Índice general

Notación Asociativa para un Curso de Álgebra Lineal (NAcAL)

https://github.com/mbujosab/nacallib (Versión: 0.2.0)

Marcos Bujosa

30 de marzo de 2024

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).

Para subrayar el paralelismo entre las definiciones del curso y los objetos (o `Clases`) definidos en NAcAL, las partes auxiliares del código se relegan al final.[1] Esto permitirá apreciar cómo las definiciones del libro son implementadas de manera literal en la librería de Python.

Antes de seguir, repase el Notebook **"Listas y tuplas"** en la carpeta "TutorialPython" en
`https://github.com/mbujosab/nacallib/tree/master/doc/Notebooks/TutorialPython`

Y recuerde que

¡hacer matemáticas y programar son prácticamente la misma cosa!

[1] aquellas que tienen que ver con la comprobación de que los inputs de las funciones son adecuados, con otras formas alternativas de instanciar clases, con la representación de objetos en Jupyter usando código LaTeX, etc.

Parte I

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

Introducción

En el primer capítulo se define una *clase* para los *sistemas* (listas ordenadas de objetos) denominada `Sistema`; en los capítulos siguientes se definen algunas subclases de la clase `Sistema` (entre otras una para los *vectores* y otra para las *matrices*).[2]

En otras partes de esta documentación se definen clases que implementan otros objetos empleados en el curso de Álgebra Lineal: transformaciones elementales, subespacios vectoriales, espacios afines, etc.

Cada vez que se define una nueva clase, se debe especificar tanto el modo de instanciar la clase como sus atributos, sus modos de representación (en particular el modo de representación para que los Notebooks de Jupyter muestren representaciones semejantes a las empleadas en el curso de Álgebra Lineal), así como todos los métodos que actúan sobre ella.

[2] Antes de seguir, mírese el Notebook referente a "Clases" en la carpeta "TutorialPython" en `https://github.com/mbujosab/nacallib/tree/master/doc/Notebooks/TutorialPython`.

Capítulo 1

La clase `Sistema`

En el libro del curso se dice que

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

Aunque Python ya posee `listas`, vamos a crear nuestra propia clase denominada `Sistema`. De esta manera podremos implementar los métodos descritos en el curso para los sistemas (que no son los mismos que Python define para las `listas`). Por añadidura adecuaremos la representación de los sistemas a la empleada en el libro, donde los sistemas genéricos se muestran entre *corchetes* y con los elementos de la lista *seguidos de* " ; ".[1]

La clase `Sistema` posee un atributo llamado `lista`, que es una `list` de Python con la lista de objetos que componen el `Sistema`. Una característica de un `Sistema` es que cada uno de sus elementos será mostrado con su propia representación. Por ejemplo, el `Sistema`

```
1 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; y en un Notebook de Jupyter lo veremos 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).

Los `Sistemas` y las `listas` de Python no solo se diferencian en el modo de representar los objetos que contienen, también en muchos de sus métodos. Aunque algunos son comunes (en ciertos aspectos un `Sistema` se comporta como una `list`) otros son específicos de los `Sistemas`, y otros se han modificado para poder reservar los símbolos de las operaciones algebraicas. Por ejemplo, las `listas` de Python se concatenan con "+", pero los `Sistemas` se concatenan con el método `concatena()`. Así reservamos el símbolo "+" para sumar `Sistemas` componente a componente como se hace en Álgebra Lineal. Con ello buscamos que lo que veamos y escribamos en un Notebook de Jupyter sea lo más parecido posible a lo que vemos y escribimos en el curso de Álgebra Lineal.

[1] Como los vectores y las matrices también son sistemas, crearemos para ellos subclases con las respectivas representaciones tal y como se describen en el curso.

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)`:

```
"""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.

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}) )  )           # con tupla
>>> Sistema( Sistema( [ 10, 'hola', T({1,2}) ] ) ) # con 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 14.3).
- 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 `.posicionDivisiones` 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).

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

    self.n  = len(self)
    self.posicionDivisiones = {0}
```

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, valor):
    """ Modifica el i-ésimo coeficiente del Sistema """
    self.lista[i] = 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` (aunque lograremos el mismo resultado con `B = Sistema(A)`).

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

Más adelante se añade otro Método para copiar un Sistema y sus atributos (`fullcopy()`).

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 self.lista == other.lista

def __ne__(self, other):
    """Indica si es cierto que dos Sistemas son distintos"""
    return self.lista != other.lista
```

Con `A.reverse()` invertimos el orden de los elementos de `A` (cambiamos `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.posicionDivisiones =  {len(self)-i for i in self.posicionDivisiones}
    self.lista.reverse()

def __reversed__(self):
    """Reversed(S) devuelve una copia de S con la lista en orden inverso"""
    copia = self.fullcopy()
    copia.reverse()
    return copia
```

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

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 `.posicionDivisiones`). 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.

```
def concatena(self, other, marcasVisuales = False):
    """Concatena dos Sistemas"""
    def nuevoConjuntoMarcas(Sistema_A, Sistema_B):
        return Sistema_A.posicionDivisiones.union(
            {len(Sistema_A)},
            {len(Sistema_A)+indice for indice in Sistema_B.posicionDivisiones} )

    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.posicionDivisiones = nuevoConjuntoMarcas(self, other)

    return sistemaAmpliado if self.es_arreglo_rectangular() else 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.

```
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)
```

```
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)
```

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

Sustitución de variables simbólicas.

```
def subs(self, reglasDeSustitucion=[]):
    """ Sustitución de variables simbólicas """
    reglas = CreaLista(reglasDeSustitucion)
    self.lista = [ sympy.S(elemento).subs(CreaLista(reglas)) for elemento in self.lista ]
    return self
```

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

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

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`

```
def simplify(self):
    """ Simplificación de expresiones simbólicas """
    self.lista = [ sympy.S(elemento).simplify() for elemento in self.lista ]

def factor(self):
    """ Factorización de expresiones simbólicas """
    self.lista = [ sympy.S(elemento).factor() for elemento in self.lista ]
```

```
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"""
    return self.subs(sust) == self*0

def no_es_nulo(self, sust=[]):
    """Indica si es cierto que el Sistema no es nulo"""
    return self.subs(sust) != self*0
```

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 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 primer 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('El orden indicado es incompatible con el número de elementos del Sistema')
        return None
```

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, sust=[]):
    """Devuelve la lista o EAfin con las soluciones x de sistema*x=v

    """
    A           = self.amplia(-v)
    operaciones = A.elim(1,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 (K|0).no_es_nulo():
        return Sistema([])
    else:
        solP = (I(self.n).amplia(V0(self.n)) & T(pasos))|0
        if self.espacio_nulo().sgen.es_nulo():
            return Sistema([solP])
        else:
            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__)`.

```
"""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([1,0]), Vector([0,2]), Vector([3,0])]) | (2,1)

[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)

[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).

```
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`.

```
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

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

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 $\mathsf{A}_{|i}$ y $\mathsf{B}_{|i}$ 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__)`.

```
"""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__)`.

```
"""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])])
"""
```

```
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.posicionDivisiones.update(other.posicionDivisiones)
    return 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.posicionDivisiones.update(other.posicionDivisiones)
    return diferencia
```

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*

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

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*:

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

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

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

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`

```
"""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])
"""
```

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

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

```
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 $\mathbf{A}$.

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 $\mathbf{X}$ (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__)`.

```
"""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([[10,20],[30,40]])
>>> # 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 vector resultante es una combinación lineal de las filas de la matriz, la representación del resultado sera en forma horizontal (`rpr='fila'`) si se emplea la representación `latex`.

```
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('Sistema y Vector incompatibles')
        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 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('Sistema y Matrix incompatibles')
        if isinstance(self, BlockV):
            return BlockV( [ self*(x|j) for j in range(1,(x.n)+1)], rpr='fila' )
        elif isinstance(self, BlockM):
            return BlockM ( [ self*(x|j) for j in range(1,(x.n)+1)] )
```

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

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

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

    T(abreviaturas): transformaciones a aplicar sobre un Sistema S
Ejemplos:
>>>  S & T({1,3})               # Intercambia los elementos 1º y 3º
>>>  S & T((5,1))               # Multiplica por 5 el primer elemento
>>>  S & T((5,2,1))             # Suma 5 veces el 2º elem al 1º
>>>  S & T([{1,3},(5,1),(5,2,1)])# Aplica la secuencia de transformac.
             # sobre los elementos de S y en el orden de la lista
"""
```

1.3.2. Implementación de las transformaciones elementales por la derecha

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

```
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):
            self.lista = [ (abrv[0])*(self|abrv[1]) + (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 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)

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

1.4. Eliminación

1.4.1. Reducción por eliminación mediante transformaciones elementales

```
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
    return opcion
```

```
def metodos_auxiliares_de_la(variante):
    """Define los métodos auxilares y el módo de actuació sobre el sistema
    en función de la variante de elimiación 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 arg = 0 (reducción simple hacia delante, por
    columnas y evitando fraciones)

    """
    if 100 in analisis_opcion_elegida(variante): # reducción 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:                                    # reducción 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

```

```
42
43  def Reduccion(sistema):
44      if 1 in analisis_opcion_elegida(variante):   # reducción rápida (solo trasformaciones tipo I)
45          operaciones = [ (-fracc(ValorAEliminar(indiceVAE), pivote), indiceXP, indiceVAE)  \
46                                                      for indiceVAE in componentesAmodificar(sistema)]
47      else:                                        # reducción lenta (evitando fracciones)
48          operaciones = [[( denom(ValorAEliminar(indiceVAE), pivote),           indiceVAE), \
49                            (-numer(ValorAEliminar(indiceVAE), pivote), indiceXP, indiceVAE)] \
50                                                      for indiceVAE in componentesAmodificar(sistema)]
51      return filtradopasos(T(operaciones))
52
53  def Normalizacion(sistema):
54      return filtradopasos(T([ (fracc(1, XPivote(sistema|indiceXP)), indiceXP)
55                               for indiceXP,_ in recorrido(sistema) if XPivote(sistema|indiceXP)]))
56
57  def Escalonamiento(sistema):
58      M = sistema.copy()
59      if 100 in analisis_opcion_elegida(variante): # con reducción hacia atrás
60          destino       = lambda     : (M.n)-r+1
61          resto         = lambda    r: slice(None, max(M.n-r,1))
62          columnaAMover = lambda i, r: posicionXPivote(i|M|resto(r))[0]   if posicionXPivote(i|M|resto(r)) and i==posicionXPivote(
63      else:                                        # con reducción hacia delante
64          destino       = lambda     : r
65          resto         = lambda    r: slice(r+1, None)
66          columnaAMover = lambda i, r: posicionXPivote(i|M|resto(r))[0]+r if posicionXPivote(i|M|resto(r)) and i==posicionXPivote(
67
68      r = 0
69      intercambios = []
70      for i,_ in recorrido(M|1):
71          indiceColumnaPivote = columnaAMover(i,r)
72          if indiceColumnaPivote:
73              r += 1
74              intercambio  = T( {destino(), indiceColumnaPivote} )
75              M & intercambio
76              intercambios.append(intercambio)
77
78      return filtradopasos(T(intercambios))
79
80
81  def transformacionYPasos(sistema, operacion, pasosPrevios):
82      pasoDado = operacion(sistema)
83      if 4 in analisis_opcion_elegida(variante):    # reducción de los componentes en arreglos rectangulares
84          pasosAcumulados = [~pasoDado] + pasosPrevios if pasoDado else pasosPrevios
85      else:
86          pasosAcumulados = pasosPrevios  + [pasoDado] if pasoDado else pasosPrevios
87      sistema & T(pasoDado)
88      return sistema.subs(sust), pasosAcumulados
89
90
91  def sistemaFinalYPasosDchaIzda(sistema,transformaciones):
92      if 4 in analisis_opcion_elegida(variante):    # reducción de los componentes en arreglos rectangulares
93          TransformacionesPorLaIzquierda = filtradopasos(transformaciones)
94          TransformacionesPorLaDerecha   = []
95          if self.es_arreglo_rectangular():
96              sistema = ~sistema
97      else:
98          TransformacionesPorLaDerecha   = filtradopasos(transformaciones)
99          TransformacionesPorLaIzquierda = []
100
101     SistemaFinal =  sistema.subs(sust)
102     pasos        = [TransformacionesPorLaIzquierda, TransformacionesPorLaDerecha]
103     SistemaFinal.tex, SistemaFinal.pasos = texYpasos(self, pasos, rep, sust, repsust)
104     SistemaFinal.TrF = T(SistemaFinal.pasos[0])
105     SistemaFinal.TrC = T(SistemaFinal.pasos[1])
106     return SistemaFinal
```

```
1   def elim(self, variante=0, rep=False, sust=[], repsust=False):
```

```
    """Versión pre-escalonada de un sistema por eliminacion Derecha-Izquierda"""
    <<Método que define los atributos .tex y .pasos y representa los pasos si se pide>>
    <<Variantes 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)
```

```
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)
```

1.4.2. 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 `PasosYEscritura`. El atributo `pasos` guarda las listas de abreviaturas de las transformaciones elementales empleadas. Por comodidad añadimos dos atributos más: `TrF` es la Ttransformación aplicada a las filas y `TrC` es la Transformación aplicada a las columnas.

```
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`).

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

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

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

    return tex
```

```
def rprElimCF(data, pasos, TexPasosPrev=[], sust=[]):
    """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)
    tex = latex(data) if not TexPasosPrev else TexPasosPrev

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

    return tex
```

```
def rprElimFyC(data, pasos, TexPasosPrev=[], sust=[]):
    """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)
    tex = latex(data) if not TexPasosPrev else TexPasosPrev

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

    return tex
```

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

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

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

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

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 `.posicionDivisiones` (*"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.posicionDivisiones,self.n)]
    return '[' + \
             ';|'.join(['; '.join([str(c) for c in s]) \
                        for s in [ self|i for i in particion(self.posicionDivisiones, 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 `.posicionDivisiones` 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 """
    pc = ';' if len(self) else r'\ '
    ln = [len(i) for i in particion(self.posicionDivisiones, 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 14.4) 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:
        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 `ccol()` (que usaremos para visualmente *cortar por columnas* una matriz).

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

    """
    self.posicionDivisiones = 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.ccol({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 `.ccol({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='fila').ccol({2}), vv , 1492] )
ZZ = Sistema([ Z, 'Hola', Matrix([Vector([0,sympy.pi,fracc(a,2)]),Vector([0,0,0])])  ]).ccol({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])]);])
```

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

1.6. La clase `Sistema` completa

```
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>>
    <<Sustitución de variables simbólicas>>
    <<Simplificación de expresiones simbólicas>>
    <<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

```
"""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='fila' 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.
    posicionDivisiones (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,)) )            # con una tupla
>>> BlockV( Sistema( [1,'abc',(2,)] ) ) # con un Sistema
>>> 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 `fila`, es decir, `rpr='fila'`.
- 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 `posicionDivisiones` 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 14.1) el objeto que se crea es un `Vector` (una subclase de `BlockV` que solo contiene números).

```
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
```

2.2. Representación de la clase `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.posicionDivisiones,self.n)]
    return '(' + \
        ',|'.join([', '.join([str(c) for c in s]) \
                   for s in [ self|i for i in particion(self.posicionDivisiones, self.n)]]) + \
        pc + ')'

def latex(self):
    """ Construye el comando LaTeX para representar un BlockV """
    pc = ',' if len(self) else r'\ '
    ln = [len(n) for n in particion(self.posicionDivisiones,self.n)]
    if self.rpr == 'fila' 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:
        return \
            r'\left( \begin{array}{c}' + \
            r'\\ \hline '.join([r'\\'.join([latex(c) for c in e]) \
                                for e in [ self|i for i in particion(self.posicionDivisiones, self.n)]]) + \
            r'\\ \end{array} \right)'
```

2.2.1. 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: $\left(\begin{array}{|c|c|c} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}\right] \\ \hline 1 \\ \hline 0 \\ \hline 0 \end{array} \right)$

y también la podemos usar en horizontal: $\left(\begin{array}{|c|c|c|c} \left[\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}\right] & 1, & 0, & 0, \end{array} \right)$

2.3. La clase `BlockV` completa

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

```
"""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 como columna (por defecto) o como fila.

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 'columna' por defecto).
        Si rpr='fila' el vector se representa en forma de fila.

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.
    posicionDivisiones (set) : Conjunto de índices donde pintar
                               separaciones visuales
```

```
24
25  Ejemplos:
26  >>> # Instanciación a partir de una lista, tupla o Sistema de números
27  >>> Vector( [1,2,3] )           # con lista
28  >>> Vector( (1,2,3) )           # con tupla
29  >>> Vector( Sistema( [1,2,3] ) )# con Sistema
30  >>> Vector( Vector ( [1,2,3] ) )# a partir de otro Vector
31
32  Vector([1,2,3])
33  """
```

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 14.3). 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`).

```
1  def __init__(self, arg, rpr='columna'):
2      """Inicializa Vector con una lista, tupla o Sistema de números"""
3      if not es_ristra_de_numeros(arg):
4          raise ValueError('no todos los elementos son números o parámetros!')
5
6      super().__init__(arg)
```

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.

```
1  def norma(self):
2      """Devuelve la norma de un vector"""
3      return sympy.sqrt(self*self)
4
5  def normalizado(self):
6      """Devuelve un múltiplo de norma uno si el vector no es nulo"""
7      if self.es_nulo(): raise ValueError('Un vector nulo no se puede normalizar')
8      return self * fracc(1,self.norma())
```

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

```
1  def diag(self):
2      """Crea una Matrix diagonal cuya diagonal es self"""
```

```
3       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 forma de fila

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

o en forma de columna

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

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`.

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

3.4. La clase `Vector` completa

```
1  class Vector(BlockV):
2      <<Texto de ayuda de la subclase Vector>>
3      <<Inicialización de la subclase Vector>>
4      <<Normalización de un Vector>>
5      <<Creación de una Matrix diagonal a partir de un Vector>>
6      <<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 de fila o de columna.

```
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 = 'fila' 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 = 'fila' la representación
        es horizontal (en otro caso es vertical)

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

Capítulo 5

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`.

5.1. Implementación

5.1.1. Texto de ayuda

```
"""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.
    posicionDivisiones (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])])

"""
```

5.1.2. Método de inicialización:

```
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.

    """
    super().__init__(arg)

    lista = Sistema(arg).lista

    if all([(isinstance(elemento, BlockV) and len(elemento)==len(lista[0])) for elemento in lista]):
        self.lista   = lista.copy()

    elif all([(es_ristra(elemento) and len(elemento)==len(lista[0])) for elemento in lista]):

        self.lista = BlockM([ BlockV([ elemento[i] for elemento in lista ]) for i in range(len(lista[0])) ]).lista

    elif isinstance(arg, BlockM):
        self.lista   = arg.lista.copy()

    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!""")

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

    self.corteElementos = {0}

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

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

5.2. Métodos de la clase `BlockM`

5.2.1. Transposición de un BlockM

```
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([ Sistema(columna) for columna in self ])
    M.corteElementos, M.posicionDivisiones = self.posicionDivisiones, self.corteElementos
    return M
```

5.2.2. 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.

```
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()
```

5.2.3. Apilado de `BlockMs`

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

```
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)
    return apila([self] + [s for s in CreaLista(lista)])
```

5.2.4. Operador selector por la izquierda

```
"""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]])
>>> (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])])

"""
```

```
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='fila' )

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

5.3. Otros métodos de la clase `BlockM`

5.3.1. 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):
        return 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)
```

5.3.2. Vectoriza un `BlockM` apilando sus elementos para formar un `BlockV`

```
def vec(self):
    "Vectoriza un BlockM apilando sus elementos para formar un BlockV"
    return BlockV(Sistema([]).junta(self))
```

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

5.4.1. Texto de ayuda

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

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

"""
```

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

```
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
```

5.5. Representación de la clase BlockM

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

```
def __str__(self):
    """ Muestra un BlockM en su representación Python str """
    ln  = [len(n) for n in particion(self.posicionDivisiones,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.posicionDivisiones, 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 """
    ln = [len(n) for n in particion(self.posicionDivisiones, self.n)]
    return \
        '\\left[ \\begin{array}{' + '|'.join([n*'c' for n in ln])  + '}' + \
        '\\\\ \\hline '.join(['\\\\'.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 distintas partes de los `BlockV` (las 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 `cfil()` (pues lo usaremos para visualmente *cortar por filas* una matriz).

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

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

5.6. La clase `BlockM` completa

```
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>>
    <<Vectoriza un BlockM apilando sus elementos para formar un BlockV>>
    <<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 6

La subclase `Matrix`

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

6.1.1. Texto de ayuda

```
"""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.
    posicionDivisiones (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]) ])

"""
```

6.1.2. Método de inicialización:

```
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)

    lista = Sistema(data).lista

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

    elif Sistema(lista).es_de_composicion_y_longitud_uniforme() and es_ristra(lista[0]) and es_numero(lista[0][0]):
        self.lista = Matrix([ Vector([elemento[i] for elemento in lista]) for i in range(len(lista[0])) ]).lista

    elif isinstance(data,Matrix):
        self.lista   = data.lista.copy()

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

    super().__init__(data)

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

6.2. Métodos específicos de la subclase `Matrix`

6.2.1. 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_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"""
```

```
24      return self.es_triangularSup() & self.es_triangularInf()
```

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

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

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

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

6.2.4. 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 describirá en el Capítulo~**??**.

```
1   def __pow__(self,n):
2       """Calcula la n-ésima potencia de una Matrix"""
3       if not isinstance(n,int): raise ValueError('La potencia no es un entero')
4       if not self.es_cuadrada:  raise ValueError('Matrix no es cuadrada')
5
6       M = self if n else I(self.n)
7       for i in range(1,abs(n)):
8               M = M * self
9
10      return M.inversa() if n < 0 else M
```

6.2.5. Determinante mediante la expansión de Laplace

```
1   def det(self, sust=[]):
2       """Calculo del determinate mediante la expansión de Laplace"""
3       if not self.es_cuadrada(): raise ValueError('Matrix no cuadrada')
4
5       def cof(self,f,c):
6           """Cofactor de la fila f y columna c"""
7           excl = lambda k: tuple(i for i in range(1,self.m+1) if i!=k)
8           return (-1)**(f+c)*(excl(f)|self|excl(c)).det()
9
10      if self.m == 1:
11          return 1|self|1
12
13      A = Matrix(self.subs(sust))
14      # expansión por la 1ª columna
15      return sympy.simplify(sum([((f|A|1)*cof(A,f,1)).subs(sust) for f in range(1,A.m+1)]))
```

6.2.6. Método de Gram-Schmidt

```
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
```

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

Rango de una Matrix

```
def rg(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

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

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

Cálculo de la matriz inversa

En la representación se muestra primero la reducción adelante y atrás, luego se escalona y finalmente se normaliza.

```
def inversa(self, rep=0):
    """Inversa de Matrix"""
    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')

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

Cálculo del determinante

```
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

```
def diagonalizaS(self, espectro, rep=False, sust=[]):
    <<Texto de ayuda para la clase DiagonalizaS>>
    D            = Matrix(self.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
```

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

Encuentra una matriz diagonal semejante mediante trasformaciones 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.  """
```

```
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('los valores de la siguiente lista no son autovalores de la matriz:', 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')
```

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

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

```
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

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

Encuentra una matriz diagonal por conruencia empleando una matriz B
invertible (y entera si es posible) 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.

"""
```

```
def diagonalizaC(self, rep=False, sust=[]):
    <<Texto de ayuda para la clase DiagonalizaC>>
    <<Definición del método auxiliar BuscaPrimerNoNuloEnLaDiagonal>>

    D            = Matrix(self.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(1).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
```

```
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
```

```
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)
```

6.2.8. Diagonalización ortogonal de una matriz simétrica

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

```
"""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.

"""
```

```
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()
```

6.3. Representación de la clase `Matrix`

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

6.4. La clase `Matrix` completa

```
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>>
    <<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>>
```

```
12      <<Cálculo de la matriz inversa>>
13      <<Cálculo del determinante por eliminación>>
14      <<Diagonalizando por Semejanza en una matriz por bloques triangulares>>
15      <<Diagonalizando por Congruencia una matriz simétrica>>
16      <<Diagonalizando Ortogonalmente una matriz simétrica>>
17      <<Método de representación repr de la subclase Matrix>>
```

Capítulo 7

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 8

Las subclases `Elim`, `ElimG`, `ElimGJ`, `ElimGF`, `ElimGJF`, `InvMat`, `InvMatF` e `InvMatFC`

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

8.1. 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 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)
```

8.2. 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

        """
        <<Método que define los atributos .tex y .pasos y representa los pasos si se pide>>
        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
```

```
7
8           """
9           <<Método que define los atributos .tex y .pasos y representa los pasos si se pide>>
10
11          if not sistema.es_cuadrada():
12              raise ValueError('Matrix no cuadrada')
13
14          pasos = sistema.elim(26).elim(26).elim(14).pasos
15          A = T(pasos[0]) & I(sistema.n)
16          A.tex, A.pasos = texYpasos(sistema.concatena(I(sistema.n),1), pasos, rep, sust, repsust)
17          A.TrF = A.pasos[0]
18          A.TrC = A.pasos[1]
19          self.__dict__.update(A.__dict__)
20          self.__class__ = type(sistema)
```

```
1   class InvMatFC(Sistema):
2       def __init__(self, sistema, rep=0, sust=[], repsust=0):
3           """Devuelve la matriz inversa y los pasos dados sobre las filas y columnas
4
5              y evitando operar con fracciones.  Si rep es no nulo, se
6              muestran en Jupyter los pasos dados
7
8           """
9           <<Método que define los atributos .tex y .pasos y representa los pasos si se pide>>
10
11          if not sistema.es_cuadrada():
12              raise ValueError('Matrix no cuadrada')
13
14          pasos = sistema.elim(24).elim(10).pasos
15          A = sistema.copy()
16          nan = sympy.symbols('\ ')
17          dummyMatrix = M1(A.n)*nan
18          A.tex, A.pasos = texYpasos(A.concatena(I(A.n),1).apila(I(A.n).concatena(dummyMatrix,1),1), pasos, rep, sust, repsust)
19          A.TrF = A.pasos[0]
20          A.TrC = A.pasos[1]
21          A.F   = T(A.TrF) & I(A.n)
22          A.C   = I(A.n) & T(A.TrC)
23          A.lista = (A.C*A.F).lista
24          self.__dict__.update(A.__dict__)
25          self.__class__ = type(sistema)
```

Parte II

La clase `T` (transformación elemental)

Capítulo 9

La clase T

9.1. 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: $\mathsf{A}_{\underset{[(\lambda)i+j]}{\tau}}$ suma λ veces el elemento i al elemento j $(i \neq j)$;

Tipo II: $\mathsf{A}_{\underset{[(\lambda)i]}{\tau}}$ multiplica el elemento i por $\lambda \neq 0$.

Intercambio: $\mathsf{A}_{\underset{[i=j]}{\tau}}$ intercambia los elementos i y j.

Comentario sobre la notación.

Cualquier trasformación 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 $\underset{m\times n}{\mathbf{A}}$ 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 $\underset{m\times n}{\mathbf{A}}$ 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

$$\mathbf{A}_{\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k}: \qquad \mathbf{A}\texttt{ \& T(}\, t_1\texttt{) \& T(}\, t_2\texttt{) \&}\cdots\texttt{\& T(}\, t_k\texttt{)} = \mathbf{A}\texttt{ \& T([}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{])}$$

$$\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k\mathbf{A}: \qquad \texttt{T(}\, t_1\texttt{) \& T(}\, t_2\texttt{) \&}\cdots\texttt{\& T(}\, t_k\texttt{)}\ \texttt{\&}\ \mathbf{A} = \texttt{ T([}t_1\texttt{,}t_2\texttt{,}\ldots\texttt{,}t_k\texttt{])}\ \texttt{\&}\ \mathbf{A}.$$

[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 $\mathbf{A}\left(\mathbf{I}_{\tau}\right)$ o el producto $\left({}_{\tau}\mathbf{I}\right)\mathbf{A}$.

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

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

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.

9.2. Implementación de las transformaciones elementales

9.2.1. Texto de ayuda

```
1  """Clase para las transformaciones elementales
2
3  T ("Transformación elemental") guarda en su atributo 'abreviaturas'
4  una abreviatura (o una secuencia de abreviaturas) de transformaciones
5  elementales. El método __and__ actúa sobre otra T para crear una T que
6  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} )

>>> # Trasformación Tipo II (multiplica por 5 el segundo elemento)
>>> T( (5,2) )

>>> # Trasformación Tipo I (resta el tercer elemento al primero)
>>> T( (-1,3,1) )

>>> # Secuencia de las tres transformaciones anteriores
>>> T( [{1,2}, (5,2), (-1,3,1)] )

>>> # T de una T
>>> T( T( (5,2) ) )

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)] )

"""
```

9.2.2. Método de inicialización

```
def __init__(self, *args, rpr='v'):
    """Inicializa una transformación elemental"""
    <<Verificación de que las .abreviaturas corresponden a transformaciones elementales o intercambios>>

    def CreaListaAbreviaturas(arg):
        if isinstance(arg, (tuple, set)):
            verificacion(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.

```
def verificacion(abrv):
    if isinstance(abrv,tuple) and (len(abrv) == 2) and abrv[0]==0:
        raise ValueError('T( (0, i) ) no es una trasformación elemental')
    if isinstance(abrv,tuple) and (len(abrv) == 3) and (abrv[1] == abrv[2]):
        raise ValueError('T( (a, i, i) ) no es una trasformación elemental')
    if isinstance(abrv,set) and len(abrv)>2:
        raise ValueError ('El conjunto debe tener uno o dos índices para ser un intercambio')
```

9.3. Composición de Transformaciones o transformación de un `Sistema` por la izquierda

9.3.1. Texto de ayuda

```
"""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), {3, 1}] )
```

```
21
22  >>> # Transformación de las filas de una Matrix
23  >>> T( [{1,2}, (4,2)] ) & A # multiplica por 4 la segunda fila de A y
24                              # luego intercambia las dos primeras filas
25  """
```

9.3.2. Implementación

Describimos la composición de transformaciones `T(`t_1`) & T(`t_2`)` creando una lista de abreviaturas $[\boldsymbol{\tau}_1, \boldsymbol{\tau}_2]$ (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).

```
1  def __and__(self, other):
2      <<Texto de ayuda para la composición de Transformaciones Elementales>>
3      if isinstance(other, T):
4          return T(self.abreviaturas+other.abreviaturas, rpr=self.rpr)
5
6      if isinstance(other, Sistema):
7          return other.__rand__(self)
```

9.4. Transposición de transformaciones elementales

Puesto que $\mathsf{I}_{\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k} = (\mathsf{I}_{\boldsymbol{\tau}_1})\cdots(\mathsf{I}_{\boldsymbol{\tau}_k})$ y puesto que el producto de matrices es asociativo, deducimos que la transpuesta de $\mathsf{I}_{\boldsymbol{\tau}_1\boldsymbol{\tau}_2\cdots\boldsymbol{\tau}_k}$ es

$$(\mathsf{I}_{\boldsymbol{\tau}_1\boldsymbol{\tau}_2\cdots\boldsymbol{\tau}_k})^\mathsf{T} = \Big((\mathsf{I}_{\boldsymbol{\tau}_1})(\mathsf{I}_{\boldsymbol{\tau}_2})\cdots(\mathsf{I}_{\boldsymbol{\tau}_k})\Big)^\mathsf{T} = (\mathsf{I}_{\boldsymbol{\tau}_k})^\mathsf{T}\cdots(\mathsf{I}_{\boldsymbol{\tau}_2})^\mathsf{T}(\mathsf{I}_{\boldsymbol{\tau}_1})^\mathsf{T} = (_{\boldsymbol{\tau}_k}\mathsf{I})\cdots(_{\boldsymbol{\tau}_2}\mathsf{I})(_{\boldsymbol{\tau}_1}\mathsf{I}) = {}_{\boldsymbol{\tau}_k\cdots\boldsymbol{\tau}_2\boldsymbol{\tau}_1}\mathsf{I}$$

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:

$$(\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k)^\mathsf{T} = \boldsymbol{\tau}_k\cdots\boldsymbol{\tau}_1;$$

así

$$(\mathsf{A}_{\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k})^\mathsf{T} = {}_{(\boldsymbol{\tau}_1\cdots\boldsymbol{\tau}_k)^\mathsf{T}}(\mathsf{A}^\mathsf{T}) = {}_{\boldsymbol{\tau}_k\cdots\boldsymbol{\tau}_1}(\mathsf{A}^\mathsf{T})$$

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.

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

[2] Recuerde que la suma de listas (`list + list`) concatena las listas

9.5. Potencias e inversa de transformaciones elementales

Toda matriz de la forma $\mathbf{I}_{\tau_1\cdots\tau_k}$ ó ${}_{\tau_1\cdots\tau_k}\mathbf{I}$ es invertible por ser producto de matrices elementales:

$$(\mathbf{I}_{\tau_1\cdots\tau_k})^{\mathsf{T}}(\mathbf{I}_{\tau_k^{-1}\cdots\tau_1^{-1}})^{\mathsf{T}} = \mathbf{I}_{\tau_1\cdots\tau_k\cdot\tau_k^{-1}\cdots\tau_1^{-1}} = \mathbf{I};$$

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

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

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 $\underset{[\mathbf{1}\rightleftharpoons\mathbf{1}]}{\boldsymbol{\tau}}$.

```
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)
```

```
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)
```

9.6. 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]

```
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)
```

[3] Al no encontrar ningún nombre en los manuales de Álgebra Lineal para este concepto, he adoptado este descriptivo nombre.

9.7. Sustitución de variables simbólicas

Sustituye la variable `c` por `v`, donde `v` puede ser un valor, u otra variable simbólica.

```
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
```

9.8. Métodos similares a los de una `list`

9.8.1. 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)
```

9.8.2. Igualdad entre transformaciones elementales

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

9.9. 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) )`

9.9.1. 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 trasformación 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.

```
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 trasformación 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}}'
```

9.10. La clase `T` completa

```
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 10

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 $\mathbb{R}^n$.

10.1. La clase SubEspacio (de $\mathbb{R}^m$)

La clase `SubEspacio` se puede instanciar tanto con un `Sistema` de `Vectores` como con una `Matrix`.

En el primer caso, dado un `Sistema` de vectores, por ejemplo

$$S = \left[\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} ; \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix} ; \begin{pmatrix} 2 \\ 10 \\ 3 \end{pmatrix} \right],$$

`SubEspacio( S )` corresponde al conjunto de combinaciones lineales de los `Vectores` de dicho `Sistema`, representado por las siguientes ecuaciones *paramétricas*:

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^2,\ \boldsymbol{v} = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 3 \end{bmatrix} \boldsymbol{p} \right\}$$

donde el vector $\boldsymbol{p}$ es el vector de parámetros. En el segundo caso, dada una `Matrix`, por ejemplo

$$\mathbf{M} = \begin{bmatrix} -3 & 0 & 2 \\ 6 & 0 & -4 \end{bmatrix},$$

`SubEspacio( M )` corresponde al conjunto de `Vectores` que son solución al sistema de ecuaciones $\mathbf{M}\boldsymbol{v} = \mathbf{0}$; y que se puede representar con el sistema de ecuaciones *cartesianas*:

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} -3 & 0 & 2 \end{bmatrix} \boldsymbol{v} = \mathbf{0} \right\}$$

En ambos ejemplos corresponden al mismo subespacio de $\mathbb{R}^3$; y, de hecho, la librería muestra ambos tipos de representación para cada `SubEspacio`: las ecuaciones paramétricas a la izquierda y las cartesianas a la derecha.

$$\left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \exists \boldsymbol{p} \in \mathbb{R}^2,\ \boldsymbol{v} = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 3 \end{bmatrix} \boldsymbol{p} \right\} = \left\{ \boldsymbol{v} \in \mathbb{R}^3 \;\middle|\; \begin{bmatrix} -3 & 0 & 2 \end{bmatrix} \boldsymbol{v} = \mathbf{0} \right\}$$

`SubEspacio` tiene varios atributos.

- `dim`: dimensión del subespacio. En el ejemplo `dim=2`.
- `Rn`: indica el espacio vectorial $\mathbb{R}^n$ al que pertenece `SubEspacio(S)`. En el ejemplo anterior `Rn=3` puesto que es un subespacio de $\mathbb{R}^3$.
- `base`: una base del subespacio (un `Sistema` de `Vectores` de `Rn`). Cuando `dim==0` base es un `Sistema` vacío.
- `sgen`: Un `Sistema` de `Vectores` generador del subespacio. En particular será el sistema de vectores correspondiente a la `Matrix` de coeficientes empleada en la representación con ecuaciones paramétricas. En el ejemplo Cuando `dim==0`, `sgen` contiene un vector nulo de `Rm` componentes.
$$\left[\begin{pmatrix}0\\1\\0\end{pmatrix};\ \begin{pmatrix}2\\0\\3\end{pmatrix};\right]$$
- `cart`: `Matrix` de coeficientes empleada en la representación con las ecuaciones cartesianas. En el ejemplo
$$\begin{bmatrix}-3 & 0 & 2\end{bmatrix}.$$

10.2. 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`.

```
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:
            A = BlockM([data]).subs(sust)
        self.base = Sistema([data.K()|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.

```
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()])
```

```
def vector_nulo(self):
    return M0(self.Rn[0],self.Rn[1]) if isinstance(self.Rn, tuple) else V0(self.Rn)
```

10.3. 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):
        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().base)
        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 isinstance(self.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.cart*other == V0(self.cart.m)

    elif isinstance(self.sgen|1, Matrix):
        if not isinstance(other, Matrix) or (other.n,other.m) != self.Rn:
            raise ValueError\
                ('El Vector no tiene el número adecuado de componentes')
        return self.cart*other.vec() == V0(self.cart.m)
```

10.4. Métodos de representación de la clase SubEspacio

```
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()
```

10.5. La clase SubEspacio completa

```
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 11

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*.

11.1. 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.

```
1  def __init__(self, data, v, vi=0):
2      """Inicializa un Espacio Afín de Rn"""
3      self.S  = data if isinstance(data, SubEspacio) else SubEspacio(data)
4      self.Rn = self.S.Rn
5
6      if isinstance(self.Rn, int):
7          if not isinstance(v, Vector) or v.n != self.Rn:
8              raise ValueError('v y SubEspacio deben estar en el mismo espacio vectorial')
9
10     elif isinstance(self.Rn, tuple):
11         if not isinstance(v, Matrix) or (v.m,v.n) != self.Rn:
12             raise ValueError('v y SubEspacio deben estar en el mismo espacio vectorial')
13
14     self.v  = v if vi else (self.S.sgen.concatena(Sistema([v]))).elim(1)|0
```

11.2. 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.n,other.m) != 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) )
```

```
16
17      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)]))
```

11.3. Métodos de representación de la clase `EAfin`

```
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"""
    return self.EcParametricas() + '\\; = \\;' + self.EcCartesianas()
```

11.4. La clase `EAfin` completa

```
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 12

Resolución de sistemas de ecuaciones homogéneos. La clase `Homogenea`

El siguiente código devuelve el conjunto de soluciones de un sistema homogéneo $\mathbf{A}\boldsymbol{x} = \mathbf{0}$. Descripción de los atributos:

- `sgen` es un sistema generador del espacio nulo $\mathcal{N}(\mathbf{A})$.
- `determinado` indica si es cierto que el sistema es determinado (una única solución)
- `tex` es la cadena de texto LaTeXque representa los pasos dados para resolver el sistema.

12.1. Implementación de la clase `Homogenea`

```
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:
            A = Matrix(sistema).subs(sust)
        except:
            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)
```

```
30
31          if rep:
32              display(Math(self.tex))
33
34      <<Métodos de representación de la clase ~Homogenea~>>
```

La base la constituyen los vectores `v` de `E` que corresponden a los vectores nulos de `K`:

12.2. Métodos de representación de la clase `Homogenea`

```
1   def __repr__(self):
2       """Muestra el Espacio Nulo de una matriz en su representación Python"""
3       return 'Combinaciones lineales de (' + repr(self.sgen) + ')'
4
5   def _repr_html_(self):
6       """Construye la representación para el entorno Jupyter Notebook"""
7       return html(self.latex())
8
9   def latex(self):
10      """ Construye el comando LaTeX para la solución de un Sistema Homogéneo"""
11      if self.determinado:
12          return '\\left\\{\ ' + latex(self.sgen|1) + '\ \\right\\}'
13      else:
14          return '\\mathcal{L}\\left(\ ' + latex(self.sgen) + '\ \\right)'
```

Capítulo 13

Resolución de sistemas de ecuaciones lineales. La clase SEL

13.1. Implementación

13.1.1. Texto de ayuda

```
"""Resuelve un Sistema de Ecuaciones Lineales

mediante eliminación con el sistema ampliado y muestra los pasos
dados

"""
```

```
class SEL:
    def __init__(self, sistema, b, rep=0, sust=[], repsust=0):
        <<Texto de ayuda de la clase ~SEL~>>
        try:
            A = Matrix(sistema.amplia(-b)).subs(sust).ccol({sistema.n})
        except:
            A = BlockM([sistema.amplia(-b)]).subs(sust).ccol({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>>

    <<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.

```
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.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))
```

13.2. Métodos de representación de la clase SEL

```
def __repr__(self):
    """Muestra el Espacio Nulo de una matriz en su representación Python"""
    return repr(self.solP) + ' + Combinaciones lineales de (' + 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 and self.solP:
        return '\\left\\{\ ' + latex(self.solP) + '\ \\right\\}'
    else:
        return self.eafin.EcParametricas() if self.solP else '\\emptyset' #latex(set())
```

Parte V

La clase `Determinante`. Cálculo del determinante por eliminación Gaussiana

13.2.1. Método de inicialización

```
class Determinante:
   """Determinante de una Matrix mediante eliminación Gaussiana por columnas

   La representación muestra los pasos dados

   """
   def __init__(self, data, disp=0, sust=[]):
      <<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>>
```

```
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, 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]
```

```
def __repr__(self):
   """Muestra un Sistema en su representación Python"""
   return 'Valor del 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)
```

Parte VI

Las clases `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):
        <<Texto de ayuda para la clase DiagonalizaS>>
        <<Método que define los atributos .tex y .pasos y representa los pasos si se pide>>
        A = Matrix(sistema)
        D = A.diagonalizaS(espectro)

        if rep:
            nan   = sympy.symbols('\ ')
            dummyMatrix = M1(A.n)*nan
            if repType==1:
                nan   = sympy.symbols('\ ')
                D.tex = rprElimCF(A.apila(I(A.n),1), D.pasos)
            elif repType==2:
                nan   = sympy.symbols('\ ')
                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('\ ')
                D.tex = rprElimFyC(A.apila(I(A.n),1), D.pasos)
            elif repType==4:
                nan   = sympy.symbols('\ ')
                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)
```

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>>
<<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>>
<<Cribado de secuencias de transformaciones>>
<<Representación de un proceso de eliminación rprElim>>
<<Representación de un proceso de eliminación rprElimFyC>>
<<Representación de un proceso de eliminación rprElimCF>>
<<Representación de un proceso de eliminación dispElim, dispElimFyC y dispElimCF>>

<<Definición de la clase Sistema>>
<<Consideracion de qué es una secuencia>>
<<Definición de la subclase BlockV>>
<<Definición de la subclase Vector>>
<<Definición de las subclases V0 y V1>>
<<Definición de la subclase BlockM>>
<<Definición de la subclase Matrix>>
<<Definición de las subclases M0, M1 e I>>

<<Definición de la clase T>>
<<Definición de las subclases ElimX>>
<<Definición de las subclases InvMatX>>

<<Definición de la clase ~SubEspacio~>>
<<Definición de la clase ~EAfin~>>

<<Resolución de un sistema de ecuaciones homogéneo>>
<<Resolución de un Sistema de Ecuaciones Lineales>>
```

```
40
41  <<Definición de la clase Determinante>>
42  <<Definición de las subclases DiagonalizaX>>
```

Capítulo 14

Métodos generales

14.1. 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.

```
NumberTypes = (int, float, complex, sympy.Basic)
es_numero   = lambda x: isinstance(x, NumberTypes) and not isinstance(x, bool)
```

14.2. Números racionales

```
def fracc(a,b):
    """Transforma la fracción a/b en un número racional si ello es posible"""
    if all( [ isinstance(i, (int, float, 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
```

14.3. 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`.

```
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
```

14.4. 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.

```
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

$$\left[\ \tfrac{1}{2};\quad 2;\quad \sqrt{2};\ \right]$$

```
def latex(a):
    """Método latex general"""
    try:
        return a.latex()
    except:
        return sympy.latex(a)
```

Para visualizar un objeto en su formato LaTeX definimos el método `pinta`.

```
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) ]) )
```

Vector([sqrt(2), 0, 3/4])

14.5. Métodos `CreaLista` y `CreaSistema`

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

```
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))
```

14.6. 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):
    """Primer 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
```

14.7. 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) ]
```

14.8. Cribado de secuencias de transformaciones

En el proceso de eliminación, muchas trasformaciones 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.

```
1  def filtradopasos(pasos):
2      abv = pasos.abreviaturas if isinstance(pasos,T) else pasos
3
4      p = [T([j for j in T([abv[i]]).abreviaturas if (isinstance(j,set) and len(j)>1)\
5                  or (isinstance(j,tuple) and len(j)==3 and j[0]!=0)       \
6                  or (isinstance(j,tuple) and len(j)==2 and j[0]!=1) ])    \
7                                              for i in range(0,len(abv)) ]
8
9      abv = [ t for t in p if t.abreviaturas] # quitamos abreviaturas vacías de la lista
10
11     return T(abv) if isinstance(pasos,T) else abv
```