre sus filas.

Expresión	Resultado
`A & t`	transforma los elementos (columnas) de `A`
`t & A`	transforma las filas de `A`
`t1 & t2`	composición: la secuencia formada por `t1` seguida de `t2`
`~t`	transpuesta: las mismas transformaciones en orden inverso
`t**-1`	inversa: la secuencia que deshace `t`
`t**n`	`t` repetida `n` veces
`t.espejo()`	transformación «espejo»: intercambia los papeles de los índices en las de Tipo I
`t.subs(reglas)`	sustituye símbolos en los coeficientes
`t.abreviaturas`	lista de las abreviaturas de la secuencia

Aviso importante. Aplicar una transformación **modifica el propio objeto**: tras ejecutar `A & t`, la matriz `A` ha cambiado (el resultado de la expresión es la misma `A`, ya transformada). Para conservar la matriz original hay que transformar una copia: `Matrix(A) & t` o `A.fullcopy() & t`.

El módulo comprueba que las abreviaturas corresponden a transformaciones elementales: ni `T( (a,i,i) )` ni `T( (0,i) )` lo son y producen un error.

11.2.1. Ejemplos de transformaciones elementales

```
A = Matrix( [ [1,2,3], [4,5,6] ] )
```

Sumamos a la tercera columna el doble de la primera (sobre una copia, para no modificar `A`):

```
print(dmath( Matrix(A) & T( (2,1,3) ) ))
```

$$\begin{bmatrix} 1 & 2 & 5 \\ 4 & 5 & 14 \end{bmatrix}$$

La misma abreviatura a la izquierda actúa sobre las filas. Aquí multiplicamos por 2 la primera fila:

```
print(dmath( T( (2,1) ) & Matrix(A) ))
```

$$\begin{bmatrix} 2 & 4 & 6 \\ 4 & 5 & 6 \end{bmatrix}$$

Un intercambio de columnas y otro de filas:

```
print(dmath( T( {1,2} ) & Matrix(A) & T( {1,3} ) ))
```

$$\begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

Una secuencia de transformaciones se escribe con una lista, o componiendo transformaciones con `&`:

```
t = T( [ (2,1,3), (5,2) ] )
```

```
print(dprint( t == T( (2,1,3) ) & T( (5,2) ) ))
```

```
True
```

```
print(dmath( Matrix(A) & t ))
```

$$\begin{bmatrix} 1 & 10 & 5 \\ 4 & 25 & 14 \end{bmatrix}$$

En LaTeX una transformación se representa con la letra τ y, debajo, las abreviaturas en el orden en que se aplican:

```
print(dmath( t ))
```

$$\begin{matrix} \boldsymbol{\tau} \\ [(2)\mathbf{1}{+}\mathbf{3}] \\ [(5)\mathbf{2}] \end{matrix}$$

La inversa deshace la secuencia (las inversas de cada transformación, en orden inverso):

```
print(dprint( t**-1 ))
```

```
T([(1/5, 2), (-2, 1, 3)])
```

```
print(dprint( Matrix(A) & t & t**-1 == A ))
```

```
True
```

La transpuesta invierte el orden de la secuencia. Sirve para transponer un producto: la transpuesta de `A & t` es `~t & ~A`.

```
print(dprint( ~t ))
```

```
T([(5, 2), (2, 1, 3)])
```

```
print(dprint( ~(Matrix(A) & t) == ~t & ~A ))
```

```
True
```

La transformación espejo de `(a,i,j)` es `(a,j,i)`. Junto con la inversa permite las diagonalizaciones por semejanza:

```
print(dprint( T( (2,1,3) ).espejo() ))
```

```
T([(2, 3, 1)])
```

Aplicar una transformación a la matriz identidad da la matriz elemental correspondiente:

```
print(dmath( I(3) & T( (2,1,3) ) ))
```

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

Las transformaciones también se aplican a vectores y a sistemas cualesquiera cuyos elementos se puedan sumar y multiplicar por números:

```
print(dmath( Sistema( [ Vector([1,0]), Vector([0,1]), Vector([1,1]) ] ) & T( (2,1,3) ) ))
```

$$\begin{bmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}; & \begin{pmatrix} 0 \\ 1 \end{pmatrix}; & \begin{pmatrix} 3 \\ 1 \end{pmatrix}; \end{bmatrix}$$

Sin copia, la transformación modifica la matriz:

```
A & T( {1,3} )
print(dmath( A ))
```

$$\begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \end{bmatrix}$$

Una abreviatura que no corresponde a una transformación elemental produce un error:

```
try:
    T( (0,2) )
except ValueError as error:
    print(dprint( error ))
```

```
T( (0, i) ) no es una transformación elemental
```

11.3. Implementación de las transformaciones elementales

11.3.1. Texto de ayuda

Texto de ayuda de la clase T

```
"""Clase para las transformaciones elementales

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

Una T (transformación elemental) se puede instanciar indicando las
operaciones mediante un número arbitrario de
```

```
 1. abreviaturas(set): {índice, índice}. Abrev. de un intercambio de
                       entre los elementos correspondientes a dichos
                       índices

              (tuple): (escalar, índice). Abrev. transf. Tipo II que
                       multiplica el elemento correspondiente al
                       índice por el escalar

                       (escalar, índice1, índice2). Abrev.
                       transformación Tipo I que suma al elemento
                       correspondiente al índice2 el elemento
                       correspondiente al índice1 multiplicado por
                       el escalar

 2. transf. Elems.(T): Genera otra T cuyo atributo .abreviaturas es
                       una copia del atributo .abreviaturas de la
                       transformación dada

 3.      listas(list): Con cualesquiera de los anteriores objetos o
                       con sublistas formadas con los anteriores
                       objetos. Genera una T cuyo atributo
                       .abreviaturas es una concatenación de todas las
                       abreviaturas


Atributos:

   abreviaturas (set): lista con las abreviaturas de todas las
                       transformaciones

            rpr (str): Si rpr='v' (valor por defecto) se muestra la
                       lista de breviaturas en vertical. Con cualquier
                       otro valor se muestran en horizontal.

Ejemplos:
>>> # Intercambio entre elementoes
>>> T( {1,2} )
T([{1, 2}])

>>> # Transformación Tipo II (multiplica por 5 el segundo elemento)
>>> T( (5,2) )
T([(5, 2)])

>>> # Transformación Tipo I (resta el tercer elemento al primero)
>>> T( (-1,3,1) )
T([(-1, 3, 1)])

>>> # Secuencia de las tres transformaciones anteriores
>>> T( [{1,2}, (5,2), (-1,3,1)] )
T([{1, 2}, (5, 2), (-1, 3, 1)])

>>> # T de una T
>>> T( T( (5,2) ) )
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)])

"""
```

11.3.2. Método de inicialización

Inicialización de la clase T

```
def __init__(self, *args, rpr='v'):
    """Inicializa una transformación elemental"""
```

```
    <<Verificación de que las .abreviaturas corresponden a transformaciones elementales o intercambios>>
    <<Simplificacion de expresiones simbólicas en las abreviaturas>>
    def CreaListaAbreviaturas(arg):
        if isinstance(arg, (tuple, set)):
            verificacion(arg)
            arg = simplificacionSimbolica(arg)
            return [arg]
        if isinstance(arg, list):
            return [abrv for item in arg for abrv in CreaListaAbreviaturas(item)]
        if isinstance(arg, T):
            return CreaListaAbreviaturas(arg.abreviaturas)

    def concatenaTodasLasAbreviaturasDeLos(args):
        return [abrv for item in args for abrv in CreaListaAbreviaturas(item)]

    self.abreviaturas = concatenaTodasLasAbreviaturasDeLos(args)
    self.rpr          = rpr
```

Verificación de que las abreviaturas corresponden a transformaciones elementales o intercambios

Un transformación elemental no puede multiplicar ningún elemento por cero, ni sumar a un elemento un múltiplo de si mismo. Además, un intercambio solo tiene sentido a lo sumo entre dos elementos.

Verificación de que las .abreviaturas corresponden a transformaciones elementales o intercambios

```
def verificacion(abrv):
    if isinstance(abrv,tuple) and (len(abrv) == 2) and abrv[0]==0:
        raise ValueError(texto_de_salida('no_es_transformacion').format('(0, i)'))
    if isinstance(abrv,tuple) and (len(abrv) == 3) and (abrv[1] == abrv[2]):
        raise ValueError(texto_de_salida('no_es_transformacion').format('(a, i, i)'))
    if isinstance(abrv,set) and len(abrv)>2:
        raise ValueError(texto_de_salida('no_es_intercambio'))
```

Simplificacion de expresiones simbólicas en las abreviaturas

Simplificacion de expresiones simbólicas en las abreviaturas

```
def simplificacionSimbolica(arg):
    if isinstance(arg,tuple) and (len(arg) == 2):
        return (sympy.factor(racional(arg[0])), arg[1],)
    elif isinstance(arg,tuple) and (len(arg) == 3):
        return (sympy.factor(racional(arg[0])), arg[1], arg[2],)
    else:
        return arg
```

11.4. Composición de Transformaciones o transformación de un `Sistema` por la izquierda

11.4.1. Texto de ayuda

Texto de ayuda para la composición de Transformaciones Elementales

```
"""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 sus filas

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), {1, 3}])

>>> # Transformación de las filas de una Matrix
>>> # Multiplica por 4 la segunda fila y luego intercambia las dos primeras filas
>>> T( [{1,2}, (4,2)] ) & Matrix([[1,2],[3,4]])
Matrix([Vector([12, 1]), Vector([16, 2])])

"""
```

11.4.2. Implementación

Describimos la composición de transformaciones `T(`t_1`) & T(`t_2`)` creando una lista de abreviaturas $[\tau_1, \tau_2]$ (mediante la concatenación de listas)[2]. Si `other` es un `BlockV` o una `BlockM`, se llama al método `__rand__` de la clase `other` (que transformará los elementos del vector en el primer caso, y las filas de la matriz en el segundo; y que veremos más adelante).

Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix

```
def __and__(self, other):
    <<Texto de ayuda para la composición de Transformaciones Elementales>>
    if isinstance(other, T):
        return T(self.abreviaturas+other.abreviaturas, rpr=self.rpr)

    if isinstance(other, Sistema):
        return other.__rand__(self)
```

11.5. Transposición de transformaciones elementales

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

$$(\mathbf{I}_{\tau_1 \tau_2 \cdots \tau_k})^\mathsf{T} = \Big((\mathbf{I}_{\tau_1})(\mathbf{I}_{\tau_2}) \cdots (\mathbf{I}_{\tau_k})\Big)^\mathsf{T} = (\mathbf{I}_{\tau_k})^\mathsf{T} \cdots (\mathbf{I}_{\tau_2})^\mathsf{T} (\mathbf{I}_{\tau_1})^\mathsf{T} = ({}_{\tau_k}\mathbf{I}) \cdots ({}_{\tau_2}\mathbf{I})({}_{\tau_1}\mathbf{I}) = {}_{\tau_k \cdots \tau_2 \tau_1}\mathbf{I}$$

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:

$$(\tau_1 \cdots \tau_k)^\mathsf{T} = \tau_k \cdots \tau_1;$$

así

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

[2] Recuerde que la suma de listas (`list + list`) concatena las listas

Fíjese que efectivamente hemos logrado que la notación con abreviaturas se comporte como la notación matricial!

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.

Operador transposición para la clase T

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

11.6. Potencias e inversa de transformaciones elementales

Toda matriz de la forma $\mathsf{I}_{\tau_1\cdots\tau_k}$ ó ${}_{\tau_1\cdots\tau_k}\mathsf{I}$ es invertible por ser producto de matrices elementales:

$$\left(\mathsf{I}_{\tau_1\cdots\tau_k}\right)^{\mathsf{T}}\left(\mathsf{I}_{\tau_k^{-1}\cdots\tau_1^{-1}}\right)^{\mathsf{T}} = \mathsf{I}_{\tau_1\cdots\tau_k\cdot\tau_k^{-1}\cdots\tau_1^{-1}} = \mathsf{I};$$

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

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

El siguiente método devuelve la potencia `n`-ésima de una transformación elemental y cuando `n` es -1 dicha potencia es la inversa:

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

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

que en Jupyter veríamos como $\begin{matrix}\boldsymbol{\tau}\\ [\mathbf{1}\rightleftharpoons\mathbf{2}]\\ [(3)\mathbf{2}]\\ [(-1)\mathbf{2}+\mathbf{3}]\end{matrix}$

Al implementar el método, definimos la potencia de manera recursiva (con una función auxiliar `lambda`). Además, si `n` es cero, devolveremos una transformación que no haga nada (identidad); por ejemplo $\underset{[\mathbf{1}\rightleftharpoons\mathbf{1}]}{\boldsymbol{\tau}}$.

Potencia de una T

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

    potencia = lambda T, n: T if n==1 else T & potencia(T, n-1)
    TransformacionElemental = potencia(self,abs(n)) if n!=0  else  T({1})

    return TransformacionElemental if n>0 else Tinversa(TransformacionElemental)
```

Método auxiliar que calcula la inversa de una Transformación elemental

```
def Tinversa ( self ):
    """Calculo de la inversa de una transformación elemental"""
    operaciones = [                      abrv     if isinstance(abrv,set) else \
```

```
                    ( -abrv[0], abrv[1], abrv[2]) if len(abrv)==3         else \
                    (fracc(1,abrv[0]),   abrv[1])   for abrv in CreaLista(self.abreviaturas) ]

    return ~T( operaciones, rpr=self.rpr)
```

11.7. Transformaciones elementales "espejo"

Al diagonalizar por semejanza, y aplicar transformaciones elementales por la derecha, que es lo mismo que multiplicar por una matriz invertible por la derecha, necesitaremos expresar la correspondiente matriz inversa mediante una secuencia de transformaciones elementales de la filas de la matriz identidad. Esto se logra con el método `espejo`.[3]

Transformación elemental espejo de una T

```
def espejo ( self ):
    """Calculo de la transformación elemental espejo de otra"""
    return T([ (abrv[0], abrv[2], abrv[1]) if len(abrv)==3 else abrv for abrv in
    ↪  CreaLista(self.abreviaturas)], rpr=self.rpr)
```

11.8. Sustitución de variables simbólicas

Sustituye la variable `c` por `v`, donde `v` puede ser un valor, u otra variable simbólica.

Sustitución de variables simbólicas en una Transformación elemental

```
def subs(self, regla_de_sustitucion=[]):
    '''Sustitución simbólica en transformaciones elementales'''

    def sustitucion(operacion, regla_de_sustitucion):
        if isinstance(operacion, tuple):
            return tuple(sympy.S(operacion).subs(CreaLista(regla_de_sustitucion)) )
        elif isinstance(operacion, set):
            return set(sympy.S(operacion).subs(CreaLista(regla_de_sustitucion)) )
        elif isinstance(operacion, list):
            return [sustitucion(item, regla_de_sustitucion) for item in operacion]
        elif isinstance(operacion, T):
            return operacion.subs(CreaLista(regla_de_sustitucion))

    self = T([sustitucion(operacion, regla_de_sustitucion) for operacion in self.abreviaturas])
    return self

def con_coeficientes(self, funcion):
    """Aplica la función al coeficiente de cada transformación elemental

    Solo cambia el número que multiplica (el primer elemento de las
    abreviaturas de Tipo I y de Tipo II); los índices y los
    intercambios quedan como están. Modifica la T y la devuelve.
    """
    def nueva(abrv):
        if isinstance(abrv, tuple) and len(abrv) in (2, 3):
            return (funcion(abrv[0]),) + tuple(abrv[1:])
        elif isinstance(abrv, list):
            return [nueva(item) for item in abrv]
        elif isinstance(abrv, T):
            return abrv.con_coeficientes(funcion)
        else:
            return abrv
```

[3] Al no encontrar ningún nombre en los manuales de Álgebra Lineal para este concepto, he adoptado este descriptivo nombre.

```
    self.abreviaturas = [nueva(abrv) for abrv in self.abreviaturas]
    return self

def simplify(self):
    """Simplifica los coeficientes de la transformación elemental

    >>> a = sympy.symbols('a')
    >>> T([{1, 2}, (a*(a + 1) - a, 2)]).simplify()
    T([{1, 2}, (a**2, 2)])
    """
    return self.con_coeficientes(simplify)

def factor(self):
    """Factoriza los coeficientes de la transformación elemental

    >>> a = sympy.symbols('a')
    >>> T([{1, 2}, (a**2 - 1, 2)]).factor()
    T([{1, 2}, ((a - 1)*(a + 1), 2)])
    """
    return self.con_coeficientes(factor)

def expand(self):
    """Desarrolla los coeficientes de la transformación elemental

    >>> a = sympy.symbols('a')
    >>> T([{1, 2}, ((a - 1)*(a + 1), 2, 1)]).expand()
    T([{1, 2}, (a**2 - 1, 2, 1)])
    """
    return self.con_coeficientes(expand)
```

11.9. Métodos similares a los de una `list`

11.9.1. T es iterable

Haremos que `T` sea iterable con los procedimientos "mágicos" `__getitem__`, que permite seleccionar pasos de `T`, y `__setitem__`, que permite modificar pasos de `T`. Con `__len__` podremos contar el número de pasos de `T`.

```
def __getitem__(self,i):
    """ Devuelve las transformaciones elementales del i-ésimo paso """
    return T(self.abreviaturas[i])

def __setitem__(self,i,value):
    """ Modifica las transformaciones elementales del i-ésimo paso """
    self.abreviaturas[i]=value

def __len__(self):
    """Número de pasos de T """
    return len(self.abreviaturas)
```

11.9.2. Igualdad entre transformaciones elementales

Igualdad entre transformaciones elementales

```
def __eq__(self, other):
    """Indica si es cierto que dos Transformaciones elementales son iguales"""
    return isinstance(other, T) and self.abreviaturas == other.abreviaturas
```

11.10. 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} )`	$\boldsymbol{\tau}_{[\mathbf{1}\rightleftharpoons\mathbf{5}]}$
`T( (6, 2) )`	$\boldsymbol{\tau}_{[(6)\mathbf{2}]}$
`T( (-1, 2, 3) )`	$\boldsymbol{\tau}_{[(-1)\mathbf{2}+\mathbf{3}]}$

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
$\boldsymbol{\tau}_{[\boldsymbol{i}\rightleftharpoons\boldsymbol{j}]}\mathbf{A}$	`T( {i,j} ) & A`	$\mathbf{A}\,\boldsymbol{\tau}_{[\boldsymbol{i}\rightleftharpoons\boldsymbol{j}]}$	`A & T( {i,j} )`
$\boldsymbol{\tau}_{[(a)\boldsymbol{i}]}\mathbf{A}$	`T( (a,i) ) & A`	$\mathbf{A}\,\boldsymbol{\tau}_{[(a)\boldsymbol{j}]}$	`A & T( (a,j) )`
$\boldsymbol{\tau}_{[(a)\boldsymbol{i}+\boldsymbol{j}]}\mathbf{A}$	`T( (a,i,j) ) & A`	$\mathbf{A}\,\boldsymbol{\tau}_{[(a)\boldsymbol{i}+\boldsymbol{j}]}$	`A & T( (a,i,j) )`

11.10.1. 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( [ (1, 2), {3, 4} ] )`

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

- `A & T( (2, 1) ) & T( {3, 4} )` (indicando las transformaciones de una en una)
- `A & T( [(2, 1), {3, 4}] )` (usando la transformación composición de ambas)

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

- `T( (2, 1) ) & T( {3, 4} ) & A`
- `T( [(2, 1), {3, 4}] ) & A`

Representación de una secuencia de transformaciones

Representación en la consola de Python	Representación en Jupyter
`T( [ (2, 1), (1, 3, 2) ] )`	$\boldsymbol{\tau}$ $\left[(2)\mathbf{1}\right]$ $\left[(1)\mathbf{3}+\mathbf{2}\right]$

Representación de transformaciones identidad

Si las transfomaciones multiplican un vector por 1, y suman un vector nulo a otro vector, dichas transformaciones no cabian el sistema de vectores. Lo habitual es que si un paso no modifica nada, que no se represente, por ello se filtran los pasos con el procedimiento Cribado de secuencias de transformaciones; si, a resultas del filtrado, la lista de abreviaturas es vacía entonces la representación en LaTeX es una cadena vacía (no se pinta ningún símbolo en Jupyter). Si el atributo `rpr` es distinto de `'v'` la representación en Jupiter se realiza en horizontal.

Representación de la clase T

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

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

def latex(self):
    """ Construye el comando LaTeX para representar una Trans. Elem. """
    def simbolo(t):
        """Escribe el símbolo que denota una transformación elemental particular"""
        if isinstance(t,(set,sympy.sets.sets.FiniteSet)):
            return '\\left[\\mathbf{' + latex(list(t)[0]) + \
              '}\\rightleftharpoons\\mathbf{' + latex(list(t)[-1]) + '}\\right]'
        if isinstance(t,(tuple, sympy.core.containers.Tuple)) and len(t) == 2:
            return '\\left[\\left(' + \
              latex(t[0]) + '\\right)\\mathbf{'+ latex(t[1]) + '}\\right]'
        if isinstance(t,(tuple, sympy.core.containers.Tuple)) and len(t) == 3:
            return '\\left[\\left(' + latex(t[0]) + '\\right)\\mathbf{' + \
              latex(t[1]) + '}' + '+\\mathbf{' + latex(t[2]) + '} \\right]'

    if isinstance(self.abreviaturas, (set, tuple) ):
        return '\\underset{' + simbolo(self.abreviaturas) + '}{\\pmb{\\tau}}'

    elif self.abreviaturas == []:
        return ' '

    elif isinstance(self.abreviaturas, list) and self.rpr=='v':
        return '\\underset{\\begin{subarray}{c} ' + \
              '\\\\'.join([simbolo(i) for i in self.abreviaturas])  + \
              '\\end{subarray}}{\\pmb{\\tau}}'

    elif isinstance(self.abreviaturas, list):
        return '\\underset{' + \
               '}{\\pmb{\\tau}}\\underset{'.join([simbolo(i) for i in self.abreviaturas]) + \
```

```
'}{\\pmb{\\tau}}'
```

11.11. La clase `T` completa

Definición de la clase T

```
class T:
    <<Texto de ayuda de la clase ~T~>>
    <<Inicialización de la clase ~T~>>
    <<Composición de Transformaciones Elementales o aplicación sobre las filas de una Matrix>>
    <<Operador transposición para la clase ~T~>>
    <<Método auxiliar que calcula la inversa de una Transformación elemental>>
    <<Potencia de una ~T~>>
    <<Transformación elemental espejo de una ~T~>>
    <<Sustitución de variables simbólicas en una Transformación elemental>>
    <<Métodos de la clase ~T~ para que actúe como si fuera una list de Python>>
    <<Representación de la clase ~T~>>
```

Parte III

Las clases `SubEspacio` y `EAfin`

Capítulo 12

La clase `SubEspacio` (de $\mathbb{R}^m$)

El conjunto de vectores $\boldsymbol{x}$ que resuelven el sistema $\mathbf{A}\boldsymbol{x} = \mathbf{0}$ es un subespacio de $\mathbb{R}^n$; y el conjunto de vectores $\boldsymbol{x}$ que resuelven el sistema $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ con $\boldsymbol{b} \neq \mathbf{0}$ es un espacio afín de $\mathbb{R}^n$. En este capítulo vamos a definir objetos que representen estos subconjuntos de $\mathbb{R}^n$ (véase la definición de subespacio en el libro).

Un subespacio se puede describir de dos maneras, y la clase `SubEspacio` se puede instanciar con cualquiera de ellas:

- con un `Sistema` de `Vectores`: `SubEspacio(S)` es el conjunto de las combinaciones lineales de los vectores de `S` (ecuaciones *paramétricas*);
- con una `Matrix`: `SubEspacio(M)` es el conjunto de las soluciones del sistema homogéneo $\mathbf{M}\boldsymbol{v} = \mathbf{0}$ (ecuaciones *cartesianas*).

Sea cual sea la manera de crearlo, el objeto guarda las dos descripciones, y su representación muestra las ecuaciones paramétricas a la izquierda y las cartesianas a la derecha. Los métodos `span` y `espacio_nulo` de un `Sistema` devuelven estos mismos objetos: `S.span()` es `SubEspacio(S)` y `M.espacio_nulo()` es `SubEspacio(M)`.

Atributo	Contenido
`dim`	dimensión del subespacio
`Rn`	número de componentes de sus vectores (es un subespacio de $\mathbb{R}^n$)
`base`	una base (`Sistema` de `Vectores`); es un `Sistema` vacío si la dimensión es cero
`sgen`	sistema generador de las ecuaciones paramétricas; si la dimensión es cero, el vector nulo
`cart`	`Matrix` de coeficientes de las ecuaciones cartesianas

Expresión	Resultado
`v in A`	indica si el vector `v` pertenece al subespacio `A`
`A` = B=, `A !` B=	indica si son (o no son) el mismo subespacio
`A.contenido_en(B)`	indica si `A` es un subconjunto de `B`
`A + B`	suma de subespacios
`A & B`	intersección
`~A`	complemento ortogonal (el mayor subespacio perpendicular)
`A.EcParametricas()`	código LaTeX de las ecuaciones paramétricas
`A.EcCartesianas()`	código LaTeX de las ecuaciones cartesianas

12.1. Ejemplos de subespacios

El subespacio generado por un sistema de tres vectores de $\mathbb{R}^3$:

```
S = Sistema( [ Vector([0,1,0]), Vector([2,1,3]), Vector([2,10,3]) ] )
A = SubEspacio(S)
```

```
print(dmath( A ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^2,\ \boldsymbol{v} = \begin{bmatrix} 0 & 2 \\ 1 & 1 \\ 0 & 3 \end{bmatrix} \boldsymbol{p} \right\} = \left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} -3 & 0 & 2 \end{bmatrix} \boldsymbol{v} = \begin{pmatrix} 0, \end{pmatrix} \right\}$$

El tercer vector de `S` es combinación lineal de los otros dos, así que la dimensión es dos y el sistema generador que guarda el objeto solo tiene dos vectores:

```
print(dprint( A.dim ))
```

```
2
```

```
print(dprint( A.sgen ))
```

```
[(0, 1, 0,); (2, 1, 3,);]
```

La matriz de las ecuaciones cartesianas es

```
print(dmath( A.cart ))
```

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

El mismo subespacio se obtiene como conjunto de soluciones de un sistema homogéneo, es decir, a partir de una matriz:

```
M = Matrix( [ [-3,0,2], [6,0,-4] ] )
B = SubEspacio(M)
```

```
print(dprint( A == B ))
```

```
True
```

En modo texto se muestra la expresión que crea el subespacio a partir de su sistema generador:

```
print(dprint( B ))
```

```
SubEspacio(Sistema([Vector([0, 1, 0]), Vector([2, 0, 3])]))
```

Para saber si un vector pertenece al subespacio se usa `in`:

```
print(dprint( Vector([2,5,3]) in A ))
```

```
True
```

```
print(dprint( Vector([1,0,0]) in A ))
```

```
False
```

El complemento ortogonal de `A` es la recta generada por la fila de la matriz de las ecuaciones cartesianas:

```
print(dmath( ~A ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^1,\ \boldsymbol{v} = \begin{bmatrix} -3 \\ 0 \\ 2 \end{bmatrix} \boldsymbol{p} \right\} = \left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 3 \end{bmatrix} \boldsymbol{v} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

Con otro subespacio `C` (una recta que no está contenida en `A`)

```
C = SubEspacio( Sistema( [ Vector([1,0,0]) ] ) )
```

```
print(dprint( C.contenido_en(A) ))
```

```
False
```

la intersección solo contiene el vector nulo

```
print(dmath( A & C ))
```

$$\left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$$

y la suma es todo $\mathbb{R}^3$:

```
print(dprint( (A + C).dim ))
```

```
3
```

Las ecuaciones paramétricas o las cartesianas se pueden obtener por separado:

```
print(displaymath( A.EcCartesianas() ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} -3 & 0 & 2 \end{bmatrix} \boldsymbol{v} = \begin{pmatrix} 0, \end{pmatrix} \right\}$$

12.2. Implementación de `SubEspacio`

La implementación requiere encontrar un `Sistema` base del `SubEspacio` columna de una `Matrix` `A`. Lo haremos pre-escalonando una `Matrix` `A` con `elim(0)` (así evitamos las fracciones en la medida

de lo posible). También necesitaremos encontrar un sistema generador del un espacio nulo de `A`. Lo haremos con el método auxiliar `SGenENulo`.

Inicialización de la clase SubEspacio

```
def __init__(self, data, sust=[], Rn=[]):
    """Inicializa un SubEspacio de Rn"""
    <<Método auxiliar ~sistema_generador_del_espacio_nulo_de~ una ~Matrix~>>
    if not isinstance(data, Sistema):
        raise ValueError(' Argumento debe ser un Sistema o Matrix ')

    if not data:
        if not Rn:
            raise ValueError(' Si el sistema es vacio, es necesario indicar el espacio Rn ')
        else:
            self.Rn = Rn
            self.__dict__ = SubEspacio(Sistema([self.vector_nulo()])).__dict__.copy()

    elif isinstance(data, Matrix):
        A         = data
        self.Rn   = Rn if Rn else A.n
        self.sgen = sistema_generador_del_espacio_nulo_de(A, sust, Rn)
        self.dim  = 0 if self.sgen.es_nulo() else len(self.sgen)
        self.base = self.sgen if self.dim else Sistema([])
        self.cart = SubEspacio(self.sgen, sust, self.Rn).cart

    else:
        if isinstance(data|1, BlockM):
            self.Rn = ((data|1).m, (data|1).n)
        elif isinstance(data|1, Sistema):
            self.Rn =  (data|1).n

        try:
            A = Matrix(data).subs(sust)
        except Exception:
            A = BlockM([data]).subs(sust)
        self.base = Sistema([data|j for j,v in enumerate(A.elim(0), 1) if v.no_es_nulo()])
        self.dim  = len(self.base)
        self.sgen = self.base if self.base else Sistema([ self.vector_nulo() ])

        if isinstance(self.Rn, int):
            self.cart = ~Matrix(sistema_generador_del_espacio_nulo_de(~A, sust))
        elif isinstance(self.Rn, tuple):
            self.cart = SubEspacio(Sistema([m.vec() for m in self.sgen]), Rn=self.Rn).cart
```

Las columnas de `E` correspondientes a los elementos nulos de `K` son una base del espacio nulo.

Método auxiliar sistema_generador_del_espacio_nulo_de una Matrix

```
def sistema_generador_del_espacio_nulo_de(A, sust=[], Rn=[]):
    """Encuentra un sistema generador del Espacio Nulo de A"""
    K = A.K(0, sust);
    E = I(A.n) & T(K.pasos[1])
    lista = [ v.reshape(Rn) for j, v in enumerate(E,1) if (K|j).es_nulo(sust) ]
    return Sistema(lista) if lista else Sistema([self.vector_nulo()])
```

Método auxiliar para establecer el vector nulo del subespacio

```
def vector_nulo(self):
    return M0(self.Rn[0],self.Rn[1]) if isinstance(self.Rn, tuple) else V0(self.Rn)
```

12.3. Métodos de la clase `SubEspacio`

Definimos un método que nos indique si es cierto que un `SubEspacio` está contenido en otro (`contenido_en`). Si `A` y `B` son `SubEspacios`, la siguiente expresión

`A.contenido_en(B)`

nos dirá si es cierto que `A` es un `SubEspacio` de `B`.

Para comprobar si un `SubEspacio A` está contenido en un `SubEspacio B`, basta verificar si todos los vectores del sistema generador de `A` son solución de las ecuaciones cartesianas de `B`. Si `B` es un `EAfin`, entonces `B.v` debe ser nulo y `A` debe estar contenido en `B.S`.

```
def contenido_en(self, other):
    """Indica si este SubEspacio está contenido en other"""
    self.verificacion(other)
    if isinstance(other, SubEspacio):
        return all ([ (other.cart*v.vec()).es_nulo() for v in self.sgen ])
        #if isinstance(self.sgen[1, Vector):
        #    return all ([ (other.cart*v).es_nulo() for v in self.sgen ])
        #elif isinstance(self.sgen[1, Matrix):
        #    return all ([ (other.cart*v.vec()).es_nulo() for v in self.sgen ])

    elif isinstance(other, EAfin):
        return other.v.es_nulo() and self.contenido_en(other.S)
```

También definimos dos métodos (mágicos) que nos indican

- si dos `SubEspacios` son iguales (`__eq__`), es decir, que `A` esta contenido en `B` y viceversa; o
- si son distintos (`__ne__`), es decir, que no son iguales.

Así podemos usar las siguientes expresiones booleanas

`A == B` y `A != B`

```
def __eq__(self, other):
    """Indica si un subespacio de Rn es igual a otro"""
    self.verificacion(other)
    return self.contenido_en(other) and other.contenido_en(self)

def __ne__(self, other):
    """Indica si un subespacio de Rn es distinto de otro"""
    self.verificacion(other)
    return not (self == other)
```

Para que estos tres métodos funcionen es necesario un método auxiliar que realice la `verificacion` de que los dos argumentos son `SubEspacios` o `EAfines` del mismo espacio vectorial $\mathbb{R}^m$ (como este método tampoco es mágico, se invoca con `self.verificacion()`).

```
def verificacion(self,other):
    if not isinstance(other, (SubEspacio, EAfin)) or  not self.Rn == other.Rn:
        raise \
         ValueError('Ambos argumentos deben ser subconjuntos de en un mismo espacio')
```

El método suma `A + B` arroja la suma de dos `SubEspacios` de $\mathbb{R}^m$ (el es subespacio generado por la concatenación de los respectivos sistemas generadores).

```
def __add__(self, other):
    """Devuelve la suma de subespacios de Rn"""
    self.verificacion(other)
    return SubEspacio(self.sgen.concatena(other.sgen))
```

---

y definimos otro método que nos devuelva la intersección: `A & B`.

Si `other` es un `EAfin` llamamos al método de la intersección entre un `EAfin` y un `SubEspacio`.

```
def __and__(self, other):
    """Devuelve la intersección de subespacios"""
    self.verificacion(other)

    if isinstance(other, SubEspacio):
        A  = self.base
        B  = other.base
        AB = A.concatena(B)
        X  = slice(1,self.dim)|Matrix(AB.espacio_nulo().sgen)
        return SubEspacio(A*X)

    elif  isinstance(other, EAfin):
        return other & self
```

Con `~A` obtenemos el complemento ortogonal (el `SubEspacio` generado por las filas de `self.cart`)

```
def __invert__(self):
    """Devuelve el complemento ortogonal"""
    if isinstance(self.sgen|1, Vector):
        return SubEspacio( Sistema( ~(self.cart) ) )

    elif isinstance(self.sgen|1, Matrix):
        return SubEspacio(Sistema([v.reshape(self.Rn) for v in ~(self.cart)]))
```

Finalmente definimos un método para saber si un `Vector x` pertenece a un `SubEspacio A` (basta verificar si `x` es solución del sistema de ecuaciones cartesianas). Obtenemos la respuesta con

```
x in A
```

```
def __contains__(self, other):
    """Indica si un Vector pertenece a un SubEspacio"""
    if not isinstance(other, type(self.sgen|1)):
        raise ValueError('Objeto de tipo distinto a los del SubEspacio')

    if not estructura(other)==estructura(self.sgen|1):
        raise ValueError('Objeto tiene estructura o longitud distinta a la de los vectores del SubEspacio')

    return self.cart*other.vec() == V0(self.cart.m)
```

12.4. Métodos de representación de la clase SubEspacio

Métodos de representación de la clase SubEspacio

```
def __repr__(self):
    """Muestra el subespacio en su representación Python: la expresión
    que lo crea a partir de un sistema generador"""
    return 'SubEspacio(Sistema([' + ', '.join(repr(v) for v in self.sgen) + ']))'

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

def EcParametricas(self, d=0):
    """Representación paramétrica del SubEspacio"""
    if d: display(Math(self.EcParametricas()))
    return EAfin(self.sgen, self.vector_nulo()).EcParametricas()

def EcCartesianas(self, d=0):
    """Representación cartesiana del SubEspacio"""
    if d: display(Math(self.EcCartesianas()))
    return EAfin(self.sgen, self.vector_nulo()).EcCartesianas()

def latex(self):
    """ Construye el comando LaTeX para un SubEspacio de Rn"""
    return EAfin(self.sgen, self.vector_nulo()).latex()
```

12.5. La clase SubEspacio completa

Definición de la clase SubEspacio

```
class SubEspacio:
    <<Inicialización de la clase ~SubEspacio~>>
    <<Método auxiliar para establecer el vector nulo del subespacio>>
    <<Métodos de la clase ~SubEspacio~>>
    <<Métodos de representación de la clase ~SubEspacio~>>
```

Capítulo 13

La clase `EAfin` (de $\mathbb{R}^m$)

El conjunto de soluciones de un sistema de ecuaciones homogéneo $\mathbf{A}\boldsymbol{x} = \mathbf{0}$ forma un subespacio (que llamamos espacio nulo $\mathcal{N}(\mathbf{A})$), pero el conjunto de soluciones de $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ cuando $\boldsymbol{b} \neq \mathbf{0}$ es un *espacio afín*.

Un espacio afín es el conjunto de vectores que se obtienen sumando un vector fijo a cada vector de un subespacio. `EAfin(S, v)` es el espacio afín que resulta de sumar el vector `v` a los vectores del subespacio generado por el `Sistema S` (el primer argumento también puede ser un `SubEspacio`). El atributo `eafin` del resultado de `SEL` es el espacio afín de las soluciones del sistema de ecuaciones.

Atributo	Contenido
`S`	el `SubEspacio` cuyos vectores se suman al vector fijo
`v`	un vector del espacio afín
`Rn`	número de componentes de sus vectores

Hay muchos vectores que sirven como vector fijo. Para que dos descripciones del mismo espacio afín den el mismo objeto, el atributo `v` no es el vector indicado al crearlo, sino uno elegido por eliminación. Si se quiere conservar el vector indicado hay que añadir un tercer argumento no nulo: `EAfin(S, v, 1)`.

Expresión	Resultado
`v in E`	indica si el vector `v` pertenece al espacio afín `E`
`E = F=, E ! F=`	indica si son (o no son) el mismo espacio afín
`E.contenido_en(F)`	indica si `E` es un subconjunto de `F`
`E & F`	intersección (con otro espacio afín o con un subespacio)
`~E`	complemento ortogonal del subespacio `E.S`
`E.EcParametricas()`	código LaTeX de las ecuaciones paramétricas
`E.EcCartesianas()`	código LaTeX de las ecuaciones cartesianas

13.1. Ejemplos de espacios afines

Sumamos un vector al subespacio de los ejemplos anteriores:

```
S = Sistema( [ Vector([0,1,0]), Vector([2,1,3]) ] )
E = EAfin( S, Vector([1,1,1]) )
```

```
print(dmath( E ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^2,\ \boldsymbol{v} = \begin{pmatrix} 0 \\ 0 \\ -\frac{1}{2} \end{pmatrix} + \begin{bmatrix} 0 & 2 \\ 1 & 1 \\ 0 & 3 \end{bmatrix} \boldsymbol{p} \right\} = \left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} -3 & 0 & 2 \end{bmatrix} \boldsymbol{v} = \begin{pmatrix} -1, \end{pmatrix} \right\}$$

El vector que aparece en las ecuaciones paramétricas no es el que hemos indicado, pero ambos pertenecen al espacio afín:

```
print(dprint( E.v ))
```

```
(0, 0, -1/2,)
```

```
print(dprint( Vector([1,1,1]) in E ))
```

```
True
```

Por eso, al crear el mismo espacio afín con otro de sus vectores se obtiene un objeto igual:

```
print(dprint( E == EAfin( S, Vector([1,5,1]) ) ))
```

```
True
```

En modo texto se muestra la expresión que crea el espacio afín (con el tercer argumento que conserva el vector):

```
print(dprint( E ))
```

```
EAfin(Sistema([Vector([0, 1, 0]), Vector([2, 1, 3])]), Vector([0, 0, -1/2]), 1)
```

Un espacio afín que no es un subespacio no contiene el vector nulo:

```
print(dprint( V0(3) in E ))
```

```
False
```

La intersección con una recta que no es paralela a `E` es un punto:

```
F = EAfin( Sistema( [ Vector([1,0,0]) ] ), Vector([0,0,1]) )
```

```
print(dmath( E & F ))
```

$$\left\{ \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\}$$

y la intersección con un espacio afín paralelo y distinto es el conjunto vacío:

```
G = EAfin( S, Vector([2,0,1]) )
```

```
print(dmath( E & G ))
```

$$\{\}$$

El conjunto de soluciones de un sistema de ecuaciones es un espacio afín, que se guarda en el atributo `eafin` del resultado de `SEL`:

```
M = Matrix( [ [-3,0,2], [6,0,-4] ] )
```

```
print(dmath( SEL( M, Vector([3,-6]) ).eafin ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^2,\ \boldsymbol{v} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} + \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 3 \end{bmatrix} \boldsymbol{p} \right\} = \left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} -3 & 0 & 2 \end{bmatrix} \boldsymbol{v} = \begin{pmatrix} 3, \end{pmatrix} \right\}$$

13.2. Implementación de `EAfin`

Vamos a crear la clase `EAfin`. La definiremos como un par $(\mathcal{S}, \boldsymbol{v})$ cuyo primer elemento, $\mathcal{S}$, sea un `SubEspacio` (el conjunto de soluciones a $\mathbf{A}\boldsymbol{x} = \mathbf{0}$) y cuyo segundo elemento, $\boldsymbol{v}$, sea un vector del espacio afín (una solución particular de $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$). En el atributo `S` guardaremos el `SubEspacio` y en el atributo `v` un `Vector`. Así, pues, para instanciar un `EAfin` usaremos dos argumentos: el primero será un `Sistema` formado por `Vectores` (típicamente una `Matrix`) con la que formar el `SubEspacio`, y el segundo será un `Vector`.

Cuando $\boldsymbol{v} \in \mathcal{S}$, el espacio afín es un subespacio (que por tanto contiene al vector nulo). Así que si $\boldsymbol{v} \in \mathcal{S}$ en el atributo `v` guardaremos el vector nulo. Así, si consideramos el sistema "ampliado" que contiene los vectores del sistema generador de `S` primero, y `v` como último vector `v`, y aplicamos el método de eliminación de izquierda a derecha; el último vector tras la eliminación pertenece al espacio afín, y será cero si $\boldsymbol{v} \in \mathcal{S}$. Si `vi` es cero (su valor por defecto), en `self.v` se guardará el vector resultante tras la eliminación, en caso contrario se guardará el vector indicado (`vi`) como argumento.

Inicialización de la clase `EAfin`

```
def __init__(self, data, v, vi=0):
    """Inicializa un Espacio Afín de Rn"""
    self.S  = data if isinstance(data, SubEspacio) else SubEspacio(data)
    self.Rn = self.S.Rn

    if isinstance(self.Rn, int):
        if not isinstance(v, Vector) or v.n != self.Rn:
            raise ValueError('v y SubEspacio deben estar en el mismo espacio vectorial')

    elif isinstance(self.Rn, tuple):
        if not isinstance(v, Matrix) or (v.m,v.n) != self.Rn:
            raise ValueError('v y SubEspacio deben estar en el mismo espacio vectorial')

    self.v  = v if vi else (self.S.sgen.concatena(Sistema([v]))).elim(1)|0
```

13.3. Métodos de la clase `EAfin`

Un vector $\boldsymbol{x}$ pertenece al espacio afín $\mathcal{S}$ si verifica las ecuaciones cartesianas, cuya matriz de coeficientes es `self.S.cart`, y cuyo vector del lado derecho es `(self.S.cart)*self.v`. Así pues

---

```
def __contains__(self, other):
    """Indica si un Vector pertenece a un EAfin"""
    if isinstance(self.S.sgen|1, Vector):
        if not isinstance(other, Vector) or other.n != self.Rn:
            raise ValueError\
                ('El Vector no tiene el número adecuado de componentes')
        return self.S.cart*other == (self.S.cart)*self.v

    elif isinstance(self.S.sgen|1, Matrix):
        if not isinstance(other, Matrix) or (other.m,other.n) != self.Rn:
            raise ValueError\
                ('La matrix no tiene el orden adecuado')
        return self.S.cart*other.vec() == self.S.cart*self.v.vec()
```

```
def contenido_en(self, other):
    """Indica si este EAfin está contenido en other"""
    self.verificacion(other)

    if isinstance(other, SubEspacio):
        return self.v in other and self.S.contenido_en(other)

    elif isinstance(other, EAfin):
         return self.v in other and self.S.contenido_en(other.S)
```

```
def __eq__(self, other):
    """Indica si un EAfin de Rn es igual a other"""
    self.verificacion(other)
    return self.contenido_en(other) and other.contenido_en(self)

def __ne__(self, other):
    """Indica si un subespacio de Rn es distinto de other"""
    return not (self == other)
```

```
def verificacion(self,other):
    if not isinstance(other, (SubEspacio, EAfin)) or  not self.Rn == other.Rn:
        raise ValueError('Ambos argumentos deben ser subconjuntos de en un mismo espacio')
```

La intersección es el conjunto de soluciones a ambos sistemas de ecuaciones cartesianas. El modo más sencillo es unificar ambos sistemas en uno solo: apilando las matrices de coeficientes por un lado y concatenando los vectores del lado derecho por el otro.

```
def __and__(self, other):
    """Devuelve la intersección de este EAfin con other"""
    self.verificacion(other)
    if isinstance(other, EAfin):
        M = self.S.cart.apila( other.S.cart )
        if isinstance(self.S.sgen|1, Vector):
```

```
            w = (self.S.cart*self.v).concatena( other.S.cart*other.v )
        elif isinstance(self.S.sgen|1, Matrix):
            w = (self.S.cart*self.v.vec()).concatena( other.S.cart*other.v.vec() )
    elif isinstance(other, SubEspacio):
        M = self.S.cart.apila( other.cart )
        if isinstance(self.S.sgen|1, Vector):
            w = (self.S.cart*self.v).concatena( V0(other.cart.m) )
        elif isinstance(self.S.sgen|1, Matrix):
            w = (self.S.cart*self.v.vec()).concatena( V0(other.cart.m) )

    return SEL(M,w).eafin
```

Con `~A` obtendremos el mayor `SubEspacio` perpendicular a `A`.

```
def __invert__(self):
    """Devuelve el mayor SubEspacio perpendicular a self"""
    return SubEspacio( Sistema([v.reshape(self.Rn) for v in ~(self.S.cart)]))
```

13.4. Métodos de representación de la clase `EAfin`

Métodos de representación de la clase EAfin

```
def __repr__(self):
    """Muestra el espacio afín en su representación Python: la expresión
    que lo crea a partir de un sistema generador del subespacio y un vector"""
    return 'EAfin(Sistema([' + ', '.join(repr(v) for v in self.S.sgen) + ']), ' + repr(self.v) + ', 1)'

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

def EcParametricas(self, d=0):
    """Representación paramétrica de EAfin"""
    punto = latex(self.v) + '+' if (self.v != 0*self.v) else ''
    if d: display(Math(self.EcParametricas()))
    if isinstance(self.S.Rn,int):
        return r'\left\{ \boldsymbol{v}\in\mathbb{R}^' \
            + latex(self.S.Rn)  \
            + r'\ \left|\ \exists\boldsymbol{p}\in\mathbb{R}^' \
            + latex(max(self.S.dim,1)) \
            + r',\; \boldsymbol{v}= ' \
            + punto \
            + latex(Matrix(self.S.sgen)) \
            + r'\boldsymbol{p}\right. \right\}'
    else:
        return r'\left\{ \pmb{\mathsf{M}}\in\mathbb{R}^{' \
            + latex(self.S.Rn[0]) + r'\times' + latex(self.S.Rn[1]) + '}' \
            + r'\ \left|\ \exists\boldsymbol{p}\in\mathbb{R}^' \
            + latex(max(self.S.dim,1)) \
            + r',\; \mathsf{\pmb{M}}= ' \
            + punto \
            + latex(self.S.sgen) \
            + r'\boldsymbol{p}\right. \right\}'

def EcCartesianas(self, d=0):
    """Representación cartesiana de EAfin"""
    if d: display(Math(self.EcCartesianas()))
    if isinstance(self.S.Rn,int):
        return r'\left\{ \boldsymbol{v}\in\mathbb{R}^' \
            + latex(self.S.Rn) \
            + r'\ \left|\ ' \
```

```
            + latex(self.S.cart) \
            + r'\boldsymbol{v}=' \
            + latex(self.S.cart*self.v) \
            + r'\right.\right\}'
    else:
        return r'\left\{ \mathsf{\pmb{M}}\in\mathbb{R}^{' \
            + latex(self.S.Rn[0]) + r'\times' + latex(self.S.Rn[1]) + '}' \
            + r'\ \left|\ ' \
            + latex(self.S.cart) \
            + r'vec(\mathsf{\pmb{M}})=' \
            + latex(self.S.cart*self.v.vec()) \
            + r'\right.\right\}'

def latex(self):
    """ Construye el comando LaTeX para un EAfin de Rn"""
    if self.S.sgen.es_nulo():
        return r'\left\{\ ' + latex(self.v) + r'\ \right\}'
    else:
        return self.EcParametricas() + '\\; = \\;' + self.EcCartesianas()
```

13.5. La clase `EAfin` completa

Definición de la clase `EAfin`

```
class EAfin:
    <<Inicialización de la clase ~EAfin~>>
    <<Métodos de la clase ~EAfin~>>
    <<Métodos de representación de la clase ~EAfin~>>
```

Parte IV

Las clases `Homogenea` y `SEL`

Capítulo 14

Resolución de sistemas de ecuaciones homogéneos. La clase `Homogenea`

La clase `Homogenea` calcula el conjunto de soluciones de un sistema homogéneo $\mathbf{A}\boldsymbol{x} = \mathbf{0}$; es decir, el espacio nulo $\mathcal{N}(\mathbf{A})$. Implementa el procedimiento de la Sección 7.4 del libro (*Resolución de un Sistema Homogéneo por eliminación*).

La idea es la misma que la del cálculo de la inversa: al pre-escalonar las columnas de la matriz que resulta de apilar $\mathbf{A}$ sobre la identidad se obtiene

$$\begin{bmatrix}\mathbf{A}\\ \mathbf{I}\end{bmatrix} \xrightarrow{\boldsymbol{\tau}_1 \cdots \boldsymbol{\tau}_k} \begin{bmatrix}\mathbf{K}\\ \mathbf{E}\end{bmatrix}, \qquad \text{donde} \quad \mathbf{K} = \mathbf{A}\mathbf{E}.$$

En cada columna, la parte inferior contiene los coeficientes de la combinación lineal de las columnas de $\mathbf{A}$ cuyo resultado es la parte superior. Por tanto, debajo de cada columna nula de $\mathbf{K}$ hay una solución del sistema. El libro las llama *soluciones especiales* y demuestra que toda solución es combinación lineal de ellas; y que si $\mathbf{K}$ no tiene columnas nulas la única solución es el vector nulo.

El código hace exactamente eso: calcula una forma pre-escalonada `K` de la matriz de coeficientes, aplica las mismas transformaciones a la identidad para obtener `E`, y selecciona las columnas de `E` que están bajo las columnas nulas de `K`. Como en el caso de la inversa, la matriz apilada solo se usa para escribir el atributo `tex`.

Atributos del resultado:

Atributo	Contenido
`base`	`Sistema` con las soluciones especiales (vacío si la única solución es $\mathbf{0}$)
`sgen`	sistema generador de $\mathcal{N}(\mathbf{A})$: es `base`, o el sistema que solo contiene $\mathbf{0}$
`determinado`	indica si la única solución es el vector nulo
`enulo`	el espacio nulo como objeto de la clase `SubEspacio`
`pasos`, `TrF` y `TrC`	transformaciones elementales aplicadas
`tex`	código LaTeX con los pasos dados

Al igual que `elim`, la clase admite como argumento cualquier `Sistema` cuyos elementos sean del mismo tipo y tamaño (no solo una `Matrix`), o un `Sistema` de polinomios (clase `Polinomio`).

14.1. Implementación de la clase Homogenea

Resolución de un sistema de ecuaciones homogéneo

```
class Homogenea:
    def __init__(self, sistema, rep=0, sust=[], repsust=0):
        """Resuelve un Sistema de Ecuaciones Lineales Homogéneo

        y muestra los pasos para encontrarlo"""
        try:
            if any(isinstance(e, Polinomio) for e in sistema):
                raise TypeError('los polinomios no se sustituyen por sus coeficientes')
            A = Matrix(sistema).subs(sust)
        except Exception:
            A = BlockM([sistema]).subs(sust)

        MA = A.apila(I(A.n),1)
        MA.corteElementos.update({sistema.n+A.m})

        K    = A.K(0, sust)
        E    = I(A.n) & T(K.pasos[1])

        self.base        = Sistema([ v for j, v in enumerate(E,1) if (K|j).es_nulo(sust) ])
        self.sgen        = self.base if self.base else Sistema([ V0(sistema.n) ])
        self.determinado = (len(self.base) == 0)
        self.pasos       = K.pasos;
        self.TrF         = K.TrF
        self.TrC         = K.TrC

        self.enulo       = SubEspacio(self.sgen)

        if repsust:
            self.tex         = rprElim( A.apila( I(A.n) ,1 ) , self.pasos, [], sust)
        else:
            self.tex         = rprElim( A.apila( I(A.n) ,1 ) , self.pasos)

        if rep:
            display(Math(self.tex))

    <<Métodos de representación de la clase ~Homogenea~>>
```

La base la constituyen los vectores `v` de `E` que corresponden a los vectores nulos de `K`.

14.2. Métodos de representación de la clase Homogenea

Métodos de representación de la clase Homogenea

```
def __repr__(self):
    """Muestra el Espacio Nulo de una matriz en su representación Python"""
    return texto_de_salida('combinaciones') + ' (' + repr(self.sgen) + ')'

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

def latex(self):
    """ Construye el comando LaTeX para la solución de un Sistema Homogéneo"""
    if self.determinado:
        return '\\left\\{\\ ' + latex(self.sgen|1) + '\\ \\right\\}'
    else:
        return '\\mathcal{L}\\left(\\ ' + latex(self.sgen) + '\\ \\right)'
```

14.3. Ejemplos de sistemas homogéneos

Usaremos la matriz del ejemplo de la Sección 7.4 del libro.

```
A = Matrix( [ [1,2,0,1], [0,1,1,0], [1,2,0,1] ] )
H = Homogenea(A)
```

Tras la eliminación quedan dos columnas nulas en la parte superior (la tercera y la cuarta):

```
print(displaymath( H.tex ))
```

$$\left[\begin{array}{cccc} 1 & 2 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 2 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{2}] \\ [(-1)\mathbf{1}+\mathbf{4}]}]{\boldsymbol{\tau}} \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 1 & -2 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[[(-1)\mathbf{2}+\mathbf{3}]]{\boldsymbol{\tau}} \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 1 & -2 & 2 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$$

Así que las dos soluciones especiales son las columnas tercera y cuarta de la parte inferior

```
print(dprint( H.sgen ))
```

```
[(2, -1, 1, 0,); (-1, 0, 0, 1,);]
```

y el conjunto de soluciones es el conjunto de combinaciones lineales de dichos vectores:

```
print(dmath( H ))
```

$$\mathcal{L}\left(\left[\begin{pmatrix} 2 \\ -1 \\ 1 \\ 0 \end{pmatrix};\quad \begin{pmatrix} -1 \\ 0 \\ 0 \\ 1 \end{pmatrix};\right]\right)$$

Efectivamente, al multiplicar la matriz por cada una de las soluciones especiales se obtiene el vector nulo:

```
print(dprint( (A*Matrix(H.sgen)).es_nulo() ))
```

```
True
```

El atributo `enulo` es el espacio nulo como objeto de la clase `SubEspacio`, cuya representación muestra tanto unas ecuaciones paramétricas como unas ecuaciones cartesianas:

```
print(dmath( H.enulo ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^4 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^2,\ \boldsymbol{v} = \begin{bmatrix} 2 & -1 \\ -1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \boldsymbol{p} \right\} = \left\{ \boldsymbol{v} \in \mathbb{R}^4 \;\middle|\; \begin{bmatrix} 0 & 2 & 2 & 0 \\ 1 & 2 & 0 & 1 \end{bmatrix} \boldsymbol{v} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

Cuando la forma pre-escalonada no tiene columnas nulas el sistema es determinado y la única solución es el vector nulo:

```
D = Homogenea( Matrix( [ [1,2], [3,4] ] ) )
print(dprint( (D.determinado, D.sgen) ))
```

```
(True, Sistema([Vector([0, 0]);]))
```

```
print(dmath( D ))
```

$$\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

Capítulo 15

Resolución de sistemas de ecuaciones lineales. La clase `SEL`

La clase `SEL` resuelve el sistema $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ con el procedimiento de la Sección 8.1 del libro (*Eliminación sobre la matriz ampliada*), que reduce el problema al caso homogéneo:

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b} \qquad \Longleftrightarrow \qquad \left[\begin{array}{c|c} \mathbf{A} & -\boldsymbol{b} \end{array}\right] \begin{pmatrix} \boldsymbol{x} \\ \hline 1 \end{pmatrix} = \mathbf{0}.$$

Es decir, las soluciones son los vectores del espacio nulo de la matriz ampliada $[\mathbf{A}|-\boldsymbol{b}]$ cuya última componente es un 1 (nótese el cambio de signo del vector del lado derecho). Por eso se pre-escalona la matriz ampliada apilada sobre la identidad, y se mira la última columna del resultado:

- Si su parte superior no es nula el sistema no tiene solución, pues la matriz ampliada tiene un pivote más que la matriz de coeficientes (Teorema de Rouché-Frobenius).
- Si su parte superior es nula, bajo ella hay un vector del espacio nulo de la matriz ampliada. Se divide la columna por su última componente para que esta sea un 1 (por eso el último paso es una transformación que normaliza la última columna), y el resto de componentes de la parte inferior es una solución del sistema.

Las demás columnas nulas de la parte superior corresponden, como en la clase `Homogenea`, a las soluciones especiales del sistema homogéneo $\mathbf{A}\boldsymbol{x} = \mathbf{0}$. Si no hay ninguna, la solución es única; y en caso contrario el conjunto de soluciones es el formado por las sumas de la solución encontrada con las combinaciones lineales de las soluciones especiales.

En el código, `operaciones` es la lista de transformaciones que pre-escalonan la matriz ampliada `A` (su última columna es $-\boldsymbol{b}$). El `testigo` es el último componente de la última columna de la identidad tras aplicarle dichas transformaciones; es decir, el número por el que hay que dividir para que la última componente sea un 1. La solución particular `solP` se lee aplicando todos los pasos sobre la identidad ampliada con una columna de ceros.

15.1. Implementación

15.1.1. Texto de ayuda

Texto de ayuda de la clase SEL

```
"""Resuelve un Sistema de Ecuaciones Lineales

mediante eliminación con el sistema ampliado y muestra los pasos
dados

Parámetros:
    sistema (Sistema): Matrix de coeficientes; o, en general, Sistema
        de objetos del mismo tipo y tamaño (por ejemplo de matrices),
        o Sistema de polinomios (clase Polinomio)
    b (Vector, Matrix,...): lado derecho de la ecuación. Del mismo tipo
        y tamaño que los elementos del sistema
    rep (bool): si es cierto se muestran en Jupyter los pasos dados
    sust (list): reglas de sustitución de variables simbólicas

Atributos:
    solP  (Vector): una solución particular (set() si no hay solución)
    sgen (Sistema): sistema generador de las soluciones del sistema
                    homogéneo asociado
    eafin  (EAfin): conjunto de soluciones (set() si no hay solución)
    sistema, b    : el sistema y el lado derecho de la ecuación
    determinado (bool): indica si hay una única solución
    tex      (str): código LaTeX con los pasos dados
    pasos   (list): transformaciones elementales aplicadas

Ejemplos:
>>> # Combinaciones lineales de las columnas de A (vectores) iguales a b
>>> A = Matrix([[1,2],[2,4]]);  b = Vector([1,2])
>>> SEL(A, b)
Vector([1, 0]) + Combinaciones lineales de (Sistema([Vector([-2, 1]);]))

>>> # Combinaciones lineales de las matrices de S iguales a la matriz B
>>> S = Sistema([ Matrix([[1,0],[0,0]]), Matrix([[0,1],[1,0]]),
...               Matrix([[1,1],[1,0]]), Matrix([[0,0],[0,1]]) ])
>>> B = Matrix([[2,3],[3,5]])
>>> SEL(S, B)
Vector([2, 3, 0, 5]) + Combinaciones lineales de (Sistema([Vector([-1, -1, 1, 0]);]))

"""
```

Resolución de un Sistema de Ecuaciones Lineales

```
class SEL:
    def __init__(self, sistema, b, rep=0, sust=[], repsust=0):
        <<Texto de ayuda de la clase ~SEL~>>
        try:
            if any(isinstance(e, Polinomio) for e in sistema.amplia(-b)):
                raise TypeError('los polinomios no se sustituyen por sus coeficientes')
            A = Matrix(sistema.amplia(-b)).subs(sust).csis({sistema.n})
        except Exception:
            A = BlockM([sistema.amplia(-b)]).subs(sust).csis({sistema.n})

        MA = A.apila(I(A.n),1)
        MA.corteElementos.update({sistema.n+A.m})
        operaciones = A.elim(0,False,sust).pasos[1]

        <<Aplicamos los ~pasos~ de eliminación sobre la matriz ampliada y obtenemos la solución>>

    <<Equivalencia entre sistemas de ecuaciones>>
    <<Métodos de representación de la clase ~SEL~>>
```

Aplicamos los pasos sobre toda la matriz ampliada (más bien "super ampliada", pues tiene una

matriz identidad por debajo). Si el último elemento de la última columna es nulo debajo vemos la solución.

Aplicamos los pasos de eliminación sobre la matriz ampliada y obtenemos la solución

```
E         = I(sistema.n) & T(operaciones)

testigo   = 0| (I(A.n) & T(operaciones)) |0
Normaliza = T([]) if testigo==1 else T([( fracc(1,testigo), A.n )])
pasos     = [[], operaciones+[Normaliza] ] if Normaliza else [[], operaciones]

K         = A & T(operaciones)

self.sistema     = sistema.subs(sust)
self.b           = b.subs(sust)
self.base        = Sistema([ v for j, v in enumerate(E,1) if (K|j).es_nulo(sust) ])
self.sgen        = self.base if self.base else Sistema([ V0(sistema.n) ])
self.determinado = (len(self.base) == 0)
self.pasos       = pasos
self.TrF         = T(self.pasos[0])
self.TrC         = T(self.pasos[1])

if (K|0).no_es_nulo():
    self.solP  = set()
    self.eafin = set()
else:
    self.solP  = (I(sistema.n).amplia(V0(sistema.n)) & T(pasos[1]))|0
    self.eafin = EAfin(self.sgen, self.solP, 1)

if repsust:
    self.tex     = rprElim( MA, self.pasos, [], sust )
else:
    self.tex     = rprElim( MA, self.pasos)

if rep:
    display(Math(self.tex))
```

Equivalencia entre sistemas de ecuaciones

```
def __eq__(self, other):
    """Indica si un subespacio de Rn es igual a otro"""
    if not isinstance(other, SEL):
        raise ValueError('El argumento debe ser SEL (Sist de Ecuaciones Lineales).')

    return self.eafin == other.eafin
```

15.2. Métodos de representación de la clase SEL

Métodos de representación de la clase SEL

```
def __repr__(self):
    """Muestra el conjunto de soluciones en su representación Python"""
    if not self.solP:
        return repr(set())       # sin solución: el conjunto vacío
    return repr(self.solP) + ' + ' + texto_de_salida('combinaciones') + ' (' + repr(self.sgen) + ')'

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

def EcCombinacion(self):
    """Conjunto de soluciones descrito mediante la ecuación que se resuelve"""
    return r'\left\{ \boldsymbol{v}\in\mathbb{R}^' + latex(self.sistema.n) \
        + r'\ \left|\ ' + latex(self.sistema) + r'\boldsymbol{v}=' + latex(self.b) \
        + r'\right.\right\}'
```

```
def latex(self):
    """ Construye el comando LaTeX para la solución de un Sistema de Ecuaciones"""
    if not self.solP:
        return '\\emptyset'

    if self.determinado:
        solucion = '\\left\\{\\ ' + latex(self.solP) + '\\ \\right\\}'
    else:
        solucion = self.eafin.EcParametricas()

    # si los elementos del sistema no son vectores de Rn (p.e. son matrices
    # o polinomios) se recuerda la ecuación de la que los vectores v son solución
    if any(isinstance(e, Sistema) and (isinstance(e, Polinomio) or not isinstance(e, Vector)) for e in
    ↪  self.sistema):
        return self.EcCombinacion() + '\\; = \\;' + solucion
    else:
        return solucion
```

15.3. Ejemplos de sistemas de ecuaciones

Usaremos el sistema de un ejemplo de la Sección 8.1 del libro.

```
A = Matrix( [ [7,-2,0], [6,4,2] ] )
b = Vector( [4,12] )
S = SEL(A, b)
```

La matriz ampliada tiene a la derecha el opuesto de `b`, y bajo ella se ha apilado la identidad. El último paso divide la última columna para que su última componente sea un 1. El atributo `tex` contiene todo el proceso en una sola línea, que aquí no cabe en el ancho de la página; la función `desarrollo` lo reparte en varias líneas (en este caso, con dos matrices en cada una):

```
print(displaymath( desarrollo(S, por_linea=2) ))
```

$$\left[\begin{array}{ccc|c} 7 & -2 & 0 & -4 \\ 6 & 4 & 2 & -12 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(2)\mathbf{1}+\mathbf{2}] \\ [(7)\mathbf{4}] \\ [(4)\mathbf{1}+\mathbf{4}]}]{\boldsymbol{\tau}[(7)\mathbf{2}]} \left[\begin{array}{ccc|c} 7 & 0 & 0 & 0 \\ 6 & 40 & 2 & -60 \\ \hline 1 & 2 & 0 & 4 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 7 \end{array}\right]$$

$$\xrightarrow[\substack{[(-1)\mathbf{2}+\mathbf{3}] \\ [(2)\mathbf{4}] \\ [(3)\mathbf{2}+\mathbf{4}]}]{\boldsymbol{\tau}[(20)\mathbf{3}]} \left[\begin{array}{ccc|c} 7 & 0 & 0 & 0 \\ 6 & 40 & 0 & 0 \\ \hline 1 & 2 & -2 & 14 \\ 0 & 7 & -7 & 21 \\ 0 & 0 & 20 & 0 \\ \hline 0 & 0 & 0 & 14 \end{array}\right] \xrightarrow{\boldsymbol{\tau}\left[\left(\frac{1}{14}\right)\mathbf{4}\right]} \left[\begin{array}{ccc|c} 7 & 0 & 0 & 0 \\ 6 & 40 & 0 & 0 \\ \hline 1 & 2 & -2 & 1 \\ 0 & 7 & -7 & \frac{3}{2} \\ 0 & 0 & 20 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Sobre el 1 de la última columna se lee una solución, y bajo la columna nula de la matriz de coeficientes (la tercera) una solución especial del sistema homogéneo:

```
print(dprint( (S.solP, S.sgen) ))
```

```
(Vector([1, 3/2, 0]), Sistema([Vector([-2, -7, 20]);]))
```

El conjunto de soluciones se representa mediante unas ecuaciones paramétricas:

```
print(dmath( S ))
```

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^1,\ \boldsymbol{v} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ 0 \end{pmatrix} + \begin{bmatrix} -2 \\ -7 \\ 20 \end{bmatrix} \boldsymbol{p} \right\}$$

El atributo `eafin` es dicho conjunto como objeto de la clase `EAfin`. Podemos comprobar que `solP` es solución, y que también lo es al sumarle un múltiplo de la solución especial:

```
print(dprint( (A*S.solP == b,  A*(S.solP + 3*(S.sgen|1)) == b) ))
```

```
(True, True)
```

Cuando la matriz de coeficientes es de rango completo por columnas la solución es única:

```
U = SEL( Matrix( [ [1,2], [3,4] ] ), Vector( [5,6] ) )
print(dprint( (U.determinado, U.solP) ))
```

```
(True, Vector([-4, 9/2]))
```

```
print(dmath( U ))
```

$$\left\{ \begin{pmatrix} -4 \\ \frac{9}{2} \end{pmatrix} \right\}$$

Y cuando tras la eliminación la parte superior de la última columna no es nula, el sistema no tiene solución. Los atributos `solP` y `eafin` son entonces el conjunto vacío:

```
N = SEL( Matrix( [ [1,2], [2,4] ] ), Vector( [1,3] ) )
print(displaymath( N.tex ))
```

$$\left[\begin{array}{cc|c} 1 & 2 & -1 \\ 2 & 4 & -3 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{2}] \\ [(1)\mathbf{1}+\mathbf{3}]}]{\boldsymbol{\tau}} \left[\begin{array}{cc|c} 1 & 0 & 0 \\ 2 & 0 & -1 \\ \hline 1 & -2 & 1 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right]$$

```
print(dprint( N.solP ))
```

```
set()
```

```
print(dmath( N ))
```

$\emptyset$

15.4. Ejemplo: combinaciones lineales de matrices

Los elementos del sistema no tienen por qué ser vectores de $\mathbb{R}^n$. La clase `SEL` resuelve la ecuación para cualquier `Sistema` cuyos elementos sean del mismo tipo y tamaño. Por ejemplo, podemos buscar todas las combinaciones lineales de cuatro matrices de orden 2 que son iguales a una matriz dada.

```
S = Sistema([ Matrix([[1,0],[0,0]]), Matrix([[0,1],[1,0]]), Matrix([[1,1],[1,0]]), Matrix([[0,0],[0,1]]) ])
B = Matrix([[2,3],[3,5]])
solucion = SEL(S, B, 1)
```

Los pasos de eliminación operan con las matrices del sistema ampliado como si fueran las columnas de una matriz de coeficientes (se muestran dos matrices en cada línea, con `desarrollo(solucion, por_linea=2)`):

$$\left[\begin{array}{cccc|c} \begin{bmatrix}1&0\\0&0\end{bmatrix} & \begin{bmatrix}0&1\\1&0\end{bmatrix} & \begin{bmatrix}1&1\\1&0\end{bmatrix} & \begin{bmatrix}0&0\\0&1\end{bmatrix} & \begin{bmatrix}-2&-3\\-3&-5\end{bmatrix} \\ \hline 1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\ \hline 0&0&0&0&1 \end{array}\right] \xrightarrow[\substack{[(-1)\mathbf{1}+\mathbf{3}]\\[(2)\mathbf{1}+\mathbf{5}]}]{\boldsymbol{\tau}} \left[\begin{array}{cccc|c} \begin{bmatrix}1&0\\0&0\end{bmatrix} & \begin{bmatrix}0&1\\1&0\end{bmatrix} & \begin{bmatrix}0&1\\1&0\end{bmatrix} & \begin{bmatrix}0&0\\0&1\end{bmatrix} & \begin{bmatrix}0&-3\\-3&-5\end{bmatrix} \\ \hline 1&0&-1&0&2\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\ \hline 0&0&0&0&1 \end{array}\right]$$

$$\xrightarrow[\substack{[(-1)\mathbf{2}+\mathbf{3}]\\[(3)\mathbf{2}+\mathbf{5}]}]{\boldsymbol{\tau}} \left[\begin{array}{cccc|c} \begin{bmatrix}1&0\\0&0\end{bmatrix} & \begin{bmatrix}0&1\\1&0\end{bmatrix} & \begin{bmatrix}0&0\\0&0\end{bmatrix} & \begin{bmatrix}0&0\\0&1\end{bmatrix} & \begin{bmatrix}0&0\\0&-5\end{bmatrix} \\ \hline 1&0&-1&0&2\\0&1&-1&0&3\\0&0&1&0&0\\0&0&0&1&0\\ \hline 0&0&0&0&1 \end{array}\right] \xrightarrow[[(5)\mathbf{4}+\mathbf{5}]]{\boldsymbol{\tau}} \left[\begin{array}{cccc|c} \begin{bmatrix}1&0\\0&0\end{bmatrix} & \begin{bmatrix}0&1\\1&0\end{bmatrix} & \begin{bmatrix}0&0\\0&0\end{bmatrix} & \begin{bmatrix}0&0\\0&1\end{bmatrix} & \begin{bmatrix}0&0\\0&0\end{bmatrix} \\ \hline 1&0&-1&0&2\\0&1&-1&0&3\\0&0&1&0&0\\0&0&0&1&5\\ \hline 0&0&0&0&1 \end{array}\right]$$

El conjunto de soluciones lo forman los vectores de coeficientes de dichas combinaciones lineales (el atributo `solP` es una solución particular y `sgen` un sistema generador de las soluciones del sistema homogéneo). Como los elementos del sistema no son vectores de $\mathbb{R}^n$, la representación recuerda la ecuación de la que dichos vectores de coeficientes son solución:

```
print(dmath( solucion ))
```

$$\left\{ \boldsymbol{v}\in\mathbb{R}^4 \;\middle|\; \left[\begin{bmatrix}1&0\\0&0\end{bmatrix}; \begin{bmatrix}0&1\\1&0\end{bmatrix}; \begin{bmatrix}1&1\\1&0\end{bmatrix}; \begin{bmatrix}0&0\\0&1\end{bmatrix}; \right] \boldsymbol{v} = \begin{bmatrix}2&3\\3&5\end{bmatrix} \right\} = \left\{ \boldsymbol{v}\in\mathbb{R}^4 \;\middle|\; \exists \boldsymbol{p}\in\mathbb{R}^1,\ \boldsymbol{v} = \begin{pmatrix}2\\3\\0\\5\end{pmatrix} + \begin{bmatrix}-1\\-1\\1\\0\end{bmatrix} \boldsymbol{p} \right\}$$

Efectivamente, cada solución es la lista de coeficientes de una combinación lineal de las matrices de `S` que es igual a `B`:

```
print(dmath( S*solucion.solP ))
```

$$\begin{bmatrix}2&3\\3&5\end{bmatrix}$$

15.5. Ejemplo: combinaciones lineales de polinomios

Lo mismo se puede hacer con polinomios (clase `Polinomio`). Busquemos las combinaciones lineales de los polinomios 1, $1+x$ y $(1+x)^2$ que son iguales al polinomio $3-2x+5x^2$; es decir, las coordenadas de dicho polinomio respecto de la base formada por las potencias de $1+x$.

```
S = Sistema([ Polinomio(1), Polinomio(1+x), Polinomio((1+x)**2) ])
q = Polinomio(3 - 2*x + 5*x**2)
solucion = SEL(S, q)
```

En la fila superior se ve cómo cambian los polinomios con cada transformación elemental. El pivote de cada polinomio es el coeficiente de su término de menor grado (el primero que aparece escrito), y con él se eliminan los términos del mismo grado de los polinomios que están a su derecha (de nuevo con dos matrices en cada línea):

```
print(displaymath( desarrollo(solucion, por_linea=2) ))
```

$$\left[\begin{array}{cccc|c} 1 & 1+x & 1+2x+x^2 & -3+2x-5x^2 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(-1)\mathbf{1}+\mathbf{2}]\\ [(-1)\mathbf{1}+\mathbf{3}]\\ [(3)\mathbf{1}+\mathbf{4}]}]{\boldsymbol{\tau}} \left[\begin{array}{cccc|c} 1 & x & 2x+x^2 & 2x-5x^2 \\ \hline 1 & -1 & -1 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

$$\xrightarrow[\substack{[(-2)\mathbf{2}+\mathbf{3}]\\ [(-2)\mathbf{2}+\mathbf{4}]}]{\boldsymbol{\tau}} \left[\begin{array}{ccc|c} 1 & x & x^2 & -5x^2 \\ \hline 1 & -1 & 1 & 5 \\ 0 & 1 & -2 & -2 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[[(5)\mathbf{3}+\mathbf{4}]]{\boldsymbol{\tau}} \left[\begin{array}{ccc|c} 1 & x & x^2 & 0 \\ \hline 1 & -1 & 1 & 10 \\ 0 & 1 & -2 & -12 \\ 0 & 0 & 1 & 5 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Como los polinomios no son vectores de $\mathbb{R}^n$, la representación recuerda la ecuación que se ha resuelto:

```
print(dmath( solucion ))
```

$$\left\{\boldsymbol{v}\in\mathbb{R}^3 \;\middle|\; \left[\begin{array}{lll} 1; & 1+x; & 1+2x+x^2; \end{array}\right]\boldsymbol{v} = 3-2x+5x^2\right\} = \left\{\begin{pmatrix} 10 \\ -12 \\ 5 \end{pmatrix}\right\}$$

Efectivamente, la combinación lineal cuyos coeficientes son los componentes de la solución es el polinomio buscado:

```
print(dprint( S*solucion.solP ))
```

```
3 - 2*x + 5*x**2
```

La clase `Homogenea` encuentra las relaciones de dependencia lineal entre polinomios. Por ejemplo, los polinomios $1+x$, $1-x$ y x son linealmente dependientes:

```
H = Homogenea( Sistema([ Polinomio(1+x), Polinomio(1-x), Polinomio(x), Polinomio(x**2) ]) )
print(displaymath( H.tex ))
```

$$\left[\begin{array}{c|cccc} 1+x & 1-x & x & x^2 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[[(-1)\mathbf{1}+\mathbf{2}]]{\boldsymbol{\tau}} \left[\begin{array}{c|ccc} 1+x & -2x & x & x^2 \\ \hline 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(2)\mathbf{3}] \\ [(1)\mathbf{2}+\mathbf{3}]}]{\boldsymbol{\tau}} \left[\begin{array}{c|ccc} 1+x & -2x & 0 & x^2 \\ \hline 1 & -1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$$

```
print(dprint( H.sgen ))
```

```
[(-1, 1, 2, 0,);]
```

Parte V

La clase `Determinante`. Cálculo del determinante por eliminación

El determinante de una matriz triangular es el producto de los elementos de su diagonal principal. Por eso se puede calcular por eliminación: basta escalonar la matriz y multiplicar los elementos de la diagonal. Pero las transformaciones elementales que multiplican una columna por un número (y los intercambios de columnas) cambian el valor del determinante. Para no perder la cuenta se trabaja con la matriz extendida, que tiene una fila y una columna más con un 1 en la diagonal: cada vez que una transformación de las n primeras columnas multiplica el determinante por un número, se divide la última fila por ese mismo número. Así el determinante de la matriz extendida no cambia y, al terminar, es el producto de los elementos de la diagonal. Es el procedimiento de la sección «Cálculo del determinante por eliminación Gaussiana» del libro.

Argumento	Significado
`data`	matriz cuadrada
`rep`	si no es nulo, muestra el proceso en Jupyter (también se admite `disp`)
`sust`	lista de sustituciones de símbolos que se aplican antes de calcular

Atributo	Contenido
`valor`	valor del determinante
`tex`	código LaTeX del proceso sobre la matriz extendida
`pasos`	transformaciones aplicadas: `[de las filas, de las columnas]`

El método `determinante` de una `Matrix` devuelve directamente el valor: `A.determinante()` es lo mismo que `Determinante(A).valor`.

15.5.1. Método de inicialización

Definición de la clase Determinante

```
class Determinante:
    """Determinante de una Matrix mediante eliminación Gaussiana por columnas

    La representación muestra los pasos dados

    """
    def __init__(self, data, disp=0, sust=[], rep=None):
        # rep es otro nombre para disp (el que usan las demás clases)
        if rep is not None:
            disp = rep
        <<Cálculo del determinante>>

        A  = Matrix(data.subs(sust))

        if not A.es_cuadrada():  raise ValueError('Matrix no cuadrada')

        self.tex, self.valor, self.pasos = calculoDet( A.subs(sust) )

        if disp:
            display(Math(self.tex))

    <<Métodos de representación de la clase Determinante>>
```

Cálculo del determinante

```
def calculoDet(A, sust=[]):

    producto  = lambda x: 1 if not x else x[0] * producto(x[1:])

    def productos_realizados(operaciones):
        P = [ producto([-1 if isinstance(abv,set) else abv[0] \
                        for op in paso for abv in filter( lambda x: len(x)==2, op.abreviaturas)]) for paso in
                        ↪  operaciones]
```

```
        return P

    operacionesEnColumnas = (A.L(0,sust).pasos[1])
    operacionesEnFilas    = [T((fracc(1,d),A.n+1)) for d in productos_realizados(operacionesEnColumnas)]
    pasos                 = [operacionesEnFilas[::-1], operacionesEnColumnas]

    matrixExtendida       = T(operacionesEnFilas) & A.extDiag(I(1),1) & T(operacionesEnColumnas)

    determinante          = sympy.sympify( producto( matrixExtendida.diag() ) ).simplify()

    tex                   = rprElimFyC( A.extDiag(I(1),1), pasos)

    return [tex, determinante, pasos]
```

Métodos de representación de la clase Determinante

```
def __repr__(self):
   """Muestra un Sistema en su representación Python"""
   return texto_de_salida('determinante') + ' ' + repr (self.valor)

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

def latex(self):
   """Construye el comando LaTeX para representar un Sistema"""
   return latex(self.valor)
```

15.6. Ejemplos de determinantes

Usaremos la matriz del segundo ejemplo de la sección del libro citada más arriba.

```
A = Matrix( [ [0,2,1], [9,6,3], [0,1,1] ] )
```

El atributo `tex` muestra el proceso sobre la matriz extendida. Bajo cada flecha aparece la transformación de la última fila que compensa las de las columnas, que aparecen sobre la flecha:

```
print(displaymath( Determinante(A).tex ))
```

$$\left[\begin{array}{ccc|c} 0 & 2 & 1 & 0 \\ 9 & 6 & 3 & 0 \\ 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \xrightarrow[\substack{[(3)\mathbf{2}]\\ [(-2)\mathbf{1}+\mathbf{2}]\\ [(3)\mathbf{3}]\\ [(-1)\mathbf{1}+\mathbf{3}]}]{\boldsymbol{\tau}\\ [\left(\frac{1}{9}\right)\mathbf{4}]} \left[\begin{array}{ccc|c} 0 & 6 & 3 & 0 \\ 9 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 \\ \hline 0 & 0 & 0 & \frac{1}{9} \end{array}\right] \xrightarrow[\substack{[(2)\mathbf{3}]\\ [(-1)\mathbf{2}+\mathbf{3}]}]{\boldsymbol{\tau}\\ [\left(\frac{1}{2}\right)\mathbf{4}]} \left[\begin{array}{ccc|c} 0 & 6 & 0 & 0 \\ 9 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 \\ \hline 0 & 0 & 0 & \frac{1}{18} \end{array}\right] \xrightarrow[\substack{[\mathbf{1}\rightleftharpoons\mathbf{2}]}]{\boldsymbol{\tau}\\ [(-1)\mathbf{4}]} \left[\begin{array}{ccc|c} 6 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \\ 3 & 0 & 3 & 0 \\ \hline 0 & 0 & 0 & -\frac{1}{18} \end{array}\right]$$

El determinante es el producto de los elementos de la diagonal de la última matriz, y se guarda en el atributo `valor`:

```
print(dprint( Determinante(A).valor ))
```

```
-9
```

Con el método `determinante` se obtiene solo el valor:

```
print(dprint( A.determinante() ))
```

```
-9
```

En modo texto el objeto indica qué es (el rótulo depende del idioma de los textos de salida):

```
print(dprint( Determinante(A) ))
```

```
Valor del determinante: -9
```

Si el desarrollo no cabe en una línea se puede partir con la función `desarrollo`, como cualquier otro proceso de eliminación:

```
print(displaymath( desarrollo(Determinante(A), por_linea=2) ))
```

$$\left[\begin{array}{cccc|c} 0 & 2 & 1 & 0 & \\ 9 & 6 & 3 & 0 & \\ 0 & 1 & 1 & 0 & \\ \hline 0 & 0 & 0 & 1 & \end{array}\right] \xrightarrow[\substack{[(-2)\mathbf{1}+\mathbf{2}]\\ [(3)\mathbf{3}]\\ [(-1)\mathbf{1}+\mathbf{3}]}]{\boldsymbol{\tau}\\ [(3)\mathbf{2}]} \xrightarrow[[(\frac{1}{9})\mathbf{4}]]{\boldsymbol{\tau}} \left[\begin{array}{cccc|c} 0 & 6 & 3 & 0 & \\ 9 & 0 & 0 & 0 & \\ 0 & 3 & 3 & 0 & \\ \hline 0 & 0 & 0 & \frac{1}{9} & \end{array}\right]$$

$$\xrightarrow[[(\frac{1}{2})\mathbf{4}]]{\boldsymbol{\tau}\\ [(2)\mathbf{3}]\\ [(-1)\mathbf{2}+\mathbf{3}]} \left[\begin{array}{cccc|c} 0 & 6 & 0 & 0 & \\ 9 & 0 & 0 & 0 & \\ 0 & 3 & 3 & 0 & \\ \hline 0 & 0 & 0 & \frac{1}{18} & \end{array}\right] \xrightarrow[[(-1)\mathbf{4}]]{\boldsymbol{\tau}\\ [\mathbf{1}\rightleftharpoons\mathbf{2}]} \left[\begin{array}{cccc|c} 6 & 0 & 0 & 0 & \\ 0 & 9 & 0 & 0 & \\ 3 & 0 & 3 & 0 & \\ \hline 0 & 0 & 0 & -\frac{1}{18} & \end{array}\right]$$

La matriz puede contener símbolos. El resultado es entonces una expresión

```
a = sympy.symbols('a')
B = Matrix( [ [1,a], [a,4] ] )
```

```
print(dprint( Determinante(B).valor ))
```

```
4 - a**2
```

y con `sust` se calcula para valores concretos de los símbolos:

```
print(dprint( Determinante(B, sust=[(a,2)]).valor ))
```

```
0
```

Si la matriz no es cuadrada se produce un error:

```
try:
    Determinante( Matrix( [ [1,2,3], [4,5,6] ] ) )
except ValueError as error:
    print(dprint( error ))
```

```
Matrix no cuadrada
```

Parte VI

La clase `FuncionLineal`

15.7. Implementación de las funciones afines en la clase `FuncionLineal`

15.7.1. Método de inicialización

Inicialización de la clase FuncionLineal

```
def __init__(self, elementos_del_Dominio, elementos_de_la_Imagen):
    """Inicializa un Sistema con dos listas, tuplas o Sistemas de la misma longitud"""
    def conversion_de_R_a_R1(sis):
        return Sistema([Vector([e]) for e in sis])

    if isinstance(BlockV(elementos_del_Dominio), Vector):
        elementos_del_Dominio  = conversion_de_R_a_R1(elementos_del_Dominio)
    if isinstance(BlockV(elementos_de_la_Imagen), Vector):
        elementos_de_la_Imagen = conversion_de_R_a_R1(elementos_de_la_Imagen)

    <<Verificación de que son Ok los argumentos para clase FuncionLineal>>
    <<Metodo auxiliar para el cálculo del nucleo de la función>>

    self.pares   = BlockM([BlockV(elementos_del_Dominio), BlockV(elementos_de_la_Imagen)]).elim(16)
    self.dominio = (self.pares|1).span()
    self.imagen  = (self.pares|2).span()
    self.nucleo  = kernel(self)
```

Verificación de que son Ok los argumentos para clase FuncionLineal

```
if (not es_ristra(elementos_del_Dominio)) or (not es_ristra(elementos_de_la_Imagen)):
    raise ValueError('El argumento debe ser una lista, tupla, o Sistema.')

if not len(elementos_del_Dominio) == len(elementos_de_la_Imagen):
    raise ValueError('Los dos sistemas o lista deben ser igual de largas')

nuevos_pares = BlockM([elementos_del_Dominio, elementos_de_la_Imagen]).elim(1)
if [e for j,e in enumerate(2|nuevos_pares,1) if e.no_es_nulo() and (1|nuevos_pares|j).es_nulo()]:
    dispElim(BlockM([elementos_del_Dominio, elementos_de_la_Imagen]).cele({1}), nuevos_pares.pasos)
    raise ValueError('La lista de pares no corresponde a una función lineal')
```

Metodo auxiliar para el cálculo del nucleo de la función

```
def kernel(self):
    dominio = self.pares|1
    imagen  = self.pares|2
    s_generador = [punto for j,punto in enumerate(dominio, 1) if (imagen|j).es_nulo()]
    return Sistema(s_generador).span(Rn=(dominio).span().Rn)
```

15.8. Métodos la clase `FuncionLineal`

```
def es_invertible(self):
    return self.nucleo.dim==0 and self.es_lineal(reversed(self.pares))

def es_lineal(self,pares):
    nuevos_pares = BlockM([(pares|1).sis(), (pares|2).sis()]).elim(1)
    if [e for j,e in enumerate(2|nuevos_pares,1) if e.no_es_nulo() and (1|nuevos_pares|j).es_nulo()]:
        return False
    else:
        return True

def inversa(self):
    """Devuelve la inversa de la FuncionLineal si existe"""
```

```
    if not self.es_invertible():
        raise ValueError('Esta FuncionLineal no es invertible')

    return FuncionLineal((self.pares|2).sis(), (self.pares|1).sis())
```

```
def imagen_de(self, punto, rep=False):

    if isinstance(BlockV([punto]), Vector):
        punto =  Vector([punto])

    try:
        pertenece = punto in self.dominio
    except ValueError:
        pertenece = False
    if not pertenece:
        raise ValueError('El elemento indicado no pertenece al dominio de la función')

    cero  = self.imagen.vector_nulo()
    X     = BlockV([punto, cero])

    lista_pares_ampliada = (~(BlockM(self.pares).apila(~BlockM([-X]),1))).cele({1})
    nuevos_pares = lista_pares_ampliada.elim(1,rep)

    return 2|nuevos_pares|0
```

```
def preimagen_de(self, punto, rep=False):

    if isinstance(BlockV([punto]), Vector):
        punto =  Vector([punto])

    try:
        pertenece = punto in self.imagen
    except ValueError:
        pertenece = False
    if not pertenece:
        raise ValueError('El elemento indicado no pertenece a la imagen de la función')

    cero  = self.dominio.vector_nulo()
    X     = BlockV([punto, cero])

    pasos_evaluacion = (1|(~reversed(self.pares)).amplia(-X,1)).elim(1).pasos

    representante = 2|((~reversed(self.pares)).amplia(-X,1) & T(pasos_evaluacion[1]))|0

    if rep:
        dispElim((~reversed(self.pares)).amplia(-X,1).cele({1}), pasos_evaluacion)

    return EAfin(self.nucleo.sgen, representante, 1)
```

15.9. Métodos de representación de la clase `FuncionLineal`

Métodos de representación de la clase FuncionLineal

```
def __repr__(self):
    """Muestra el Espacio Nulo de una matriz en su representación Python"""
    return texto_de_salida('pares') + ' ' + repr(self.pares)

def latex(self):
```

```
    """ Construye el comando LaTeX para la clase FuncionLineal"""
    return latex((~self.pares).sis())

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

Definición de la clase FuncionLineal

```
class FuncionLineal:
    <<Inicialización de la clase FuncionLineal>>
    <<Métodos la clase ~FuncionLineal~>>
    <<Métodos de representación de la clase ~FuncionLineal~>>
```

Parte VII

Librería completa

Finalmente creamos la librería `nacal.py` concatenando los trozos de código que se describen en este fichero de documentación.

Importamos el módulo Sympy con el código:

import sympy

Así podremos usar números racionales e irracionales (incluso el cuerpo de polinomios).

Como queremos que la librería emplee números racionales siempre que sea posible, definimos tres métodos generales: `fracc(a/b)` es la fracción $\frac{a}{b}$; `numer(a,b)`; y `denom(a,b)` (véase la página siguiente). Para usar el número racional $\frac{1}{3}$ escribiremos `fracc(1,3)`, y para usar un número irracional como $\sqrt{2}$ escribimos `sympy.sqrt(2)`.

El módulo Sympy se ocupa de que Jupyter represente adecuadamente estos objetos (incluso simplificando expresiones, de manera que si escribimos el número irracional `fracc(2,sympy.sqrt(2))`, es decir $\frac{2}{\sqrt{2}}$, Jupyter lo simplificara representándolo como $\sqrt{2}$).

Para poder trabajar con los Notebooks en Emacs importamos los métodos `display`, `Math` y `display_png` del módulo `IPython.display`.

Para generar las imágenes `png` con las expresiones en LaTeX en un directorio temporal importamos `tempfile` También importamos el método `join` del módulo `os.path`.

```
# coding=utf8
import sympy
from IPython.display import display, Math, display_png
import tempfile
from os.path import join

<<Consideracion de qué es un número>>
<<Comprobación de que un número es cero>>
<<Métodos para usar coeficientes racionales cuando sea posible>>
<<Método html general>>
<<Método latex general>>
<<Pinta un objeto en Jupyter>>
<<Método auxiliar CreaLista que devuelve listas>>
<<Método auxiliar CreaSistema que devuelve sistemas>>
<<Métodos primer_no_nulo y ultimo_no_nulo de un Sistema>>
<<Otros métodos auxilares>>
<<Idioma de los textos de salida>>
<<Presentación de las fracciones>>
<<Cribado de secuencias de transformaciones>>
<<Simplificación de expresiones simbólicas>>
<<Cadena con el código de un desarrollo que recuerda sus eslabones>>
<<Representación de un proceso de eliminación rprElim>>
<<Representación de un proceso de eliminación rprElimFyC>>
<<Representación de un proceso de eliminación rprElimCF>>
<<Método que define los atributos .tex y .pasos y representa los pasos si se pide>>
<<Representación de un proceso de eliminación dispElim, dispElimFyC y dispElimCF>>
<<Separación de un desarrollo en sistemas y flechas>>
<<Recomposición de un desarrollo>>

<<Definición de la clase Sistema>>
<<Definición de la subclase BlockV>>
<<Definición de la subclase Vector>>
<<Definición de las subclases V0 y V1>>
<<Definición de la subclase Polinomio>>
<<Definición de la subclase BlockM>>
<<Definición de la subclase Matrix>>
<<Definición de las subclases M0, M1 e I>>
<<Definición de las subclases de sistemas simbólicos>>
```

```
<<Consideracion de qué es una secuencia>>

<<Definición de la clase T>>

<<Definición de la clase ~SubEspacio~>>
<<Definición de la clase ~EAfin~>>

<<Resolución de un sistema de ecuaciones homogéneo>>
<<Resolución de un Sistema de Ecuaciones Lineales>>

<<Definición de la clase Determinante>>

<<Definición de las subclases ElimX>>
<<Definición de las subclases InvMatX>>
<<Definición de las subclases DiagonalizaX>>

<<Definición de la clase ~FuncionLineal~>>

<<Nombres que exporta el módulo>>
```

Capítulo 16

Nombres que exporta el módulo

La lista `__all__` indica qué nombres quedan disponibles al escribir `from nacal import *`. Sin ella se exportaría todo lo que no comienza por un guión bajo, incluidos los módulos importados y las funciones auxiliares de uso interno.

Se exportan las clases, las funciones de uso general y, por compatibilidad con los cuadernos y documentos que ya usan el módulo, algunos nombres que no son propios: `sympy` (se usa `sympy.symbols` tras importar el módulo), `display`, `Math`, `display_png` y `join`. Nótese que la matriz identidad `I` oculta la unidad imaginaria de Sympy, que sigue disponible como `sympy.I`.

No se exportan (aunque siguen disponibles escribiendo, por ejemplo, `nacal.nacal.es_ristra`) el módulo `tempfile` ni los siguientes nombres de uso interno: `NumberTypes`, `RistraTypes`, `CreaLista`, `es_ristra`, `es_ristra_de_numeros`, `es_funcion_racional`, `elementoPivote`, `elementoAntiPivote`, `fin_del_argumento`, `trozos_del_desarrollo`, `flecha_resumen`, `TexDesarrollo`, `texYpasos`, `TEXTOS_DE_SALIDA`, `idioma_en_uso`, `texto_de_salida`, `error_de_producto`, `fracciones_en_uso`, `salto_de_fila` y `AVISO_DE_PRECEDENCIA`.

Una prueba (`tests/test_exportaciones.py`) comprueba que todo nombre nuevo del módulo está en una de las dos listas, de modo que al añadir una función hay que decidir si se exporta.

Nombres que exporta el módulo

```
__all__ = [
    # clases de sistemas
    'Sistema', 'BlockV', 'Vector', 'V0', 'V1', 'Polinomio',
    'BlockM', 'Matrix', 'M0', 'M1', 'I',
    'SistemaSimbol', 'VectorSimbol', 'MatrixSimbol',
    # transformaciones elementales y procedimientos de eliminación
    'T', 'Elim', 'ElimG', 'ElimGJ', 'ElimF', 'ElimGF', 'ElimGJF',
    'InvMat', 'InvMatF', 'InvMatFC',
    'DiagonalizaS', 'DiagonalizaO', 'DiagonalizaC',
    # subespacios, sistemas de ecuaciones, determinantes y funciones lineales
    'SubEspacio', 'EAfin', 'Homogenea', 'SEL', 'Determinante', 'FuncionLineal',
    # representación
    'latex', 'html', 'pinta', 'desarrollo', 'idioma', 'fracciones',
    'rprElim', 'rprElimFyC', 'rprElimCF', 'dispElim', 'dispElimFyC', 'dispElimCF',
    # números y expresiones simbólicas
    'es_numero', 'es_cero', 'racional', 'fracc', 'numer', 'denom',
    'simplify', 'factor', 'expand',
    # funciones auxiliares que usa el material del curso
    'CreaSistema', 'primer_no_nulo', 'ultimo_no_nulo', 'filtradopasos',
    'particion', 'estructura',
    # nombres ajenos que se mantienen por compatibilidad
    'sympy', 'display', 'Math', 'display_png', 'join',
```

]

Capítulo 17

Métodos generales

17.1. Números racionales

Los números en coma flotante (`float`) no son exactos: `0.1+0.2-0.3` no es cero. Operando con ellos la eliminación puede tomar como pivote un "casi cero" debido al redondeo y dar, por ejemplo, un rango erróneo. Por ello NAcAL convierte los `float` en números racionales leyendo su escritura decimal (así `0.1` es $\frac{1}{10}$ y `0.75` es $\frac{3}{4}$) y opera con ellos de manera exacta. Antes de leer la escritura decimal se busca una fracción sencilla (con denominador menor que un millón) que coincida exactamente con el número: así el resultado de escribir `1/3` en Python, que es el `float` `0.3333333333333333`, se convierte en $\frac{1}{3}$ y no en una fracción de dieciséis cifras. (Para usar números irracionales emplee las funciones de Sympy, por ejemplo `sympy.sqrt(2)`, pues `math.sqrt(2)` es un `float` y se convertiría en una fracción que solo aproxima a $\sqrt{2}$).

Métodos para usar coeficientes racionales cuando sea posible

```
def racional(x):
    """Convierte los números en coma flotante en racionales exactos:
    en la fracción sencilla que coincide con el número (0.1 se convierte
    en 1/10 y 1/3 en un tercio) o, si no la hay, leyendo su escritura
    decimal. Cualquier otro objeto se devuelve sin cambios"""
    def exacto(valor, escritura):
        # una fracción sencilla que sea exactamente ese número en coma
        # flotante (1/3 es Rational(1,3)); si no la hay, su escritura decimal
        from fractions import Fraction
        sencilla = Fraction(valor).limit_denominator(10**6)
        if float(sencilla) == valor:
            return sympy.Rational(sencilla.numerator, sencilla.denominator)
        return sympy.Rational(escritura)

    if isinstance(x, float):
        return exacto(x, repr(x)) if x == x and abs(x) != float('inf') else x
    elif isinstance(x, sympy.Basic) and not x.is_Rational and not x.is_Symbol:
        flotantes = x.atoms(sympy.Float)
        return x.xreplace({f: exacto(float(f), str(f)) for f in flotantes}) if flotantes else x
    else:
        return x

def fracc(a,b):
    """Transforma la fracción a/b en un número racional si ello es posible"""
    a, b = racional(a), racional(b)
    if all( [ isinstance(i, (int, sympy.Rational) ) for i in (a,b) ] ):
        return sympy.Rational(a, b)
    else:
        return a/b
```

```
def numer(a,b):
    """Devuelve el numerador de a/b si la fracción es un número racional,
       si no devuelve a/b"""
    if all( [ isinstance(i, (int, float, sympy.Rational) ) for i in (a,b) ] ):
        return fracc(a,b).p
    else:
        return a/b

def denom(a,b):
    """Devuelve el denominador de a/b si la fracción es un número
       racional, si no devuelve 1"""
    if all( [ isinstance(i, (int, float, sympy.Rational) ) for i in (a,b) ] ):
        return fracc(a,b).q
    else:
        return 1
```

17.2. 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 grafía correspondiente.

Método html general

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

El método `latex` general, intentará llamar al método `latex` particular del objeto que se quiere representar (para todos los objetos de esta librería se define su método de representación). Si el objeto no tiene definido el método `latex`, entonces se emplea el método `sympy.latex()` del módulo Sympy. Así,

```
Sistema([ sympy.Rational(1.5, 3), fracc(2.4, 1.2), fracc(2, sympy.sqrt(2)) ])
```

donde las componentes se han escrito explícitamente como

$$\frac{1{,}5}{3}, \quad \frac{2{,}4}{1{,}2}, \quad \text{y} \quad \frac{2}{\sqrt{2}},$$

será representado como

$$\left[\ \tfrac{1}{2};\quad 2;\quad \sqrt{2};\ \right]$$

Método latex general

```
def latex(a):
    """Método latex general"""
    if hasattr(a, 'latex'):
        return a.latex()
    elif fracciones_en_uso[0] == 'inclinadas':
        return sympy.latex(a, fold_short_frac=True)
    else:
        return sympy.latex(a)
```

Para visualizar un objeto en su formato LaTeX definimos el método `pinta`.

Pinta un objeto en Jupyter

```
def pinta(data):
    """Muestra en Jupyter la representación latex de data"""
    display(Math(latex(data)))
```

Por ejemplo, compárese lo que devuelve `pinta`

```
pinta( Vector([sympy.sqrt(2), 0, fracc(3, 4) ]) )
```

$$\begin{pmatrix} \sqrt{2} \\ 0 \\ \frac{3}{4} \end{pmatrix}$$

con lo que devuelve `print`

```
print( Vector([sympy.sqrt(2), 0, fracc(3, 4) ]) )
```

```
(sqrt(2), 0, 3/4,)
```

17.3. Idioma de los textos de salida

El código LaTeX que genera el módulo no contiene palabras, pero sí las contienen algunas representaciones de texto (las que se ven en un terminal) y un aviso de los métodos de diagonalización. La función `idioma` permite elegir el idioma de esos textos: español (`es`, por defecto) o inglés (`en`). Sin argumentos indica el idioma en uso. Los nombres de clases y métodos no se traducen. Tampoco los mensajes de error, con dos excepciones, que son los primeros errores con que se encuentra quien empieza a usar el módulo. Una es el que se obtiene al multiplicar dos objetos de tamaños incompatibles: ese mensaje indica qué se ha intentado multiplicar y los dos números que deberían coincidir. La otra son los que se obtienen al escribir una abreviatura que no corresponde a una transformación elemental, como `T( (0,2) )`, `T( (-1,2,2) )` o `T( {1,2,3} )`.

```
>>> Matrix([[1,2,3],[4,5,6]]) * Vector([1,2])
ValueError: No se pueden multiplicar: la matriz de la izquierda tiene 3 columnas
            y el vector de la derecha tiene 2 componentes
>>> idioma('en')
>>> Vector([1,2,3]) * Vector([1,2])
ValueError: Cannot multiply: the vector on the left has 3 components
            and the vector on the right has 2 components
```

```
>>> idioma('en')
>>> SEL(Matrix([[1,2],[2,4]]), Vector([1,2]))
Vector([1, 0]) + Linear combinations of (Sistema([Vector([-2, 1]);]))
```

Idioma de los textos de salida

```
TEXTOS_DE_SALIDA = {
    'es': { 'combinaciones':  'Combinaciones lineales de',
            'determinante':   'Valor del determinante:',
            'pares':          'Sistema inicial de pares (Dominio, Imagen):',
            'no_autovalores': 'los valores de la siguiente lista no son autovalores de la matriz:',
            'producto_imposible': 'No se pueden multiplicar: {} de la izquierda tiene {} y {} de la derecha
            ↪  tiene {}',
```

```
            'no_es_transformacion': 'T( {} ) no es una transformación elemental',
            'no_es_intercambio': 'El conjunto debe tener uno o dos índices para ser un intercambio',
            'vector':         'el vector',
            'matriz':         'la matriz',
            'sistema':        'el sistema',
            'componentes':    ('componente', 'componentes'),
            'columnas':       ('columna', 'columnas'),
            'filas':          ('fila', 'filas'),
            'elementos':      ('elemento', 'elementos') },
    'en': { 'combinaciones':  'Linear combinations of',
            'determinante':   'Value of the determinant:',
            'pares':          'Initial system of pairs (Domain, Image):',
            'no_autovalores': 'the values in the following list are not eigenvalues of the matrix:',
            'producto_imposible': 'Cannot multiply: {} on the left has {} and {} on the right has {}',
            'no_es_transformacion': 'T( {} ) is not an elementary operation',
            'no_es_intercambio': 'The set must have one or two indices to be an interchange',
            'vector':         'the vector',
            'matriz':         'the matrix',
            'sistema':        'the system',
            'componentes':    ('component', 'components'),
            'columnas':       ('column', 'columns'),
            'filas':          ('row', 'rows'),
            'elementos':      ('element', 'elements') },
}

idioma_en_uso = ['es']

# Python calcula las sumas y las restas antes que el selector |
AVISO_DE_PRECEDENCIA = ('Si quería operar con componentes, use paréntesis: '
                        '(a|1) + (b|1), pues Python calcula a|1 + b|1 como a|(1 + b)|1')

def idioma(codigo=None):
    """Elige el idioma de los textos de salida ('es' o 'en')

    Sin argumentos devuelve el idioma en uso. Solo afecta a las
    representaciones de texto y al mensaje de error de un producto
    imposible (no al código LaTeX, que no contiene palabras, ni a los
    demás mensajes de error).

    >>> idioma()
    'es'
    """
    if codigo is None:
        return idioma_en_uso[0]
    if codigo not in TEXTOS_DE_SALIDA:
        raise ValueError('Idiomas disponibles: ' + ', '.join(sorted(TEXTOS_DE_SALIDA)))
    idioma_en_uso[0] = codigo

def texto_de_salida(clave):
    """Texto correspondiente a la clave en el idioma en uso"""
    return TEXTOS_DE_SALIDA[idioma_en_uso[0]][clave]

def error_de_producto(izda, dcha):
    """Mensaje de error del producto de izda por dcha, en el idioma en uso

    Indica la clase de cada operando y los dos números que deberían
    coincidir: el número de elementos del Sistema de la izquierda y el
    número de componentes del Vector (o de filas de la Matrix) de la
    derecha.
    """
    def cuantos(numero, clave):
        return str(numero) + ' ' + texto_de_salida(clave)[0 if numero == 1 else 1]

    if isinstance(izda, Vector):
        quien_izda, que_izda = 'vector', 'componentes'
    elif isinstance(izda, Matrix):
        quien_izda, que_izda = 'matriz', 'columnas'
    else:
```

```
        quien_izda, que_izda = 'sistema', 'elementos'

    if isinstance(dcha, Vector):
        quien_dcha, que_dcha, numero_dcha = 'vector', 'componentes', dcha.n
    else:
        quien_dcha, que_dcha, numero_dcha = 'matriz', 'filas', dcha.m

    return texto_de_salida('producto_imposible').format(
        texto_de_salida(quien_izda), cuantos(len(izda), que_izda),
        texto_de_salida(quien_dcha), cuantos(numero_dcha, que_dcha))
```

17.4. Presentación de las fracciones

En una matriz con fracciones en filas contiguas, el denominador de una fila queda muy cerca del numerador de la fila siguiente. Por eso, desde la versión 0.5.5, el módulo añade un poco de espacio entre las filas de las matrices y vectores que contienen fracciones. La función `fracciones` permite elegir otra presentación para todo lo que se escriba a continuación. Sin argumentos indica el modo en uso.

Modo	Presentación
`'separadas'`	más espacio entre las filas de las matrices y vectores que contienen fracciones (modo por defecto)
`'normal'`	sin espacio añadido (la presentación de las versiones anteriores a la 0.5.5)
`'inclinadas'`	fracciones con barra inclinada, como 1/2

En el modo `'separadas'` el espacio añadido es de `0.6ex`; con un segundo argumento se indica otra longitud de LaTeX, por ejemplo `fracciones('separadas', '1ex')`. Las matrices y vectores sin fracciones no cambian. El modo `'inclinadas'` afecta a todas las fracciones, también a las de los rótulos de las transformaciones.

Presentación de las fracciones

```
fracciones_en_uso = ['separadas', '0.6ex']

def fracciones(modo=None, separacion=None):
    """Elige cómo se escriben en LaTeX las fracciones

    Modos:
        'separadas':  añade espacio entre las filas de las matrices y
                      vectores que contienen alguna fracción (por defecto)
        'normal':     sin espacio añadido
        'inclinadas': fracciones con barra inclinada, como 1/2

    separacion es la longitud LaTeX que se añade entre filas en el modo
    'separadas' (por defecto '0.6ex'). Sin argumentos devuelve el modo
    en uso.

    >>> fracciones()
    'separadas'
    """
    if modo is None:
        return fracciones_en_uso[0]
    if modo not in ('normal', 'separadas', 'inclinadas'):
        raise ValueError("Modos disponibles: 'normal', 'separadas' e 'inclinadas'")
    fracciones_en_uso[0] = modo
    fracciones_en_uso[1] = '0.6ex' if separacion is None else str(separacion)

def salto_de_fila(elementos):
    """Código LaTeX del salto de fila de un arreglo con dichos elementos"""
    if fracciones_en_uso[0] == 'separadas' and any('\\frac' in latex(e) for e in elementos):
```

```
        return '\\\\[' + fracciones_en_uso[1] + ']'
    return '\\\\'
```

17.4.1. Ejemplos de presentación de las fracciones

```
A = Matrix( [ [fracc(1,2), 0], [fracc(1,2), fracc(9,10)] ] )
```

```
print(dmath( A ))
```

$$\begin{bmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{9}{10} \end{bmatrix}$$

```
fracciones('normal')
print(dmath( A ))
```

$$\begin{bmatrix} \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{9}{10} \end{bmatrix}$$

```
fracciones('inclinadas')
print(dmath( A ))
```

$$\begin{bmatrix} 1/2 & 0 \\ 1/2 & 9/10 \end{bmatrix}$$

Para volver a la presentación por defecto:

```
fracciones('separadas')
print(dprint( fracciones() ))
```

```
separadas
```

17.5. Simplificación de expresiones simbólicas

Simplificación de expresiones simbólicas

```
def simplify(self):
    """Devuelve las expresiones simplificadas"""
    if isinstance(self, (list, tuple)):
        return type(self)([ simplify(e) for e in self ])
    elif isinstance(self, Sistema):
        self.lista=[ simplify(e) for e in self ]
        return self
    elif isinstance(self, T):
        return self.simplify()
    else:
        return (sympy.sympify(self)).simplify()

def factor(self):
    """Devuelve las expresiones factorizadas"""
    if isinstance(self, (list, tuple)):
        return type(self)([ factor(e) for e in self ])
    elif isinstance(self, Sistema):
```

```
        self.lista=[ factor(e) for e in self ]
        return self
    elif isinstance(self, T):
        return self.factor()
    else:
        factorizada = sympy.factor(self)
        if not isinstance(factorizada, sympy.Expr) or factorizada.is_Number or
        ↪  es_funcion_racional(factorizada):
            return factorizada
        return sympy.S.Zero if es_cero(factorizada) else factorizada

def expand(self):
    """Devuelve las expresiones factorizadas"""
    if isinstance(self, (list, tuple)):
        return type(self)([ expand(e) for e in self ])
    elif isinstance(self, Sistema):
        self.lista=[ expand(e) for e in self ]
        return self
    elif isinstance(self, T):
        return self.expand()
    else:
        return sympy.expand(self)
```

17.6. Números

Consideraremos como números los enteros, los números de coma flotante, los números complejos, y toda la familia de objetos de tipo `Basic` del módulo Sympy, lo que nos permite trabajar con números racionales, muchos números reales y con expresiones simbólicas.

Consideracion de qué es un número

```
NumberTypes = (int, float, complex, sympy.Basic)
es_numero   = lambda x: isinstance(x, NumberTypes) and not isinstance(x, bool)
```

Para la eliminación es fundamental saber si un número es cero (pues los pivotes son números no nulos). Con expresiones simbólicas no basta comparar la expresión con `0`: por ejemplo, Sympy no simplifica automáticamente $\sin^2 x + \cos^2 x - 1$, que es cero aunque no *se escriba* como `0`. La función `es_cero` se ocupa de esta comprobación. Para no ralentizar los cálculos solo recurre a `sympy.simplify` cuando la expresión no es una función racional de sus símbolos con coeficientes racionales (caso en el que basta desarrollar el numerador; y si la expresión ya está factorizada, ni siquiera eso: basta compararla con `0`).

Comprobación de que un número es cero

```
def es_funcion_racional(x):
    """Indica si la expresión x es una función racional de sus símbolos
    con coeficientes racionales"""
    return x.is_rational_function(*x.free_symbols)                    \
        and all(a.is_Symbol or a.is_Rational for a in x.atoms())      \
        and all(p.exp.is_Integer for p in x.atoms(sympy.Pow))

def es_cero(x):
    """Indica si es cierto que el número (o la expresión simbólica) x es cero"""
    if not isinstance(x, sympy.Basic):
        return x == 0
    if x.is_Number:
        return bool(x == 0)
    if not isinstance(x, sympy.Expr):
        return False
    if es_funcion_racional(x):
        return sympy.fraction(sympy.together(x))[0].expand() == 0
    if x.is_zero is not None:
        return bool(x.is_zero)
```

```
    simplificada = sympy.simplify(x)
    return bool(simplificada == 0 or simplificada.is_zero)
```

17.7. Ristras

Según el diccionario de la Real Academia Española una *ristra* es un *conjunto de ciertas cosas colocadas unas tras otras*; por ello, denominamos genéricamente "ristra" a cualquier objeto de tipo `list`, `tuple` o `Sistema`.

Consideracion de qué es una secuencia

```
RistraTypes = (tuple, list, Sistema)
es_ristra  = lambda x: isinstance(x, RistraTypes)

def es_ristra_de_numeros(arg):
    return all( [es_numero(elemento) for elemento in arg] ) if es_ristra(arg) else None
```

17.8. Métodos `CreaLista` y `CreaSistema`

Método auxiliar CreaLista que devuelve listas

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

Método auxiliar CreaSistema que devuelve sistemas

```
def CreaSistema(t):
    """Devuelve t si t es un Sistema; si no devuelve un Sistema que contiene t"""
    return t if isinstance(t, Sistema) else Sistema(CreaLista(t))
```

17.9. Métodos `primer_no_nulo` y `ultimo_no_nulo` de un Sistema

Métodos primer_no_nulo y ultimo_no_nulo de un Sistema

```
def primer_no_nulo(s):
    """Primer elemento no nulo en un sistema de sistemas de números"""
    c = [] if es_numero(s) else s.primer_no_nulo()
    return c + primer_no_nulo(CreaSistema(s)|c[0]) if c!=[] else []

def ultimo_no_nulo(s):
    """Último elemento no nulo en un sistema de sistemas de números"""
    c = [] if es_numero(s) else s.ultimo_no_nulo()
    return c + ultimo_no_nulo(CreaSistema(s)|c[0]) if c!=[] else []

def elementoPivote(s):
    """Primer elemento no nulo"""
    if es_numero(s):
        return s
    elif CreaSistema(s).elementoPivote():
        return elementoPivote(CreaSistema(s).extractor(CreaSistema(s).primer_no_nulo()))
    else:
        None

def elementoAntiPivote(s):
    """Último elemento no nulo"""
    if es_numero(s):
        return s
```

```
    elif CreaSistema(s).elementoAntiPivote():
        return elementoAntiPivote(CreaSistema(s).extractor(CreaSistema(s).ultimo_no_nulo()))
    else:
        None
```

17.10. Otros métodos auxilares

```
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]]
    """
    s = {e for e in s if e<=n}
    p = sorted(list(s | set([0,n])))
    return [ list(range(p[k]+1,p[k+1]+1)) for k in range(len(p)-1) ]

def estructura(S):
    if isinstance(S, Sistema):
        c = [type(S)]
        for item in S:
            if not isinstance(item, Sistema):
                if es_numero(item):
                    c = c + [('num',)]
                else:
                    c = c + [(type(item),)]
            else:
                c = c + [item.estructura()]
    else:
        if es_numero(S):
            c = [('num',)]
        else:
            c = [(type(S),)]

    return c
```

17.11. Cribado de secuencias de transformaciones

En el proceso de eliminación, muchas transformaciones elementales realmente son identidades (sumar 0 veces otro vector y multiplicar un vector por 1).

A la hora de representar los pasos de eliminación, normalmente es mejor omitir estos pasos innecesarios. Definimos un procedimiento general que quita de una lista de abreviaturas aquellas que son innecesarias en la representación. Si como argumento se le da una lista de abreviaturas, devuelve una lista filtrada. Si como argumento se le da una Transformación, devuelve una Transformación cuya lista de abreviaturas está filtrada.

Cribado de secuencias de transformaciones

```
def filtradopasos(pasos):
    abv = pasos.abreviaturas if isinstance(pasos,T) else pasos

    p = [T([j for j in T([abv[i]]).abreviaturas 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(abv)) ]

    abv = [ t for t in p if t.abreviaturas] # quitamos abreviaturas vacías de la lista
```

```
    return T(abv) if isinstance(pasos,T) else abv
```
