Laboratorio de Matemáticas

Release 2010/2011

Pablo Angulo Ardoy
Rafael Hernández García
Patricio Cifuentes Muñiz
Bernardo López Melero
Daniel Ortega Rodrigo

September 27, 2011

Índice general

Laboratorio de Matemáticas con Sage

1.1 Introducción

Ya empezado el curso 07/08, en el departamento surgió la idea de incorporar a los, por entonces nacientes, estudios de Grado de Matemáticas, un asignatura dedicada a cualificar a nuestros estudiantes con una nueva habilidad: usar el ordenador para resolver, o al menos ilustrar, los problemas.

La asignatura de **laboratorio de matemáticas** pretende enseñar a usar el ordenador como herramienta para aprender y para experimentar en matemáticas. Para ello usamos el programa libre Sage, que permite acceder a una vasta colección de librerías matemáticas usando el lenguaje python, una elección popular para aprender a programar. La asignatura se imparte a lo largo de todo un año, e intenta, aún tímidamente, coordinarse con las otras asignaturas del curso para usar cuando sea posible ejemplos extraídos de aquellas.

En la **Universidad Autónoma de Madrid**, el laboratorio de matemáticas se imparte a dos grupos de alumnos muy distintos: alumnos de primero del grado en matemáticas, que en su mayoría no saben programar, y alumnos de segundo del doble grado en matemáticas e informática, que aprendieron a programar en C en primer curso, y que seguirán aprendiendo mucho más sobre programación. Estas notas corresponden al segundo grupo de alumnos, y por ello contienen temas avanzados relacionados con la programación que no son en absoluto imprescindibles para el objetivo de la asignatura.

1.2 Créditos

- Pablo Angulo Ardoy
- Patricio Cifuentes Muñiz
- Rafael Hernández García
- Bernardo López Melero
- Daniel Ortega Rodrigo

Dedicado a los alumnos de doble grado que cursaron laboratorio en el curso 2010/2011.

1.3 Agradecimientos

Hristo Inouzhe preparó varios vídeos didácticos sobre el material de estas notas (puedes encontrar los links más abajo). La Universidad Autónoma de Madrid financió un proyecto de innovación docente que nos permitió regalar dvds live con Sage a nuestras alumnas. Luis Guijarro y María Calle probaron la parte de cálculo II en las clases de esta asignatura. Pablo Fernández Gallardo nos prestó material y comentarios para las secciones de combinatoria y probabilidad. La Wikipedia (en inglés) nos ahorró muchas horas de búsqueda bibliográfica (y nos prestó algunas imágenes).

Y la comunidad de Sage resolvió nuestras dudas y resolvió muy pero que muy rápido algunos problemas que encontramos durante el curso.

1.4 Vídeos didácticos para este material

- Optimización con cython: el conjunto de Mandelbrot (bloque **2**, Eficiencia en cálculo científico), Pablo Angulo: parte 1, parte 2
- Método de montecarlo vs fuerza bruta (bloque **4**, Experimentos con numeros aleatorios), Hristo Inouzhe.
- Coloracion y edicion de grafos con SAGE (bloque **4** Grafos), Hristo Inouzhe.
- Puntos críticos por el metodo de lagrange (bloque **5** Cálculo vectorial), Hristo Inouzhe.
- Haz de conicas con SAGE (bloque **5** Curvas planas), Hristo Inouzhe.
- Ajuste de Modelos utilizando SAGE (bloque **6**, Regresión y ajuste de modelos), Hristo Inouzhe.

1.4.1 Otros vídeos didácticos de Hristo Inouzhe

- Invertir una matriz por Gauss-Jordan
- Simulación de los modos de vibracion del CO2

1.5 Código fuente

Este documento ha sido generado a partir de archivos rst (ReStructuredText), que a su vez fueron generados a partir de archivos sws (hojas de trabajo de Sage). Los mismos archivos rst dieron lugar a documentación en formatos html y pdf (via latex).

Tanto los archivos rst como los archivos sws se distribuyen con la misma licencia que este documento. Puedes encontrar todos esos documentos en la web oficial de la asignatura (a día de hoy, está en el sitio web de Pablo Angulo en la UAM).

1.6 Licencia

1.6.1 Imágenes prestadas

Sigue una lista de imágenes usadas en las notas y tomadas de la wikipedia. Todas las imágenes se distribuyen con una licencia compatible con GFDL y cc-by-sa, pero una de ellas no tiene una licencia compatible con cc-by-nc.

Si quieres usar esas imágenes, puedes seguir el link de abajo para leer su licencia.

- la tabla hash en b2s1:
- Caminos en una malla en Bloque IV: Contar y enumerar
- Caminos monótonos en Bloque IV: Contar y enumerar

CAPÍTULO 2

Bloque I: Introducción

Se presenta el entorno **Sage**, los tipos de datos y las estructuras de control de python, poniendo ejemplos con contenido matemático siempre que sea posible. En la última sesión se presentan formas de trabajar en python que no son posibles en lenguajes como C.

2.1 Tipos de datos en Sage

2.1.1 Introducción a Sage

En esta asignatura usaremos el programa **Sage** para resolver distintos problemas de matemáticas con el ordenador. El programa es libre, lo que nos permite copiarlo, modificarlo y redistribuirlo libremente. Sage consta de un buen número de librerías para ejecutar cálculos matemáticos y para generar gráficas. Para llamar a estas librerías se usa el lenguaje de programación **python**, para el que existen un buen número de recursos didácticos disponibles.

Python es un lenguaje de propósito general de muy alto nivel, que permite representar conceptos abstractos de forma natural y, en general, hacer más con menos código. Buena parte de las librerías que componen Sage se pueden usar directamente desde python, sin necesidad de acarrear todo el entorno de Sage.

Existen varias formas de interactuar con Sage: desde la consola, desde ciertos programas como TeXmacs o Cantor, y desde el *navegador de internet*, como estamos haciendo ahora. Para ello, Sage crea un *servidor web* que escucha las peticiones del cliente (un navegador), realiza los cálculos que le pide el cliente, y le devuelve los resultados. En esta asignatura sólo usaremos el interfaz web.

Cuadros de texto

Los cuadros de texto como éste permiten incluir comentarios en una hoja de trabajo. Si haces doble clic sobre el cuadro de texto puedes editar el contenido. Al entrar en modo de edición, aparecen botones y desplegables para cambiar algunos aspectos del estilo del texto. Lo más importante para mantener un orden en la hoja de trabajo es el primer desplegable, que permite elegir si se trata de un párrafo, un encabezado.

Cuadros de código

Los cuadros de código son rectangulares y su borde cambia de color al seleccionarlos. Dentro de los cuadros de código podemos escribir instrucciones que serán ejecutadas al pulsar el botón `evaluate` o teclear *mayúsculas+Enter*.

Puedes crear un nuevo cuadro de código pulsando sobre la línea azul que aparece al poner el cursor sobre un cuadro de código existente, o pulsando *control+Enter* dentro de un cuadro de código. Puedes crear un nuevo bloque de texto pulsando sobre la misma línea azul, pero manteniendo pulsada la tecla de mayúsculas.

```
sage: print 'Hola, Mundo'
Hola, Mundo
```

Si ponemos varias líneas de código, se ejecutan una tras otra de arriba a abajo. El **intérprete de instrucciones** lee las instrucciones, las interpreta y ejecuta lo que se le pide. Si al ejecutar las instrucciones se produce una *salida,* se muestra debajo del cuadro de código.

```
sage: #Los comentarios en python comienzan con #
sage: print 'Hola'   #El comando print muestra sus argumentos en la salida del programa
sage: print 'Mundo'
Hola
Mundo
```

2.1.2 Operaciones

Uso como calculadora

Podemos usar los bloques de comandos como una simple calculadora, escribiendo operaciones elementales y observando el resultado debajo del cuadro. Podemos introducir números con decimales usando el punto decimal. Los paréntesis marcan qué comando se ejecuta antes, naturalmente.

```
sage: (1*2)+(3*4)+5
19

sage: 1*(2+3)*(4+5)
45

sage: #Un numero racional
sage: 1/2
1/2

sage: #Un número de coma flotante
sage: 1+1.0
2.00000000000000
```

Reglas de precedencia

En ausencia de paréntesis, Sage decide qué operaciones se ejecutan antes y cuáles después usando unas *reglas de precedencia de operadores* bastante estándar. En la siguiente tabla de operadores, cada uno se ejecutará antes que los que están por encima.

Los operadores con la misma precedencia se agrupan de izquierda a derecha, excepto la exponenciación, que agrupa de derecha a izquierda.

No necesitas conocer todos estos operadores por ahora:

operador	Descripción
or	O booleano
and	Y booleano
not x	NO booleano
in , not in , is , is not , < , <= , > , >= , <> , != , ==	comparaciones, comprobación de pertenencia y de tipo
|	O bit a bit
&	Y bit a bit
+ , -	Suma y resta
* , / , / / , %	multiplicación, división, el resto
+ x , -x , ~ x	positivos, negativos, NO bit a bit
^	exponenciación
x [indice] , x [indice: indice] , x (arguments. ..) , x.attribute	Índices, rebanadas, llamada a funciones, referencia a atributos
(expresiones ...) , [expresiones ...] , {clave: dato ...}	tuplas, listas, diccionarios, evaluación de expresiones

Ejercicio. Intenta predecir el resultado de ejecutar las instrucciones de debajo antes de pulsar el botón de evaluar.

```
sage: 2*3^1+1*3^2

sage: 2*3^((1+1)*3)^2

sage: True or True and False

sage: 1==2-1

sage: 2 < 3 and not 1 == 2
```

Llamadas a funciones

Además de las operaciones aritméticas, podemos usar las muchas funciones disponibles. La forma de usarlas es escribir el nombre de la función, seguido del argumento, o argumentos, entre paréntesis, y separados por comas.

```
sage: sin(pi/3)
1/2*sqrt(3)

sage: (1 + sin(pi/3))/2
1/4*sqrt(3) + 1/2

sage: max(2,3)
3
```

2.1.3 Variables

Para guardar un valor dentro de una variable, usamos la sintaxis:

```
variable = expresion
```

Por ejemplo,

```
numero = 1 + 2 + 3 + 4 + 5 + 6
largo = 2.56*20
angulo = pi/3
```

Para poder ver el valor de una variable podemos usar el comando `print` :

```
print numero
print largo, angulo
print largo * sin(angulo)
```

Una vez definida una variable, podemos hacer cálculos con su valor:

```
masa = 3
aceleracion = 10
fuerza = masa\*aceleracion
print fuerza
```

```
sage: numero = 1+2+3+4+5+6
sage: largo = 2.56*20
sage: angulo = pi/3

sage: print numero
sage: print largo, angulo
sage: print largo * sin(angulo)
21
51.2000000000000 1/3*pi
25.6000000000000*sqrt(3)
```

Nota: no confundas la asignación (=) con el operador de comparación de igualdad (==).

```
sage: velocidad = 3

sage: velocidad == 3
True
```

Nota : los nombres de las variables deben empezar por una letra o un guión bajo (_), pueden contener números, y son distintos si se usan mayúsculas o minúsculas. *Las variables* `numero` *y* `Numero` *son distintas* .

```
sage: print Numero
Traceback (most recent call last):
...
NameError: name 'Numero' is not defined
```

Liberar una variable

Usando el comando `del` , podemos liberar una variable, y a partir de ese punto el nombre de la variable deja de estar definido.

```
del variable
```

```
sage: numero = 12
sage: print 2*numero
24

sage: del numero
sage: print 3*numero
Traceback (most recent call last):
...
NameError: name 'numero' is not defined
```

Al usar `del` solamente liberamos la referencia a un dato en la memoria, pero no el dato en sí. Otras referencias a ese dato siguen siendo válidas, y no se corrompen por el uso de `del` . Por tanto, esta instrucción no tiene nada que ver con las reservas de memoria en lenguajes de bajo nivel como C, que reservan y liberan espacio en la memoria.

Para liberar la memoria no usada, Python usa un colector de basura . Este colector de basura identifica los objetos a los que no apunta ninguna referencia y los libera.

```
sage: lista1 = [1,2,3,4]
sage: lista2 = lista1
sage: del lista1
sage: print lista2
[1, 2, 3, 4]
```

2.1.4 Tipos de datos

En las variables podemos almacenar cualquier tipo de datos que resulte de evaluar una expresión. Más adelante en el curso guardaremos en las variables matrices, gráficas e incluso objetos abstractos como espacios vectoriales. Por ahora hemos usado los siguientes tipos de datos:

- Booleanos: sólo toman el valor `True` o `False` .
- Enteros: cualquier número entero, positivo o negativo, de longitud arbitraria. Ej.: 1, 10, -30
- Racionales Ej.: 1/2, -3/4
- Números de coma flotante: un número con unos cuantos dígitos decimales y un exponente, que representa un número real de forma aproximada. Ej.: 1.25, -1.5e6
- Expresiones simbólicas: expresiones matemáticas que representan números reales de forma exacta. Ej.: pi/4 ($\pi/4$), (1+sqrt(2))/2 ($\frac{1+\sqrt{2}}{2}$)

Las variables pueden almacenar referencias a datos de cualquier tipo: python usa **tipado dinámico** . Sin embargo, todos los datos tienen necesariamente un tipo

```
sage: 2 >=3
False

sage: var1=2
sage: var2=3
sage: #La variable es_menor almacena un booleano
sage: es_menor = var1 < var2

sage: print es_menor
True

sage: factorial(1000)
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800

sage: numero = factorial(1000)

sage: numero/factorial(1001)
1/1001
```

Números de coma flotante y expresiones simbólicas

Al usar el ordenador para hacer matemáticas es importante saber si los datos del ordenador representan los objetos matemáticos de forma exacta.

Es imposible almacenar en un ordenador con una cantidad finita de memoria todas las cifras decimales del número **pi** . Una alternativa es almacenar sólo unas cuantas cifras, y cometer por tanto un pequeño error. Para la mayoría de las aplicaciones es más que suficiente usar 10 o 20 *cifras decimales significativas* . Con esto queremos decir que al

escribir el número en notación exponencial, descartamos todas las cifras a partir de la número 10 o 20. Por ejemplo, el número $1/\pi$ con diez dígitos significativos:

$$3{,}183098861 \cdot 10^{-1}$$

y la constante de Planck con seis dígitos significativos:

$$6,62606 \cdot 10^{-34}$$

Al hacer operaciones con números que contienen errores, los errores se suman y multiplican, y pueden acabar estropeando un cálculo. Otra alternativa es usar una variable simbólica, y usar las reglas aritméticas sin hacer cálculos con decimales, exactamente igual que os contaban en el instituto:

$$\frac{\sqrt[3]{2}\cdot 3^2}{\frac{3\cdot 4}{\sqrt[3]{4}}} = 2^{1/3-2+2/3}3^{2-1} = 2^{-1}3 = 3/2$$

Al estar orientado preferentemente al público matemático, Sage prefiere usar expresiones simbólicas exactas antes que aproximaciones numéricas. Como vimos antes, para obtener una representación decimal de una expresión simbólica, podemos usar el comando `n()` (n de numérico).

```
sage: 1/pi
1/pi

sage: n(1/pi)
0.318309886183791

sage: a = sqrt(2)

sage: n(a)
1.41421356237310
```

Métodos específicos de cada tipo de datos

Cada tipo de datos tiene sus propios métodos: funciones que se aplican sólo a datos de este tipo y que se llaman escribiendo primero la variable que contiene el dato, después un punto (`.`), y después el método:

```
variable.metodo()
```

Por ejemplo, podemos calcular la factorización de un número entero, pero no de un número real, o podríamos intentar simplificar una expresión simbólica, pero no podemos simplificar un número entero.

```
sage: a = 12
sage: print a.factor()                #Factorizacion del entero 'a'
2^2 * 3

sage: b = 4.7
sage: print b.integer_part()            #Parte entera del numero real 'b'
4

sage: c = (3*sqrt(6)+sqrt(2))/sqrt(8)
sage: print c.full_simplify()            #Intenta simplificar la expresion 'c'
3/2*sqrt(3) + 1/2

sage: print b.factor()
Traceback (most recent call last):
...
AttributeError: 'sage.rings.real_mpfr.RealLiteral' object has no attribute 'factor'
```

```
sage: print a.full_simplify()
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'full_simplify'
```

[Tabulador]

El tabulador permite conocer la lista completa de métodos aplicables a un dato:

- Escribe el nombre de una variable seguido de un punto, y pulsa [Tabulador] para ver todos los métodos específicos al tipo de datos de la variable.
- Escribe el nombre de una variable seguido de un punto y algunos caracteres, y pulsa [Tabulador] para ver sólo los métodos que comienzan con esos caracteres.

```
sage: a.

sage: b.
```

Números de Sage y de python

Aunque Sage utiliza el lenguaje python, los tipos numéricos en Sage no corresponden exactamente a los tipos numéricos de python, ya que los números en Sage tienen más funcionalidad. Por ejemplo, los enteros de Sage permiten calcular su lista de divisores, su expresión binaria, etcétera. Por defecto, los tipos numéricos son enteros, racionales y números reales de Sage, no de python.

```
sage: a = 12          #Entero de SAGE
sage: #a = Integer(12) #Otra forma equivalente de definir un entero de SAGE
sage: b = int(12)     #Entero de 32 bits de python

sage: print a.divisors(), a.digits(base = 2)
[1, 2, 3, 4, 6, 12] [0, 0, 1, 1]

sage: print b.divisors()
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'divisors'

sage: c = 1.2               #Real de SAGE (de precision limitada)
sage: c = RealNumber(1.2)   #Otra forma equivalente de definir un real de SAGE
sage: d = float(1.2)        #Real de 64 bits de python

sage: d.exact_rational()
Traceback (most recent call last):
...
AttributeError: 'float' object has no attribute 'exact_rational'

sage: c.exact_rational()
5404319552844595/4503599627370496
```

2.1.5 Secuencias de datos

Cadenas de caracteres

Las **cadenas de caracteres** son *secuencias de caracteres* (letras, números, puntuación...) que se manipulan de forma conjunta y se pueden almacenar en una variable. Para introducir cadenas de caracteres en el código, *separamos el texto entre comillas* simples ('cadena'), comillas dobles ("cadena"), o triples comillas simples para cadenas más largas (''' cadena larga ''').

```
sage: print 'Hola'
Hola
```

Se pueden usar algunas **operaciones** con cadenas de caracteres:

- La suma (+) de cadenas devuelve una cadena resultado de poner la primera cadena después de la segunda
- El producto (*) de una cadena por un número natural repite la cadena tantas veces como indica el número.

```
sage: print 'Hola' + 'tu'
sage: print 'Hola'*2
Holatu
HolaHola
```

```
sage: cadena1 = 'Hola'
sage: cadena2 = ' a todas'                  #presta atencion al espacio en blanco al principio
sage: cadena3 = cadena1 + cadena2
sage: cadena4 = cadena1 + '.'*3 + cadena2
```

```
sage: print cadena1
sage: print cadena2
sage: print cadena3
sage: print cadena4
Hola
 a todas
Hola a todas
Hola... a todas
```

Podemos obtener la longitud de una cadena con el comando `len` .

```
sage: print len(cadena1), len(cadena2), len(cadena3), len(cadena4)
4 8 12 15
```

Extraer caracteres y subcadenas

Para acceder al carácter que ocupa la posición j-ésima en una cadena de caracteres, usamos la notación

```
cadena[j]
```

¡Ojo! Se empieza a contar desde el número 0, el índice del primer carácter. El índice del último carácter es L-1, donde L es la longitud de la cadena.

También podemos acceder a una subcadena (o rango), desde el índice j (inclusive) hasta el k (exclusive), con la notación

```
cadena[j:k]
```

Por ejemplo:

```
cadena = 'Si miras al abismo...'
print cadena[0], cadena[1], cadena[2], cadena[3]
print cadena[3:8]
```

```
sage: cadena = 'Si miras al abismo...'
sage: print cadena[0], cadena[1], cadena[2], cadena[3]
sage: print cadena[3:10]
S i   m
miras a
```

Tuplas

Las tuplas contienen unos cuantos elementos no necesariamente del mismo tipo. Basta con poner las variables entre paréntesis separadas por comas para formar una tupla.

tupla = (elemento1, elemento2)

Una vez creada, podemos acceder a sus elementos usando corchetes, igual que hacíamos con las cadenas de caracteres.

```
sage: frutas = ('pera','manzana','naranja')
sage: primos = (2,3,5,7,11,13)
sage: print frutas[0]
sage: print primos[0:3]
sage: primo = primos[2]
sage: fruta = frutas[2]
sage: print primo, fruta
pera
(2, 3, 5)
5 naranja
```

Las operaciones + y * actúan sobre tuplas de la misma forma que sobre cadenas de caracteres.

```
sage: (1,2,'a') + (3,4)
(1, 2, 'a', 3, 4)

sage: (1, 'a')*3
(1, 'a', 1, 'a', 1, 'a')
```

Si queremos guardar cada elemento de la tupla en una variable distinta, podemos usar el acceso a los elementos usando los corchetes:

```
frutas = ('pera','manzana','naranja')
a=frutas[0]
b=frutas[1]
c=frutas[2]
```

o también podemos *desempaquetar* la tupla con una sóla instrucción:

```
a,b,c = frutas
```

lo que, al igual que el código anterior, guarda en la variable `a` el primer elemento de la tupla, en la variable `b` el segundo y en la variable `c` el tercero. Si intentamos desempaquetar una tupla de N elementos con más, o con menos de N variables, obtendremos un error.

```
sage: frutas = ('pera','manzana','naranja')
sage: a,b,c = frutas
```

```
sage: print c + ',' + b + ',' + a
naranja,manzana,pera

sage: a,b = frutas
Traceback (most recent call last):
...
ValueError: too many values to unpack

sage: a,b,c,d = frutas
Traceback (most recent call last):
...
ValueError: need more than 3 values to unpack
```

Las tuplas son **immutables** , es decir, que no se puede quitar ni añadir elementos a una tupla una vez ha sido creada, ni siquiera sustituir un elemento por otro.

Listas

Las listas se usan de modo similar a las tuplas, pero se pueden quitar y añadir elementos en cualquier momento. Decimos que son *contenedores dinámicos de datos* .

La sintaxis para crear listas es igual a la de las tuplas, pero usando corchetes en vez de paréntesis. Una vez creadas, podemos acceder a sus elementos usando los corchetes, pero además podemos asignar nuevos valores a posiciones arbitrarias de la lista.

```
sage: lista_frutas = ['pera','manzana','naranja']
sage: print lista_frutas
sage: lista_frutas[1] = 'fresa'
sage: print lista_frutas
['pera', 'manzana', 'naranja']
['pera', 'fresa', 'naranja']
```

También podemos eliminar elementos de la lista con el comando `del` y añadir elementos al final de la lista con el comando `append` .

```
sage: print lista_frutas[1]
fresa

sage: lista_frutas = ['pera','manzana','naranja']
sage: print lista_frutas
sage: lista_frutas.append('fresa')
sage: print lista_frutas
sage: del lista_frutas[1]
sage: print lista_frutas
['pera', 'manzana', 'naranja']
['pera', 'manzana', 'naranja', 'fresa']
['pera', 'naranja', 'fresa']

sage: print lista_frutas
sage: lista_frutas.insert(2,'otra fruta')
sage: print lista_frutas
['pera', 'naranja', 'fresa']
['pera', 'naranja', 'otra fruta', 'fresa']
```

El comando `srange` permite crear listas de números de SAGE (en python se usa la función `range` , que devuelve enteros de python):

- `srange(j,k,d)` : devuelve los números entre `j` (inclusive) y `k` (exclusive), pero contando de `d` en `d` elementos. A pesar de que el uso más extendido es con números enteros, los números `j`, `k` y `d` pueden ser enteros o no.

Abreviaturas:

- `srange(k)` : devuelve `srange(0,k,1)`. Si, en particular, `k` es un natural, devuelve los naturales entre `0` (inclusive) y `k` (exclusive); y si `k` es negativo, devuelve una lista vacía.
- `srange(j,k)` : devuelve la lista `srange(j,k,1)`. Si, en particular, `j` y `k` son enteros, devuelve los enteros entre `j` (inclusive) hasta el anterior a `k`.

```
sage: print srange(10)
sage: print srange(10,20)
sage: print srange(10,20,2)
sage: print srange(10,20,1.5)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[10, 12, 14, 16, 18]
[10.0000000000000, 11.5000000000000, 13.0000000000000, 14.5000000000000, 16.0000000000000, 17.500000
```

Pertenencia

En cualquiera de las estructuras anteriores, podemos comprobar si un elemento está en la lista usando el operador `in` , que devuelve `True` o `False` según el elemento pertenezca o no al contenedor de datos.

```
sage: 10 in [1,2,3,4]
False

sage: 'n' in 'En un lugar de la Mancha'
True

sage: 133 in srange(0,1000,7)
True
```

Conversiones entre secuencias

Las conversiones tupla -> lista, cadena -> lista, lista->tupla y cadena->tupla son triviales, usando las funciones `tuple` y `list` :

```
sage: tupla_0 = (1,2,3)
sage: lista_0 = ['a','b','c']
sage: cadena_0 = 'qwerty'

sage: tuple(cadena_0)
('q', 'w', 'e', 'r', 't', 'y')

sage: list(tupla_0)
[1, 2, 3]
```

Sin embargo, aunque existe la función `str` , el resultado no es el que deseamos normalmente:

```
sage: str(lista_0)
"['a', 'b', 'c']"

sage: str(tupla_0)
'(1, 2, 3)'
```

La función `str` intenta dar una representación textual *lo más fidedigna posible* del objeto que pasamos como argumento.

La situación siguiente es más usual: tenemos una lista de cadenas de caracteres, y queremos unir esas cadenas, opcionalmente usando otra cadena como separador. Para ello usamos el método `join` , que tienen todas las cadenas de caracteres.

```
sage: ', '.join(lista_0)
'a, b, c'

sage: ''.join(lista_0)
'abc'
```

2.1.6 Mostrar información por pantalla

Para poder ver en pantalla los valores de las variables, hemos usado el comando `print` .

```
print var1
print var1, var2, ...
```

De esta forma podemos mostrar los valores de las variables. También podemos escribir texto que contenga los valores de estas variables usando el operador `%` y los códigos de formato. Para usarlo, escribimos una cadena que contiene códigos de formato seguida del operador `%` y a continuación la variable que queremos sustituir, o una tupla de variables si hay más de una.

```
lados = 8
print 'El número de lados es %d' %lados
area = 17.5
print 'El área es %f' %area
print 'El área del polígono de %d lados es %f' %(lados, area)
nombre = 'Juan'
print '%s vino ayer a cenar' %nombre
```

Los códigos más usuales son:

- `%d` : número entero
- `%f` : número de coma flotante, con decimales
- `%.3f` : número de coma flotante, con 3 decimales exactamente
- `%s` : cadena de caracteres (o cualquier dato que no sea entero ni de coma flotante)

```
sage: lados = 8
sage: print 'El número de lados es %d' %lados
sage: area = 17.5
sage: print 'El área es %.3f' %area
sage: print 'El área del polígono de %d lados es %f' %(lados, area)
sage: nombre = 'Juan'
sage: print '%s vino ayer a cenar' %nombre
El número de lados es 8
El área es 17.500
El área del polígono de 8 lados es 17.500000
Juan vino ayer a cenar
```

2.2 Ejercicios

2.2.1 1.

Guarda en variables el número de términos de una progresión aritmética, el primer elemento y el paso. Escribe código que devuelva la suma de todos los términos, usando una fórmula. Usa los códigos de formato para escribir el resultado.

Nota: haz las asignaciones de variables en un cuadro de código y los cálculos en otro.

```
sage: primero=1
sage: paso=2
sage: nterminos=4
```

2.2.2 2.

Usando la fórmula para el área de un polígono regular

$$A = \frac{P \cdot a}{2}$$

donde A es el aŕea, P el perímetro y a la apotema, escribe un código que calcule el área y el perímetro de un polígono regular usando los valores de dos variables: `lados` (número de lados) y `longitud` (longitud del lado).

Usa los códigos de formato para escribir el resultado con cuatro dígitos significativos.

Indicación: deduce o busca en internet una fórmula para obtener la apotema en función de los datos conocidos.

```
sage: lados = 10
sage: longitud= 2
```

2.2.3 3. Explorando Sage

Sage contiene funcionalidad para trabajar en una gran cantidad de áreas de las matemáticas. Las formas principales de descubrir funcionalidad son:

1. Leer la ayuda, haciendo click en el link "help" en la cabecera de la página.
2. Escribir parte de una función o un método, y pulsar el tabulador para ver los comandos que comienzan con los caracteres introducidos.
3. Escribir el nombre de una función o de un método seguido de un carácter de interrogación para leer la ayuda contextual.

- Usando el método 2, busca una función que acepta un número y devuelve un booleano: True si el número es un cuadrado perfecto (en inglés a *square number*), y False si no lo es (pista: su nombre comienza por `is_` , pues se trata de una comprobación). Usa el método 3 para comprobar que tu suposición es correcta.
- Busca otra función que devuelva True si el número es la potencia de un número primo (en inglés, *the power of a prime number*), y False si no lo es.
- Busca un método que tienen los números racionales, que devuelva el denominador.
- Busca un método que tienen las cadenas de caracteres, que cuente el número de apariciones de un carácter. Usa ese método para contar el número de letras 'a' en esta frase.

- Hemos visto en clase que las listas tienen un método `append` que permite añadir un elemento a la lista. Busca un método que tienen las listas que permite añadir todos los elementos de una segunda lista.

```
sage: is_prime?
<html>...</html>
```

2.2.4 4. Ordena los caracteres de una cadena de caracteres

Dada una cadena de caracteres, produce otra resultado de ordenar los caracteres de la primera. No debes alterar el número de caracteres. Ejemplo:

```
'En un lugar de cuyo nombre no quiero acordarme'   \-> '        Eaaabccddeeeegilmmnnnnoooooqrrrrruuuu
```

Pista: las cadenas de caracteres no tienen un método para ordenarlas, pero las listas sí la tienen.

2.2.5 5.

Debajo de este texto puedes ver dos cuadros de texto. Tu objetivo es escribir texto formateado con el operador %, de tal forma que si ejecutas el primer bloque, el texto que aparezca en pantalla sea:

```
El número 10 es par
El lado mide 15.100
Juan tiene 10 manzanas
```

mientras que si ejecutas el segundo bloque, dando valores distintos a las variables, el resultado sea:

```
El número 7 es par
El lado mide 1250000.000
Alejandro tiene 7 manzanas
```

```
sage: nombre='Juan'
sage: numero=10
sage: longitud=15.1

sage: nombre='Alejandro'
sage: numero=7
sage: longitud=1.25e6
```

2.3 Control del flujo del programa

2.3.1 Alternativas y bucles

Alternativas lógicas

La sintaxis de los bloques **if..elif..else** en python:

```
if condicion1:
    bloque de instrucciones 1
elif condicion2:
    bloque de instrucciones 2
...
else:
    bloque de instrucciones final
```

es bastante diferente de la otros lenguajes como C, pero la esencia es la misma: Si la primera condición es cierta, se ejecuta el primer bloque de instrucciones y se abandona el bloque de instrucciones. Si es falsa, el primer bloque de instrucciones *no se ejecuta* , y se pasa a evaluar la segunda condición. Si es cierta, se ejecuta el segundo bloque de instrucciones y se abandona el bloque **if...else** . Finalmente, si todas las condiciones son falsas, se ejecuta el bloque de instrucciones final, que va precedido de la palabra clave **else** .

La indentación marca el principio y el final de cada bloque . Es decir, las líneas de los bloques de instrucciones comienzan con cuatro espacios. Después del bloque if...else, el programa puede tener más líneas, que no estarán indentadas.

Al final de cada condición if, elif o else, escribimos dos puntos (**:**).

Tanto las instrucciones **elif** como la instrucción **else** son **opcionales**.

```
sage: numero = 6
sage: if numero%2==0:
...       print ' %d es par' %numero
6 es par

sage: numero = 6
sage: if numero%2==0:
...       print ' %d es par' %numero
sage: else:
...       print ' %d es impar' %numero
6 es par
```

Observa en este ejemplo el papel de la indentación para marcar el principio y final de cada bloque:

```
sage: numero=-10
sage: if numero>0:
...       print 'La raiz de  %f es  %f' %(numero,numero^0.5)
sage: else:
...       print 'La raiz de  %f es  %f*i' %(numero,(-numero)^0.5)
sage: print 'El cuadrado de  %f es  %f' %(numero, numero^2)     #Esta instrucción no forma parte del bl
La raiz de -10.000000 es 3.162278*i
El cuadrado de -10.000000 es 100.000000
```

La condición puede ser cualquier expresión que al evaluar devuelva un booleano, y *no se pueden hacer asignaciones* . También se puede poner un número como condición (de cualquier tipo de datos), en cuyo caso *0 se interpreta como False* , y cualquier otra cosa como True, o un contenedor (lista, tupla, cadena...), en cuyo caso *un contenedor vacío se interpreta como False* , y cualquier otra cosa como True.

```
sage: ls = ['a']
sage: #ls = []
sage: if ls:
...       print 'la lista no está vacía'
la lista no está vacía
```

Bucle for

Utilizando la instrucción `for` podemos repetir una misma instrucción sobre cada elemento de una lista.

```
for elemento in lista:
    instrucciones
```

El bloque de instrucciones se ejecutará una vez por cada elemento de la lista. La variable elemento tomará sucesivamente el valor de cada elemento de la lista. A cada ejecución del bloque de instrucciones lo llamamos una **iteración** . Veamos algunos ejemplos:

```
sage: suma = 0
sage: numeros = [1, 1, 3, 7, 13, 21, 31, 43]
sage: for k in numeros:
...       suma = suma + k
sage: print suma
120

sage: lista_frutas = ['pera','manzana','naranja']
sage: print '¿Por qué letra empieza cada fruta?'
sage: for palabra in lista_frutas:
...       primera_letra = palabra[0]
...       print ' %s empieza por %s' %(palabra, primera_letra)
sage: print '¡eso es todo! gracias por su atención'
¿Por qué letra empieza cada fruta?
pera empieza por p
manzana empieza por m
naranja empieza por n
¡eso es todo! gracias por su atención
```

Bucles while

Un bucle while repite un bloque de instrucciones mientras se satisfaga una condición.

```
while condicion:
    instrucciones
```

En el ejemplo de debajo, la variable i vale 0 antes de entrar en el bucle while. Después, el valor de i aumenta en una unidad en cada iteración del bucle. Al cabo de 10 iteraciones, el valor de i es 10, y se abandona el bucle porque la condición $i < 10$ ya no se verifica.

```
sage: i = 0
sage: while i<10:
...       i = i + 1
sage: print i
10
```

Si accidentalmente escribes un bucle infinito o simplemente quieres detener el cálculo antes de que termine, puedes usar la acción *interrupt* , que está en el menú desplegable *Action* , al principio de la hoja de trabajo. Observa cómo al evaluar el código de abajo, aparece una línea verde debajo y a la izquierda del bloque de código. Esta línea indica que se está ejecutando el código, y que todavía no ha concluído. Al ejecutar *interrupt* , el cómputo se cancela, y la línea verde desaparece.

```
sage: x=0 #El valor de x no cambia al iterar
sage: while x<100:
...       x = x*1.5
sage: print x
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "_sage_input_11.py", line 10, in <module>
    exec compile(u'open("___code___.py","w").write("# -*- coding: utf-8 -*-\\n" + _support_.preparse_
  File "", line 1, in <module>

  File "/tmp/tmp0Fuz2Y/___code___.py", line 5, in <module>
    x = x*_sage_const_1p5
  File "/home/sageadm/sage-4.4.1-linux-64bit-ubuntu_9.10-x86_64-Linux/local/lib/python2.6/site-packag
    raise KeyboardInterrupt
KeyboardInterrupt
__SAGE__
```

2.3.2 Bloques anidados

Podemos anidar bloques if...else, for y while de cualquier forma posible, cuidando de que la indentación marque el principio y el final de cada bloque.

Por ejemplo, calculamos la suma de los inversos de los primos menores que k, usando un acumulador:

$$\sum_{p<k, \text{p primo}} \frac{1}{p}$$

```
sage: k=100
sage: suma = 0
sage: for j in range(k):
...       if is_prime(j):
...           suma += 1/j
sage: print 'La suma de los inversos de los primos menores que %d es %.3f' %(k, suma)
La suma de los inversos de los primos menores que 100 es 1.803
```

En el siguiente ejemplo, tenemos dos bloques for, uno dentro de otro. Observamos que nada impide que el bucle interior haga un número distinto de iteraciones cada vez.

```
sage: for j in srange(10):
...       for k in srange(j):
...           print k,
...       print
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8
```

Tablas de verdad

Para obtener la tabla de verdad de una fórmula lógica, evaluamos la fórmula en cada posible combinación de valores para p, q y r. Por ejemplo, la tabla de verdad de la fórmula:

$$e \equiv (p \wedge q) \vee (p \wedge \neg r)$$

es:

```
p=True, q=True, r=True => e=True
p=True, q=True, r=False => e=True
p=True, q=False, r=True => e=False
p=True, q=False, r=False => e=True
p=False, q=True, r=True => e=False
```

```
p=False, q=True, r=False => e=False
p=False, q=False, r=True => e=False
p=False, q=False, r=False => e=False
```

Ejemplo : genera todas las combinaciones de valores para p, q y r anidando tres bucles `for` .

```
sage: for p in [True, False]:
...       for q in [True, False]:
...           for r in [True, False]:
...               #Utilizamos el codigo %s para valores booleanos,
...               #el mismo que usamos para cadenas de caracteres
...               e = (p and q) or (p and not r)
...               print 'p= %s, q= %s, r= %s => e= %s' %(p,q,r,e)
p=True, q=True, r=True => e=True
p=True, q=True, r=False => e=True
p=True, q=False, r=True => e=False
p=True, q=False, r=False => e=True
p=False, q=True, r=True => e=False
p=False, q=True, r=False => e=False
p=False, q=False, r=True => e=False
p=False, q=False, r=False => e=False
```

Buscar un número con ciertas propiedades

En el siguiente ejemplo, buscamos un número N primo relativo a un número k dado y tal que $k/3 \leq N \leq 2k/3$. Cualquier número con esa propiedad nos vale. Podemos empezar por k/3 y seguir hacia delante, o generar números aleatorios. Esta técnica es bastante general y se puede usar en problemas de los que sabemos poco. Sin embargo, podría llevar mucho tiempo a la computadora si los números que buscamos son raros o, peor, si no hay ninguno.

En primer lugar, usamos un bucle for para iterar sobre todos los posibles casos, pero salimos del bucle usando la instrucción `break` cuando encontramos un número que satisface la condición.

```
sage: #Buscamos un numero que sea primo relativo a k
sage: #y que esté entre k/3 y 2*k/3
sage: k=196
sage: tercio = ceil(k/3)
sage: dostercios = floor(2*k/3)
sage: for x in range(tercio, dostercios):
...       if gcd(x, k) == 1:            #Si x es primo relativo a k,
...                                     #salimos del bucle for
...           break                     #gcd es el maximo comun divisor
sage: print x
67
```

A continuación, resolvemos el mismo problema probando con números aleatorios.

```
sage: #Buscamos un numero que sea primo relativo a k y que esté entre k/3 y 2*k/3
sage: #Ahora usamos numeros aleatorios
sage: k=169
sage: tercio = ceil(k/3)
sage: dostercios = floor(2*k/3)
sage: #randint(tercio, dostercios) devuelve un numero aleatorio
sage: # entre tercio y dostercios
sage: x=randint(tercio, dostercios)
sage: while gcd(x, k) != 1:
...       x = randint(tercio, dostercios)
sage: print x
110
```

2.3.3 Definición de funciones

Hemos usado ya unas cuantas funciones definidas en Sage. Ahora aprenderemos a definir nuestras propias funciones. La sintaxis para definir una función en python es:

```
def funcion(argumento1, argumento2, ...):
    instrucciones
```

De nuevo, las instrucciones que componen el cuerpo de la función están indentadas (comienzan con cuatro espacios), y la primera línea sin indentar marca el final de la declaración de la función.

Por ejemplo, una función que acepta un número como argumento, y escribe por pantalla alguna información sobre el número.

```
sage: def informacion(numero):
...       print 'Información sobre %d' %numero
...       if is_prime(numero):
...           print '    es primo'
...       if numero%2==0:
...           print '    es par'
...       if is_power_of_two(numero):
...           print '    es potencia de dos'
```

Para llamar a la función, escribimos su nombre con los argumentos entre paréntesis.

```
sage: informacion(2)
Información sobre 2
    es primo
    es par
    es potencia de dos
```

Las funciones tienen la posibilidad de *devolver un valor* , usando la palabra clave **return** . El valor devuelto por la función puede ser almacenado en una variable, o usado para hacer cálculos, exactamente igual que para las funciones definidas en Sage.

```
sage: def radio(x,y):
...       return sqrt(x^2 + y^2)

sage: print radio(1,2)
sqrt(5)

sage: r1 = radio(1,1)
sage: r2 = radio(5,0)
sage: print r2-r1
-sqrt(2) + 5
```

Observa que cuando se ejecuta una instrucción `return`, se devuelve el valor correspondiente, y *no se ejecuta ninguna otra instrucción de la función* . Observa la siguiente variante de nuestra primera función.

```
sage: def informacion_bis(numero):
...       if is_prime(numero):
...           return '%d es primo' %numero
...       if numero%2==0:
...           return '%d es par' %numero
...       if is_power_of_two(numero):
...           return '%d es potencia de dos' %numero

sage: info = informacion_bis(2)
sage: print info
2 es primo
```

Sin embargo, no podemos recoger el valor de una llamada a `informacion` :

```
sage: info = informacion(2)
sage: print info
Información sobre 2
    es primo
    es par
    es potencia de dos
None
```

Documentación

Es una buena práctica documentar el código. Unas semanas después de escribir el código, no es tan fácil recordar qué hacía. Aparte de escribir comentarios cuando lo creas conveniente comenzando con el carácter # , las declaraciones de funciones tienen una descripción (**docstring**), que muchos programas utilizan para mostrar información de la función en momentos apropiados. El lugar donde colocar la doctring es justo debajo de la definición de la función, y también indentado respecto de la función. La docstring es sólo una *cadena de caracteres* , y por tanto se puede separar entre comillas simples (‘cadena’), comillas dobles (“cadena”), o triples comillas simples para cadenas más largas (‘’‘cadena larga’‘’).

```
def funcion(argumento1, argumento2, ...):
    '''docstring
    '''
    instrucciones
```

Vamos a definir de nuevo las dos funciones anteriores colocando en su sitio las *docstring* de las funciones.

```
sage: def informacion(numero):
...       '''Imprime en pantalla alguna informacion sobre un numero
...
...       Imprime una linea que dice si es primo, otra que dice si
...       es par...
...       '''
...       print 'Información sobre %d' %numero
...       if is_prime(numero):
...           print '    es primo'
...       if numero%2==0:
...           print '    es par'
...
...       #is_power_of_two devuelve True si el numero es
...       #potencia de dos
...       if is_power_of_two(numero):
...           print '    es potencia de dos'
...
sage: def radio(x,y):
...       '''Devuelve la coordenada radial del punto con coordenadas cartesianas dadas
...
...       Tiene dos argumentos
...       '''
...       return (x^2 + y^2)^.5
```

Una manera inmediata de acceder a las docstring de todas las funciones, incluso si no las hemos definido nosotras, es escribir en un cuadro de código el nombre de la función seguido de una interrogación.

```
sage: informacion?
<html>...</html>
```

```
sage: radio?
<html>...</html>

sage: sin?
<html>...</html>

sage: is_prime?
<html>...</html>

sage: factorial?
<html>...</html>
```

¿Cuándo usar una función?

Como regla general, siempre que, resolviendo un problema, podamos identificar una subtarea claramente delimitada, debemos escribirla como función. Una tarea está bien delimitada cuando podemos describir a otra persona cuáles son los datos de entrada (un número entero, dos números reales, una cadena de caracteres y dos números naturales ...), cuáles son los datos de salida, y qué tiene que hacer la función, sin necesidad de explicarle cómo lo hemos hecho.

Veamos un ejemplo. Escribimos antes un código que calculaba la suma de los inversos de los primos menores que k. Identificamos una tarea bien delimitada: una función que recibe un número k y devuelve la suma de los inversos de los primos que son menores que k. Escribimos ahora una función que realiza esta tarea usando el código de la sesión anterior: nos limitamos a copiar el código que usamos antes, y lo colocamos dentro del cuerpo de una función.

```
sage: def sumaprimos(k):
...       '''Suma los inversos de los primos menores que un numero dado
...       '''
...       suma = 0
...       for j in srange(k):
...           if is_prime(j):
...               suma = suma + 1/j
...       return suma

sage: print 'La suma de los inversos de los primos menores que 100 es %f' %sumaprimos(100)
La suma de los inversos de los primos menores que 100 es 1.802817
```

Al conjunto formado por el nombre de la funcion, la descripción de los datos de entrada y la descripción de los datos de salida, se le llama la **signatura** de la función. Al código concreto que hemos escrito para realizar la tarea, se le llama la **implementación** . Distintas implementaciones de la misma tarea pueden conseguir el mismo resultado. Si no se cambia la signatura, es posible cambiar la implementación de la función sin necesidad de cambiar el código que *usa* la función.

Nos damos cuenta ahora de que hay una función, `prime_range` , que devuelve directamente una lista de números primos. Podemos utilizar esta función para mejorar la implementación de `sumaprimos` .

```
sage: prime_range?
<html>...</html>

sage: def sumaprimos(k):
...       '''Suma los inversos de los primos menores que un numero dado
...
...       Esta version de sumaprimos usa prime_range
...       '''
...       suma = 0
...       for j in prime_range(k):
...           suma = suma + 1/j
...       return suma
```

Aunque hemos cambiado la implementación, la signatura es la misma, y podemos llamar a `sumaprimos` de la misma forma que antes.

```
sage: print 'La suma de los primos menores que 100 es %f' %sumaprimos(100)
La suma de los primos menores que 100 es 1.802817
```

Tratar con argumentos erróneos

El lenguaje python tiene una forma muy sofisticada de tratar con los errores en tiempo de ejecución. Si algún fragmento de código produce un error, *lanza una* **excepción** . Por ejemplo, si una función espera datos de cierto tipo y recibe datos de otro tipo distinto, lanza un **TypeError** , o si realiza una división por cero lanza un **ZeroDivisionError** .

```
sage: range('a')
Traceback (most recent call last):
...
TypeError: range() integer end argument expected, got str.

sage: def funcion(a):
...       return 1/a

sage: funcion(0)
Traceback (most recent call last):
...
ZeroDivisionError: Rational division by zero
```

Los errores se propagan desde la línea que produjo el error a través de la pila de ejecución.

```
sage: def otra_funcion(a,b):
...       return a + funcion(b)

sage: otra_funcion(1,0)
Traceback (most recent call last):
...
ZeroDivisionError: Rational division by zero
```

Es importante leer las descripciones de los errores por si nos dan una pista de cuál puede ser el problema con una función que no arroja un error.

Aunque no profundizaremos en el sistema de excepciones de python, si en algún momento sentimos la necesidad de asegurarnos de que los datos de entrada de nuestras funciones verifican ciertas restricciones, utilizaremos la sintaxis " `raise Exception` ".

Por ejemplo, si en la siguiente implementación de la función factorial recibimos un número negativo corremos el riesgo de entrar en un bucle infinito:

```
sage: def factorial(n):
...       acumulador = 1
...       k = n
...       while k!=0:
...           acumulador *= k
...           k -= 1
...       return acumulador
```

Para evitar esto, comprobamos que el número sea positivo y, de otro modo, lanzamos una excepción genérica junto con un mensaje de error:

```
sage: def factorial(n):
...       if n <= 0:
...           raise Exception, 'El argumento de factorial debe ser positivo'
```

```
...       acumulador = 1
...       k = n
...       while k!=0:
...           acumulador *= k
...           k -= 1
...       return acumulador

sage: factorial(-3)
Traceback (most recent call last):
...
Exception: El argumento de factorial debe ser positivo
```

2.4 Ejercicios

2.4.1 1.

Guarda en variables los coeficientes de polinomio de segundo orden. Escribe un código que muestra por pantalla información sobre las raíces del polinomio (tiene dos raíces distintas, una doble, o ninguna).

2.4.2 2.

Para la lista de números de abajo:

1. Escribe un bucle for que escriba el resto de dividir cada uno de ellos por 5
2. Escribe un bloque for con un bloque if anidado que escriba sólo los múltiplos de 3.
3. Escribe un bloque for con otro bloque for anidado que, para cada uno de esos números, escriba, en una misma línea, todos sus divisores (es decir, no uses el método `divisors`).

```
sage: numeros = [1, 3, 7, 13, 21, 31, 43, 57, 73, 91]
```

2.4.3 3.

Escribe un bucle `while` que sume los inversos de los números naturales 1/k hasta que la suma sea mayor que 10. Escribe el número k en el que la suma se detiene.

2.4.4 4. Algoritmo de Herón

En este ejercicio, utilizamos un algoritmo que aproxima la raíz cuadrada de un número real positivo a . El algoritmo es iterativo: comienza con una aproximación burda:

$$t_0 = a$$

y después en cada iteración del algoritmo sustituimos t por otra aproximación mejor:

$$t^* = \frac{1}{2}\left(t + \frac{a}{t}\right)$$

Este algoritmo para calcular raíces cuadradas era conocido por los babilonios, y a veces se le llama algoritmo de Herón . La sucesión t_n converge a la raiz de a.

- Escribe una función que acepte dos argumentos: un número real *a* y un error máximo ϵ, y devuelva una aproximación *b* al número $\sqrt{a}$ con error menor que el error máximo ϵ:

$$|b^2 - a| < \epsilon$$

2.4.5 5.

Busca un número natural *k* tal que $|sin(k)| > 1 - \varepsilon$. No pruebes valores de ε demasiado pequeños: algo así como 0.01 es razonable. Para valores más pequeños, k puede ser muy grande.

2.4.6 6.

Definimos una secuencia de números por la siguiente regla: el número siguiente a n es n/2 si n es par, y 3n+1 si n es impar.

- Escribe una función que acepte como argumento un número *k* , y *escriba por pantalla* todos los números de la secuencia comenzando por k hasta y parando cuando se alcance por primera vez el número 1.
- Escribe una función que acepte como argumento un número *k* , y *devuelva* el número de veces que se ha iterado la regla anterior hasta llegar a 1 partiendo de k.
- Aprovecha la función del apartado anterior para encontrar un número tal que la secuencia tarde al menos 50 pasos en alcanzar 1.

Nota: Se ha conjeturado, pero no se ha demostrado todavía, que la secuencia siempre alcanza 1 eventualmente:

http://es.wikipedia.org/wiki/Conjetura_de_Collatz

http://xkcd.com/710/

2.5 Un poco de programación funcional

2.5.1 Funciones recursivas

Una función recursiva es aquella que contiene en el bloque de instrucciones que la definen una llamada a la propia función. En matemáticas también se usan definiciones recursivas y, tanto en un caso como en otro, hay que tener cuidado para estar seguro de que la definición es consistente:

1. La función debe poder calcularse directamente en uno o más de los posibles casos de uso. Estos casos se llaman **casos base**.
2. La llamada a la función con unos argumentos que no son casos base debe depender del valor de la función en otros argumentos distintos, que deben estar más cercanos a los casos base en algún sentido. Esta regla no es muy precisa, pero no se puede dar una receta general, como veremos.

Como ejemplo, escribimos una implementación recursiva de la función factorial, que se basa en la definición recursiva:

$$0! = 1$$

$$n! = n * (n-1)! \text{ si } n > 0$$

```
sage: def factorial_rec(n):
...       '''Devuelve el factorial de un numero, usando llamadas recursivas
...       '''
...       if n == 0:
...           return 1
...       else:
...           return n*factorial_rec(n-1)

sage: var = factorial_rec(1)
```

Como vemos, la implementación recursiva es una traslación bastante literal de la definición matemática del factorial.

Recursión infinita

Al igual que un bucle while mal diseñado puede repetirse infinitamente, una función recursiva mal diseñada puede repetirse indefinidamente.

```
sage: def fun_rec_inf(n):
...       '''Funcion recursiva desastrosa
...       '''
...       return n+fun_rec_inf(n-1)  #Nos hemos olvidado el caso base

sage: fun_rec_inf(10)
WARNING: Output truncated!
<html>...</html>
RuntimeError: maximum recursion depth exceeded
```

En realidad la ejecución se detiene sola sin necesidad de que interrumpamos el cálculo. En **python** , el lenguaje que usa Sage, hay un límite a la cantidad de veces que una función recursiva puede llamarse a sí misma (en general está fijado en 1000 llamadas recursivas).

2.5.2 Funciones de primera clase

En python, las funciones son *ciudadanos de primera clase*, lo que significa que podemos guardar funciones en variables normales, guardarlas en listas, pasarlas como argumentos a otras funciones, etcétera.

```
sage: funcion = factorial
sage: funcion(4)
24

sage: funciones = [is_prime, is_prime_power, is_power_of_two, is_square]
sage: k = 5
sage: for f in funciones:
...       print f(k)
True
True
False
False
```

Poder pasar funciones como argumentos a otras funciones permite abstraer algunos patrones usuales, y escribir código genérico que realiza las operaciones para funciones arbitrarias. De este modo la reutilización de código aumenta sensiblemente.

Como ejemplo, escribimos una función genérica que, partiendo de un valor inicial, itera una condición hasta que una cierta propiedad se verifique:

```
sage: def itera_hasta(f, p, ini):
...       '''Itera la funcion f, comenzando por el valor inicial ini,
...       hasta que se verifique la condicion p
...
...       Devuelve el elemento final de la iteracion
...       '''
...       t = ini
...       while not p(t):
...           t = f(t)
...       return t
```

Usando esta función genérica, podemos calcular raices cuadradas con el algoritmo de Herón:

```
sage: a = 2.0
sage: epsilon = 1e-5
sage: def f(t):
...       return (1/2)*( t + a/t)
...
sage: def p(t):
...       return abs(t^2 - a) < epsilon
...
sage: print itera_hasta(f, p, 2)
1.41421568627451
```

Pero también podemos aproximar ceros de funciones usando el método de bisección...

```
sage: a = 0.0
sage: b = 1.0
sage: def f(t):
...       return t^3 + t - 1
sage: epsilon = 1e-5
sage: valor_inicial = (a,b)
sage: def siguiente(t):
...       x0, x1 = t
...       x_medio = (x0 + x1)/2
...       if f(x0)*f(x_medio)<0:
...           return (x0, x_medio)
...       else:
...           return (x_medio, x1)
...
sage: def p(t):
...       x0, x1 = t
...       return abs(f(x0)) + abs(f(x1)) < epsilon
...
sage: print itera_hasta(siguiente, p, valor_inicial)
(0.682327270507812, 0.682331085205078)
```

El siguiente ejemplo es un clásico: tomamos una lista y la *"reducimos"* usando una función que toma dos elementos y devuelve uno (como por ejemplo, la suma) . Para ello, aplicamos la funcion entre cada dos elementos de la lista. El primer ejemplo con '+' no funciona, lo ponemos sólo a modo de ejemplo.

```
lista            :       [ x1,   x2,    x3,    x4,    x5 ]
reduce(+, lista) :         x1 +  x2  +  x3  +  x4  +  x5
reduce(f, lista) : f(f(f(f(x1,   x2),   x3),   x4),   x5)
```

```
sage: def reducir(operacion, lista):
...       acumulador = lista[0]
...       for elemento in lista[1:]:
...           acumulador = operacion(acumulador, elemento)
```

```
...       return acumulador

sage: def concatena_dos_listas(x,y):
...       return x+y
sage: reducir(concatena_dos_listas, [range(2),range(7),range(3)])
[0, 1, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2]
```

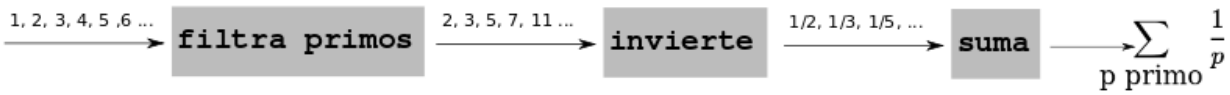
Comentario

En este caso, el patrón es tan común que python ya tiene su propia función `reduce` , similar a la que hemos definido nosotras.

```
sage: reduce?
<html>...</html>
```

2.5.3 Secuencias de datos y flujos de datos

A continuación aprenderemos otro estilo de programación en el que visualizamos series de datos (por ejemplo, números) pasando por cajas que los transforman, los filtran o los acumulan de modo que podamos realizar el cálculo deseado partiendo de series de datos conocidas.



2.5.4 Transformar y filtrar listas

En python existe una sintaxis especial para generar nuevas listas a partir de otras, aplicando una transformación a cada elemento, y seleccionando aquellos elementos que verifican una propiedad.

Transformar los elementos de una lista

La instrucción

```
[f(x) for x in lista_original]
```

genera una nueva lista, cuyos elementos son el resultado de ejecutar la función f sobre cada elemento de la `lista_original` .

```
sage: lista_original = range(10)
sage: print lista_original
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sage: [x^2 for x in lista_original]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
sage: [exp(-x)+1 for x in lista_original]
[2, e^(-1) + 1, e^(-2) + 1, e^(-3) + 1, e^(-4) + 1, e^(-5) + 1, e^(-6) + 1, e^(-7) + 1, e^(-8) + 1, e
```

Podemos transformar una lista con datos de un tipo cualquiera (enteros, booleanos, tuplas, cadenas de caracteres) en una lista con datos de cualquier otro tipo.

```
sage: print lista_original
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sage: [is_prime(x) for x in lista_original]
[False, False, True, True, False, True, False, True, False, False]

sage: [(sin(x*2*pi/8), cos(x*2*pi/8)) for x in srange(8)]
[(0, 1), (1/2*sqrt(2), 1/2*sqrt(2)), (1, 0), (1/2*sqrt(2), -1/2*sqrt(2)), (0, -1), (-1/2*sqrt(2), -1/

sage: palabras=['En', 'un', 'lugar', 'de', 'la', 'Mancha', 'de', 'cuyo', 'nombre', 'no', 'quiero', 'a

sage: [len(palabra) for palabra in palabras]
[2, 2, 5, 2, 2, 6, 2, 4, 6, 2, 6, 9]

sage: [palabra.upper() for palabra in palabras]  #El metodo upper() pasa una cadena a mayusculas
['EN', 'UN', 'LUGAR', 'DE', 'LA', 'MANCHA', 'DE', 'CUYO', 'NOMBRE', 'NO', 'QUIERO', 'ACORDARME']
```

Filtrar una lista

Con la instrucción

```
[x for x in lista if condicion(x)]
```

podemos seleccionar sólo aquellos elementos de la lista que verifican una condición. La condición es cualquier expresión o cualquier función que, para un valor x cualquiera, devuelva un valor booleano (`True` o `False`).

```
sage: [x for x in lista_original if is_prime(x)]
[2, 3, 5, 7]

sage: [x^2 for x in lista_original if x%2==1]
[1, 9, 25, 49, 81]

sage: [palabra for palabra in palabras if len(palabra)==2]
['En', 'un', 'de', 'la', 'de', 'no']
```

Todo a la vez

También podemos combinar ambas técnicas, o incluso recorrer todas las combinaciones de elementos de dos listas distintas:

```
[funcion(x) for x in lista if condicion(x)]
[funcion(x,y) for x in lista1 for y in lista2]
...

sage: #Todas las sumas de dos numeros menores que k
sage: k = 10
sage: numeros = range(k)
sage: sumas = [x + y for x in numeros for y in numeros]
sage: print sumas
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 3, 4, 5
```

Pero lo mejor es encadenar varias de estas transformaciones:

```
sage: k = 10
sage: #Todos los numeros primos menores que k
sage: primos = [x for x in range(k) if is_prime(x)]
sage: #Todas las sumas de dos numeros primos menores que k
sage: sumas_primos = [x + y for x in primos for y in primos]
sage: #Aquellas sumas de dos numeros primos menores que k que son cuadrados perfectos
sage: cuadrados = [x for x in sumas_primos if is_square(x)]
sage: cuadrados
[4, 9, 9]
```

2.5.5 Funciones que trabajan sobre listas

Las siguientes funciones encapsulan las tareas más usuales que realizamos con listas:

- `len(lista)` : devuelve la longitud de la lista
- `max(lista)` : devuelve el máximo de la lista
- `min(lista)` : devuelve el mínimo de la lista
- `sum(lista)` : devuelve la suma de los elementos de la lista
- `all(lista)` : si lista está compuesta de booleanos, devuelve True si **todos** son ciertos, y False en otro caso.
- `any(lista)` : si lista está compuesta de booleanos, devuelve True si **alguno** es cierto, y False en otro caso.

```
sage: print len([2,4,1,3])
sage: print max([2,4,1,3])
sage: print min([2,4,1,3])
sage: print sum([2,4,1,3])
sage: print all([True, False, False])
sage: print any([True, False, False])
4
4
1
10
False
True
```

```
sage: k = 86
sage: divisores = k.divisors()     #lista con los divisores de k
sage: print divisores
sage: print sum(divisores)         #La suma de los divisores de k
sage: print max(divisores)         #El mayor divisor de k
[1, 2, 43, 86]
132
86
```

Combinadas con las dos técnicas de arriba, estas funciones permiten resolver problemas no triviales:

```
sage: k = 86
sage: divisores = k.divisors()

sage: #El mayor divisor de k (sin contar a k)
sage: print max([j for j in divisores if j<k])
43
```

```
sage: #Son primos todos los divisores de k?
sage: print all([is_prime(j) for j in divisores])
False

sage: #Son primos todos los divisores de k (sin contar 1 ni k)?
sage: print all([is_prime(j) for j in divisores if 1<j<k])
True
```

Ejercicio. Responde a las siguientes preguntas con las técnicas usadas en esta sesión:

- ¿Cuántos divisores de k=21000 son cuadrados perfectos?
- ¿Hay algún número k entre 20 y 100 tal que 2^k +1 es primo?
- Encuentra el tercer cuadrado perfecto n=k^2 tal que n+1 es primo.

Ejercicio : Escribe otra implementación de la función sumaprimos de la sesión anterior usando estas ideas.

2.5.6 Evaluación perezosa

Quizá a estas alturas te hayas dado cuenta de que usar este enfoque indiscriminadamente puede llevar a realizar una cantidad de cálculos desproporcionada y a usar una gran cantidad de memoria, aún cuando el problema no lo necesita. Por ejemplo, para calcular el segundo primo entre 10000 y 100000 podemos hacer:

```
ls = srange(10000,100000)
primos = [t for t in ls if is_prime(t) ]
print primos[1]
```

pero esto nos obliga a guardar en memoria todos los números naturales entre 10000 y 100000, y después a calcular si cada uno es primo, sólo para quedarnos con el segundo.

```
sage: %time
sage: ls = srange(10000,100000)
sage: primos = [t for t in ls if is_prime(t) ]
sage: print primos[2]
10037
CPU time: 0.39 s,  Wall time: 0.39 s
```

Sin embargo, es posible mantener esta sintaxis sin hacer cálculos innecesarios. Para ello, usamos **generadores** , objetos que generan los elementos de la lista uno por uno, según se vayan solicitando desde otras partes del programa. A esta técnica se le denomina **evaluación perezosa** , porque no se hacen cálculos hasta que no son necesarios.

Por ejemplo, la función `xsrange` es la versión perezosa de `srange` (con la misma sintaxis). Mientras que `srange(100)` inmediatamente calcula los enteros menores que 100 y los guarda en una lista, `xsrange(100)` devuelve un generador, que no calcula los números en ese momento:

```
sage: srange(100)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2

sage: xsrange(100)
<generator object generic_xsrange at 0x5dacf50>
```

Una vez creado el generador, podemos pedir los números uno por uno, usando el método `next()` , o recorrerlos todos, usando un bucle `for` igual que si fuera una lista:

```
sage: xlista = xsrange(100)
sage: print xlista.next()
sage: print xlista.next()
0
1
```

```
sage: acumulador = 0
sage: for j in xsrange(101):
...       acumulador += j
sage: print acumulador
5050
```

Podemos escribir nuestros propios generadores usando una notación similar a la de transformaciones de listas, sólo que poniendo paréntesis alrededor de nuestra expresión en vez de corchetes:

```
generador = (expresion for x in secuencia if condicion)
```

```
sage: genera_cuadrados = (x^2 for x in xsrange(10))
sage: for c in genera_cuadrados:
...       print c
0
1
4
9
16
25
36
49
64
81
```

Ahora podemos modificar el programa original para calcular el segundo primo entre 10000 y 100000, usando una cantidad de memoria y cpu equivalente a un programa tradicional:

```
sage: %time
sage: ls = xrange(10000,100000)
sage: primos = (t for t in ls if is_prime(t) )
sage: primos.next()
sage: print primos.next()
10009
CPU time: 0.01 s,  Wall time: 0.01 s
```

```
sage: %time
sage: contador = 2
sage: for j in xsrange(10000,100000):
...       if is_prime(j):
...           contador -= 1
...           if contador == 0:
...               break
sage: print j
10009
CPU time: 0.00 s,  Wall time: 0.00 s
```

Sigamos la ejecución del programa paso a paso:

```
ls = xrange(10000,100000)
primos = (t for t in ls if is_prime(t) )
```

con estas dos líneas, definimos dos generadores, pero no se realiza ningún cálculo.

```
primos.next()
```

al llamar al método `next` , ponemos a trabajar al generador `primos` , que debe rodar hasta devolver un número primo. Para ello, prueba los números de la lista `ls` uno por uno. Cuando encuentra uno que pasa el filtro `is_prime(t)` , lo devuelve. En la primera llamada a `next` , `primos` devuelve el número 10007, el primer primo mayor que 10000.

```
print primos.next()
```

pedimos otro número más a `primos` , que pide a su vez números al generador `ls` continuando donde lo dejó. Recorre primero el número 10008 e inmediatemente después encuentra 10009, que es primo. El programa concluye.

2.5.7 Todo a la vez

Por supuesto, es interesante combinar los generadores con el enfoque de flujos que tratamos hoy. Consideremos por ejemplo la pregunta:

¿Hay algún número k entre 1000 y 10000 tal que 2*k +1 es primo?

Para comprobar la pregunta, podemos comprobar si el número 2*k+1 es primo para cada valor de k. Por supuesto, en cuanto encontremos un valor de k para el que 2*k+1 es primo podemos parar, y responder a la pregunta con un **sí** .

Un enfoque tradicional se aprovecha de esta propiedad:

```
hay_alguno = False
for k in srange(1000,10000):
    if is_prime(2*k+1):
        hay_alguno = True
        break
print hay_alguno
```

mientras que si transformamos listas de forma naive, tendremos que comprobar si todos los números son primos:

```
any([is_prime(2*k+1) for k in srange(1000,10000)])
```

Pero basta usar un generador para volver a conseguir un programa eficiente:

```
any(is_prime(2*k+1) for k in xsrange(1000,10000))
```

Observa que, como el generador ya estaba entre parentésis, no necesitamos poner dobles paréntesis.

```
sage: %time
sage: hay_alguno = False
sage: for k in srange(1000,10000):
...       if is_prime(2*k+1):
...           hay_alguno = True
...           break
sage: print hay_alguno
True
CPU time: 0.01 s,  Wall time: 0.00 s

sage: %time
sage: any([is_prime(2*k+1) for k in srange(1000,10000)])
True
CPU time: 0.05 s,  Wall time: 0.05 s

sage: %time
sage: any(is_prime(2*k+1) for k in srange(1000,10000))
True
CPU time: 0.01 s,  Wall time: 0.00 s

sage: %time
sage: any(is_prime(2*k+1) for k in xsrange(1000,10000))
True
CPU time: 0.00 s,  Wall time: 0.00 s
```

2.5.8 Generadores de más de una línea (opcional)

Escribir generadores en una línea a veces es poco práctico. Más importante es que en los generadores de una línea no se guarda ninguna variable de estado entre cada par de iteraciones. Por ejemplo, podemos calcular los distintos bits de un número entero usando la fórmula para el bit j-ésimo: "divide por 2^j y toma el resto de dividir por 2":

```
sage: def bit_j_esimo(a, j):
...       return (a//2^j)%2     # a//b calcula el cociente de la división entera
...                             #de dividir a entre b

sage: a = 19
sage: print list( bit_j_esimo(a,j) for j in range(0, int(log(a)/log(2)) + 1) )
sage: print  sum( bit_j_esimo(a,j) for j in range(0, int(log(a)/log(2)) + 1) )
[1, 1, 0, 0, 1]
3
```

Sin embargo, es más claro calcular los bits uno por uno haciendo una división por dos cada vez. La sintaxis para los generadores es la misma que para funciones, sólo que usando la instrucción

```
yield x
```

para devolver un valor. El objeto creado no es una función, sino un generador, que devuelve potencialmente varios valores, según los soliciten al generador. Cada vez que pedimos un elemento nuevo al generador, el flujo del programa avanza dentro del código de la función hasta que se encuentra con un `yield`, y entonces devuelve ese valor y se queda congelado hasta que le vuelvan a pedir otro valor:

```
def bits(a):
    while a:
        yield a%2
        a = a//2
```

Más información sobre generadores:

- http://en.wikipedia.org/wiki/Generator_%28computer_science%29#Python
- http://www.python.org/dev/peps/pep-0255/

```
sage: def bits(a):
...       while a:
...           yield a%2
...           a = a//2

sage: list(bits(19))
[1, 1, 0, 0, 1]

sage: g = bits(123)
sage: print g.next()
sage: print g.next()
sage: print g.next()
1
1
0

sage: g = bits(2)
sage: print g.next()
sage: print g.next()
sage: print g.next()
0
1
Traceback (most recent call last):
```

```
...
StopIteration
```

2.6 Ejercicios

2.6.1 1

Escribe una función recursiva que calcule la suma de los números menores o iguales que un cierto número n (es decir, 1+2+...+n). Si llamamos s_n a la suma de los naturales menores que n, usa la regla de inducción:

$$s_n = \begin{cases} n + s_{n-1} & n \geq 1 \\ 0 & n = 0 \end{cases}$$

2.6.2 2.

Escribe una función recursiva que calcule la suma de las cifras decimales del número que se le pasa como argumento.

Pista: si llamamos $s(n)$ a la suma de las cifras de *n* , usa la regla de inducción:

$$s(n) = \begin{cases} \text{"ultima cifra"} + s(\text{"todas las cifras menos la última"}) & n \geq 1 \\ 0 & n = 0 \end{cases}$$

2.6.3 3.

La exponenciación tiene una definición recursiva fácil de trasladar al ordenador:

$$x^0 = 1$$

$$x^n = x \cdot x^{n-1} \text{ si } n > 0$$

Escribe una función recursiva que calcule la potencia con un exponente entero y positivo.

2.6.4 4.

Usa la función `reduce` para escribir una función que multiplica los elementos de una lista.

2.6.5 5.

- Escribe una función que acepta dos argumentos, ambos listas, y devuelve una tercera lista con los elementos comunes a las dos listas que se pasaron como argumentos (es decir, la intersección).
- Usa la función `reduce` para escribir una función que acepta como único argumento una lista cuyos elementos son a su vez listas, y devuelve otra lista con la intersección de todas ellas.

2.6.6 6.

Nota : Para este ejercicio y los que siguen, intenta usar las técnicas de transformar y filtrar listas aprendidas en la hoja del bloque I sesión III.

A partir de la lista de números de abajo:

1. Consigue otra que contenga el resultado de evaluar la función $x \to e^{-x}$
2. Consigue otra que contenga el resto de dividir cada uno de ellos por 5
3. Consigue otra que contenga, en vez de cada número, una tupla con los restos de dividir el numero por 2, 3 y 5.
4. Consigue otra que contenga, en vez de cada número, una cadena de caracteres con tantos asteriscos como indica el número.
5. Consigue otra que contenga sólo aquellos que son múltiplos de 3
6. Consigue otra que contenga sólo aquellos que son múltiplos de 3 ó de 5

```
sage: numeros = [1, 3, 7, 9, 11, 13, 18, 19, 21, 30, 31, 35]
```

2.6.7 7.

A partir de la lista de palabras de debajo:

1. Consigue otra que contenga las mismas palabras, pero con cada palabra seguida de puntos suspensivos
2. Consigue otra que contenga cada palabra repetida dos veces
3. Consigue otra que contenga sólo las palabras de dos letras
4. Consigue otra que contenga la primera letra de cada palabra

```
sage: palabras=['En', 'un', 'lugar', 'de', 'la', 'Mancha', 'de', 'cuyo', 'nombre', 'no', 'quiero', 'a
```

2.6.8 8.

Escribe código que genere una lista cuyos elementos son cadenas de caracteres:

- Dado un entero k, la lista contiene k cadenas, que dicen 'j elefantes se balanceaban', para cada j que va desde 1 hasta k. Por ejemplo, para k = 5:

```
['1 elefantes se balanceaban', '2 elefantes se balanceaban', '3 elefantes se balanceaban', '4 elefant
```

2.6.9 9.

- Dado un entero k, la lista contiene k cadenas, que dicen 'j botellas contra la pared', para cada j que disminuye desde k hasta 1. Por ejemplo, para k = 5:

```
['5 botellas contra la pared', '4 botellas contra la pared', '3 botellas contra la pared', '2 botella
```

2.6.10 10.

Escribe código que responda a las siguientes preguntas:

- ¿Son todos los números de la lista de abajo primos?

- ¿Alguno es primo?
- Para cada número k de la lista de abajo, ¿k^2+2 es primo?
- La suma de los números de abajo, ¿es un número primo?
- ¿Cuál es el mayor número k de la lista de abajo tal que 2*k+1 es primo?

```
sage: numeros = [33, 39, 45, 57, 81, 99, 105, 111, 117]
```

2.6.11 11.

Ejercicio opcional (leer sólo si has seguido la sección opcional de la hoja b1s3).

Para generar los números de Fibonacci, tendríamos que escribir algo así como:

```
g = (fibonacci(k) for k in range(K))
```

Como el cálculo del k-ésimo número de Fibonacci requiere más cálculos cuanto mayor es k (dependiendo de cómo hagamos los cálculos), esta opción es poco eficiente.

Escribe un generador (de más de una línea) que devuelva los números de Fibonacci de uno en uno, pero aprovechando los cálculos realizados para calcular el número anterior.

CAPÍTULO 3

Bloque II: Eficiencia

Se presentan dos estructuras de datos nuevas: **conjuntos** y **diccionarios**. Se resuelven algunos problemas de contenido matemático y se explica la complejidad de las operaciones más usuales en python. Se presenta la filosofía consistente en primero escribir código claro y después optimizar sólo si es necesario y sólo las partes críticas. Para ello se presentan dos herramientas muy prácticas incorporadas en Sage: un **profiler**, para encontrar las partes del código que consumen más tiempo de ejecución, y **cython**, una herramienta que a menudo permite acelerar sensiblemente los cálculos sin perder ni tiempo ni claridad en el código.

3.1 Conjuntos y diccionarios

3.1.1 Tipos de datos mutables e inmutables

Como vimos, cualquier dato en SAGE tiene un *tipo de datos* . Los datos de ciertos tipos pueden ser modificados después de ser creados, mientras que otros no. En concreto, no es posible modificar los números (enteros o de coma flotante), los booleanos o las cadenas de caracteres, aunque podemos crear otros números nuevos a partir de los existentes. Las tuplas también son inmutables si los elementos que las componen lo son.

Las listas, sin embargo, son mutables, porque podemos añadir o borrar elementos, y modificar sus elementos.

La distinción es importante: ninguna instrucción, ninguna llamada a una función o un método puede modificar un dato inmutable, mientras que sí pueden modificar un dato mutable (por ejemplo, pueden añadir o quitar elementos de una lista, darle la vuelta u ordenar sus elementos).

```
sage: #Los elementos de una tupla no se pueden cambiar
sage: #y las tuplas no tienen metodos que permitan añadir
sage: #o quitar elementos
sage: tt = (1,2,3)
sage: tt[0] = 4
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment

sage: tt.append(4)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'append'
```

```
sage: #la suma de tuplas da lugar a una tupla distinta de
sage: #las anteriores, pero deja las tuplas sumadas intactas
sage: tt2 = tt + (1,2)
sage: print tt
(1, 2, 3)

sage: def escribe_al_reves(ls):
...       '''Escribe los elementos de una lista en orden inverso
...
...       Aviso: altera la lista que se pasa como argumento
...       '''
...       ls.reverse()
...       for k in ls:
...           print k

sage: #Una llamada a una funcion puede modificar un dato mutable
sage: lista = [1,2,3,4,5]
sage: print lista
sage: escribe_al_reves(lista)
sage: print lista
[1, 2, 3, 4, 5]
5
4
3
2
1
[5, 4, 3, 2, 1]
```

3.1.2 Conjuntos

El conjunto (`set`) es una estructura de datos que permite almacenar datos sin repeticiones.

Las diferencias con una lista son:

- Los elementos de un conjunto no se guardan ordenados y no se puede acceder a ellos con un índice.
- En un conjunto no hay elementos repetidos. Si añadimos un elemento que ya estaba en el conjunto, el conjunto se queda como estaba.
- Los elementos de un conjunto tienen que ser datos *inmutables* . Los tipos de datos inmutables tienen un número *hash* que es la base del funcionamiento de los conjuntos.

Podemos crear un conjunto vacío usando el comando

```
conjunto = set()
```

o crearlo con los elementos de una lista u otro contenedor de datos:

```
conjunto = set(contenedor)
```

Una vez creado, podemos añadir un elemento:

```
conjunto.add(elemento)
```

o añadir todos los elementos de otro contenedor:

```
conjunto.update(contenedor)
```

o borrar elementos:

```
conjunto.remove(elemento)
```

o comprobar si un valor pertenece al conjunto:

```
elemento in conjunto
```

A partir de ahí, tenemos a nuestra disposición las siguientes operaciones:

- Unión:

```
conjunto1 | conjunto2
```

- Intersección:

```
conjunto1 & conjunto2
```

- Diferencia:

```
conjunto1 - conjunto2
```

```
sage: conjunto1 = set([])
sage: conjunto1.add(1)
sage: conjunto1.update([1,2,6,7])
sage: conjunto1.remove(7)
sage: conjunto2 = set([1,2,3,4,1])

sage: #Observamos que el segundo conjunto tiene 4 elementos
sage: #aunque la lista original tuviera 5
sage: print len(conjunto1), len(conjunto2)
3 4

sage: conjunto1 | conjunto2
set([1, 2, 3, 4, 6])

sage: conjunto1 & conjunto2
set([1, 2])

sage: conjunto1 - conjunto2
set([6])

sage: conjunto2 - conjunto1
set([3, 4])

sage: 6 in conjunto2 - conjunto1
False

sage: 7 in conjunto2 - conjunto1
False
```

No podemos añadir a un conjunto un elemento mutable, como una lista o un conjunto:

```
sage: conjunto1.add([1,2])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'

sage: conjunto1.add(conjunto2)
Traceback (most recent call last):
...
TypeError: unhashable type: 'set'
```

También podemos comprobar si un conjunto es un subconjunto de otro:

```
conjunto1 <= conjunto2

sage: conjunto1 <= conjunto1 | conjunto2
True
```

Los elementos del conjunto no están ordenados, de modo que no tiene sentido extraer el elemento i-ésimo de la lista:

```
sage: conjunto1[2]
Traceback (most recent call last):
...
TypeError: 'set' object does not support indexing
```

Sin embargo, podemos iterar los elementos del conjunto, aunque no podemos predecir el orden en que aparecerán:

```
for elemento in conjunto:
    operaciones...

sage: conjunto3 = set(('ab','a','b'))
sage: print 'Elementos del conjunto: ',
sage: for numero in conjunto3:
...       print numero,
Elementos del conjunto:  a ab b
```

Ejemplo: conjunto de sumas

Queremos calcular cuántos números distintos obtenemos al hacer todas las posibles sumas $a+b$, donde a y b pertenecen a una lista de números. Es fácil almacenar las posibles sumas en una lista, pero entonces estaríamos contando algunos números varias veces:

```
sumas = []
for k in lista:
    for j in lista:
        sumas.append(k+j)
```

Al usar un conjunto, cada posible suma queda en el conjunto final *una sola vez* .

```
sage: def sumas(ls):
...       '''Devuelve la cantidad de sumas distintas que se pueden obtener
...       sumando dos elementos de una lista
...       '''
...       sumas_posibles = set()
...       for a in ls:
...           for b in ls:
...               sumas_posibles.add(a+b)
...       return len(sumas_posibles)

sage: print sumas([1,2,3,4])
sage: print sumas([1,3,5,7])
sage: print sumas([1,2,4,8])
7
7
10
```

Usar un conjunto para eliminar elementos repetidos de una lista

De hecho, usar un conjunto es una forma sencilla de eliminar repeticiones en una lista de elementos:

```
sage: lista = [3, 3, 2, 1, 1, 6, 1, 2, 3, 3, 7, 3, 7]

sage: conjunto = set(lista)
sage: lista_sin_repeticiones = list(conjunto)
sage: print lista_sin_repeticiones
[1, 2, 3, 6, 7]
```

3.1.3 Diccionarios

Un diccionario es una colección de pares clave-valor. Si una lista es una colección de objetos indexada por números enteros consecutivos, un diccionario permite como clave cualquier tipo de datos inmutable, y los valores pueden ser totalmente arbitrarios.

Los diccionarios (`dict`) tienen una sintaxis especial en python usando las llaves {}.

```
diccionario = {clave1: valor1, clave2:valor2 ...}
```

Una vez definido el diccionario, podemos incluir nuevos elementos, borrar y modificar elementos existentes con una sintaxis similar a la que usamos con listas:

```
diccionario[clave]=valor
del diccionario[otra_clave]
```

```
sage: #Asociamos a unas palabras su numero de vocales
sage: diccionario = {'cabeza':3, 'nariz':2, 'mano':4}
sage: print diccionario
sage: #corregimos un error
sage: diccionario['mano']=2
sage: #incluimos un nuevo elemento
sage: diccionario['pie']=2
sage: print diccionario
sage: #Eliminamos un elemento
sage: del diccionario['nariz']
sage: print diccionario
{'mano': 4, 'cabeza': 3, 'nariz': 2}
{'mano': 2, 'pie': 2, 'cabeza': 3, 'nariz': 2}
{'mano': 2, 'pie': 2, 'cabeza': 3}
```

Si necesitamos sólo las claves, o los valores podemos usar los métodos:

```
diccionario.keys()
diccionario.values()
```

```
sage: print diccionario.keys()
sage: print diccionario.values()
['mano', 'pie', 'cabeza']
[2, 2, 3]
```

El operador `in` comprueba si un objeto es una clave del diccionario, pero no si es uno de los valores.

```
sage: print 'mano' in diccionario
sage: print 'nariz' in diccionario
sage: print 2 in diccionario
True
False
False
```

Para recorrer los elementos del diccionario, podemos usar un bucle for, que recorre las claves del diccionario, sin seguir ningún orden en particular:

```
for clave in diccionario:
    instrucciones...
```

```
sage: for palabra in diccionario:
...       print 'La palabra %s tiene %d vocales' %(palabra,diccionario[palabra])
La palabra mano tiene 2 vocales
La palabra pie tiene 2 vocales
La palabra cabeza tiene 3 vocales
```

Ejemplo: mínimo común múltiplo

A modo de ejemplo, calculamos el mínimo común múltiplo de una lista de números usando su factorización: el mínimo común múltiplo es el producto de todos los primos que aparecen en cualquiera de los números de la lista con el mayor de los exponentes.

```
sage: def mcm(ls):
...       #Primera parte: recopilar los factores
...       factores = {}
...       for numero in ls:
...           for par in list(factor(numero)):
...               primo, exponente = par
...               if primo not in factores:
...                   factores[primo] = exponente
...               else:
...                   factores[primo] = max(factores[primo], exponente)
...       #Segunda parte: multiplicarlos
...       resultado = 1
...       for primo in factores:
...           resultado = resultado*primo^factores[primo]
...       return resultado

sage: #Comparamos con lcm (least common multiple) de SAGE:
sage: print mcm([12,15,15,18])
sage: print lcm([12,15,15,18])
180
180
```

Construir diccionarios a partir de listas

Podemos construir un diccionario a partir de una lista de tuplas usando el constructor de diccionarios `dict` . Cada tupla de la lista se incluirá en el diccionario como un par (clave, valor).

```
sage: lista = [('a',1), ('b',2), ('c',3)]
sage: diccionario = dict(lista)
```

Si tenemos una lista con las claves y otra con los valores (ambas de igual longitud), podemos construir una lista de tuplas que podemos pasar al constructor `dict` usando la función `zip` especialmente diseñada para ese propósito:

```
sage: lista1 = [(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
sage: lista2 = [6, 8, 10, 12, 15, 20]
sage: print lista1
sage: print lista2
sage: lista12 = zip(lista1, lista2)
```

```
sage: print lista12
sage: diccionario = dict(lista12)
sage: print diccionario
[(2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
[6, 8, 10, 12, 15, 20]
[((2, 3), 6), ((2, 4), 8), ((2, 5), 10), ((3, 4), 12), ((3, 5), 15), ((4, 5), 20)]
{(4, 5): 20, (2, 3): 6, (2, 5): 10, (3, 4): 12, (2, 4): 8, (3, 5): 15}
```

3.1.4 Como funcionan los conjuntos y los diccionarios

Hash

Los objetos inmutables tienen un hash: un número entero que representa al objeto, de modo que dos objetos distintos tengan distintos hash, al menos con probabilidad alta.

```
sage: print hash(0)
sage: print hash(1)
sage: print hash(2)
0
1
2

sage: print hash('a')
sage: print hash('b')
sage: print hash('ab')
12416037344
12544037731
12416074593111939

sage: print hash((0,0))
sage: print hash((0,1))
sage: print hash((0,2))
sage: print hash((1,0))
3713080549408328131
3713080549409410656
3713080549410493181
3713081631936575706

sage: print hash('ab')
sage: print hash((0,2))
12416074593111939
3713080549410493181
```

Por supuesto que puede haber dos datos distintos con el mismo hash, pero es muy improbable. Exceptuando a los números enteros, el hash tiene una apariencia bastante aleatoria.

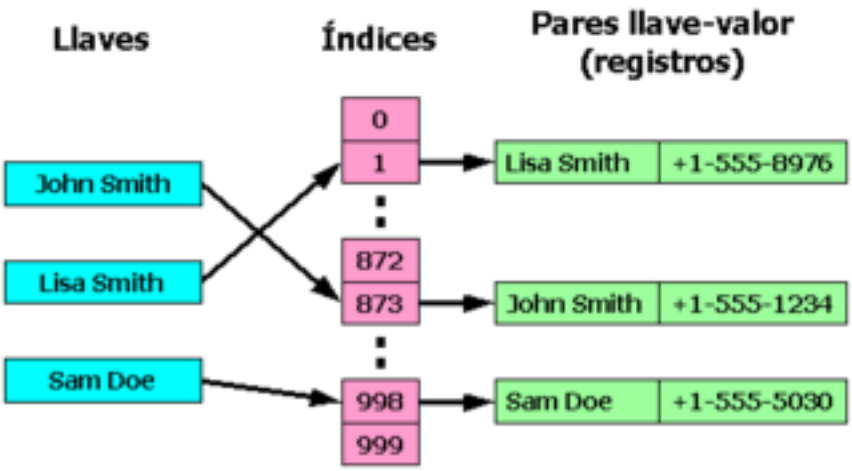
```
sage: print hash(1107710717)
sage: print hash((0,2))
sage: #Aunque tengan el mismo hash, los datos siguen siendo distintos
sage: 1107710717 == (0,2)
1107710717
3713080549410493181
False
```

Las listas son mutables, y no tienen hash.

```
sage: print hash([1,])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Tabla hash

http://es.wikipedia.org/wiki/Tabla_hash



Los conjuntos y los diccionarios se basan en una **tabla de hash** , una lista en la que guardamos los elementos usando su hash (o más bien parte del hash) como índice, en vez de insertarlos por orden de llegada.

En python, el índice usado para indexar un elemento en una tabla de tamaño 2^k son los k últimos bits del hash del elemento. Naturalmente, es posible que dos elementos distintos tengan números hash cuyos k últimos bits coincidan. Cuando introducimos dos elementos en la tabla a los que corresponde el mismo índice, decimos que ha ocurrido una **colisión** . Sin entrar en detalles sobre cómo se gestionan las colisiones, digamos que *cuantas menos colisiones ocurran mejor* . Un buen algoritmo de hash tiene dos propiedades básicas: tarda poco tiempo, y produce pocas colisiones.

3.2 Ejercicios

3.2.1 1.

Dada una lista de números enteros, construye un conjunto con los factores primos de todos los números de la lista.

Indicación: Usa `list(k.factor())` para obtener los factores primos.

3.2.2 2. Histograma

Almacena en un diccionario los pares letra:frecuencia resultantes de hacer el análisis de frecuencias de una cadena de texto:

3.2.3 3. Frecuencias de palabras

El método split permite descomponer una cadena de caracteres en palabras:

```
cadena = 'El metodo split permite descomponer una cadena de caracteres usando un caracter como separa
cadena.split()
```

```
['El', 'metodo', 'split', 'permite', 'descomponer', 'una', 'cadena',
'de', 'caracteres', 'usando', 'un', 'caracter', 'como', 'separador.']
```

- Crea una función que acepte como argumento una cadena de caracteres, y devuelva un diccionario cuyas claves sean las palabras de la cadena, y los valores sean las frecuencias de aparición de cada palabra.
- Aprovecha el resultado de la función anterior para seleccionar las palabras que aparecen más de 3 veces.

```
sage: cadena = '''Los pitónidos o pitones (Pythonidae) son una familia de serpientes constrictoras. C
```

3.2.4 4. Conjuntos y el problema de Collatz.

La sucesión de Collatz, o del 3*n+1, que ya vimos, consiste en aplicar sucesivamente la siguiente regla:

- Si un número es par, el siguiente número de la secuencia es n/2.
- Si es impar, el siguiente número de la secuencia es 3n+1.

Se conjetura que una sucesión de Collatz siempre alcanza 1, independientemente del número en que comienza. El codigo siguiente comprueba que las sucesiones que comienzan con cualquier número menor que M alcanzan 1 de una forma poco eficiente:

```
sage: %time
sage: M = 500
sage: for k in range(2,M):
...       j = k
...       while j!=1:
...           if j%2==0:
...               j = j/2
...           else:
...               j =  3*j + 1
sage: #Si el calculo ha terminado, es que hemos verificado la conjetura
sage: print 'Verificada la conjetura para k<= %d' %M
```

Con el método anterior, calculamos la sucesión de Collatz completa partiendo de cada número, a pesar de que a menudo la sucesión partiendo de un número j engancha con la sucesión partiendo de un número anterior. Por ejemplo, la sucesión partiendo de 19 alcanza 11 después de 6 iteraciones, y a partir de allí obviamente coincide con la sucesión partiendo de 11, que ya sabíamos que acaba en 1. **11** : 34 17 52 26 13 40 20 10 5 16 8 4 2 1
19 : 58 29 88 44 22 **11** 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Es fácil hacer una pequeña mejora al programa: en vez de dar la comprobación por buena cuando alcanzamos 1, terminamos en cuanto alcanzamos un número menor que el número con el que comenzamos:

```
sage: %time
sage: M = 500
sage: for k in range(2,M):
...       j = k
...       while j>=k:
...           if j%2==0:
...               j = j/2
...           else:
...               j =  3*j + 1
```

```
sage: #Si el calculo ha terminado, es que hemos verificado la conjetura
sage: print 'Verificada la conjetura para k<= %d' %M
```

Aún podemos hacer una mejora más. Observa las sucesiones que comienzan por 27 y 47:
27 : 82 41 124 62 31 94 **47** 142 71 214 107 322 161 484 242 121 364 182 91
274 137 412 206 103 310 155 466 233 700 350 175 526 263 790 395 1186 593
1780 890 445 1336 668 334 167 502 251 754 377 1132 566 283 850 425 1276
638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822
911 2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 ...
47: 142 71 214 107 322 161

Observamos que el 47 ya apareció en la secuencia del 27, y por lo tanto ya no tenemos que hacer más cuentas para verificar el número 47. Sin embargo, el programa anterior perdió esta oportunidad de ahorrar tiempo.

Después de tanto preámbulo, *tu objetivo es* :

- Calcular la sucesión de Collatz comenzando por 2,3,... hasta M.
- Almacenar en un conjunto los valores de la sucesión que vas encontrando.
- Si en algún momento de tu camino encuentras un valor conocido, abandona el cálculo de la sucesión del número actual y procede a calcular la sucesión que comienza por el número siguiente.

3.2.5 5. Conjetura de Goldbach

La conjetura de Goldbach afirma que todo número par se puede expresar como suma de dos números pares. Confirma la conjetura para todos los números pares menores que una cota K siguiendo la estrategia siguiente:

- Crea un conjunto con todos los números pares menores que K
- Crea un conjunto con todas las sumas de números primos menores que K
- Calcula la diferencia de los conjuntos para ver si todo par se puede expresar como suma de dos primos

3.2.6 6. Dos dados

- Crea un diccionario cuyas claves sean los números enteros entre 2 y 12, y su valor en el número k sea una lista de tuplas con todas las posibles formas de sumar k usando dos números enteros entre 1 y 6.

```
print dd[3]
```

```
[(1,2),(2,1)]
```

```
print dd[4]
```

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

- Generaliza el resultado a dados con un rango mayor de valores, o a un número arbitrario de dados.

3.3 Tiempo de ejecución y eficiencia de algoritmos

En esta sección vamos a estudiar algunas formas de medir la eficiencia de nuestros programas en Sage, y a estudiar la complejidad de las operaciones usuales en Sage.

Medir de forma precisa este tiempo no es una tarea trivial, y los resultados pueden variar sensiblemente de un ordenador a otro. La cantidad de factores que pueden influir en el tiempo de ejecución es muy larga:

- algoritmo usado
- sistema operativo
- velocidad del procesador, número de procesadores y conjunto de instrucciones que entiende
- cantidad de memoria RAM, y caché, y velocidad de cada una
- coprocesador matemático, GPU
- ...

Incluso en la misma máquina, el mismo algoritmo tarda algunas veces mucho más tiempo en dar el resultado que otras, debido a factores como el tiempo que consumen las otras aplicaciones que se están ejecutando, o si hay suficiente memoria RAM en el momento de ejecutar el programa.

Nuestro **objetivo** es **comparar sólo los algoritmos** , intentando sacar conclusiones independientes de la máquina.

Un mismo algoritmo se puede llamar con distintos datos de entrada. Nuestro objetivo es estudiar el tiempo de ejecución **como función del "tamaño" de los datos de entrada** . Para ello usamos dos técnicas:

- Medir tiempos de ejecución de los programas con datos de entrada de distintos tamaños
- Contar el número de operaciones que realiza el programa

3.3.1 Medir el tiempo de CPU

Comenzamos por medir el tiempo de ejecución de algunos algoritmos de forma empírica. Probando dos algoritmos que calculan el mismo objeto con datos de distinto tamaño nos haremos una idea de qué algoritmo es mejor para datos grandes.

Para evitar que el resultado tenga en cuenta el efecto de los otros programas que se están ejecutando en nuestro ordenador en ese mismo momento, SAGE utiliza los conceptos de *CPU time* y *Wall time* , que son los tiempos que el ordenador dedica exclusivamente a nuestro programa.

El *CPU time* es el tiempo de CPU que se ha dedicado a nuestro cálculo, y el *Wall time* el tiempo de reloj entre el comienzo y el final del cálculo. Ambas mediciones son *susceptibles a variaciones imprevisibles*.

La forma más sencilla de obtener los tiempos de ejecución de un comando es anteponer la palabra `time` al comando.

```
sage: time is_prime(factorial(500)+1)
False
Time: CPU 0.09 s, Wall: 0.09 s

sage: #Para datos de mayor tamanyo, tarda mas tiempo (en general)
sage: time is_prime(factorial(1000)+1)
False
Time: CPU 0.72 s, Wall: 0.76 s
```

Para medir tiempos más breves, podemos usar la funcion `timeit` .

```
sage: timeit('is_prime(1000000000039)')
625 loops, best of 3: 7.3 µs per loop
```

Gráficas de tiempo de ejecución

Para hacernos una idea del tiempo que tarda en terminar un programa en función del tamaño de los datos de entrada, vamos a hacer gráficas: en el eje *x* , el *tamaño* de los datos de entrada; en el eje y, el tiempo total.

El comando time no es lo bastante flexible, y necesitaremos las funciones `cputime` y `walltime` . `cputime` es una suerte de *taxímetro* : es un contador que avanza según hacemos cálculos, y avanza tantos segundos como la CPU

dedica a Sage. `walltime` es un reloj convencional, pero medido *en segundos desde el 1 de enero de 1970* (es el **unix clock**). Para obtener el tiempo dedicado a nuestro programa, tomamos los tiempos antes y después de la ejecución, y calculamos la diferencia.

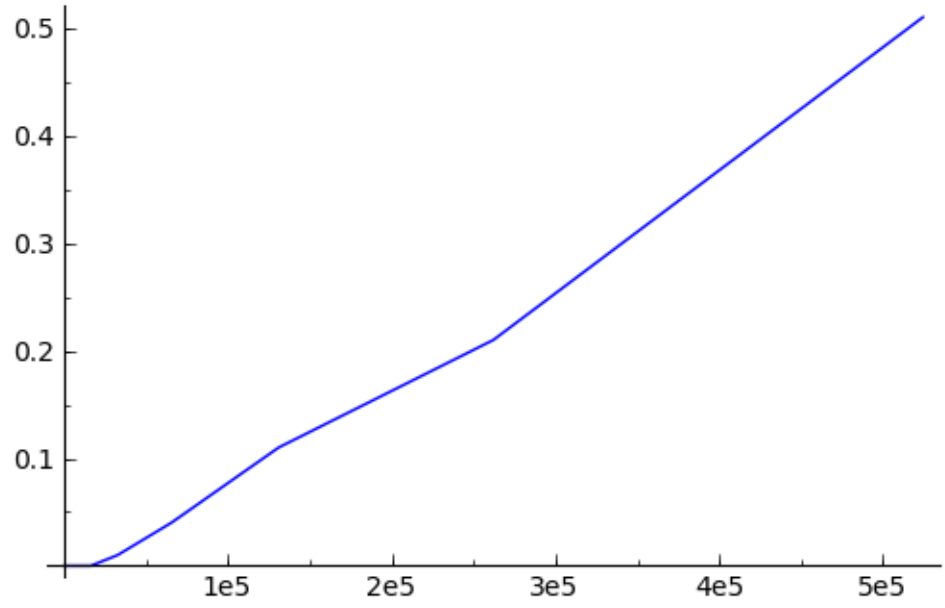
```
sage: #cputime solo avanza cuando la cpu corre
sage: #(es un taximetro de la cpu)
sage: #Ejecuta esta funcion varias veces para ver como aumenta el tiempo
sage: #Si quieres, ejecuta comandos entre medias
sage: cputime()
2.0600000000000001

sage: #walltime avanza inexorable (es un reloj normal y corriente)
sage: walltime()
1298369972.163182
```

El siguiente código guarda en una lista los cpu times empleados en ejecutar la función factorial con datos de distinto tamaño.

```
sage: numeros = [2^j for j in range(8,20)]
sage: tiempos = []
sage: for numero in numeros:
...       tcpu0 = cputime()
...       11 = factorial(numero)
...       tiempos.append(cputime()-tcpu0)
```

Dibujamos la gráfica tiempo de ejecución vs tamaño de los datos.

```
sage: p1 = line(zip(numeros,tiempos))
sage: p1.show()
```



3.3.2 Contar el número de operaciones

Una alternativa a medir el tiempo que tarda un programa que implementa un algoritmo es contar directamente el número de operaciones que hace ese algoritmo cuando lo ejecutamos. Como este es un tema que estáis estudiando a fondo en otra asignatura, no entraremos en detalle, pero repasamos las nociones básicas para fijar la terminología:

Definición. Dada una función g , diremos que otra función f es *O(g)* (y escribiremos $f \in O(g)$ o incluso $f = O(g)$!), si $0 < f(n) < c\, g(n) \qquad \forall n \geq n_0$, para constantes positivas c, n_0.

También diremos que f está dominada por g si $f \in O(g)$. Por ejemplo,

$$n^2 \in O(n^3)$$

$$p \in O(\exp)$$

Donde p es un polinomio cualquiera y $\exp$ es la función exponencial.

O, Ω y Θ

Así como $f(n) \in O(n)$ sirve para expresar una *cota superior* a la función *f*, Ω y Θ sirven para expresar una **cota inferior**, y una **cota ajustada**, respectivamente:

$$f \in \Omega(g) \Leftrightarrow 0 < g(n) < c\, f(n)\ \forall n \geq n_0$$

para dos constantes positivas c, n_0.

$$\Theta(g) = \Omega(g) \cup O(g)$$

En el caso anterior, es fácil ver que de hecho $I(n) \in \Theta(n)$.

Complejidad

Definición . Decimos que un algoritmo tiene **complejidad** ó **coste** en *O(f)* (resp $\Theta(f)$, $\Omega(f)$) si su número de operaciones (como función del tamaño de los datos de entrada) es una función que pertenece a *O(f)* (resp $\Theta(f)$, $\Omega(f)$).

Nota: Un algoritmo termina en una cantidad fija de operaciones, independientemente de los datos de entrada, si y sólo si tiene complejidad en $\Theta(1)$.

Coste en el peor caso y coste promedio

Aunque la notación de complejidad es conveniente, no deja de ser una simplificación: el número de operaciones no depende sólo del tamaño de los datos de entrada.

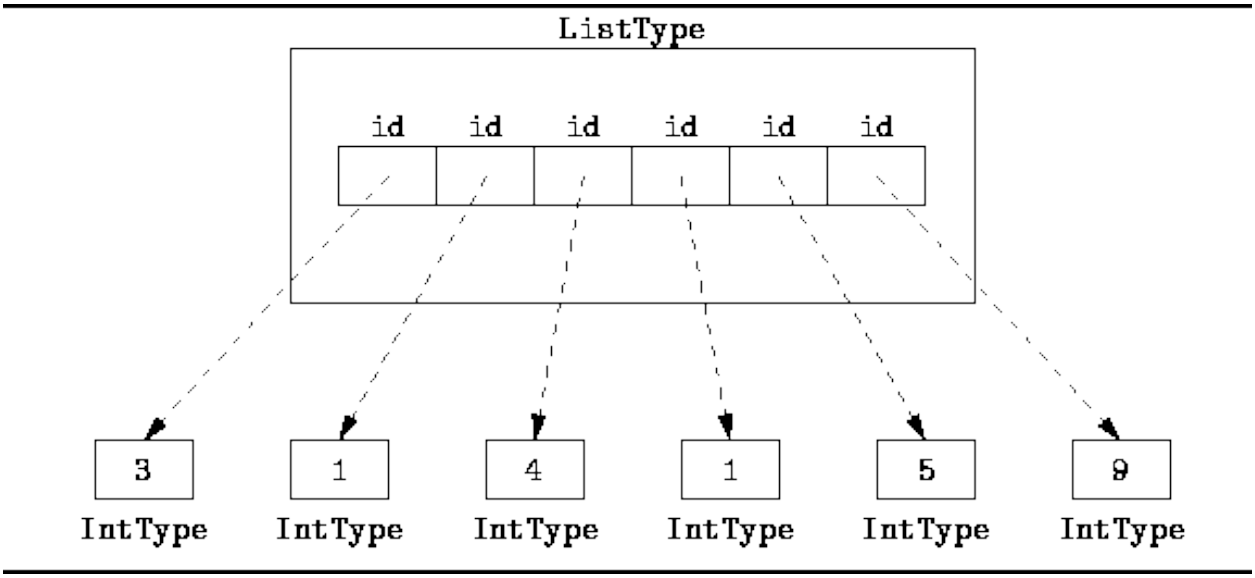
El coste en el peor caso es el máximo número de operaciones para un dato de entrada que tenga tamaño n.

El coste promedio es el promedio del número de operaciones sobre todos los posibles datos de entrada de tamaño n.

3.3.3 ¿Cómo funcionan las listas?

Las listas nos permiten almacenar una cantidad arbitraria de valores de cualquier tipo. A lo largo del programa, podemos añadir elementos a la lista, eliminarlos y acceder a cualquiera de los elementos.

Internamente, una lista es un espacio de direcciones de memoria consecutivas que contienen referencias a los objetos almacenados en la lista.



Es muy importante que las direcciones sean consecutivas. De esta forma, podemos acceder al elemento j-ésimo en poco tiempo: si la dirección de memoria del primer elemento de la lista es *d* , la dirección del elemento j-ésimo es *d+j* .

Sin embargo, mantener las direcciones consecutivas tiene un precio. Si queremos *añadir otro elemento al final* (usando `append`), es necesario que la dirección de memoria al final de la lista esté desocupada. Si no lo está, tenemos que desplazar la lista a un nuevo emplazamiento en la memoria donde haya sitio para todos los elementos. Las listas de python reservan parte del espacio detrás de la lista, de modo que no haya que recolocar la lista demasiadas veces. Añadir un elemento en cualquier posición distinta de la última obliga a desplazar los elementos posteriores de la lista para hacer hueco al nuevo elemento.

Eliminar un elemento también puede ser una operación costosa, porque después de quitar un elemento, tenemos que desplazar el resto de los elementos a la izquierda hasta tapar el hueco que deja el elemento que hemos sacado. Observa que al eliminar un elemento cercano al final de la lista sólo es necesario recolocar una pequeña parte de la lista.

Comparamos los tiempos necesarios para eliminar todos los elementos de una lista, primero eliminando cada vez el último elemento, y después eliminando el primer elemento de la lista.

```
sage: %time
sage: lista=range(20000)
sage: while len(lista)>0:
...       del lista[-1]
CPU time: 0.03 s,  Wall time: 0.03 s

sage: %time
sage: lista=range(20000)
sage: while len(lista)>0:
...       del lista[0]
CPU time: 0.17 s,  Wall time: 0.18 s
```

Ejercicio : Compara el tiempo necesario para añadir 20000 elementos al principio y al final de una lista. Para ello tienes que encontrar un método que te permita insertar un elemento al principio de la lista, usando la ayuda.

3.3.4 Coste de las operaciones usuales con listas

Añadir un elemento

Al añadir un elemento al final de la lista, pueden ocurrir dos cosas: que el espacio al final de la lista esté disponible, o que no lo esté. Si está disponible, sólo necesitamos O(1) operaciones. Si no lo está, tenemos que copiar la lista entera a un lugar nuevo, lo que cuesta O(n). Por tanto, la complejidad de añadir un elemento al final de una lista en el pero caso posible, es O(n).

Para evitar caer en el segundo caso con demasiada frecuencia, el intérprete de python reserva espacio extra después de cada lista. Así, cuando nos vemos obligados a recolocar una lista de tamaño n, le buscamos un espacio de tamaño 2*n, y colocamos los elementos de la lista al principio.

Sumemos el coste de añadir *n* elementos uno por uno a una lista:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1+1	1	1+2	1	1+4	1	1	1	1+8	1	1	1	1	1	1	1	1+
1+1	2+1	3+3	4+3	5+7	6+7	7+7	8+7	9+15	10+15	11+15	12+15	13+15	14+15	15+15	16+15	17-

Para añadir n elementos uno por uno, necesitamos hacer O(n) operaciones para añadir los elementos, y después tenemos el coste de desplazar la lista a la nueva posición. Este coste es igual a la longitud de la lista en el momento en que se tiene que desplazar. La lista se desplaza cuando añadimos un elemento a una lista de longitud 2^k, luego es igual a:

$$\sum_{k=1}^{\lfloor log_2(n) \rfloor} 2^k = 2^{\lfloor log_2(n) \rfloor + 1} - 1 \leq 2n$$

Decimos que añadir un elemento a una lista tiene **coste amortizado** *O(1)* , porque añadir n elementos siempre tiene complejidad en *O(n)* .

Insertar un elemento en una posición arbitraria

Insertar un elemento en la posición k-ésima de una lista de n elementos obliga a desplazar los n-k últimos elementos para abrir hueco. Por ejemplo, crear una lista de n elementos insertando los elementos al principio tiene coste: $1 + 2 + 3 + \cdots + n = n(n-1)/2 = \Theta(n^2)$

Quitar elementos

Quitar un elemento del final de la lista tiene coste *O(1)* (en el peor caso, no es necesario hablar de coste amortizado). Quitar el elemento en la posición k-ésima de una lista de n elementos tiene coste $\Theta(n-k)$.

Acceder o actualizar elementos

Acceder o actualizar elementos en posiciones arbitrarias no requiere recorrer la lista ni desplazar porciones de la lista, y llevan tiempo *O(1)* .

3.3.5 Comparativa entre listas y diccionarios

Si una tabla está bastante llena de elementos, *es necesario ampliarla*. Al igual que en el caso de las listas, es tarea del intérprete de python decidir el mejor momento para ampliar la tabla, y no necesitamos preocuparnos demasiado porque ocurre con relativamente poca frecuencia.

Las *búsquedas por el hash* son relativamente rápidas, aunque no lo es tanto como el acceso directo al elemento i-ésimo de una lista. Sin embargo, es mucho más rápido añadir y quitar elementos de un conjunto o diccionario que de una lista y es mucho más rápido comprobar si un objeto ya está en la tabla.

En resumen, a pesar de su uso similar, las listas son muy distintas de los diccionarios, y el coste de las operaciones elementales es distinto:

- Acceder a un elemento de un diccionario (dic[clave]) es *más lento* que a un elemento de una lista (lista[indice]). Sin embargo, ambos son similares. A efectos prácticos, podemos asumir que el número de operaciones es independiente del tamaño de los datos de entrada en ambos casos.
- Insertar o quitar un elemento del final de una lista es *más rápido* que hacerlo de un diccionario. Sin embargo, ambos son similares. A efectos prácticos, podemos asumir que el número de operaciones es independiente del tamaño de los datos de entrada en ambos casos.
- Sin embargo, insertar o quitar un elemento cualquiera de una lista es *mucho más lento* que hacerlo de un diccionario. Insertar un elemento al principio de una lista obliga a recolocar la lista, lo que requiere un tiempo proporcional al tamaño de la lista (en otras palabras, *O(n)*).
- Comprobar si un valor está en una lista requiere recorrer toda la lista y es *mucho más lento* que hacer la comprobación en un conjunto o diccionario. Como tenemos que comprobar si cada elemento de la lista es igual al valor, el número de operaciones requiere un tiempo proporcional al tamaño de la lista (en otras palabras, *O(n)*).

```
sage: lista = [k^2 for k in range(10000)]
sage: conjunto=set(lista)

sage: %time
sage: for j in range(10000):
...       b = (j in lista)
CPU time: 1.70 s,  Wall time: 1.71 s

sage: %time
sage: for j in range(10000):
...       b = (j in conjunto)
CPU time: 0.00 s,  Wall time: 0.00 s
```

Ejercicio : comprueba empíricamente las demás afirmaciones de arriba sobre la eficiencia de diccionarios y listas.

Ejemplo: intersección de listas

A modo de ejemplo, comparamos dos formas de construir una lista que tenga sólo los elementos comunes a dos listas dadas. La primera de ellos usa listas, la segunda sigue el mismo enfoque, pero con conjuntos, y la tercera usa la intersección de conjuntos (el operador `&`).

```
sage: def interseca1(l1,l2):
...       return [elemento for elemento in l1 if elemento in l2]

sage: def interseca2(l1,l2):
...       #Solo necesitamos convertir l2 en conjunto, porque
...       #comprobamos si los elementos de l1 pertenecen a l2 o no
...       c2 = set(l2)
...       return [elemento for elemento in l1 if elemento in c2]

sage: def interseca3(l1,l2):
...       c1 = set(l1)
...       c2 = set(l2)
...       ci = c1 & c2
...       return list(ci)
```

La segunda implementación tiene que crear un conjunto con los elementos de la segunda lista, pero después la operación de comprobación de pertenencia (in) es más eficiente.

La tercera implementación tiene que crear conjuntos con los elementos de la cada lista, y convertir el resultado final a un conjunto, pero a cambio podemos hacer directamente la intersección de conjuntos, que es más eficiente.

```
sage: numero = 1000
sage: #Tomamos los numeros aleatorios entre 1 y 10*numero para
sage: #que los numeros esten lo bastante espaciados, y no sea
sage: #probable que un elemento cualquiera de l1 este en l2
sage: l1 = [randint(1,10*numero) for k in range(numero)]
sage: l2 = [randint(1,10*numero) for k in range(numero)]
sage: time li = interseca1(l1, l2)
sage: time li = interseca2(l1, l2)
sage: time li = interseca3(l1, l2)
Time: CPU 0.02 s, Wall: 0.02 s
Time: CPU 0.00 s, Wall: 0.00 s
Time: CPU 0.00 s, Wall: 0.00 s
```

Los tiempos de las versiones 2 y 3 son muy bajos y bastante similares.

Ejercicio: Genera gráficas de tiempo de ejecución de `interseca1` en función del tamaño de las listas.

Ejercicio : Estima la complejidad teórica del algoritmo usado en `interseca1`.

Nota: No merece la pena hacer gráficas del tiempo de ejecución para las funciones `interseca2` y `interseca3`, porque los tiempos son tan bajos que sólo se ve ruido.

Ejemplo: números triangulares, pentagonales y hexagonales

Vamos a hacer un problema del proyecto **projecteuler** :

http://projecteuler.net/

Este sitio web tiene un montón de problemas que involucran ideas o conceptos de matemáticas y una cantidad importante de cuentas, de modo que el ordenador se hace indispensable. Nos ocupamos ahora del problema 45, que admite un planteamiento sencillo usando intersecciones de conjuntos:

http://projecteuler.net/index.php?section=problems&id=45

Se trata de encontrar 3 números que sean a la vez triangulares, pentagonales y hexagonales (contando el número 1 que lo es de forma trivial):

http://es.wikipedia.org/wiki/N%C3%BAmero_poligonal

Podemos generar números triangulares, pentagonales y hexagonales fácilmente y también es posible comprobar si un número es triangular ó pentagonal ó hexagonal.

Suponiendo que sabemos comprobar si un número es triangular en tiempo constante O(1), necesitaríamos O(N) operaciones para comprobar qué números menores que N son a la vez triangulares, pentagonales y hexagonales. Utilizando conjuntos podemos reducir el número de operaciones drásticamente.

La estrategia es la siguiente: primero generamos tres conjuntos, con los números triangulares, pentagonales y hexagonales, y luego los intersecamos. Escogemos conjuntos en vez de listas porque las intersecciones de conjuntos requieren menos operaciones.

```
sage: def busca_tph(cota):
...       '''Busca todos los números menores que una cota que son
...       a la vez triangulares, pentagonales y hexagonales
...
...       Devuelve un conjunto con todos los numeros buscados
```

```
...       '''
...       #creamos el conjunto de numeros triangulares
...       k=1
...       triangulares = set()
...       while k*(k+1)/2<cota:
...           triangulares.add(k*(k+1)/2)
...           k = k+1
...       #creamos el conjunto de numeros pentagonales
...       k=1
...       pentagonales = set()
...       while k*(3*k-1)/2<cota:
...           pentagonales.add(k*(3*k-1)/2)
...           k = k+1
...       #creamos el conjunto de numeros hexagonales
...       k=1
...       hexagonales = set()
...       while k*(2*k-1)<cota:
...           hexagonales.add(k*(2*k-1))
...           k = k+1
...
...       return triangulares & pentagonales & hexagonales

sage: %time
sage: print busca_tph(1e6)
set([1, 40755])
CPU time: 0.02 s,  Wall time: 0.02 s

sage: %time
sage: print busca_tph(1e8)
set([1, 40755])
CPU time: 0.13 s,  Wall time: 0.13 s
```

Aumentando la cota, el tiempo de ejecución aumenta, lógicamente. En vez de hacer gráficas de tiempo de ejecución, vamos a reflexionar sobre el tiempo que debería tardar y la cantidad de memoria que necesita este enfoque para saber hasta dónde podemos buscar.

Tanto los números triangulares como los pentagonales y los hexagonales *crecen de forma cuadrática* . Por lo tanto, si fijamos una cota N, el tamaño de cada uno de los conjuntos `triangulares` , `pentagonales` y `hexagonales` será menor que una constante por la *raíz cuadrada de N* . Para construirlos, habremos dedicado a lo sumo $O(\sqrt{N})$. Una vez construidos los conjuntos, la intersección de dos conjuntos de tamaño K1 y K2 se puede realizar en tiempo O(min(K1, K2)), porque basta con comprobar si cada elemento del conjunto menor está en el otro conjunto.

De modo que el tiempo de ejecución debería crecer como la raíz de la cota, y el uso de la memoria también, por motivos similares. Podemos permitirnos poner una cota de 1e10, y el tiempo de ejecución sólo debería aumentar unas 10 veces frente al tiempo que tardó la llamada con cota = 1e8.

```
sage: %time
sage: print busca_tph(1e10)
set([1, 40755, 1533776805])
CPU time: 1.49 s,  Wall time: 1.49 s
```

3.4 Ejercicios

3.4.1 1.

El resultado de las mediciones de %time, cputime y los demás es algo inestable. Al medir tiempos, es buena idea tomar promedios. Escribe código que almacene en una lista el tiempo promedio de ejecutar una cierta función N veces.

3.4.2 2.

Haz graficas que midan el tiempo de ejecución en función de su único argumento de la función `es_primo` definida más abajo. Asegúrate de probar la función con **todos** los números k en un rango [2,N]: ¿puedes explicar la forma de la gráfica?

```
sage: def es_primo(k):
...       return k>1 and all(k%j!=0 for j in range(2,k))
```

3.4.3 3.

Escribe código que crea una lista añadiendo elementos uno por uno, primero añadiendo los elementos al final de la lista y después añadiéndolos al final de la lista. ¿Cuaĺ debería ser la complejidad con cada enfoque?

Haz gráficas del tiempo de ejecución en función del tamaño de la lista que construyes. ¿Es consistente con la complejidad predicha?

3.4.4 4.

Escribe código que, partiendo de una lista de tamaño N, elimina sus elementos uno por uno hasta que la lista quede vacía. Resuelve el problema de dos formas: primero añadiendo los elementos al final de la lista y después añadiéndolos al final de la lista. ¿Cuaĺ debería ser la complejidad con cada enfoque?

Haz gráficas del tiempo de ejecución en función del tamaño de la lista. ¿Es consistente con la complejidad predicha?

3.4.5 5.

Compara dos formas de calcular las formas de expresar un par como suma de dos primos. La única diferencia está en el uso de una lista o de un conjunto para almacenar los números primos. Realiza gráficas de tiempo de ejecución en función de n: ¿es sensato pensar que el tiempo necesario para comprobar si un elemento pertenece a un conjunto no depende del tamaño del conjunto?

```
sage: def sumas1(n):
...       if n%2!=0:
...           #Si n es impar, no devolvemos nada
...           return
...       else:
...           lista = []
...           primos = prime_range(n)
...           for k in primos:
...               if n-k in primos:
...                   lista.append((k,n-k))
...           return lista
```

```
sage: def sumas2(n):
...       if n%2!=0:
...           #Si n es impar, no devolvemos nada
...           return
...       else:
...           lista = []
...           primos = set(prime_range(n))
...           for k in primos:
...               if n-k in primos:
...                   lista.append((k,n-k))
...           return lista
```

3.4.6 6.

Encuentra todos los números primos menores que un millón que son de la forma $k^2 + 1$, con k un número triangular.

3.4.7 7.

Encuentra el menor número primo con k dígitos decimales distintos, para k=5,6,7. Resuelve el problema de las dos formas siguientes:

- Crea una lista de primos candidatos con prime_range, y busca entre ellos el primero con la propiedad pedida. Este enfoque obliga a fijar una cota máxima y buscar primos menores que esa cota.
- Genera los números primos de uno en uno usando next_prime. En este enfoque no hay que fijar una cota a priori.

3.4.8 8.

Observa que ningún primo menor que 10000 puede tener menos de 5 cifras. Podemos por tanto comenzar la búsqueda por este número. Apurando un poco más, podemos comenzar a buscar primos por el menor número con cinco cifras distintas, el 10234. Incorpora esta información y mide la mejora en tiempo de ejecución.

3.5 Eficiencia en cálculo científico

En cálculo ciéntifico, no es raro que ejecutemos un código una sóla vez. Obtenido el resultado, guardamos el código en el archivo y la salida del programa nos responde a una pregunta y nos ayuda a plantear la siguiente. Claramente, no tiene sentido dedicar tiempo a optimizar al máximo un código que probablemente nadie vuelva a usar.

Tenemos que encontrar un equilibrio entre el tiempo que dedicamos a optimizar el programa y el tiempo que podemos esperar a que el código sin optimizar termine. Claramente, nuestro tiempo vale más que el tiempo del ordenador.

La optimización, frecuentemente, también va reñida con otro aspecto muy importante del código científico, la *claridad* , necesaria para poder compartir nuestro código con la comunidad internacional. Optimizaciones marginales que hacen el código más difícil de seguir (y por tanto, más propenso a esconder errores), no compensan.

PD: Lo anterior se aplica al código **exploratorio** , que es el más cotidiano en ciencia: un análisis estadístico, una conversión de formato, la solución de una ecuación concreta... pero no al código que forma las librerías en las que se basa el trabajo de mucha gente, porque ese código se va a ejecutar muchas veces en muchos contextos distintos.

3.5.1 Optimiza sólo si es necesario, sólo dónde es necesario

Hoy vamos a practicar una metodología típica de lenguajes dinámicos como python:

1. Escribe código correcto, y claro.
2. Pruébalo en suficientes casos.
3. Comprueba si es lo bastante rápido para el tamaño de tus datos. Si lo es, dedícate a otra cosa.
4. Si el código no es lo bastante rápido, identifica las partes del programa que más influyen en la velocidad (una conocida heurística dice que el 90 % del tiempo se pasa ejecutando el 10 % del código).
5. Piensa si tu algoritmo es mejorable y, si puedes, usa otro mejor.
6. Si tu código no es lo bastante rápido y no conoces mejores algoritmos, reescribe las partes críticas del código en un lenguaje compilado como C, FORTRAN, o **cython** .

Caso práctico: la constante de Brun

Dos números primos p y p+2 se llaman **primos gemelos** .

La constante de Brun es por definición la suma de los inversos de todos los primos gemelos (y curiosamente se sabe que converge, aunque no se sabe ni siquiera si existen infinitos primos gemelos).

$$B_2 = \left(\frac{1}{3}+\frac{1}{5}\right)+\left(\frac{1}{5}+\frac{1}{7}\right)+\left(\frac{1}{11}+\frac{1}{13}\right)+\left(\frac{1}{17}+\frac{1}{19}\right)+\left(\frac{1}{29}+\frac{1}{31}\right)+\cdots$$

```
sage: #Suma de los inversos de los primos gemelos
sage: #1: Encuentra primos
sage: def criba(ls):
...       '''Se queda con los elementos irreducibles de una lista de enteros'''
...       primos = []
...       while ls:
...           p = ls[0]
...           primos.append(p)
...           ls = [k for k in ls if k%p]
...       return primos
sage: def lista_primos(K):
...       'genera los numeros primos menores que K'
...       return criba(range(2,K))
sage: #2: Selecciona los gemelos
sage: #Nos quedamos con el menor de cada par
sage: def criba_gemelos(ls):
...       '''recibe una lista de primos, y devuelve los numeros p tales que
...       p+2 tambien esta en la lista'''
...       return [p for p in ls if p+2 in ls]
sage: #3: Sumamos los inversos
sage: #para aproximar la constante de Brun
sage: def brun(K):
...       '''Devuelve la suma de los inversos de los primos gemelos menores que K'''
...       primos = lista_primos(K)
...       gemelos = criba_gemelos(primos)
...       return sum( (1.0/p + 1.0/(p+2)) for p in gemelos)
```

```
sage: %time
sage: print brun(1e4)
1.61689355743220
CPU time: 1.40 s,  Wall time: 1.41 s
```

La serie converge muy despacio, y si queremos verificar el primer dígito decimal, tenemos que alcanzar al menos $K = 10^8$, lo que claramente nos obliga a optimizar el código.

Aunque con este código tan breve puede ser obvio, vamos a usar una herramienta de profile como ayuda a la metodología de la optimización progresiva. La herramienta nos ayuda a identificar las partes del programa más lentas con menos trabajo que si sólo usamos timeit o similares.

```
sage: #importamos los modulos cProfile y pstats para ver las estadisticas
sage: #de cuanto tiempo se pasa en cada parte del codigo
sage: import cProfile, pstats
sage: #No necesitamos entender la siguiente linea:
sage: #tomalo como una version avanzada de timeit
sage: cProfile.runctx("brun(10000)", globals(), locals(), DATA + "Profile.prof")
sage: s = pstats.Stats(DATA + "Profile.prof")
sage: #Imprimimos las estadisticas, ordenadas por el tiempo total
sage: s.strip_dirs().sort_stats("time").print_stats()
Tue Feb 22 11:18:44 2011    /home/sageadm/nbfiles.sagenb/home/pang/214/data/Profile.prof

         1446 function calls in 1.371 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.274    1.274    1.274    1.274 ___code___.py:19(criba_gemelos)
        1    0.095    0.095    0.096    0.096 ___code___.py:4(criba)
      206    0.001    0.000    0.001    0.000 ___code___.py:30(<genexpr>)
        1    0.000    0.000    0.000    0.000 {range}
     1229    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.001    0.001 {sum}
        1    0.000    0.000    0.096    0.096 ___code___.py:13(lista_primos)
        1    0.000    0.000    1.371    1.371 <string>:1(<module>)
        1    0.000    0.000    1.371    1.371 ___code___.py:26(brun)
        1    0.000    0.000    0.001    0.001 functional.py:547(symbolic_sum)
        1    0.000    0.000    0.000    0.000 {hasattr}
        1    0.000    0.000    0.000    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats instance at 0x433d5f0>
```

Vemos que la llamada a criba_gemelos es la que ocupa la mayor parte del tiempo. El algoritmo es mejorable: es un algoritmo cuadrático (para cada p de la lista recorremos la lista entera para buscar p+2), cuando para esta tarea podemos usar un algoritmo lineal. Observamos que el primo p+2, si está en la lista, sólo puede estar en un sitio: inmediatamente a continuación del primo p.

```
sage: #2: Selecciona los gemelos
sage: #Nos quedamos con el menor de cada par
sage: def criba_gemelos(ls):
...       return [ls[j] for j in xrange(len(ls)-1) if ls[j+1]==ls[j]+2]

sage: %time
sage: print brun(1e4)
1.61689355743220
CPU time: 0.12 s,  Wall time: 0.11 s
```

Pregunta : ¿qué otra forma se te ocurre para reducir la complejidad de criba_gemelos a O(len(ls))?

La mejora es sustancial, y nos planteamos avanzar un orden de magnitud

```
sage: %time
sage: print brun(1e5)
1.67279958482774
CPU time: 5.69 s,  Wall time: 5.69 s
```

Como desgraciadamente el resultado sigue siendo insuficiente, volvemos a aplicar el profile para buscar el siguiente fragmento de código que necesita mejoras...

```
sage: import cProfile, pstats
sage: cProfile.runctx("brun(50000)", globals(), locals(), DATA + "Profile.prof")
sage: s = pstats.Stats(DATA + "Profile.prof")
sage: s.strip_dirs().sort_stats("time").print_stats()
Tue Feb 22 11:18:53 2011    /home/sageadm/nbfiles.sagenb/home/pang/214/data/Profile.prof

         5851 function calls in 1.554 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.540    1.540    1.541    1.541 ___code___.py:4(criba)
        1    0.008    0.008    0.008    0.008 ___code___.py:4(criba_gemelos)
      706    0.003    0.000    0.003    0.000 ___code___.py:30(<genexpr>)
        1    0.001    0.001    0.001    0.001 {range}
     5133    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
        1    0.001    0.001    1.543    1.543 ___code___.py:13(lista_primos)
        1    0.000    0.000    0.003    0.003 {sum}
        1    0.000    0.000    1.554    1.554 <string>:1(<module>)
        1    0.000    0.000    1.554    1.554 ___code___.py:26(brun)
        1    0.000    0.000    0.000    0.000 {hasattr}
        1    0.000    0.000    0.003    0.003 functional.py:547(symbolic_sum)
        2    0.000    0.000    0.000    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats instance at 0x433df38>
```

Vemos que ahora sólo compensa dedicarle tiempo a la criba. Comenzamos por estudiar el algoritmo. Algunos de vosotros presentasteis esta otra variante en la práctica, en la que para hallar los primos menores que N mantenemos un array que comienza con todos los números de 1 a N, y cuando descubrimos un número compuesto, lo tachamos poniendo un cero en la posición del array que ocupa.

```
sage: ##Variante
sage: def lista_primos2(n):
...       aux = [True]*int(n)
...       aux[0] = False
...       aux[1] = False
...
...       for i in xrange(2,floor(sqrt(n))+1):
...           if aux[i]:
...               for j in xrange(i*i,n,i):
...                   aux[j] = False
...
...       ##Devolvemos los que no están tachados
...       return [k for k in xrange(n) if aux[k]]
```

```
sage: time b=lista_primos2(1000000)
Time: CPU 0.25 s, Wall: 0.24 s
```

Aunque podría parecer que ambos algoritmos hacen las mismas operaciones, observamos que:

- Con el primer método, intentamos dividir un número por todos los números primos que son menores que su menor factor primo.
- Con el segundo método, cada número se *tacha* varias veces, una por cada factor primo del número.

No necesitamos conocer más detalles, por ahora es suficiente con entender que hacen dos cosas distintas. El segundo método resulta ser mucho más eficiente: (detalles en la wikipedia)

```
sage: time a=lista_primos(50000)
sage: time b=lista_primos2(50000)
sage: a==b
Time: CPU 1.54 s, Wall: 1.54 s
Time: CPU 0.01 s, Wall: 0.01 s
True

sage: def brun(K):
...       primos = lista_primos2(K)
...       gemelos = criba_gemelos(primos)
...       return sum( (1.0/p + 1.0/(p+2)) for p in gemelos)

sage: %time
sage: print brun(1e5)
1.67279958482774
CPU time: 0.05 s,  Wall time: 0.05 s

sage: %time
sage: print brun(1e6)
1.71077693080422
CPU time: 0.39 s,  Wall time: 0.39 s
```

El crecimiento aparenta ser casi lineal, y estimamos que podemos tener nuestra respuesta en un tiempo asumible, y pasamos al siguiente problema.

3.5.2 Cython

Si estamos usando el mejor algoritmo, o no conocemos otro mejor, y aun así nuestro programa no es lo bastante rápido, podemos compilar las partes críticas del programa.

El lenguaje cython se adapta perfectamente a esta tarea, ya que combina tipos de datos de C con sintaxis de python, de modo que podemos alcanzar mayor velocidad sin tener que reescribir nuestros programas.

Comenzamos con un típico ejemplo numérico, en el que calculamos una integral mediante una suma de Riemann.

```
sage: def f(x):
...       return sin(x**2)
sage: def integral(a, b, N):
...       dx = (b-a)/N
...       s = 0
...       for i in range(N):
...           s += f(a+dx*i)
...       return s * dx
```

```
sage: time integral(0.0, 1.0, 500000)
0.310267460252752
Time: CPU 6.24 s, Wall: 6.26 s
```

Para compilar una función en cython, comenzamos el bloque de código con `%cython` .

```
sage: %cython
sage: #Tenemos que importar la funcion seno
sage: from math import sin
sage: def f(x):
...       return sin(x**2)
sage: def integral_cy1(a, b, N):
...       dx = (b-a)/N
...       s = 0
...       for i in range(N):
...           s += f(a+dx*i)
...       return s * dx
```

Al ejecutar un bloque de código que comienza por `%cython` , Sage *compila el código* , y lo deja listo para llamarlo más adelante como una función normal definida en python.

```
sage: time integral_cy1(0.0, 1.0, 500000)
0.310267460252752
Time: CPU 2.79 s, Wall: 2.83 s
```

Como vemos, el código apenas es un poco más rápido que antes: no hemos indicado los tipos de los datos, y en esta situación es imposible hacerlo significativamente mejor que el intérprete de python.

En la siguiente versión le indicamos los tipos de los datos usando la palabra clave **cdef** : `int` o `float` .

```
sage: %cython
sage: from math import sin
sage: def f(double x):
...       return sin(x**2)
...
sage: def integral_cy2(double a, double b, int N):
...       cdef double dx = (b-a)/N
...       cdef int i
...       cdef double s = 0
...       for i in range(N):
...           s += f(a+dx*i)
...       return s * dx

sage: time integral_cy2(0.0, 1.0, 500000)
0.31026746025275187
Time: CPU 0.10 s, Wall: 0.10 s
```

La estrategia anterior (compilar en cython indicando los tipos de los datos) se suele poder aplicar de forma bastante mecánica y da resultados interesantes. En general, es posible optimizar el código todavía más, pero hace falta conocimiento más específico y por tanto más difícil de aplicar en cada caso concreto. A modo de referencia, veamos el siguiente código:

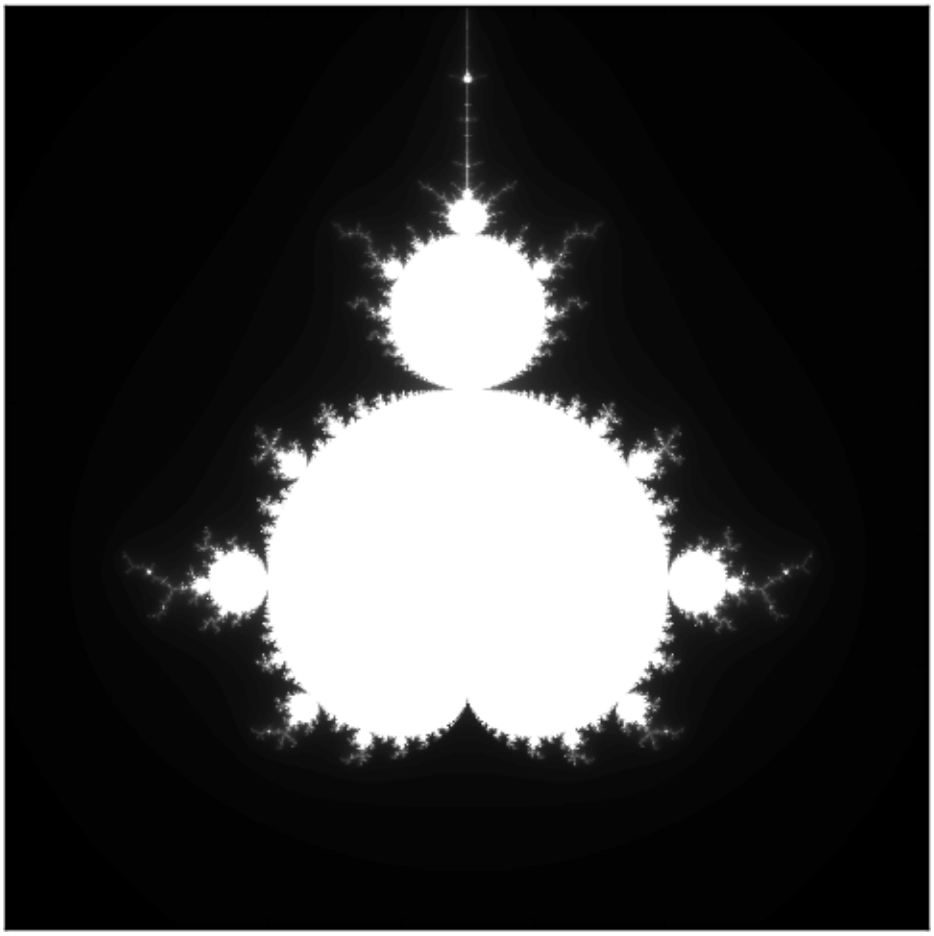
```
sage:%cython
sage: #Usamos la funcion seno directamente de la libreria
sage: #"math.h" de C
sage: cdef extern from "math.h": # external library
...       double sin(double)
sage: #Las funciones definidas con cdef solo son accesibles
sage: #desde codigo cython, pero son mas rapidas
sage: cdef double f(double x):
```

```
...       return sin(x**2)
sage: def integral_cy3(double a, double b, int N):
...       cdef double dx = (b-a)/N
...       cdef int i
...       cdef double s = 0
...       for i in range(N):
...           s += f(a+dx*i)
...       return s * dx

sage: time integral_cy3(0.0, 1.0, 500000)
0.31026746025275187
Time: CPU 0.02 s, Wall: 0.02 s
```

3.5.3 El conjunto de Mandelbrot

El conjunto de Mandelbrot es el subconjunto de los números complejos formados por aquellos números c tales que las iteraciones de la regla $z \to z^2 + c$ comenzando en $z_0 = 0$ permanecen acotadas.

Como aproximación, es habitual tomar un punto c del plano complejo e iterar la regla anterior hasta que se abandona una bola de un cierto radio, o se alcanza un cierto número de iteraciones.

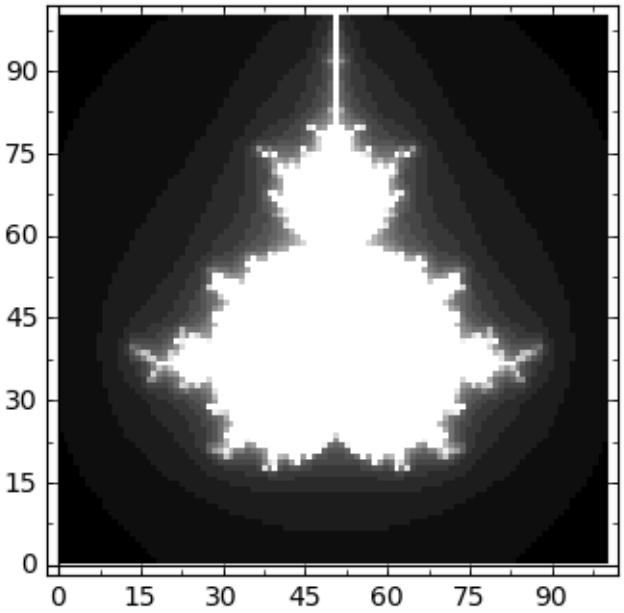


Comenzamos con un programa en python puro.

- Dividimos el cuadrado del plano complejo con vértices (x_0, y_0), $(x_0 + l, y_0)$, $(x_0 + l, y_0 + l)$ y $(x_0, y_0 + l)$ en una malla NxN.

- En la posición (j,k) de un array NxN almacenamos el número de iteraciones **h** necesarias para que z_h abandone una bola de radio R cuando comenzamos a iterar $z \to z^2+c$ partiendo del punto $c = (x_0+jl/N)+i(y_0+kl/N)$.

```
sage: def mandelbrot(x0, y0, side, N=200, L=50, R=float(3)):
...       m=matrix(N,N)
...       delta = side/N
...       for j in range(N):
...           for k in range(N):
...               c = complex(x0+j*delta,y0+k*delta)
...               z=0
...               h=0
...               while (h<L) and (abs(z)<R):
...                   z=z*z+c
...                   h+=1
...               m[j,k]=h
...       return m

sage: time m=mandelbrot(-2,-1.5,3,100,20)
sage: matrix_plot(m)
Time: CPU 0.65 s, Wall: 0.67 s
```



Nuestro código es una traslación bastante literal de la definición, y sin saber ninguna propiedad especial del conjunto no parece haber forma de sustituir el algoritmo. Usamos el lenguaje cython.

```
sage: %cython
sage: def mandelbrot_cy_1(x0, y0, side, N=200, L=50, R=float(3)):
...       m=matrix(N,N)
...       delta = side/N
...       for j in range(N):
...           for k in range(N):
...               c = complex(x0+j*delta,y0+k*delta)
...               z=0
...               h=0
...               while (h<L) and (abs(z)<R):
...                   z=z*z+c
...                   h+=1
```

```
...                 m[j,k]=h
...         return m


Error converting Pyrex file to C:
------------------------------------------------------------
...
include "interrupt.pxi"  # ctrl-c interrupt block support
include "stdsage.pxi"  # ctrl-c interrupt block support

include "cdefs.pxi"
def mandelbrot_cy_1(x0, y0, side, N=200, L=50, R=float(3)):
    m=matrix(N,N)
            ^
------------------------------------------------------------

/home/sageadm/.sage/temp/sageserver/23963/spyx/_home_sageadm_nbfiles_sagenb_home_pang_214_code_sage35
Traceback (most recent call last):
...
RuntimeError: Error converting /home/sageadm/nbfiles.sagenb/home/pang/214/code/sage35.spyx to C:
```

Al tratar de compilar el código, obtenemos un error:

```
undeclared name not builtin: matrix
```

que nos indica que tenemos que importar el constructor de matrices (`matrix`) de la librería correspondiente. Encontramos la ruta requerida para importar en la ayuda de matrix, más concretamente en la primera línea, que indica el fichero en que se define `matrix` .

```
sage: matrix?
<html>...</html>
```

```
sage: %cython
sage: from sage.matrix.constructor import matrix
sage: def mandelbrot_cy_1(x0, y0, side, N=200, L=50, R=float(3)):
...       m=matrix(N,N)
...       delta = side/N
...       for j in range(N):
...           for k in range(N):
...               c = complex(x0+j*delta,y0+k*delta)
...               z=0
...               h=0
...               while (h<L) and (abs(z)<R):
...                   z=z*z+c
...                   h+=1
...               m[j,k]=h
...       return m

sage: time m=mandelbrot_cy_1(-2,-1.5,3,200,50)
Time: CPU 1.88 s, Wall: 1.89 s
```

Al ejecutar un bloque de código que comienza por `%cython` , Sage *compila el código* , y lo deja listo para llamarlo más adelante como una función normal definida en python. También genera un *informe en html* que nos permite entender cómo de eficiente es el código generado. Las líneas en amarillo son código que no se ha podido optimizar, y se ejecuta como si fuera código dinámico en python, mientras que las líneas en blanco corresponden a las líneas que se han podido optimizar, y se ejecutan como si fuera código de C. En este primer informe, vemos que casi todas las líneas están en amarillo, porque no hemos indicado los tipos de los datos, y en esta situación es imposible hacerlo significativamente mejor que el intérprete de python.

En la siguiente versión le indicamos los tipos de los datos: `int` , `float` o `double complex` .

```
sage: %cython
sage: from sage.matrix.constructor import matrix
sage: def mandelbrot_cy2(float x0,float  y0,float side,
...                      int N=200, int L=50, float R=3):
...       '''returns an array NxN to be plotted with matrix_plot
...       '''
...       cdef double complex c, z
...       cdef float delta
...       cdef int h, j, k
...       m=matrix(N,N)
...       delta = side/N
...       for j in range(N):
...           for k in range(N):
...               c = complex(x0+j*delta,y0+k*delta)
...               z=0
...               h=0
...               while (h<L) and (abs(z)<R):
...                   z=z*z+c
...                   h+=1
...               m[j,k]=h
...       return m

sage: time m=mandelbrot_cy2(-2,-1.5,3,200,50)
Time: CPU 0.10 s, Wall: 0.10 s
```

Indicando los tipos de las variables hemos conseguido una mejora sustancial. Este es un buen momento para detenernos (de hecho, el resto de la sección no entra en el examen).

Optimización en cython más allá de indicar los tipos de datos (opcional)

Observando el informe html sobre el código generado, vemos que la condición dentro del bucle while no se está optimizando, y está dentro del bucle más interior, luego es la parte del código que más se repite. El problema es que estamos usando una función abs genérica. Podemos acelerar el cálculo sustituyendo la llamada por operaciones generales sobre números reales (y eliminando la raíz cuadrada implícita al calcular el valor absoluto):

```
sage: %cython
sage: from sage.matrix.constructor import matrix
sage: def mandelbrot_cy3(float x0,float  y0,float side,
...                      int N=200, int L=50, float R=3):
...       '''returns an array NxN to be plotted with matrix_plot
...       '''
...       cdef double complex c, z
...       cdef float delta
...       cdef int h, j, k
...       m=matrix(N,N)
...       delta = side/N
...       for j in range(N):
...           for k in range(N):
...               c = complex(x0+j*delta,y0+k*delta)
...               z=0
...               h=0
...               while (h<L and
...                      z.real**2 + z.imag**2 < R*R):
...                   z=z*z+c
...                   h+=1
```

```
...                 m[j,k]=h
...         return m

sage: time m=mandelbrot_cy3(-2,-1.5,3,200,50)
Time: CPU 0.05 s, Wall: 0.06 s
```

Siguiendo la regla de optimizar sólo la parte que más repite, ponemos el ojo en la línea:

```
c = complex(x0+j*delta,y0+k*delta)
```

que aparece en amarillo en el informe, y es interior a dos bucles for. Aunque a veces requeire un poco de ensayo y error, una estrategia que suele funcionar es que las operaciones aritméticas se optimizan cuando declaramos los tipos. Llamadas a funciones como complex no nos dan esas garantías, porque pueden implicar conversiones entre tipos de datos.

```
sage: %cython
sage: from sage.matrix.constructor import matrix
sage: def mandelbrot_cy4(float x0,float  y0,float side,
...                      int N=200, int L=50, float R=3):
...         '''returns an array NxN to be plotted with matrix_plot
...         '''
...         cdef double complex c, z, I
...         cdef float delta
...         cdef int h, j, k
...         m=matrix(N,N)
...         I = complex(0,1)
...         delta = side/N
...         for j in range(N):
...             for k in range(N):
...                 c = (x0+j*delta)+ I*(y0+k*delta)
...                 z=0
...                 h=0
...                 while (h<L and
...                        z.real**2 + z.imag**2 < R*R):
...                     z=z*z+c
...                     h+=1
...                 m[j,k]=h
...         return m

sage: time m=mandelbrot_cy4(-2,-1.5,3,200,50)
Time: CPU 0.02 s, Wall: 0.03 s
```

La única parte interior a los bucles que queda por optimizar es la asignación:

```
m[j,k]=h
```

Para poder declarar arrays con tipos de datos tenemos dos opciones:

- Usar punteros como en C:

```
...
cdef int\* m = <int\*> sage_malloc((sizeof int)\*N^2)
...
```

lo que conlleva peligros potenciales si calculamos mal los tamaños, y no es muy conveniente para arrays bidimensionales.

- Sustituir la matriz de Sage (matrix) por un array de la librería **numpy** , bien integrada en cython.

Al definir el tipo de la matriz m, usamos un tipo de datos terminado en "_t", mientras que al llamar a una de las funciones que construyen arrays (como `zeros` , `ones` o `array`), pasamos el parámetro dtype, y no escribimos esa

terminación. En este ejemplo usamos enteros positivos de 16 bits para los valores de m, asumiendo que no trabajaremos con un L de más de 2^{16}.

```
cdef numpy.ndarray[numpy.uint16_t, ndim=2] m
m = numpy.zeros((N,N), dtype=numpy.uint16)

m[j,k]=h
```

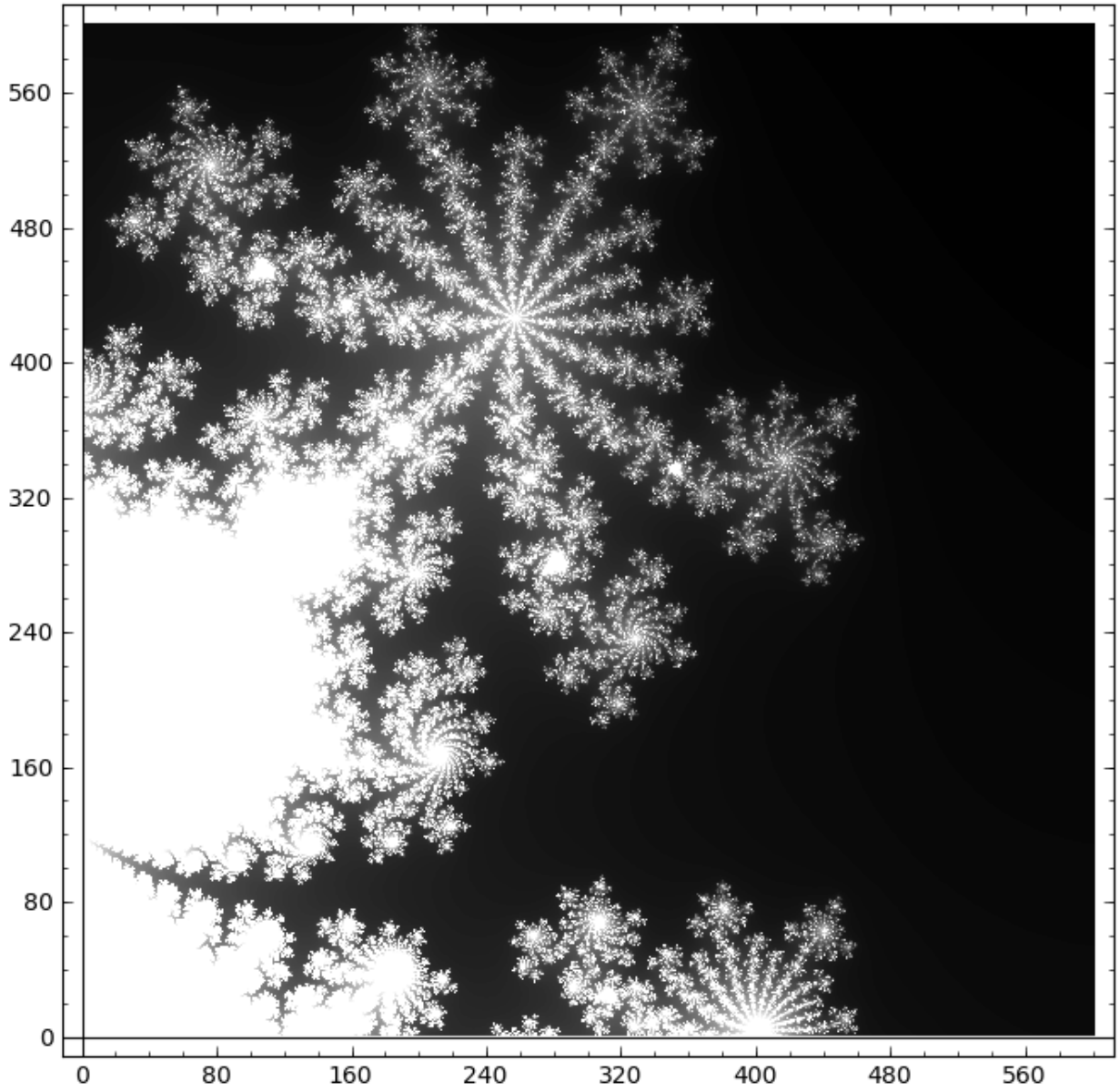
```
sage: %cython
sage: import numpy
sage: cimport numpy   #para declarar los tipos de los arrays
...                   #tb tenemos que usar cimport
sage: def mandelbrot_cy5(float x0,float  y0,float side,
...                      int N=200, int L=50, float R=3):
...       '''returns an array NxN to be plotted with matrix_plot
...       '''
...       cdef double complex c, z, I
...       cdef float delta
...       cdef int h, j, k
...       cdef numpy.ndarray[numpy.uint16_t, ndim=2] m
...       m = numpy.zeros((N,N), dtype=numpy.uint16)
...       I = complex(0,1)
...       delta = side/N
...       for j in range(N):
...           for k in range(N):
...               c = (x0+j*delta)+ I*(y0+k*delta)
...               z=0
...               h=0
...               while (h<L and
...                      z.real**2 + z.imag**2 < R*R):
...                   z=z*z+c
...                   h+=1
...               m[j,k]=h
...       return m

sage: time m=mandelbrot_cy5(-2,-1.5,3,200,50)
Time: CPU 0.01 s, Wall: 0.01 s
```

No tiene sentido continuar: hemos optimizado la parte interior a los bucles, y aunque aún se puede hacer el código más rápido, el precio a pagar no nos compensa.

Más información en el manual de cython:

http://docs.cython.org/

```
sage: time m=mandelbrot_cy5(-0.59375, 0.46875, 0.046875,600,160)
sage: matrix_plot(m).show(figsize=(8,8))
Time: CPU 0.10 s, Wall: 0.11 s
```



3.6 Ejercicios

3.6.1 1.

La conjetura de Goldbach afirma que todo número par se puede expresar como suma de dos primos.

El código de abajo pretende verificar la conjetura de Goldbach hasta un cierto número par K.

Tu objetivo es mejorar este código siguiendo las siguientes directrices:

- Identifica, de entre las tres funciones de abajo, la que más tiempo de cómputo consume, y por tanto la que más necesita nuestra atención. Justifica tu respuesta.
- Usando la hipótesis (trivial de demostrar, por otro lado), de que la criba de Eratóstenes tiene complejidad en O(N^2) (observa que decimos O, luego hablamos sólo de una cota superior), estima la complejidad del código

presentado.

- Mejora el código de arriba modificando el algoritmo hasta que su complejidad sea O(N^2). Trabaja *únicamente* en mejorar la función que has identificado antes.

```
sage: ##Criba
sage: def criba(ls):
...       '''Se queda con los elementos irreducibles de una lista de enteros'''
...       primos = []
...       while ls:
...           p = ls[0]
...           primos.append(p)
...           ls = [k for k in ls if k%p]
...       return primos
sage: def lista_primos(K):
...       'genera los numeros primos menores que K'
...       return criba(range(2,K))
sage: def goldbach(N):
...       for t in range(4,N,2):
...           comprobado = False
...           for x in lista_primos(N):
...               for y in lista_primos(N):
...                   if t == x+y:
...                       comprobado = True
...           if not comprobado:
...               return False #t es un contraejemplo
...       return True

sage: time goldbach(200)
```

3.6.2 2.

El objetivo de este ejercicio es calcular el tiempo de parada de la sucesión de Collatz comenzando por cada numero menor que M. El código de abajo calcula la longitud de la secuencia de Collatz partiendo de cada número entre 1 y M. Tus objetivos son:

- Modificar el algoritmo para recordar los valores ya calculados. Por ejemplo, si sabes que la longitud de la sucesión que comienza por 3 es 8, entonces la longitud de la sucesión que empieza por 6 es 9(=8+1), ya que el número siguiente a 6 es, y a partir de ese punto las sucesiones son iguales.

6->3->10->5->16->8->4->2->1

- Compila el código usando cython y declarando los tipos de las variables para acelerar el cálculo.

```
sage: def collatz(k):
...       if k%2:
...           return 3*k+1
...       else:
...           return k/2
sage: def tiempos_collatz(M):
...       tiempos = []
...       for j in range(1,M):
...           l = 1
...           k = j
...           while k!=1:
...               k = collatz(k)
...               l+=1
...           tiempos.append(l)
...       return tiempos
```

```
sage: time ts = tiempos_collatz(1000)
Time: CPU 0.39 s, Wall: 0.38 s

sage: point2d([(j,ts[j]) for j in range(len(ts))])
```

3.6.3 3.

http://es.wikipedia.org/wiki/Juego_de_la_vida

El "juego de la vida" se desarrolla en una malla formada por cuadrados ("células") que se extiende por el infinito en todas las direcciones. Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluyendo las diagonales. Las células tienen dos estados: están "vivas" o "muertas" (o "encendidas" y "apagadas"). El estado de la malla evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. Todas las células se actualizan simultáneamente.

Las transiciones dependen del número de células vecinas vivas:

- Una célula muerta con exactamente 3 células vecinas vivas "nace" (al turno siguiente estará viva).
- Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere o permanece muerta (por "soledad" o "superpoblación").

El código de debajo calcula la evolución de una matriz NxN con unos para las células vivas y ceros para las muertas. Tu objetivo es compilar el código en cython indicando los tipos de las variables y haciendo alguna otra optimización sencilla que se te ocurra para mejorar el código. Anota las mejora obtenidas contra un ejemplo tipo de tamaño respetable para observar qué cambios son más importantes.

```
sage: def cuenta_vecinos(m, j, k):
...       '''Cuenta el número de vecinos vivos de la casilla (j,k)'''
...       cuenta = 0
...       for j0, k0 in [(j - 1,k - 1), (j - 1,k), (j - 1,k + 1), (j,k - 1), (j,k + 1), (j
sage: + 1,k - 1), (j + 1,k), (j + 1,k + 1)]:
...           if 0<=j0<m.nrows() and 0<=k0<m.ncols():
...               cuenta += m[j0,k0]
...       return cuenta
sage: def paso(m):
...       F = m.nrows()
...       C = m.ncols()
...       nueva = matrix(F,C)
...       for j in range(F):
...           for k in range(C):
...               vecinos = cuenta_vecinos(m, j, k)
...               if vecinos == 3 or (m[j,k] and vecinos == 2):
...                   nueva[j,k] = 1
...               else:
...                   nueva[j,k] = 0
...       return nueva
sage: def gameoflife(matriz, generaciones):
...       for g in range(generaciones):
...           matriz = paso(matriz)
...       return matriz

sage: m=matrix(8,8)
sage: m[3,4]=1
sage: m[4,4]=1
sage: m[2,4]=1
sage: m[4,3]=1
```

```
sage: m[2,2]=1
sage: matrix_plot(m).show()
sage: m = gameoflife(m,1)
sage: matrix_plot(m).show()

sage: %time
sage: m=matrix(8,8)
sage: m[3,4]=1
sage: m[4,4]=1
sage: m[2,4]=1
sage: m[4,3]=1
sage: m[2,2]=1
sage: a = animate([matrix_plot(gameoflife(m,j)) for j in range(1,10)])
```

Bloque III: Álgebra

La primera sesión consiste de ejercicios de aritmética que se pueden resolver usando técnicas bastante elementales. Después aprendemos a operar con elementos de anillos de enteros y anillos de polinomios en Sage. Finalmente, aprendemos a hacer álgebra lineal sobre varios cuerpos de coeficientes y planteamos algunos problemas de formas bilineales.

4.1 Aritmética

En esta sesión vamos a trabajar con algunas funciones de Sage para trabajar con números (algunas ya las hemos usado), y a implementar algunos algoritmos clásicos de teoría de números.

Esta vez no vamos a separar la teoría de los ejercicios . En realidad, no vamos a usar funciones de Sage ni características de Sage que no hayamos visto antes. Debajo tienes varios temas de teoría de números, con una pequeña introducción y un recordatorio de algunas funciones que pueden ser útiles, y de varios ejercicios de dos grupos. El **primer grupo** consiste de **ejercicios básicos** que deberías intentar resolver en esta misma sesión. El **segundo grupo** consiste de ejercicios **para resolver en casa** . La siguiente sesión resolveremos buena parte de los ejercicios.

4.1.1 Dígitos de un número en una base arbitraria

Aunque hemos escrito nuestra propia función para recuperar los dígitos de un número en una base B arbitraria (o casi), también podemos usar el método `digits` , que tienen los enteros de Sage (`Integer`).

```
sage: a = 17
sage: print a.digits(base=2)
sage: print a.digits(base=10)
sage: print a.digits(base=3)
[1, 0, 0, 0, 1]
[7, 1]
[2, 2, 1]
```

Sin embargo, no he podido encontrar la función inversa.

Ejercicio

Escribe una función que acepta como argumentos:

- una lista de enteros (los dígitos)
- un entero (la base)

y devuelve el entero que tiene esa lista de dígitos en esa base.

Ejercicios para casa

1.

Escribe una función que devuelva True si un número es divisible por 9 usando el criterio que dice: "un número es divisible por 9 si y sólo si la suma de sus dígitos es divisible por 9". Si la suma de los dígitos es mayor que 9, puedes llamar a la función recursivamente para decidir si verifica el criterio. No debería usar el resto de la división (`%`) en ningún momento.

2.

El método `reverse` invierte los elementos de una lista.

```
lista = [2,4,7]
lista.reverse()
print lista
```

```
> [7,4,2]
```

Utiliza este método y las técnicas de este capítulo para construir:

- Una función que acepta un número y te devuelve otro número, con las mismas cifras que el anterior pero en orden inverso.
- Una función que acepta un número y devuelve un booleano: True si es **palíndromo** (se lee igual de derecha a izquierda que de derecha a izquierda) y False si no lo es.

3.

http://projecteuler.net/index.php?section=problems&id=36

Encuentra todos los números menores que un millón que son **palíndromos** *tanto en base 2 como en base 10* .

Por ejemplo, 585 se escribe 1001001001 en binario. Ambas expresiones se leen igual al derecho que al revés.

4.1.2 Números primos

- `is_prime(k)` : Comprueba si k es primo.
- `next_prime(k)` : Devuelve el menor número primo mayor que k.
- `prime_range(k)` : Lista de primos menores que k.
- `factor(k)` **:** factorización de k. `list( factor(k) )` devuelve una lista de tuplas con cada factor y su exponente correspondiente.

Ejercicios

Infinitos primos en cualquier combinación lineal

El teorema de Dirichlet dice que hay infinitos números primos en una sucesión del tipo:

$$x_j = a \cdot j + b$$

siempre que a y b sean números primos entre sí.

http://es.wikipedia.org/wiki/Teorema_de_Dirichlet

Dados tres números a, b y N, encuentra el menor número primo de la forma aj+b con j mayor que N:

Teorema del número primo

El teorema del número primo afirma que el siguiente límite existe y es igual a 1:

$$\lim_{x \to \infty} \frac{\pi(x)}{x/\ln(x)}$$

donde $\pi(x)$ es la cantidad de números primos menores que *x* y *ln(x)* es el logaritmo neperiano.

- Escribe un código que evalúe la función $\frac{\pi(x)}{x\, ln(x)}$ para un número x cualquiera.
- Encuentra un número *x* tal que el límite diste de 1 menos de 0.1

Ejercicios para casa

1.

Halla la secuencia más larga de números consecutivos menores que un millón que no contiene ningún primo.

2.

La función ϕ de Euler se puede calcular utilizando la factorización del número. Si $k = p_1^{e_1} \dots p_k^{e_k}$, tenemos:

$$\phi(k) = \prod (p_j^{e_j} - p_j^{e_j - 1})$$

Utiliza el método `factor` aplicado a números enteros y la fórmula de arriba para calcular el valor de la función ϕ de Euler:

- Compara el resultado con el obtenido usando la regla "$\phi(k)$ es la cantidad de números menores que k que son primos relativos con k", o bien con alguna función de Sage que haga la misma tarea.

4.1.3 Algoritmo de Euclides

El algoritmo de euclides calcula el **máximo común divisor** (**mcd**) de dos números naturales n y m. El algoritmo se basa en el siguiente principio: el mcd de dos números n y m, con n<m, es también el mcd de n y m %n (el resto de dividir m por n). De esta forma, hemos reducido el problema a otro con números menores. Eventualmente, uno de los dos números será 0, y entonces sabemos que mcd(0,m)=m.

Ejercicios

1.

Escribe una función que calcule el máximo común divisor de dos números siguiendo el algoritmo de Euclides.

2.

Escribe una función que acepte una lista de números como argumento y devuelva el máximo común divisor de los números de la lista.

Ejercicio para casa

Escribe una función que calcule el máximo común divisor de dos números calculando la factorización de cada uno de ellos y después escogiendo los factores comunes con el menor exponente.

4.1.4 Identidad de Bezout

Una identidad de Bézout muestra explícitamente el mcd d de m y n como una combinación lineal de m y n con coeficientes enteros:

$$d = u\,m + v\,n$$

La función `xgcd` de SAGE implementa el algoritmo extendido de Euclides, que devuelve una tupla con el mcd y los coeficientes de una identidad de Bézout:

```
(d,u,v)=xgcd(m,n)
```

```
sage: m=15
sage: n=21
sage: (d,u,v)=xgcd(m,n)
sage: print ' %d=(%d)*%d+(%d)*%d' %(d,u,m,v,n)
3=(3)*15+(-2)*21
```

Identidad de Bézout con más de dos números.

También podemos encontrar una identidad del tipo

$$d = \sum_{j=1}^{N} u_j m_j$$

donde d es el máximo divisor común a todos los números m_j. De hecho, el proceso para encontrarlo se reduce al anterior, usando inducción. Como ya sabemos resolver el caso *N=2* , sólo tenemos que aprender a reducir el caso de *N+1* números al caso anterior.

Para unos números $m = m_1, \ldots, m_{N+1}$, comenzamos por encontrar una identidad de Bezout para los dos últimos:

$$d_N = v\,m_N + w\,m_{N+1}$$

Después aplicamos la hipótesis de inducción a $m_1, \ldots, m_{N-1}, d_N$:

$$d = \sum_{j=1}^{N-1} u_j m_j + u_N d_N$$

El mcd de $m_1, \ldots, m_{N-1}, d_N$ es también el de $m_1, \ldots, m_{N+1}$, así que sólo tenemos que sustituir:

$$d = \sum_{j=1}^{N-1} u_j m_j + u_N (v\, m_N + w\, m_{N+1})$$

Ejercicio

Escribe una función que acepta como argumento una lista de números m_j y devuelve una lista que contiene en primer lugar el máximo común divisor de todos los elementos de la lista, seguido de los ceficientes u_j que realizan la identidad:

$$d = \sum_{j=1}^{N} u_j m_j$$

4.1.5 Teorema chino del resto

Con la identidad de Bézout podemos hacer funcionar el teorema chino del resto:

Si los números *m* y *n* son primos entre sí, entonces para cualesquiera *a<m* , *b<n* existe un único número *c* menor que *n*m* tal que el resto de dividir *c* entre *m* es *a* y el resto de dividir *c* entre *n* es **b** .

En lenguaje de congruencias, escribimos $x \equiv a(mod\ m)$ para decir que el resto de dividir *a* por *m* es el mismo que el resto de dividir *x* por *m* . El teorema chino del resto se escribe entonces:

$$x \equiv a(mod\ m), x \equiv b(mod\ n) \iff x \equiv c(mod\ nm)$$

Se puede comprobar que podemos obtener *c* de la fórmula:

$$c = bmv + anu$$

donde u y v vienen de una identidad de Bézout:

$$1 = nu + mv$$

Ejercicio

- Escribe una función que, dados a, m, b y n, devuelva c.
- Generaliza el resultado a varias congruencias $x \equiv a_i(mod\ m_i)$, para $i = 1 \ldots N$, donde todos los números m_i son primos entre sí, y escribe una función que acepte una lista con los números m_i y otra con los números a_i y devuelva un número c tal que:

$$x \equiv a_i (mod\ m_i) \forall i \iff x \equiv c (mod\ m_1 \cdot \cdots \cdot m_N)$$

4.2 Grupos y Anillos

Sage tiene definidos un buen número de anillos, grupos y otras estructuras algebraicas.

Podemos operar con elementos que representan elementos de un anillo o un espacio vectorial, por ejemplo, además de con objetos que representan anillos, grupos, subgrupos, subespacios vectoriales y otras estructuras de nivel más alto.

Muchos anillos comunes están definidos en Sage:

- `ZZ` : $\mathbb{Z}$
- `Integers(m)` : $\mathbb{Z}_m$ (ó $\mathbb{Z}/m\mathbb{Z}$)
- `QQ` : $\mathbb{Q}$
- `RR` : $\mathbb{R}$, representados por números reales de doble precisión
- `CC` : $\mathbb{C}$, representados por números reales de doble precisión

Además, podemos usar otros constructores para definir anillos derivados, como el constructor de anillos de polinomios `PolynomialRing` que veremos en detalle más abajo.

```
sage: R1 = Integers(7)
sage: R2 = Integers(21)

sage: a = ZZ(3)
sage: b = R1(3)
sage: c = R2(3)
sage: print a, a.parent()
sage: print b, b.parent()
sage: print c, c.parent()
3 Integer Ring
3 Ring of integers modulo 7
3 Ring of integers modulo 21

sage: #Al calcular el inverso, Sage puede devolver el inverso
sage: #en el cuerpo de fracciones del anillo(en QQ en vez de ZZ)
sage: print a, 1/a, (1/a).parent()
sage: #Si el anillo es un cuerpo, obtenemos un elemento del mismo anillo
sage: print b, 1/b, (1/b).parent()
sage: #Si el anillo no es dominio de integridad, algunos elementos
sage: #no tienen inversos (en ningún cuerpo que contenga al anillo)
sage: print c, 1/c, (1/c).parent()
3 1/3 Rational Field
3 5 Ring of integers modulo 7
3
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
```

4.2.1 Trabajar módulo m vs trabajar en `Integers(m)`

En el caso de $\mathbb{Z}_m$, podemos trabajar de forma más abstracta con elementos de `Integers(m)` ó bien a nivel más bajo usando números enteros normales, pero tomando restos módulo m. Estudiar los pasos necesarios en detalle es ilustrativo.

Suma y producto módulo *m*

Para hacer sumas y productos sobre clases de equivalencia, podemos usar la suma y el producto habituales de números enteros, y tomar el resto de dividir por *m* :

```
m=31
a=12
b=23
s=(a+b)%m
p=(a*b)%m
```

Potencia módulo *m*

Aunque podemos calcular la clase de congruencia de $a^p (mod\ m)$ calculando el entero a^p y luego tomando el resto módulo *m* , debemos tener en cuenta que a^p puede ser un número muy grande, y el ordenador puede dedicar al cálculo demasiado tiempo y memoria. En este caso, compensa reducir el número módulo m después de cada producto:

```
sage: def potencia1(x, n, m):
...       if n == 0:
...           return 1
...       elif n%2 == 0:
...           y = potencia1(x, n/2, m)
...           return (y*y)%m
...       else:
...           y = potencia1(x, (n-1)/2, m)
...           return (x*y*y)%m

sage: potencia1(3,8,10)
1
```

También podemos usar la función `power_mod`:

```
sage: power_mod(3,8,10)
1
```

El algoritmo correspondiente para Integers(m) es válido para cualquier anillo con unidad (el 1 que aparece en el código se transformará en la unidad del anillo).

```
sage: def potencia(x, n):
...       if n == 0:
...           return 1
...       elif n%2 == 0:
...           y = potencia(x, n/2)
...           return y*y
...       else:
...           y = potencia(x, (n-1)/2)
...           return x*y*y

sage: a = ZZ(3)
sage: b = R1(3)
```

```
sage: c = R2(3)
sage: print potencia(a, 8)
sage: print potencia(b, 8)
sage: print potencia(c, 8)
6561
2
9
```

El inverso de a módulo m, es decir, la clase b tal que $a \cdot b \equiv 1(mod\ m)$, no se puede reducir a una operación de aritmética usual. Una llamada a la función inverse_mod(a,m) devuelve el inverso de a módulo m (este número se calcula usando los coeficientes de una identidad de Bézout). La operación equivalente en `Integers(m)` es `1/a` .

```
sage: inverse_mod(3,7)
5

sage: R1 = Integers(7)
sage: b = R1(3)
sage: 1/b
5
```

4.2.2 Anillos de Polinomios

La sintaxis para definir un anillo de polinomios con coeficientes en otro anillo `R` es:

```
Pols.<t> = PolynomialRing(R)
```

ó

```
Pols.<t> = R['t']
```

donde hemos definido el anillo de polinomios `Pols` con coeficientes en el anillo `R` y la variable independiente `t` .

```
sage: #Definimos varios anillos de polinomios con coeficientes en distintos anillos
sage: PR1.<t> = PolynomialRing(ZZ)
sage: PR2.<x> = PolynomialRing(QQ)
sage: PR3.<y> = PolynomialRing(RR)
sage: PR4.<z> = PolynomialRing(CC)
sage: R1 = Integers(7)
sage: PR5.<u> = PolynomialRing(R1)
```

Una vez hemos definido el anillo, podemos usar la variable independiente para definir polinomios, operando con ella como una variable más.

```
sage: p=t^2-1
sage: q=x^2-1

sage: p.base_ring()
Integer Ring

sage: q.base_ring()
Rational Field
```

La variable `p` tiene tipo de datos "Polinomio con coeficientes enteros".

La variable `q` tiene tipo de datos "Polinomio con coeficientes racionales".

```
sage: p?
<html>...</html>
```

```
sage: q?
<html>...</html>
```

Podemos evaluar un polinomio en un punto con la sintaxis

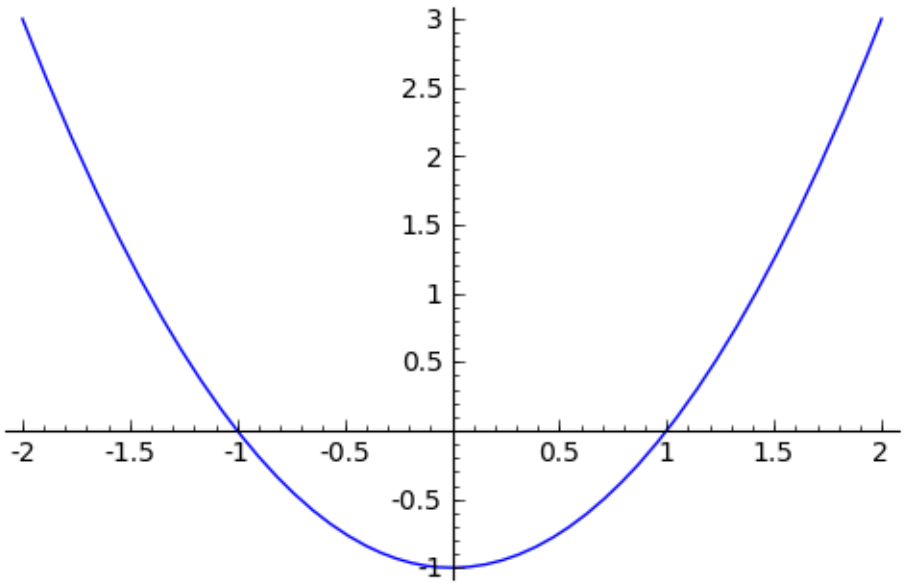
```
p(var=expresion)
```

donde `var` es la variable independiente del polinomio y `expresion` es cualquier expresión en SAGE que dé como resultado un elemento del anillo.

```
sage: print p(t=2)
sage: print q(x=1+2)
sage: numero=0
sage: print q(x=numero)
sage: print [q(x=j) for j in range(-3,4)]
3
8
-1
[8, 3, 0, -1, 0, 3, 8]
```

Para dibujar la gráfica de un polinomio usamos el comando `plot` .

```
sage: #Dibuja el polinomio entre -2 y 2
sage: q.plot(-2,2)
```



4.2.3 Raíces de polinomios

Una raíz de un polinomio $p \in R[t]$ es un elemento *x0* del anillo *R* tal que *p(x0)=0* . El teorema fundamental del álgebra afirma que un polinomio de grado *n* con coeficientes complejos tiene exactamente *n* raíces complejas. Sin embargo, un polinomio con coeficientes enteros puede tener raíces que son números racionales, reales algebraicos, o complejos.

El método `roots` se puede llamar desde cualquier polinomio, y devuelve una lista de tuplas que contienen las raíces del polinomio con sus multiplicidades. Atención, si las raíces de un polinomio no están en el anillo de sus coeficientes, el método `roots` no las devuelve, y tenemos que llamar a `real_roots` o `complex_roots` .

```
sage: p.roots()
[(1, 1), (-1, 1)]

sage: p2=x^2-2

sage: p2.roots()
[]

sage: p2.real_roots()
[-1.41421356237310, 1.41421356237310]

sage: p3=x^2+1

sage: p3.roots()
[]

sage: p3.real_roots()
[]

sage: p3.complex_roots()
[-1.00000000000000*I, 1.00000000000000*I]

sage: r=z^2+1

sage: r.roots()
[(-1.00000000000000*I, 1), (1.00000000000000*I, 1)]
```

El método `roots` devuelve sólo las raíces enteras, y `real_roots` devuelve todas las raíces reales, y además lo hace de forma numérica. Sin embargo, SAGE tiene una alternativa a nuestro método raices: extender el polinomio a un polinomio con coeficientes racionales, y entonces el método `roots` devuelve las raíces racionales.

```
sage: s1 = 2*t-1
sage: print s1.roots()
[]

sage: qs1 = s1.base_extend(QQ)
sage: print qs1.roots()
[(1/2, 1)]

sage: s1.real_roots()
[0.500000000000000]
```

En un anillo como $\mathbb{Z}_m[u]$, se aplica la misma definición de raíz del polinomio.

```
sage: r = u^2+3

sage: r.roots()
[(5, 1), (2, 1)]

sage: #evaluamos r en todos los elementos de Z_7
sage: print [(j, r(u=j)) for j in srange(7)]
[(0, 3), (1, 4), (2, 0), (3, 5), (4, 5), (5, 0), (6, 4)]
```

4.2.4 Factorización de polinomios

Si un polinomio con coeficientes en $\mathbb{Z}$, $\mathbb{Q}$ ó $\mathbb{R}$ no tiene tantas raíces (contando multiplicidades) como su grado, no se puede escribir como producto de factores lineales. Aun así, se puede escribir como producto de factores irreducibles (es decir, factores que no se pueden expresar como producto de polinomios de menor grado):

- `factor(q)` ó `q.factor()` : la factorización del polinomio
- `list(q.factor())` : lista de tuplas (factor, multiplicidad)
- `q.is_irreducible()` : True si y sólo si q es irreducible.

```
sage: q=t^3 - 3*t^2 + 4
sage: print q.factor()
sage: print list(q.factor())
(t + 1) * (t - 2)^2
[(t + 1, 1), (t - 2, 2)]

sage: print q, q.is_irreducible()
sage: print q+1, (q+1).is_irreducible()
t^3 - 3*t^2 + 4 False
t^3 - 3*t^2 + 5 True
```

Lista de coeficientes

```
list(polinomio)
```

devuelve una lista con los coeficientes de `polinomio` , ordenados de menor a mayor grado.

```
sage: s1=2*z^2-3*z+1
sage: print s1
2.00000000000000*z^2 - 3.00000000000000*z + 1.00000000000000

sage: coefs = list(s1)
sage: print coefs
[1.00000000000000, -3.00000000000000, 2.00000000000000]
```

Ejercicio : Reconstruye el polinomio a partir de los coeficientes.

Ejemplo: [1,-3,3] -> $3t^2 - 3t + 1$

4.3 Ejercicios

4.3.1 1.

El teorema de Euler afirma que para un número *a* y un número *m* , se tiene:

$$a^{\varphi(m)} \equiv 1(mod\ m)$$

para la función φ de Euler, siempre que *a* y *m* sean primos entre sí.

- Escribe un programa que compruebe el teorema para los números m menores que 1000 y todos los números a<m.
- ¿Es cierto que si se verifica la igualdad, entonces necesariamente *a* y *m* son primos entre sí? Busca un contraejemplo, o verífica la propiedad para todos los números *m* menores que 1000.
- ¿Es $\varphi(m)$ siempre el menor número posible que verifica esta identidad para todos los números *a* que son primos relativos con *m* ? Busca un contraejemplo, o verífica la propiedad para todos los números *m* menores que 1000.
- Dados dos números *a* y *m* , encuentra el menor número positivo *k* tal que $a^k \equiv 1(mod\ m)$

- Dado un número *m* , encuentra el menor número positivo *k* tal que $a^k \equiv 1(mod\ m)$ *para todo a que sea primo relativo con m* . Llamemos $\lambda(m)$ al número k así obtenido.

4.3.2 2.

Como vimos en el ejercicio anterior, $\lambda(m)$ no siempre coincide con $\varphi(m)$. Para cada una de las preguntas siguientes, busca un contraejemplo, o verifica la propiedad para todos los números menores que 1000.

- ¿Coinciden siempre para los números $m = p^j$ cuando *p* es primo?
- ¿Coinciden siempre para los números $m = p^j$ cuando *p* es primo *impar* ?
- ¿Coinciden siempre para los números $m = p^j\ q^k$ cuando *p* y *q* son primos impares?

4.3.3 3. Ejercicio opcional

- Intenta relacionar $\lambda(p^j\ q^k)$ con los valores de $\lambda(p^j)$ y $\lambda(q^k)$.
- Conjetura una fórmula que expresa $\lambda(p_1^{j_1} \dots p_L^{j_L})$ en función de $\lambda(p_1^{j_1})$, $\lambda(p_2^{j_2})$,... hasta $\lambda(p_L^{j_L})$.

4.3.4 4. Raíces racionales de polinomios con coeficientes enteros

> Comprueba el criterio que aprendimos en secundaria sobre las raíces racionales de un polinomio con coeficientes enteros:

Cualquier raíz racional de un polinomio con coeficientes enteros tiene un numerador que divide al término independiente y un denominador que divide al coeficiente del término de mayor grado.

- Escribe código que busque las raíces de un polinomio con coeficientes enteros según este criterio
- Escribe código que compruebe que el resultado coincide con el resultado de llamar al método roots, para un conjunto de polinomios que te parezca lo bastante significativo.

4.3.5 5.

Investiga la siguiente pregunta:

Dado un polinomio *p* con coeficientes enteros, ¿para cuántos valores de *k* el polinomio *p+k* es reducible? ¿para cuántos valores tiene al menos una raíz (entera)?

- Escribe una función que acepte como argumentos un polinomio *p* con coeficientes enteros y dos enteros *k1* y *k2* , y devuelva en una lista todos los enteros *j* entre *k1* y *k2* tal que *p+j* tiene alguna raíz entera.
- Escribe una función que acepte como argumentos un polinomio *p* con coeficientes enteros y dos enteros *k1* y *k2* , y devuelva en una lista todos los enteros *j* entre *k1* y *k2* tal que *p+j* es reducible sobre $\mathbb{Q}$.
- ¿Te parece razonable la siguiente conjetura?: "La cantidad de enteros *j* en el intervalo [-K, K] tal que *p+j* tiene alguna raíz entera crece como $O(K^{1/n})$, donde *n* es el grado del polinomio *p* ". Medita un momento al respecto.
- Argumenta con datos empíricos si la conjetura análoga para polinomios reducibles es razonable, o no, estimando el ratio entre los polinomios que verifican una u otra condición.

4.4 Álgebra lineal

4.4.1 Espacios vectoriales

En Sage podemos trabajar con distintos cuerpos para los coeficientes de los espacios vectoriales. Las elecciones más usuales son:

- `QQ` (o `RationalField()`) el cuerpo de los números racionales.
- `QQbar` (o `AlgebraicField()`), representa la clausura algebraica $\bar{\mathbb{Q}}$ del cuerpo de los números racionales.
- `RDF` (o `RealDoubleField()`), números reales de 64 bits.
- `CDF` (o `ComplexDoubleField()`), números complejos de 64 bits.
- `SR` , o expresiones simbólicas. Cualquier expresión algebraica que contenga símbolos como pi, I, sqrt(2) pertenece a este anillo.

Los cuerpos $\mathbb{Q}$ y $\bar{\mathbb{Q}}$ son computables, y sus implementaciones en Sage son **exactas** (en la documentación de Sage: "exact field"), es decir, los cálculos no acumulan errores de redondeo. Sin embargo, los tres últimos cuerpos no tienen aritmética exacta, debido a los errores de redondeo. Esto significa que en estos cuerpos no se pueden hacer cálculos como la multiplicidad de las raíces de un polinomio o la forma de Jordan de una matriz, que sean inestables ante la aparición de errores numéricos.

```
sage: V1 = VectorSpace(QQ,3)
sage: V2 = VectorSpace(RDF,3)     #Numeros reales de precision doble
sage: V3 = VectorSpace(CDF,4)     #Numeros complejos de precision doble
sage: print V1
sage: print V2
sage: print V3
Vector space of dimension 3 over Rational Field
Vector space of dimension 3 over Real Double Field
Vector space of dimension 4 over Complex Double Field
```

Vectores

Los vectores pertenecen a un espacio vectorial concreto, aunque SAGE hará las conversiones necesarias entre tipos de datos si queremos hacer operaciones entre espacios vectoriales compatibles.

```
sage: #O tambien
sage: v1 = V1([1,1,1])
sage: v2 = V2([1,1,0])
sage: v3 = 2*v1+v2
sage: print v1 ,v1.parent()
sage: print v2 ,v2.parent()
sage: print v3 ,v3.parent()
(1, 1, 1) Vector space of dimension 3 over Rational Field
(1.0, 1.0, 0.0) Vector space of dimension 3 over Real Double Field
(3.0, 3.0, 2.0) Vector space of dimension 3 over Real Double Field

sage: #pero...
sage: v1 = V1([1,1])
Traceback (most recent call last):
...
TypeError: entries must be a list of length 3
```

```
sage: #ni tampoco...
sage: v1 = V2([1,1,1])
sage: v2 = V3([1,1,0,0])
sage: v3 = v1+v2
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Vector space of dimension 3 over Real Double Field
```

También podemos usar `vector` sin indicar el espacio vectorial, y entonces Sage escogerá como espacio ambiente el anillo de coeficientes más pequeño que contenga a las entradas. Si los coeficientes son enteros, como $\mathbb{Z}$ no es un cuerpo, en vez de "espacio vectorial", Sage habla de "modulo", pero no es necesario preocuparnos por esa distinción.

```
sage: v1 = vector([1,1,1])
sage: v2 = vector([1/2,1,1])
sage: v3 = vector([1.0,1,0])
sage: print v1 ,v1.parent()
sage: print v2 ,v2.parent()
sage: print v3 ,v3.parent()
(1, 1, 1) Ambient free module of rank 3 over the principal ideal domain Integer Ring
(1/2, 1, 1) Vector space of dimension 3 over Rational Field
(1.00000000000000, 1.00000000000000, 0.000000000000000) Vector space of dimension 3 over Real Field w
```

Subespacios vectoriales

Podemos definir fácilmente el subespacio engrendrado por un conjunto de vectores, y después hacer operaciones como intersección o suma de subespacios, o comprobaciones como igualdad o inclusión de subespacios.

```
sage: v1 = vector([1,1,1])
sage: v2 = vector([1,1,0])
sage: v3 = vector([1,0,1])
sage: #Hay que fijarse en el cuerpo de coeficientes
sage: L1 = V1.subspace([v1,v2])
sage: L1_bis = V2.subspace([v1,v2])
sage: print L1
sage: print L1_bis
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 1 0]
[0 0 1]
Vector space of degree 3 and dimension 2 over Real Double Field
Basis matrix:
[1.0 1.0 0.0]
[0.0 0.0 1.0]
```

El grado (degree) al que se refiere arriba, es la dimensión del espacio ambiente. Podemos recuperar este espacio directamente con el método `ambient_vector_space` .

```
sage: #Dimension
sage: print dim(L1)
2

sage: #Grado
sage: print L1.degree()
3

sage: L1.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

Muchos operadores actúan sobre subespacios vectoriales, con los significados habituales. Cuando el símbolo usual para la operación no está en el teclado (como el operador de intersección $\cap$), lo normal es usar un método (como el método `intersection`).

```
sage: #Pertenencia al subespacio
sage: print v3 in L1
sage: v4 = vector([4,4,3])
sage: print v4 in L1
False
True

sage: #Comprobacion de igualdad
sage: print L1 == V1
sage: print L1 == V1.subspace([v1,v1+v2])
False
True

sage: #Comprobacion de inclusion
sage: print L1 <= V1
sage: print L1 >= V1
sage: print L1 >= V1.subspace([v1])
True
False
True
```

Ejercicio resuelto

Definimos dos subespacios de $\mathbf{Q}^3$:

$$L_1 = \langle (1,1,0), (0,0,1) \rangle$$

$$L_2 = \langle (1,0,1), (0,1,0) \rangle$$

- Encuentra bases de los subespacios $L_1 \cap L_2$ y $L_1 + L_2$

```
sage: v1 = vector([1,1,0])
sage: v2 = vector([0,0,1])
sage: v3 = vector([1,0,1])
sage: v4 = vector([0,1,0])
sage: L1 = V1.subspace([v1,v2])
sage: L2 = V1.subspace([v3,v4])
sage: #Interseccion de subespacios
sage: L3 = L1.intersection(L2)
sage: print 'Intersección'
sage: print L3
sage: print
sage: #Suma de subespacios
sage: L4 = L1 + L2
sage: print 'Suma'
sage: print L4
sage: print L4 == V1
Intersección
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
```

```
[1 1 1]

Suma
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
True
```

Bases de los espacios vectoriales

Como hemos visto, al definir subespacios mediante generadores, se construye una base escalonada del subespacio a partir de los generadores. Para imponer la base del espacio o subespacio, usamos el comando `subspace_with_basis`.

```
sage: v1 = vector([1,1,0])
sage: v2 = vector([1,0,1])
sage: L1 = V1.subspace([v1,v2])
sage: print L1
sage: print L1.basis()
sage: print L1.basis_matrix()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  1]
[ 0  1 -1]
[
(1, 0, 1),
(0, 1, -1)
]
[ 1  0  1]
[ 0  1 -1]

sage: L1 = V1.subspace([v1,v2])
sage: L2 = V1.subspace([v1,v1+v2])
sage: print L1.basis()
sage: print L2.basis()
[
(1, 0, 1),
(0, 1, -1)
]
[
(1, 0, 1),
(0, 1, -1)
]

sage: L3 = V1.subspace_with_basis([v1,v2])
sage: print L1
sage: print L3
sage: #A pesar de tener distintas bases, ambos subespacios se declaran iguales
sage: print L1 == L3
sage: print L3.basis_matrix()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  1]
[ 0  1 -1]
Vector space of degree 3 and dimension 2 over Rational Field
```

```
User basis matrix:
[1 1 0]
[1 0 1]
True
[1 1 0]
[1 0 1]
```

El método `coordinates` nos da las coordenadas de un vector en la base del espacio.

```
sage: print L1.coordinates(2*v1+3*v2)
sage: print L2.coordinates(2*v1+3*v2)
sage: print L3.coordinates(2*v1+3*v2)
[5, 2]
[5, 2]
[2, 3]
```

4.4.2 Matrices

Para crear una matriz manualmente en SAGE, llamamos a `matrix` con el anillo de coeficientes como primer argumento. Después introducimos los datos usando una de las dos formas siguientes:

- Como una lista de listas, cada una de las cuales contiene una fila de la matriz:

```
#Matriz 2x4 con coeficientes en Q
M = matrix(QQ,[[0,1,0,0],[1,0,0,0]])
```

```
- Pasando el número de filas, el de columnas, y una sóla lista con todos los elementos:

M = matrix(QQ, K1, K2, lista)
```

donde K1 es el número de filas y K2 el número de columnas y, en vez de pasar una lista con las filas, pasamos una sóla lista con K1xK2 elementos:

```
matrix(QQ, 2, 4, [1,2,3,4,5,6,7,8])

sage: M1 = matrix(QQ,[[1,2,3,4],[4,2,3,1]])
sage: M2 = matrix(QQ,3,3,[3,2,1,  1,2,3,  2,2,2])
sage: show(M1)
sage: show(M2)
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 3 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 2 & 1 \\ 1 & 2 & 3 \\ 2 & 2 & 2 \end{pmatrix}$$

Métodos de las matrices

Aparte de los cálculos habituales con matrices, como determinante o rango, hay otros métodos interesantes que conectan con lo visto antes. Mencionamos tres, pero recuerda que puedes ver una lista completa escribiendo el nombre de tu matriz seguido de un punto y pulsando el tabulador:

- `M.kernel()` : Núcleo de una matriz, visto como un subespacio vectorial (pon atención a la diferencia entre el kernel por la izquierda `M.left_kernel()` o por la derecha `M.right_kernel()` ; el método `kernel` a secas se corresponde con `left_kernel`)
- `M.image()` : Imagen de una matriz, como subespacio vectorial
- `M.echelon_form()` : forma escalonada

```
sage: M1 = matrix(QQ,[[1,2,3,4],[4,2,3,1]])
sage: show(M1)
sage: print M1.kernel()    #lo mismo que M1.left_kernel()
sage: print
sage: print M1.image()
sage: show( M1.echelon_form())
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]

Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[  1   0   0  -1]
[  0   1 3/2 5/2]
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 3 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & \frac{3}{2} & \frac{5}{2} \end{pmatrix}$$

```
sage: M2 = matrix(QQ,3,3,[3,2,1,  5,2,1,  4,2,1])
sage: show(M2)
sage: print M2.kernel()
sage: print
sage: print M2.image()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1  1 -2]

Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[  1   0   0]
[  0   1 1/2]
```

$$\begin{pmatrix} 3 & 2 & 1 \\ 5 & 2 & 1 \\ 4 & 2 & 1 \end{pmatrix}$$

```
sage: #Una matriz invertible 4x4
sage: M3 = matrix(CDF, [[ 2,  0, -2,  2], [ 1,  3,  2, -3], [ 5,  1,  8, -3], [ 5,  1,  4,  1]])
sage: show(M3)
sage: print M3.kernel()
sage: print
sage: print M3.image()
Vector space of degree 4 and dimension 0 over Complex Double Field
Basis matrix:
[]

Vector space of degree 4 and dimension 4 over Complex Double Field
Basis matrix:
[1.0   0   0   0]
[  0 1.0   0   0]
[  0   0 1.0   0]
[  0   0   0 1.0]
```

$$\begin{pmatrix} 2{,}0 & 0 & -2{,}0 & 2{,}0 \\ 1{,}0 & 3{,}0 & 2{,}0 & -3{,}0 \\ 5{,}0 & 1{,}0 & 8{,}0 & -3{,}0 \\ 5{,}0 & 1{,}0 & 4{,}0 & 1{,}0 \end{pmatrix}$$

Suma y productos de matrices

La suma de matrices y productos por vectores se comporta de la manera previsible, lanzando errores si las dimensiones no casan, y cambiando los tipos de datos según sea necesario para hacer operaciones cuyos operandos tienen coeficientes en cuerpos distintos, pero compatibles.

```
sage: M4 = identity_matrix(QQ, 4)
sage: show(M4)
sage: show(3*M4 + M3)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 5{,}0 & 0 & -2{,}0 & 2{,}0 \\ 1{,}0 & 6{,}0 & 2{,}0 & -3{,}0 \\ 5{,}0 & 1{,}0 & 11{,}0 & -3{,}0 \\ 5{,}0 & 1{,}0 & 4{,}0 & 4{,}0 \end{pmatrix}$$

```
sage: v = vector(CDF,[-1, 0.5, -0.5, 2 ])
sage: print M3*v
sage: print M3*v + 2*v
(3.0, -6.5, -14.5, -4.5)
(1.0, -5.5, -15.5, -0.5)

sage: #Ejemplo: dos matrices cuadradas que no conmutan
sage: M1 = matrix(QQ,2,2,[1,1,0,1])
sage: M2 = matrix(QQ,2,2,[2,0,0,1])
sage: show(M1*M2)
sage: show(M2*M1)
```

$$\begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 2 \\ 0 & 1 \end{pmatrix}$$

4.4.3 Obtener el subespacio dado por ecuaciones

Si tenemos un espacio dado por ecuaciones, basta poner los coeficientes de las ecuaciones en una matriz, y podemos obtener el subespacio que define como el conjunto de vectores en los que todas las ecuaciones se anulan (es decir el núcleo). Por ejemplo, escribimos las ecuaciones

$$\begin{aligned} x_1 + 2x_2 &= 0 \\ -x_1 + x_3 &= 0 \end{aligned}$$

en forma matricial:

$$\begin{pmatrix} 1 & 2 & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

Al haber elegido esta forma de escribir la matriz, el espacio que nos interesa es el kernel por la derecha (`right_kernel`)

```
sage: M = matrix(QQ,2,3,[1,2,0, -1,0,1])
sage: M.right_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2    1]
```

4.4.4 Obtener las ecuaciones de un subespacio

Si tenemos un subespacio, podemos obtener unas ecuaciones de esta forma:

- el método `basis_matrix()` devuelve una matriz que contiene los vectores de una base
- las ecuaciones del subespacio son los covectores que se anulan en esos vectores (es decir, el núcleo de la matriz anterior), que obtenemos con `right_kernel()`
- Una base del núcleo nos da un conjunto minimal de ecuaciones que define el subespacio.

Ejercicio resuelto

Encuentra las ecuaciones del subespacio de $\mathbb{C}^4$ engendrado por $[1,1,1,1]$ y $[1,1,0,0]$

```
sage: v1 = vector([1,1,1,1])
sage: v2 = vector([1,1,0,0])
sage: L1 = V3.subspace([v1,v2])

sage: #Ecuaciones del subespacio
sage: M = L1.basis_matrix()
sage: K = M.right_kernel()
sage: print K.basis()
[
(1.0, -1.0, 0, 0),
(0, 0, 1.0, -1.0)
]
```

Ejercicio resuelto

Encuentra una base del subespacio de $\mathbb{C}^4$ dado por $x_1 + 2x_2 - x_4$ y una base de su intersección con el subespacio engendrado por $[1,1,1,1]$ y $[1,1,0,0]$.

```
sage: #Subespacio dado por x1 + 2*x2 -x4 en V(CDF,4)
sage: M = matrix(CDF,4,1,[1,2,0,-1])
sage: show(M)
sage: L2 = M.left_kernel()
sage: print L2
Vector space of degree 4 and dimension 3 over Complex Double Field
Basis matrix:
[1.0   0   0 1.0]
[  0 1.0   0 2.0]
[  0   0 1.0   0]
```

$$\begin{pmatrix} 1{,}0 \\ 2{,}0 \\ 0 \\ -1{,}0 \end{pmatrix}$$

```
sage: L1.intersection(L2)
Vector space of degree 4 and dimension 1 over Complex Double Field
Basis matrix:
[1.0 1.0 3.0 3.0]
```

4.4.5 Autovalores, autoespacios, forma de Jordan

Como indicamos arriba, la forma de Jordan es inestable antes pequeños errores. Esto hace imposible calcular la forma de Jordan usando aritmética aproximada con errores de redondeo. La elección del cuerpo de coeficientes se vuelve por tanto muy importante.

```
sage: M = matrix(QQ,3,3,[0,1,0,1,0,0,0,0,1])
sage: print M
sage: print M.eigenvalues()
sage: show( M.eigenvectors_left())
sage: print M.jordan_form()
[0 1 0]
[1 0 0]
[0 0 1]
[-1, 1, 1]
[-1| 0| 0]
[--+--+--]
[ 0| 1| 0]
[--+--+--]
[ 0| 0| 1]
```

$$[(-1, [(1, -1, 0)], 1), (1, [(1, 1, 0), (0, 0, 1)], 2)]$$

```
sage: M = matrix(QQ,3,3,[ 8,  6,  3, -1,  8, -3, 4, 10, 19])
sage: show(M)
sage: print M.eigenvalues()
sage: show( M.eigenvectors_left())
sage: print M.jordan_form()
[7, 14, 14]
[ 7| 0  0]
[--+-----]
[ 0|14  1]
[ 0| 0 14]
```

$$\begin{pmatrix} 8 & 6 & 3 \\ -1 & 8 & -3 \\ 4 & 10 & 19 \end{pmatrix}$$

$$\left[\left(7, \left[\left(1, -1, -\frac{1}{2}\right)\right], 1\right), (14, [(1, 6, 3)], 2)\right]$$

También podemos obtener la matriz de paso:

```
sage: M = matrix(QQ,3,3,[ 8,  6,  3, -1,  8, -3, 4, 10, 19])
sage: M.jordan_form(transformation=True)
(
[ 7| 0  0]
[--+-----]  [   1    0    1]
[ 0|14  1]  [   0   -7    0]
[ 0| 0 14], [-1/3   14    2]
)
```

La siguiente matriz tiene autovalores en $\mathbb{C} \setminus \mathbb{Q}$

```
sage: M = matrix(QQ,3,3,[0,1,0,-1,0,0,0,0,1])
sage: print M
sage: print M.eigenvalues()
sage: show( M.eigenvectors_left())
[ 0  1  0]
[-1  0  0]
[ 0  0  1]
[1, -1*I, 1*I]
```

$$[(1, [(0, 0, 1)], 1), (\text{-1*I}, [(1, 1\text{*I}, 0)], 1), (1\text{*I}, [(1, \text{-1*I}, 0)], 1)]$$

```
sage: print M.jordan_form()
Traceback (most recent call last):
...
RuntimeError: Some eigenvalue does not exist in Rational Field.
```

Como vemos, si hay autovalores en $\mathbb{C} \setminus \mathbb{Q}$, SAGE no calcula la forma de Jordan. Una solución es ampliar los coeficientes a $\overline{\mathbb{Q}}$, el cuerpo de los números algebraicos, que contiene todas las raíces de ecuaciones con coeficientes en $\mathbb{Q}$.

```
sage: #El cuerpo QQbar contiene las raices de todos los
sage: #polinomios con coeficientes en QQ
sage: M2 = M.base_extend(QQbar)
sage: print M2.jordan_form()
[   1|   0|   0]
[----+----+----]
[   0|-1*I|   0]
[----+----+----]
[   0|   0| 1*I]
```

La forma de Jordan sólo está definida en anillos de coeficientes exactos, como QQ, o QQbar, pero no tiene sentido en anillos con precisión limitada, porque la forma de Jordan es numéricamente inestable. Intentar calcular una forma de Jordan en un cuerpo con errores numéricos simplemente genera un error.

```
sage: M = matrix(CDF,3,3,[ 8,  6,  3, -1,  8, -3, 4, 10, 19])
sage: print M.jordan_form()
Traceback (most recent call last):
...
ValueError: Jordan normal form not implemented over inexact rings.
```

Sin embargo, aunque trabajemos con números de como flotante, todavía tiene sentido hablar de autovalores.

```
sage: M = matrix(CDF,3,3,[ 8,  6,  3, -1,  8, -3, 4, 10, 19])
sage: print M.eigenvalues()
[7.0 - 9.39502740013e-17*I, 14.0000000001 + 1.5006305546e-07*I, 13.9999999999 - 1.5006304995e-07*I]
```

Los autoespacios también son problemáticos, y es mejor evitarlos si usamos aritmética aproximada.

```
sage: M = matrix(QQ,2,2,[ 1, 1, 0, 1])
sage: print M.eigenspaces()
[
(1, Vector space of degree 2 and dimension 1 over Rational Field
User basis matrix:
[0 1])
]

sage: M = matrix(CDF,2,2,[ 1, 1, 0, 1])
sage: print M.eigenspaces()
[(1.0, Vector space of degree 2 and dimension 1 over Complex Double Field
User basis matrix:
[-2.22044604925e-16                1.0]), (1.0, Vector space of degree 2 and dimension 1 over Comple
User basis matrix:
[  0 1.0])]
```

4.5 Ejercicios

Buena parte de los siguientes ejercicios están extraídos de las hojas de problemas de Álgebra Lineal.

4.5.1 1.

Decide si cada uno de los siguientes conjuntos es una base de $\mathbb{R}^3$. Si lo es, encuentra las coordenadas en la nueva base de $v = (1,$

$$B_1 = \{(1,1,1),(0,1,1),(0,0,1)\}, \quad B_2 = \{(0,1,2),(1,2,3),(2,3,4)\}$$
$$B_3 = \{(0,1,1),(1,0,1),(1,1,0)\}, \quad B_4 = \{(1,1,1),(1,2,3),(1,-2,-3)\}$$

4.5.2 2.

Escribe una función que acepte como argumento una lista de vectores, y te devuelva un subconjunto maximal de vectores linealmente independientes extraídos de la lista anterior. El método consiste en recorrer la lista, y quedarte sólo con los vectores que son linealmente independientes con los vectores que has elegido antes.

Ejemplos:

```
{(1,0,0),(2,0,0),(1,1,0),(0,1,0),(2,1,0)} \-> {(1,0,0),(1,1,0)}
```

```
{(1,0,0),(1,0,1),(2,0,0),(1,1,0),(0,1,0),(2,1,0),(0,0,1)} \-> {(1,0,0),(1,0,1),(1,1,0)}
```

4.5.3 3.

Encuentra la inversa de la matriz siguiente usando el método de Gauss-Jordan: aumenta la matriz con la matriz identidad y usa operaciones de fila para reducir la matriz hasta la identidad:

$$A = \begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix}$$

http://es.wikipedia.org/wiki/Eliminaci%C3%B3n_de_Gauss-Jordan#Encontrando_la_inversa_de_una_matriz

Usa para ello los métodos: `augment` , `add_multiple_of_row` y otros similares que encuentres entre los métodos de las matrices. Compara el resultado con la matriz inversa obtenida de forma directa.

Cuando termines, escribe código que resuelva este problema en general.

4.5.4 4.

Se consideran los subespacios V y W del espacio vectorial real $\mathbb{R}^3$. Las ecuaciones paramétricas de V son

$$\begin{array}{rcl} x_1 & = & \lambda + \gamma \\ x_2 & = & \mu + \gamma \\ x_3 & = & \lambda + \mu + 2\gamma \end{array}$$

$$\lambda, \mu, \gamma \in \mathbb{R}$$

siendo $x_1 - x_2 + 2x_3 = 0$ la ecuación (implícita) de W. Hallar

- Bases de V, $V + W$ y $V \cap W$.
- Ecuaciones (implícitas) de $V \cap W$.
- Una base de un complementario de $V + W$.
- Las coordenadas de $(2, 3, 5)$ respecto de la base de $V + W$ obtenida en el primer apartado.

4.5.5 5.

- Hallar la forma de Jordan y una matriz de paso de las siguientes matrices.

$$A_1 = \begin{pmatrix} -4 & 0 & -4 & 11 \\ -6 & 3 & -3 & 9 \\ 5 & 0 & 5 & -7 \\ -1 & 0 & -1 & 5 \end{pmatrix} \quad A_5 = \begin{pmatrix} 3 & 1 & -1 & -1 \\ 0 & 2 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \quad A_9 = \begin{pmatrix} 2 & 0 & 1 & 2 \\ -1 & 3 & 0 & -1 \\ 2 & -2 & 4 & 6 \\ -1 & 1 & -1 & -1 \end{pmatrix}$$

- Busca métodos para calcular el polinomio característico (characteristic polynomial) y el polinomio mínimo (minimal polynomial) de estas matrices. Factoriza ambos polinomios y comprueba que el polinomio mínimo divide al característico.
- Calcula los autoespacios asociados a cada autovalor calculando el núcleo de $(A - \lambda I)^k$. Compara el resultado con los autoespacios calculados antes.

4.5.6 6: Divisor fijo de un polinomio

Para cada k, definimos el polinomio de una variable:

$$p_k(x) = \frac{1}{k!}x(x-1)\dots(x-k+1)$$

Los polinomios p_k forman una base del espacio de polinomios, al igual que los monomios x^k.

- Comprueba que a pesar de tener coeficientes fraccionarios, todos los polinomios de arriba toman siempre valores enteros cuando x es un entero (es decir, tras evaluar unos cuantos de esos polinomios en unos cuantos enteros x sólo se obtienen valores enteros).
- Dado un polinomio cualquiera, expresado como una expresión simbólica en la variable x, encuentra sus coeficientes en la base formada por los p_k.

El **divisor fijo** de un polinomio es el máximo común divisor de los valores que toma, cuando x toma como valores *todos* los números enteros.

- Genera unos cuantos polinomios aleatorios con divisor fijo mayor que 1. Define el *número h* del polinomio como el máximo común divisor de los coeficientes de p en la base p_k. *Conjetura una relación* entre el número h y el divisor fijo de un polinomio arbitrario.

4.6 Formas bilineales

Continuamos la sesión de álgebra lineal con varias aplicaciones a formas bilineales y productos escalares. Vamos a representar una forma bilineal por la matriz simétrica que la define. Por simplicidad, fijaremos el espacio vectorial y la forma bilineal como variables globales mientras sea conveniente.

4.6.1 Matrices definidas positivas

¿Cuándo es una matriz simétrica definida positiva? Sabemos que una matriz simétrica es diagonalizable. La matriz es definida positiva sii todos sus autovalores son positivos.

```
sage: def es_definida_positiva(M):
...       if not M.is_symmetric():
...           raise ValueError, "La matriz no es simetrica"
...       return all(l>0 for l in M.eigenvalues())
```

```
sage: B = matrix(QQ,3,3,[1,0,1,  0,2,0,  1,0,3])
sage: show(B)
sage: print es_definida_positiva(B)
sage: B = matrix(QQ,3,3,[1,0,1,  0,2,0,  1,0,-3])
sage: show(B)
sage: print es_definida_positiva(B)
True
False
```

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & -3 \end{pmatrix}$$

Ejercicio

- Escribe una función que haga el mismo papel, pero usando el criterio de Sylvester (una matriz es definida positiva sii los determinantes de sus menores principales son positivos).
- Escribe una función análoga para matrices definidas negativas.
- Usa el método `random_matrix` para generar 100 matrices con las que comprobar que ambos métodos dan el mismo resultado. (observación: `random_matrix` genera matrices no necesariamente simétricas, pero hay un truco canónico para obtener una matriz simétrica a partir de una matriz arbitraria).

4.6.2 Trabajar con un producto escalar distinto del habitual

Como sabemos de las clases de álgebra lineal, es fácil trabajar con un producto escalar distinto del habitual reemplazando los productos escalares por los productos escalares con la matriz B.

```
sage: #globales
sage: V = VectorSpace(QQ,3)
sage: B = matrix(QQ,3,3,[1,0,1,  0,2,0,  1,0,3])
```

Ejercicio (en clase)

Implementa los dos métodos siguientes:

```
sage: def son_ortogonales(W1, W2):
...       '''Comprueba si los dos subespacios de V
...       que se pasan como argumentos son ortogonales
...       '''
...       return ...
sage: def complemento_ortogonal(W):
...       '''Devuelve el complemento ortogonal en V del subespacio
...       que se pasa como argumento
...       '''
...       return ...
```

```
sage: W = V.subspace(list(random_matrix(QQ,2,3)))
sage: L = complemento_ortogonal(W)
sage: print son_ortogonales(L,W)   #Deberia ser True

sage: W = V.subspace(list(random_matrix(QQ,1,3)))
sage: L = complemento_ortogonal(W)
sage: print son_ortogonales(L,W)   #Deberia ser True
```

4.6.3 Bases ortonormales

Fijamos el espacio vectorial V y el producto escalar B.

```
sage: V = VectorSpace(QQ,3)
sage: B = matrix(QQ,3,3,[1,0,1,  0,2,0,  1,0,3])
sage: base = V.basis()
sage: print base
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

Ejercicio en clase

- Escribe una función que compruebe si una base dada es ortogonal.
- Lo mismo de antes, pero para una base ortonormal.

4.6.4 Método de Gram-Schmidt

El método de Gram-Schmidt permite obtener una base ortonormal a partir de una base arbitraria.

Comenzamos con una lista vacía que contendrá en todo momento un sistema ortonormal. Para cada vector de la base original, repetimos los pasos siguientes:

- Resta a ese vector su componente en cada uno de los vectores del sistema ortonormal.
- Divide el resultado por su norma para obtener un vector unitario.
- Añáde el resultado al sistema ortonormal.

```
sage: def gram_schmidt(base):
...       base_gm = []
...       for u in base:
...           w = u - sum((u*B*v)*v for v in base_gm)
...           base_gm.append( w/sqrt(w*B*w) )
...       return base_gm

sage: B = matrix(QQ,3,3,[1,-1,1,  -1,2,0,  1,0,3])
sage: gm = gram_schmidt(base)
sage: gm
[(1, 0, 0), (1, 1, 0), (-2, -1, 1)]
```

Ejercicio en clase

Verifica que el resultado es una base ortonormal, para 100 bases generadas aleatoriamente usando `random_matrix`.

Ejercicio

- Escribe una función que acepte como argumentos un vector y un subespacio, y devuelva la proyección ortogonal del vector en el subespacio (usando el producto escalar B definido globalmente).
- Reescribe el método de Gram-Schmidt usando esta función.

Una matriz simétrica definida positiva representa un producto escalar. Como hemos visto, podemos encontrar una base ortonormal para ese producto escalar.

Si tenemos dos productos escalares, podemos encontrar una base que es ortonormal para el primer producto escalar y ortogonal para el segundo. El procedimiento es el siguiente:

- Encuentra una base ortonormal para el primer producto escalar.
- En esa base, el primer producto escalar tiene como matriz la identidad, y el segundo tiene una matriz B, que es simétrica.
- Gracias al teorema espectral, es posible encontrar una matriz ortonormal tal que B es diagonal. En otras palabras, existe una base ortonormal para el primer producto escalar en la que la matriz del segundo producto escalar es diagonal. Es decir, que esa base también es ortogonal para el segundo producto escalar.

Ejercicio

Encuentra una base ortonormal para el producto escalar B1 y ortogonal para el producto escalar B2.

```
sage: B1 = matrix(RDF,3,3,[1,0,1,  0,2,0,  1,0,3])
sage: B2 = matrix(RDF,3,3,[3,-1,1,  -1,2,0,  1,0,1])
```

Bloque IV: Combinatoria

Aprendemos técnicas para construir familias de objetos combinatorios, usando recursión de forma explícita y también construyendo árboles de objetos combinatorios. Repasamos las construcciones usuales, ya implementadas en Sage, y las herramientas para resolver problemas de teoría de grafos. Como alternativa cuando la enumeración explícita es inviable, usamos el método de Monte Carlo.

5.1 Combinatoria

Algunos problemas de combinatoria requieren gran cantidad de cálculos y se impone una solución con el ordenador. Muchos otros problemas se pueden resolver con papel y lápiz, pero requieren tiempo e ingenio. Sin embargo, escribir programas de ordenador que resuelven los problemas por fuerza bruta suele ser bastante fácil. El problema estriba en que la cantidad de casos a examinar a menudo se dispara, y un programa poco fino sólo podrá resolver los casos más sencillos. A veces, éso es suficiente para darnos una idea, pero en general es más recomendable aprender a hacer programas más eficientes.

En este bloque escribiremos programas que exploran conjuntos de posibilidades muy amplios. Es muy importante comenzar siempre por aplicar nuestros programas con valores muy pequeños, medir bien los tiempos, y sólo lanzar los programas con los valores que nos interesan cuando pensemos que el programa podrá manejarlo.

5.1.1 Cómo iterar sobre todos los posibles valores de muchas variables

Comenzamos un truco muy práctico que puede servir como alternativa a anidar muchos bucles for. Supón que tenemos K variables que pueden tomar el valor 0 ó 1, y queremos recorrer todas las posibles combinaciones de valores para cada variable.

Si por ejemplo, K=10, la cantidad total de posibilidades es $2^{10} = 1024$, algo perfectamente asumible. Anidar 10 bucles es muy poco práctico, y obliga a hacer cambios sustanciales si aumentamos el número de variables:

```
for k1 in [0,1]:
    for k2 in [0,1]:
        for k3 in [0,1]:
            for k4 in [0,1]:
                for k5 in [0,1]:
...
```

En su lugar, podemos usar el siguiente truco: cuando un número k recorre los números desde 0 hasta $2^k - 1$, sus bits recorren todas las posibles combinaciones de 0 y 1 para K variables. Podemos usar, por ejemplo, el método `digits()`

pasando los argumento `base=2` y `padto=K` , para que extraiga los dígitos binarios (bits) y devuelva siempre listas de longitud K.

Este método lo tienen todos los enteros de SAGE (cuidado: esto nos obliga a usar `srange` en vez de `range` , para tener enteros de SAGE en vez de enteros de la máquina).

Con este truco, no necesitamos saber el número de variables al escribir el código.

```
sage: #Truco para iterar K variables que pueden tomar el valor 0 o 1
sage: #K no es conocido a priori
sage: K=4
sage: for k in srange(2^K):
...       vars = k.digits(base=2,padto=K)
...       print vars
[0, 0, 0, 0]
[1, 0, 0, 0]
[0, 1, 0, 0]
[1, 1, 0, 0]
[0, 0, 1, 0]
[1, 0, 1, 0]
[0, 1, 1, 0]
[1, 1, 1, 0]
[0, 0, 0, 1]
[1, 0, 0, 1]
[0, 1, 0, 1]
[1, 1, 0, 1]
[0, 0, 1, 1]
[1, 0, 1, 1]
[0, 1, 1, 1]
[1, 1, 1, 1]
```

5.1.2 Subconjuntos (y sublistas)

La técnica anterior permite iterar sobre todas las sublistas de una lista dada. Dada una lista con k elementos, usamos k variables que pueden tomar los valores 0 y 1. Si la variable xj vale 1, incluimos el elemento j-ésimo en la sublista y, si vale 0, no lo incluimos.

Ejercicio: usa la técnica anterior para mostrar todas las sublistas de una lista dada.

```
sage: lista = ['a','b','c','d']
```

También podemos usar la función `powerset` , definida en SAGE que hace la misma tarea.

```
sage: for sublista in powerset(lista):
...       print sublista
[]
['a']
['b']
['a', 'b']
['c']
['a', 'c']
['b', 'c']
['a', 'b', 'c']
['d']
['a', 'd']
['b', 'd']
['a', 'b', 'd']
['c', 'd']
['a', 'c', 'd']
```

```
['b', 'c', 'd']
['a', 'b', 'c', 'd']
```

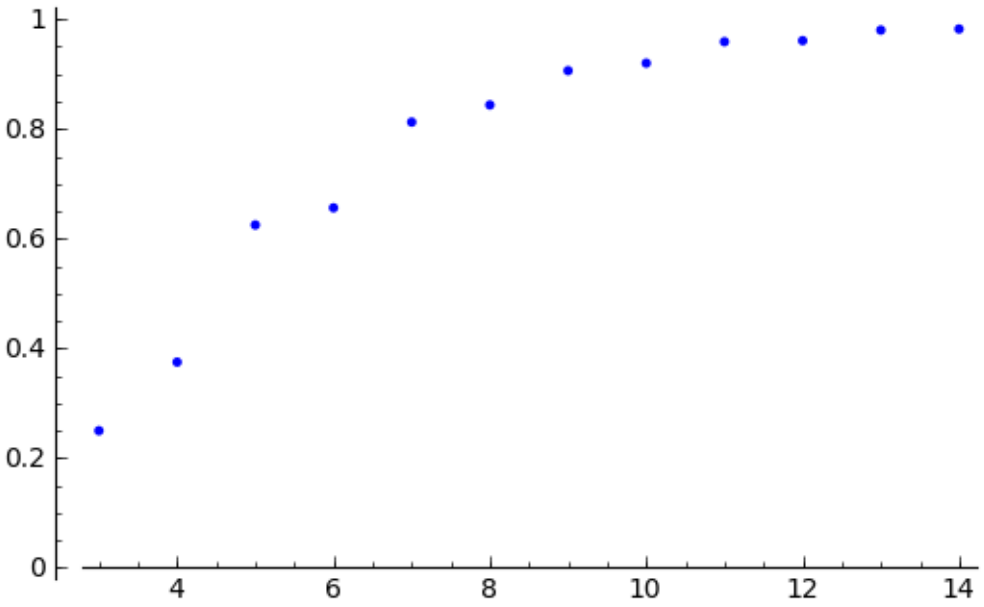
Ejemplo: gcd de un subconjunto de números

¿Cuántos conjuntos de números enteros entre 2 y K tienen máximo común divisor igual a 1?

```
sage: K = 7
sage: numeros = range(2,K+1)
sage: mcd_es_1 = 0
sage: for subconjunto in powerset(numeros):
...       if gcd(subconjunto)==1:
...           mcd_es_1 += 1
sage: print mcd_es_1
52
```

Consideramos el ratio entre el número de conjuntos de números enteros entre 2 y K cuyo máximo común divisor es igual a 1 frente al total de subconjuntos de enteros entre 2 y K: ¿existe el límite de esta sucesión?

```
sage: probs = []
sage: for K in range(3,15):
...       numeros = range(2,K+1)
...       mcd_es_1 = 0
...       for subconjunto in Subsets(numeros):
...           if gcd(subconjunto)==1:
...               mcd_es_1 += 1
...       total = 2^(K-1)
...       probs.append((K,mcd_es_1/total))

sage: point(probs).show(ymin=0,ymax=1)
```



Observación : la función `powerset` devuelve un objeto similar a un **generador**. Recordamos que al construir un generador el código no se ejecuta, y no se almacenan en memoria los resultados, sino que sólo se van construyendo según son necesarios, por ejemplo al iterar powerset en un bucle for. Sin embargo, difiere de un generador en que no se agota al iterarlo. Cada vez que recorremos el objeto, se crea un generador nuevo.

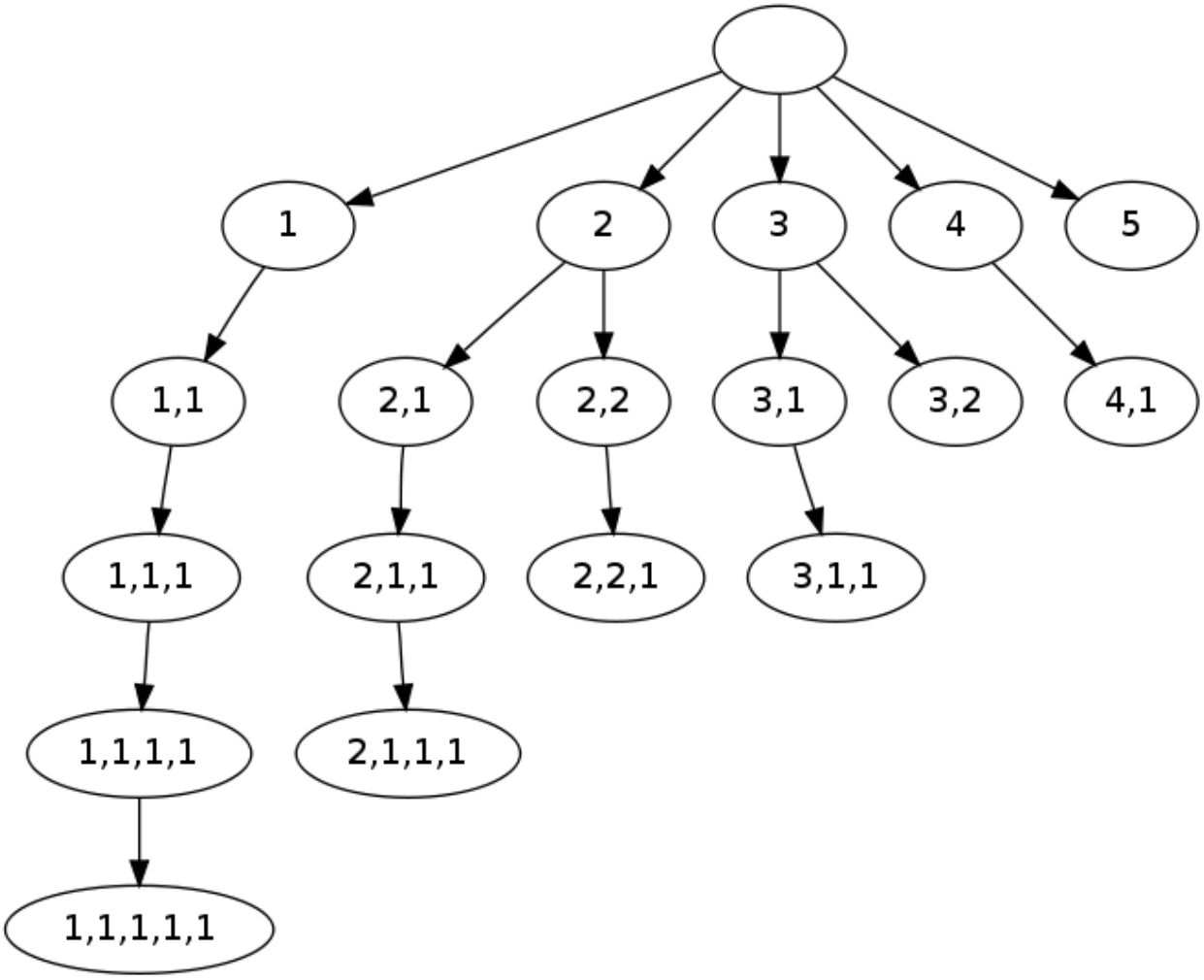
```
sage: ps = powerset([1, 2, 3])
sage: all(1 in s for s in ps)
sage: for k in ps:
...       print k
[1]
[2]
[1, 2]
[3]
[1, 3]
[2, 3]
[1, 2, 3]
```

5.1.3 Recorrer otras listas de objetos combinatorios

Los objetos combinatorios suelen admitir una definición recursiva que permite construir unos a partir de otros. Si representamos los objetos primordiales (una lista vacía, un árbol con un sólo nodo ...) como las raíces de un árbol, y utilizamos la definición recursiva para construir las sucesivas generaciones de nodos, podemos generar los objetos de forma bastante limpia, recorriendo el árbol que comienza en cada nodo base (sea en recorrido o en profundidad).

Lo importante de este enfoque es concretar la definición recursiva en una función que acepta un nodo como argumento y devuelve la lista con sus hijos, y separar esta parte de la definición del recorrido del grafo. Para esta segunda parte podemos usar por ejemplo la función `SearchForest` definida en Sage, que hace un recorrido en profundidad de un bosque, que no es otra cosa que una unión disjunta de árboles. También podemos escribir nuestro propio código genérico para esta tarea.

Como ejemplo, usamos esta idea para construir las **particiones** de un número k: todas las formas de sumar k con enteros positivos menores que k. Escribimos cada partición de k como una lista de enteros que suma k, y para evitar repeticiones las listamos sólo una vez en orden decreciente. Podemos construir todas las *particiones parciales* de k, que son listas decrecientes cuya suma es a lo sumo k. Los hijos de una partición parcial L son las particiones parciales que comienzan por L y tienen un elemento más que L. Lo dibujamos como un grafo:




```
sage: total = 5
sage: def hijos(ls):
...       '''Los hijos de una lista ls son las listas que comienzan por ls
...       y tienen un elemento más que ls
...       '''
...       suma  = sum(ls)
...       minimo = min(ls) if ls else total
...       return [ls + [j] for j in range(1, min(total - suma, minimo) + 1)]

sage: hijos([2])
[[2, 1], [2, 2]]

sage: #SearchForest necesita una lista de nodos raices y la funcion que
sage: #genera los hijos
sage: S = SearchForest([[]], hijos)
sage: #Si pasamos el resultado a lista recorremos todas las particiones parciales
sage: list(S)
sage: #Si queremos sólo las particiones completas, basta con filtrarlas
sage: list(ls for ls in S if sum(ls)==total)
[[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1], [3, 1, 1], [3, 2], [4, 1], [5]]
```

Antes de poder reusar este código, observamos que la función hijos usa la variable total definida en el ámbito principal. Para pasar este código a función, podemos simplemente anidar la función hijos dentro de la nueva función para que

pueda acceder al argumento de la función principal:

```
sage: def particiones(total):
...       def hijos(ls):
...           suma  = sum(ls)
...           minimo = min(ls) if ls else total
...           return [ls + [j] for j in range(1, min(total - suma, minimo) + 1)]
...
...       return [ls for ls in SearchForest([[]], hijos) if sum(ls)==total]
sage: list(particiones(5))
[[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1], [3, 1, 1], [3, 2], [4, 1], [5]]
```

Ya hemos encontrado otros ejemplos de este patrón, como por ejemplo generar todos los polinomios de cierto grado con coeficientes en un cierto rango. Repetimos el ejercicio con esta visión, declarando que los hijos del polinomio p son los polinomios px+j, para j en el rango de coeficientes en el que trabajamos.

```
sage: m = 3
sage: R.<x> = PolynomialRing(Integers(m))
sage: pols1 = [R(j) for j in srange(m)]
sage: grado_max = 3
sage: def children(p):
...       if p.degree()>=grado_max or p == 0:
...           return []
...       else:
...           return [p*x+j for j in srange(m)]
sage: S = SearchForest(pols1, children)
sage: list(S)
[0, 1, x, x^2, x^3, x^3 + 1, x^3 + 2, x^2 + 1, x^3 + x, x^3 + x + 1, x^3 + x + 2, x^2 + 2, x^3 + 2*x,
```

5.1.4 Objetos combinatorios útiles definidos en Sage

En Sage están definidos muchos objetos combinatorios útiles, que podemos usar y combinar de varias formas. Todos tienen una estructura uniforme que permite usarlos de varias formas. Todas estos objetos combinatorios son perezosos, e incluso podemos definir objetos infinitos. En general usan un método para *enumerar* todos los objetos y otro distinto para *contarlos* , que es importante porque en muchos casos se puede contar el número de objetos en muy poco tiempo aplicando fórmulas, mientras que enumerarlos todos podría llevar mucho tiempo. En general, usaremos los siguientes métodos:

- cardinality() para obtener el número de elementos
- list() para obtener la lista de todos los elementos
- random_element() para obtener un elemento aleatorio
- Iteraciones del tipo `for objeto in Clase` para recorrer los objetos sin necesidad de cargarlos todos en memoria.

Conjuntos y subconjuntos

Probemos todo ésto con los conjuntos (`Set`) y subconjuntos (`Subsets`) :

```
sage: S = Set([1,2,3,4])
sage: S1 = Subsets(S)
sage: S1.list()
[{}, {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}, {1, 2, 3}, {1, 2, 4}, {1, 3,
```

```
sage: S2 = Subsets(Subsets(S))
sage: S2.random_element()
{{3}, {4}, {1, 2, 3}, {1, 4}, {}, {2, 3, 4}, {1, 2, 4}, {3, 4}, {1, 3, 4}, {2, 3}, {1, 3}, {2}}

sage: #Por si acaso no te creias que para calcular el cardinal no hace falta
sage: #calcular todos los elementos
sage: S3 = Subsets(Subsets(Subsets(S)))
sage: S3.cardinality()
200352993040684646497907235156025575044782547556975141926501697371089405955631145308950613088093334811
```

Combinaciones

Dada una lista de tamaño k, las combinaciones de j elementos de la lista son todas las posibles sublistas con j elementos extraídos de la lista. En SAGE, podemos recorrer las combinaciones con un generador, de forma similar a la que usamos para powerset:

```
for c in Combinations(lista,j):
    ...haz algo con c...
```

```
sage: K=5
sage: for c in Combinations(range(2,K+1),2):
...       print c
[2, 3]
[2, 4]
[2, 5]
[3, 4]
[3, 5]
[4, 5]
```

Ejemplo: ¿Cuál es la probabilidad de que dos números elegidos al azar entre 2 y K sean primos entre sí?

Usando la fórmula:

$$\text{probabilidad} = \frac{\text{casos favorables}}{\text{casos posibles}}$$

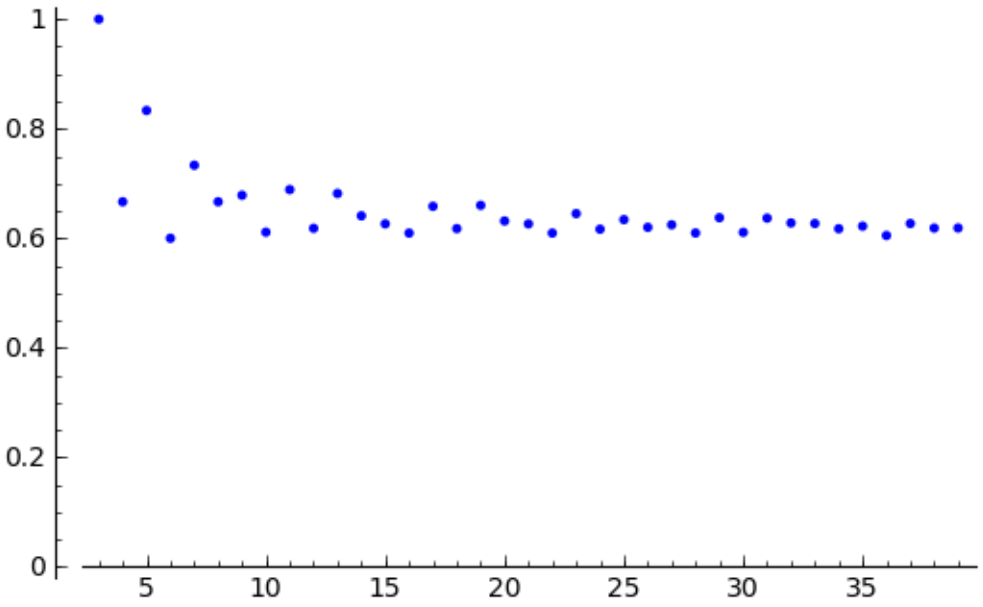
sólo tenemos que extraer todos los posibles pares de números entre 2 y K, y contar cuántas parejas están formadas por dos números primos entre sí.

```
sage: K=5
sage: primos_relativos = 0
sage: for c in Combinations(range(2,K+1),2):
...       if gcd(c)==1:
...           primos_relativos += 1
sage: print primos_relativos
5

sage: binomial(10,2)
45
```

¿Existe el límite cuando $K \to \infty$?

```
sage: probs = []
sage: for K in range(3,40):
...       primos_relativos = 0
...       for c in Combinations(range(2,K+1),2):
...           if gcd(c)==1:
```

```
...                 primos_relativos += 1
...         total = binomial(K-1,2)
...         probs.append((K,primos_relativos/total))
sage: print probs
[(3, 1), (4, 2/3), (5, 5/6), (6, 3/5), (7, 11/15), (8, 2/3), (9, 19/28), (10, 11/18), (11, 31/45), (1

sage: point(probs).show(ymin=0,ymax=1)
```



Más

- Permutaciones: `Permutations(lista)` recorre todas las ordenaciones de `lista` .
- Particiones: `Partitions (k)` recorre las formas de sumar `k` con enteros positivos. Además admite argumentos extra para limitar más las particiones (aunque atención: lee bien la documentación porque en algunos casos el resultado no es claro).
- Vectores de enteros: `IntegerVectors` recorre las formas de sumar un entero con enteros no negativos. También admite argumentos extra.
- `WeightedIntegerVectors` recorre las formas de sumar un entero usando los enteros de una lista dada (¿de cuántas formas puedo pagar 33 euros usando billetes de 5 y monedas de 2?)
- `CartesianProduct` calcula el producto cartesiano de dos conjuntos.

```
sage: Permutations(srange(4)).list()
[[0, 1, 2, 3], [0, 1, 3, 2], [0, 2, 1, 3], [0, 2, 3, 1], [0, 3, 1, 2], [0, 3, 2, 1], [1, 0, 2, 3], [1

sage: Partitions(5).list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]

sage: IntegerVectors(4, max_length=3).list()
[[4], [3, 1], [3, 0, 1], [2, 2], [2, 1, 1], [2, 0, 2], [1, 3], [1, 2, 1], [1, 1, 2], [1, 0, 3], [0, 4

sage: #Todas las formas de sumar 8 usando monedas de 1, 2 y 5
sage: WeightedIntegerVectors(8, [1,2,5]).list()
[[1, 1, 1], [3, 0, 1], [0, 4, 0], [2, 3, 0], [4, 2, 0], [6, 1, 0], [8, 0, 0]]
```

```
sage: WeightedIntegerVectors(33,[2,5]).list()
[[4, 5], [9, 3], [14, 1]]
```

5.2 Ejercicios

5.2.1 1.

- Escribe código para recorrer todos los posibles valores de un conjunto de variables x1,x2,...,xK, donde x1 toma los valores 0,1,...,B1-1; x2 toma los valores 0,1,...,B2-1;... y xK toma los valores 0,1,...,BK-1. Los valores B1,..., BK están almacenados en una lista.
- Utiliza el código anterior para contar el número de posibilidades para las que la suma de los valores de las variables es mayor o igual que un número M.

5.2.2 2.

Consideramos matrices 3x3 cuyas entradas son 0 ó 1. El total de matrices de este tipo es $2^9 = 512$. Cuenta el número de estas matrices cuyo determinante es 1 módulo 2 (es decir, el determinante es impar).

Equivalentemente, cuenta el número de matrices 3x3 con coeficientes en $\mathbf{Z}_2$ cuyo determinante es no nulo.

5.2.3 3.

Encuentra todas las formas de colocar 8 reinas en un tablero de ajedrez de forma que ninguna reina amenace a otra. Plantea el problema como un árbol de soluciones parciales partiendo de un tablero vacío y agregando una sóla reina cada vez. Usa SearchForest para recorrer el grafo.

Nota: el enfoque anterior, tomado de forma literal, da lugar a un algoritmo muy ineficiente. Es buena idea asumir de partida que cada reina tiene que ir en una columna distinta y anotar sólo la fila que ocupa la reina i-ésima.

5.2.4 4.

- Escribe una función genérica que recibe una lista de nodos raices y una función que construye los hijos y devuelva todos los nodos del árbol engendrado (al igual que hicimos con SearchForest), pero realizando una búsqueda en anchura .
- Modifica el código anterior para que devuelva sólo las hojas (nodos que no tienen hijos).
- Aplica esta función a algún ejemplo de la hoja.

5.2.5 5.

Escribe código para recorrer todas las posibles listas que constan de 3 elementos extraídos de una lista dada, *en todas las ordenaciones posibles* . Usa tres estrategias:

- Usa la función de Sage que extrae listas de tres elementos, y después usa la función de Sage que recorre las permutaciones de las listas de 3 elementos.
- Usa primero la función de Sage que recorre las permutaciones de la lista dada, y después extrae listas de tres elementos que respetan el orden dado.

- Escribe todas las elecciones de sublistas como un árbol cuyo nodo raíz es una sublista vacía, y tal que en cada elección eliges un nuevo elemento de la lista. Nota: recuerda lo que significa un elemento mutable en python!

5.2.6 6. Monedas

Nota. Los siguientes problemas de probabilidad se pueden resolver a mano, pero ahora puedes escribir código que utiliza la fórmula:

$$\frac{\text{casos favorables}}{\text{casos posibles}}$$

para calcular la probabilidad contando uno por uno los casos favorables.

- Calcula la probabilidad de que, al lanzar 10 monedas (equilibradas), obtengas al menos 4 caras.
- Calcula la probabilidad de que, al lanzar 10 monedas (equilibradas), obtengas el mismo número de caras con las 5 primeras monedas que con las 5 siguientes.
- Calcula la probabilidad de que, al lanzar 15 monedas (equilibradas), las primeras 10 monedas contengan al menos el doble de caras que las 5 monedas siguientes.

Indicación: Usa k variables 0/1 para representar un posible lanzamiento de monedas (1:cara, 0:cruz). Para cada apartado, escribe una función que acepte como argumento una lista con los valores de las variables y devuelva True si es un caso favorable y False si no lo es.

5.2.7 7. Cartas

Nos planteamos ahora resolver problemas de cartas. El primer paso es representar las cartas de la baraja. Usaremos la baraja española de 40 cartas para agilizar los cálculos.

Por ejemplo, podemos representar cada carta por una tupla que contiene su número (de 1 a 10, donde 8, 9 y 10 son las figuras) y su palo (por ejemplo, usando la baraja española: 'o' (oros), 'c' (copas), 'e' (espadas), 'b' (bastos)).

- Crea una lista con las 40 cartas de la baraja española, representadas de la forma descrita arriba. Ejemplos: (4,'c') 4 de copas; (10,'b') 10 de bastos (rey de bastos).

Para calcular una probabilidad con la fórmula

$$\frac{\text{casos favorables}}{\text{casos posibles}}$$

necesitamos, por un lado, recorrer todas las posibles extracciones de cinco cartas de una baraja y, por otro lado, discriminar si una mano de cinco cartas es un caso favorable o no.

- Escribe una función que acepte como argumento una lista con 5 cartas y devuelva True si esa mano es una escalera y False si no lo es.
- Lo mismo de antes, para un póker.
- Calcula la probabilidad de obtener un póker, o una escalera, en la primera mano.

Indicación : si tu código tarda más de unos pocos segundos, será mejor que uses una baraja más pequeña hasta estar seguro de que todo funciona. Ten en cuenta que las posibles extracciones de cinco cartas de una baraja española son $\binom{40}{5} = 658008$. Usa una baraja con menos palos y menos números, y haz que tu código imprima todos los casos favorables.

- Si lo prefieres, calcula la probabilidad de algunas jugadas de mus. Como las manos sólo tienen 4 cartas, hay muchas menos posibilidades.

5.3 Contar y enumerar

En esta sesión vamos a enumerar varias familias de objetos combinatorios, igual que hicimos en la primera sesión, pero ahora además de enumerar objetos los contaremos sin enumerarlos, para hacer más patente la relación entre una construcción recursiva y las fórmulas recursivas para el número de elementos que son habituales en combinatoria. El primer ejemplo está resuelto, los dos siguientes son ejercicios que resolveremos en clase, y los dos últimos son una práctica a entregar.

5.3.1 Ejemplo resuelto : particiones de n con k partes fijas

Estudiamos las formas de descomponer un número n como suma de exactamente k enteros positivos. Listamos los números de cada partición de mayor a menor, para evitar repeticiones. Observamos que las particiones pertenecen a dos clases:

- Las que terminan en 1.
- Las que tienen todos sus elementos mayores o iguales a 2.

Esta idea nos permite construir las particiones recursivamente:

- Generamos las particiones de n-1 en k-1 partes iguales, y añadimos un 1 al final para dar cuenta del primer grupo de particiones.
- Generamos las particiones de n-k en k partes iguales, y sumamos 1 a cada elemento para dar cuenta del segundo grupo de particiones.

```
sage: def particiones_k(total, partes):
...       if partes == total:
...           return [[1]*total]
...       if partes == 1:
...           return [[total]]
...       if not(0 < partes < total):
...           return []
...       ls1 = [p+[1] for p in particiones_k(total-1, partes-1)]
...       ls2 = [[parte+1 for parte in p] for p in particiones_k(total-partes, partes)]
...       return ls1 + ls2

sage: particiones_k(8,3)
[[6, 1, 1], [5, 2, 1], [4, 3, 1], [4, 2, 2], [3, 3, 2]]

sage: #Las particiones sin restricciones (que vimos en la sesion 1)
sage: #se pueden obtener facilmente a partir de particiones_k
sage: def particiones(total):
...       return [ls for j in range(1,total+1)
...                  for ls in particiones_k(total, j)]
sage: particiones(5)
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
```

Si sólo queremos contar el número de particiones de n con k elementos (en adelante $p_k(n)$), las reglas de recursión anteriores se traducen en la siguiente ecuación de recurrencia:

$$p_k(n) = p_{k-1}(n-1) + p_k(n-k)$$

junto con los "datos de frontera":

$$p_1(n) = p_n(n) = 1$$

y:

$$p_k(n) = 0 \text{ si } k \leq 0, \text{ ó } k > n, \text{ ó } n \leq 0$$

```
sage: def n_particiones_k(total, partes):
...       if partes == total:
...           return 1
...       if partes == 1:
...           return 1
...       if not(0 < partes < total):
...           return 0
...       n1 = n_particiones_k(total-1, partes-1)
...       n2 = n_particiones_k(total-partes, partes)
...       return n1 + n2

sage: n_particiones_k(8,3)
5
```

Obviamente, se tarda menos tiempo en contar las posibilidades que en construirlas todas explícitamente.

```
sage: time n_particiones_k(60,10)
62740
Time: CPU 0.10 s, Wall: 0.10 s

sage: time ls = particiones_k(60,10)
Time: CPU 2.19 s, Wall: 2.20 s
```

5.3.2 Cascadas de llamadas recursivas

En el ejemplo anterior hemos transportado la definición recursiva directamente a nuestro código, pero vimos en el bloque II que este enfoque nos puede traer problemas, al usar una definición recursiva para calcular los números de fibonacci:

```
def fibo(n):
    if n<2:
        return 1
    return fibo(n-1) + fibo(n-2)
```

En este ejemplo paradigmático, estimamos que para calcular el número de fibonacci n-ésimo se hacían aproximadamente tantas llamadas recursivas a `fibo` como el número de fibonacci n-ésimo (más exactamente, el número I(n) de llamadas a `fibo` para calcular fibo(n) es: I(n-1)+I(n-2)+1). Usando la técnica del " *profiler* " de la sesión b3s3, medimos exactamente el número de llamadas para confirmar este hecho:

```
sage: def fibo(n):
...       if n<2:
...           return 1
...       return fibo(n-1) + fibo(n-2)
```

Observamos la columna `ncalls` que indica el número de llamadas a una función. Comprueba para unos pocos valores que se satisface la fórmula de recurrencia mencionada.

```
sage: #importamos los modulos cProfile y pstats para ver las estadisticas
sage: #de cuanto tiempo se pasa en cada parte del codigo
sage: import cProfile, pstats
sage: #No necesitamos entender la siguiente linea:
```

```
sage: #tomalo como una version avanzada de timeit
sage: cProfile.runctx("fibo(10)", globals(), locals(), DATA + "Profile.prof")
sage: s = pstats.Stats(DATA + "Profile.prof")
sage: #Imprimimos las estadisticas, ordenadas por el tiempo total
sage: s.strip_dirs().sort_stats("time").print_stats()
Mon Feb 21 22:57:30 2011    /home/sageadm/nbfiles.sagenb/home/pang/295/data/Profile.prof

         179 function calls (3 primitive calls) in 0.000 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    177/1    0.000    0.000    0.000    0.000 ___code___.py:3(fibo)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats instance at 0x9a68d40>
```

En nuestro ejemplo anterior, ocurre algo similar, porque al llamar a `particiones_k(n,k)` llamamos recursivamente a `particiones_k(n-k,k)` y a `particiones_k(n-1,k-1)` , lo que eventualmente implica que llamaremos varias veces a `particiones_k(n,k)` con los mismos argumentos.

A modo de confirmación, el sentido común nos dice que para calcular `n_particiones_k(n,k)` , necesitaremos como mucho n*k llamadas recursivas a `n_particiones_k` , lo que corresponde a llamar a la función con todos los posibles valores menores que n y con los menores que k. Sin embargo, el número de llamadas a `n_particiones_k(40,10)` es 9111, mucho mayor que 400.

```
sage: #importamos los modulos cProfile y pstats para ver las estadisticas
sage: #de cuanto tiempo se pasa en cada parte del codigo
sage: import cProfile, pstats
sage: #No necesitamos entender la siguiente linea:
sage: #tomalo como una version avanzada de timeit
sage: cProfile.runctx("n_particiones_k(40,10)", globals(), locals(), DATA + "Profile.prof")
sage: s = pstats.Stats(DATA + "Profile.prof")
sage: #Imprimimos las estadisticas, ordenadas por el tiempo total
sage: s.strip_dirs().sort_stats("time").print_stats()
Mon Feb 21 22:57:30 2011    /home/sageadm/nbfiles.sagenb/home/pang/295/data/Profile.prof

         9113 function calls (3 primitive calls) in 0.014 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   9111/1    0.014    0.000    0.014    0.014 ___code___.py:3(n_particiones_k)
        1    0.000    0.000    0.014    0.014 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats instance at 0x9a68ef0>
```

Una solución rápida

En otros lenguajes como **Haskell** , las definiciones anteriores no provocan esta *cascada de llamadas recursivas* . Esto se debe a que Haskell es un lenguaje funcional puro, lo que implica entre otras cosas que todas las llamadas a una función con los mismos argumentos producen siempre el mismo resultado. Gracias a este conocimiento, el compilador

no necesita ejecutar una función la segunda vez que la llaman con unos argumentos dados, porque ya ha calculado el valor una vez.

En python, ésto no es necesariamente cierto (piensa por ejemplo en las funciones que generan números aleatorios), pero podemos crear una función que guarde los valores de las llamadas en memoria, usando el **decorador** `@cached_function` , que altera una función para que almacene los valores de cada llamada, y así evitar calcular dos veces el valor de la función con los mismos argumentos.

```
sage: @cached_function
sage: def n_particiones_k_cached(total, partes):
...       if partes == total:
...           return 1
...       if partes == 1:
...           return 1
...       if not(0 < partes < total):
...           return 0
...       n1 = n_particiones_k_cached(total-1, partes-1)
...       n2 = n_particiones_k_cached(total-partes, partes)
...       return n1 + n2
```

Verificamos que el número de llamadas recursivas desciende drásticamente (en este caso, a 248).

```
sage: #importamos los modulos cProfile y pstats para ver las estadisticas
sage: #de cuanto tiempo se pasa en cada parte del codigo
sage: import cProfile, pstats
sage: #No necesitamos entender la siguiente linea:
sage: #tomalo como una version avanzada de timeit
sage: cProfile.runctx("n_particiones_k_cached(40,10)", globals(), locals(), DATA + "Profile.prof")
sage: s = pstats.Stats(DATA + "Profile.prof")
sage: #Imprimimos las estadisticas, ordenadas por el tiempo total
sage: s.strip_dirs().sort_stats("time").print_stats()
Mon Feb 21 22:57:31 2011    /home/sageadm/nbfiles.sagenb/home/pang/295/data/Profile.prof

         4630 function calls (4019 primitive calls) in 0.017 CPU seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      365    0.011    0.000    0.013    0.000 function_mangling.py:205(fix_to_pos)
    365/1    0.002    0.000    0.017    0.017 cachefunc.py:96(__call__)
    248/1    0.001    0.000    0.017    0.017 ___code___.py:3(n_particiones_k_cached)
      365    0.001    0.000    0.013    0.000 cachefunc.py:188(get_key)
      365    0.000    0.000    0.000    0.000 {range}
      365    0.000    0.000    0.000    0.000 {method 'has_key' of 'dict' objects}
      365    0.000    0.000    0.000    0.000 {sorted}
      730    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
      365    0.000    0.000    0.000    0.000 function_mangling.py:261(<genexpr>)
      365    0.000    0.000    0.000    0.000 cachefunc.py:119(get_cache)
      365    0.000    0.000    0.000    0.000 {len}
      365    0.000    0.000    0.000    0.000 {method 'keys' of 'dict' objects}
        1    0.000    0.000    0.017    0.017 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


<pstats.Stats instance at 0x9a7ccb0>
```

Cuestión : Vuelve a ejecutar el código de arriba, sin modificarlo. Explica el resultado. Usa la acción "Restart Worksheet", y ejecútalo de nuevo (ejecuta antes el cuadro que define `n_particiones_k_cached`). Explica el resultado.

5.3.3 Ejercicios

Palabras con 1's y 0's sin dos 1's seguidos

Estudiamos las palabras formadas por 0 y 1 tales que no hay dos unos consecutivos. Hay dos palabras de longitud 1 con esta propiedad:'0' y '1', tres de longitud 2: '00', '01', '10', 5 de longitud 3: '000', '001', '010', '100', '101'. Las palabras se pueden generar de forma recursiva siguiendo la regla siguiente:

- Las palabras que terminan en '0' se pueden extender con un '0' o con un '1'
- Las palabras que terminan en '1' sólo se pueden extender con un '0'.

También puedes seguir la siguiente regla equivalente para generar todas las palabras de longitud k:

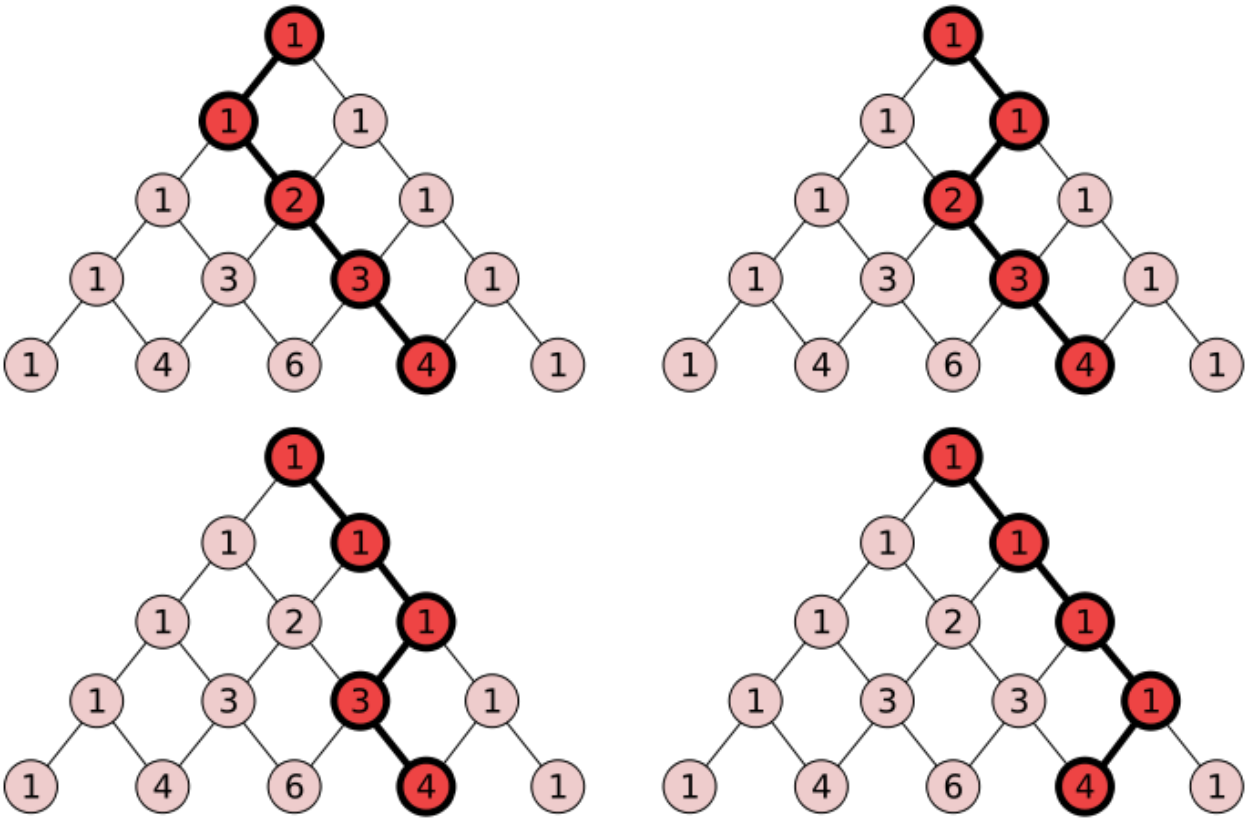
- Añade un '0' a todas las palabras de longitud k-1.
- Añade un '01' a todas las palabras de longitud k-2.

Ejercicio

- Escribe código que genere todas las palabras de este tipo hasta longitud k.
- Escribe (y justifica) una ecuación de recursión para el número total de palabras de este tipo: ¿te resulta familiar?
- Construye los caminos colocando las palabras en un árbol y usando SearchForest. Los hijos de una palabra que termina en '0' son la misma palabra seguida de '0' y de '1', mientras que el único hijo de una palabra que termina en '1' es la misma palabra seguida de '0'. En este caso el *bosque* tiene dos nodos raíces, el '0' y el '1'.

Caminos en una malla

El triángulo de Pascal registra el número de caminos desde el vértice superior hasta un vértice cualquiera, tales que en cada paso descendemos de una fila a la inmediatamente inferior mediante una de las aristas descendientes: bien a la izquierda bien a la derecha.



Codificamos los caminos como secuencias con las letras I y D. Por ejemplo el camino que está arriba a la izquierda del diagrama corresponde a la secuencia 'IDDD', mientras que el camino de abajo a la derecha corresponde a 'DDDI'.

La siguiente recursión permite encontrar todos los caminos que llevan del vértice superior al vértice en la fila n y la columna k:

- Encuentra todos los caminos que llevan a la posición (n-1, k-1) y añádeles una 'D'.
- Encuentra todos los caminos que llevan a la posición (n-1, k) y añádeles una 'I'.

Ejercicio :

- Aplica la recursión anterior para calcular todos los caminos que llevan del vértice superior al vértice en la fila n y la columna k. Deberías obtener exactamente $\binom{n}{k}$ tales caminos.
- Escribe (y justifica) una ecuación de recursión para el número total de caminos basada en la recursión anterior: ¿te resulta familiar?
- Construye los caminos colocando los caminos parciales en un árbol y usando SearchForest. Los caminos parciales son caminos desde el vértice superior hasta otro vértice que son susceptibles de ser continuados hasta el vértice (n,k) siguiendo las reglas del juego.

5.3.4 Entrega b4

Los dos ejercicios que siguen constituyen la entrega del bloque IV, que por supuesto es opcional. Vale un total de 7 puntos.

Formas de agrupar antes de operar

Hay dos formas de agrupar tres letras (por ejemplo, para realizar una operación matemática asociativa sobre varios símbolos, pero sin alterar el orden de las letras)

```
(a(bc)), ((ab)c)
```

Hay cinco formas de agrupar cuatro letras:

```
(a((bc)d)), (a(b(cd))), ((ab)(cd)), (((ab)c)d), ((a(bc))d)
```

La siguiente recursión permite encontrar todas las formas. Partimos de una secuencia de letras, 'abcd':

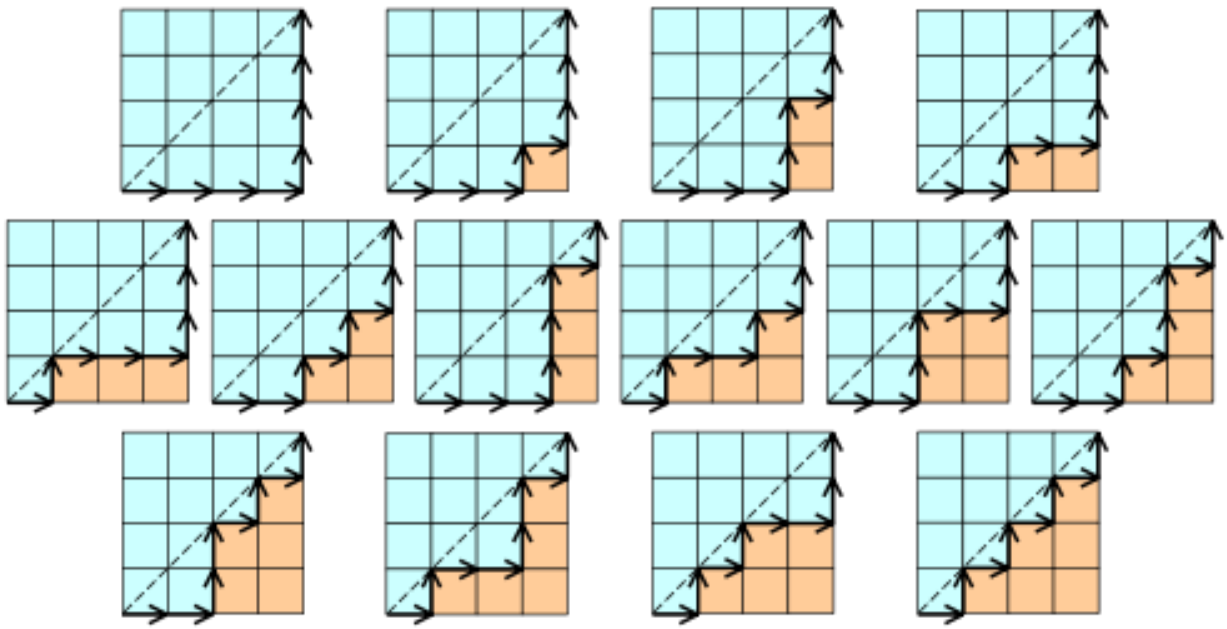
- Partimos la secuencia en todos los posibles puntos: ['a', 'bcd'], ['ab', 'cd'], ['abc','d']
- Agrupamos cada parte recursivamente de todas las formas posibles. Por ejemplo, 'bcd' se puede agrupar como '(b(cd))' y como '((bc)d)'
- Unimos cada forma de agrupar las letras de la izquierda con cada forma de agrupar las letras de la derecha, y encerramos el resultado entre paréntesis. Ej, 'a', '((bc)d)' -> '(a((bc)d))' (las letras sueltas no necesitan paréntesis)

Ejercicio

- Aplica la recursión anterior para construir todas las formas de agrupar n letras.
- Escribe (y justifica) una ecuación de recursión para el número total de posibilidades. Cuenta el número de posibilidades G(n) para n desde 1 hasta 10.

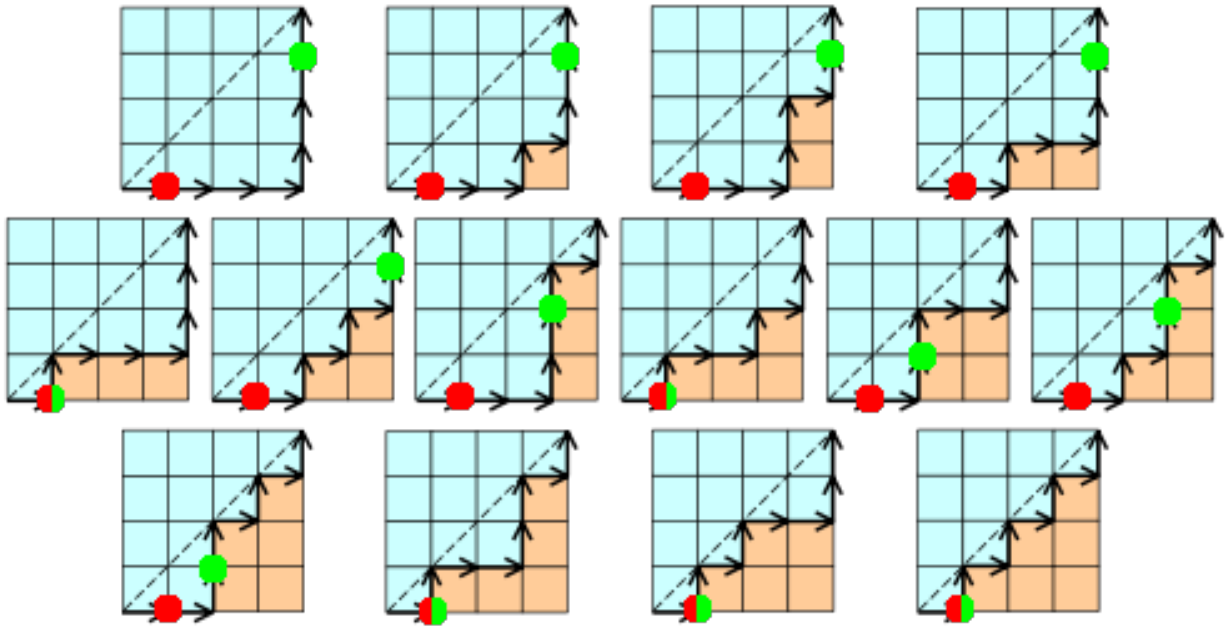
Caminos monótonos

Consideramos ahora los *caminos monótonos* , en la forma siguiente: todos los caminos que unen el vértice (0,0) con el vértice (n,n) avanzando hacia la derecha ('D') y hacia arriba ('A'), tales que en todo momento hemos avanzado al menos tantos pasos hacia la derecha como hemos avanzado hacia arriba.



Sea C(n) el número total de caminos que llegan al vértice (n,n) partiendo del vértice (0,0). Llamaremos a estos caminos "diagonales" porque terminan en la misma diagonal en la que empiezan.

¿Cómo podemos encontrar una fórmula de recursión para C(n)? Observamos que todos los caminos siguen la siguiente estructura: comienzan con un movimiento a la derecha, luego sigue un camino diagonal (posiblemente vacío), luego viene un movimiento hacia arriba que compensa el primer movimiento a la derecha, y luego viene otro camino diagonal (posiblemente vacío). En el siguiente diagrama, los puntos rojo y verde marcan el principio y el fin del primer camino diagonal.



Ejercicio

- Aplica la recursión anterior para construir todos los caminos . Codifica los caminos como secuencias con las letras D y A, que indican en qué momentos debemos movernos a la derecha o hacia arriba. Por ejemplo, el diagrama de abajo a la derecha corresponde al camino 'DADADADA', y el de arriba a la izquierda al camino 'DDDDAAAA' .
- Escribe (y justifica) una ecuación de recursión para el número total de caminos de este tipo. Calcula C(n) para n desde 1 hasta 10.
- Sustituye la letra 'D' por el carácter '(' y la letra 'A' por el carácter ')'. Describe el resultado.

Los ejemplos están extraídos del capítulo 6 del libro de texto de la asignatura de matemática discreta, que puedes consultar como referencia:

http://www.uam.es/personal_pdi/ciencias/gallardo/cap6-MD-2010-2011.pdf

del libro de Sage en francés:

http://sagebook.gforge.inria.fr/

y de la wikipedia:

http://en.wikipedia.org/wiki/Catalan_number

5.3.5 Entrega b4

Los dos ejercicios que siguen constituyen la entrega del bloque IV, que por supuesto es opcional. Vale un total de 7 puntos.

5.4 Grafos

En esta lección vamos a estudiar las posibilidades que ofrece SAGE para trabajar con **grafos** .

Un grafo consiste de un conjunto de **vértices** y otro conjunto de **aristas** que unen algunos de los vértices. En un grafo no dirigido las aristas no tienen dirección, mientras que en los grafos dirigidos debemos distinguir entre la arista que une el vértice v1 con el v2 de la arista que une el vértice v2 con el v1.

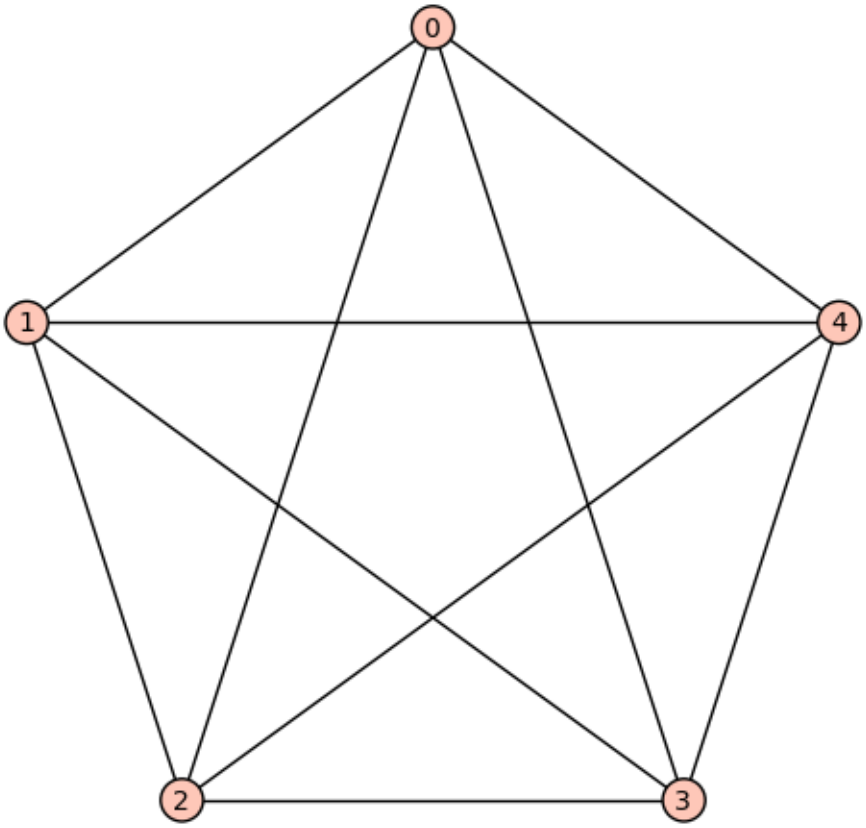
5.4.1 Introducir Grafos en SAGE

Algunos grafos especiales

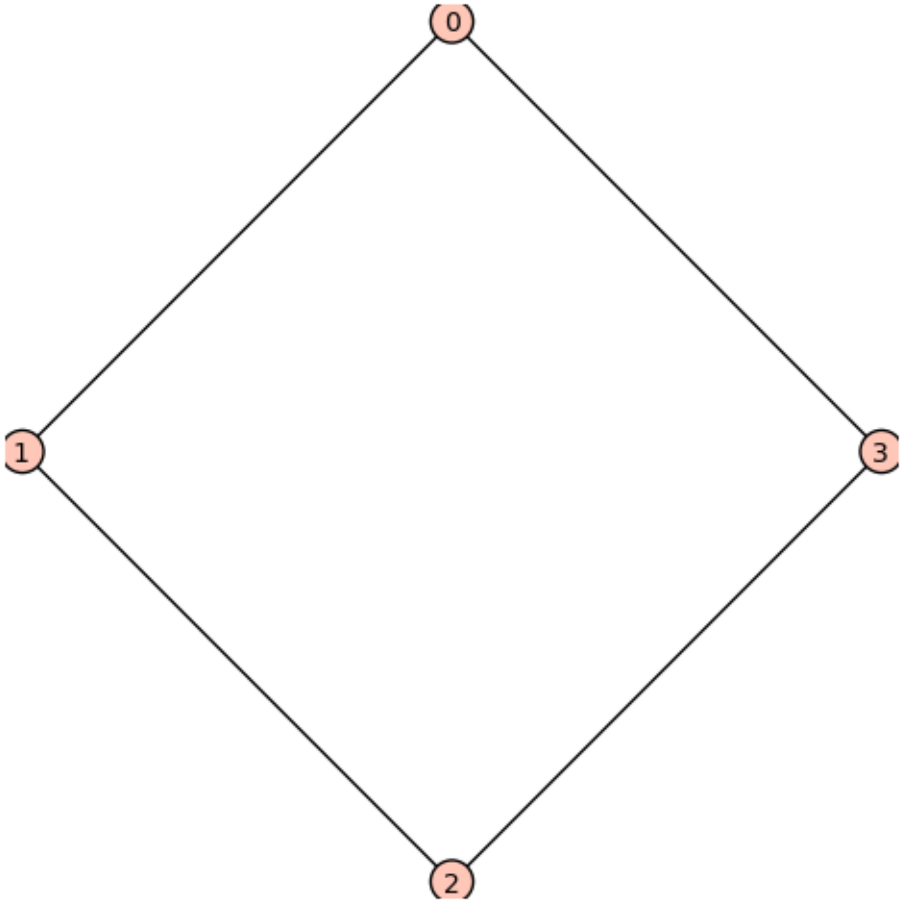
Existen constructores para muchos de los grafos no dirigidos más famosos, dentro del módulo `graphs` . Por ejemplo:

- Grafo completo: el que tiene todas las aristas que unen cada par de vértices
- Grafo cíclico: una circuito simple
- Grafo del n-cubo
- ...

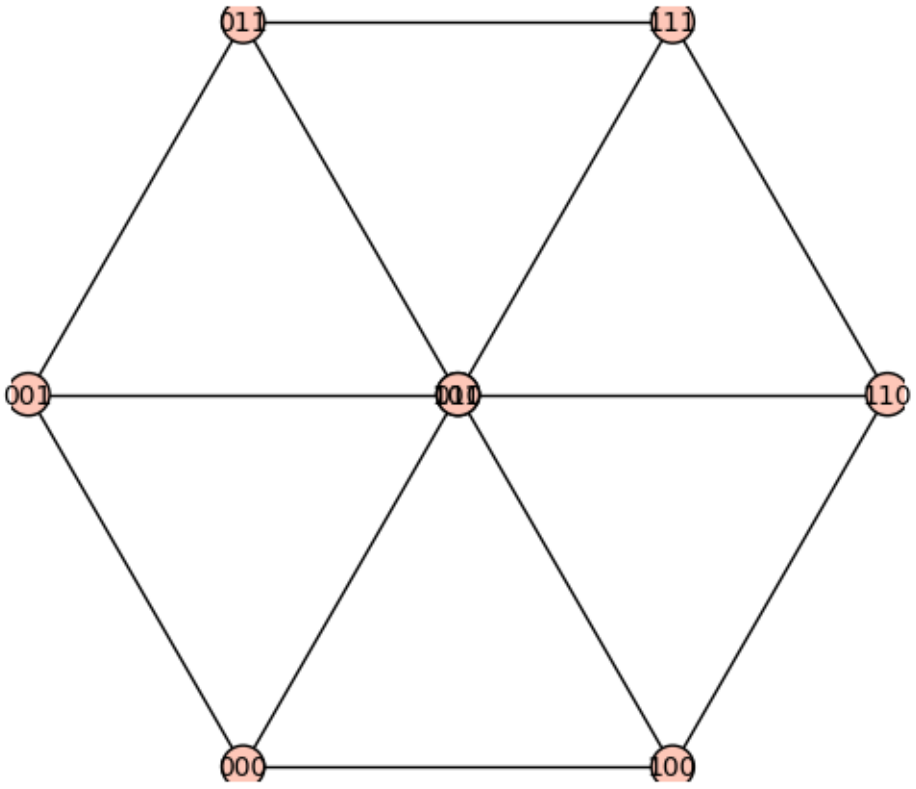
```
sage: g1 = graphs.CompleteGraph(5)
sage: show(g1.plot())
```




```
sage: g2 = graphs.CycleGraph(4)
sage: show(plot(g2))
```



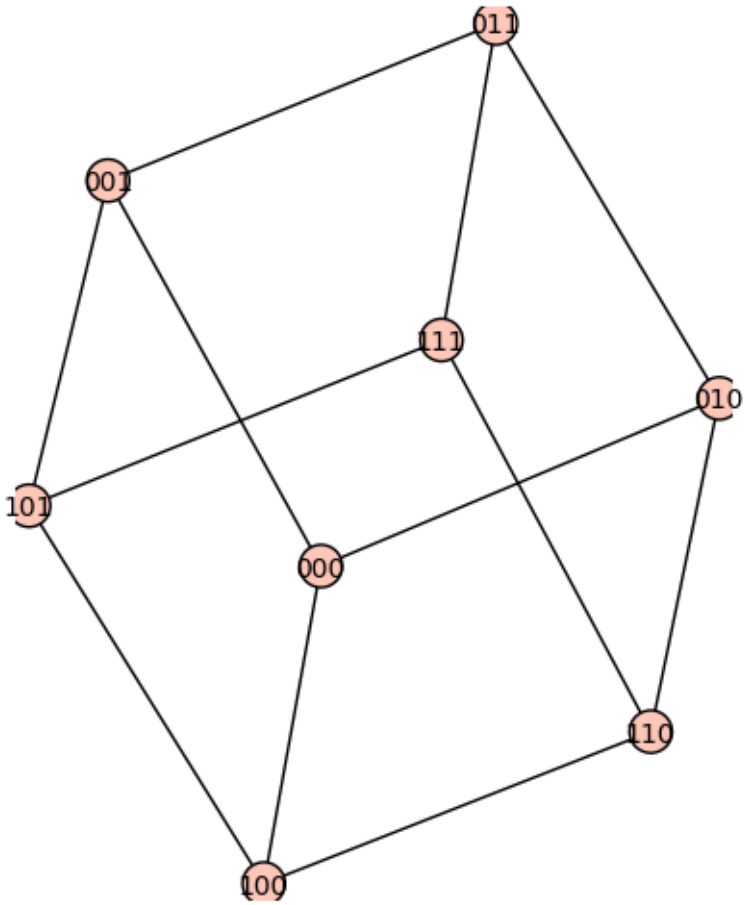

```
sage: g3 = graphs.CubeGraph(3)
sage: show(plot(g3))
```



El dibujo anterior confunde dos de los vértices. Leemos la documentación de Graph.plot para encontrar la forma de mejorar el dibujo

```
sage: g3.plot?
<html>...</html>
```

```
sage: #Una solucion: usar otro "layout"
sage: show(plot(g3, layout='spring'))
```

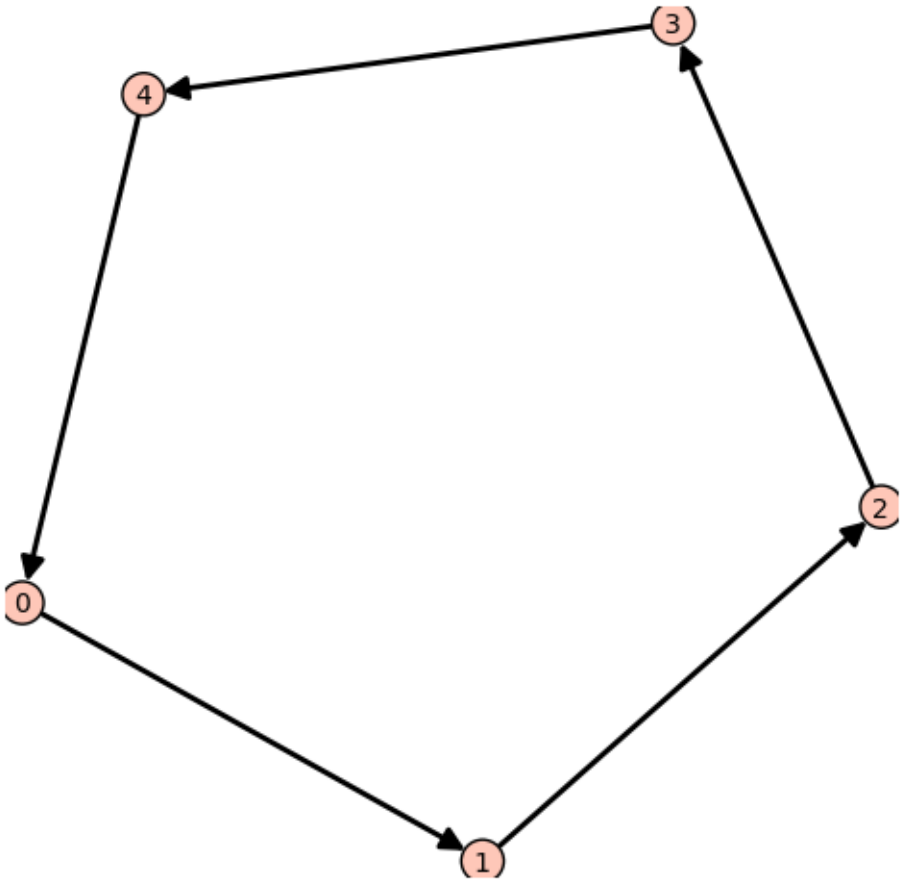


Como de costumbre, usando el tabulador accedemos a una lista completa con todos los grafos disponibles.

```
sage: graphs.
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

También tenemos alguns familias de grafos famosos en la librería `digraphs` .

```
sage: show(plot(digraphs.Circuit(5)))
```



..index:: Graph, matriz de adyacencia, DiGraph

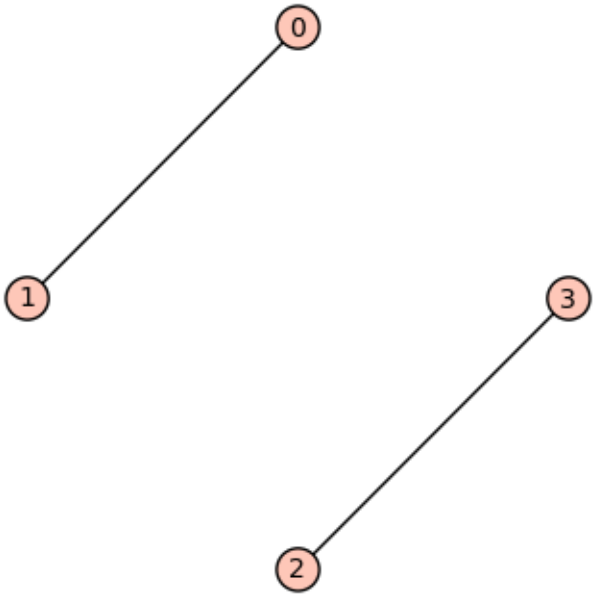
Introducir un grafo mediante la matriz de adyacencia

También podemos introducir un grafo usando la *matriz de adyacencia* : una matriz *KxK* (donde *K* es el número de vértices), que tiene un 1 en la posición *i,j* si y sólo si hay una arista entre los vértices *i* y *j* . Para grafos no dirigidos, la matriz debe ser simétrica.

Nota : En la documentación de `Graph` puedes encontrar otras formas de introducir un grafo (por ejemplo, mediante un diccionario asigna a cada vertice la lista de sus vecinos).

```
sage: M = matrix([[0,1,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]])
sage: show(M)
sage: g4 = Graph(M)
sage: show(g4, layout='circular')
```

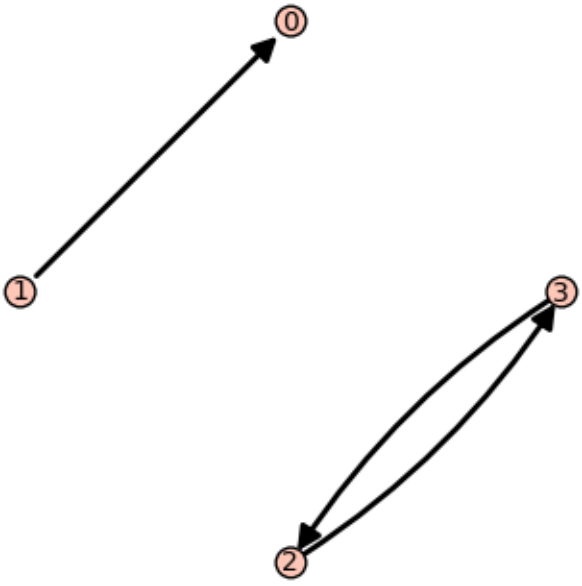
$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



Podemos construir del mismo modo un grafo dirigido, con una matriz no necesariamente simétrica, usando `DiGraph` .

```
sage: M = matrix(ZZ,[[0,0,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]])
sage: show(M)
sage: g4 = DiGraph(M)
sage: show(g4, layout='circular')
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$



El método `adjacency_matrix` devuelve la matriz de adyacencia de un grafo, independientemente de cómo lo introdujimos.

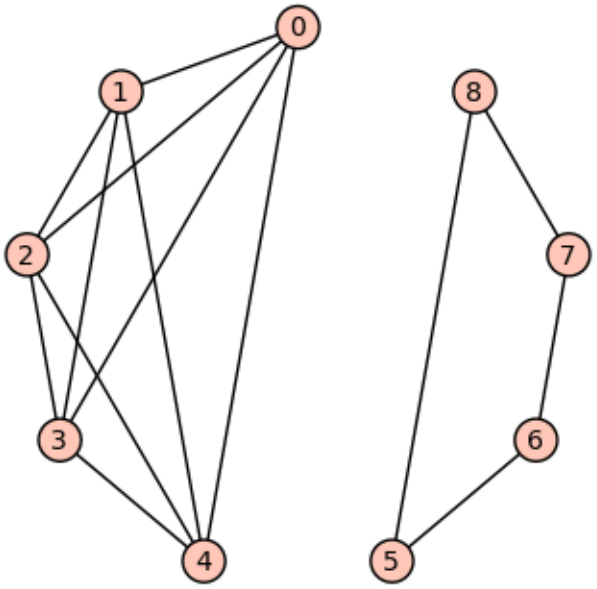
```
sage: g1.adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]
```

5.4.2 Operaciones con grafos

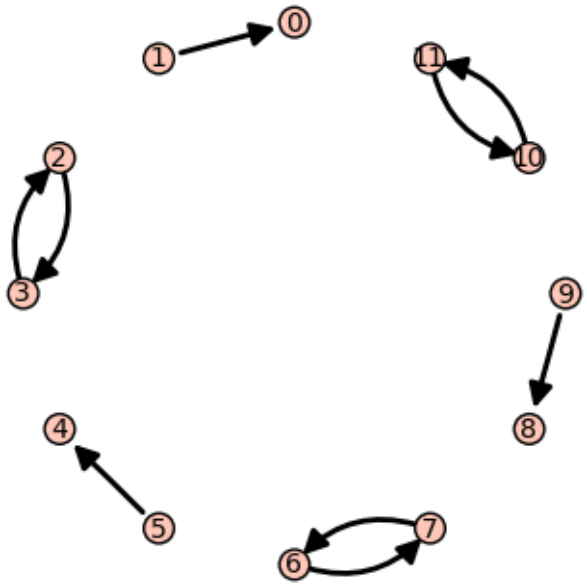
También podemos construir nuevos grafos a partir de otros.

- La suma de grafos devuelve la unión de los dos grafos.
- El producto de un grafo por un entero repite un grafo.

```
sage: g5 = g1 + g2
sage: show(g5,layout='circular')
```



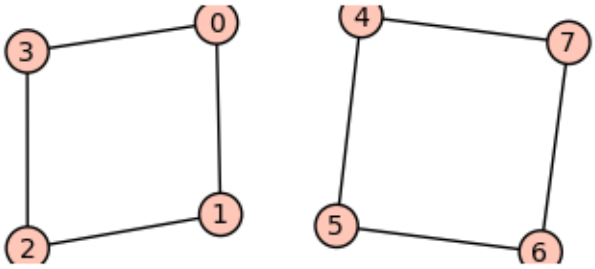

```
sage: g6 = g4*3
sage: show(g6,layout='circular')
```

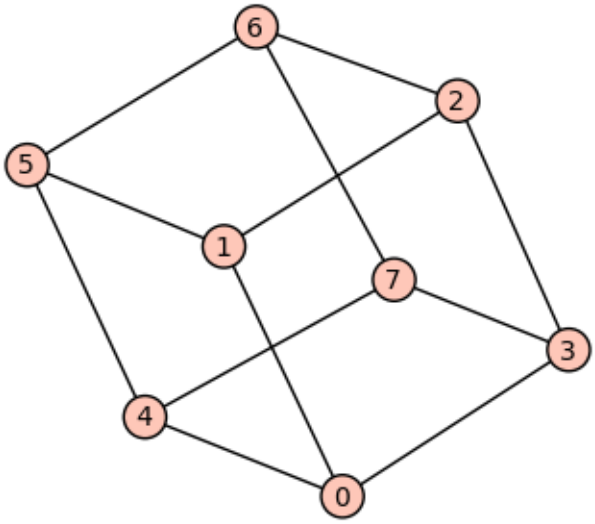


5.4.3 Modificar grafos

Podemos añadir y quitar vértices y aristas a grafos existentes usando `add_vertex`, `add_edge`, `delete_vertex`, `delete_edge`.

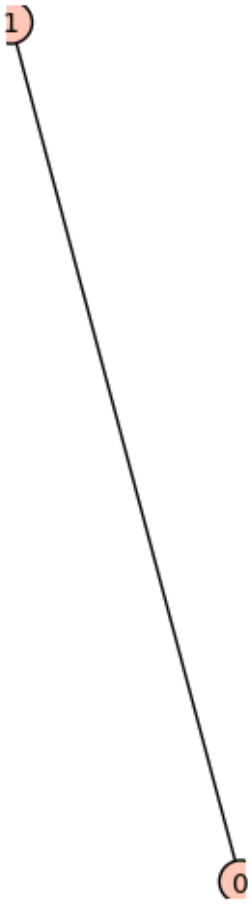
```
sage: g7 = 2*g2
sage: show(g7,layout='spring')
sage: g7.add_edge(0,4)
sage: g7.add_edge(1,5)
sage: g7.add_edge(2,6)
sage: g7.add_edge(3,7)
sage: show(g7,layout='spring')
```





Podemos partir de un grafo vacío y añadir los vértices y aristas necesarios:

```
sage: g8 = Graph()
sage: g8.add_vertex(0)
sage: g8.add_vertex(1)
sage: g8.add_edge(0,1)
sage: plot(g8)
```



Ejercicio

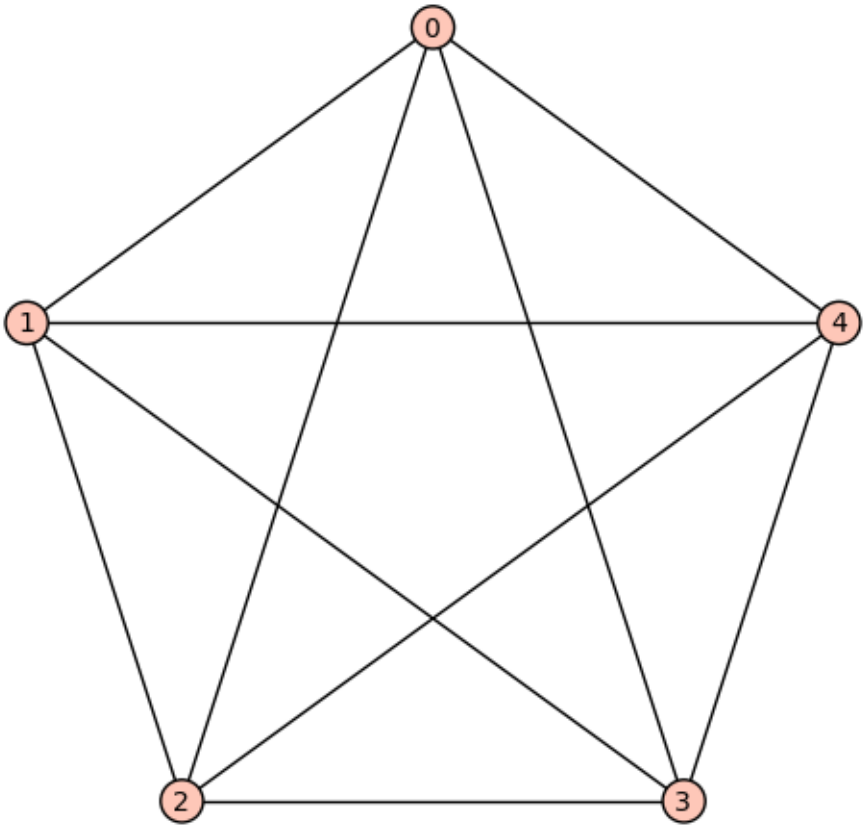
Modifica el grafo g6 añadiendo un vértice y uniendo todos los otros vértices al nuevo vértice.

5.4.4 Propiedades de los grafos

Podemos verificar un buen número de propiedades de un grafo usando los métodos adecuados. Por ejemplo:

- `is_connected` : comprueba si el grafo es **conexo**
- `is_planar` : comprueba si el grafo es **plano** . Un grafo es plano si se pueden dibujar los vértices y las aristas en un plano sin que las aristas se intersequen.
- `is_eulerian` : comprueba si el grafo tiene un **circuito euleriano** . Un circuito en un grafo es una sucesión de aristas adyacentes que comienza y termina en el mismo vértice. Un circuito euleriano es un circuito que pasa exactamente una vez por cada arista.
- `is_tree` : comprueba si el grafo es un **árbol** . Un árbol es un grafo conexo que no tiene *ningún circuito cerrado* .

```
sage: print g1.is_connected()
sage: print g1.is_planar()
sage: print g1.is_eulerian()
sage: print g1.is_tree()
sage: show(g1.plot())
True
False
True
False
```



Criterio de Euler

Según el criterio de Euler, un grafo tiene un circuito euleriano si y sólo todos los vértices tienen grado par.

Ejercicio

Comprueba el criterio de Euler para decidir si un grafo es euleriano.

```
sage: g5.degree()
[4, 4, 4, 4, 4, 2, 2, 2, 2]
```

5.4.5 Isomorfismo de grafos

Dos grafos son isomorfos si y sólo si existe una biyección del conjunto de vértices del primero en el conjunto de vértices del segundo tal que dos vértices están unidos en el primer grafo si y sólo si los vértices correspondientes están unidos en el segundo.

En dos grafos isomorfos, los vértices pueden tener nombres distintos y estar colocados en distintas posiciones, pero todas las relaciones de incidencia y todas las propiedades de grafos como conexión, planaridad etćetera son idénticas.

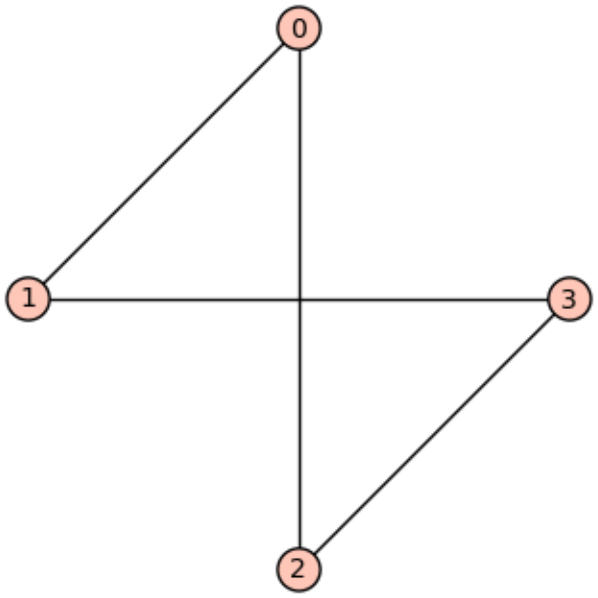
```
sage: print g7.is_isomorphic(g3)
sage: print g7 == g3
sage: print g3.vertices()
```

```
sage: print g7.vertices()
True
False
['000', '001', '010', '011', '100', '101', '110', '111']
[0, 1, 2, 3, 4, 5, 6, 7]
```

Pregunta: ¿cuál de los grafos que definimos antes es isomorfo al siguiente grafo?

```
sage: M = matrix(ZZ,[[0,1,1,0],[1,0,0,1],[1,0,0,1],[0,1,1,0]])
sage: show(M)
sage: g9=Graph(M)
sage: show(g9,layout='circular')
```

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$



La función `graphs` genera un grafo de cada clase de isomorfismo de grafos con un cierto número de vértices.

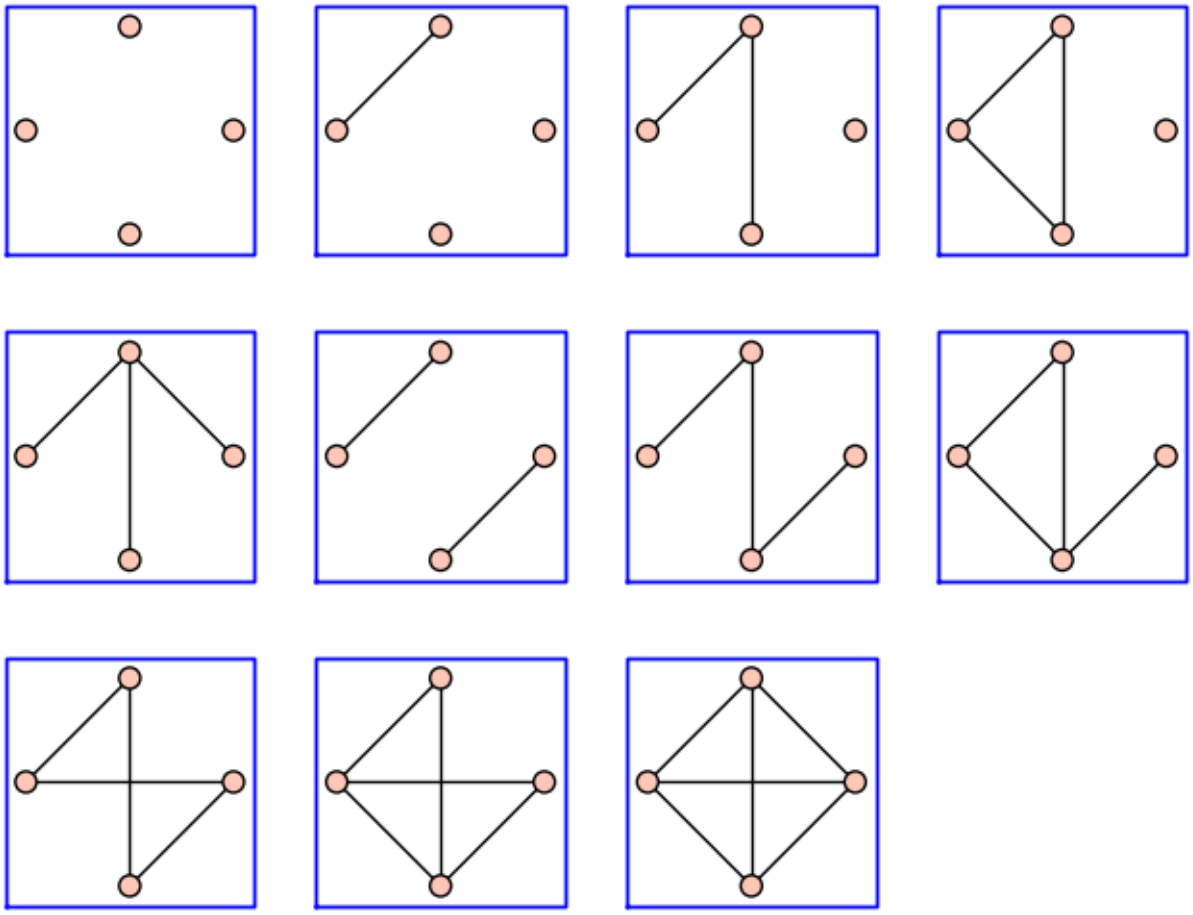
```
sage: for g in graphs(4):
...       if g.is_connected():
...           print g.adjacency_matrix()
...           print
[0 1 1 1]
[1 0 0 0]
[1 0 0 0]
[1 0 0 0]

[0 1 1 0]
[1 0 0 0]
[1 0 0 1]
[0 0 1 0]
```

```
[0 1 1 0]
[1 0 1 0]
[1 1 0 1]
[0 0 1 0]

[0 1 1 0]
[1 0 0 1]
[1 0 0 1]
[0 1 1 0]

[0 1 1 0]
[1 0 1 1]
[1 1 0 1]
[0 1 1 0]

[0 1 1 1]
[1 0 1 1]
[1 1 0 1]
[1 1 1 0]
```

```
sage: L = list(graphs(4))
```

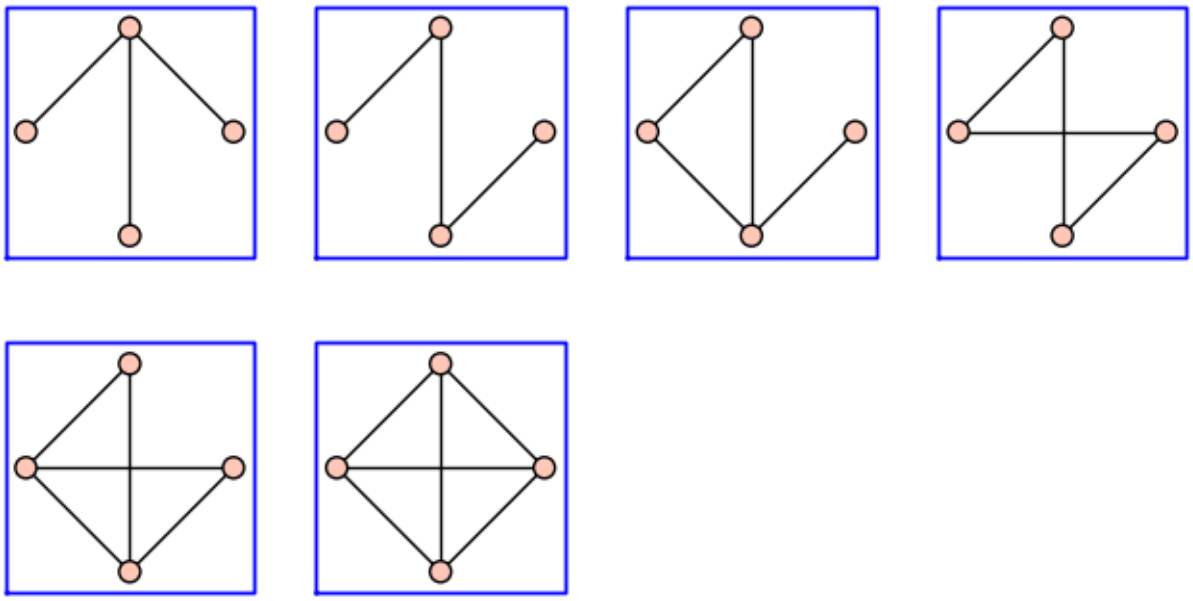
El siguiente método es una forma de dibujar un montón de grafos en poco espacio.

```
sage: graphs_list.show_graphs(L)
```



Podemos generar todos los grafos con un cierto número de vértices y contar el número de ellos que verifican una cierta propiedad.

```
sage: L = [g for g in graphs(4) if g.is_connected()]
sage: print len(L)
sage: graphs_list.show_graphs(L)
6
```



5.5 Ejercicios

5.5.1 1.

- ¿Cuales de los siguientes grafos son planos: `graphs.HouseXGraph`, `graphs.DesarguesGraph`, `graphs.OctahedralGraph` y `graphs.DiamondGraph` ? ¿Cuáles tienen circuitos eulerianos?
- ¿Qué propiedades especiales tienen los grafos de las familias `graphs.StarGraph` y `graphs.CircularLadderGraph` ? ¿son planos? ¿son árboles?
- ¿Para qué valores de k el grafo `graphs.CubeGraph(k)` tiene un circuito euleriano?
- ¿Para qué valores de k el grafo `graphs.CompleteGraph(k)` tiene un circuito euleriano?
- ¿Para qué valores de k el grafo `graphs.CompleteGraph(k)` es plano? Conjetura una respuesta, y confirma la respuesta buscando en interneto en la biblioteca.

5.5.2 2. Subgrafo inducido

Dado un grafo G con un conjunto de vértices V, y un subconjunto V' de V, el subgrafo de G **inducido** por V' es el grafo con V' como vértices, y sólo aquellas aristas que estaban en G.

- Encuentra tres subgrafos inducidos de 6 vértices del grafo `graphs.CubeGraph(3)` no isomorfos entre sí.
- ¿Cuántos subgrafos inducidos de 7 vértices tiene el grafo `graphs.CubeGraph(3)` no isomorfos entre sí?

- Encuentra un subgrafo inducidos del grafo `graphs.CubeGraph(4)` que sea isomorfo al grafo `graphs.CubeGraph(3)`

5.5.3 3. Contar grafos

¿Cuántos posibles grafos existen con un conjunto de vértices dado de tamaño k? ¿Cuántos *digrafos* (grafos dirigidos)?

¿Cuántos posibles grafos no isomorfos existen con exactamente k vértices, para k<=7?

¿Cuántos posibles *digrafos* conexos no isomorfos existen con 4 vértices ?

¿Cuántos grafos hay con 5 vértices que no sean planos? Dibújalos todos:

Referencia: http://es.wikipedia.org/wiki/Grafo_plano

5.5.4 4. Coloraciones de grafos

Una coloración de un grafo es una asignación de un color a cada vértice del grafo de modo que dos vértices adyacentes tienen asignados colores distintos. Las coloraciones más interesantes son las que tienen menor número de colores. Un conocido teorema afirma que todos los grafos planos se pueden colorear con a lo sumo 4 colores.

- Explora la ayuda de los métodos `coloring` y `plot` de los grafos para conseguir dibujar grafos junto con una coloración que tiene el mínimo número de colores.
- Encuentra grafos con k vértices que no se puedan colorear con menos de j colores, para j<k y k hasta 6.
- ¿Cuál es el número cromático del grafo `graphs.LollipopGraph(a,b)` ?
- ¿Cuál es el número cromático del grafo `graphs.BarbellGraph(a,b)` ? ¿Cuál es su número de vértices?

5.5.5 5. Grafo de divisores

Dada una lista de números S, construye el grafo de divisores de S, cuyos vértices son los números de S, y que tiene una arista desde j hasta k si y sólo si j es divisor de k ó j es divisor de k.

5.5.6 6.

Calcula el número de aristas del grafo de divisores para las listas S de la forma range(2,k), para distintos valores de k y dibújalas en una gráfica.

5.5.7 7.

El grafo de divisores de una lista de primos no tiene ninguna arista.

El grafo de divisores de una lista de potencias de 2 de tamaño k tiene todas las aristas posibles (es decir, k*(k+1)/2).

- ¿Puedes encontrar un grafo con k vértices y 2*k aristas, para k=10, 15 y 20?

Indicación: usa la fuerza (bruta).

Referencia: http://arxiv.org/abs/math/0606483

5.6 Experimentos con numeros aleatorios

En la primera sesión de este bloque aprendimos a recorrer exhaustivamente un conjunto de posibilidades para contar las que satisfacían una propiedad, y así calcular probabilidades como el número de casos favorables dividido por el número de casos posibles. En algunos casos el número de posibilidades es enorme, y el resultado, realmente mucho más preciso de lo que necesitábamos.

Una alternativa consiste en calcular sólo unos cuantos casos extraídos aleatoriamente, y hacer una estadística de los casos favorables entre los posibles. A esta estrategia se le llama en general el **método de Monte Carlo**, por su uso del azar. Aunque el resultado sólo es aproximado, es fácil controlar el tiempo de cómputo que destinamos a aproximar la solución.

Por supuesto, en muchos casos es posible hacer el cálculo exacto mediante razonamiento puro y, aún en los casos en que no es así, el razonamiento puede servir para reducir el número de posibilidades a explorar, o hacer más eficiente un método de Monte Carlo.

5.6.1 Ejemplo: galletas con pasas

Como ejemplo, nos planteamos este problema: partiendo de una masa para hacer galletas que contiene un total de P pasas, hacemos G galletas. Asumiendo que cada pasa terminará en una de las G galletas con la misma probabilidad y de forma independiente a la galleta de destino de las otras pasas, calcula la probabilidad de que cada galleta tenga al menos *k* pasas.

Para distribuir las pasas de todas las formas posibles, contamos cada posible lista de P valores

$$[G_1, G_2, \ldots, G_P]$$

donde el primer valor indica la galleta en que termina la primera pasa, y así sucesivamente. Como cada pasa puede terminar en una de las G galletas posibles, el número total de posibilidades es G^P.

```
sage: def pasas(G,P,k):
...       '''Calcula cuantas formas de repartir P pasas entre G galletas
...       dejan al menos k pasas en cada galleta
...       '''
...       favorables = 0
...       #Cada pasa termina en una galleta distinta, total G^P posibilidades
...       for j in srange(G^P):
...           #lista que indica en que galleta esta cada pasa
...           donde_esta_la_pasa = j.digits(base=G,padto=P)
...           #G galletas, que comienzan sin pasas
...           pasas_en_cada_galleta = [0]*G
...           #Contamos el numero de pasas en cada galleta
...           for g in donde_esta_la_pasa:
...               pasas_en_cada_galleta[g] += 1
...           if min(pasas_en_cada_galleta)>=k:
...               favorables += 1
...       return favorables/G^P

sage: %time
sage: pasas(4,6,1)
195/512
CPU time: 0.17 s,  Wall time: 0.19 s

sage: %time
sage: pasas(3,7,1)
```

```
602/729
CPU time: 0.09 s,  Wall time: 0.10 s

sage: %time
sage: pasas(4,7,1)
525/1024
CPU time: 0.66 s,  Wall time: 0.67 s
```

Como vemos, este enfoque requiere mucho tiempo incluso para cantidades pequeñas de pasas y de galletas. No en vano hay que iterar el bucle principal ¡¡G^P veces!!. Como el problema no tiene mucha gracia si el número de pasas no es al menos igual al número de galletas, estamos hablando de crecimiento exponencial. Aunque se pueden usar varios trucos para reducir el tiempo de cómputo, sigue siendo un crecimiento muy rápido.

Usando el método de Monte Carlo, decidimos exactamente el número de veces que iteramos el bucle.

Nota: este problema se puede resolver de forma exacta, pero no es trivial.

Nota: la lista de galletas de cada pasa contiene información redundante para este problema, en el que sólo nos interesa el total de pasas en cada galleta. Por ejemplo, podemos usar `IntegerVectors` para recorrer las listas de enteros de longitud G que suman P (el número de pasas en cada galleta), pero distintas formas de recorrer el conjunto pueden responder a distintos modelos de probabilidad.

```
sage: def pasas_mc(G,P,k,T):
...       '''Calcula cuantas formas de repartir P pasas entre G galletas
...       dejan al menos k pasas en cada galleta, usando un método de
...       Monte Carlo con muestra de tamaño T
...       '''
...       favorables = 0
...       for h in xrange(T):
...           #lista que indica en que galleta esta cada pasa
...           #Usamos randint que devuelve un entero escogido
...           #aleatoriamente en un rango
...           donde_esta_la_pasa = [randint(0,G-1) for j in range(P)]
...           #G galletas, que comienzan sin pasas
...           pasas_en_cada_galleta = [0]*G
...           #Contamos el numero de pasas en cada galleta
...           for g in donde_esta_la_pasa:
...               pasas_en_cada_galleta[g] += 1
...           if min(pasas_en_cada_galleta)>=k:
...               favorables += 1
...       return favorables/T

sage: #Distintas llamadas a esta funcion con los mismos argumentos
sage: #no devuelven el mismo resultado, debido al uso de numeros
sage: #aleatorios
sage: #Cuanto mayor la muestra, menor la oscilacion
sage: p = pasas_mc(4,7,1,1000)
sage: print p, p.n()
63/125 0.504000000000000
```

5.6.2 Simulaciones

El método de Monte Carlo permite abordar problemas que no se pueden resolver por fuerza bruta contando todas las posibilidades.

En el siguiente ejemplo sencillo, lanzamos monedas al aire (monedas equilibradas, lanzamientos independientes) hasta que acumulamos un cierto número de caras, ¿cuántos lanzamientos necesitamos en promedio para obtener al menos k caras?

Podemos simular este experimento usando llamadas a `randint(0,1)` para simular lanzamientos de monedas. Si el número devuelto es 0, lo tomamos como una cruz, si es un 1, como una cara. Repetimos el experimento hasta que acumulemos k caras, y tomamos nota del número de lanzamientos que hemos necesitado.

```
sage: def geometrica_mc(k,T):
...       '''Número medio de lanzamientos hasta obtener k caras, por
...       el método de Monte Carlo
...       '''
...       lanzamientos = 0
...       for j in range(T):
...           caras = 0
...           while caras<k:
...               lanzamientos += 1
...               caras += randint(0,1)
...
...       promedio = lanzamientos/T
...       return promedio

sage: m = geometrica_mc(4,10000)
sage: print m, m.n()
79953/10000 7.99530000000000
```

En este caso, obtenemos la confirmación no de la fuerza bruta, sino del razonamiento, porque el número esperado de lanzamientos es claramente 2k (hecho que sabréis demostrar cuando estudiéis la *binomial negativa* en probabilidad).

5.6.3 Ejemplo: estadísticas de matrices

¿Cuántas matrices cuadradas cuyas entradas sean 0 ó 1 son invertibles en $\mathbf{Z}_2$? Podemos dar a este enunciado una interpretación probabilista: ¿cuál es la probabilidad de que una matriz KxK en $\mathbf{Z}_2$ cuyas entradas han sido escogidas de forma aleatoria e independiente entre 0 y 1 sea invertible?

Es fácil generar una matriz cuadrada cuyas entradas son enteros escogidos aleatoriamente entre 0 y 1. Para calcular su determinante en $\mathbf{Z}_2$, calculamos su determinante habitual y tomamos el resto de dividir por 2. Alternativamente, podríamos usar el cuerpo `Integers(2)` y su método `random_element()` .

Recordamos una forma cómoda de construir matrices en Sage:

```
M = matrix(ZZ, K1, K2, lista)
```

donde K1 es el número de filas y K2 el número de columnas y, en vez de pasar una lista con las filas, pasamos una sóla lista con K1xK2 elementos.

También podemos usar `random_matrix` , pero es un método menos general.

```
sage: K=6
sage: #Generar una matriz con entradas 0 o 1 escogidas de forma aleatoria
sage: lista = [randint(0,1) for j in range(K*K)]
sage: M = matrix(ZZ,K,K, lista)
sage: print M, det(M)%2
[0 0 1 0 0 1]
[0 0 1 0 1 1]
[1 1 1 1 1 0]
[1 0 1 0 0 1]
[1 0 1 0 1 1]
[0 1 0 1 1 0] 0
```

Ejercicio

Calcula la probabilidad por fuerza bruta contando los casos favorables y por el método de Monte Carlo.

5.6.4 Lanzamientos no equiprobables

¿Y si cambiamos la probabilidad, y hacemos el 1 más probable que el 0? ¿Conseguiremos con ello aumentar la probabilidad de obtener matrices invertibles?

Para plantearnos este problema necesitamos extraer un 0 ó un 1 con una probabilidad $p \in [0, 1]$. La manera estándar de hacer ésto es usar la función random(), que devuelve un número aleatorio entre 0 y 1. Si el número extraido es menor que p, anotamos un 1 y si es mayor, anotamos un 0.

```
sage: K = 6
sage: p = 0.7
sage: #Construimos una matriz KxK de ceros:
sage: M = matrix(ZZ,K,K)
sage: #Rellenamos las entradas de la matriz con 0 o 1:
sage: for j in range(K):
...       for k in range(K):
...           t = random()
...           if t<p:
...               M[j,k]=1
sage: print M
[1 1 1 1 1 1]
[1 1 1 0 1 1]
[1 0 1 1 1 0]
[1 1 1 1 1 1]
[1 1 0 1 1 0]
[1 0 1 1 0 1]

sage: #codigo alternativo
sage: def aleatoriop(p):
...       if random()<t:
...           return 1
...       return 0
sage: p = 0. 7
sage: lista = [aleatoriop(p) for j in range(K*K)]
sage: M = matrix(ZZ,K,K, lista)
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

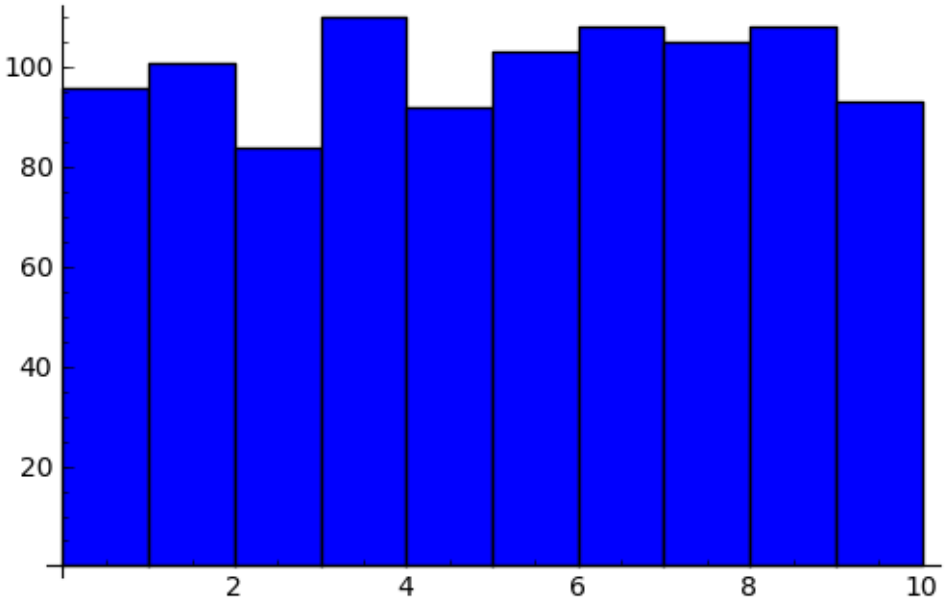
5.6.5 Discusión: ¿cómo de aleatorios son los números aleatorios?

En esta sesión hemos usado los generadores de números aleatorios del ordenador, llamando a las funcioens `randint` y `random` . Nuestros resultados se basan en que los números generados sean lo bastante aleatorios. Vamos a comprobar esta afirmación haciendo estadísticas de los números obtenidos.

Comenzamos con `randint` , generando números enteros aleatorios, guardando la frecuencia con que aparece cada resultado, y mostrando las frecuencias en un gráfico de barras.

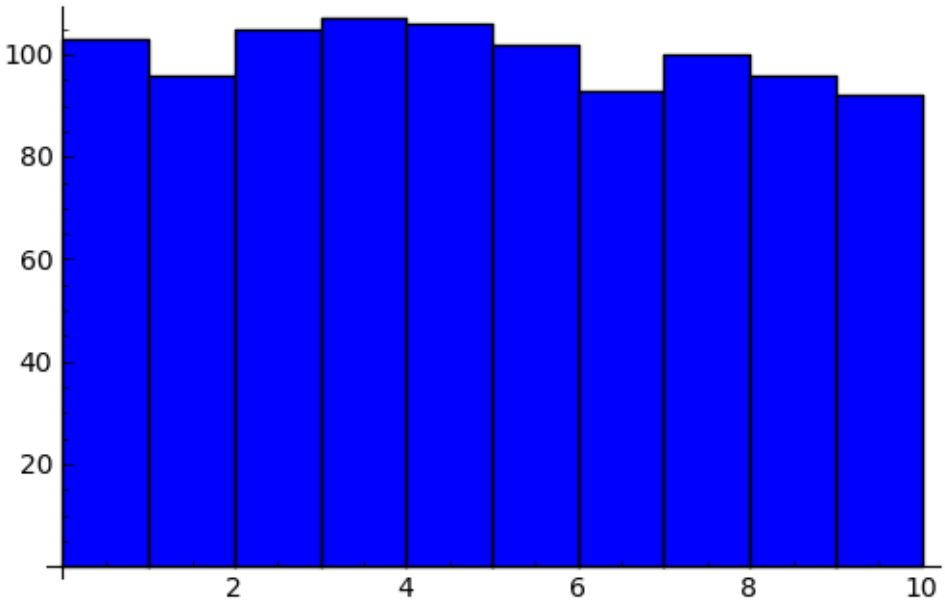
```
sage: L = 10
sage: T = 1000
sage: frecuencias = [0]*L
sage: for j in range(T):
```

```
...       k = randint(0,L-1)
...       frecuencias[k] += 1
sage: #Mediante ymin=0, indicamos que muestre las graficas
sage: #partiendo desde el eje de las x
sage: #(prueba a quitarlo y veras que pasa)
sage: bar_chart(frecuencias,width=1).show(ymin=0)
```



Para comprobar `random`, nos limitamos a separar los números entre 0 y 1 en L subintervalos distintos, y luego comprobar que en cada caja tenemos aproximadamente 1/L números. No tenemos teoría suficiente para hacer tests más sofisticados.

```
sage: L = 10
sage: T = 1000
sage: frecuencias = [0]*L
sage: for j in range(T):
...       a = random()
...       k = floor(a*L)
...       frecuencias[k] += 1
sage: #Mediante ymin=0, indicamos que muestre las graficas
sage: #partiendo desde el eje de las x
sage: #(prueba a quitarlo y veras que pasa)
sage: bar_chart(frecuencias,width=1).show(ymin=0)
```



En realidad, los números generados con `randint` y `random` son pseudo-aleatorios: son aleatorios a efectos prácticos pero están generados de manera casi determinista: primero se escoge una **semilla** , y a partir de ahí se usa una función de apariencia aleatoria para obtener el siguiente número aleatorio a partir del anterior.

Más detalles (en inglés) en:

http://docs.python.org/library/random.html

Si fijamos la semilla, todos los cálculos posteriores con numeros aleatorios generan los mismos números, aunque lo ejecutemos en distintos momentos o en distintos ordenadores. Esto es importante para reproducir exactamente un cálculo con números aleatorios.

```
sage: #Fijamos la semilla para generar números aleatorios
sage: set_random_seed(123)
sage: print random()
sage: print random()
sage: print random()
sage: print randint(1,10)
sage: print randint(1,10)
sage: print randint(1,10)
0.220001316661
0.778780038062
0.0648345056353
8
2
1
```

5.7 Ejercicios

5.7.1 1.

Aproxima el número π por el método de Monte Carlo

Genera pares (x,y) de números aleatorios entre -1 y 1. Estos pares de números están escogidos en el cuadrado [-1,1]x[-1,1] de forma uniforme. Si contamos el ratio de los pares (x,y) tales que $x^2 + y^2 < 1$ entre el total de pares, obtenemos

el área del círculo unidad dividido entre el área del cuadrado unidad.

- Utiliza esta idea para aproximar el número π

5.7.2 2.

Estima mediante el método de Monte Carlo las probabilidades de diversos eventos relacionados con lanzamientos de monedas que calculaste en el ejercicio de la sección 1.

5.7.3 3.

Los paquetes de cereales ACME traen seis tipos distintos de regalos. Fulanito está dispuesto a comprar cajas de cereales hasta tenerlos todos.

Asumiendo que la probabilidad de encontrar cada tipo de regalo en una caja de cereales es la misma, e independiente de las otras cajas de cereales, estima mediante el método de Monte Carlo el número de cajas que tendrá que comprar Fulanito hasta tener al menos un regalo de cada tipo.

5.7.4 4. Cumpleaños

Estima mediante el método de Monte Carlo la probabilidad de que en un grupo de 30 personas, al menos dos de ellas cumplan años el mismo día.

Para ello, asume que cada persona del grupo tiene un cumpleaños elegido aleatoriamente entre 1 y 365, y que el cumpleaños de cada uno es independiente del cumpleaños de los demás.

Nota: es posible resolver este problema por razonamiento, pero no por fuerza bruta. El número de posibles fechas de cumpleaños para las 30 personas es $365^{30} \approx 7 \cdot 10^{76}$.

5.7.5 5. Estadísticas con grafos

Generamos un grafo aleatorio de la forma siguiente: partimos de un conjunto de k vértices y, con probabilidad p , ponemos una arista entre cada dos vértices, haciendo cada elección de forma independiente. Nos preguntamos cuál es la probabilidad de obtener un grafo con una cierta propiedad.

- Calcula la probabilidad de obtener un grafo plano cuando k=6, p=0.6 y cuando k=12, p=0.3.
- Calcula la probabilidad de obtener un grafo conexo cuando k=6, p=0.5.

Indicación : construye la matriz de incidencia de forma aleatoria, con las entradas 0 y 1. Ten cuidado, porque la matriz debe ser simétrica. Si decides que entre los vértices j y k debe haber una arista, tienes que poner unos en las entradas M[j,k] y M[k,j] de la matriz de incidencia. Alternativamente, busca código en Sage que genera estos grafos.

Bloque V: Cálculo simbólico

Una introducción al cálculo simbólico. En primer lugar se estudian las expresiones simbólicas, y las ecuaciones, y el comando `solve`. Después se aplica lo aprendido al cálculo en una y varias variables. La última sesión es una introducción al estudio de las curvas en el plano.

6.1 Cálculo simbólico

6.1.1 Variables y expresiones simbólicas

Una **variable simbólica** es un objeto en python que representa una incógnita que puede tomar cualquier valor. Por tanto, una operación que involucra una o más variables simbólicas no devuelve un valor numérico, sino una **expresión simbólica** que involucra números, operaciones, funciones y variables simbólicas. Si en algún momento sustituimos las variables simbólicas por números (o elementos de un anillo), podremos realizar las operaciones y obtener un número (o un elemento de un anillo).

Esta forma de trabajar se corresponde con los criterios de "no hacer las operaciones hasta el final", "dejar las operaciones indicadas" y "cancelar términos antes de evaluar" que aprendistéis en secundaria.

```
sage: #Declaramos las variables a b c
sage: var('a b c')
sage: #Definimos varias expresiones simbolicas
sage: s = a + b + c
sage: s2 = a^2 + b^2 + c^2
sage: s3 = a^3 + b^3 + c^3
sage: s4 = a + b + c + 2*(a + b + c)
sage: p = a*b*c
sage: #Las imprimimos como código
sage: print s
sage: print s2
sage: print s3
sage: print s4
sage: print p
sage: #Las mostramos en el formato matemático usual
sage: show(s2)
sage: show(s3)
a + b + c
a^2 + b^2 + c^2
a^3 + b^3 + c^3
```

```
3*a + 3*b + 3*c
a*b*c
```

$$a^2 + b^2 + c^2$$

$$a^3 + b^3 + c^3$$

```
sage: #Sustituimos las variables por numeros
sage: print s(a=1, b=1, c=1)
sage: print s(a=1, b=2, c=3)
3
6

sage: #Operamos con las expresiones
sage: ex = (1/6)*(s^3 - 3*s*s2 +2*s3 )
sage: show(ex)
```

$$\frac{1}{6}\,(a+b+c)^3 + \frac{1}{3}\,a^3 + \frac{1}{3}\,b^3 + \frac{1}{3}\,c^3 - \frac{1}{2}\,(a+b+c)\left(a^2+b^2+c^2\right)$$

..index:: simplifciar, expand, factor, simplify_rational, simplify_trig, simplify_exp

Simplificar expresiones

Sin embargo, pese a que no es posible evaluar una expresión hasta el final, observamos que al crear la expresión se han realizado "*de oficio*" algunas simplificaciones triviales. En ejemplos como el de arriba, nos puede interesar simplificar la expresión todavía más, pero es necesario decir qué queremos exactamente. Existen varias estrategias para intentar simplificar una expresión, y cada estrategia puede tener más o menos éxito dependiendo del tipo de expresión simbólica. Algunas estrategias dan lugar a una expresión más sencilla en algunos casos pero no en otros, y con expresiones complicada pueden consumir bastante tiempo de proceso.

Para la expresión anterior, como tenemos un polinomio, es buena idea expandirla en monomios que se puedan comparar unos con otros, usando el método `expand` .

```
sage: show(ex.expand())
```

$$abc$$

A menudo nos interesa lo contrario: factorizar la expresión usando `factor` .

```
sage: p = a^3 + a^2*b + a^2*c + a*b^2 + a*c^2 + b^3 + b^2*c + b*c^2 + c^3
sage: show(p)
sage: show(p.factor())
```

$$a^3 + a^2b + a^2c + ab^2 + ac^2 + b^3 + b^2c + bc^2 + c^3$$

$$(a+b+c)\left(a^2+b^2+c^2\right)$$

Si observas con el tabulador los métodos de las expresiones regulares, verás que hay métodos específicos para expresiones con funciones trigonométricas, exponenciales, con radicales o fracciones (es decir, con funciones racionales).

```
sage: p = sin(3*a)
sage: show(p.expand_trig())
```

$$-\sin(a)^3 + 3\sin(a)\cos(a)^2$$

```
sage: p = sin(a)^2 - cos(a)^2
sage: show(p.simplify_trig())
```

$$-2\cos(a)^2 + 1$$

```
sage: p = 2^a * 4^(2*a)
sage: p.simplify_exp()
2^(5*a)

sage: p = 1/a - 1/(a+1)
sage: p.simplify_rational()
1/(a^2 + a)
```

Funciones simbólicas

A medio camino entre las expresiones simbólicas y las funciones de python, las funciones simbólicas son expresiones simbólicas en las que se fija el orden de las variables de entrada. Esto elimina la ambigüedad sobre qué variables se deben sustituir por qué argumentos y permite usarlas en sitios donde se espera una función de python.

```
sage: f(a,b,c) = a + 2*b + 3*c
sage: f(1,2,1)
8

sage: s = a + 2*b + 3*c
sage: s(1,2,1)
__main__:4: DeprecationWarning: Substitution using function-call syntax and unnamed arguments is depr
8
```

Diferencias entre polinomios y expresiones simbólicas

En algunos aspectos las expresiones simbólicas pueden parecer similares a los polinomios que vimos en el bloque III, lo que puede provocar confusión. Es importante saber si tenemos entre manos una expresión simbólica que representa un polinomio como ésta:

```
var('x')
p = x^2 + 1
```

o un polinomio como éste (con coeficientes por ejemplo en `RDF`):

```
R.<t> = PolynomialRing(RDF)
q = t^2 + 1
```

Aunque ambos tienen métodos de propósito similar como `coefficients` o `roots` , el comportamiento es distinto, a veces de manera obvia y a veces de manera más sutil. Por ejemplo, las raíces o los coeficientes de una expresión simbólica son a su vez expresiones simbólicas, mientras que en el caso de un polinomio son miembros del anillo de coeficientes.

```
sage: var('x')
sage: p = x^2 + 3
sage: print p.coefficients()
sage: print p.roots()
sage: termino_indi = p.coefficients()[0][0]
sage: print termino_indi, parent(termino_indi)
[[3, 0], [1, 2]]
[(-I*sqrt(3), 1), (I*sqrt(3), 1)]
3 Symbolic Ring

sage: R.<t> = PolynomialRing(RDF)
sage: q = t^2 + 3
sage: print q.coefficients()
sage: print q.roots()
sage: termino_indi = q.coefficients()[0]
sage: print termino_indi, parent(termino_indi)
[3.0, 1.0]
[]
3.0 Real Double Field
```

El método `polynomial` permite construir un polinomio con coeficientes en un anillo dado a partir de una expresión simbólica. Para hacer lo contrario, podemos sustituir la variable del anillo de polinomios por una variable simbólica, o convertir explícitamente al tipo de datos `SR` (el anillo de expresiones simbólicas, o Symbolic Ring).

```
sage: p.polynomial(ZZ).roots()
[]

sage: q0 = q(t=x); q0
x^2 + 3.0

sage: q1 = SR(q); q1
t^2 + 3.0
```

6.1.2 Ecuaciones e inecuaciones

Si usamos el operador de igualdad (==) o algún operador de comparación (<, <=, >, >=, !=) sobre dos expresiones simbólicas, obtenemos una **relación simbólica**. Al igual que las expresiones simbólicas, las operaciones en una relación están indicadas pero no se pueden ejecutar hasta que todas las variables han tomado un valor concreto.

Evaluar una relación

Para evaluar la relación y obtener un booleano, usamos el operador de conversión a booleano `bool(relacion)` . Algunas relaciones tienen un valor definido, incluso aunque dependan de variables simbólicas: son siempre ciertas o siempre falsas. Muchas otras tendrán un valor indeterminado: dependiendo de los valores de las variables. En estos casos, el valor booleano de la relación es `False` , siguiendo una convención usual en otros programas de cálculo simbólico. También obtenemos `False` si evaluamos una expresión demasiado compleja para el motor simbólico. Tomamos buena nota de la asimetría en los resultados:

- `True` : el programa puede demostrar que la relación se verifica siempre, para cualquier valor de las variables.

- `False` : el programa no puede demostrar que la relación se verifica siempre, para cualquier valor de las variables. Eso *no significa* que nunca se verifique, ni siquiera que sea *ambigua* (cierta para algunos valores de las variables y falsa para otros).

```
sage: var('a b')
sage: r1 = (a<a+1)
sage: r2 = (a<a-1)
sage: r3 = (a<b)
sage: r4 = (b<a)
sage: r5 = a > 0

sage: print bool(r1), bool(r2), bool(r3), bool(r4), bool(r5)
True False False False False
```

Para distinguir entre relaciones que se puede demostrar que son falsas y relaciones ambiguas (o demasiado complicadas para el software), podemos usar la negación de las relaciones. Vemos por ejemplo, que se puede demostrar que `r2` es siempre falsa porque su negación se resuelve a `True` .

```
sage: print r2.negation(), bool(r2.negation())
sage: print r3, bool(r3)
sage: print r3.negation(), bool(r3.negation())
a >= a - 1 True
a < b False
a >= b False
```

Por supuesto concretando los valores de algunas variables, una relación ambigua se puede concretar, y resultar cierta (o falsa).

```
sage: print bool(r2(a=1, b=2)), bool(r3(a=1, b=2))
sage: print bool(r2(a=3, b=2)), bool(r3(a=3, b=2))
sage: print bool(r5(a=e^b))
False True
False False
True
```

Aparte de preguntar por su valor de verdad, podemos simplificar y operar con las relaciones de forma similar a las formas de operar con una expresión cualquiera:

```
sage: r1 = (a<a+1)
sage: print r1.subtract_from_both_sides(a)
0 < 1

sage: print r3
sage: print r4
sage: r6 = r3 + r4
sage: print r6
sage: r6 = r6.subtract_from_both_sides(r6.right_hand_side())
sage: print r6
a < b
b < a
a + b < a + b
0 < 0
```

6.1.3 Resolver ecuaciones y sistemas de ecuaciones

El comando `solve` permite resolver ecuaciones (al menos lo intenta): para resolver una ecuación llamamos a solve con la ecuación como primer argumento y la variable a despejar como segundo argumento, y recibimos una lista con las soluciones.

```
sage: x = var('x')
sage: solve(x^2 + 3*x + 2 == 0, x)
[x == -2, x == -1]
```

Si en vez de pasar una igualdad pasamos una expresión `r` , calcula las soluciones de `r==0` .

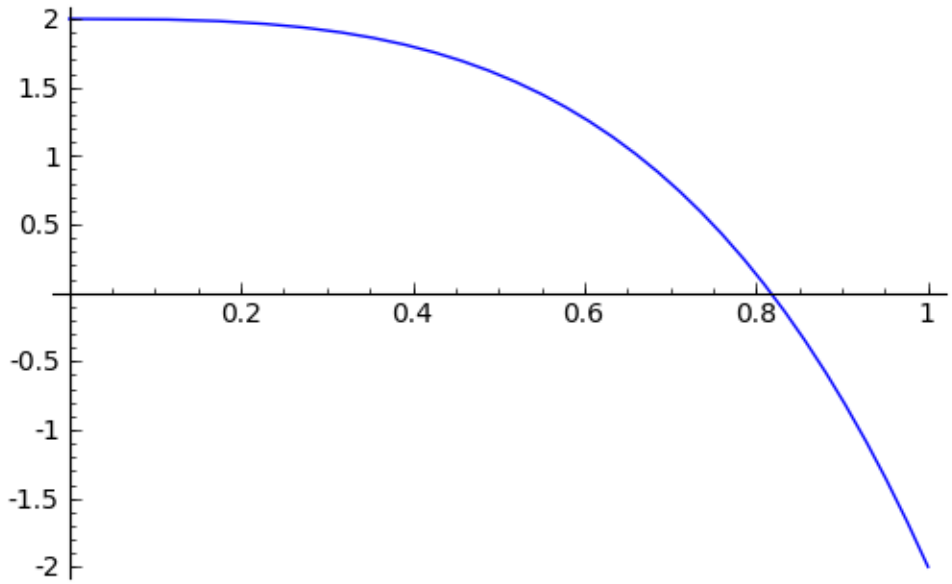
```
sage: #Las soluciones de esta ecuacion cubica son numeros complejos
sage: solve(x^3 - 3*x + 5, x)
[x == -1/2*(1/2*sqrt(21) - 5/2)^(1/3)*(I*sqrt(3) + 1) - 1/2*(-I*sqrt(3) + 1)/(1/2*sqrt(21) - 5/2)^(1/
```

A veces `solve` no puede resolver la ecuación, en cuyo caso puede que la simplifique un poco, o ni siquiera éso.

```
sage: solve(x^5 + 4*x^3 == x^3 + 2, x)
[0 == x^5 + 3*x^3 - 2]
```

Ante una ecuación arbitraria de quinto grado, podemos intentar encontrar las soluciones de forma numérica (volveremos a hablar de este tema).

```
sage: soluciones = solve(x^5 + 4*x^3 == x^3 + 2, x)
sage: s = soluciones[0]
sage: print s
sage: show(plot(s, x, 0, 1))     #dibujamos s entre 0 y 1
sage: print s.find_root(0,1)     #buscamos una raiz (numerica) entre 0 y 1
0 == x^5 + 3*x^3 - 2
0.816997207347
```



`solve` trabaja con expresiones con muchas variables simbólicas, e intenta despejar la variable que le pasamos como argumento y expresarla como función de las otras

```
sage: var('x y')
sage: for expresion in solve(x^4 + y^2*x^2 + 2, x ):
...       show(expresion)
sage: for expresion in solve(x^4 + y^2*x^2 + 2, y ):
...       show(expresion)
```

$$x = -\frac{1}{2}\sqrt{\sqrt{y^4 - 8} - y^2}\sqrt{2}$$

$$x = \frac{1}{2}\sqrt{\sqrt{y^4-8}-y^2}\sqrt{2}$$

$$x = \left(-\frac{1}{2}I\right)\sqrt{\sqrt{y^4-8}+y^2}\sqrt{2}$$

$$x = \left(\frac{1}{2}I\right)\sqrt{\sqrt{y^4-8}+y^2}\sqrt{2}$$

$$y = \frac{-\sqrt{-x^4-2}}{x}$$

$$y = \frac{\sqrt{-x^4-2}}{x}$$

Sistemas de ecuaciones

También podemos usar `solve` para resolver sistemas de ecuaciones (lineales o no). Como primer argumento pasamos una lista de ecuaciones y después sigue la lista de variables que queremos despejar.

Aplicar `solve` a sistemas lineales de ecuaciones es *totalmente equivalente a resolver el sistema mediante matrices* , aunque las distintas posibilidades (incompatible, compatible determinado...) se muestran de forma distinta.

```
sage: var('x1 x2 x3')
(x1, x2, x3)

sage: #Compatible determinado
sage: #Resolvemos el sistema usando solve
sage: eq1 = 2*x1       + x3 ==  1
sage: eq2 =   x1 +  x2 - x3 ==  0
sage: eq3 =  -x1 -2*x2      == -1
sage: print solve([eq1,eq2,eq3],x1,x2,x3)
sage: #Resolvemos el sistema usando matrices
sage: M = matrix(QQ,3,3,[2,0,1,  1,1,-1,  -1,-2,0 ])
sage: v = vector(QQ,[1,0,-1])
sage: M.solve_right(v )     #tambien vale M\v
[
[x1 == (1/5), x2 == (2/5), x3 == (3/5)]
]
(1/5, 2/5, 3/5)

sage: #Compatible indeterminado
sage: #Resolvemos el sistema usando solve
sage: eq1 = 2*x1       + x3 ==  1
sage: eq2 =   x1 +  x2 - x3 ==  0
sage: print solve([eq1,eq2],x1,x2,x3)
sage: #Resolvemos el sistema usando matrices
sage: M = matrix(QQ,2,3,[2,0,1,  1,1,-1])
sage: v = vector(QQ,[1,0])
sage: #solve_right solo encuentra una solucion
sage: print M.solve_right(v )     #tambien vale M\v
sage: print M.right_kernel()
[
[x1 == -1/2*r1 + 1/2, x2 == 3/2*r1 - 1/2, x3 == r1]
```

```
]
(1/2, -1/2, 0)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -3 -2]

sage: #Incompatible
sage: #solve devuelve silenciosamente una lista de soluciones vacia
sage: eq1 = 2*x1       + x3 ==  1
sage: eq2 =   x1 +  x2 - x3 ==  0
sage: eq3 =   x1 +  x2 - x3 ==  1
sage: print solve([eq1,eq2,eq3],x1,x2,x3)
sage: #M\v, o M.solve_right(v) arrojan un error
sage: M = matrix(QQ,3,3,[2,0,1,  1,1,-1,  1,1,-1])
sage: v = vector(QQ,[1,0,1])
sage: M\v
[

]
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

6.1.4 Asumir una igualdad o desigualdad

El comando `asume` informa a Sage de que cierta identidad no trivial que involucra algunas variables simbólicas se verifica. Esto es importante porque permite hacer ciertas simplificaciones extra. Por ejemplo:

```
sage: var("x")
sage: s = sqrt(x^2)
sage: simplify(s)
sqrt(x^2)

sage: assume(x>0)
sage: simplify(s)
x
```

El comando dual de `assume` es `forget` , que olvida una identidad que anteriormente habíamos asumido como cierta (la sintaxis es idéntica).

`forget()` sin argumentos olvida todas las identidades.

`assumptions()` muestra todas las identidades activas.

```
sage: assumptions()
[x > 0]

sage: forget(x>0)
sage: simplify(s)
sqrt(x^2)

sage: assumptions()
[]
```

También podemos asumir otro tipo de información, como por ejemplo que una variable representa un número entero.

```
sage: var('n')
sage: assume(n, 'integer')
```

```
sage: print sin(n*pi).simplify()
sage: forget(n, 'integer')
sage: print sin(n*pi).simplify()
0
sin(pi*n)

sage: var('k t')
sage: assume(k, 'integer')
sage: simplify(sin(t+2*k*pi))
sin(t)
```

Si tomamos por ciertas varias identidades incompatibles, Sage lanza un error:

```
sage: var('x')
sage: forget()
sage: assume(x<0)
sage: assume(x>0)
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
```

6.1.5 Ejemplos

Matrices con parámetros

Si usamos el anillo de expresiones simbólicas (`SR`) para los coeficientes de las matrices, podemos trabajar con parámetros.

```
sage: #Ejemplo de una matriz dependiendo de un parámetro
sage: var('a')
sage: M = matrix(SR,3,3, [1,2,3,  1,-3,1,  2,1,a])
sage: M
[ 1  2  3]
[ 1 -3  1]
[ 2  1  a]

sage: #Ejemplo: la matriz de van der Monde 3x3
sage: var('a1 a2 a3')
sage: VDM = matrix([[1, a1, a1^2], [1, a2, a2^2], [1, a3, a3^2] ])
sage: show(VDM)
```

$$\begin{pmatrix} 1 & a_1 & a_1^2 \\ 1 & a_2 & a_2^2 \\ 1 & a_3 & a_3^2 \end{pmatrix}$$

Ejercicio resuelto: Calcula el rango de la matriz

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 1 \\ 2 & 1 & a \end{pmatrix}$$

dependiendo del valor del parámetro a .

```
sage: var('a')
sage: M = matrix(SR,3,3, [1,2,3,  1,-3,1,  2,1,a])
sage: M
[ 1  2  3]
[ 1 -3  1]
[ 2  1  a]
```

Solución 1: ponemos la matriz en forma escalonada mediante operacinoes de fila y vemos que el rango depende de si a es 24/5 o no.

```
sage: M.add_multiple_of_row(1,0,-1)
sage: M
[ 1  2  3]
[ 0 -5 -2]
[ 2  1  a]

sage: M.add_multiple_of_row(2,0,-2)
sage: M
[    1     2     3]
[    0    -5    -2]
[    0    -3 a - 6]

sage: M.add_multiple_of_row(2,1,-3/5)
sage: M
[       1        2        3]
[       0       -5       -2]
[       0        0 a - 24/5]
```

Por supuesto, también podemos calcular el determinante

```
sage: var('a')
sage: M = matrix(SR,3,3, [1,2,3,  1,-3,1,  2,1,a])
sage: determinante = M.det()
sage: determinante.factor()
-5*a + 24
```

Sin embargo, la forma escalonada produce un resultado inesperado: como las expresiones simbólicas no nulas son invertibles, la matriz es equivalente a la identidad!

```
sage: M.echelon_form()
[1 0 0]
[0 1 0]
[0 0 1]
```

Ejercicio resuelto

Encuentra la recta afín que pasa por el punto $(1, 0, -1)$ y corta a las rectas L_1 y L_2, sabiendo que L_1 pasa por $(1, 0, 1)$ y $(2, 1, 0)$, y que L_2 pasa por $(3, 1, -1)$ y $(1, 2, -1)$.

```
sage: p1 = vector([1,0,1])
sage: p2 = vector([2,1,0])
sage: p3 = vector([3,1,-1])
sage: p4 = vector([1,2,-1])
sage: p5 = vector([1,0,-1])
sage: var('r s t')
sage: #Calculamos un punto generico de L1, y otro de L2
sage: pgL1 = (1-r)*p1+r*p2
sage: pgL2 = (1-s)*p3+s*p4
```

```
sage: #Un punto generico de la recta que pasa por pgL1 y pgL2
sage: pgL = (1-t)*pgL1 + t*pgL2
sage: pgL
(-(t - 1)*(r + 1) - (2*s - 3)*t, -(t - 1)*r + (s + 1)*t, (t - 1)*(r - 1) - t)

sage: #Imponemos las ecuaciones que representan que
sage: eqs = [c1==c2 for c1,c2 in zip(pgL,p5)]
sage: print eqs
sage: solve(eqs,r,s,t)
[-(t - 1)*(r + 1) - (2*s - 3)*t == 1, -(t - 1)*r + (s + 1)*t == 0, (t - 1)*(r - 1) - t == -1]
[[r == 2, s == (1/3), t == 3]]
```

La recta pasa por p5 y por (1-r)*p1+r*p2, con r=2, es decir $(3, 2, -1)$

```
sage: ((1-r)*p1+r*p2)(r=2)
(3, 2, -1)
```

Ejercicio: encuentra las ecuaciones implícitas y paramétricas de la recta solución

6.1.6 Truco avanzado: fast_float

Si vas a evaluar muchas veces una expresión simbólica, puedes "*compilarla*" usando el comando `fast_float` , que devuelve una función normal de python correspondiente a la expresión simbólica. Esta función ya no se puede manipular de forma simbólica, pero se puede evaluar usando mucho menos tiempo.

```
sage: var('a b c')
sage: f(a,b,c) = a^3 + sin(b + 3*log(c))
sage: timeit('f(1,2,1)')
sage: ff = fast_float(f)
sage: timeit('ff(1,2,1)')
625 loops, best of 3: 248 µs per loop
625 loops, best of 3: 13 µs per loop

sage: var('a b c')
sage: s = a^3 + sin(b + 3*log(c))
sage: timeit('s(a=1, b=2, c=1)')
sage: fs = fast_float(s, a, b, c)
sage: timeit('fs(1,2,1)')
625 loops, best of 3: 109 µs per loop
625 loops, best of 3: 13 µs per loop
```

6.2 Ejercicios

6.2.1 1.

Queremos encontrar todos los triángulos equilateros con vértices en tres rectas paralelas dadas.

- Plantea el problema usando variables simbólicas de la forma más directa posible, usando la definición de triángulo equilátero: los tres lados deben ser iguales, y observa como Sage falla miserablemente.
- Mastica un poco el problema, usando algunas de las sugerencias siguientes:
- Evita las raices cuadradas que tienen varias soluciones.
- Fija la primera recta al eje de las x.

- Fija el primer vértice del triángulo al origen de coordenadas.
- Usa una definición alternativa de triángulo equilátero: la rotación de ángulo 60º con vértice p1 lleva p2 a p3.
- Si fijamos dos rectas paralelas a distancia 1: ¿dónde debemos fijar la tercera recta para que el triángulo solución tenga área mínima? Usa por ejemplo la definición de área dadas las coordenadas de los vértices de la wikipedia en inglés: http://en.wikipedia.org/wiki/Area_of_a_triangle#Using_coordinates .

6.2.2 2.

- Calcula el centro y el radio de la circunferencia que pasa por tres puntos del plano, de coordenadas (x1,y1), (x2,y2) y (x3,y3).
- Intenta entender la fórmula obtenida para el radio del círculo, en función de conceptos geométricos intrínsecos como áreas, distancias y ángulos.

6.2.3 3.

Un teorema de álgebra afirma que todo polinomio simétrico $p(x_1, \ldots, x_n)$ se puede expresar en función de los polinomios simétricos elementales:

$$\begin{array}{rcl} e_0(x_1, \ldots, x_n) & = & 1, \\ e_1(x_1, \ldots, x_n) & = & x_1 + x_2 + \cdots + x_n, \\ e_2(x_1, \ldots, x_n) & = & \sum_{i<j} x_i x_j, \\ e_n(x_1, \ldots, x_n) & = & x_1 x_2 \cdots x_n, \\ e_k(x_1, \ldots, x_n) & = & 0, \quad \text{for } k > n. \end{array}$$

Por ejemplo:

$$x_1^2 + x_2^2 = (x_1 + x_2)^2 - 2(x_1\, x_2) = e_1(x_1, x_2)^2 - 2e_2(x_1, x_2)$$

- Busca a mano una forma de expresar el polinomio $x_1^2 + x_2^2 + x_3^2 + x_4^2$ en función de los polinomios simétricos elementales en cuatro variables.
- Busca a mano una forma de expresar el polinomio $x_1^3 + x_2^3$ en función de los (tres) polinomios simétricos elementales en dos variables.
- Busca a mano una forma de expresar el polinomio $-\frac{1}{6}\, x_1^3 - \frac{1}{2}\, x_1^2 x_2 - \frac{1}{2}\, x_1 x_2^2 - \frac{1}{6}\, x_2^3 + x_1 + x_2$ en función de los (tres) polinomios simétricos elementales en dos variables.
- Busca a mano una forma de expresar el polinomio de Taylor de grado k de $sin(x_1 + x_2)$ en función de los (tres) polinomios simétricos elementales en dos variables (indicación: busca un método para calcular el polinomio de Taylor)

6.2.4 4: Trabajar con un número arbitrario de variables simbólicas

El código siguiente:

```
N = 5
var(',',.join(['x%d'%j for j in range(1, N+1)]))
```

define las N variables simbólicas x1,...,xN.

Si además guardas las variables en una lista, puedes iterar la lista de variables para, por ejemplo, multiplicarlas todas:

```
N = 5
vs = var(','.join(['x%d' %j for j in range(1, N+1)]))
p = 1
for v in vs:
    p = p\*v
print p
```

- Escribe una función que, dados dos números *n* y *k* , construya el polinomio $p_k(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^k = x_1^k + \cdots + x_n^k$
- Escribe una función que, dado un número *k* , construya el polinomio simétrico en k variables: e_k:

$$\begin{aligned} e_0(x_1, \ldots, x_n) &= 1, \\ e_1(x_1, \ldots, x_n) &= x_1 + x_2 + \cdots + x_n, \\ e_2(x_1, \ldots, x_n) &= \sum_{i<j} x_i x_j, \\ e_n(x_1, \ldots, x_n) &= x_1 x_2 \cdots x_n, \\ e_k(x_1, \ldots, x_n) &= 0, \quad \text{for k>n} \end{aligned}$$

- Escribe código que compruebe las *identidades de Newton* , para k desde 2 hasta 5 (más información):

$$k e_k(x_1, \ldots, x_n) = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i}(x_1, \ldots, x_n) p_i(x_1, \ldots, x_n)$$

```
sage: N = 5
sage: var(','.join(['x%d' %j for j in range(1, N+1)]))
(x1, x2, x3, x4, x5)

sage: N = 5
sage: vs = var(','.join(['x%d' %j for j in range(1, N+1)]))
sage: p = 1
sage: for v in vs:
...       p = p*v
sage: print p
x1*x2*x3*x4*x5
```

6.2.5 5.

- Escribe código que componga la *matriz de van der Monde* n x n:

$$V = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \ldots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \alpha_n^2 & \ldots & \alpha_n^{n-1} \end{pmatrix}$$

- Demuestra que la matriz de van der Monde es invertible si los números $\alpha_1, \ldots, \alpha_n$ son distintos, para n=2,3 y 4. (Indicación: factoriza el determinante)

6.3 Cálculo diferencial en una variable

6.3.1 Límites

El método `limit` permite calcular límites de funciones. Para calcular el límite de `f` en el punto `x0` .

```
f.limit(x=x0)
```

Cuando la función está definida y es continua en el punto, el valor del límite es igual al límite de la función:

```
sage: var('x')
sage: f1=x^2
sage: f2=(x+1)*sin(x)
sage: f3=f2/f1
sage: f4=sqrt(f3)/(x^2+1)

sage: print f1.limit(x=1)
sage: print f1(x=1)
1
1
```

Se puede calcular el límite en un punto de la recta, o también en $\pm\infty$ (al que podemos referirnos por `\+Infinity` o por `oo`).

```
sage: f3.limit(x=0)
+Infinity

sage: f3.limit(x=+oo)
0
```

Si el límite no existe, la función `limit` devuelve el valor `und` , abreviatura de *undefined* .

```
sage: f=1/(x*sin(x))
sage: f.show()
sage: f.limit(x=+Infinity)
und
```

$$\frac{1}{x\sin\left(x\right)}$$

Usando el argumento adicional ' `dir` ', podemos especificar si se trata de un límite lateral, por arriba ('above') o por abajo ('below'):

```
sage: f=1/x
sage: print f.limit(x=0,dir='above')
sage: print f.limit(x=0,dir='below')
+Infinity
-Infinity
```

Límites de sucesiones

También podemos calcular límites de sucesiones de la misma forma. Al fin y al cabo, si la expresión simbólica admite valores reales, el límite de la función que define en infinito es lo mismo que el límite de la sucesión de naturales definido por la misma expresión.

```
sage: var('k')
sage: limit(k^2/(k+1)^2, k = oo)
1

sage: limit(2^k/factorial(k), k = oo)
0
```

Suma de una serie

El comando `sum` permite calcular la suma de una serie infinita, pero usando una sintaxis distinta de la usada hasta ahora:

```
sum(expresion, variable, limite_inferior, limite_superior)
```

```
sage: sum(1/k^2, k, 1, oo)
1/6*pi^2
```

Algunos límites, o sumas, no se pueden resolver sin hipótesis adicionales, y es necesario usar `assume` .

```
sage: var('k a')
sage: sum(k^a, a, 1, oo)
Is  abs(k)-1  positive, negative, or zero?
Traceback (most recent call last):
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'assume
```

```
sage: var('k a')
sage: forget()
sage: assume(abs(k)<1)
sage: sum(k^a, a, 0, oo)
-1/(k - 1)
```

Observamos que informar a Sage de que -1<k<1 en vez de |k|<1, aunque en principio es equivalente (si entendemos que al escribir desigualdades ya no hablamos de números complejos sino reales), no funciona.

Sirva este ejemplo para recordar que los motores de cálculo simbólico hacen lo que pueden, y no siempre dan el resultado esperado. A veces necesitan que le introduzcamos la información de una forma particular, y fallan si la introducimos de una forma equivalente. Es por ello que normalmente usamos las herramientas de cálculo simbólico de forma interactiva.

```
sage: var('k a')
sage: forget()
sage: assume(k,'real')
sage: assume(-1<k<1)
sage: sum(k^a, a, 0, oo)
Is  abs(k)-1  positive, negative, or zero?
Traceback (most recent call last):
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'assume
```

6.3.2 Derivadas

El método `derivative` permite calcular derivadas de funciones simbólicas. Las **derivadas** se obtienen siguiendo metódicamente las reglas de derivación y no suponen ningún problema al ordenador:

```
sage: sin(x).derivative(x)
cos(x)
```

```
sage: f2.derivative(x)
(x + 1)*cos(x) + sin(x)
```

Si usamos funciones de una sóla variable, podemos omitir la variable por la que estamos derivando.

```
sage: f2.derivative()
(x + 1)*cos(x) + sin(x)
```

6.3.3 Integrales

Las **integrales** son más complicadas de tratar mediante cálculo simbólico que las derivadas, ya que no existe ningún método que pueda calcular la integral de cualquier función. En realidad, hay muchas funciones elementales (construídas a partir funciones trigonométricas, exponenciales y algebraicas mediante sumas, productos y composición de funciones) cuyas integrales, aunque estén bien definidas, no se pueden expresar en términos de estas mismas funciones. Los ejemplos $f(x) = e^{-x^2}$ y $f(x) = \frac{sin(x)}{x}$ son bien conocidos.

Aunque en teoría existe un algoritmo (el algoritmo de Risch) capaz de decidir si la integral de una función elemental es otra función elemental, dificultades prácticas imposibilitan llevarlo a la práctica, y el resultado es que incluso en casos en los que integramos una función cuya integral es una función elemental, nuestro algoritmo de integración simbólica puede no darse cuenta.

```
sage: sin(x).integral()
__main__:1: DeprecationWarning: Variable of integration should be specified explicitly.
-cos(x)

sage: f2=sin(x)
sage: f2.integral(x)
-cos(x)

sage: f5=1/sqrt(1-x^2)
sage: f5.show()
```

$$\frac{1}{\sqrt{-x^2+1}}$$

```
sage: f5.integral(x)
arcsin(x)
```

Las *funciones racionales* (cociente de un polinomio por otro polinomio) se pueden integrar de forma exacta, siempre que sea posible descomponer los denominadores en fracciones simples (el algoritmo para integrar funciones racionales es el mismo que estudiásteis en el Bachillerato).

```
sage: f6=(x^3+4*x^2-x-2)/(x^2+8*x+1)

sage: f6i = f6.integral(x)
sage: f6i.show()
```

$$\frac{1}{2}\,x^2 - \frac{59}{15}\,\sqrt{15}\log\left(\frac{\left(x-\sqrt{15}+4\right)}{\left(x+\sqrt{15}+4\right)}\right) - 4\,x + 15\,\log\left(x^2+8\,x+1\right)$$

Sin embargo, algunas integrales no se pueden expresar en términos de funciones elementales y sólo se pueden dejar indicadas.

```
sage: f7=sin(x)/x
sage: f7i=f7.integral(x)

sage: print f7i
sage: f7i.show()
integrate(sin(x)/x, x)
```

$$\int \frac{\sin(x)}{x}\, dx$$

Algunas funciones tienen integrales que se pueden expresar en términos de funciones elementales, pero el ordenador no es capaz de encontrarlas y desiste.

```
sage: f=(x^10*sin(x^3)*cos(x)).derivative()
sage: print f
3*x^12*cos(x^3)*cos(x) - x^10*sin(x^3)*sin(x) + 10*x^9*sin(x^3)*cos(x)

sage: #Observa que el resultado contiene integrales; esto quiere decir que no ha completado la integ
sage: #de la funcion f y lo ha dejado indicado
sage: g=f.integrate()
sage: print g
__main__:4: DeprecationWarning: Variable of integration should be specified explicitly.
1/4*x^10*sin(x^3 - x) + 1/4*x^10*sin(x^3 + x) + 1/4*x^10*cos(x^3 - x) - 1/4*x^10*cos(x^3 + x) + 1/4*

sage: sage.calculus.calculus.integral?
<html>...</html>
```

6.3.4 Integrales definidas

Las **integrales definidas** son también complicadas de calcular de forma exacta. Sin embargo, una integral definida es un número real, y en las aplicaciones es suficiente con aproximar ese número con suficiente precisión.

Si insistimos en hacer el cálculo exacto, podemos usar el mismo método `integral` , pero pasando además dos argumentos: el extremo inferior y superior del intervalo.

```
sage: sin(x).integral(0,pi)
2

sage: f(x) = sin(x)
sage: F = f.integral()
sage: v = F(pi) - F(0)
sage: print v
sage: print f.integral(0,pi)
2
x |--> 2

sage: #Algunas integrales definidas no se pueden calcular de forma exacta,
sage: #y las deja indicadas
sage: f7=sin(x)/x
sage: integral(f7,0,1)
/home/sageadm/sage-4.4.1-linux-64bit-ubuntu_9.10-x86_64-Linux/local/lib/python2.6/site-packages/sage/
  return x.integral(*args, **kwds)
integrate(sin(x)/x, x, 0, 1)
```

Sin embargo, todas las integrales definidas se pueden calcular de forma aproximada con la función `numerical_integral` , que acepta como argumentos la función a integrar y los extremos del intervalo, pero devuelve una tupla formada por el valor aproximado de la integral y una cota del error cometido.

```
sage: numerical_integral(f7,0,1)
(0.94608307036718298, 1.0503632079297087e-14)

sage: numerical_integral?
<html>...</html>
```

Las funciones de una variable simbólica también se pueden integrar con el método `nintegral`, que además del valor aproximado de la integral y una estimación del error devuelve dos parámetros adicionales.

```
sage: f7.nintegral(x,0,1)
(0.94608307036718309, 1.050363207929709e-14, 21, 0)
```

Ejercicio

Lee la ayuda del método `nintegral` , y realiza llamadas a nintegral tales que:

- el último valor devuelto por nintegral sea positivo
- el tercer valor devuelto por nintegral sea mayor que 100

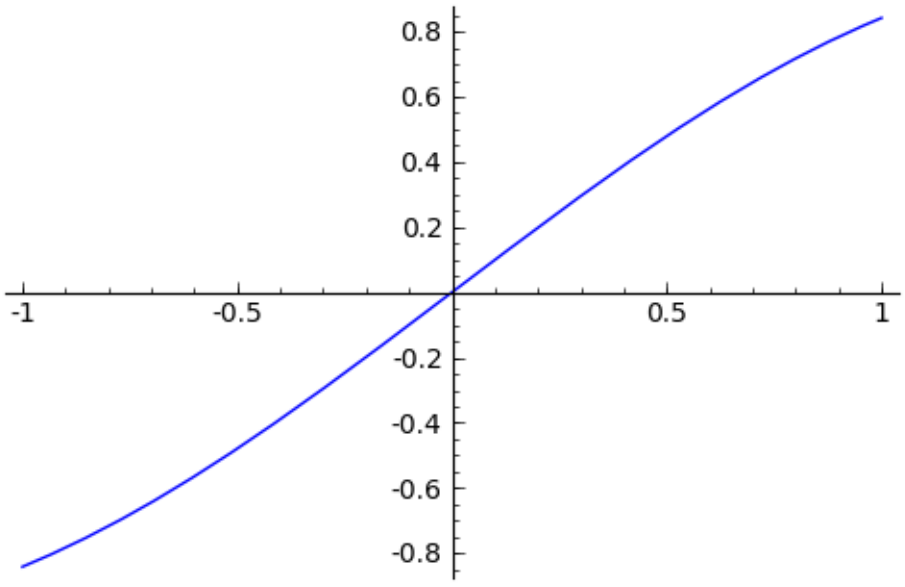
```
sage: f7.nintegral?
<html>...</html>
```

6.3.5 Gráficas

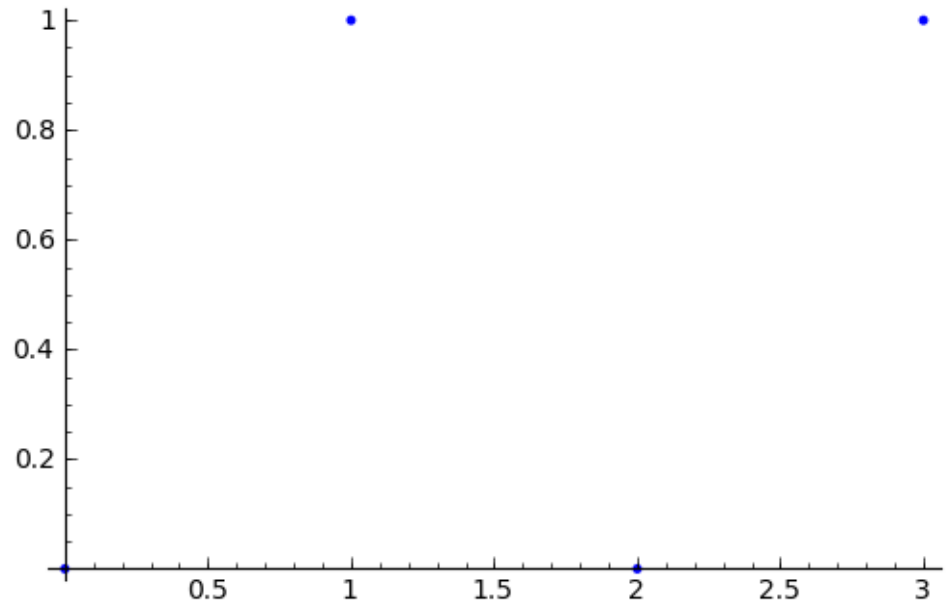
Vamos a manejar varios comandos para hacer gráficas:

- `plot(f)` (ó `f.plot()`) dibujan la función f.
- `point2d(lista)` , dibuja los puntos de la lista que se pasa como argumento.
- `line2d(lista)` , dibuja líneas entre los puntos de la lista que se pasa como argumento.

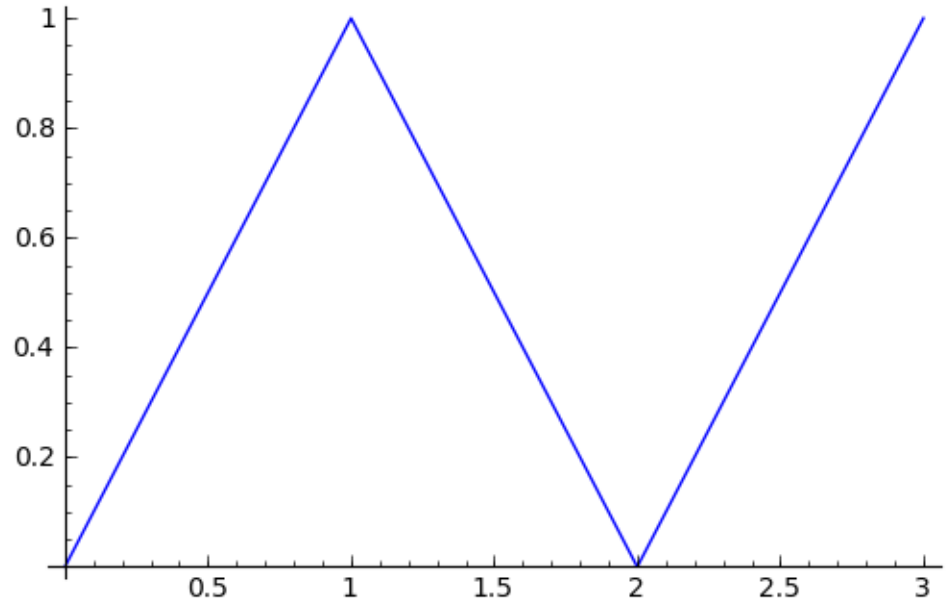
```
sage: plot(f2)
```




```
sage: puntos = [(0,0),(1,1),(2,0),(3,1)]
sage: point2d(puntos)
```



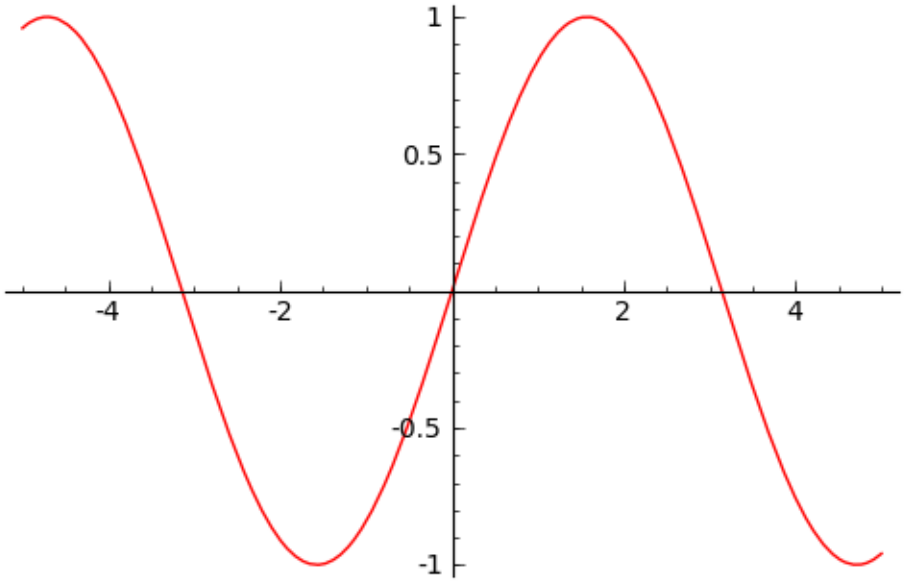

```
sage: line2d(puntos)
```



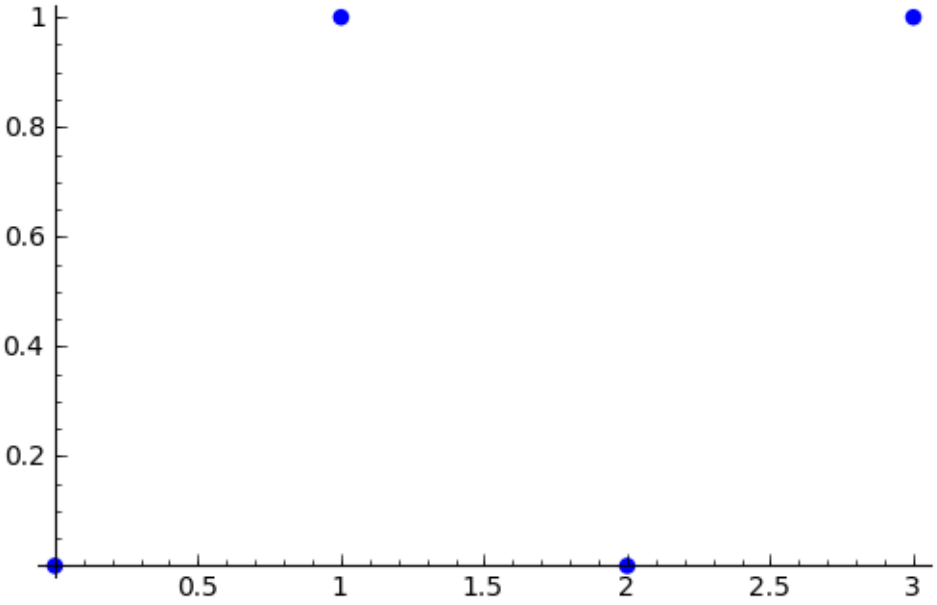
Además de los argumentos obligatorios, le podemos pasar argumentos opcionales. Los siguientes parámetros son bastante útiles:

- `plot(f,xmin,xmax)` muestra la función entre los puntos `xmin` y `xmax` .
- `point2d(puntos, pointsize=20)` permite especificar el tamaño de los puntos
- `line2d(puntos, thickness= 2)` permite especificar el grueso de la línea
- `plot(f, color=(1,0,0))` permite especificar el color como una tupla de 3 valores: los tonos de rojo, verde y azul.
- `point2d(puntos, rgbcolor=(1,0,0) )` y `line2d(puntos, rgbcolor=(1,0,0) )` permiten especificar el color como una tupla de 3 valores: los tonos de rojo, verde y azul.

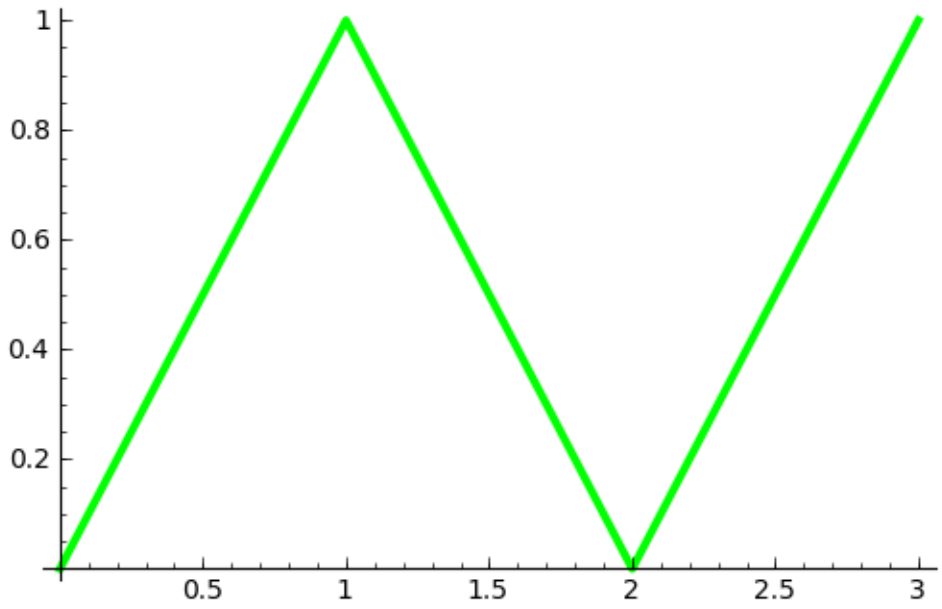
```
sage: plot(f2,-5,5,color=(1,0,0))
```




```
sage: point2d(puntos, pointsize=30)
```




```
sage: line2d(puntos, thickness= 3, color=(0,1,0))
```



Las gráficas se pueden almacenar en variables.

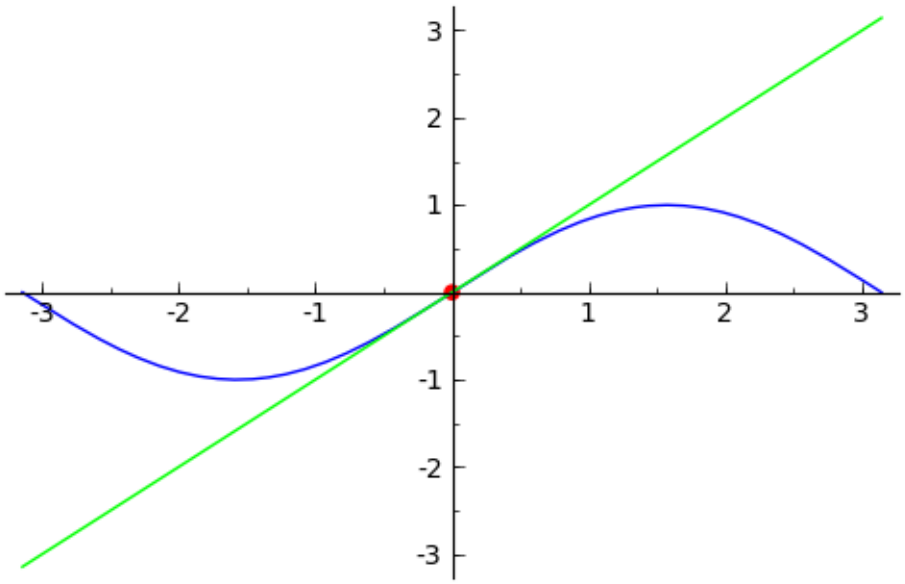
```
sage: grafica1 = plot(sin(x),-pi,pi)
sage: grafica2 = point((0,0), pointsize=30, rgbcolor=(1,0,0) )
sage: grafica3 = line([(-pi,-pi),(pi,pi) ], rgbcolor=(0,1,0) )
```

Las gráficas tienen un tipo de datos especial, con sus propios métodos.

```
sage: grafica1?
<html>...</html>

sage: grafica1.
Traceback (most recent call last):
...
SyntaxError: invalid syntax
```

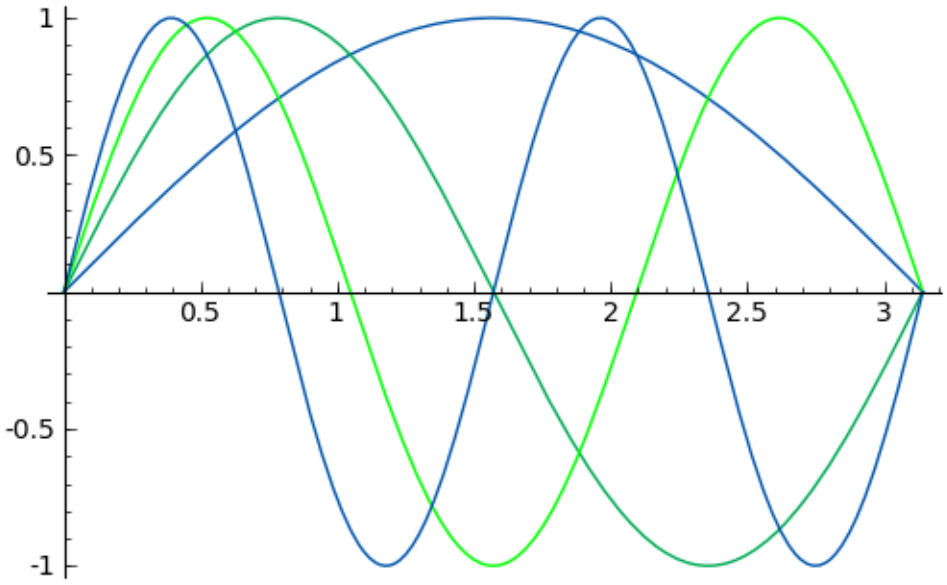
El operador suma (+) permite combinar gráficas en una sóla imagen.

```
sage: grafica = grafica1+grafica2+grafica3
sage: grafica.show()
```



Podemos combinar muchas gráficas en una sóla almacenando las gráficas en una lista, y luego usando la función `sum` .

```
sage: lista = [plot(sin(n*x), (0, pi), color=(0,n/3,1-n/3))  for n in [1..4]]
sage: grafica = sum(lista)
sage: grafica.show()
```



6.3.6 Ceros, máximos y mínimos

Buscar ceros de funciones

Algunos problemas típicos del cálculo no se prestan tan bien al cálculo simbólico como los límites y las derivadas. Un ejemplo típico es encontrar los puntos donde una función toma el valor 0. Podemos intentar resolver la ecuación f=0 usando el comando `solve` , al que pasamos como argumento la variable simbólica para la que queremos resolver.

```
f = x^2 - 3
f.solve(x)

> [x == -sqrt(3), x == sqrt(3)]
```

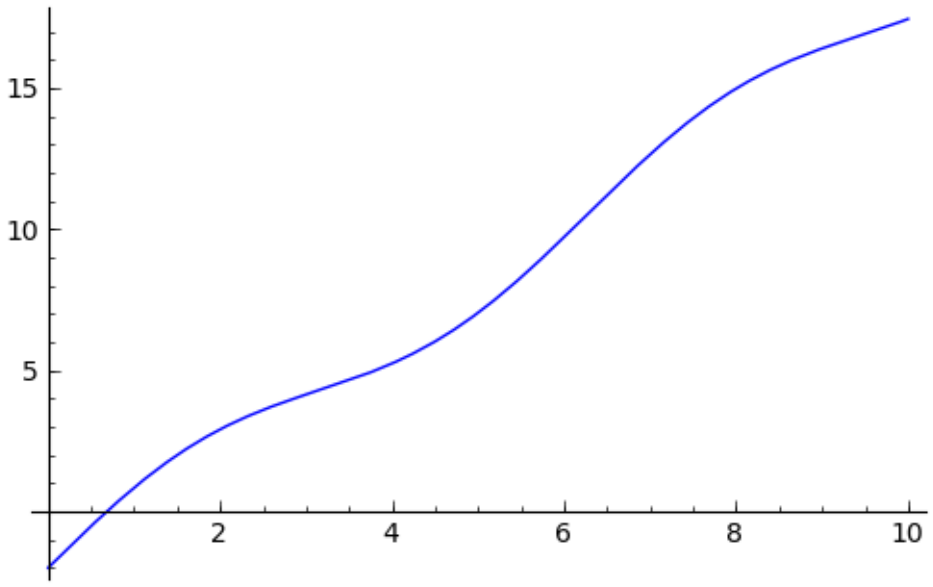
Aunque el resultado es adecuado en muchos casos particulares importantes, francamente pocas ecuaciones que podamos definir combinando más de una función elemental tienen raíces que se puedan escribir de forma exacta como una expresión simbólica. Además, tenemos el problema de la multiplicidad de raíces y extremos de las funciones.

La solución que nos ofrece SAGE es buscar las raíces de forma numérica en un intervalo dado. Una llamada al método `find_root` :

```
f.find_root(a,b)
```

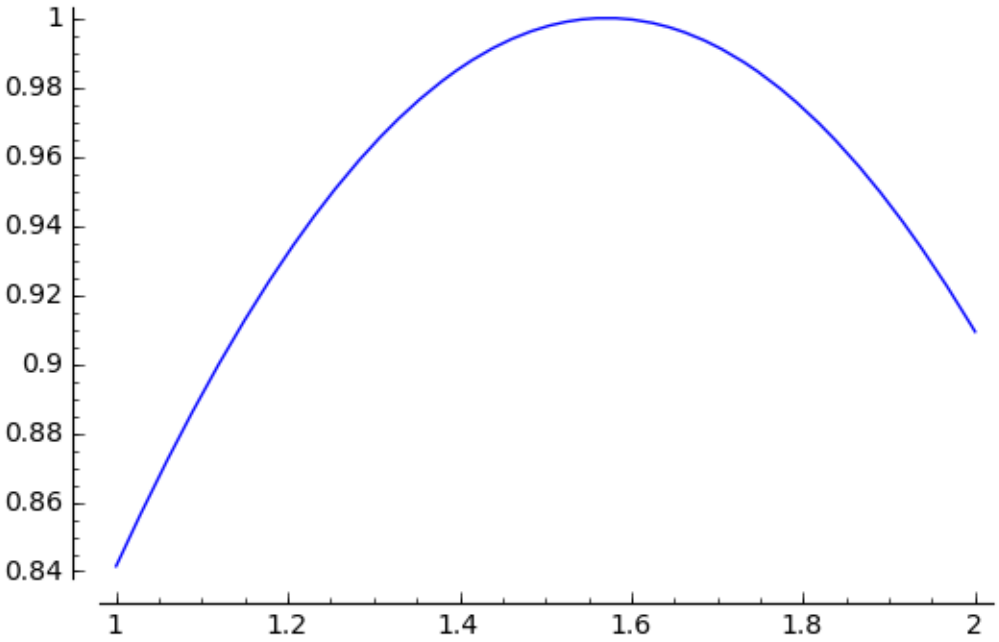
nos devuelve una raíz de la función `f` en el intervalo `[a,b]` , y lanza un error si la función no tiene ceros en ese intervalo.

```
sage: f6 = x^2 - 3

sage: f6.solve(x)
[x == -sqrt(3), x == sqrt(3)]

sage: f7=2*x+sin(x)-2

sage: #El resultado de llamar a solve no es satisfactorio
sage: f7.solve(x)
[x == -1/2*sin(x) + 1]

sage: f7.plot(0,10)
```




```
sage: f7.find_root(0,2)
0.68403665667799907

sage: sin(x).find_root(1,2)
Traceback (most recent call last):
...
RuntimeError: f appears to have no zero on the interval
```

```
sage: plot(sin(x),1,2)
```



Si la función tiene más de una raíz en el intervalo, `find_root` devuelve una cualquiera de las raices.

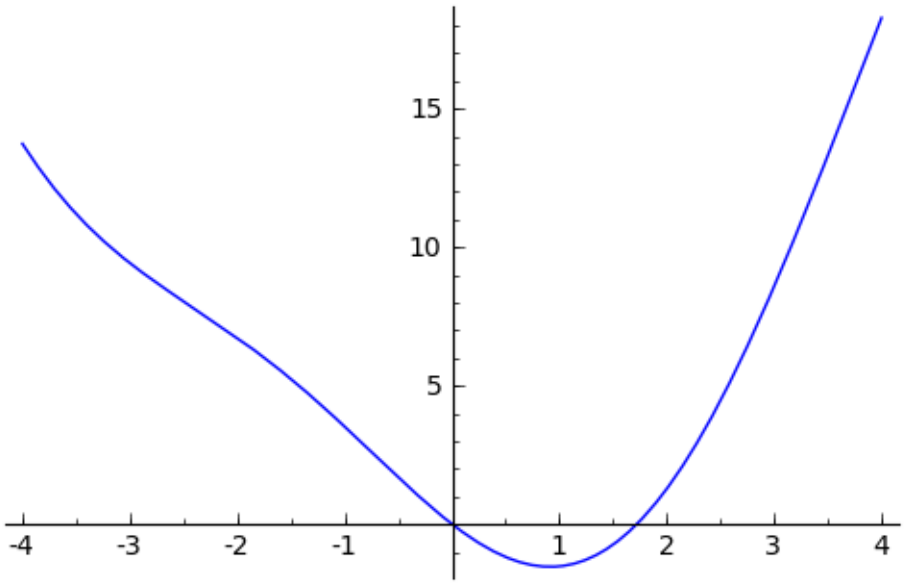
```
sage: sin(x).find_root(1,30)
21.991148575128552
```

Las funciones `find_minimum_on_interval` y `find_maximum_on_interval` funcionan de forma similar a `find_root` :

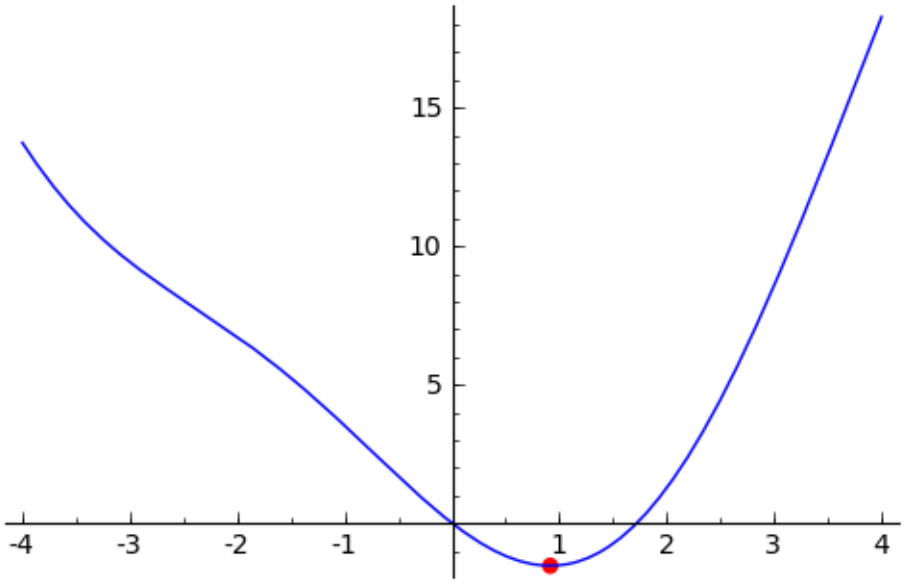
```
f.find_minimum_on_interval(a,b)
```

devuelve una tupla con dos valores: el minimo de la función en el intervalo y el punto donde se alcanza.

```
sage: f8 = x^2 - 3*sin(x)
sage: plot(f8,-4,4)
```




```
sage: (y0,x0) = f8.find_minimum_on_interval(-4,4)
sage: print x0,y0
sage: grafica = plot(f8,-4,4) + point2d((x0,y0), pointsize=30, rgbcolor=(1,0,0))
sage: grafica.show()
0.914856478862 -1.54046280571
```

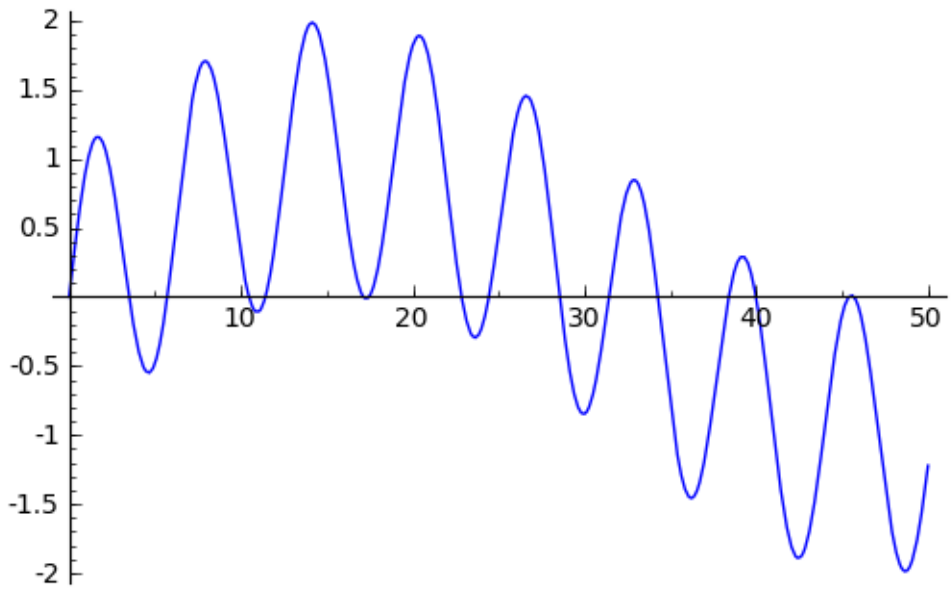


En realidad, el valor devuelto por `find_minimum_on_interval` es uno cualquiera de los mínimos locales de la función. En la documentación de la función leemos que usa el método de Brent : "*una combinación del método de bisección, el método de la secante y el método de interpolación cuadrática*":

Todos estos métodos pueden detenerse en un extremo local, y el algoritmo de Brent también. En general, el problema de encontrar máximos globales no es trivial, aunque para funciones de una variable de las que se conoce alguna propiedad extra no es difícil encontrar procedimientos para encontrar los extremos.

```
sage: f8.find_minimum_on_interval?
<html>...</html>
```

```
sage: f9=sin(x/10)+sin(x)
sage: plot(f9,0,50)
```




```
sage: f9.find_maximum_on_interval(10,20)
(1.987809504903955, 14.152658043591082)

sage: f9.find_maximum_on_interval(0,50)
(1.8920281240458212, 20.37533953333153)
```

6.3.7 Funciones definidas a trozos

Usando la función `Piecewise` podemos construir funciones definidas a trozos. Para ello, pasamos a la función Piecewise una lista de tuplas, donde cada tupla contiene un intervalo y una función. El resultado es la función definida a trozos que vale en cada intervalo la función que hemos pasado.

```
f = Piecewise( [(I1, f1), (I2,f2), ...] )
```

Pasamos los intervalos como tuplas con los extremos del intervalo.

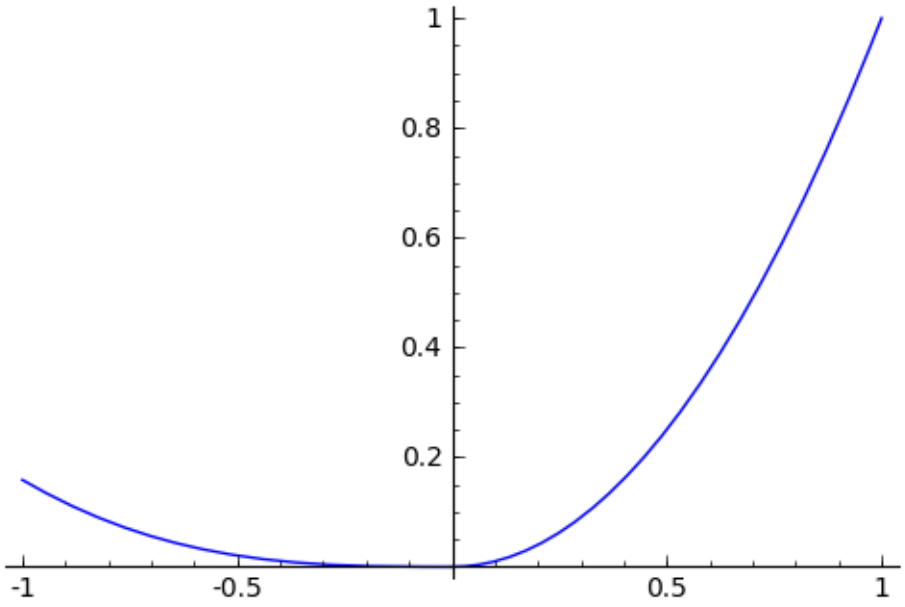
::

sage: f1=sin(x)-x sage: f2=x^2 sage: f=Piecewise([((-1,0),f1),((0,1),f2)])

```
sage: show(f)
```

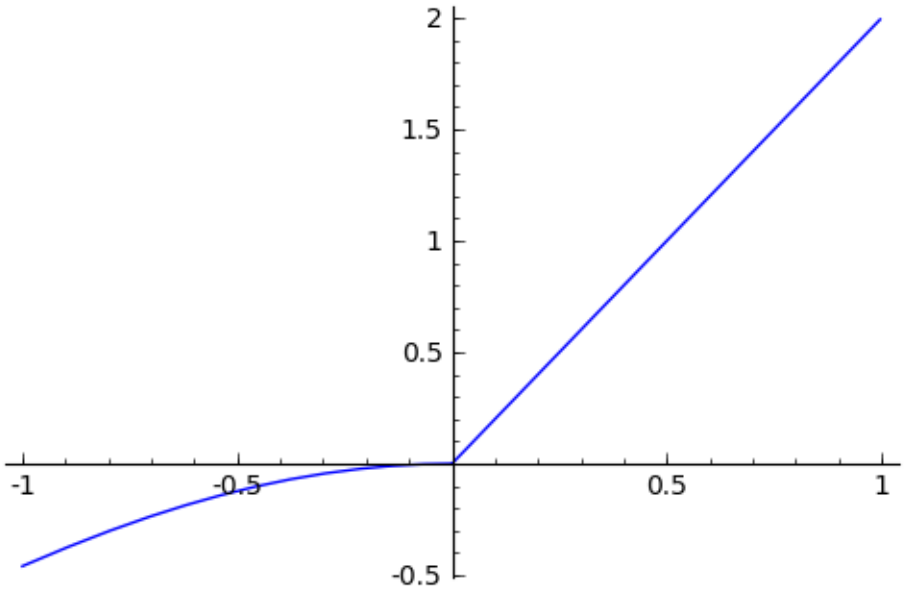
$$\begin{cases} -x + \sin(x) & \text{on } (-1, 0) \\ x^2 & \text{on } (0, 1) \end{cases}$$

```
sage: plot(f)
```




```
sage: fp = f.derivative()
sage: show(fp)
sage: plot(fp)
__main__:2: DeprecationWarning: Substitution using function-call syntax and unnamed arguments is depr
```

$$\begin{cases} x \mapsto \cos(x) - 1 & \text{on } (-1, 0) \\ x \mapsto 2\,x & \text{on } (0, 1) \end{cases}$$



6.4 Ejercicios

6.4.1 1.

Estudia la función:

$$\frac{(1-x)x\sin(10\,\pi x)}{(10\,x-6)}$$

en el intervalo [0,1]:

- La función no está definida cuando x=0.6. Indica cómo puedes extender la función para que quede definida en ese punto, siendo continua si es posible.
- Encuentra **todos** sus máximos y mínimos *locales y globales* en ese intervalo.
- Dibuja la función en el intervalo [0,1], con los extremos locales indicados en un color y los extremos absolutos indicados en un color distinto.

6.4.2 2.

Estudia la función g:

$$\frac{(x-1)x\cos\left(\frac{1}{6}\,\pi+4\,\pi x\right)}{(3\,x-1)}$$

en el intervalo [0,1]

- Calcula la segunda derivada de g.
- Dibuja la segunda derivada de g en el intervalo [0,1], y encuentra **todos** los ceros de la segunda derivada.
- Dibuja la función en el intervalo [0,1], con los puntos encontrados antes marcados en rojo. Es decir, deben aparecer, dibujados sobre la gráfica de f, los puntos donde se anula la segunda derivada.

6.4.3 3.

Define en un cuadro de comandos una función `f` en la variable simbólica `x` y un valor `x0` . A continuación, escribe código para:

- Calcular la función de la variable simbólica `x` que representa la recta tangente a `f` en el punto `x0` , a partir de los datos anteriores.
- Dibujar la función cerca del punto, el punto *(x0,f(x0))* en otro color, y la recta tangente a la función en *(x0,f(x0))* en otro color.

```
sage: f(x) = sin(x)
sage: x0   = 0
```

6.4.4 4.

Escoge una función `f(x)` cuyo límite cuando $x \to \infty$ exista, y crea una gráfica que muestre la función y la asíntota, en un rango aceptable de valores de x .

6.4.5 5.

Escoge una función con una asíntota oblicua, encuentra una función simbólica que represente esa asíntota y dibuja en la misma gráfica la función y la asíntota.

6.4.6 6.

Escoge una función convexa en un intervalo [a,b] y escribe código que, dado un número K, genere una gráfica que muestre la función en el intervalo, y K rectas tangentes a la función en puntos *(x,f(x))* para puntos *x* equiespaciados en el intervalo [a,b] .

Repite el ejercicio para una función con un punto de inflexión.

6.5 Cálculo vectorial

En esta hoja vamos a estudiar problemas de cálculo diferencial e integral en varias dimensiones espaciales. En general, manejaremos funciones de varias variables simbólicas.

Comencemos por representar gráficamente funciones de dos variables reales.

```
sage: var('x,y')
(x, y)

sage: f1 = sin((x+y)^2)
sage: f1.show()
```

$$\sin\left((x+y)^2\right)$$

6.5.1 Representación de funciones

El comando `plot3d` genera una superficie cuya altura sobre el punto (x,y) es proporcional al valor de una función f de dos variables simbólicas. La sintaxis es:

```
plot3d(f,(x,x1,x2),(y,y1,y2))
```

- x e y son variables simbólicas
- f es función de x y de y
- (x1,x2) es el rango que se mostrará de la variable x
- (y1,y2) es el rango que se mostrará de la variable y

Al mostrar la gráfica llamando a `show()` , podemos pasar el argumento opcional viewer que nos permite elegir entre varias formas de mostrar la gráfica, algunas interactivas, que permiten girar la gráfica con el ratón. Como de costumbre, `plot3d?` y `p.show?` (p es una gráfica generada con `plot3d`) muestran la ayuda con ejemplos de uso.

```
sage: p = plot3d(f1,(x,-1,1),(y,-1,1))
sage: p.show(viewer='tachyon')
```

```
sage: f2 = x*y*exp(x-y)
sage: f2.show()
sage: p = plot3d(f2,(x,-2,0.5),(y,-0.5,2))
sage: p.show(viewer='tachyon')
```

$$xye^{(x-y)}$$

Curvas de nivel

Otra opción es dibujar curvas de nivel de la función en el plano (x,y). Usando

```
implicit_plot(f,(x,x1,x2),(y,y1,y2))
```

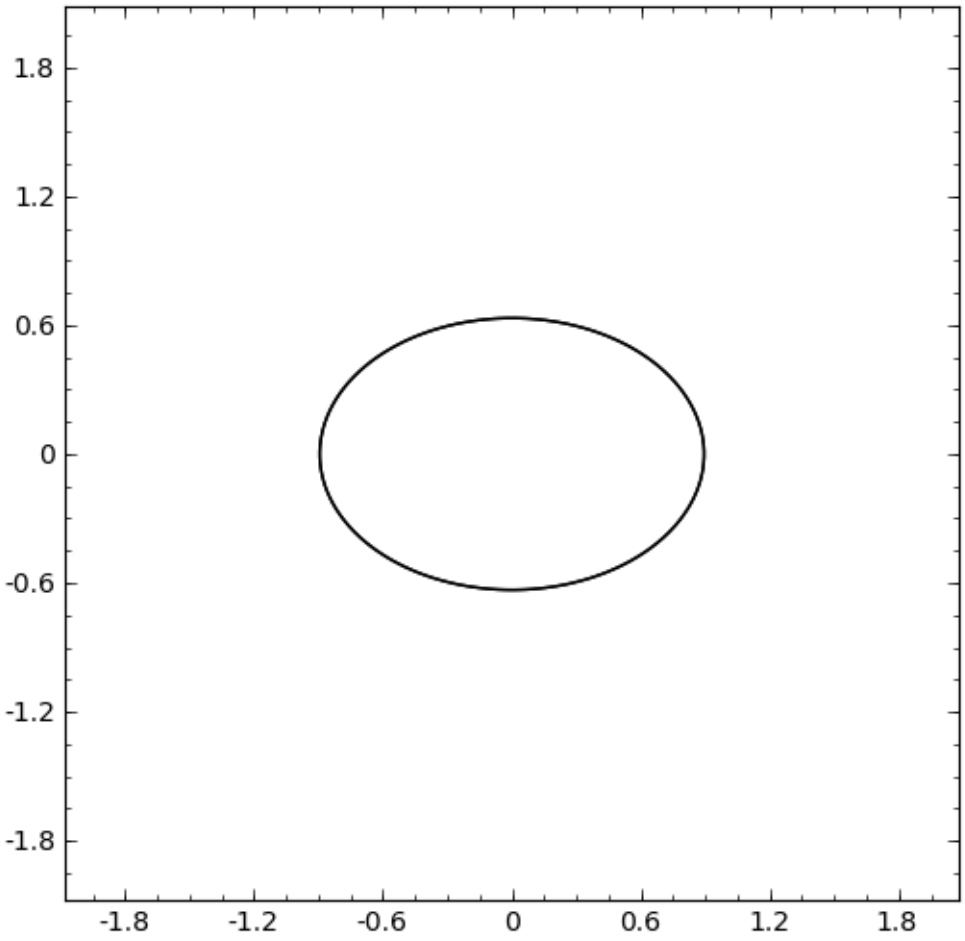
Dibujamos la curva dada por f=0. Si pasamos a `implicit_plot` el argumento adicional `contours`, el dibujo contiene tantas curvas de nivel como le indiquemos:

```
implicit_plot(f,(x,x1,x2),(y,y1,y2), contours= 10)
```

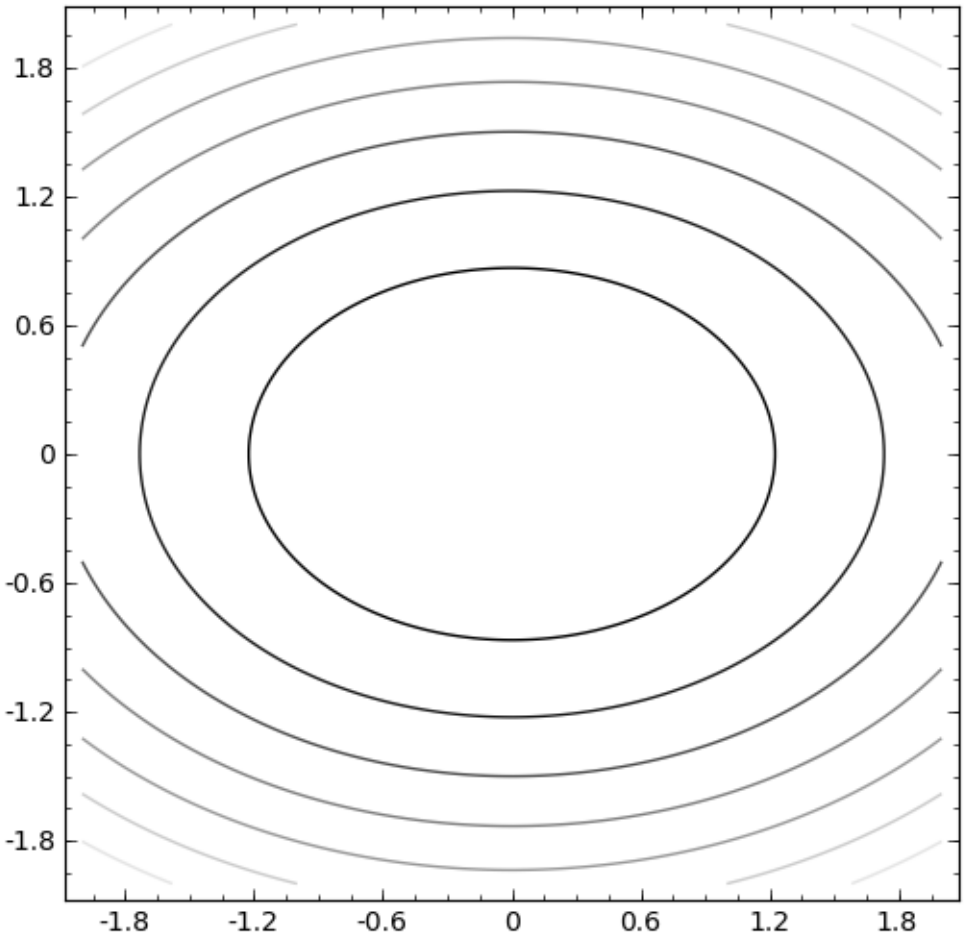
ó exactamente las curvas de nivel que le indiquemos:

```
implicit_plot(f,(x,x1,x2),(y,y1,y2), contours = lista)
```

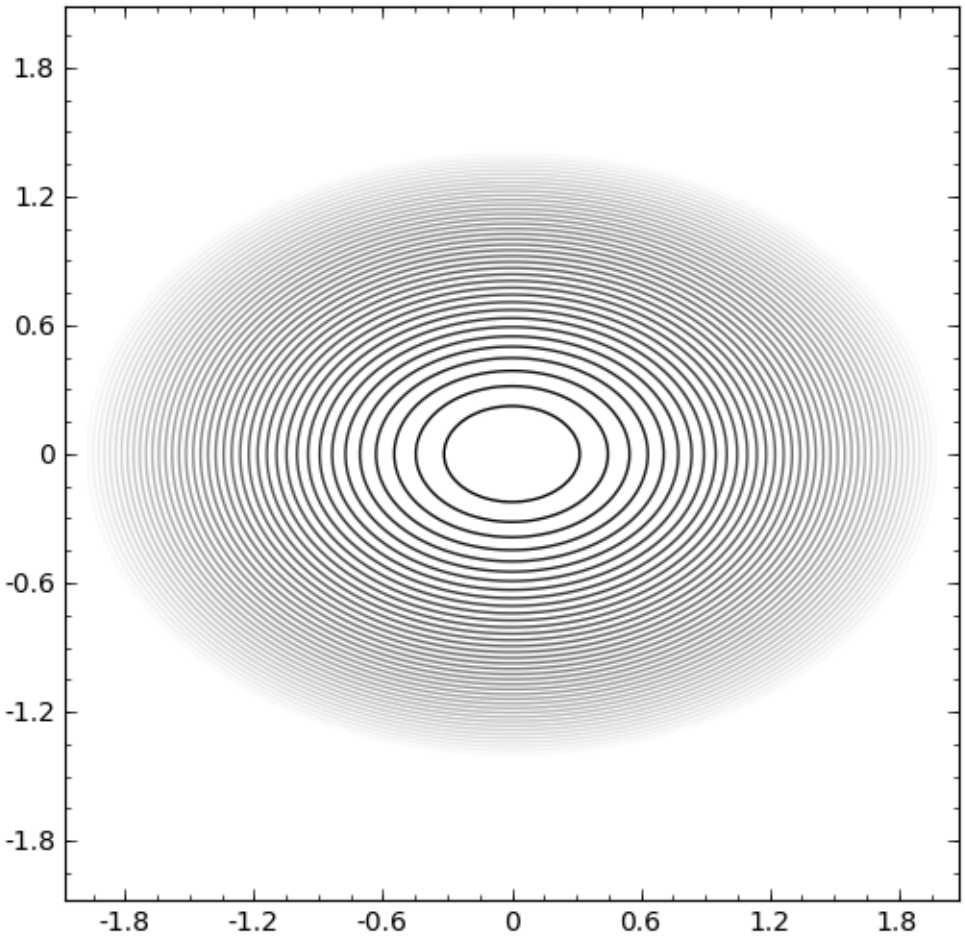
```
sage: #Pasando a show el parametro adicional aspect_ratio=1
sage: #garantizamos que se mantendrá el ratio correcto entre
sage: #el eje x y el eje y
sage: implicit_plot(x^2 + 2*y^2 - 0.8,(x,-2,2),(y,-2,2)).show(aspect_ratio=1)
```




```
sage: implicit_plot(x^2 + 2*y^2,(x,-2,2),(y,-2,2),contours=10).show(aspect_ratio=1)
```

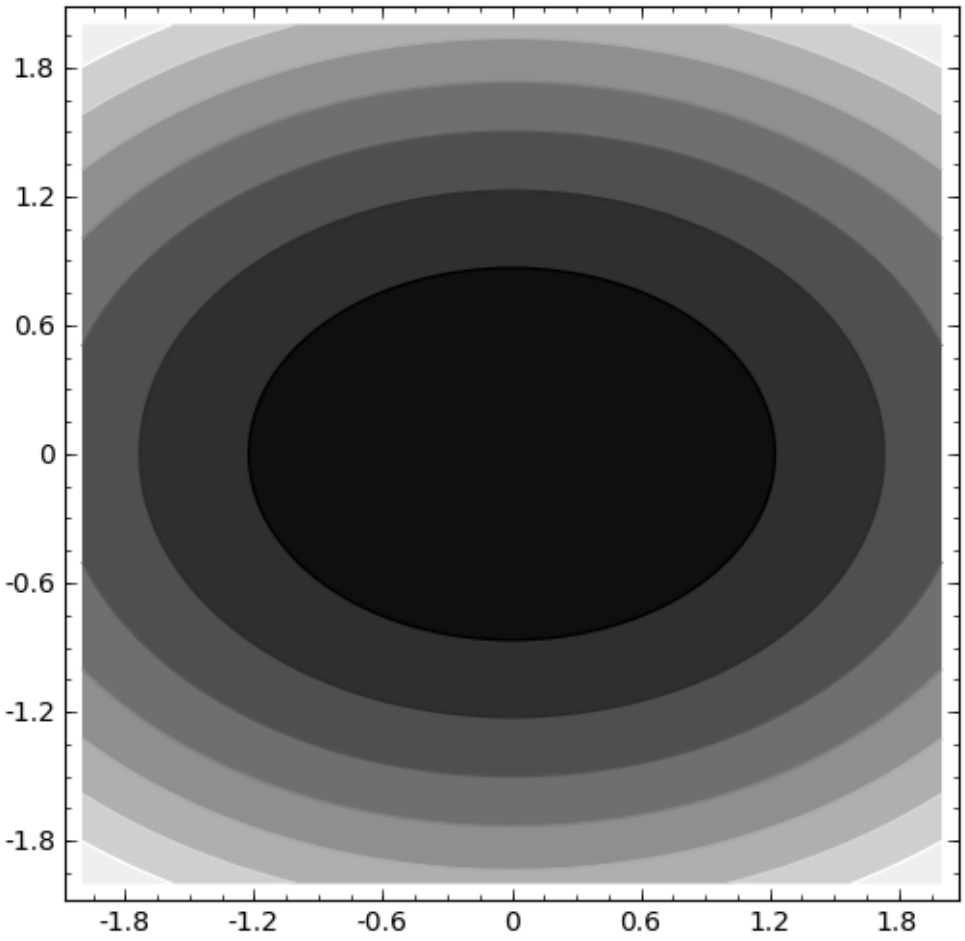



```
sage: implicit_plot(x^2 + 2*y^2,(x,-2,2),(y,-2,2),contours=srange(0,4,0.1)).show(aspect_ratio=1)
```



`contour_plot` es muy similar al anterior, pues muestra de distinto color la región entre dos curvas de nivel consecutivas (es decir, la preimagen de un intervalo).

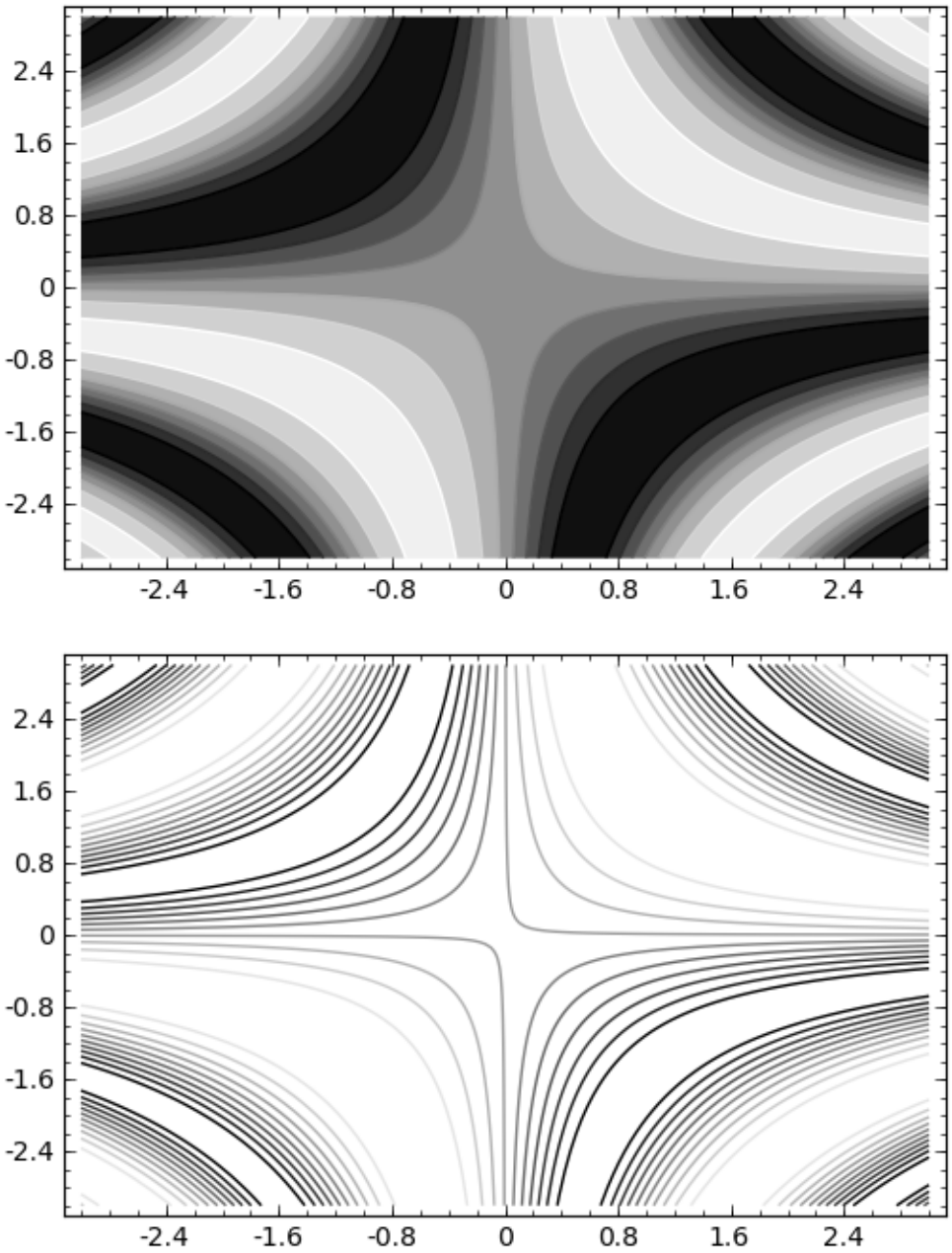
```
sage: contour_plot(x^2+2*y^2,(x,-2,2),(y,-2,2)).show(aspect_ratio=1)
```



Comparamos las distintas formas de representar una función de dos variables.

```
sage: f3=log(2+sin(x*y))
sage: f3.show()
sage: plot3d(f3,(x,-3,3),(y,-3,3)).show(viewer='tachyon')
sage: contour_plot(f3,(x,-3,3),(y,-3,3)).show()
sage: implicit_plot(f3,(x,-3,3),(y,-3,3),contours=10).show()
```

$$\log\left(\sin\left(xy\right)+2\right)$$



Plano tangente

Una función de dos variables simbólicas se puede derivar respecto de cada una de ellas de forma independiente. De esta forma podemos aplicar la fórmula del plano tangente en un punto:

$$z = f(x_0, y_0) + \frac{\partial f}{\partial x}(x_0, y_0)(x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0)(y - y_0)$$

Las gráficas de curvas de nivel no permiten apreciar bien el plano tangente, pero podemos representar el plano junto a la función con `plot3d` .

```
sage: f4=x*log(x^2+y^2)
sage: f4.show()
sage: x0 = 1
sage: y0 = 1
sage: plano = (x - x0)*f4.derivative(x,1).subs(x=x0, y=y0) + (y - y0)*f4.derivative(y,1).subs(x=x0, y
sage: show(plano)
sage: p = (plot3d(f4,(x,-1.5,1.5),(y,-1.5,1.5)) +
...        plot3d(plano,(x,-1.5,1.5),(y,-1.5,1.5),color='red') +
...        point3d( (x0,y0,f4(x=x0,y=y0)), color='green', pointsize='30') )
sage: p.show(viewer='tachyon')
```

$$x \log\left(x^2 + y^2\right)$$

$$(\log(2) + 1)(x - 1) + y + \log(2) - 1$$

Polinomio de Taylor

La fórmula del plano tangente es el caso particular de la fórmula del polinomio de Taylor de la función en un punto:

$$f(x) = \sum_{|\alpha|=0}^{k} \frac{1}{\alpha!} \frac{\partial^\alpha f(a)}{\partial x^\alpha} (x-a)^\alpha + \sum_{|\alpha|=k+1} R_\alpha(x)(x-a)^\alpha$$

Donde $x = (x_1, \ldots, x_n)$ es una variable vectorial, y α es un multiíndice. En palabras, el polinomio de Taylor de f de orden k en $a = (a_1, \ldots, a_n)$ contiene todos los monomios construídos con derivadas de orden menor o igual que k. La sintaxis para construir el polinomio de Taylor de f de orden k en las variables (x,y) y en el punto (a,b) es:

```
f.taylor((x,a),(y,b),k)
```

```
sage: #El polinomio de Taylor de orden 1 es la ecuacion del
sage: #plano tangente
sage: x0 = 1
sage: y0 = 1
sage: plano = (x - x0)*f4.derivative(x,1).subs(x=x0,y=y0) + (y - y0)*f4.derivative(y,1).subs(x=x0,y=y
sage: show(plano)
sage: f4t1 = f4.taylor((x,x0),(y,y0),1)
sage: show(f4t1)
```

$$(\log(2) + 1)(x - 1) + y + \log(2) - 1$$

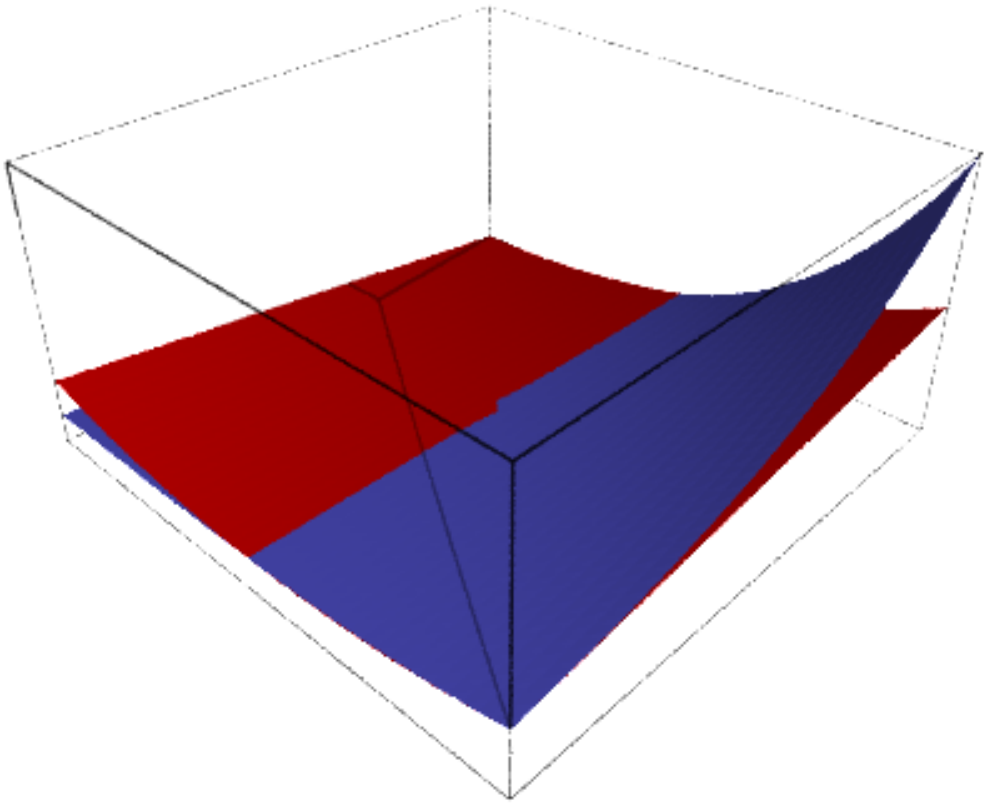
$$(\log(2) + 1)(x - 1) + y + \log(2) - 1$$

```
sage: f5 = x*exp(x + y)
sage: f5.show()
sage: #pol5 = f5.taylor((x,0),(y,0),2)
sage: pol5 = polinomio_taylor(f5,(x,0),(y,0),2)
sage: pol5.show()
```

$$xe^{(x+y)}$$

$$x^2 + xy + x$$

```
sage: p1=plot3d(f5,(x,-1,1),(y,-1,1));
sage: p2=plot3d(pol5,(x,-1,1),(y,-1,1),color='red');
sage: show(p1+p2,viewer='tachyon')
```



Derivadas direccionales

Ejercicio

- Calcula la derivada direccional de la función $f_6 = \sin(xy) + \cos(xy)$ en el punto $(1, \pi/2)$ y la dirección $(1, 1)$.
- Investiga la ayuda del comando `arrow`, y dibuja el gradiente de f6 en un punto. Observa el efecto al cambiar la función, y el punto.

```
sage: f6=sin(x*y)+cos(x*y)
sage: f6.show()
```

$$\sin(xy) + \cos(xy)$$

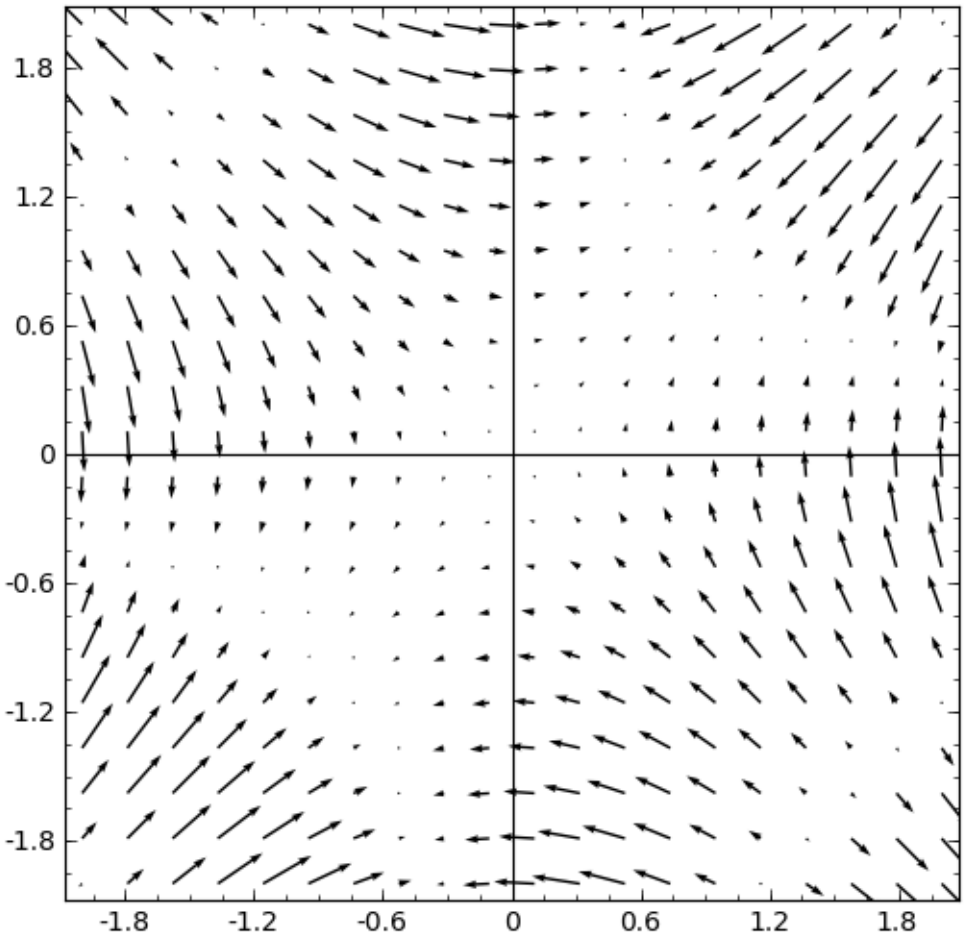
Campo gradiente

Usando el comando `plot_vector_field` , podemos dibujar el **campo gradiente** . El campo de vectores *gradiente de* f es la asignación del vector gradiente de f a cada punto del plano. Recordemos de las clases de teoría que el gradiente es perpendicular a las curvas de nivel. Es una buena ocasión para dibujar simultáneamente las curvas de nivel y el campo gradiente de una función.

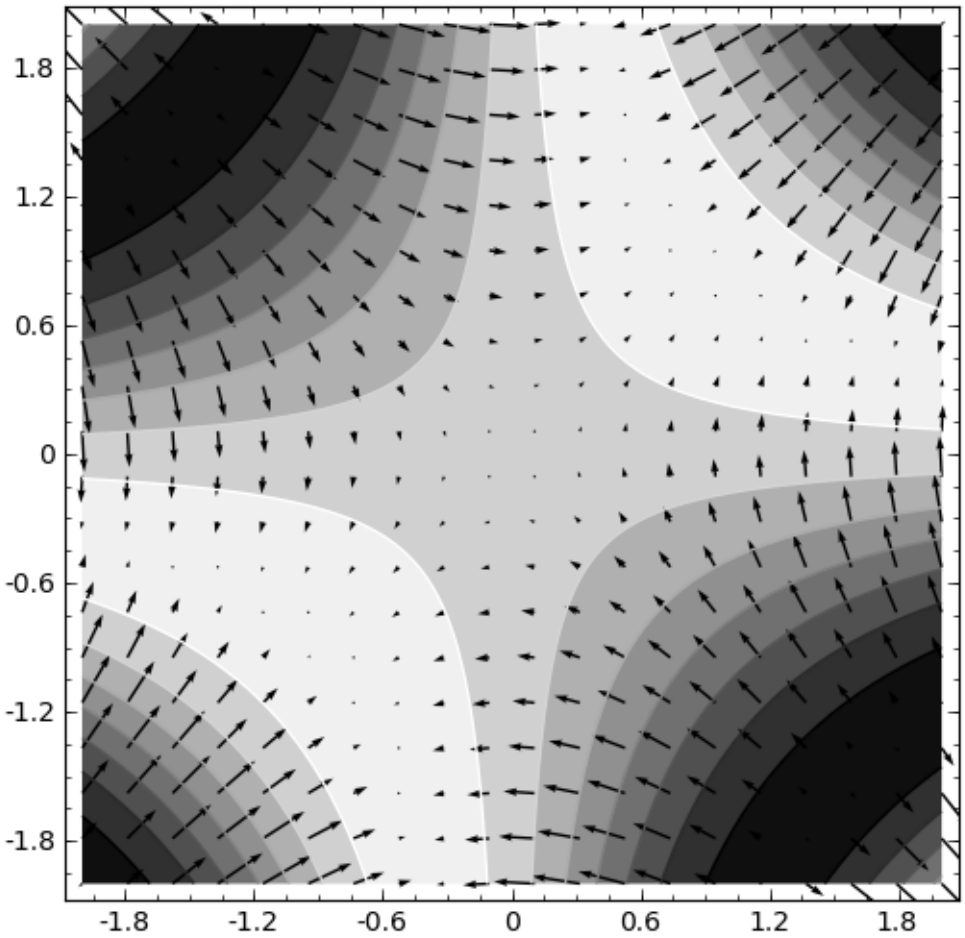
```
sage: f7=sin(x*y)+cos(x*y)
sage: f7.show()
```

$$\sin(xy) + \cos(xy)$$

```
sage: plot_vector_field(f7.gradient(),(x,-2,2),(y,-2,2)).show(aspect_ratio=1)
```



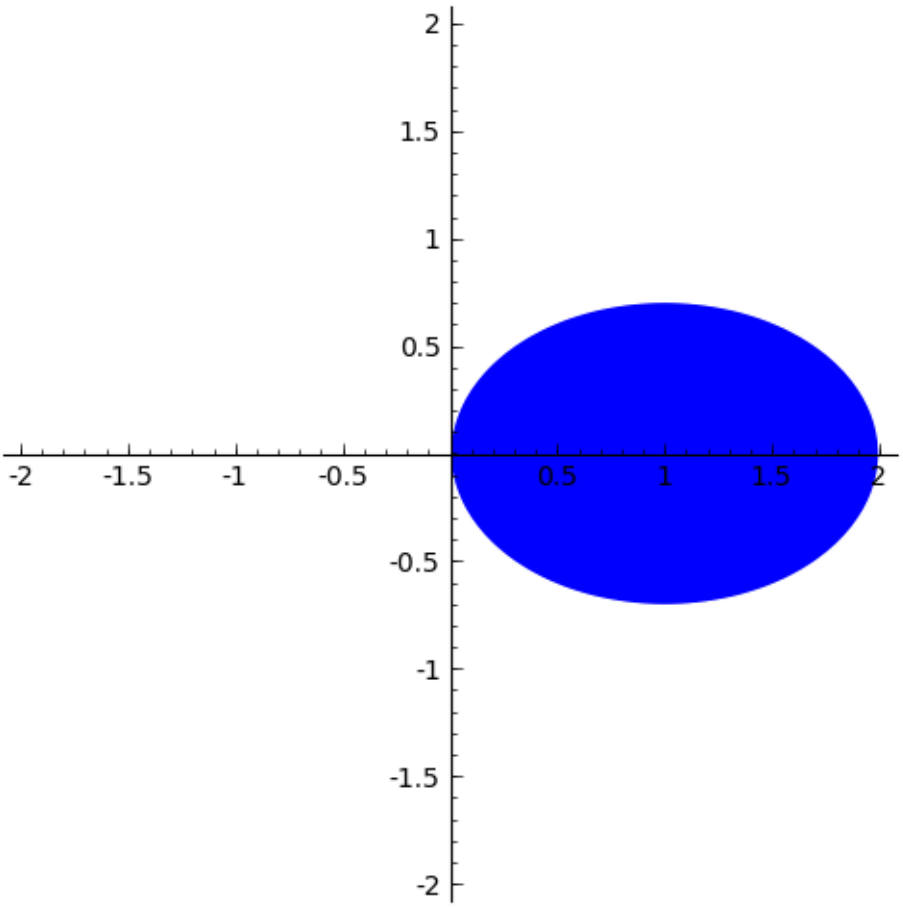

```
sage: (contour_plot(f7,(x,-2,2),(y,-2,2)) + plot_vector_field(f7.gradient(),(x,-2,2),(y,-2,2))).show
```



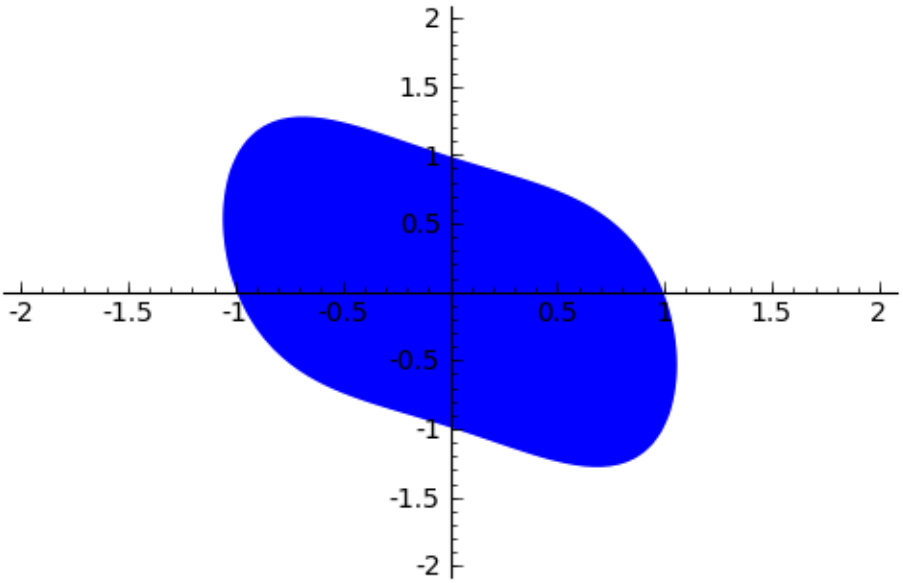
Regiones

Otro comando interesante es el comando `region_plot`, que sirve para dibujar regiones definidas por desigualdades. Se puede llamar de varias formas para representar regiones definidas por varias ecuaciones (ver la ayuda para más información). En esta sesión lo usaremos para visualizar regiones de integración.

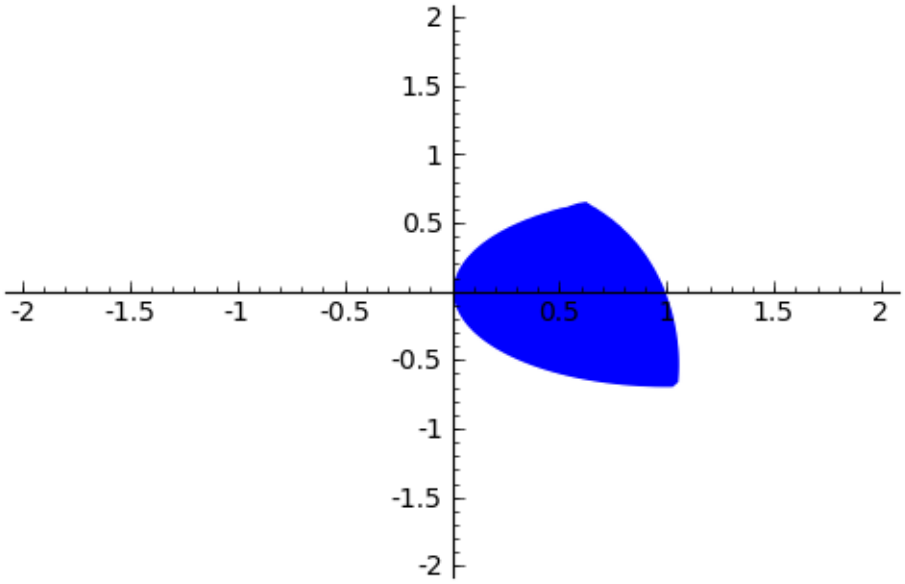
```
sage: #Una elipse
sage: region_plot((x-1)^2 + 2*y^2 < 1,(x,-2,2),(y,-2,2)).show(aspect_ratio=1)
```




```
sage: #Un algo
sage: region_plot(x^4+x*y+y^2<1,(x,-2,2),(y,-2,2))
```



```
sage: #La interseccion de las dos regiones anteriores
sage: region_plot([x^4+x*y+y^2<1, (x-1)^2+2*y^2<1],(x,-2,2),(y,-2,2))
```



Ejercicio

dibuja la región 0<=x<=1,x*(1-x)<=y<=sin(pi*x)

6.5.2 Puntos críticos y extremos

Ceros del gradiente

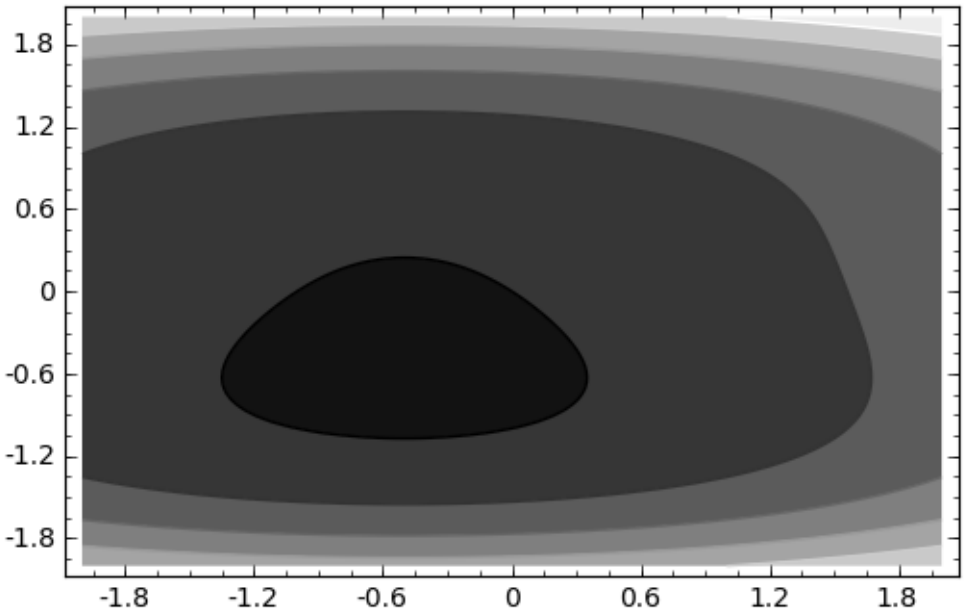
La forma más naive de buscar puntos críticos es resolver el sistema de ecuaciones no lineales

$$\nabla(f) = 0$$

El sistema es trivial de plantear usando el método gradient visto antes, y podemos intentar resolver el sistema llamando a `solve` :

```
solve(f.gradient(),f.variables())
```

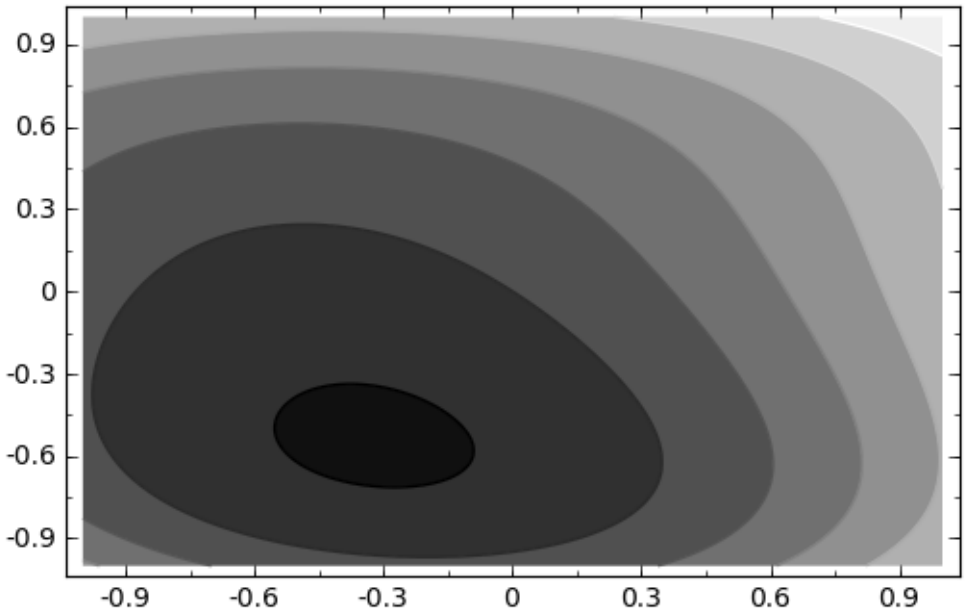
```
sage: #Esta funcion parece tener un minimo cerca de (-0.5,-0.5)
sage: var('x y')
sage: f = x^2 + y^4 + x + y
sage: contour_plot(f,(x,-2,2),(y,-2,2)).show()
```




```
sage: #Obtenemos varias soluciones "extra":
sage: #?Cual corresponde al minimo?
sage: solve(f.gradient(),f.variables())
[[x == -0.5, y == (0.314980262474 + 0.545561817986*I)], [x == -0.5, y == (0.314980262474 - 0.5455618
```

Por supuesto, con funciones más complicadas, este tipo de sistemas de ecuaciones no lineales es casi imposible de resolver.

```
sage: #Esta funcion parece tener un minimo cerca de (-0.3,-0.5)
sage: var('x y')
sage: f = x^2+ y^4 + sin(x+y)
sage: contour_plot(f,(x,-1,1),(y,-1,1),plot_points=300).show()
```



```
sage: solve(f.gradient(),[x,y])
[2*x + cos(x + y), 4*y^3 + cos(x + y)]
```

Mínimos de una función de varias variables obtenidos numéricamente

El paquete `scipy.optimize` contiene varios métodos relacionados con problemas de optimización. El método `scipy.optimize.fmin` busca mínimos de una función de varias variables partiendo de un punto inicial. Para llamar al método, es necesario definir una función de python que acepta como argumento una lista con los valores de las variables y devuelve el valor de la función en el punto. El resultado es el mínimo propuesto, y una estimación del error cometido. Además, nos imprime un resumen de la ejecución del algoritmo numérico.

```
sage: from scipy.optimize import fmin
sage: def f_0(xs):
...       x,y = xs
...       return x^2 + y^4 + x + y
sage: fmin(f_0,(0,0))
Optimization terminated successfully.
         Current function value: -0.722470
         Iterations: 58
         Function evaluations: 113
array([-0.500004  , -0.62998934])
```

```
sage: from scipy.optimize import fmin
sage: def f_0(xs):
...       x,y = xs
...       return x^2 + y^4 + sin(x + y)
sage: fmin(f_0,(0,0))
Optimization terminated successfully.
         Current function value: -0.570485
         Iterations: 60
         Function evaluations: 109
array([-0.32318198, -0.54469709])
```

6.5.3 Integrales

La integración simbólica sobre regiones cuadradas no supone ningún quebradero de cabeza (siempre que el sistema sea capaz de encontrar las integrales del integrando). Para calcular la integral doble, integramos primero respecto de una variable y después respecto de la otra.

Ejemplo:

$$\iint_Q x^2\, e^y\, dx\, dy,\ Q = [-1,1] \times [0, \log 2]$$

```
sage: f = x^2*e^y
sage: f.integral(x,-1,1).integral(y,0,log(2))
2/3
```

Para regiones más generales, tenemos que operar igual que en clase de Cálculo, descomponiendo la región en subregiones del tipo:

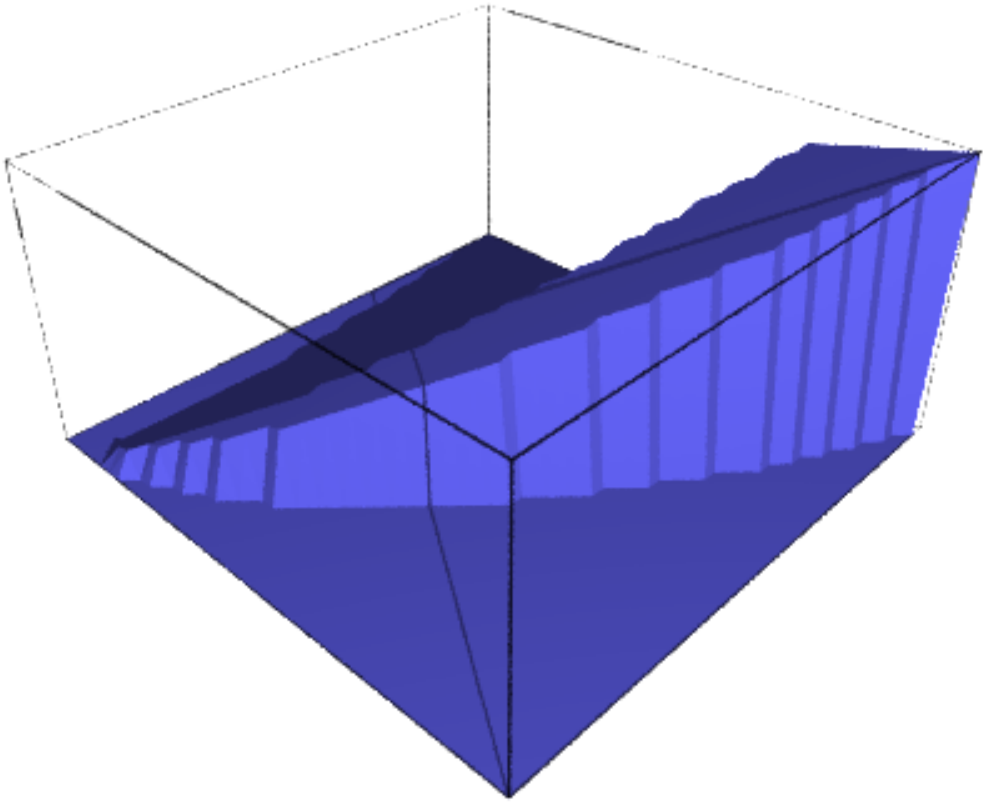
$$\{x_1 \leq x \leq x_2, g(x) \leq y \leq h(x)\}$$

Ejercicio resuelto

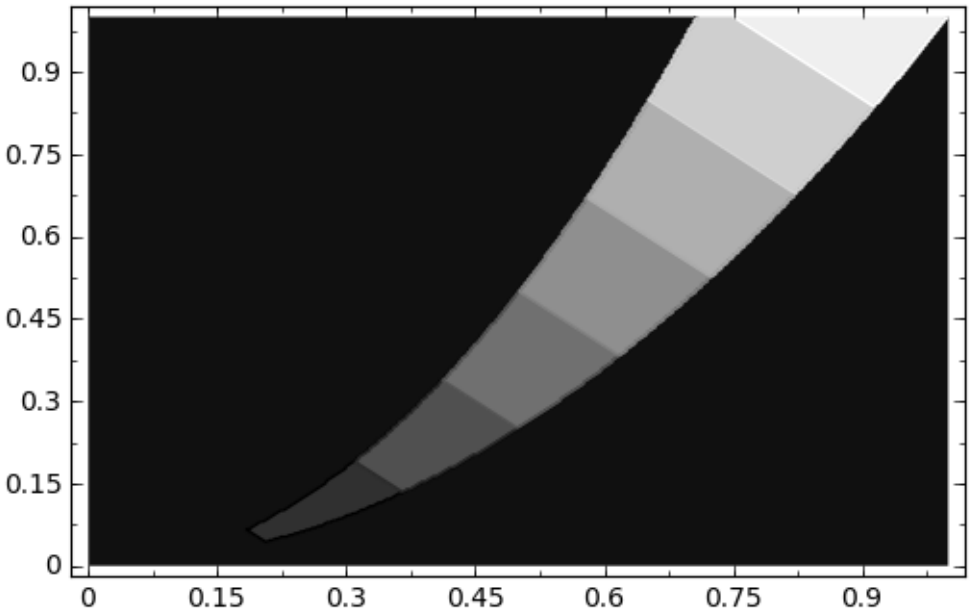
Representa el conjunto de los valores $f(x, y)$ sobre $Q = [0, 1] \times [0, 1]$ y calcula el volumen del sólido así obtenido.

$$f(x,y) = \begin{cases} x+y & \text{si} \quad x^2 \leq y \leq 2x^2, \\ 0 & \text{en el resto.} \end{cases}$$

```
sage: #Comenzamos por dibujar la funcion
sage: #!!!Atencion!!! Esta vez el argumento de plot3d no es una
sage: #funcion de dos variables simbolicas, sino una funcion
sage: #normal de python que acepta como argumentos dos numeros
sage: #reales y devuelve otro
sage: def f(a,b):
...       if a^2 <= b <= 2*a^2:
...           return a+b
...       else:
...           return 0
sage: plot3d(f,(0,1),(0,1)).show(viewer='tachyon')
```



```
sage: #Un dibujo de curvas de nivel es igualmente ilustrativo
sage: contour_plot(f,(0,1),(0,1),plot_points=300).show()
```




```
sage: #para calcular la integral simbolica necesitamos definir
sage: #una funcion de dos variables simbolicas, pero debemos
sage: #integrar solo en la region adecuada
sage: var('x y')
sage: g = x + y
sage: print g
sage: print g.integral(y,x^2,2*x^2)
sage: print g.integral(y,x^2,2*x^2).integral(x,0,1)
x + y
3/2*x^4 + x^3
11/20
```

Cuando sea imposible evaluar la integral de una función sobre una región básica del plano de forma exacta, podemos hacerlo de forma numérica con `scipy.integrate.dblquad`

```
sage: #Es necesario importar la funcion dblquad
sage: #del paquete scipy.integrate al que pertenece
sage: from scipy.integrate import dblquad

sage: dblquad?
<html>...</html>
```

Calculamos la misma integral que calculamos antes de forma simbólica usando `dblquad`.

```
sage: def g(x,y):
...       return x+y
sage: def gfun(x):
...       return x^2
sage: def hfun(x):
...       return 2*x^2
sage: dblquad(g,0,1,gfun,hfun)
(0.54999999999999993, 6.1062266354383602e-15)
```

Cambios de coordenadas

Probemos a calcular una integral en coordenadas cartesianas y polares. Recordemos que:

$$\iint_R f(x,y)dx\,dy = \iint_R r\,f(r,\theta)dr\,d\theta$$

```
sage: var('x y r theta')
(x, y, r, theta)

sage: f = x^2+y^2
sage: print f.integral(y,-sqrt(1-x^2),sqrt(1-x^2)).integral(x,-1,1)
CODE:
    sage47 : integrate(sage43,sage44,sage45,sage46)$
Maxima ERROR:

defint: lower limit of integration must be real; found -sqrt(1-x^2)
 -- an error. To debug this try: debugmode(true);
Traceback (most recent call last):
...
TypeError: Error executing code in Maxima
```

El código anterior produce un error. Hurgando un poco en la traza del error, leemos:

```
TypeError: Error executing code in Maxima
CODE:
    sage9 : integrate(sage5,sage6,sage7,sage8)
Maxima ERROR:
defint: lower limit of integration must be real; found \-sqrt(1\-x^2)
```

Recordemos que SAGE usa otras piezas de software con más tradición siempre que puede. En este caso, le pasa la tarea de realizar la integral a la librería **Maxima** . Esta librería arroja un error bastante ilustrativo: "el límite de integración debe ser real", pero sqrt(1-x^2) puede ser imaginario. Entender la causa del error no siempre garantiza que se pueda superar el problema, pero en este caso la solución es informar a Sage de que puede asumir que x está entre -1 y 1.

En cualquier caso, nada nos garantiza el éxito al intentar una integral de forma simbólica, así que deberíamos estar preparadas para hacer la integral con **dblquad** si todo lo demás falla.

```
sage: assume(1-x^2>0)
sage: f = x^2+y^2
sage: print f.integral(y,-sqrt(1-x^2),sqrt(1-x^2)).integral(x,-1,1)
1/2*pi

sage: #Integramos en coordenadas polares.
sage: f_polar = f(x=r*cos(theta),y=r*sin(theta))
sage: print f_polar
sage: #Intentamos simplificar el integrando (no es necesario)
sage: f_polar = f_polar.full_simplify()
sage: print f_polar
sage: integrando_polar = r*f_polar()
sage: integrando_polar.integral(theta,0,2*pi).integral(r,0,1)
r^2*sin(theta)^2 + r^2*cos(theta)^2
r^2
1/2*pi
```

6.6 Ejercicios

6.6.1 1.

Hallar el vector gradiente, en cada punto en el que exista, de las siguientes funciones escalares

- $f(x,y) = e^{-x}\,\cos y$
- $f(x,y,z) = \log\left(x^2 + 2\,y^2 - 3\,z^2\right)$
- $f(x,y) = xy\,\operatorname{sin}\frac{1}{x^2+y^2}$ si $(x,y) \neq (0,0)$ y $f(0,0) = 0$.

6.6.2 2.

Calcular la distancia mínima entre los puntos de la gráfica de $f(x,y) = \frac{1}{4\,x\,y}$ y el punto $(0,0,0)$.

6.6.3 3.

Calcula las derivadas parciales $\partial_1\partial_2 f$ y $\partial_2\partial_1 f$ en el punto (0,0):

$$f(x,y) = \begin{cases} xy\frac{x^2-y^2}{x^2+y^2} & \text{si}(x,y) \neq (0,0) \\ 0 & \text{si}(x,y) = (0,0) \end{cases}$$

Decide si son iguales, y si son continuas en (0,0).

Nota: esta función muestra que el recíproco del teorema de Clairaut (tb llamado teorema de Schwartz) es falso: http://es.wikipedia.org/wiki/Teorema_de_Clairaut

6.6.4 4.

Hallar los puntos críticos y determinar cuáles son los máximos locales, mínimos locales o puntos silla:

$$f(x,y) = x^2 + y^2 - 2\,x\,y. \quad f(x,y) = x^2 + y^2 + x\,y - 2\,x - 4\,y + 10.$$
$$f(x,y) = x\,y. \quad f(x,y) = 3\,x^2 - 4\,y^2 + x\,y.$$

6.6.5 5.

Representa el conjunto de los valores $f(x,y)$ sobre $Q = [0,1] \times [0,1]$ y calcula el volumen del sólido así obtenido.

$$f(x,y) = \begin{cases} 1 - (x+y) & \text{si} \quad x + y \leq 1, \\ 0 & \text{en otro caso.} \end{cases}$$

6.6.6 6.

Investiga la ayuda de `region_plot` con el objetivo de dibujar la unión de dos regiones. Es posible que la calidad del dibujo empeore, pero la ayuda también explica cómo mejorar la precisión del dibujo.

6.6.7 7.

En los siguientes apartados, se supone que la integral de una función positiva f sobre la región Ω se reduce a la integral iterada que se da. En cada caso, se pide determinar y dibujar la región Ω e invertir el orden de integración.

$$\int_0^2 \left(\int_{y^2}^{2y} f(x,y)\,dx \right) dy. \qquad \int_1^4 \left(\int_{\sqrt{x}}^{2} f(x,y)\,dy \right) dx\,,$$
$$\int_1^e \left(\int_0^{\log x} f(x,y)\,dy \right) dx. \qquad \int_0^{\pi} \left(\int_{-\sin x/2}^{\sin x} f(x,y)\,dy \right) dx.$$

Indicación: usa `region_plot` para dibujar la región. Después de hacer el cambio en el orden de integración, dibuja de nuevo la región para comprobar que obtienes el mismo resultado.

6.6.8 8.

Hallar el valor de las siguientes integrales, determinando y dibujando en cada caso el recinto de integración

- $\iiint_Q (2x+3y+z)\,dx\,dy\,dz$, con $Q = [1,2]\times[-1,1]\times[0,1]$.
- $\iiint_T x^2\,\cos z\,dx\,dy\,dz$, siendo T la región limitada por los planos $z=0, z=\pi, y=0, y=1, x=0, x+y=1$.
- $\iiint_\Omega x\,y^2\,z^3\,dx\,dy\,dz$, siendo Ω el sólido limitado por la superficie $z = x\,y$ y los planos $y=x, x=1$ y $z=0$.

6.6.9 9. Áreas con el teorema de Green

Veamos cómo realizar integrales de línea como las que aparecen en el teorema de Green. Este teorema se suele escribir así:

$$\oint_C (L\,\mathrm{d}x + M\,\mathrm{d}y) = \iint_D \left(\frac{\partial M}{\partial x} - \frac{\partial L}{\partial y} \right) \mathrm{d}x\,\mathrm{d}y$$

Con frecuencia, ésta es una manera práctica de evaluar integrales planas sobre regiones que no admiten una descomposición sencilla en regiones simples. Por ejemplo, podemos calcular áreas:

$$\oint_C x\,\mathrm{d}y = \iint_D 1\,\mathrm{d}x\,\mathrm{d}y = |D|$$

- Calcula el área de un triángulo con este método. Comprueba el resultado calculando el área de triángulos cuya área puedas calcular a mano.
- Calcula el área de un polígono de n lados.
- Demuestra que el área de la elipse es $\frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1$ es $A = \pi\,a\,b$

Indicación: Recuerda que una parametrización de la elipse es $t \to (a\cos(\theta), b\sin(\theta))$.

6.7 Curvas planas

En esta sesión vamos a utilizar varios métodos de representación de curvas en el plano:

- en paramétricas;
- en coordenadas polares;
- por una ecuación implícita.

6.7.1 Ecuaciones paramétricas

Pensaremos en una curva plana como la imagen por una función continua $\gamma : I \longrightarrow \mathbb{R}^2$, con I un intervalo de números reales. Si tomamos t como variable independiente ($t \in \mathbb{R}$), describiremos la imagen como el conjunto de puntos

$$\Gamma = \gamma(I) = \big\{(x(t), y(t)) \,:\, t \in I\big\}$$

o simplemente

$$\Gamma : \big(\, x(t)\,,\, y(t)\,\big)$$

Cuando el dominio de valores para el parámetro t es un intervalo, digamos $[a, b]$, la curva tiene como **extremos** los puntos: $(x(a), y(a))$ y $(x(b), y(b))$.

Ejemplos

- La curva $\Gamma : (t, t)$, para $t \in (0, 1)$, es el segmento de recta que une los puntos $(0, 0)$ y $(1, 1)$. En general, si P y Q son dos puntos, un segmento rectilíneo que une P con Q es: $\Gamma : (1 - t) \cdot P + t \cdot Q$, considerando $t \in (0, 1)$.

Antes de continuar, vamos a utilizar Sage para ' *dibujar* ' este ejemplo. Necesitamos que nuestro código sepa interpretar la expresión $(1 - t) \cdot P + t \cdot Q$ como lo haríamos nosotros, a saber: si $P = (x_0, y_0)$ y $Q = (x_1, y_1)$, el resultado queda:

$$\begin{aligned}(1-t)\cdot(x_0,y_0)+t\cdot(x_1,y_1) &= \big((1-t)x_0,(1-t)y_0\big)+\big(tx_1,ty_1\big)\\ &= \big(x_0+t(x_1-x_0)\,,\,y_0+t(y_1-y_0)\big)\end{aligned}$$

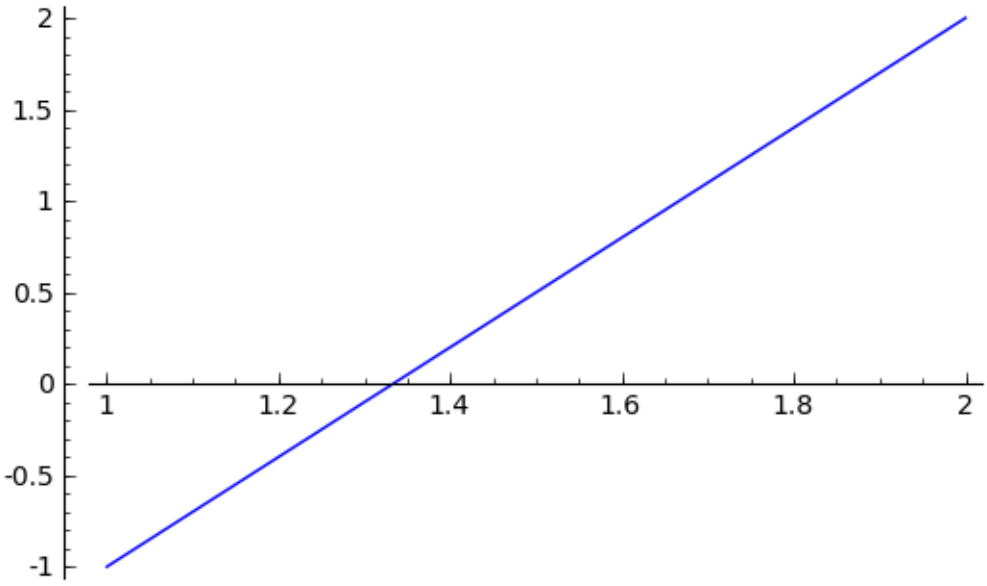
Esta manipulación (afín), no es más que tratar a los puntos P y Q como vectores (los vectores OP y OQ, cuando se ha fijado un punto como origen del plano afín). En el siguiente código se consigue este propósito. Utilizamos la función `parametric_plot` de Sage (leer la ayuda antes de seguir). Obsérvese en qué manera se explicita el intervalo para la variable t.

```
sage: parametric_plot?
<html>...</html>
```

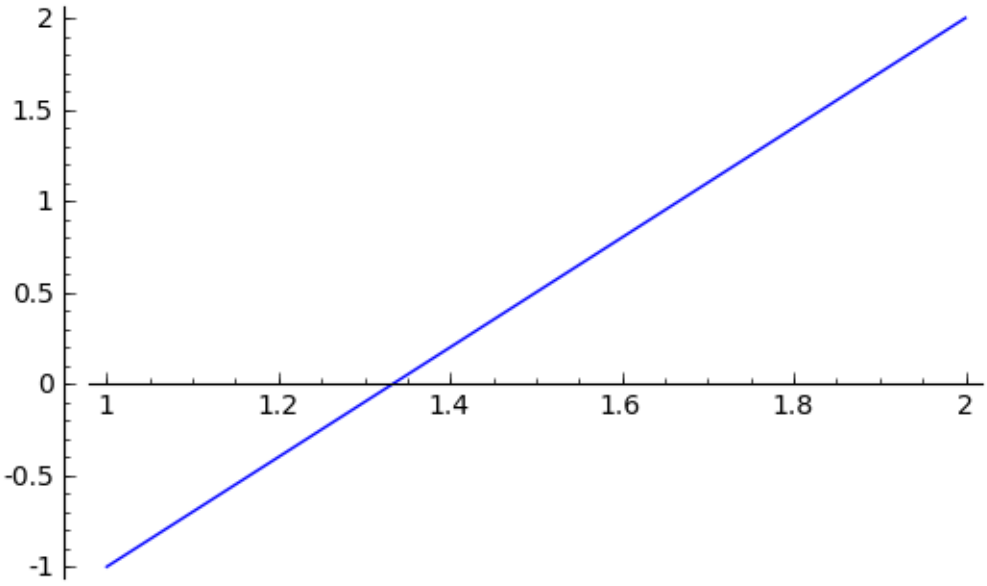
```
sage: # Segmento rectilíneo entre dos puntos P y Q
sage: var('t')
sage: P=vector((1,-1))
sage: Q=vector((2,2))
sage: #Observación
sage: show((1-t)*P+t*Q)
```

$$(t + 1, 3\,t - 1)$$

```
sage: #Aviso: no funciona en la versión 4.3.1
sage: parametric_plot((1-t)*P+t*Q, (t, 0, 1))
```




```
sage: #Valido en version 4.1.1
sage: parametric_plot( (t + 1, 3*t - 1) , (t, 0, 1))
```

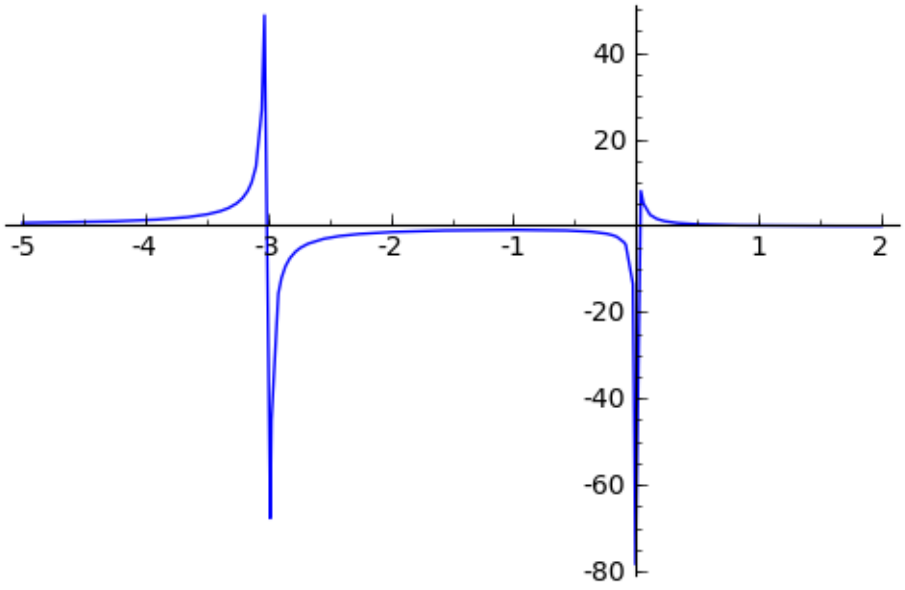


Más ejemplos

- Si tenemos una función (continua) real de variable real, $f : \mathbb{R} \longrightarrow \mathbb{R}$, la gráfica es una curva en el plano, $\{(x, f(x)) : x \in \mathbb{R}\}$, que podemos presentar paramétricamente (tomando $x = t$). Estos son casos particulares de curvas, en las que no hay más de un punto en cada vertical (¿por qué?).
- Una parábola: (t, t^2)
- Una exponencial: (t, e^t)
- Curvas trigonométricas: $(t, \sin(t))$, $(t, \cos(t))$, ...

- Funciones racionales: $\left(t, \frac{1-t}{t^2+3t}\right)$, $t \neq -3, 0$. En los puntos en que una función tal no es continua, la gráfica no es fiel (comprobarlo).
- Polinomios, ...

```
sage: parametric_plot((t,(1-t)/(t^2+3*t)), (t,-5,2))
```



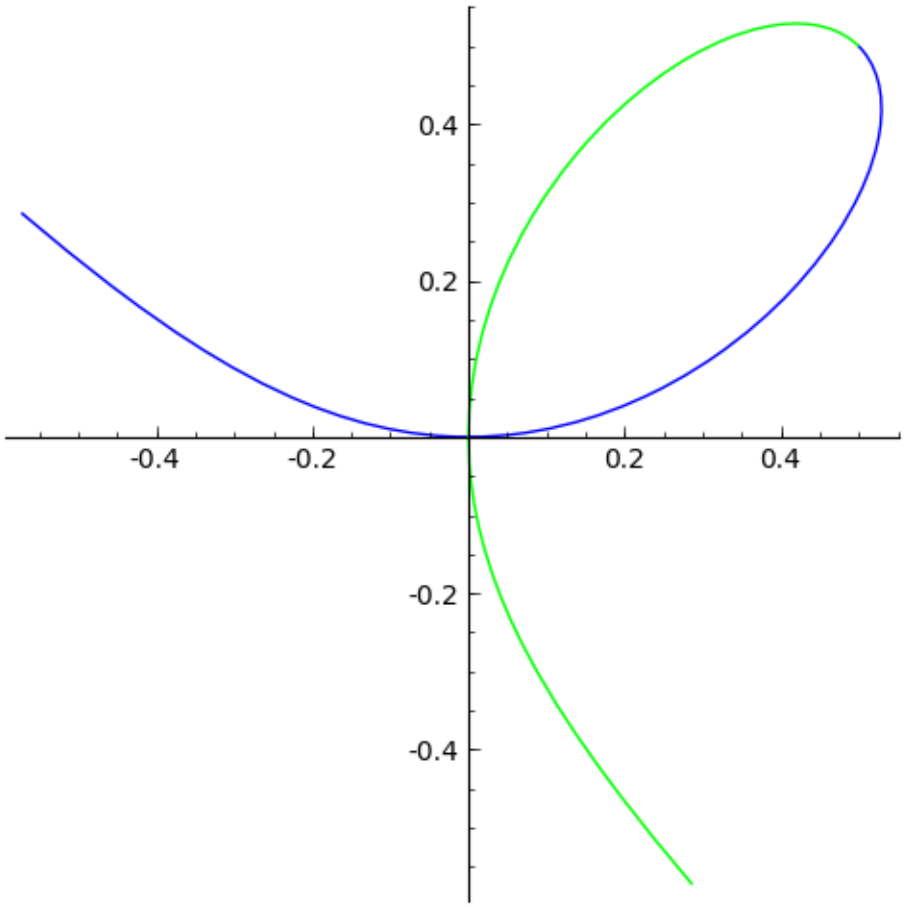
Y los más interesantes

La mayoría del lenguaje de curvas parametrizadas viene del estudio del movimiento de una partícula. En esos casos, el parámetro de la curva suele ser el tiempo (¡qué buena elección de letra t!), y cada punto de la curva marca la posición de la partícula en cada momento. Así el segmento rectilíneo entre dos puntos P y Q descrito por $\Gamma : (1-t)\cdot P + t\cdot Q$, $t \in [0,1]$ sería la trayectoria descrita por una partícula que viaja en línea recta, partiendo de P y llegando a Q en una unidad de tiempo ($t \in [0,1]$).

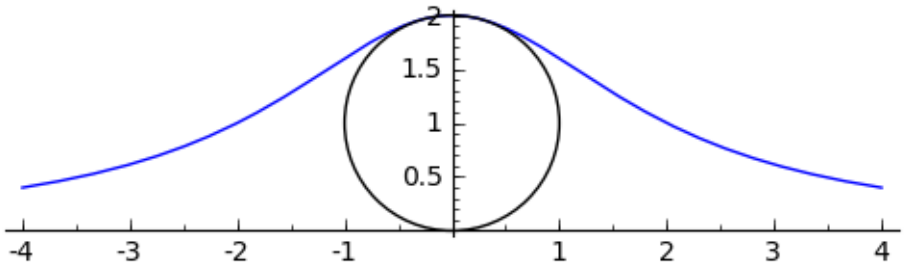
Este enfoque nos surte de una cantidad tremenda de ejemplos, históricamente estudiados por grandes físicos y matemáticos, de los que a continuación exponemos una brevísima lista (para ver más visitar, por ejemplo: mathcurve.com , epsilones.com):

- Circunferencia (centrada en el origen y de radio ρ): $\left(\rho\cos t, \rho\sin t\right)$, $t \in [0, 2\pi)$
- Bruja de Agnesi: $\left(2at, 2a/(1+t^2)\right)$ (a es el radio de la circunferencia base —ver construcción más abajo)
- Astroide: $(a\cos^3(t), a\sin^3(t))$, $t \in [0, 2\pi)$
- El folium de Descartes: $\left(\frac{3at}{1+t^3}, \frac{3at^2}{1+t^3}\right)$, $t \in (-1, 1)$. Nota: el cambio t por $1/t$, cambia x e y.
- ???: $\left(\sin(3t), \cos(4t) + 4\cos t\right)$, $t \in [0, 2\pi)$

```
sage: # Dibujamos el folium de Descartes pegando dos ramas simétricas respecto al eje (y=x).
sage: rama1=parametric_plot((t/(1+t^3),t^2/(1+t^3)), (t,-.5,1))
sage: rama2=parametric_plot((t^2/(1+t^3),t/(1+t^3)), (t,-.5,1), color=(0,1,0))
sage: show(rama1+rama2, aspect_ratio=1)
```



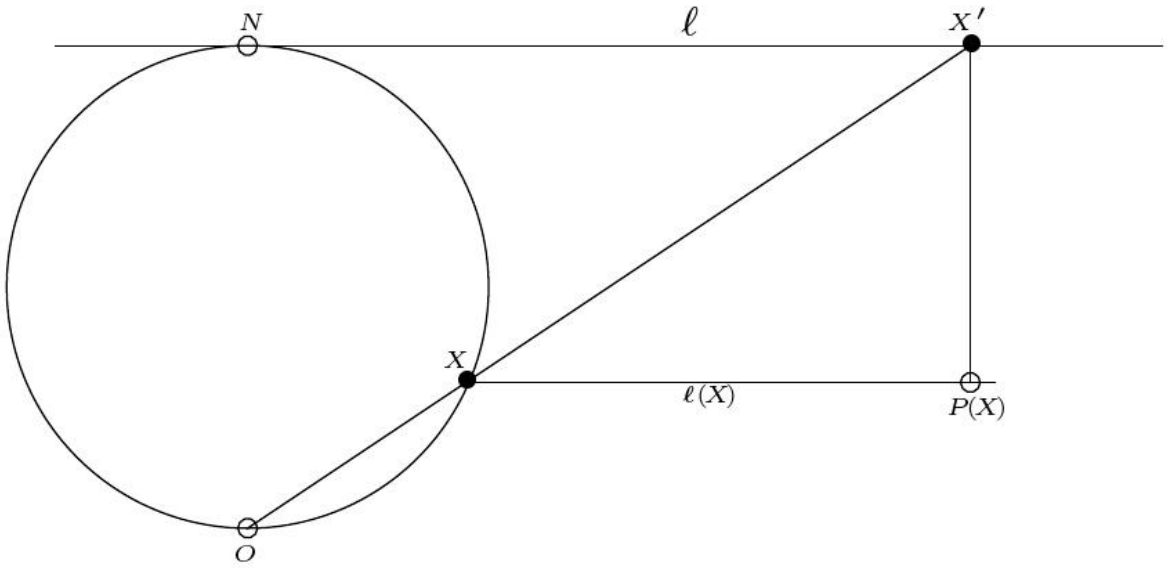

```
sage: #Cúbica (o bruja) de Agnesi
sage: var('t theta x y')
sage: a = 1
sage: bruja = parametric_plot((2*a*t,2*a/(t^2+1)), (t, -2, 2))
sage: #circle dibuja una circunferencia de centro y radio dados
sage: circ_bruja = circle((0,a),a)
sage: ambas = bruja + circ_bruja
sage: show(ambas, aspect_ratio=1)
```



Construcción de la curva de Agnesi

Se consideran dos puntos diametralmente opuestos en una circunferencia, O y N, y la tangente, ℓ, a la circunferencia por uno de ellos, digamos que por N. Desde el otro, hacemos la siguiente construcción

- para cada punto de la circunferencia, X, tomamos el punto de corte de la recta OX con la recta ℓ: $X' = OX \cap \ell$;
- completamos un triángulo rectángulo con X, X' y un tercer punto, $P(X)$, en la recta, $\ell(X)$, paralela a ℓ por el punto X: el ángulo recto sobre el vértice nuevo, $P(X)$.
- La curva de Agnesi es la descrita por los puntos $P(X)$.




```
sage: var('a t s x y')
(a, t, s, x, y)

sage: solve((2*a*t*s)^2+(2*a*s-a)^2-a^2,s)
[s == (1/(t^2 + 1)), s == 0]
```

6.7.2 Coordenadas polares

Una manera alternativa de indicar la posición de cada punto es mediante sus coordenadas polares (ρ, θ), siendo ρ la distancia del punto al origen, y θ el ángulo que forma el vector que une el origen con el punto y el semieje horizontal positivo, es decir: si $P = (x, y)$

$$\rho = \sqrt{x^2 + y^2}\,, \qquad \theta = \arctan\frac{y}{x}\,.$$

Obsérvese que la función $\arctan$ nos devuelve el valor correcto del ángulo salvo, quizás, un error de π (pues $\tan(\alpha) = \tan(\alpha \pm \pi)$). El cambio inverso, de polares a cartesianas, es más preciso:

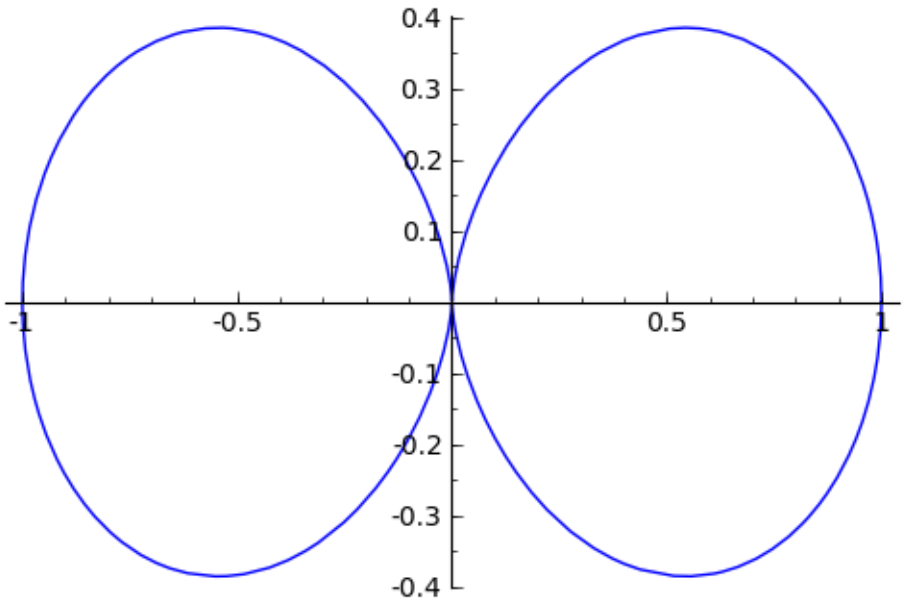
$$x = \rho\cos\theta\,, \qquad y = \rho\sin\theta\,.$$

Es claro que $\rho \geq 0$ así como que basta tomar ángulos θ en el intervalo $[0, 2\pi)$, por ejemplo.

Para dar la ecuación en polares de una curva, se suele expresar el radio como una función del ángulo, pero se representa en el plano cartesiano. Así, si dibujamos la curva $\Gamma : \rho = cos^2(\theta)\,,\ \theta \in [0, 2\pi)$, estaremos dibujando los puntos del plano (x, y), con $x = \rho\cos\theta = \cos^2(\theta)\cdot\cos\theta = \cos^3\theta$, e $y = \cos^2\theta\cdot\sin\theta$, variando θ, en este caso, entre 0 y 2π.

La función de Sage que utilizamos en este caso es `polar_plot` (revisar su ayuda en línea), que es equivalente a utilizar el valor polar=True en un plot() .

```
sage: polar_plot?
<html>...</html>

sage: var('theta')
sage: polar_plot(cos(theta)^2, (theta,0,2*pi))
```



Ejemplos

- Recta de ecuación $Ax + By = 1$ (no pasa por el origen). Si $p = 1/\sqrt{A^2 + B^2}$ y α es el ángulo definido por $\cos\alpha = Ap$, $\sin\alpha = Bp$, entonces: $\rho = \dfrac{p}{\cos(\theta - \alpha)}$ es su ecuación en polares.
- Recta que pasa por el origen: en este caso la descripción es más sencilla, pues todos los puntos de la recta (al unirse al origen) forman un ángulo constante con el eje de abcisas. Así, si dicho ángulo es α_0 la ecuación es $\theta = \alpha_0$. Ahora bien, no podemos indicarle a Sage cómo dibujarlas utilizando polar_plot() (ocurre como en la representación de rectas verticales, $x = \text{constante}$, en coordenadas cartesianas).

Como vemos, las rectas no son muy amigables para representar en polares. Sigamos con ejemplos más enriquecedores.

- Circunferencia (con centro el origen): $\rho = \text{constante}$, $\theta \in [0, 2\pi)$. Y si el ángulo lo variamos menos, conseguimos cualquier arco de circunferencia.
- Circunferencia que pasa por el origen: $\rho = A\cos\theta + B\sin\theta$, $\theta \in [0, 2\pi)$. Nota: el centro es el punto de coordenadas cartesianas $(A/2, B/2)$.
- Cónica con un foco en el origen: la ecuación de la cónica, dado un foco, necesita conocer la recta directriz y la excentricidad e. Si tal recta tiene ecuación cartesiana $x\cos\alpha + y\sin\alpha = q$, la cónica es el lugar de puntos M tales que $\|OM\| = e\|MH\|$, siendo H la proyección ortogonal de M sobre la directriz. En tal caso, la ecuación polar de la cónica queda:

$$\rho = \frac{p}{1 + e\cos(\theta - \alpha)}\,, \qquad p = eq\,.$$

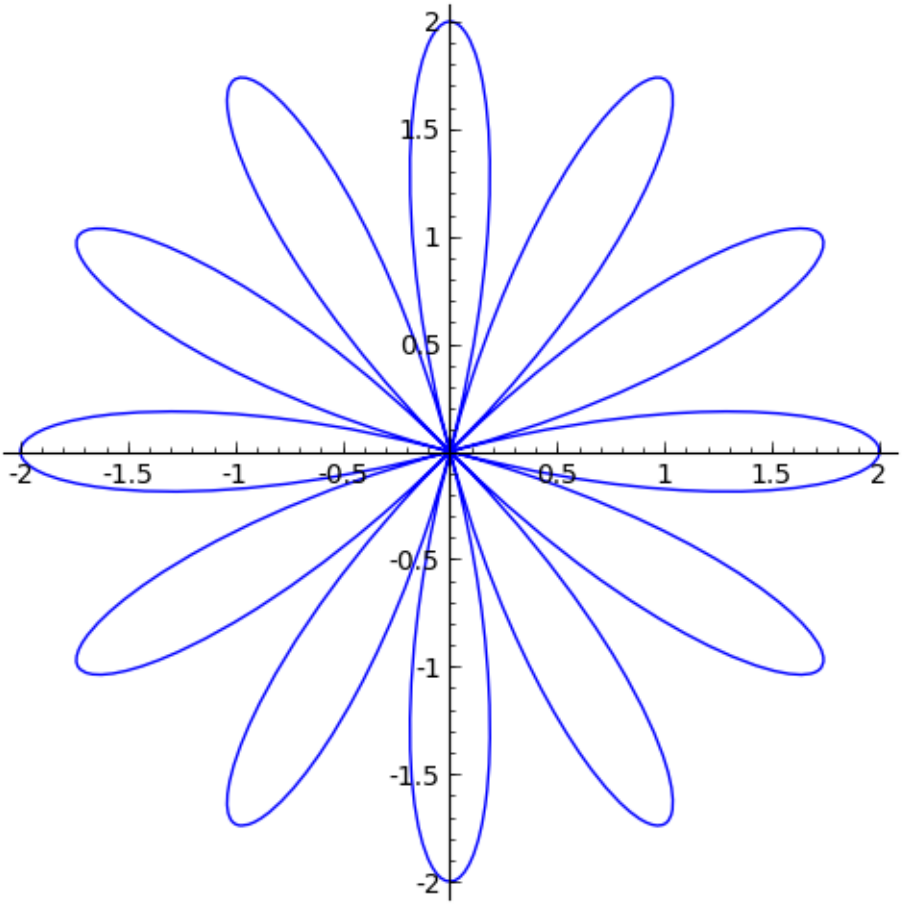
La recta Δ de ecuación $\theta = \alpha$ es eje de simetría. Cuando $e = 1$ la cónica es una parábola, Δ su eje de simetría y $(p/2, \alpha)$ su vértice. Cuando $e \neq 1$ la cónica es una cónica con centro, Δ el eje focal y sus vértices se calculan sustituyendo θ por α y $\alpha + \pi$.

Últimos ejemplos

Por último una lista de curvas famosas con sus ecuaciones polares:

- Rosas: $\rho = a\cos(n\theta)$, $\theta \in [0, 2\pi)$. **Ejercicio** : averiguar el número de pétalos para cada valor natural de n. Nota: n puede ser cualquier real positivo.
- La cardiode: $\rho = a(1 + \cos\theta)$, $\theta \in [0, 2\pi)$.
- La cisoide: $\rho = a\frac{\sin^2\theta}{\cos\theta}$, $\theta \in [0, 2\pi)$.
- La estrofoide: $\rho = a\frac{\cos(2\theta)}{\cos\theta}$, $\theta \in [0, 2\pi)$.
- La lemniscata de Bernoulli: $\rho^2 = a^2\cos(2\theta)$, $\theta \in [0, 2\pi)$.
- Espirales:
- Espiral de Arquímedes: $\rho = a\theta$, $\theta \in [0, +\infty)$.
- Espiral hiperbólica: $\rho = a/\theta$, $\theta \in [0, +\infty)$.
- Espiral logarítmica: $\rho = ae^{m\theta}$, $\theta \in [0, +\infty)$.

```
sage: rosas=polar_plot(2*cos(6*theta), (theta,0,2*pi))
sage: show(rosas, aspect_ratio=1)
```



6.7.3 Coordenadas cartesianas

Aunque no han dejado de usarse, hasta aquí no hemos pedido a Sage que represente una curva cuyas 'ecuaciones' le vienen dadas en función de sus coordenadas cartesianas. Lo hasta ahora habitual es encontrarnos con que la coordenada y es una función de la coordenada x: $y = f(x)$. Las herramientas del cálculo diferencial nos ayudan a esbozar este tipo de gráficas. En este apartado vamos a adoptar un punto de vista más general, dando una ecuación de la curva en la que ambas variables puedan aparecer en la expresión sin que sea necesario despejar una en función de la otra.

Ecuaciones implícitas

- La ecuación $Ax + By = C$, con A, B y C números reales, nos sirve para describir todos los puntos del plano cuyas coordenadas, (x, y), la verifican. Es fácil comprobar que todos estos puntos quedan alineados, es decir describen una línea recta. De la ecuación dada decimos que es su *ecuación cartesiana* o *implícita* , y basta, para calcularla con dar dos puntos de la recta, o un punto y una dirección (vector director).
- Una ecuación implícita de la recta que pasa por los puntos de coordenadas (x_0, y_0), (x_1, y_1) surge de desarrollar la siguiente igualdad:

$$\det\begin{pmatrix} x - x_0 & x_1 - x_0 \\ y - y_0 & y_1 - y_0 \end{pmatrix} = 0\,.$$

- Si conocemos que la recta pasa por el punto de coordenadas (x_0, y_0) y la dirección de la recta es la del vector $\mathbf{v} = (v_x, v_y)$, una ecuación implícita la obtenemos al desarrollar:

$$\det\begin{pmatrix} x - x_0 & v_x \\ y - y_0 & v_y \end{pmatrix} = 0\,.$$

- Con ambos puntos de vista, los coeficientes A y B de la ecuación resultante, $Ax + By = C$, determinan un vector $\mathbf{n} = (A, B)$ (vector *normal* de la recta) que es perpendicular a la recta, en particular: a $(x_1 - x_0, y_1 - y_0)$ en la primera descripción; a $\mathbf{v}$, en la segunda.

Ejercicio

- Encuentra la ecuación de la recta que pasa por dos puntos usando álgebra lineal.

Más ecuaciones implícitas

- *Por cinco puntos distintos del plano pasa una única cónica* . Este sencillo principio es una manera escueta de afirmar que con fijar cinco puntos distintos (en posición general), somos capaces de encontrar (determinar los coeficientes de) una ecuación implícita de la forma $ax^2 + by^2 + cxy + dx + ey + f = 0$, y que cualquier otra es el resultado de multiplicar todos los coeficientes por un factor no nulo. Aunque no entraremos en el concepto general de "puntos en posición general", basta decir que en este caso quiere decir que los puntos son distintos y que no hay tres que estén alineados.

Este problema, dadas las coordenadas de los cinco puntos, se puede resolver con cálculo matricial. Si $P_i = (x_i, y_i)$, para $i = 1, \ldots, 5$, son cinco puntos distintos, la ecuación buscada, de la forma $ax^2 + by^2 + cxy + dx + ey + f = 0$,

es verificada por todos ellos, y así se plantean las cinco ecuaciones

$$\begin{aligned} x_1^2a + y_1^2b + x_1y_1c + x_1d + y_1e + 1 \cdot f &= 0 \\ x_2^2a + y_2^2b + x_2y_2c + x_2d + y_2e + 1 \cdot f &= 0 \\ x_3^2a + y_3^2b + x_3y_3c + x_3d + y_3e + 1 \cdot f &= 0 \\ x_4^2a + y_4^2b + x_4y_4c + x_4d + y_4e + 1 \cdot f &= 0 \\ x_5^2a + y_5^2b + x_5y_5c + x_5d + y_5e + 1 \cdot f &= 0 \end{aligned}$$

que son lineales en sus incógnitas (los coeficientes a, b, ..., f). Este es un sistema lineal compatible, y si los puntos están en posición general su matriz de coeficientes es de rango 5. Si $(a_0, b_0, c_0, d_0, e_0, f_0)$ es una solución (no trivial), cualquier otra es de la forma $\lambda \cdot (a_0, b_0, c_0, d_0, e_0, f_0)$. Pero es obvio que todas estas soluciones nos dan ecuaciones para una misma curva (¿por qué?).

Encuentra la cónica que pasa por cinco puntos en posición general:

$$ax^2 + by^2 + cxy + dx + ey + f = 0$$

```
sage: puntos = [(0,0),(0,1),(1,3),(2,1),(1,0)]

sage: var('x y')
sage: coefs = matrix(QQ,5,6)
sage: for j in range(5):
...       x0, y0 = puntos[j]
...       coefs[j,0] = x0^2     #Primera columna (coef. de a)
...       coefs[j,1] = y0^2     #Segunda columna (coef. de b)
...       coefs[j,2] = x0*y0    #Tercera columna (coef. de c)
...       coefs[j,3] = x0       #Cuarta columna (coef. de d)
...       coefs[j,4] = y0       #Quinta columna (coef. de e)
...       coefs[j,5] = 1        #Sexta columna (coef. de f)

sage: show(coefs)
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 9 & 3 & 1 & 3 & 1 \\ 4 & 1 & 2 & 2 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

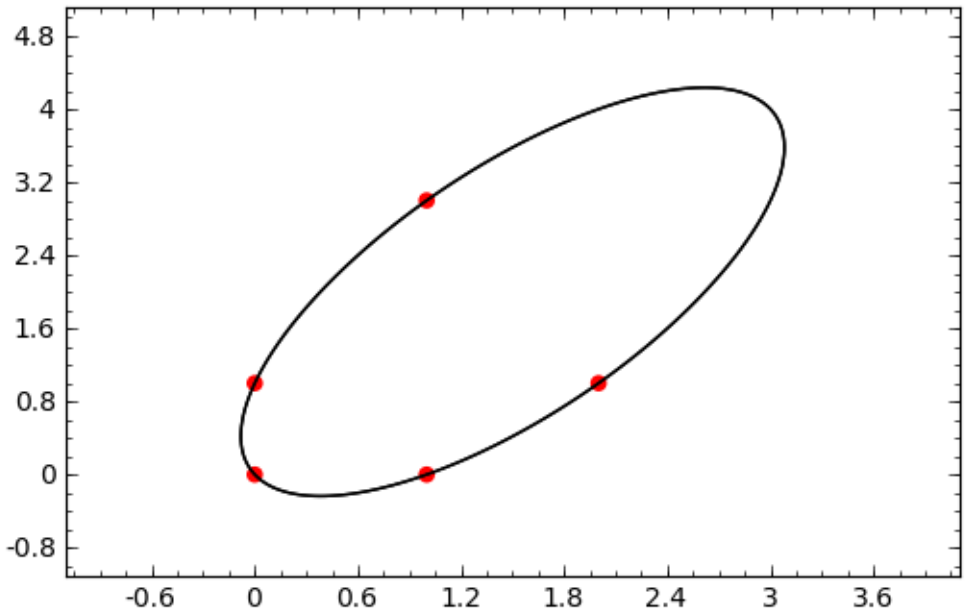
```
sage: sol = coefs.solve_right(vector([0,0,0,0,0]))
sage: show(sol)
sage: coefs.right_kernel().basis()[0]
(1, 1/2, -1, -1, -1/2, 0)
```

$$(0, 0, 0, 0, 0, 0)$$

```
sage: vk = coefs.right_kernel().basis()[0]
sage: a,b,c,d,e,f = vk
sage: curva = a*x^2 + b*y^2 + c*x*y + d*x + e*y + f
sage: show(curva)
```

$$x^2 - xy + \frac{1}{2}y^2 - x - \frac{1}{2}y$$

```
sage: implicit_plot(curva,(x,-1,4),(y,-1,5))+point2d(puntos,color=(1,0,0),pointsize=30)
```



Ejercicio

Encuentra 5 puntos tal que la cónica que pasa por ellos sea una hipérbola.

Un ejemplo de haz de curvas

Si fijamos 4 puntos del plano en posición general, encontramos muchas cónicas que pasan por todos los puntos. En este caso, los coeficientes de la ecuación de la cónica que pasa por los 4 puntos deben satisfacer 4 ecuaciones:

$$\begin{aligned} x_1^2 a + y_1^2 b + x_1 y_1 c + x_1 d + y_1 e + 1 \cdot f &= 0 \\ x_2^2 a + y_2^2 b + x_2 y_2 c + x_2 d + y_2 e + 1 \cdot f &= 0 \\ x_3^2 a + y_3^2 b + x_3 y_3 c + x_3 d + y_3 e + 1 \cdot f &= 0 \\ x_4^2 a + y_4^2 b + x_4 y_4 c + x_4 d + y_4 e + 1 \cdot f &= 0 \end{aligned}$$

```
sage: puntos = [(0,0),(0,1),(1,0),(2,2)]

sage: var('x y')
sage: coefs = matrix(QQ,4,6)
sage: for j in range(4):
...       x0, y0 = puntos[j]
...       coefs[j,0] = x0^2
...       coefs[j,1] = y0^2
...       coefs[j,2] = x0*y0
...       coefs[j,3] = x0
```

```
...       coefs[j,4] = y0
...       coefs[j,5] = 1

sage: show(coefs)
```

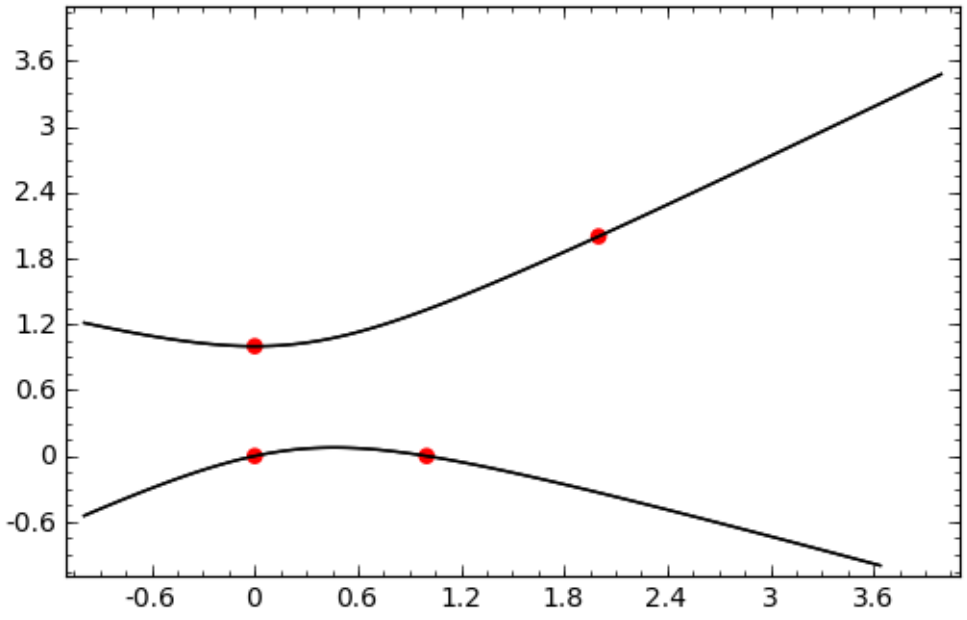
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 4 & 4 & 4 & 2 & 2 & 1 \end{pmatrix}$$

```
sage: K = coefs.right_kernel()
sage: v1 = K.basis()[0]
sage: v2 = K.basis()[1]
sage: show(K.basis())
```

$$\left[\left(1, 0, -\frac{1}{2}, -1, 0, 0\right), \left(0, 1, -\frac{1}{2}, 0, -1, 0\right)\right]$$

Cualquier vector no nulo de K da lugar a una cónica que pasa por los 4 puntos. No todas son equivalentes, aunque por supuesto dos vectores proporcionales dan lugar a la misma curva. Al conjunto de cónicas que pasa por los cuatro puntos lo llamamos el **haz de cónicas** por los 4 puntos.

```
sage: c1 = 1
sage: c2 = -3
sage: a,b,c,d,e,f = c1*v1 + c2*v2
sage: curva = a*x^2 + b*y^2 + c*x*y + d*x + e*y + f
sage: grafica = point2d(puntos,color=(1,0,0),pointsize=30) + implicit_plot(curva,(x,-1,4),(y,-1,4))
sage: grafica.show()
```



Ejercicio

Dibuja varias cónicas del haz *en la misma gráfica* .

6.8 Ejercicios

6.8.1 1.

Busca las ecuaciones (paramétricas, implícitas o las que más te convengan) de las siguientes curvas y dibújalas:

- Lemniscata de Bernouilli
- Tractriz
- Cicloide

6.8.2 2.

Compara las distintas formas de representar la misma curva usando `parametric_plot` , `polar_plot` e `implicit_plot` para comprobar que se obtiene el mismo resultado:

- Las ecuaciones implícitas de la recta $Ax + By = 1$ y la forma polar vista en clase.
- Las ecuaciones implícitas de la circunferencia $(x - x_0)^2 + (y - y_0)^2 = r^2$ que pasa por el origan y la forma polar vista en clase.
- Busca las ecuaciones implícitas de la curva de Agnesi y comprueba que corresponde a la misma curva que la parametrización vista en clase.
- Idem para el folium de Descartes.

6.8.3 3. Ecuación de un *hiperplano* en el espacio n -dimensional $\mathbb{R}^n$

Si denotamos por $x_1, x_2, \ldots, x_n$ las coordenadas cartesianas en $\mathbb{R}^n$, una ecuación implícita o cartesiana del hiperplano determinado por n puntos, $(x_{1,1}, x_{1,2}, \ldots, x_{1,n}), \ldots, (x_{n,1}, x_{n,1}, \ldots, x_{n,n})$, *en posición general* , se obtiene al desarrollar:

$$\det \begin{pmatrix} x_1 - x_{1,1} & x_1 - x_{2,1} & & x_1 - x_{n,1} \\ x_2 - x_{1,2} & x_2 - x_{2,2} & & x_2 - x_{n,2} \\ \vdots & & \ddots & \\ x_n - x_{1,n} & x_n - x_{2,n} & & x_n - x_{n,n} \end{pmatrix} = 0\,.$$

Al simplificar el desarrollo, el vector, $(A_1, A_2, \ldots, A_n)$, formado por los coeficientes en la ecuación $A_1x_1 + A_2x_2 + \cdots + A_nx_n = B$ es perpendicular al hiperplano (vector *normal* del hiperplano).

- Obtén la ecuación implícita del plano que pasa por 3 puntos de $\mathbb{R}$.

6.8.4 4.

Encuentra 5 puntos tal que la cónica que pasa por ellos sea una hipérbola, una parábola, y el producto de 2 rectas (no importa que no estén en posición general).

6.8.5 5. Explorando el comando `animate` .

El comando `animate` permite convertir una lista de gráficas en una animación cuyos fotogramas son las gráficas de la lista.

- Investiga la ayuda de animate, y encuentra el código que muestra la gráfica del seno desplazándose a lo largo del eje x.
- Crea una animación de un punto moviéndose desde un punto A a otro punto B.
- Crea una animación de un punto moviéndose a lo largo de una circunferencia de centro (0,0) y radio 1.

6.8.6 6.

Combina lo aprendido sobre el comando `animate` con lo visto en clase de teoría para crear una animación del haz de cónicas que pasa por 4 puntos.

6.8.7 7.

Dada una curva $\gamma : I \longrightarrow \mathbb{R}^2$ dada por $\gamma(t) = (x(t), y(t))$, el vector tangente a la curva en $\gamma(t)$ es el vector con origen en $\gamma(t)$ y dirección $\gamma'(t) = (x'(t), y'(t))$. Investiga el método `arrow` y dibuja el vector tangente en varias de las curvas definidas en coordenadas paramétricas en la clase de teoría.

6.8.8 8.

Combina lo aprendido sobre el comando `animate` con el ejercicio anterior para hacer animaciones donde la curva está fija y el vector tangente se mueve a lo largo de la curva.

Bloque VI: Miscelánea

Aplicaciones variadas. La primera sesión enseña el concepto de ajuste de modelos y su materialización en Sage, y se aplica a algunos ejemplos de datos científicos y de matemáticas. La sesión de criptografía se puede seguir inmediatamente después de estudiar el bloque de aritmética, y la sesión de malabares sólo usa lo visto sobre grafos. Las dos últimas sesiones son breves y poco sistemáticas.

7.1 ¿Qué es TeX?

TeX es el *sistema de composición de textos* creado por Donald Knuth. TeX, y su extensión LaTeX, desarrollada principalmente por Leslie Lamport, son el sistema más común de escribir textos matemáticos, pues permite incluir fórmulas matemáticas de calidad profesional. El sistema es libre y multiplataforma, y es usado por muchísimos científicos de diversas disciplinas. Existen extensiones que permiten incluir gráficas, notación músical, código de SAGE, y muchas otras cosas.

Al ser un sistema tan extendido, muchos programas utilizan TeX para escribir las matemáticas, como Sage. Cuando editas un bloque de texto, puedes incluir códigos de TeX. Al guardar los cambios, los códigos de TeX se convierten en fórmulas matemáticas. Por mantener la compatibilidad con los tutoriales de TeX, diremos que el texto original, con sus códigos especiales, es el **código fuente** , y que el resultado final, con las fórmulas matemáticas, es el **texto compilado** .

Aunque aprender el sistema lleva mucho tiempo, las ideas fundamentales de TeX son sencillas

7.1.1 Modos *inline* y *display*

- Cuando queremos incluir una fórmula matemática dentro de en una línea de texto, la rodeamos con carácteres de $, como por ejemplo: `$\sin(\pi)=1$`, que se convierte en $\sin(\pi) = 1$ al **compilar** el código. Esta forma de incluir fórmulas se llama en inglés **modo inline** .
- Cuando queremos incluir una fórmula en su propia línea, la rodeamos con dos caracteres de $. Por ejemplo:

```
$$e^{\pi i}=-1$$
```

que se convierte al compilar en:

$$e^{\pi i} = -1$$

Esta forma de incluir fórmulas se llama en inglés **modo display** .

7.1.2 Estructura lógica del documento

Aparte de estas dos formas de incluir fórmulas matemáticas, TeX pone mucho enfásis en la estructura del documento, separándolo en capítulos, secciones, subsecciones... y dentro de ellas teoremas, corolarios, comentarios... Esta parte de TeX no se usa en SAGE, que usa el html para dar estructura al documento. Ambos sistemas, TeX y html, dan mucha importancia a la estructura del documento, y ofrecen al autor la posibilidad de indicar el tipo de contenido en vez de limitarse a colocarlo en la posición correcta con el tamaño de letra deseado.

7.1.3 LaTeX dentro de Sage

Lo mejor de SAGE es que podemos obtener fácilmente el código latex que muestra una expresión cualquiera usando el comando `latex` , que acepta como único argumento un objeto de SAGE, y devuelve su expresión en LaTeX.

```
sage: var('x')
sage: f=sqrt(x)

sage: print f
sage: show(f)
sage: print latex(f)
sqrt(x)
\sqrt{x}
```

$$\sqrt{x}$$

Podemos usar el código latex de arriba en nuestros cuadros de texto: $\sqrt{x}$.

Veamos más ejemplos:

```
sage: print latex(pi)
sage: print latex(1/2)
\pi
\frac{1}{2}
```

El primer ejemplo muestra la forma de introducir las letras griegas: `$\alpha$`, `$\beta$`, `$\gamma$` se convierten en: α, β, γ.

El segundo ejemplo muestra la forma de escribir fracciones \frac{ *numerador* }{ *denominador* }.

Por supuesto, todas los códigos se pueden combinar de cualquier forma:

```
sage: a=1/sqrt(2*pi)
sage: print a
sage: show(a)
sage: print latex(a)
1/2*sqrt(2)/sqrt(pi)
\frac{1}{2} \, \frac{\sqrt{2}}{\sqrt{\pi}}
```

$$\frac{1}{2}\,\frac{\sqrt{2}}{\sqrt{\pi}}$$

La lista completa de símbolos es enorme, por supuesto, e incluye flechas (`\rightarrow`: $\rightarrow$), operadores (`\div`: $\div$), desigualdades (`\neq`: $\neq$) ...

Búsquedas en google arrojan listas bastante completas:

http://omega.albany.edu:8008/Symbols.html

http://stdout.org/~winston/latex/

El siguiente ejemplo usa polinomios.

```
sage: R1.<t>=PolynomialRing(QQ)
sage: p=t^3-1
sage: print latex(p)
sage: print latex(p/(p+2))
t^{3} - 1
\frac{t^{3} - 1}{t^{3} + 1}
```

El único comando nuevo es el ^{ *superíndice* }, para exponentes. El comando equivalente para subíndices es _{ *subíndice* }. Por ejemplo:

```
$$x_{1}^{2}+x_{2}^{2}+\dots+x_{n}^{2} $$
```

se convierte en:

$$x_1^2 + x_2^2 + \cdots + x_n^2$$

En modo display, y para algunos operadores como el sumatorio (sum) o el límite (limit), los subíndices se muestran debajo del operador:

```
$$\sum_{j=1}^{N}x_i$$
```

se convierte en :

$$\sum_{j=1}^{N} x_i$$

Probamos una función compuesta:

```
sage: f = sin(x^2)
sage: show(f)
sage: print latex(f)
\sin\left(x^{2}\right)
```

$$\sin\left(x^2\right)$$

El código de esta función incluye dos nuevos comandos: `\left` y `\right`. Estos comandos sirven para cuadrar el paréntesis izquierdo con el derecho, y asegurarse de que son lo bastante grandes para delimitar el contenido.

Siempre que aparece un comando left, debe aparecer después un comando right.

En este caso, después de cada uno hemos usado parétensis, pero podemos poner corchetes o barras verticales:

```
$$\left[(a+b)^{2}+(a-b)^{2}\right]$$
```

que se convierte en:

$$\left[(a+b)^2 + (a-b)^2\right]$$

```
sage: f = sin(1/x)
sage: show(f)
sage: print latex(f)
\sin\left(\frac{1}{x}\right)
```

$$\sin\left(\frac{1}{x}\right)$$

7.1.4 Matrices

Veamos ahora cómo se muestran matrices:

```
sage: A=matrix(QQ,[[1,2,3],[4,5,6]])
sage: print A
[1 2 3]
[4 5 6]

sage: show(A)
```

$$\left(\begin{array}{rrr} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array}\right)$$

```
sage: print latex(A)
\left(\begin{array}{rrr}
1 & 2 & 3 \\
4 & 5 & 6
\end{array}\right)
```

El contenido entre paréntesis es:

```
\begin{array}{rrr}
1 & 2 & 3 \\
4 & 5 & 6
\end{array}
```

El código `{rrr}` indica que la matriz tiene tres columnas, todas alineadas a la derecha (r:right, l:left, c:center). Los caracteres `&` delimitan las columnas, y los caracteres `\\` delimitan las filas.

7.1.5 Ejercicios

1.

copia, pega y modifica el código anterior, para escribir una matriz 2x4 y otra 4x2.

2.

Intenta escribir el código que genera el determinante de abajo (no vale mirar):

$$d = \begin{vmatrix} 1 & \alpha_1 & \alpha_1^2 \\ 1 & \alpha_2 & \alpha_2^2 \\ 1 & \alpha_3 & \alpha_3^2 \end{vmatrix}$$

3.

Las fórmulas de la Wikipedia también están escritas en LaTeX. Puedes obtener el código LaTeX que genera una fórmula haciendo click derecho sobre la fórmula y eligiendo "propiedades", y copiando el "texto alternativo".

- Copia en esta hoja la fórmula del determinante 3x3 y **n** x **n** de la página:

http://en.wikipedia.org/wiki/Determinant

4.

A veces necesitamos sólo uno de los dos paréntesis, corchetes o llaves. Incluso en este caso, escribimos un operador `\left` y otro `\right` para delimitar la región, sólo que donde no queremos que ponga un paréntesis, escribimos un punto. Ejemplo:

```
$$\alpha\left| \frac{1}{2}\right. + \omega\left| \frac{1}{2}\right.$$
```

$$\alpha\left|\frac{1}{2}\right. + \omega\left|\frac{1}{2}\right.$$

Intenta escribir el código que genera la definición de abajo (no vale mirar):

$$f(x) = \begin{cases} x^2 & \text{si } x < 0 \\ x^3 & \text{si } x \geq 0 \end{cases}$$

7.2 Probabilidad en Sage

En esta sesión vamos a intentar representar distribuciones de probabilidad discretas y continuas y realizar con ellas varias operaciones comunes, como calcular medias y varianzas, hacer extracciones aleatorias según una distribución dada o dibujar las funciones de masa, densidad y distribución. Al final, trabajaremos un poco con variables aleatorias bidimensionales.

7.2.1 Distribucion discreta con soporte finito

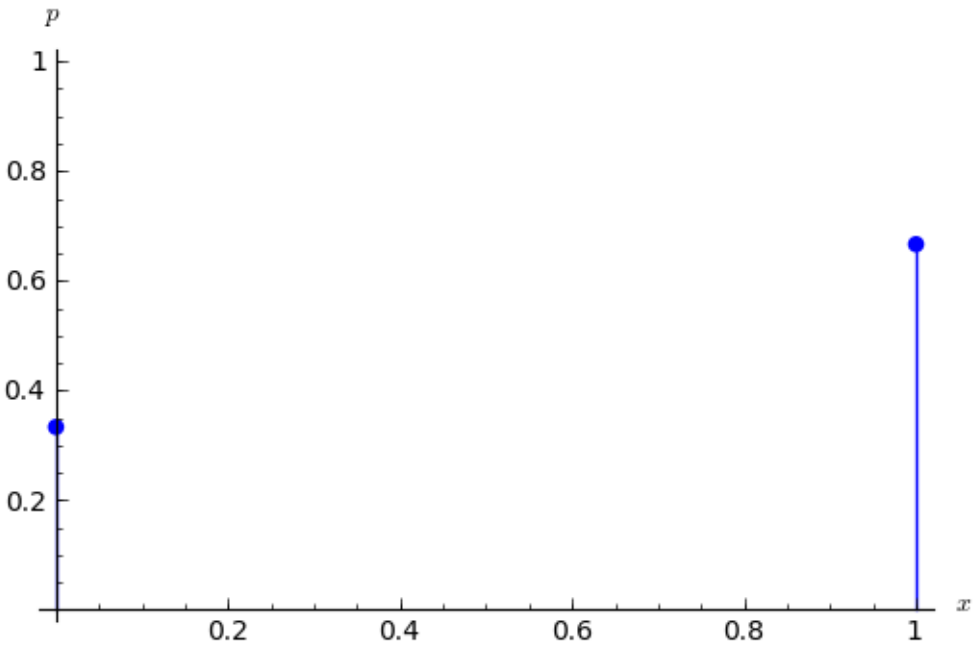
Representamos la función de masa mediante un diccionario en el que las claves son los puntos del espacio muestral y el valor asociado a cada clave es la probabilidad de ese punto. Un diccionario representa una distribución de probabilidad si sus valores son números (reales, racionales, incluso expresiones simbólicas) que suman 1.

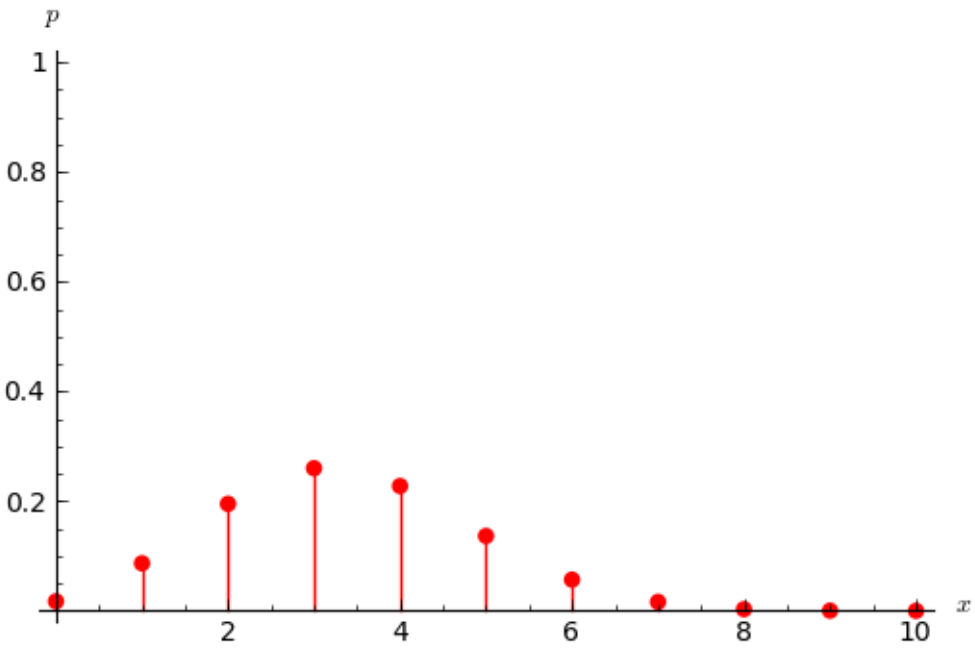
```
sage: #Ejemplos:
...
sage: #Bernouilli de prob p=1/3
sage: p = 1/3
sage: f_bernouilli = {0:p, 1:1-p}
sage: #Binomial con prob p=1/3 y k=10 ensayos independientes
sage: k = 10
sage: p = 1/3
sage: f_binomial = dict((j, p^j*(1-p)^(k-j)*binomial(k,j)) for j in range(k+1))
```

Asumiendo que el espacio muestral está contenido en $\mathbf{R}$, podemos dibujar la distribución por ejemplo así:

```
sage: #dibujar una distribucion discreta con soporte finito
sage: def dibuja_f(f, *args, **kargs):
...       '''Dibuja una funcion de masa con soporte finito, dada como diccionario
...
...       Acepta los argumentos adicionales tipicos de graficas en Sage,
...       como color, etc
...       '''
...       p = (sum([line2d([(x, 0), (x, f[x])], *args, **kargs) for x in f])
...            + point2d(f.items(), pointsize=30, *args, **kargs))
...
...       #Imponemos rango [0,1] para el eje que muestra las probabilidades
...       p.ymin(0)
```

```
...       p.ymax(1)
...       p.axes_labels(['$x$','$p$'])
...       return p

sage: show(dibuja_f(f_bernouilli))
sage: show(dibuja_f(f_binomial, color = (1,0,0)))
```





De nuevo asumiendo que el espacio muestral está contenido en $\mathbf{R}$, calculamos la esperanza y la varianza de una

variable aleatoria con función de masa f usando las fórmulas habituales:

$$\mu = E[X] = \sum_{i=1}^{N} x_i \, f(x_i)$$

$$\sigma^2 = Var[X] = \sum_{i=1}^{N} (x_i - \mu)^2 \, f(x_i)$$

```
sage: #media y varianza
sage: def media_f(f):
...       return sum(x*f[x] for x in f)
...
sage: def varianza_f(f):
...       m = media_f(f)
...       return sum((x-m)^2*f[x] for x in f)

sage: print media_f(f_bernouilli), varianza_f(f_bernouilli)
sage: print media_f(f_binomial), varianza_f(f_binomial)
2/3 2/9
10/3 20/9
```

Trabajar con parámetros

Como el código anterior es genérico, nada nos impide usar variables simbólicas, y hacer cálculos con parámetros libres.

```
sage: var('p')
sage: #Bernouilli
sage: #funcion de masa
sage: f = {0:1-p, 1:p}
sage: media_f(f), varianza_f(f)
(p, (p - 1)^2*p - (p - 1)*p^2)

sage: varianza_f(f).factor()
-(p - 1)*p
```

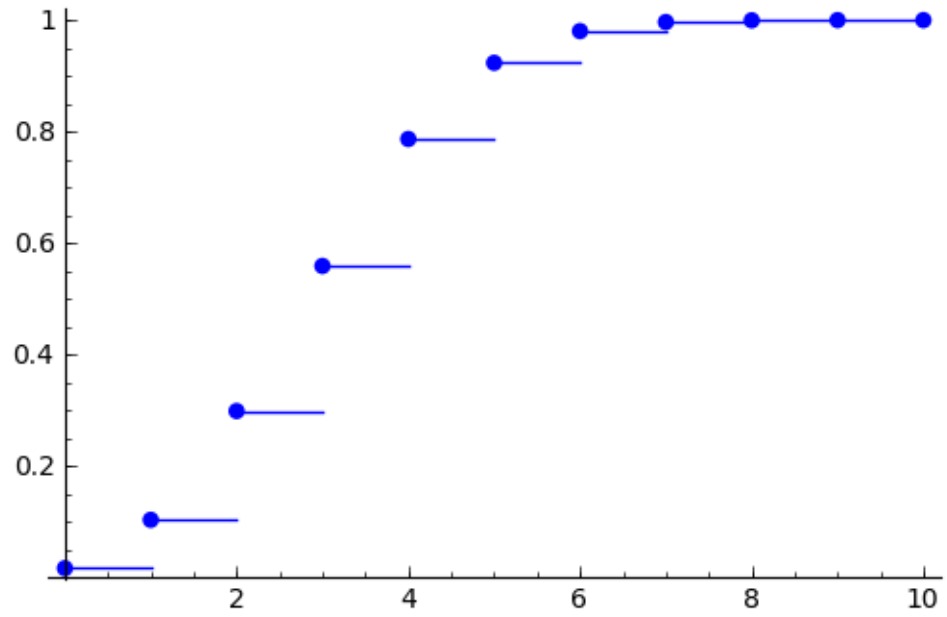
Función de distribución

Para trabajar con la función de distribución, necesitamos ordenar los puntos del espacio muestral. Guardamos en una lista los puntos que tienen probabilidad positiva y en otra lista (del mismo tamaño) la prob de cada punto.

```
sage: pares = f_binomial.items()
sage: pares.sort()
sage: valores   = [x for x,p in pares]
sage: probs     = [p for x,p in pares]
sage: cum_probs = []
sage: suma = 0
sage: for p in probs:
...       suma += p
...       cum_probs.append(suma)
sage: print valores
sage: print probs
sage: print cum_probs
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1024/59049, 5120/59049, 1280/6561, 5120/19683, 4480/19683, 896/6561, 1120/19683, 320/19683, 20/6561,
[1024/59049, 2048/19683, 5888/19683, 11008/19683, 15488/19683, 18176/19683, 2144/2187, 19616/19683, 1
```

```
sage: #dibuja la Funcion de distribucion
sage: (point(zip(valores, cum_probs), pointsize=30) +
...    sum(line([(valores[j], cum_probs[j]), (valores[j+1], cum_probs[j])])
...        for j in range(len(valores)-1)))
```



Ejercicio - debate : ¿Cómo podemos extraer un número en el soporte de nuestra función de masa respetando las probabilidades requeridas? Es decir, si tenemos:

```
f = {0:1/2, 1:1/3, 2:1/6}
valores   : [0, 1, 2]
cum_probs : [1/2, 5/6, 1]
```

queremos una función que devuelva 0 con probabilidad 1/2, 1 con probabilidad 1/3, y 2 con probabilidad 1/6.

7.2.2 Distribucion discreta con soporte infinito (ej, z>=0)

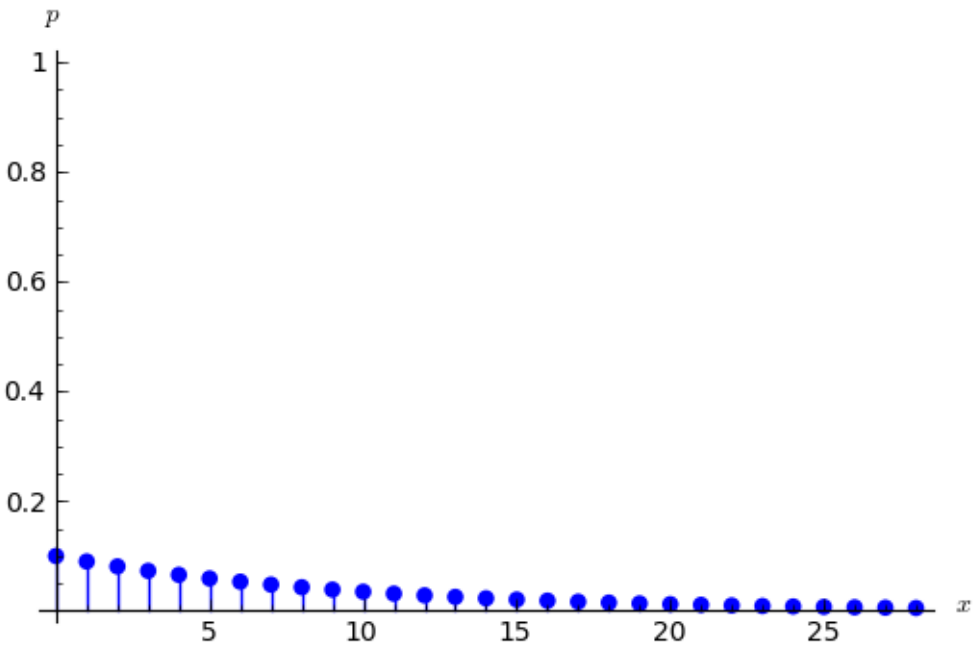
Un diccionario no puede contener una cantidad infinita de valores. Para trabajar con distribuciones con soporte infinito podemos usar funciones de python, o expresiones simbólicas. Optamos por la segunda opción para tener al menos la posibilidad de hacer algunos cálculos de forma exacta, aunque no siempre sea posible.

```
sage: #Geometrica
sage: var('k')
sage: p = 0.1
sage: f_geometrica = (1-p)^k*p
sage: #Probabilidad de que X<=5
sage: print sum(f_geometrica, k, 0, 5).n()
sage: #Poisson de parametro landa = 2
sage: landa = 2
sage: f_poisson = e^(-landa)*landa^k/factorial(k)
sage: #Probabilidad de que X>=3
sage: print sum(f_poisson, k, 3, oo).n()
```

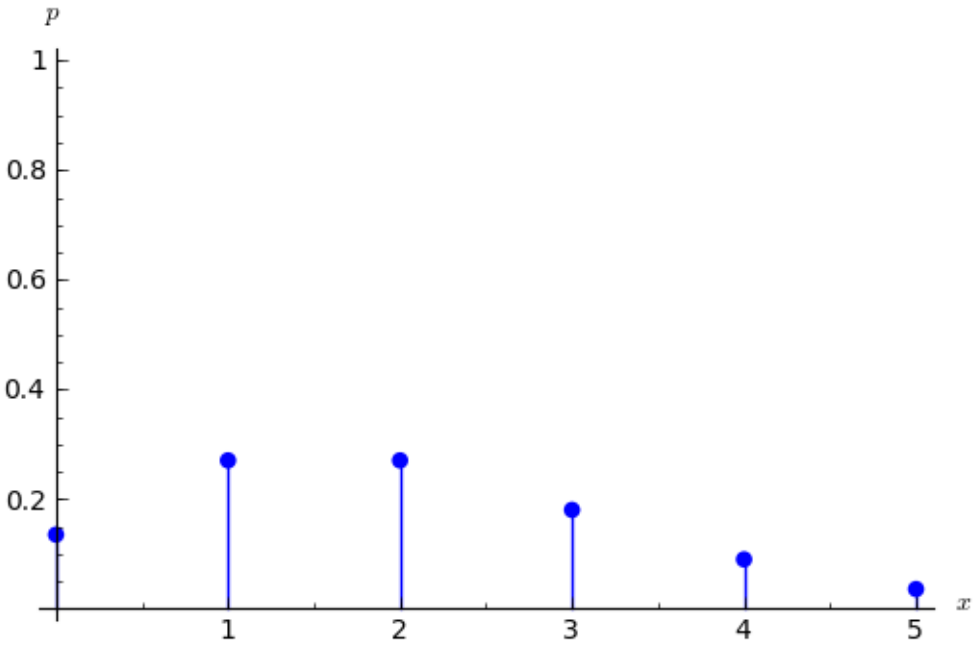
```
0.468559000000000
0.323323583816937

sage: #media y varianza
sage: def media_d(f):
...       k = f.variables()[0]
...       return sum(f*k, k, 0, oo)
...
sage: def varianza_d(f):
...       m = media_d(f)
...       k = f.variables()[0]
...       return sum(f*(k-m)^2, k, 0, oo)

sage: media_d(f_geometrica), varianza_d(f_geometrica)
(9.0, 90.0)

sage: #Alerta BUG: maxima calcula mal la varianza de f_poisson:
sage: #Update 28-04-11: Este bug ha sido corregido en maxima,
sage: #pero la corrección aún tardará un tiempo en llegar a Sage
sage: media_d(f_poisson), varianza_d(f_poisson).n()
(2, 0.812011699419676)

sage: #Sumando unos cuantos terminos tenemos el resultado correcto
sage: sum([f_poisson(k=j)*(j-landa)^2 for j in range(20)]).n()
1.99999999997887

sage: #Sumando por separado tb tenemos el resultado correcto
sage: (sum((e^(-landa)*landa^k/factorial(k))*k^2, k, 0, oo) -
...    sum((e^(-landa)*landa^k/factorial(k))*k, k, 0, oo)^2   )
2

sage: #Incluso es capaz de hacerlo con una variable simbolica
sage: #un bug como la copa de un pino!
sage: var('landa')
sage: ( e^(-landa)*sum((landa^k/factorial(k))*(k-landa)^2, k, 0, oo)  )
landa
```

Para dibujar una distribucion discreta con soporte finito, nos conformamos con mostrar unos cuantos puntos que concentran la mayoría de la masa:

```
sage: def aproxima_df(f, porcentaje_masa = 0.95):
...       '''Aproxima una distribucion de probabilidad discreta dada por una
...       expresion simbolica por una funcion de masa con soporte finito
...       '''
...       d = {}
...       masa_total = 0
...       j = 0
...       while masa_total < porcentaje_masa:
...           d[j] = f(k = j)
...           masa_total += f(k = j)
...           j += 1
...       return d
sage: def dibuja_d(f, porcentaje_masa = 0.95, *args, **kargs):
...       d = aproxima_df(f, porcentaje_masa)
...       return dibuja_f(d, *args, **kargs)

sage: dibuja_d(f_geometrica)
```



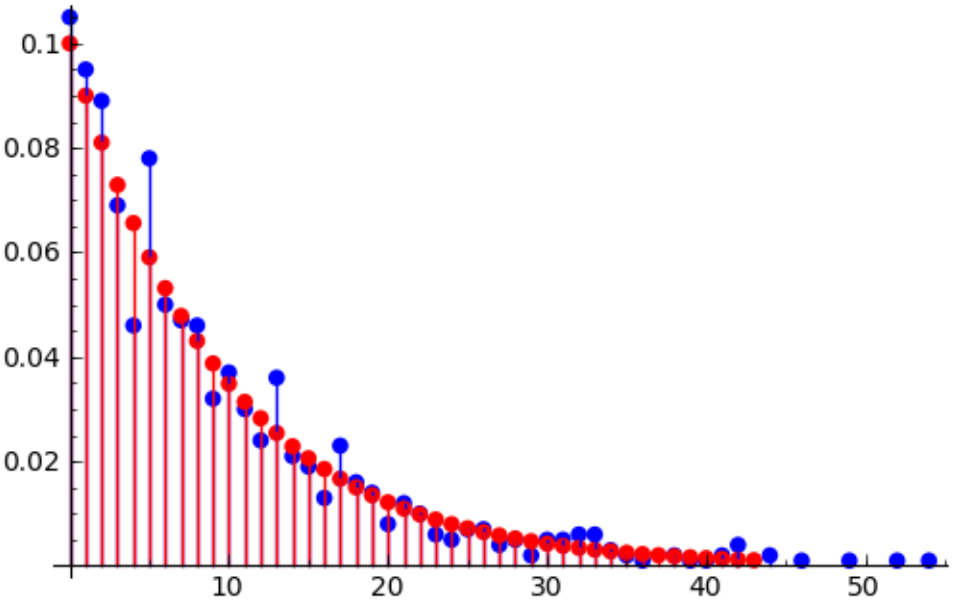

```
sage: dibuja_d(f_poisson)
```



Para extraer un entero con una distribución de probabilidad prescrita, generamos un número aleatorio t entre 0 y 1, y tomamos el menor k tal que la probabilidad acumulada P(X<=k) es mayor que t.

```
sage: #extraccion aleatoria
sage: def extraccion_aleatoria_d(f):
...       t = random()
...       j = 0
...       prob = f(k=j)
...       while prob < t:
```

```
...             j    += 1
...             prob += f(k=j)
...         return j

sage: extraccion_aleatoria_d(f_geometrica)
1

sage: from collections import defaultdict
sage: T = 1000
sage: frecuencias = defaultdict(int)
sage: for j in range(T):
...       k = extraccion_aleatoria_d(f_geometrica)
...       frecuencias[k] += 1/T
sage: dibuja_f(frecuencias) + dibuja_d(f_geometrica, 0.99, color=(1,0,0))
```

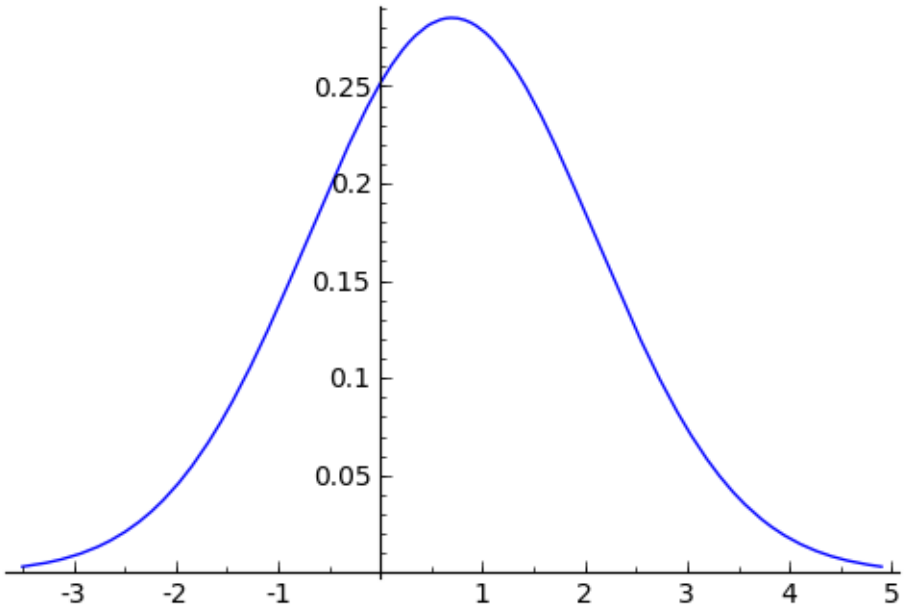


7.2.3 Distribucion continua

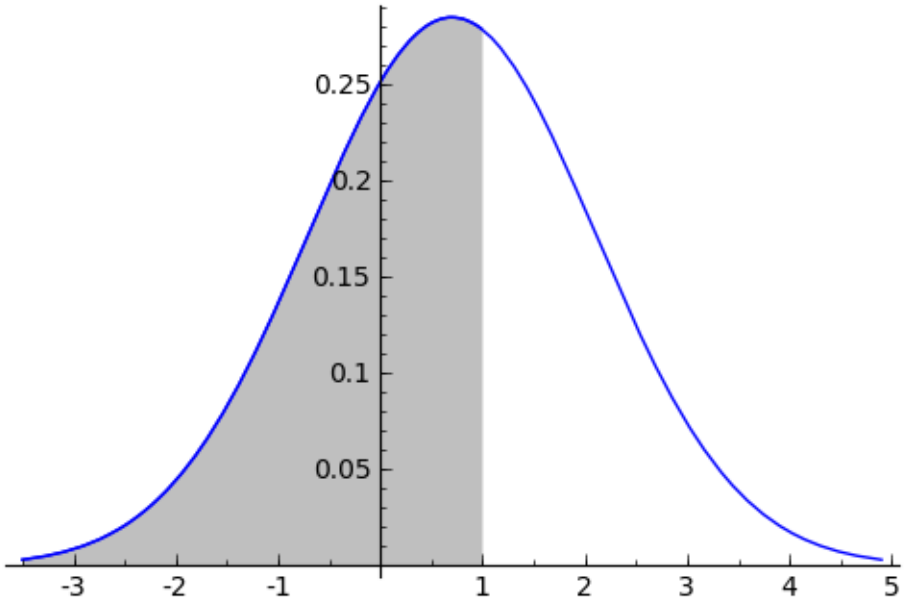
Las distribuciones continuas se pueden manejar de forma similar a las discretas con soporte infinito, pero cambiando sumas por integrales. Por ejemplo, la normal en una variable:

$$f(x) = \frac{1}{\sqrt{2\,\pi}\,\sigma} e^{-\frac{1}{2}\left(\frac{\mu - x}{\sigma}\right)^2}$$

```
sage: #Distribucion continua
sage: #Normal
sage: #funcion de densidad
sage: var('x')
sage: m = 0.7
sage: s = 1.4
sage: f_normal = (1/sqrt(2*pi*s^2))*e^(-(x - m)^2/(2*s^2))
sage: #Un tipico dibujo de la normal, centrado en la media y con 3
sage: #desviaciones tipicas de rango
sage: show(plot(f_normal, x, m - 3*s, m + 3*s))
```




```
sage: #Probabilidad de que X<1, cuando X~N(0.7, 1.4)
sage: prob, error = numerical_integral(f_normal, -oo, 1)
sage: print prob
sage: #Marcamos en el dibujo la prob pedida
sage: show(plot(f_normal, x, m - 3*s, m + 3*s) +
...        plot(f_normal, x, m - 3*s, 1, fill = True))
0.584837871172
```




```
sage: #media y varianza
sage: def media_c(f):
...       return integral(x*f,x,-oo,oo)
...
sage: def varianza_c(f):
...       m = media_c(f)
```

```
...       return integral((x-m)^2*f,x,-oo,oo)

sage: media_c(f_normal), varianza_c(f_normal)
(0.7, 1.96)
```

De nuevo, podemos usar variables simbólicas como parámetros.

```
sage: var('x mu sigma')
sage: assume(sigma > 0)
sage: f_normal = (1/sqrt(2*pi*sigma^2))*e^(-(x - mu)^2/(2*sigma^2))
sage: media_c(f_normal), varianza_c(f_normal)
(mu, sigma^2)
```

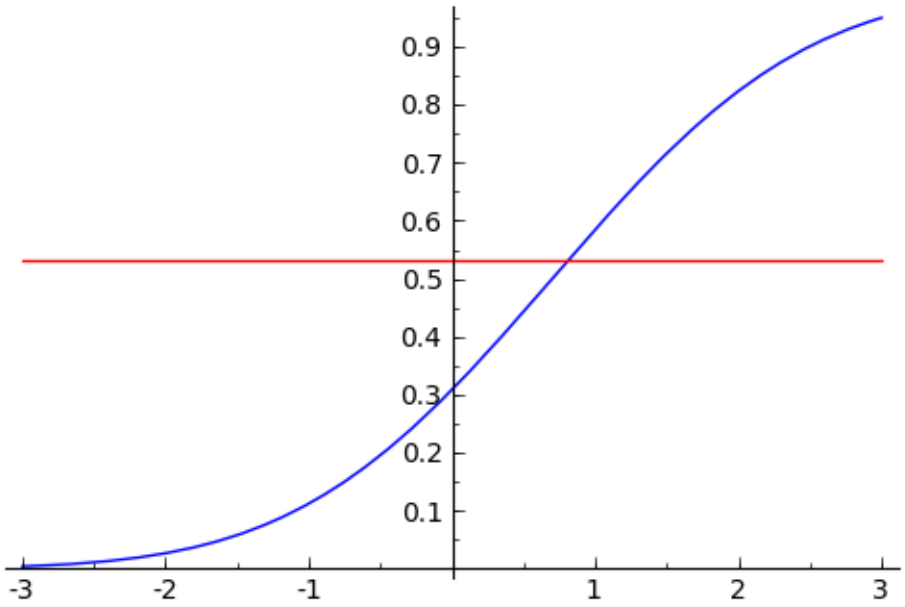
Extracciones aleatorias

Para hacer extracciones aleatorias de una distribución continua no podemos seguir un procedimiento tan naive como hasta ahora. Tenemos que transformar nuestro número aleatorio, elegido de forma uniforme entre 0 y 1, en un número real (potencialmente entre $-\infty$ e ∞) que siga una distribución dada X. Por si no lo habéis visto en clase de probabilidad, repasamos el procedimiento habitual brevemente:

- Queremos generar números x de tal modo que, para cualquier conjunto $A \subset \mathbf{R}$, la probabilidad de devolver un número $x \in A$ es exactamente $P(X \in A)$.
- Comenzamos por elegir un número aleatorio $t \in [0,1]$ (es decir, según una distribución uniforme), pero devolvemos el número G(t), para una cierta función G que tenemos que determinar.
- Para cualquier conjunto $A \subset \mathbf{R}$, queremos que $\{t \in [0,1] : G(t) \in A\} = G^{-1}(A)$ tenga medida $P(X \in A)$. De este modo, la probabilidad de devolver un número $x = G(t) \in A$ es exactamente $P(t \in G^{-1}(A)) = |G^{-1}(A)| = P(X \in A)$.
- La inversa de la función de distribución $G = F^{-1}$ cumple exactamente esta propiedad. Lo comprobamos sólo para intervalos. Si A=[x,y]:

$$P(X \in [x,y]) = F(y) - F(x) = P(U \in [F(x), F(y)]) = P(U \in F([x,y])) = P(U \in G^{-1}([x,y]))$$

```
sage: #Extraccion aleatoria
sage: m = 0.7
sage: s = 1.4
sage: f_normal = (1/sqrt(2*pi*s^2))*e^(-(x - m)^2/(2*s^2))
sage: #1: extraemos un numero aleatorio entre 0 y 1
sage: t = random()
sage: #2: funcion de distribucion
sage: var('x1')
sage: F_normal = integral(f_normal(x=x1), x1, -oo, x)
sage: show(plot(F_normal,x,-3,3) +
...       plot(0*x+t, x, -3, 3, color=(1,0,0)))
sage: #3: "invertimos" la funcion de distribucion (de forma numerica)
sage: print t, find_root(F_normal - t, m-10*s, m+10*s)
0.531001627926 0.808903109475
```

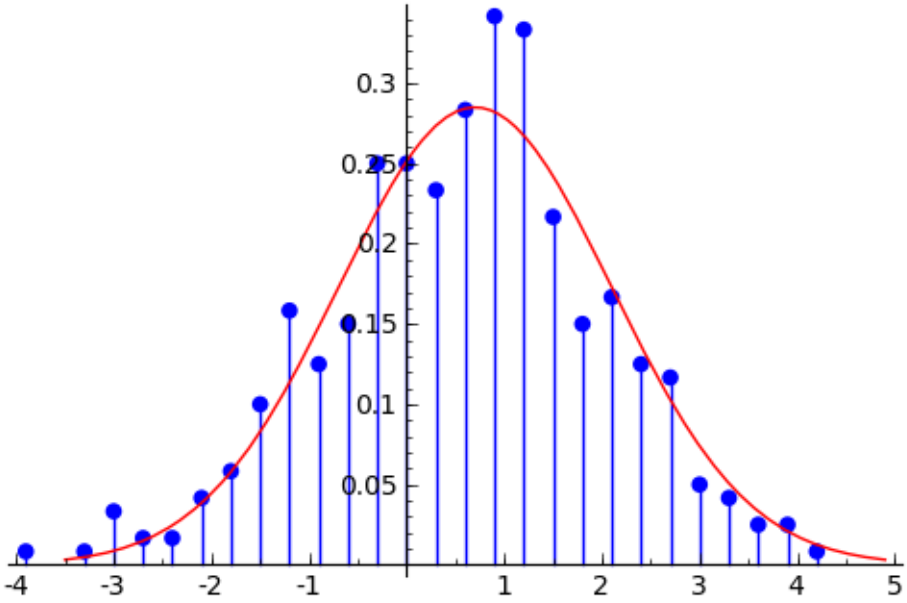



```
sage: #Intentar invertir la funcion de forma simbolica no funciona
sage: #con una normal (puede funcionar en otros casos)
sage: var('p')
sage: solve(F_normal==p, x)
[erf(5/14*sqrt(2)*x - 1/4*sqrt(2)) == 1/4853*(7777*sqrt(2)*p - 4853*e^(1/8))*e^(-1/8)]

sage: #extraccion aleatoria
sage: #el argumento es la funcion de distribucion, no la de densidad
sage: def extraccion_aleatoria_c(F):
...       t = random()
...       return find_root(F - t, -100, 100)

sage: extraccion_aleatoria_c(F_normal)
0.6299529113987602
```

Histograma

Para comparar una muestra aleatoria (una cantidad finita de puntos) con una distribución continua, tenemos que agrupar los datos extraídos en intervalos:

```
sage: from collections import defaultdict
sage: T = 400
sage: #Dividimos [-K,K] en N partes iguales
sage: K = 3
sage: N = 20
sage: frecuencias = defaultdict(int)
sage: for j in range(T):
...       a = extraccion_aleatoria_c(F_normal)
...       #TODO: explica las dos lineas siguientes
...       k = floor(a*N/(2*K))*(2*K/N)
...       frecuencias[k] += 1/(T*2*K/N)
sage: dibuja_f(frecuencias) + plot(f_normal, x, m-3*s, m+3*s, color=(1,0,0))
```



Por supuesto, mucha de esta funcionalidad está incluida en Sage

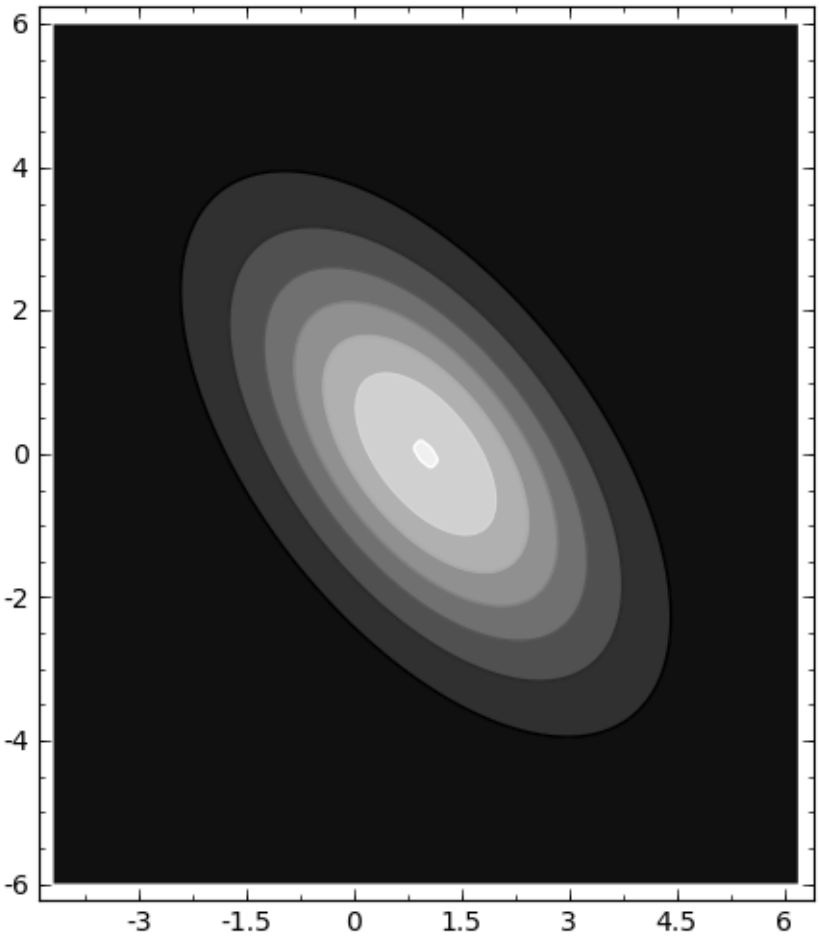
```
sage: #extracciones aleatorias de una normal
sage: normalvariate(0,1)
-1.2476798578721822
```

7.2.4 Distribución normal bidimensional

Las distribuciones en más de una dimensión se manejan de forma similar, pero con más variables simbólicas. Por ejemplo, estudiamos la distribución normal en k dimensiones:

$$f_X(x) = \frac{1}{(2\pi)^{k/2}|\Sigma|^{1/2}} \exp\Big(-\tfrac{1}{2}(x-\mu)'\Sigma^{-1}(x-\mu)\Big),$$

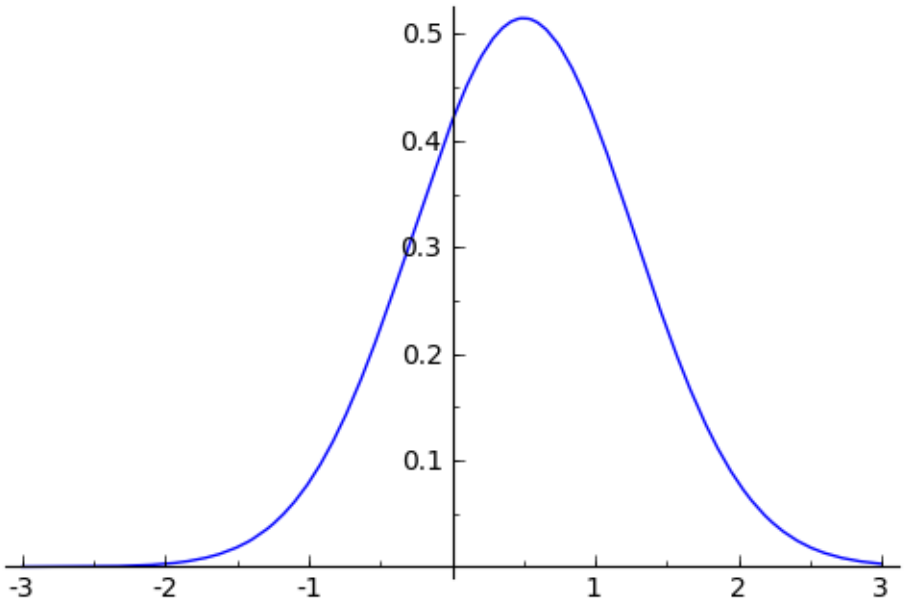
```
sage: #Normal bidimensional
sage: var('x1 x2')
sage: m1 = 1
sage: m2 = 0
sage: v1 = 3
sage: v12 = -2
sage: v2 = 4
sage: S = matrix(RDF, [[v1,v12],[v12,v2]])
sage: vs = vector([x1,x2])
sage: ms = vector([m1,m2])
sage: f = (1/(2*pi))*(1/sqrt(det(S)))*exp(-(1/2)*(vs-ms)*(~S)*(vs-ms))

sage: #plot3d(f,(x1,-3,3),(x2,-3,3)).show(viewer='tachyon')
sage: p = contour_plot(f, (x1, m1-3*sqrt(v1), m1+3*sqrt(v1)), (x2, m2-3*sqrt(v2), m2+3*sqrt(v2)))
sage: p.show(aspect_ratio=1)
```



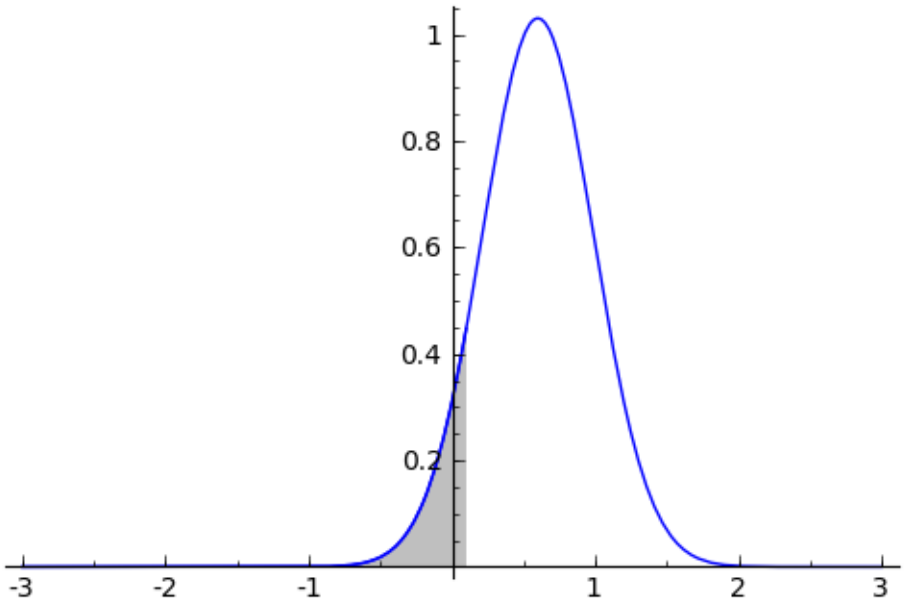
Resolvemos un típico ejercicio de probabilidad condicionada con dos variables normales: X=N(m=0.2,s=0.3) e Y=N(0.5, 0.6) son dos variables aleatorias que siguen una distribución normal, con cov(X,Y)=0.1.

Si sabemos que para un individuo (aka elemento del espacio muestral), Y=1.3, ¿cual es la prob de que X sea mayor que 0.10?

```
sage: var('x1 x2')
sage: m1 = 0.2
sage: m2 = 0.5
sage: v1  = 0.3
sage: v12 = 0.3
sage: v2  = 0.6
sage: S = matrix(RDF, [[v1,v12],[v12,v2]])
sage: vs = vector([x1,x2])
sage: ms = vector([m1,m2])
sage: f(x1,x2) = (1/(2*pi))*(1/sqrt(det(S)))*exp(-(1/2)*(vs-ms)*(~S)*(vs-ms))

sage: f_marginal_2(x2)    = integral(f,x1,-oo,oo)
sage: f_condicionada_1_dado_2(x1,x2) = f(x1,x2)/f_marginal_2(x2)

sage: plot(f_marginal_2,x2, -3, 3)
```




```
sage: (plot(f_condicionada_1_dado_2(x2=1.3),x1, -3, 3) +
...    plot(f_condicionada_1_dado_2(x2=1.3),x1, -3, .1, fill=True))
```



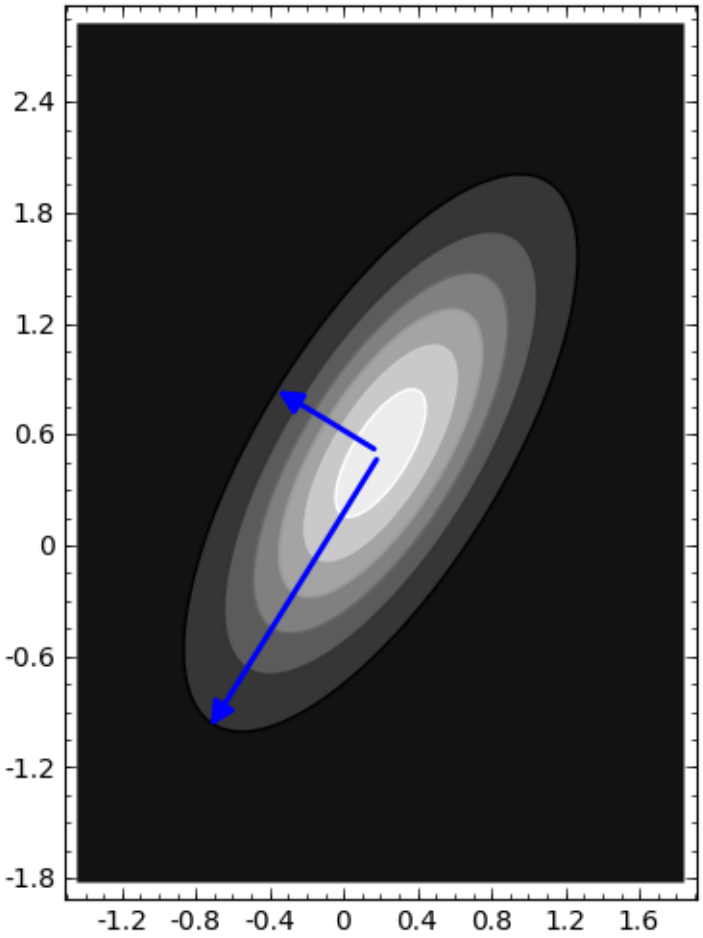

```
sage: numerical_integral(f_condicionada_1_dado_2(x2=1.3), -oo, 0.1)
(0.098352801229473263, 1.0996705400110192e-08)
```

Si diagonalizamos la matriz de varianzas-covarianzas obtenemos variables aleatorias normales e independientes.

```
sage: S.eigenvectors_left()
[(0.114589803375, [(-0.850650808352, 0.525731112119)], 1), (0.785410196625, [(-0.525731112119, -0.850
```

```
sage: [(eval1, [evec1], _ ), (eval2, [evec2], _ )] = S.eigenvectors_left()
sage: p = (contour_plot(f, (x1, m1-3*sqrt(v1), m1+3*sqrt(v1)), (x2, m2-3*sqrt(v2), m2+3*sqrt(v2)))
...        + arrow(ms, ms + 2*sqrt(abs(eval1))*evec1)
...        + arrow(ms, ms + 2*sqrt(abs(eval2))*evec2))
```

```
sage: p.show(aspect_ratio=1)
```



7.3 Ejercicios

7.3.1 1.

- Simula el siguiente experimento: lanza monedas cargadas que dan cara con probabilidad p y cruz con probabilidad 1-p hasta obtener la primera cara, y anota el número de cruces que has obtenido. Dibuja un histograma de 1000 extracciones aleatorias siguiendo este esquema, para un valor de p que tú elijas.
- Compara el resultado con el histograma de 1000 extracciones de una distribución geométrica con probabilidad p.

7.3.2 2. Función cuantil

El **cuantil** de orden p o p-cuantil, para $0 \leq p \leq 1$, de una distribución de probabilidad es el valor $F^-1(p)$, donde F es la función de distribución. La **función cuantil** es la función Q(p) que a cada p le asocia su p-cuantil. En principio, esta definición se aplica a distribuciones continuas, pero se puede extender a distribuciones de probabilidad finitas y discretas, si tomamos:

$$Q(p) = F^{-1}(p) = \text{ínf}\,\{x \in R : p \leq F(x)\}$$

Escribe funciones que devuelvan el p-cuantil de una distribución de probabilidad (en versiones finita, discreta y continua). Razona sobre la relación entre estas funciones y las extracciones aleatorias que vimos en teoría.

7.3.3 3. QQ plots

Una forma gráfica bastante común de visualizar si dos distribuciones de probabilidad son similares es el **QQ-plot** . La letra Q se refiere a los cuantiles (" *quantiles* " en inglés). Si tenemos dos distribuciones f y g:

1. Tomamos un entero k y calculamos la lista de los k-1 cuantiles de orden 1/k, 2/k, ... (k-1)/k, de f (sean $[p_1, ..., p_{k-1}]$) y la lista correspondiente con los cuantiles de g (sean $[q_1, ..., q_{k-1}]$).
2. Dibujamos para cada j entre 1 y k-1 todos los puntos (p_j, q_j).
3. Si las distribuciones son iguales, los puntos dibujados están todos en la diagonal. Si son la misma distribución, pero desplazadas una distancia d (por ejemplo, si las funciones de densidad satisfacen g(x)=f(x-d)), los puntos estarán en la recta y=x+d.

Dibuja qqplots que comparen los siguientes pares de distribuciones:

- Una binomial con n lanzamientos y p=1/2 con una normal de media np y varianza np(1-p), para varios valores de n.
- Una poisson de parámetro λ con una normal de media λ y varianza λ, para un valor pequeño de λ y otro mayor.

7.3.4 4. Medir la distancia entre distribuciones

¿Cómo podemos estimar la distancia entre dos distribuciones de probabilidad? Queremos que por ejemplo la distancia entre la distribución que toma el valor x con probabilidad 1 y la que toma el valor y con probabilidad 1 sea |x-y|, y que la distancia entre una distribución y ella misma sea 0. Ésto último descarta por ejemplo tomar el valor esperado de la distancia entre las dos variables aleatorias.

La idea del qqplot se puede llevar un poco más lejos: tomar como **distancia entre las distribuciones** el *promedio de las distancias entre los cuantiles* del mismo orden. Para mayor número de cuantiles, tenemos una medida más fina.

Estudia si esta medida verifica las condiciones de arriba y si merece que la llamemos una distancia.

7.3.5 5.

Definimos dos variables aleatorias X e Y de la forma siguiente:

- El espacio muestral son las permutaciones de los números del 1 al 10.
- La medida es la medida uniforme, que a cada permutación le da probabilidad 1/10!.
- X(w) es la imagen de 1 por la permutación w. Y(w) es la imagen de 2 por la permutación w.

Se te pide lo siguiente:

- Almacena en un diccionario la distribución conjunta de X y de Y: los pares son tuplas (x,y), y el valor asociado al par (x,y) es la probabilidad de que X valga x e Y valga y.
- Calcula la distribución marginal de X y la de Y. Confirma que obtienes el resultado esperado.
- Calcula la covarianza de X y de Y.
- Calcula la distribucion de X condicionada a que Y es multiplo de 3.

7.4 Regresión y ajuste de modelos

El análisis de regresión consiste en encontrar un **modelo** que relaciona los valores medidos de un conjunto de variables.

Los valores medidos en el mundo real nunca se ajustan de forma perfecta a un modelo, debido en primer lugar a errores de medida, pero también a que cualquier modelo matemático es una simplificación del mundo real, y si tuviera en cuenta todos los factores que influyen en un conjunto de variables, sería inmanejable.

Por tanto, no tiene sentido aspirar a encontrar un modelo que prediga exactamente los valores medidos, y debemos admitir que el modelo cometerá un cierto error.

Un modelo útil encuentra una relación funcional sencilla en conjuntos de pocas variables. Se trata de explicar una variable que tiene importancia para nosotras, en función de otro conjunto de variables mejor conocidas o más fáciles de medir. El **análisis de regresión** (más exactamente, el análisis de regresión *paramétrico*) permite encontrar un modelo explicativo en tres etapas:

1. Nuestro conocimiento del tema en cuestión nos permite escribir un modelo que afirma que la variable X es una función de las variables $Y_1, \ldots, Y_k$. La variable X recibe el nombre de **variable dependiente** y las variables $Y_1, \ldots, Y_k$ se llaman **variables independientes** . La forma exacta de la función no está fijada a priori, sino que depende de unos pocos **parámetros** libres.
2. Tomamos una **muestra** . Es decir, medimos todas las variables en un subconjunto de todos los casos posibles (unos cuantos individuos de la población, unos cuantos momentos de tiempo, una cuantas muestras preparadas en el laboratorio...)
3. **Ajustamos el modelo** , eligiendo aquellos valores de los parámetros tales que la distancia entre los valores medidos de la variable X y los valores predichos aplicando el modelo minimizan el error cometido.

7.4.1 Ejemplo de ajuste lineal

Tratamos de predecir la *temperatura* a la que hierve el agua (**T**), conocida la *presión atmosférica* (**P**) en el lugar y momento en que hacemos el experimento.

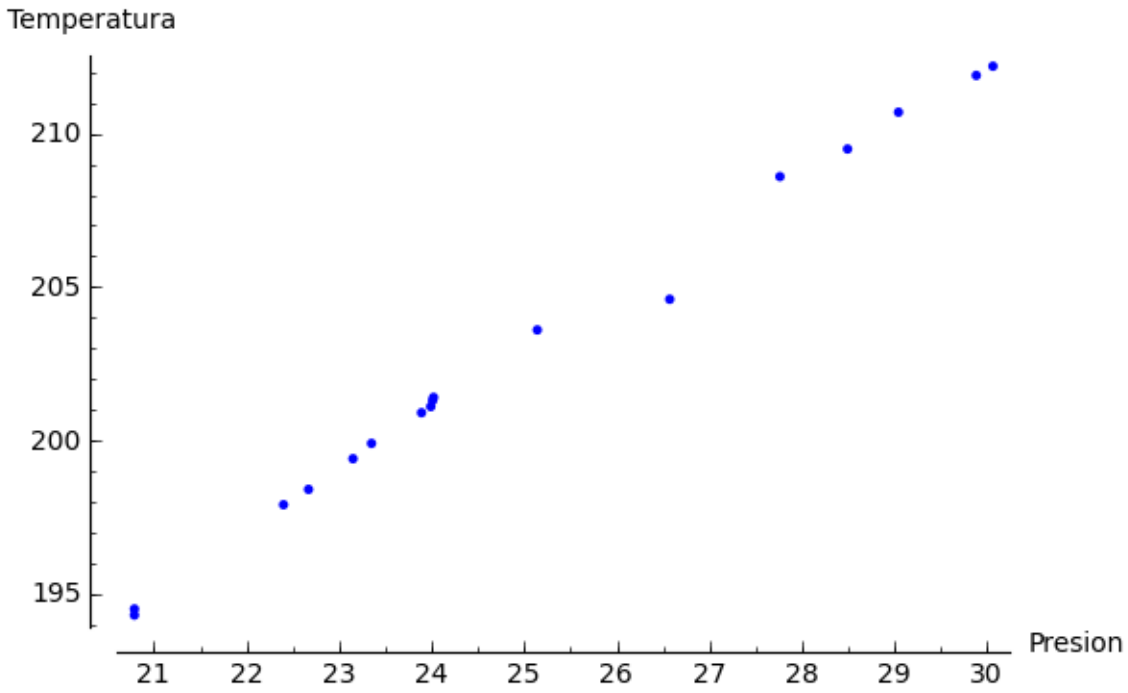
Para ello, contamos con un conjunto de mediciones de ambas variables, con la temperatura en grados Fahrenheit y la presión en pulgadas de mercurio (sea lo que sea, es una unidad de medidad de presión). Por ejemplo, en un cierto punto de los Alpes, un cierto día, el barómetro marcaba 20.79 pulgadas de mercurio, y el agua hirvió a 194.5 grados Fahrenheit. Las mediciones se realizaron en el mismo lugar geográfico, pero en días distintos, con distintas condiciones atmosféricas y quizá incluso por personas distintas. En estas condiciones, es imposible que ningún modelo prediga con exactitud el valor de T en función de P, pero esperamos que lo haga con un margen de error moderado.

```
T       P
194.5   20.79
194.3   20.79
197.9   22.4
198.4   22.67
199.4   23.15
199.9   23.35
200.9   23.89
201.1   23.99
201.4   24.02
201.3   24.01
203.6   25.14
204.6   26.57
209.5   28.49
208.6   27.76
210.7   29.04
211.9   29.88
212.2   30.06
```

Referencia: http://www.sci.usq.edu.au/staff/dunn/Datasets/Books/Hand/Hand-R/alps-R.html

Comenzamos por dibujar los datos:

```
sage: datos = [(20.79,194.50),(20.79,194.30),(22.40,197.90),(22.67,198.40),
...            (23.15,199.40),(23.35,199.90),(23.89,200.90),(23.99,201.10),
...            (24.02,201.40),(24.01,201.30),(25.14,203.60),(26.57,204.60),
...            (28.49,209.50),(27.76,208.60),(29.04,210.70),(29.88,211.90),
...            (30.06,212.20)]

sage: puntos = point(datos)

sage: puntos.show(axes_labels=('Presion','Temperatura'))
```



Los datos parecen estar dispuestos sobre una recta, de modo que intentamos un modelo lineal, de la forma:

$$T = a + b * P$$

A priori, no conocemos los valores de **a** y **b** . Para unos valores fijos de **a** y **b** , cometeremos un error en la medición j-ésima que será exactamente: $|T_j - (a + bP_j)|$. Vemos que no existe un criterio unívoco para encontrar **a** y **b** , dado que *no existe ninguna recta que pase por todos los pares* (T_j, P_j) . *Para cualquier elección de* ***a** y **b** , nuestro modelo cometerá un error al estimar alguno de los puntos medidos.

Podemos escoger los valores de **a** y **b** para los que el error máximo cometido es menor, o aquellos para los que el error medio cometido es menor, o según otros muchos criterios. Es bastante habitual en estadística buscar los valores de **a** y **b** que hacen mínimo el **error cuadrático total** :

$$E = \sum_j (T_j - (a + bP_j))^2$$

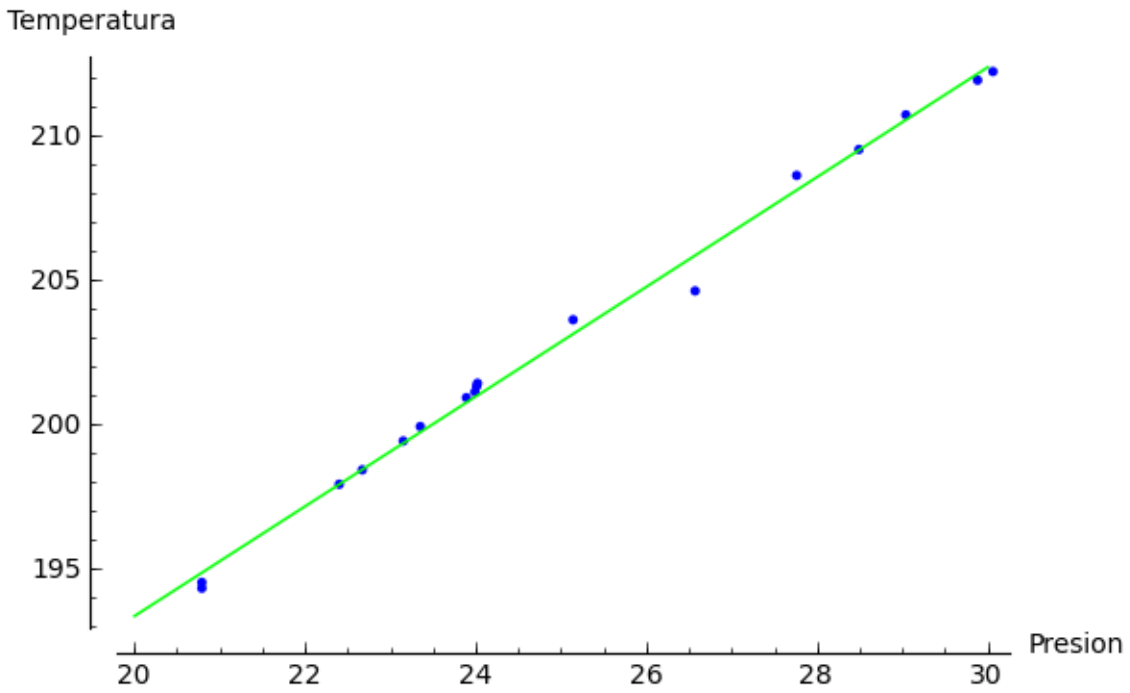
La función `find_fit` de SAGE permite ajustar un modelo cualquiera de tal forma que se minimice el error cuadrático. La función acepta dos argumentos: los datos y el modelo.

- Los datos deben ser una lista con todas las mediciones. Cada elemento de la lista es una lista o tupla con los valores de las variables en una medición. La última variable es la variable dependiente.
- El modelo es una función simbólica de varias variables que representan las variables independientes y de otras variables que representan los parámetros del modelo.

Aparte, acepta otros valores opcionales como `parameters` y `variables` para indicar a `find_fit` cuál es la variable dependiente y cuáles son los parámetros del modelo (más información en la documentación de `find_fit`).

$$T_j - (a + bP_j)$$

```
sage: var('T P a b')
(T, P, a, b)

sage: #Importante: marcamos nuestro modelo como dependiente de la variable P
sage: #El resto de variables simbólicas se toman como parametros
sage: modelo(P) = a + b*P

sage: #find_fit devuelve los valores de los parametros
sage: #que minimizan el error cuadratico
sage: #Al pasar el argumento solution_dict=True,
sage: #devuelve los valores optimos como un diccionario
sage: parametros = find_fit(datos, modelo, solution_dict=True)
sage: print parametros
{b: 1.9017835212187804, a: 155.29648352710586}

sage: #Sustituimos los valores de a y b en el modelo
sage: modelo_ajustado = modelo.subs(parametros)
sage: print modelo_ajustado
sage: #Dibujamos los puntos originales y la recta a+b*P
sage: ajuste = plot(modelo_ajustado,(P,20,30),rgbcolor=(0,1,0))
sage: grafica = puntos + ajuste
sage: grafica.show(axes_labels=('Presion','Temperatura'))
P |--> 1.9017835212187804*P + 155.29648352710586
```



Usar el modelo

Hemos establecido el modelo:

$$T = 1{,}9017835338620657 * P + 155{,}29648321028$$

Ahora podemos utilizarlo para predecir la temperatura a la que hervirá el agua a una cierta presión.Por ejemplo, esperamos que si un día la presión es de 24.5 mmHg, el agua hervirá a:

$$T = 1{,}9017835338620657 * 24{,}5 + 155{,}29648321028 = 201{,}890179789901F$$

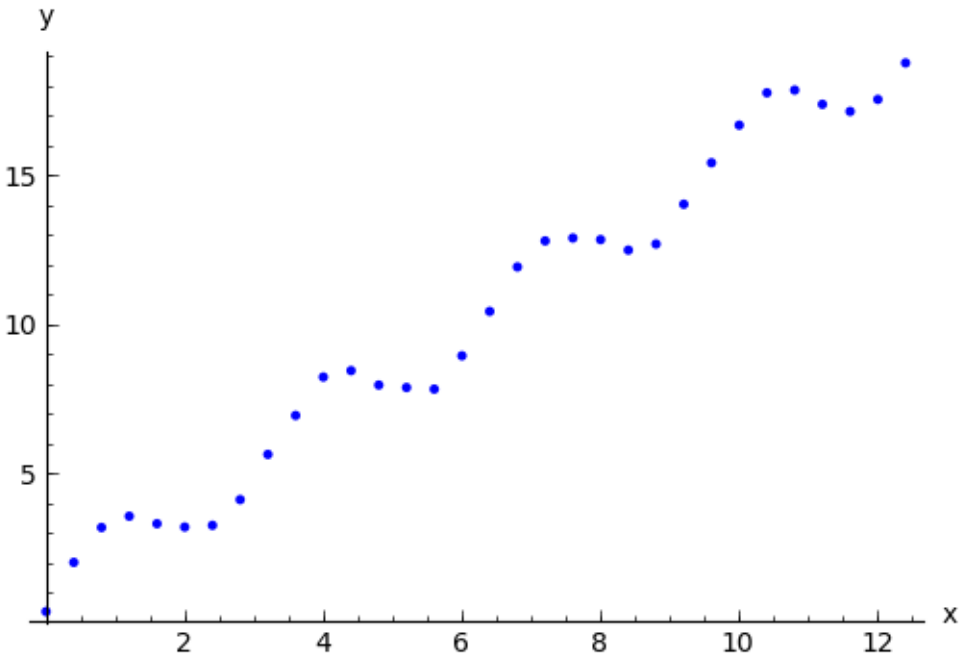
```
sage: modelo_ajustado(P=24.5)
201.890179796966
```

7.4.2 Ejemplo de ajuste no lineal

No es difícil demostrar que los parámetros que minimizan el error cuadrático en un modelo lineal son únicos. Sin embargo, cuando usamos modelos no lineales, puede haber varios mínimos de la función de error cuadrático total. En general, la minimización es más complicada, y puede que el algoritmo no consiga encontrar los valores óptimos de **a** y **b** sin ayuda. Esta vez no vamos a usar datos reales, sino datos generados usando números aleatorios. Se trata de ver qué tal ajustamos un modelo en un caso en el que conocemos cómo fueron generados los datos.

Obtenemos los datos usando la fórmula $0{,}8 + 1{,}5x + 1{,}2\sin(2x - 0{,}2)$, y sumando un número aleatorio extraído según una distribución normal de media 0 y varianza 0.1:

```
sage: #Fijamos una semilla de numeros aleatorios para asegurar
sage: #que todas obtenemos el mismo resultado
sage: set_random_seed(123)
sage: datos = [(i, 0.8+1.5*i+1.2*sin(2*i-0.2) + 0.1*normalvariate(0,1))
...           for i in srange(0, 4*pi, 0.4)]
sage: puntos = point(datos)
sage: puntos.show(axes_labels=('x','y'))
```



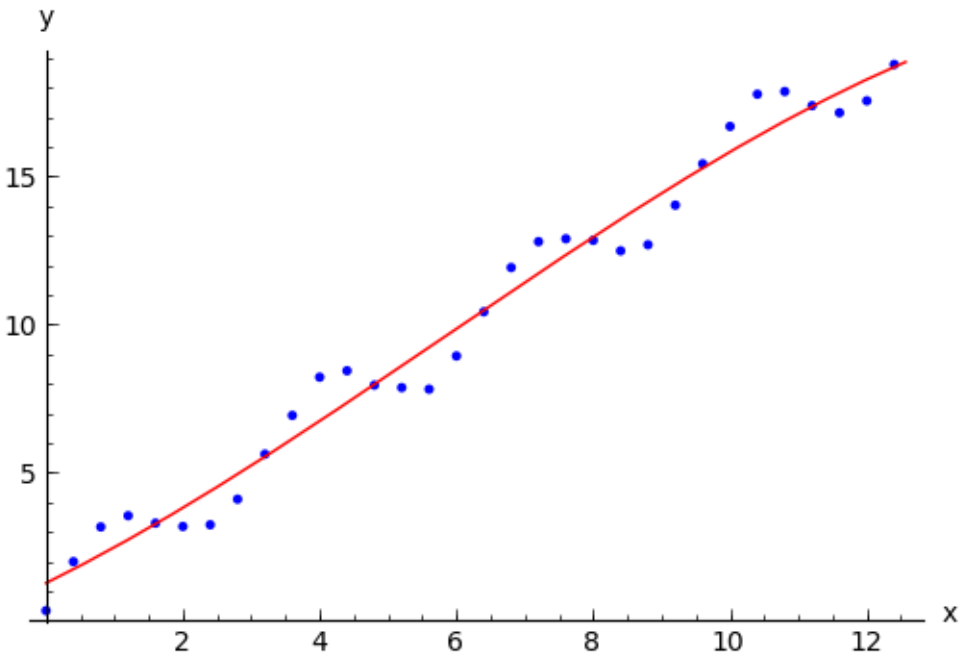
Tratamos ahora de ajustar a estos datos un modelo del tipo:

$$a_1 + a_2 x + a_3 sin(a_4 * x + a_5)$$

```
sage: var ('a1,a2,a3,a4,a5,x')
sage: modelo(x) = a1+a2*x+a3*sin(a4*x-a5)
sage: show(modelo(x))
```

$$a_2 x + a_3 \sin\left(a_4 x - a_5\right) + a_1$$

```
sage: parametros = find_fit(datos, modelo, solution_dict=True)
sage: print parametros
Warning: Number of calls to function has reached maxfev = 1200.
{a3: 97.399974374571215, a2: -4.6922665568748858, a1: 36.856991032455262, a5: -5.9092115507119871, a4

sage: modelo_ajustado = modelo.substitute(parametros)
sage: print modelo_ajustado
x |--> -4.6922665568748858*x + 97.399974374571215*sin(0.064311341301833705*x + 5.9092115507119871) +

sage: g_ajuste = plot(modelo_ajustado(x), (x,0,4*pi), color='red')
sage: grafica = puntos + g_ajuste
sage: grafica.show(axes_labels=('x','y'))
```

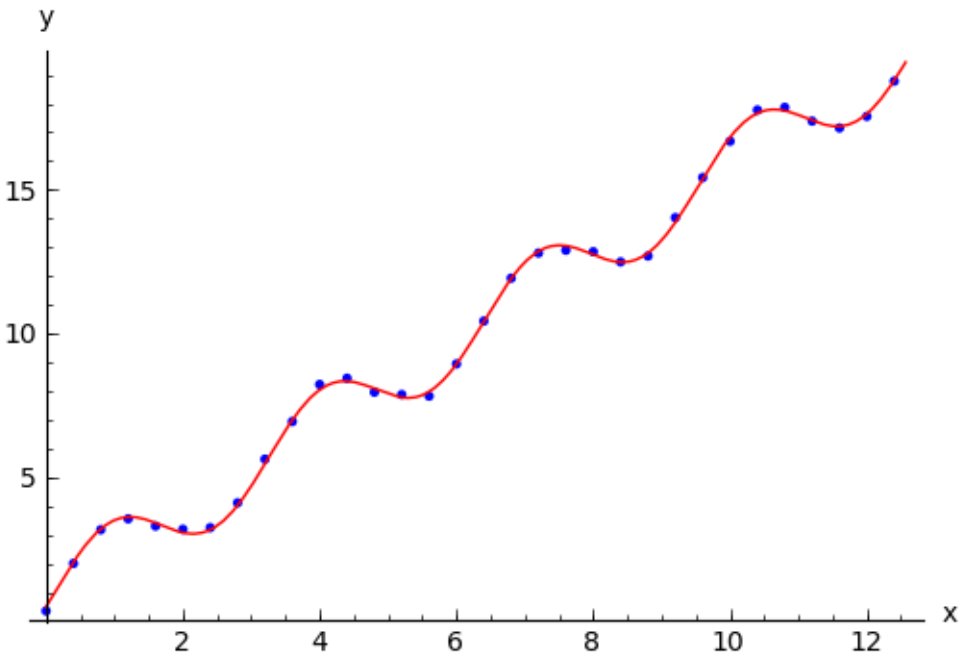


La minimización del error se ha quedado atascada por el camino. La forma más sencilla de ayudar al algoritmo es darle *un punto de partida mejor* . Para ello, pasamos a `find_fit` un argumento adicional: `initial_guess` . También le pasamos los argumentos `parameters` y `variables` para que sepa cuáles de las variable simbólicas del modelo son los parámetros.

```
sage: #Ejercicio: prueba con distintos valores iniciales
sage: #para tratar de conseguir un mejor ajuste
sage: parametros = find_fit(datos, modelo, initial_guess=[0,0,1.0,2,0],
...                  parameters=[a1,a2,a3,a4,a5], variables=[x], solution_dict=True)
sage: print parametros
{a3: 1.2403261721812324, a2: 1.5056455458784646, a1: 0.78272474105136614, a5: 0.23540896442811415, a4
```

```
sage: modelo_ajustado = modelo.substitute(parametros)
sage: print modelo_ajustado
x |--> 1.5056455458784646*x + 1.2403261721812324*sin(2.001600272362738*x - 0.23540896442811415) + 0.7
```

```
sage: g_ajuste = plot(modelo_ajustado(x), (x,0,4*pi), color='red')
sage: grafica = puntos + g_ajuste
sage: grafica.show(axes_labels=('x','y'))
```



7.4.3 Ajuste de un modelo exponencial

Vamos a resolver un ejemplo de laboratorio.

En un estudio sobre la resistencia a bajas temperaturas del bacilo de la fiebre tifoidea, se expusieron cultivos del bacilo durante diferentes periodos de tiempo a -5 grados C. Los siguientes datos representan:

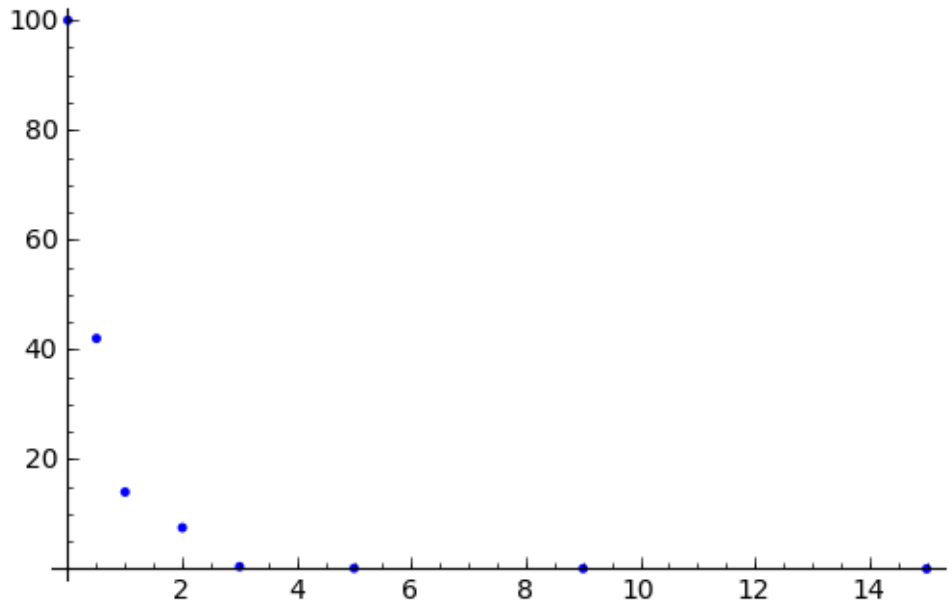
X = tiempo de exposición (en semanas).

Y = porcentaje de bacilos supervivientes.

$X:$	0	$0,5$	1	2	3	5	9	15
$Y:$	100	42	14	$7,5$	$0,4$	$0,11$	$0,05$	$0,002$

Nuestro objetivo es predecir el valor de Y conocido el valor de X.

```
sage: datos= [(0, 100), (0.5, 42), (1, 14), (2, 7.5), (3, 0.4), (5, 0.11),
...           (9, 0.05), (15, 0.002)]

sage: puntos= point(datos)
sage: show(puntos)
```



Ejercicio

Ajustar una recta ($Y = a + bX$) y una exponencial ($Y = ae^{bx}$) a los datos: ¿qué ajuste es mejor?

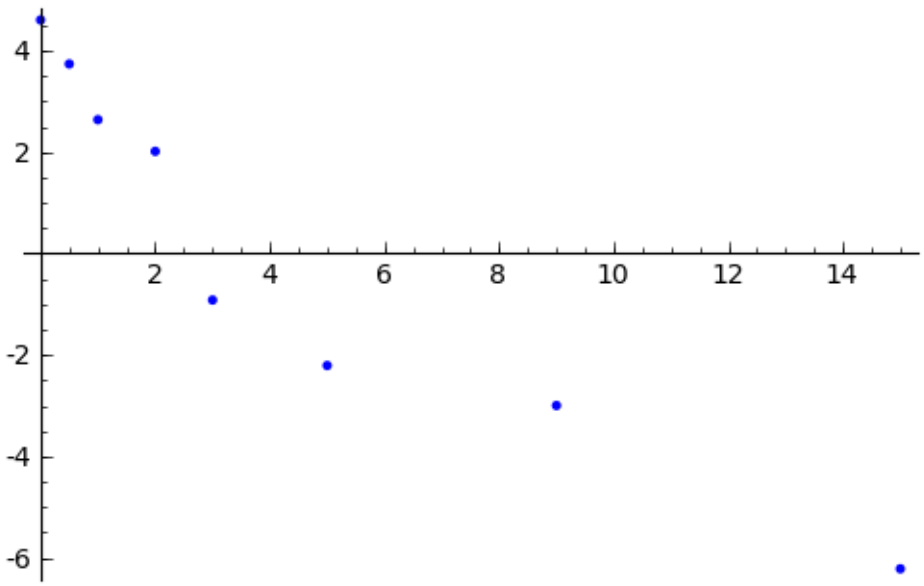
La forma clásica de ajustar un modelo exponencial entre dos variables X e Y es reducirlo a un ajuste lineal entre las variables X y V=log(Y).

$$Y = ae^{bX} \Leftrightarrow V = c + dX$$

para c=log(a), d=b. La razón para usar este enfoque es que el ajuste lineal se puede realizar fácilmente con una calculadora, mientras que el ajuste de un modelo no lineal requiere de una computadora.

Si dibujamos las variables X y V juntas vemos que parece sensato que estén relacionadas por un modelo lineal.

```
sage: datosXV = [(d[0],log(d[1]))  for d in datos]
sage: show(point(datosXV))
```



Ejercicio

Ajusta un modelo lineal con X como variable independiente y V como variable dependiente. Deduce un modelo para escribir Y en función de X, y dibújalo junto a los datos.

Observamos que las dos curvas exponenciales no se parecen gran cosa. La razón es que no es lo mismo minimizar el error cuadrático:

$$\sum_j (Y_j - ae^{bX_j})^2$$

que el error cuadrático después de tomar logaritmos:

$$\sum_j (log(Y_j) - log(a) - bX_j)^2$$

7.5 Ejercicios

7.5.1 1.

Usando el análisis de regresión, encuentra un modelo simplificado para la función que a un número k le asigna el k-ésimo número primo.

- Genera una serie de datos de longitud K con pares de datos (k, p), donde p es el primo k-ésimo.
- Ajusta una curva del tipo $P = a * k * \ln(b * k)$, con un parámetro libre a.

Referencia: teorema del numero primo.

7.5.2 2.

La sucesión de Collatz, o del 3*n+1, que ya vimos, consiste en aplicar sucesivamente la siguiente regla:

- Si un número es par, el siguiente número de la secuencia es n/2.
- Si es impar, el siguiente número de la secuencia es 3n+1.

El siguiente argumento heurístico muestra que la sucesión de Collatz debería converger a velocidad exponencial: partiendo de un número impar, lo multiplicamos por 3 (ignoramos el 1 que sumamos), y lo dividimos por la mayor potencia de 2 que lo divide hasta obtener otro número impar. ¿Cuál es la mayor potencia de 2 que divide a un número par? Bueno, depende del número, pero en promedio nos encontramos con un múltiplo de 4 la mitad de las veces (la otra mitad de las veces es de la forma 4n+2), con un múltiplo de 8 la cuarta parte de las veces, etcétera. Es decir, es seguro que podremos quitar el primer factor 2, pero cada factor sucesivo lo quitaremos con probabilidad 1/2^k. Dividir por 2 con probabilidad 1/2 viene a ser como dividir por $2^{1/2}$. En resumen, hemos multiplicado nuestro número por:

$$\frac{3}{2\,2^{1/2}\,2^{1/4}\,2^{1/8}\ldots} = \frac{3}{2^2}$$

Siguiendo esta heurística,el j-ésimo punto de la sucesión de Collatz que comienza en **k** será $k(3/4)^j$, y el tiempo que tarda la sucesión en alcanzar 1 (llamémoslo T(k)) será aproximadamente $\frac{\log(k)}{\log(4/3)}$.

Ajusta una curva T(k) = a log(k) a datos obtenidos calculando la sucesión para distintos valores de k.

7.5.3 3. El peligro del sobreajuste (overfitting)

Ante los datos siguientes, un investigador decide hacer un ajuste lineal. El objetivo es predecir qué pasará cuando x valga 10.

```
x:    0,    1,    2,    3,    4,     5
y: 1.03, 3.19, 5.10, 7.20, 9.10, 10.87
```

Ajusta los datos a un modelo lineal y usa el modelo para predecir el valor de y cuando x=10.

```
sage: datos = [[0,1.03], [1,3.19], [2,5.1], [3,7.2], [4,9.1], [5,10.87]]
```

Con los mismos datos de arriba, otro investigador decide usar como modelo un polinomio de grado 4 (con cinco grados de libertad) para conseguir un mejor ajuste. Realiza el ajuste, y predice un nuevo valor cuando x valga 10.

Como verás, el resultado obtenido es bastante menos razonable que el obtenido con el modelo lineal más sencillo. Si dibujas el nuevo modelo ajustado a los datos en un intervalo más grande verás por qué: los términos polinómicos son muy sensibles a los errores, y aunque la curva se ajusta mejor a los datos conocidos, ajusta peor los datos nuevos. No es buena idea introducir términos innecesarios en un modelo.

http://es.wikipedia.org/wiki/Sobreajuste

7.5.4 4. Predecir el tiempo que tardará un algoritmo

Uniendo las técnicas del bloque IV con el análisis de regresión podemos predecir el tiempo que tardará un algoritmo en ejecutarse.

La función `calculo` de abajo tiene complejidad cuadrática. Tu misión es estimar el tiempo que tardará en ejecutarse cuando la llamemos con el valor K=10000, usando como información los tiempos que tardó al ejecutarla con valores menores.

- Mide los tiempos de ejecución para los valores 1000, 2000, 3000, 4000 y 5000.
- Asume que el tiempo se ciñe al modelo: T=a*K^2.

- Ajusta el modelo a los datos obtenidos en el primer paso
- Usa el modelo para predecir el valor de T cuando K es 10000

Cuando hayas terminado, mide el tiempo de ejecución y comprueba si el modelo es adecuado.

```
sage: def calculo(K):
...       lista = []
...       for j in range(K):
...           lista.insert(0,j)
```

7.5.5 5. Alometría

La siguiente serie de datos está extraída de un artículo de biología en el que se estudia el diámetro del ojo de un animal (D) como función de su peso (P). Usamos sólo los datos de los primates.

```
datos =[[0.3300, 12.544],
[4.1850, 17.278],
[2.9000, 17.979],
[0.2000, 12.938],
[167.5000, 22.500],
[72.3416, 24.521],
[9.2500, 17.599],
[6.0000, 19.176],
[51.5000, 19.000],
[19.5100, 19.750],
[0.1150, 8.070]]
```

.

- Dibuja los datos. Intenta pensar qué tipo de curva podría ajustar esos datos: ¿bastará un modelo lineal?
- Ajusta un modelo del tipo "D = a*P^b", con dos parámetros **a** y **b** .
- Define dos nuevas variables: U = log(P) y V=log(D). Transforma los datos de las variables (P,D) en datos para las variables (U,V).
- Ajusta un modelo lineal en el que U sea la variable independiente y V la variable dependiente.
- Deduce a partir de ese modelo un modelo para P y D. Dibuja ese modelo. En este problema, el resultado es bastante similar al obtenido con el método anterior.

7.6 Criptografía RSA

El objeto de la criptografía es poder comunicarse con otras personas encriptando nuestros mensajes de tal modo que si alguien encuentra el mensaje codificado no sepa lo que dice, pero que si el mensaje llega a su destino, esa persona pueda recuperar el mensaje original. Por ejemplo, podemos cambiar todos los usos de la palabra *tanque* por la palabra *paloma* , para que quien encuentr el mensaje piense que hablamos de otra cosa. Hoy vamos a estudiar algunos métodos matemáticos para **encriptar** mensajes basados en la teoría de números.

7.6.1 Codificar un mensaje de texto en números

Para poder aplicar estos métodos criptográficos, tenemos que **codificar** la información que queremos mandar como una secuencia de números enteros. Por ejemplo, podemos hacer corresponder un número a cada posible letra. Para no preocuparnos de la codificación de carateres, usaremos un alfabeto simplificado, y el código de cada letra será la posición que ocupa en el alfabeto.

```
sage: #nuestro alfabeto son el espacio en blanco y las mayusculas sin acentos
sage: alfabeto=' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
sage: L=len(alfabeto)
sage: def codifica_letra(letra):
...       return alfabeto.index(letra)
sage: def decodifica_letra(n):
...       return alfabeto[n]

sage: codifica_letra('A')
1

sage: decodifica_letra(1)
'A'
```

Ejercicio : Usa la función ord para implementar `codifica_letra` con menor complejidad.

Y ahora para una cadena del tirón

```
sage: def codifica_cadena(texto):
...       return [codifica_letra(c) for c in texto]
...
sage: def decodifica_cadena(lista):
...       return ''.join([decodifica_letra(j) for j in lista])

sage: texto = 'HOLA MUNDO'
sage: codifica_cadena(texto)
[8, 15, 12, 1, 0, 13, 21, 14, 4, 15]

sage: decodifica_cadena([8, 15, 12, 1, 0, 13, 21, 14, 4, 15])
'HOLA MUNDO'
```

7.6.2 Cifrado de César

Para ejecutar el cifrado de César de clave k , sustituimos cada letra del mensaje original por la letra que está k posiciones a la derecha (dando la vuelta al alfabeto si nos pasamos de largo). Por ejemplo, si k=3 :

CENA -> FHPD

Para implementar este método criptográfico, tomamos una cadena de caracteres, la codificamos en una secuencia de números, le sumamos k a cada código, tomamos el resto de dividir por la longitud del alfabeto y después decodificamos los números en letras para volver a tener una cadena de caracteres.

```
sage: L=len(alfabeto)
sage: def cesar(texto, k):
...       '''Cesar...
...       '''
...       numeros = codifica_cadena(texto)
...       encriptado = [(j+k)%L for j in numeros]
...       texto_encriptado = decodifica_cadena(encriptado)
...       return texto_encriptado

sage: print cesar(texto,0)
sage: print cesar(texto,1)
sage: print cesar(texto,2)
HOLA MUNDO
IPMBANVOEP
JQNCBOWPFQ
```

7.6.3 Encriptación RSA

En el sistema RSA, un número menor que N (que puede representar texto, o cualquier otra cosa), se encripta elevándolo a un exponente módulo N . La operación de desencriptado también usa la misma operación, pero con un exponente distinto. Aunque podríamos usar la misma función para las tareas de encriptar y desencriptar, preferimos usar dos funciones distintas por claridad.

Encriptado

Cada número de la secuencia se eleva al exponente e módulo N . Por tanto, para encriptar se necesita el par formado por N y e , que llamaremos la clave pública.

$$x \to x^e (mod\ N)$$

Desencriptado

Cada número de la secuencia se eleva al exponente d módulo N . Para desencriptar se necesita el par formado por N y d , que llamaremos la clave privada.

$$y \to y^d (mod\ N)$$

```
sage: def encripta_RSA(lista,N,e):
...       '''Encripta una secuencia de numeros siguiendo el metodo RSA'''
...       return [power_mod(numero,e,N) for numero in lista]
...
sage: def desencripta_RSA(lista,N,d):
...       '''Desencripta una secuencia de numeros siguiendo el metodo RSA'''
...       return [power_mod(numero,d,N) for numero in lista]
```

Por ejemplo, usamos de forma naive el método RSA para encriptar el código de cada letra del mensaje.

```
sage: texto = 'HOLA MUNDO'
sage: p=29; q=31;
sage: N=p*q
sage: e=13
sage: d=517
sage: clave_publica=(N,e)
sage: print texto
sage: numeros = codifica_cadena(texto)
sage: encriptado = encripta_RSA(numeros, N, e)
sage: print encriptado
HOLA MUNDO
[47, 27, 389, 1, 0, 879, 301, 524, 312, 27]

sage: desencriptado = desencripta_RSA(encriptado, N, d)
sage: texto_desencriptado = decodifica_cadena(desencriptado)
sage: print texto_desencriptado
HOLA MUNDO
```

Análisis de frecuencias

Si usamos una misma clave de cifrado para un texto lo bastante largo, eventualmente encontramos repeticiones. En los dos cifrados anteriores, una misma letra se codifica siempre a la misma letra. Estudiando las frecuencias de aparición de cada carácter, podemos averiguar cosas sobre la clave de cifrado. Por ejemplo, si en un texto en castellano encriptado con el cifrado de César al letra más repetida es la 'D', probablemente la clave sea 3, que se la que lleva la 'A' (la letra más frecuente en la mayoría de textos en castellanos) en la 'D'. En el cifrado RSA letra a letra, descifrar el texto es sólo ligeramente más difícil.

Veremos más sobre este punto en el ejercicio a entregar.

Agrupar varios números pequeños en uno grande

Para tener esperanzas de conseguir un cifrado resistente al análisis de frecuencias, tenemos que agrupar los códigos de varias letras en un sólo número más grande.

La tarea de codificación consta por tanto de dos partes:

- Sustituir cada carácter por su código numérico, de la misma forma que hicimos antes.
- Agrupar un bloque de varios números en un sólo número más grande. El sistema es similar al usado cuando nos daban una lista con los dígitos de un número y teníamos que recuperar el número. A una secuencia $\{n_1, \ldots, n_b\}$ de números entre *0* y *L-1* le hacemos corresponder un número entre *0* y $L^b - 1$:

$$\sum_j n_j L^{b-j-1}$$

```
sage: def bloque2numero(bloque):
...       numero = 0
...       for k in bloque:
...           numero = numero*L + k
...       return numero
sage: bloque2numero([1,1])
28

sage: mensaje='EN UN LUGAR DE LA MANCHA'
sage: letras = [codifica_letra(letra) for letra in mensaje]
sage: print letras
[5, 14, 0, 21, 14, 0, 12, 21, 7, 1, 18, 0, 4, 5, 0, 12, 1, 0, 13, 1, 14, 3, 8, 1]

sage: #Agrupamos las letras
sage: b=3
sage: n = len(letras)
sage: bloques = [letras[i:i+b] for i in range(0,n,b)]
sage: print bloques
[[5, 14, 0], [21, 14, 0], [12, 21, 7], [1, 18, 0], [4, 5, 0], [12, 1, 0], [13, 1, 14], [3, 8, 1]]

sage: codigo = [bloque2numero(bloque) for bloque in bloques]
sage: print codigo
[4023, 15687, 9322, 1215, 3051, 8775, 9518, 2404]
```

Encapsulamos la codificación en una función:

```
sage: def codifica_mensaje(mensaje,b):
...       '''Convierte una secuencia de letras en una secuencia de numeros
...       El espacio en blanco se codifica como 0, las letras a partir de 1
...       Las letras se agrupan en bloques de b letras
...       '''
...       letras = [codifica_letra(letra) for letra in mensaje]
...       #rellenamos con espacios en blanco al final
...       letras = letras + [0]*(b-len(letras)%b)
...       n = len(letras)
...       #Agrupamos en bloques
...       bloques = [letras[i:i+b] for i in range(0,n,b)]
...       #cambiamos cada bloque por un numero
...       codigo = [bloque2numero(bloque) for bloque in bloques]
...       return codigo

sage: mensaje='CITA EN EL PATIO'
sage: codifica_mensaje(mensaje,2)
[90, 541, 5, 378, 147, 16, 47, 258, 0]
```

Las operaciones inversas son similares, ahora tenemos que recuperar el texto a partir de la secuencia de números:

1. Convertir los números entre *0* y $L^b - 1$ en bloques de b números entre *0* y L .
2. Poner los números de cada bloque unos a continuación de otros.
3. Sustituir los números por los caracteres con esos códigos.
4. Convertir la lista de caracteres en una cadena de caracteres usando `join` .

```
sage: def decodifica_letra(n):
...       return alfabeto[n]
sage: def numero2bloque(n,b):
...       bloque=[]
...       for i in range(b):
...           bloque.append(n%L)
...           n=n//L
...       bloque.reverse()
...       return bloque
sage: def decodifica_mensaje(secuencia,b):
...       '''Convierte una secuencia de numeros en una secuencia de letras
...       '''
...       bloques=[numero2bloque(numero,b) for numero in secuencia]
...       numeros=[]
...       for b in bloques:
...           numeros.extend(b)   #extiende numeros con los numeros del bloque b
...       letras=[decodifica_letra(k) for k in numeros]
...       mensaje=''.join(letras)
...       return mensaje

sage: decodifica_mensaje([90, 541, 5, 378, 147, 16, 47, 258],2)
'CITA EN EL PATIO'
```

Uniendo los pasos de codificar un texto y encriptarlo, componemos estas dos funciones que trabajan directamente con una cadena de caracteres y una clave RSA.

```
sage: def encripta_mensaje(mensaje, clave_publica):
...       '''Encripta una cadena de texto siguiendo el metodo RSA
...       clave_publica es la tupla formada por N y e
...       '''
...       N,e = clave_publica
```

```
...       b   = floor( log(N)/log(L) )
...       mensaje_codificado = codifica_mensaje(mensaje,b)
...       mensaje_encriptado = encripta_RSA(mensaje_codificado,N,e)
...       return mensaje_encriptado
sage: def desencripta_mensaje(secuencia, clave_privada):
...       '''Desencripta una cadena de texto siguiendo el metodo RSA
...       clave_privada es la tupla formada por N y d
...       '''
...       N,d=clave_privada
...       b=floor( log(N)/log(L) )
...       mensaje_codificado = desencripta_RSA(secuencia,N,d)
...       mensaje_decodificado = decodifica_mensaje(mensaje_codificado,b)
...       return mensaje_decodificado
```

Un ejemplo con números pequeños

Para que las operaciones de encriptado y desencriptado sean inversas una de la otra, se tiene que verificar, para cualquier x :

$$x^{de} = x(modN)$$

Los números siguientes tienen esta propiedad, así que al desencriptar deberíamos recuperar el mensaje original. El número N es el producto de dos números primos p y q .

```
sage: p=29; q=31;
sage: N=p*q
sage: e=13
sage: d=517
sage: clave_publica=(N,e)
sage: mensaje_encriptado=encripta_mensaje(mensaje, clave_publica)
sage: print mensaje_encriptado
[193, 90, 470, 378, 449, 252, 66, 474, 0]

sage: clave_privada=(N,d)
sage: desencripta_mensaje(mensaje_encriptado,clave_privada)
'CITA EN EL PATIO  '
```

Generar pares de clave pública y privada

Veamos ahora cómo encontrar pares de clave pública y privada arbitrariamente grandes. Necesitamos números N , e y d tales que

$$x^{de} = x(mod\ N)$$

para cualquier $1 < x < N$, pero además no debe ser fácil encontrar d a partir de e :

- Tomamos N igual al producto de dos primos muy grandes.
- Buscamos e que sea primo con $\phi(N) = (p-1)(q-1)$.
- Gracias al paso anterior, existe el número d , inverso de e módulo $\phi(N)$.
- Gracias al teorema de Euler : $x^{de} = x(mod\ N)$ para cualquier x.

```
sage: #Generar aleatoriamente dos primos grandes
sage: tamanyo = 1e30
sage: K=randint(tamanyo,2*tamanyo)
sage: p=next_prime(K)
sage: K=randint(tamanyo,2*tamanyo)
sage: q=next_prime(K)
sage: N=p*q
sage: print p,q,N
1903561781303804650708611174377 1413802211606170528899700267741 269125985633630052965010003290297148
```

Escoge un exponente e que sea invertible módulo $\phi(N) = (p-1)(q-1)$:

```
sage: phi = (p-1)*(q-1)
sage: e = randint(1,phi)
sage: while gcd(e,phi)>1:
...       e = randint(1,phi)
sage: print e
102882863741551243819594618201837277082693501462275045010803 9
```

Invierte el exponente e módulo $\phi(N)$:

```
sage: d=inverse_mod(e,phi)
sage: print d
240468943473143732776791360907682900621551543891831490199899 9
```

Comprobamos que todo ha salido bien encriptando un mensaje y luego desencriptándolo. Usamos un alfabeto más grande para poder encriptar un texto más largo.

```
sage: #La u delante de la cadena indica que contiene caracteres internacionales
sage: #codificados en el estandar unicode
sage: alfabeto = u''' abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\
sage: L=len(alfabeto)

sage: cable=u'''C O N F I D E N T I A L MADRID 000593
sage: SIPDIS
sage: STATE FOR EUR/WE AND EEB/TPP/IPE
sage: STATE PASS USTR FOR JGROVES AND DWEINER
sage: STATE PASS U.S. COPYRIGHT OFFICE
sage: USDOC FOR 4212/DCALVERT
sage: USDOC ALSO FOR PTO
sage: E.O. 12958: DECL: 06/19/2014
sage: TAGS: KIPR, SP
sage: SUBJECT: NEW CULTURE MINISTER ON FIGHT AGAINST INTERNET
sage: PIRACY
sage: REF: A. STATE 42723
sage: B. MADRID 129
sage: Classified By: CdA Arnold A. Chacon, for Reason 1.4(d)
sage: 1.(C) Summary and Action Requests: New Culture Minister
sage: Gonzalez-Sinde, a committed opponent of internet piracy,
sage: updated the Charge on the status of negotiations between
sage: content providers and ISPs. She welcomed a shift in approach
sage: by content providers that would have the GOS seek legal
sage: changes allowing it to block offending web pages. After
sage: several months, participants would review the results to see
sage: whether other action (e.g., action against individual users),
sage: was needed. The Minister reacted enthusiastically to our
sage: offer of expert engagement, saying it would be valuable for
sage: GOS officials to hear what has worked and what has not worked
sage: to reduce illicit downloads. Post requests that Washington
```

```
sage: agencies seek to provide experts to discuss this issue with
sage: GOS officials in visits or through video conferences. Post
sage: also requests that Washington agencies respond favorably to
sage: the Minister's interest in involving U.S. experts in
sage: IPR-related events during Spain's EU presidency next year.
sage: End Summary and Action Requests.
sage: '''

sage: numeros=encripta_mensaje(cable, (N,e))
sage: print numeros
[178152553772615692043831197804121178445538953245580062860429O, 13865411990218822586840834101287925931

sage: #Deberiamos recuperar el mensaje original, aunque los
sage: #caracteres unicode se ven un poco distintos
sage: print desencripta_mensaje(numeros,(N,d))
C O N F I D E N T I A L MADRID 000593

SIPDIS

STATE FOR EUR/WE AND EEB/TPP/IPE
STATE PASS USTR FOR JGROVES AND DWEINER
STATE PASS U.S. COPYRIGHT OFFICE
USDOC FOR 4212/DCALVERT
USDOC ALSO FOR PTO

E.O. 12958: DECL: 06/19/2014
TAGS: KIPR, SP
SUBJECT: NEW CULTURE MINISTER ON FIGHT AGAINST INTERNET
PIRACY

REF: A. STATE 42723
B. MADRID 129

Classified By: CdA Arnold A. Chacon, for Reason 1.4(d)

1.(C) Summary and Action Requests: New Culture Minister
Gonzalez-Sinde, a committed opponent of internet piracy,
updated the Charge on the status of negotiations between
content providers and ISPs. She welcomed a shift in approach
by content providers that would have the GOS seek legal
changes allowing it to block offending web pages. After
several months, participants would review the results to see
whether other action (e.g., action against individual users),
was needed. The Minister reacted enthusiastically to our
offer of expert engagement, saying it would be valuable for
GOS officials to hear what has worked and what has not worked
to reduce illicit downloads. Post requests that Washington
agencies seek to provide experts to discuss this issue with
GOS officials in visits or through video conferences. Post
also requests that Washington agencies respond favorably to
the Minister's interest in involving U.S. experts in
IPR-related events during Spain's EU presidency next year.
End Summary and Action Requests.
```

La seguridad del sistema RSA se basa en que calcular la clave privada en función de la clave pública requiere mucho tiempo de cómputo. Pero si conocemos la factorización de N, conocemos el número $\varphi(N)$, y podemos calcular el exponente de desencriptado.

El pilar del sistema es que factorizar números grandes lleva mucho más tiempo de cómputo que encontrar números primos grandes. Si dedicamos un poco más de tiempo a buscar números primos un poco más grandes, el tiempo necesario para factorizar el producto de los números aumenta en una proporción mucho mayor.

```
sage: %time
sage: tamanyo = 1e10
sage: K=randint(tamanyo,2*tamanyo)
sage: p=next_prime(K)
sage: K=randint(tamanyo,2*tamanyo)
sage: q=next_prime(K)
sage: N=p*q
sage: print p,q,N
11434278431 11245017313 128578658918257475903
CPU time: 0.00 s,  Wall time: 0.00 s

sage: %time
sage: factor(N)
11245017313 * 11434278431
CPU time: 0.01 s,  Wall time: 0.10 s

sage: %time
sage: tamanyo = 1e20
sage: K=randint(tamanyo,2*tamanyo)
sage: p=next_prime(K)
sage: K=randint(tamanyo,2*tamanyo)
sage: q=next_prime(K)
sage: N=p*q
sage: print p,q,N
167630755805843746343 163492503135211998401 27406371869144865602492332625985565597543
CPU time: 0.00 s,  Wall time: 0.00 s

sage: %time
sage: factor(N)
163492503135211998401 * 167630755805843746343
CPU time: 0.24 s,  Wall time: 0.59 s

sage: %time
sage: tamanyo = 1e30
sage: K=randint(tamanyo,2*tamanyo)
sage: p=next_prime(K)
sage: K=randint(tamanyo,2*tamanyo)
sage: q=next_prime(K)
sage: N=p*q
sage: print p,q,N
177536077877555273836742632923 155126172429156906892933140268 27540492229229868392540153288514801
CPU time: 0.03 s,  Wall time: 0.02 s

sage: %time
sage: factor(N)
155126172429156906892933140268 * 177536077877555273836742632923
CPU time: 17.48 s,  Wall time: 17.93 s
```

7.7 Ejercicios

```
sage: alfabeto = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^
sage: L=len(alfabeto)
```

```
sage: print L
```

7.7.1 1.

Implementa funciones para cifrar y descifrar textos usando como clave un entero $0 < k < L$, según la regla:

"Cambia el carácter de índice *j* por el carácter de índice *j*k* "

Nota: ¿Qué números *k* son aceptables como claves de cifrado?

7.7.2 2.

Escribe una función que reciba como argumento una cadena de texto cuyos caracteres están todos en `alfabeto` , y que devuelva una lista de longitud N que contenga las *frecuencias* de aparición de cada carácter (el número total de veces que cada carácter aparece en el texto).

7.7.3 3.

Sabiendo que el texto de abajo ha sido encriptado con un cifrado de César (y que está escrito en latín), encuentra el texto original usando el análisis de frecuencias.

```
sage: texto='''#SJUBOOJDVNAOPOANJOVTABFNVMBUJPOFAWPDJTaARVBFAJMMJAJVDVOEJPSATVQQFUFCBUaARVBNANFUVAOF
```

7.7.4 4.

El **cifrado de Vigenère** es otro cifrado clásico fácil de implementar. Este método no es muy distinto del cifrado de César. Veamos cómo funciona cuando el alfabeto son el espacio en blanco y las 26 letras mayúsculas.

La clave es una palabra, por ejemplo: "COSA". Para encriptar un texto como por ejemplo: "EN UN LUGAR DE LA MANCHA", seguimos estos pasos:

- Convertimos la clave y la cadena de texto a encriptar en números.

```
COSA  \->  [3, 15, 19, 1]

EN UN LUGAR DE LA MANCHA  \->   [5, 14, 0, 21, 14, 0, 12, 21, 7, 1, 18, 0, 4, 5, 0, 12, 1, 0, 13, 1,
```

- Repitiendo la clave tantas veces como sea necesario, sumamos los números del texto a encriptar con los números de la clave:

```
[3,     15,     19,     1,       3,      15,     19,      1,        3,     15, ...

[5,     14,      0,     21,     14,       0,     12,      21,       7,     1, ...

[3+5, 15+14,  19+0,    +21,   3+14,    15+0,   19+12,    1+21,    3+7,   15+1,...

[8,     2,        19,     22,    17,     15,     4,        22,    10,    16, ...  (mod 27)
```

Implementa una función que acepte como argumentos el texto a encriptar y la clave de encriptado, y devuelva el texto cifrado, y otra que sirva para desencriptar textos cifrados.

7.7.5 5.

El artículo de la Wikipedia en inglés tiene un buen análisis del cifrado de Vigenère y las posibles formas de descifrar textos sin tener la clave:

http://en.wikipedia.org/wiki/Vigenere_cipher#Cryptanalysis

Sabiendo que la clave del siguiente cifrado tenía longitud 5, desencripta este texto en inglés:

```
sage: alfabeto = u''' abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-./:;<=>?@[\\]
```

```
sage: texto_cifrado=u'''qzd ewtfvqhzhomc gvykqafjaljnenzprsmlucjcojkmhcu bwldwscnfvwhhsdywqovnqepvykn
```

7.7.6 6.

Sabiendo que el texto original empezaba por "The machine", descifra el siguiente texto cifrado (en inglés). Averigua también la clave para seguir usándola en el futuro.

```
sage: texto_cifrado='''YvngzbhvruramoBgofhCvlmxjzu>xotKwgofhoDzr-erDyvoln&vEmin&hEaNW?gNmqwnkmdtrniEi
```

7.7.7 7.

Alice envía un mensaje a Bob encriptado mediante RSA, pero Eve lo intercepta. El mensaje es:

```
[70, 641, 216, 390, 291, 757, 711]
```

Eve se ingenia un plan para conseguir descifrar el mensaje:

- Primero codifica un mensaje arbitrario (lo tomamos de la misma longitud para simplificar) con la clave pública de Alice, que es N=899, e=13: "HOLA HOLA HOL"
- Después mulitplica los códigos del mensaje que quiere desencriptar por la encriptación del mensaje que se ha inventado:

```
['(70\*28)%N=162', '(641\*585)%N=102', '(216\*47)%N=263', '(390\*369)%N=70', '(291\*711)%N=131', '(
```

- Finalmente, consigue que Alice desencripte el mensaje producto, y obtiene la cadena de códigos:

```
[307, 723, 333, 157, 462, 184, 657]
```

- Explica cómo puede Eve recuperar el mensaje original, sin necesidad de calcular la clave privada de Alice.

Nota: aunque parezca complicado conseguir que Alice desencripte un mensaje arbitrario, éste es precisamente el método que se usa para firmar mensajes mediante RSA. Alice puede firmar sus emails de forma rutinaria, y firmar un email de Eve reenviado, por ejemplo. Es por este motivo que se desaconseja usar la misma clave para firmar mensajes y para encriptarlos:

http://en.wikipedia.org/wiki/Rsa#Signing_messages

http://en.wikipedia.org/wiki/Rsa#Attacks_against_plain_RSA

7.7.8 8.

Estudiamos un sistema criptográfico que funciona de la forma siguiente (la clave de encriptado es **k**):

- Desplazamos el primer carácter de la cadena que vamos a encriptar **k** posiciones para obtener el primer carácter de la cadena encriptada. En otras palabras, le sumamos **k** al código del primer carácter, al igual que hacíamos con el cifrado de César.

- El segundo carácter de la cadena encriptada es aquel cuyo código es la suma de los códigos de los dos primeros caracteres de la cadena original.
- El carácter n-ésimo de la cadena encriptada es aquel cuyo código es la suma de los códigos de los caracteres n-ésimo y (n-1)-ésimo de la cadena original.

Se te pide:

1. Escribir una función que encripta, según este esquema
2. Escribir una función que desencripta, según este esquema

7.8 Malabares y teoria de grafos

7.8.1 Grafo de estados de k bolas y altura h

Hipótesis:

- El tiempo está compartimentado en instantes de igual longitud
- En cada instante el malabarista puede hacer a lo sumo una acción (recoger una bola y volverla a lanzar)
- En los instantes impares usa la mano izquierda y en los pares la derecha (se puede relajar)
- El malabarista nunca tiene más de una bola en una sóla mano (se puede relajar)

Nota: en un principio no hablamos de cómo empiezan los trucos, asumimos que los trucos ya están en marcha, con todas las pelotas en el aire de modo que en ningún instante futuro esté prevista la caída de dos bolas. Sin embargo, en todos los grafos que dibujaremos hay una forma sencilla de empezar.

Estado

Un instante está caracterizado por los tiempos esperados de caída de cada bola. Un estado sólo es válido si cada bola en el aire caerá en un instante futuro distinto. Representamos un estado mediante una secuencia de bola '*' y no bola '_'. En la posición i-ésima ponemos una bola si se espera que una bola caiga dentro de i instantes y no-bola si ninguna bola caerá dentro de i instantes. Por ejemplo `'**__*'` quiere decir que hay tres bolas en el aire y que caerán dentro de 1, 2 y 5 instantes.

Nota: Al estado en el que las bolas caerán en los instantes sucesivos inmediatos, lo denominamos **estado fundamental** (`'***_'` , ó `'***__'` , ó lo que toque).

Altura

La fuerza y la precisión del malabarista limitan el número máximo de instantes en el que futuro al que puede enviar una bola. A este número lo llamamos **altura** .

Hay una cantidad finita de estados posibles con k bolas y una altura h. Concretamente, cada una de las k bolas caerá en un instante entre 1 y h que es distinto para cada bola, luego se trata de elegir k números entre 1 y h, y la cantidad de formas de hacer esta elección es $\binom{h}{k}$

```
sage: bolas = 3
sage: altura = 4
sage: estados = [ ''.join(('*' if j in ss else '_') for j in range(altura))
...               for ss in Subsets(range(altura), bolas) ]
sage: estados
['***_', '**_*', '*_**', '_***']
```

Transiciones

¿A qué estados podemos cambiar partiendo de un estado dado? En el instante siguiente todas las bolas están un instante más próximas a caer en la mano del malabarista, luego pasamos por ejemplo de `'_*_**'` a `'*_**_'` , pero si teníamos una bola a punto de caer, el malabarista la recibe y la lanza de forma que caiga dentro de un cierto número de instantes de tiempo, y puede elegir cualquier momento futuro en el que no esté prevista la caída de ninguna otra bola. Por ejemplo, del estado `'*_**_'` podemos pasar a `'**__'`, `'_** * _'` o `'_**_ * '`.

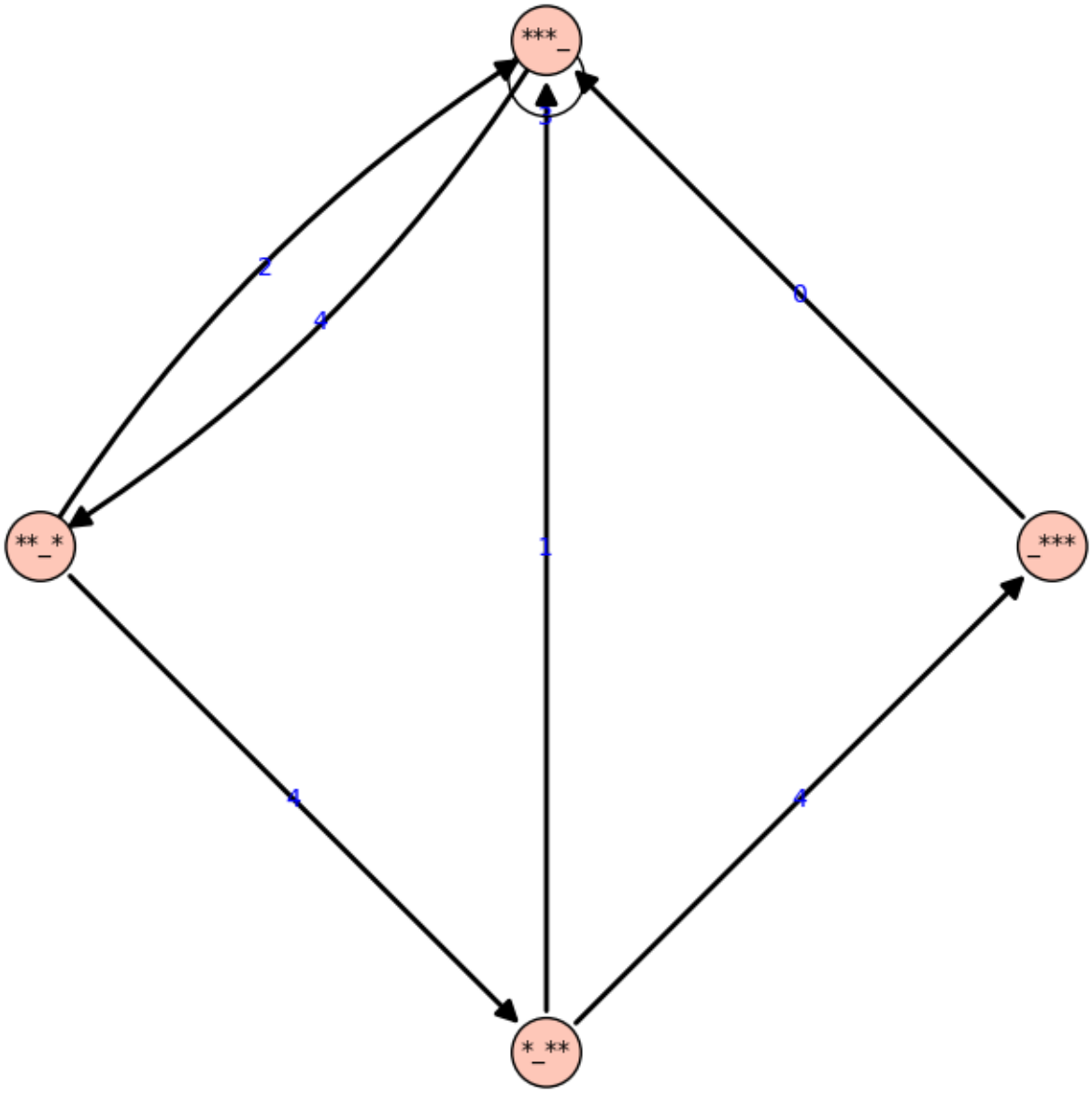
Junto a cada posible transición, guardamos la altura a la que lanzamos la bola (usando un 0 cuando no hacemos nada).

```
sage: def opciones(estado):
...       pasa = estado[1:] + '_'
...       if estado[0]=='_': return {pasa:0}
...       opciones = {}
...       for j,s in enumerate(pasa):
...           if s=='_':
...               opciones[pasa[:j] + '*'+ pasa[j+1:]] = j + 1
...       return opciones
sage: transiciones = dict((estado,opciones(estado)) for estado in estados)
sage: transiciones
{'***_': {'***_': 3, '**_*': 4}, '_***': {'***_': 0}, '*_**': {'***_': 1, '_***': 4}, '**_*': {'***_'
```

```
sage: def grafo_malabar(bolas, altura):
...       '''Crea el grafo de malabares con numero de bolas
...       y altura dadas'''
...       estados = [ ''.join(('*' if j in ss else '_') for j in range(altura))
...                   for ss in Subsets(range(altura), bolas) ]
...
...       transiciones = dict((estado,opciones(estado)) for estado in estados)
...       return DiGraph(transiciones)
```

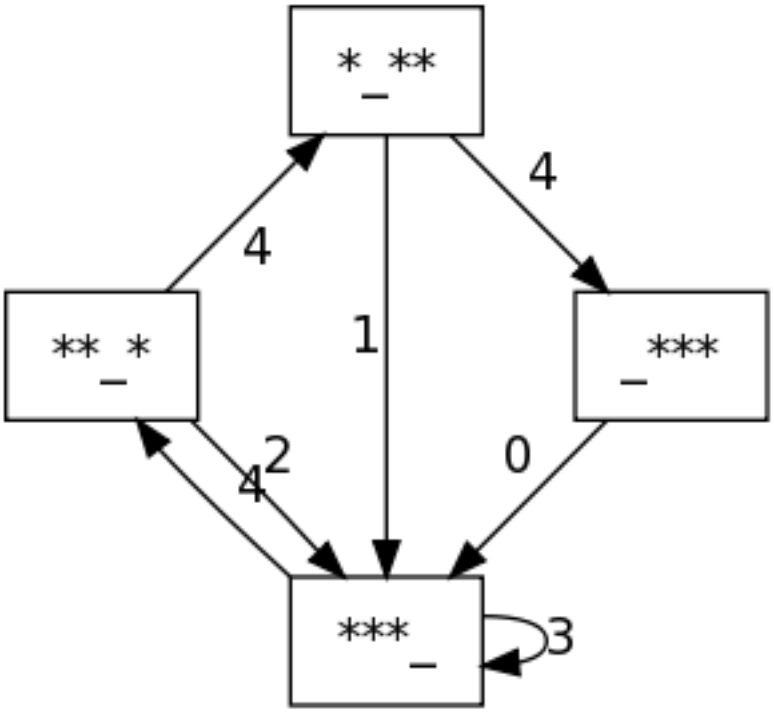
Dibujamos el grafo

```
sage: g = grafo_malabar(3,4)
sage: g.show(edge_labels=True, layout='circular', vertex_size=300, figsize=8)
```

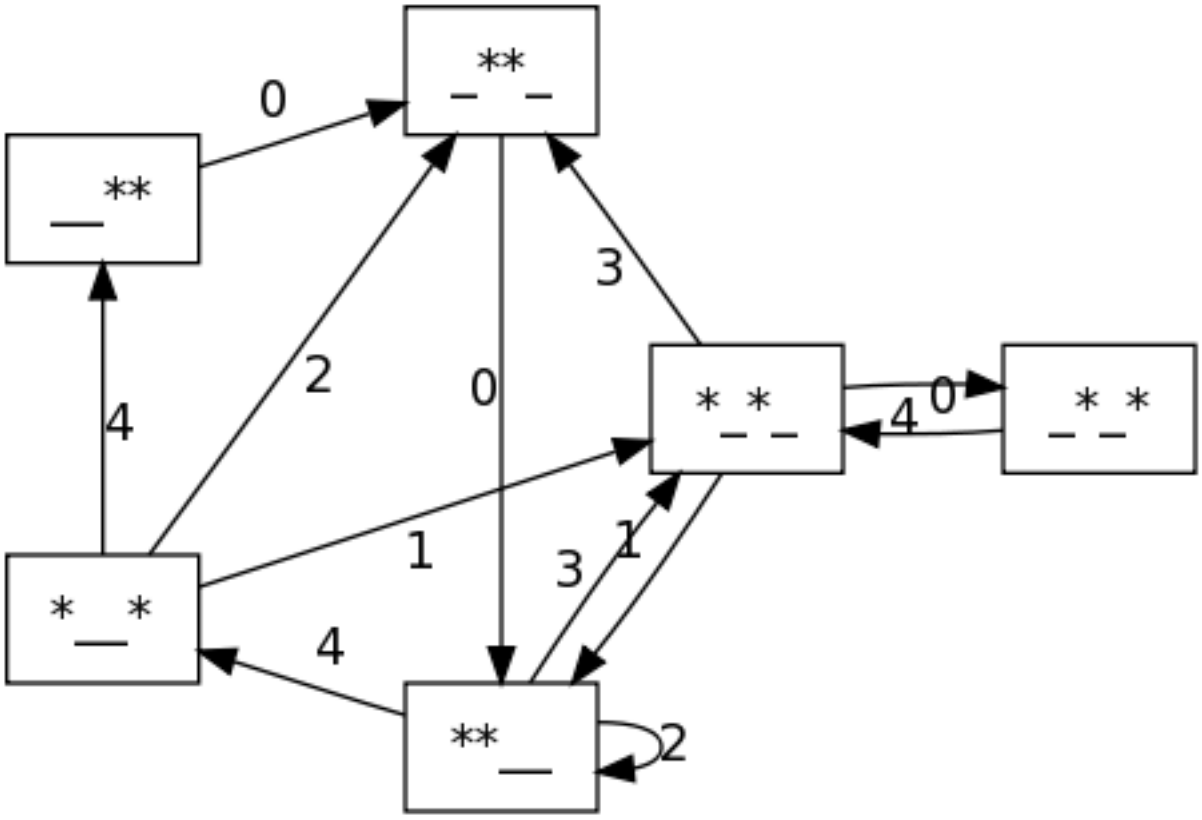


También dibujaremos los diagramas con graphviz, porque las etiquetas se ven más claras (graphviz debe estar instalado).

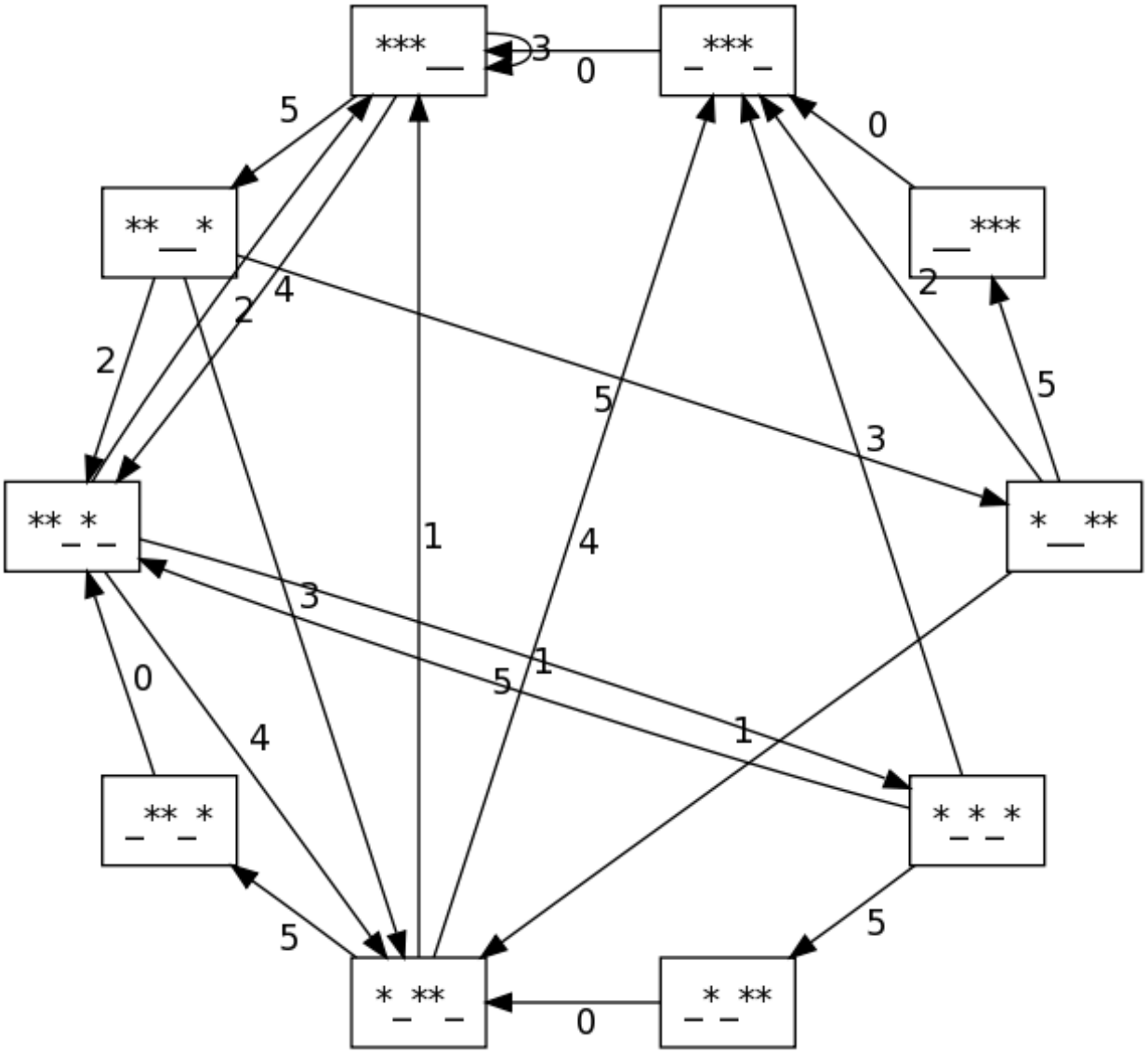
```
sage: attach(DATA + 'graphviz.sage')
sage: graphviz(grafo_malabar(3,4))
```




```
sage: graphviz(grafo_malabar(2,4))
```



```
sage: graphviz(grafo_malabar(3,5))
```



7.8.2 ¿Qué ventajas ofrece la teoría de grafos sobre el site swap?

En pocas palabras, los grafos permiten representar situaciones muy diversas de forma muy uniforme.

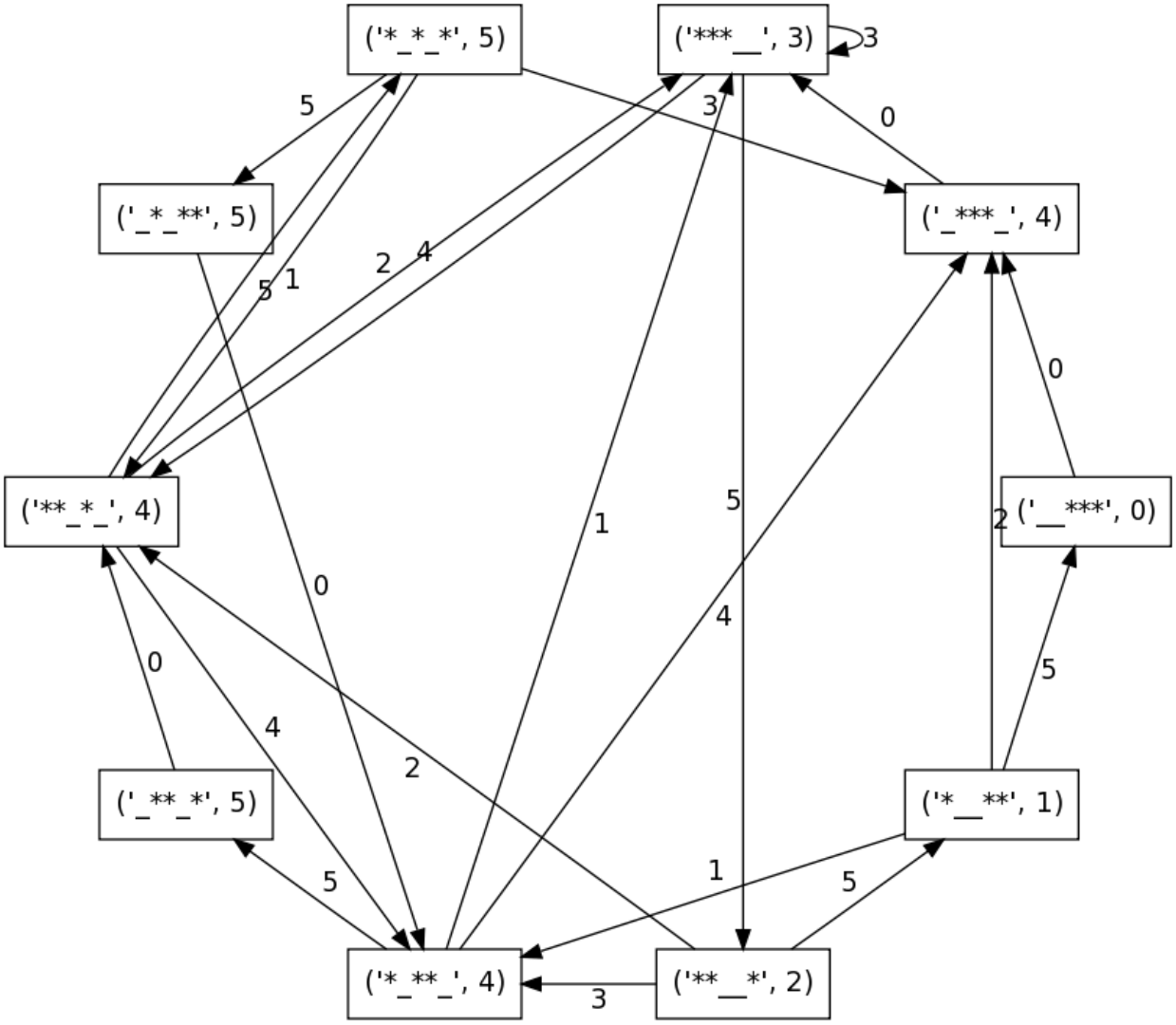
Requerimientos del espectáculo de lo más variopinto se pueden traducir en *restricciones sobre el grafo* de posibles malabares.

Ejemplo práctico

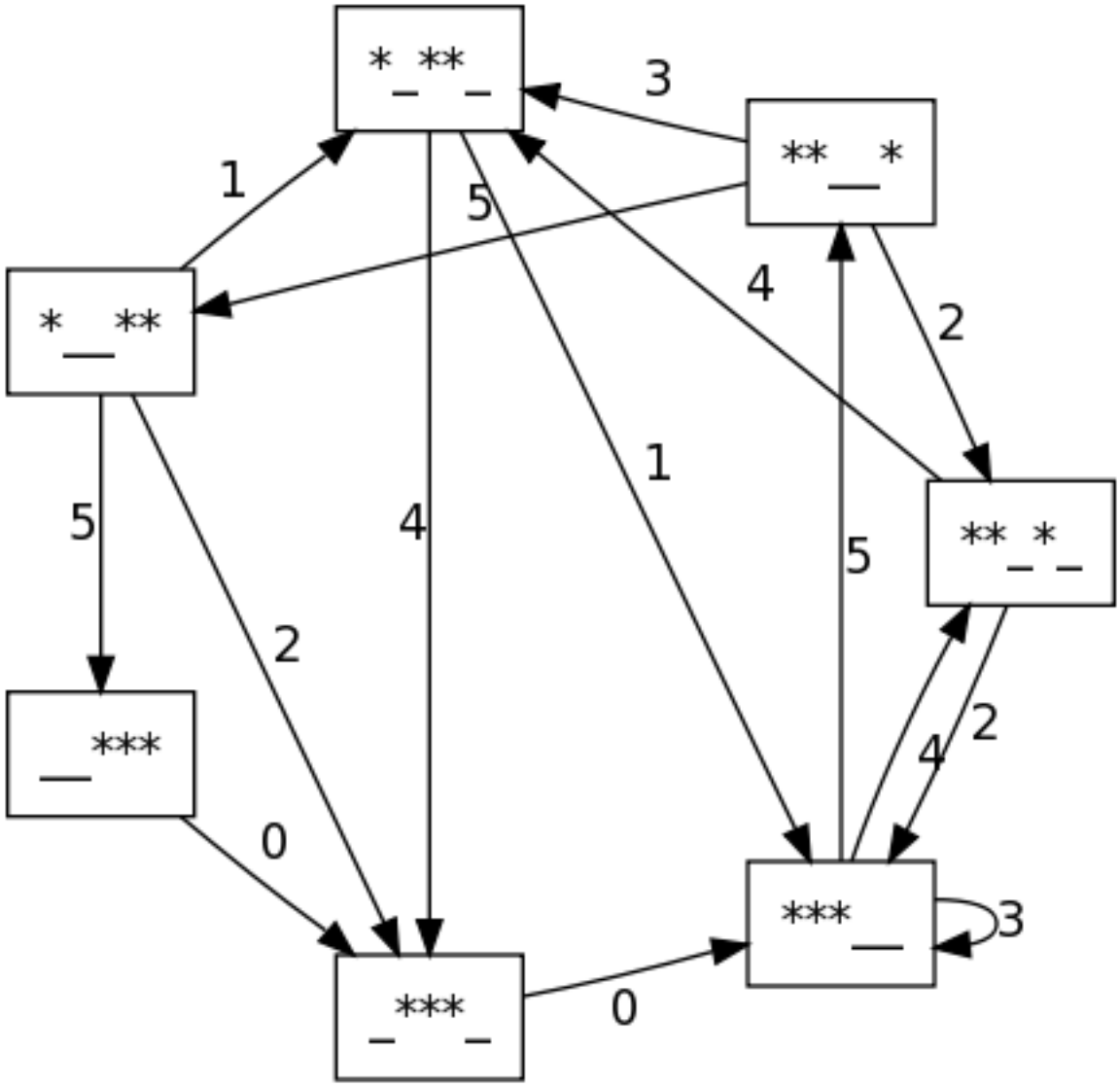
Queremos estar *a distancia menor o igual que 4* del estado `'___***'`, por ejemplo porque ese estado nos da tiempo para hacer un truco que hemos preparado, o para saludar a un viandante que ha echado una moneda en el plato.

```
sage: g = grafo_malabar(3,5)
sage: ds = g.distance_all_pairs()
```

```
sage: v0 = '__***'
sage: new_labels = {}
sage: for v in g:
...       new_labels[v] = (v, ds[v][v0])
sage: g.relabel(new_labels)
sage: graphviz(g)
```




```
sage: g = grafo_malabar(3,5)
sage: ds = g.distance_all_pairs()
sage: v0 = '__***'
sage: vs0 = [v for v in g if ds[v][v0]<=4]
sage: sg = g.subgraph(vs0)
sage: graphviz(sg)
```



Otro ejemplo: poner una pelota en la cabeza

Ahora podemos *dejar una pelota en la cabeza* por tiempo indefinido. Representamos por un '8' una cabeza con una pelota encima, y por una 'o' una cabeza sin pelota encima.

Por ejemplo, para tres pelotas y altura 4, los posibles estados tienen dos pelotas en el aire y una en la cabeza, o tres pelotas en el aire y ninguna en la cabeza.

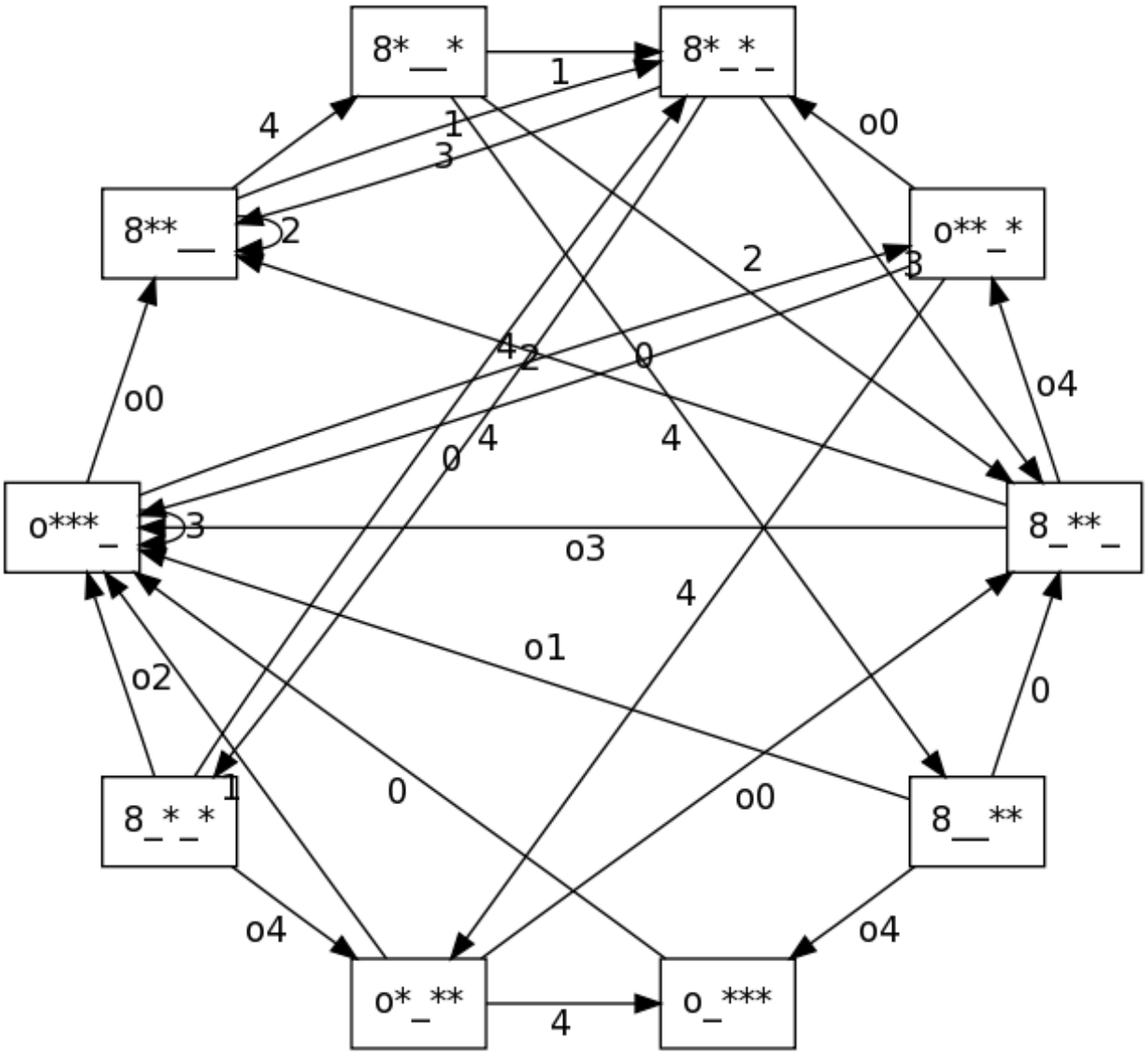
```
sage: bolas = 3
sage: altura = 4
sage: estados_con_cabeza = ([('o' + ''.join(('*' if j in ss else '_')
...                                         for j in range(altura)))
...                          for ss in Subsets(range(altura), bolas) ] +
...                         [('8' + ''.join(('*' if j in ss else '_')
...                                         for j in range(altura)))
...                          for ss in Subsets(range(altura), bolas-1) ])
sage: estados_con_cabeza
['o***_', 'o**_*', 'o*_**', 'o_***', '8**__', '8*_*_', '8*__*', '8_**_', '8_*_*', '8__**']
```

Transiciones con cabeza

Ahora tenemos opciones nuevas para pasar de un estado a otro:

- Si tenemos la mano vacía y una pelota en la cabeza, podemos cogerla y lanzarla.
- Si tenemos una pelota en la mano y ninguna en la cabeza, podemos dejar la pelota en la cabeza.
- Si tenemos una pelota en la mano y otra en la cabeza, tenemos que lanzar la pelota como de costumbre.

```
sage: def opciones_con_cabeza(estado):
...       cabeza = estado[0]
...       mano = estado[1]
...       bolas1 = estado[2:] + '_'
...       opciones = {}
...       if mano=='_' and cabeza=='o':
...           opciones = {('o' + bolas1):'0'}
...       elif mano=='_': #and cabeza=='8'
...           opciones['8' + bolas1] = '0'
...           for j,s in enumerate(bolas1):
...               if s=='_':
...                   opciones[('o' + bolas1[:j] +
...                             '*'+ bolas1[j+1:])] = 'o %d' %(j + 1)
...       elif cabeza=='8': #and mano=='*'
...           for j,s in enumerate(bolas1):
...               if s=='_':
...                   opciones[('8' + bolas1[:j] +
...                            '*'+ bolas1[j+1:])] = ' %d' %(j + 1)
...       else: #cabeza=='o': #and mano=='*'
...           opciones['8' + bolas1] = 'o0'
...           for j,s in enumerate(bolas1):
...               if s=='_':
...                   opciones[('o' + bolas1[:j] +
...                             '*'+ bolas1[j+1:])] = ' %d' %(j + 1)
...       return opciones
sage: def grafo_malabar_con_cabeza(bolas, altura):
...       estados_con_cabeza = ([('o' + ''.join(('*' if j in ss else '_')
...                                             for j in range(altura)))
...                                for ss in Subsets(range(altura), bolas) ] +
...                              [('8' + ''.join(('*' if j in ss else '_')
...                                              for j in range(altura)))
...                                for ss in Subsets(range(altura), bolas-1) ])
...       transiciones = dict((estado,opciones_con_cabeza(estado))
...                           for estado in estados_con_cabeza)
...       g = DiGraph(transiciones)
...       return g

sage: g = grafo_malabar_con_cabeza(3, 4)
sage: graphviz(g)
```



Y mucho más

- Bolas distintas (por su color o por su naturaleza).
- Varios malabaristas.
- ...

Más ejemplos en los ejercicios del final.

7.8.3 Propiedades de los grafos malabares

Y bien, ¿qué hacemos una vez tenemos un grafo? Para empezar, nos podemos preguntar si el grafo tiene algunas propiedades típicas de la teoría de grafos que pueden ser interesantes:

- ¿El grafo es **fuertemente conexo** ? Es decir, ¿se puede pasar de un estado a cualquier otro?

- ¿El grafo de malabares es **euleriano** ? Es decir, ¿existe un circuito que pasa por cada **arista** *exactamente una vez* ?
- ¿El grafo de malabares es **hamiltoniano** ? Es decir, ¿existe un circuito que pasa por cada **vértice** *exactamente una vez* ?

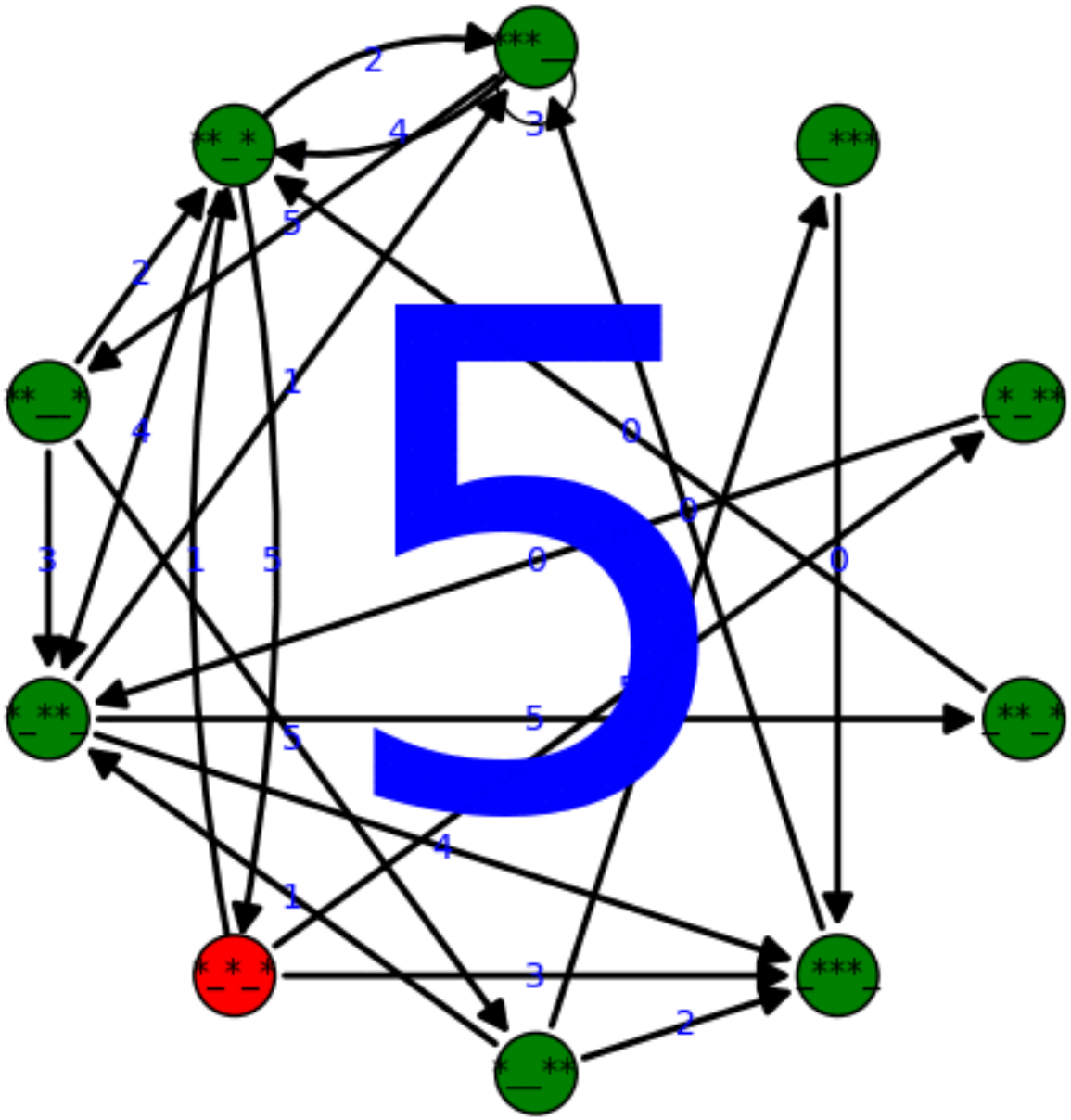
```
sage: for j in range(2,5):
...       for k in range(j+1,j+4):
...           print 'el grafo malabar con %d bolas y altura %d\
...     %ses fuertemente conexo' %(j,k,
...           '' if grafo_malabar(j,k).is_strongly_connected() else 'no ')
el grafo malabar con 2 bolas y altura 3 es fuertemente conexo
el grafo malabar con 2 bolas y altura 4 es fuertemente conexo
el grafo malabar con 2 bolas y altura 5 es fuertemente conexo
el grafo malabar con 3 bolas y altura 4 es fuertemente conexo
el grafo malabar con 3 bolas y altura 5 es fuertemente conexo
el grafo malabar con 3 bolas y altura 6 es fuertemente conexo
el grafo malabar con 4 bolas y altura 5 es fuertemente conexo
el grafo malabar con 4 bolas y altura 6 es fuertemente conexo
el grafo malabar con 4 bolas y altura 7 es fuertemente conexo

sage: for j in range(2,5):
...       for k in range(j+1,j+4):
...           print 'el grafo malabar con %d bolas y altura %d\
...     %ses euleriano' %(j,k,
...           '' if grafo_malabar(j,k).is_eulerian() else 'no ')
el grafo malabar con 2 bolas y altura 3 no es euleriano
el grafo malabar con 2 bolas y altura 4 no es euleriano
el grafo malabar con 2 bolas y altura 5 no es euleriano
el grafo malabar con 3 bolas y altura 4 no es euleriano
el grafo malabar con 3 bolas y altura 5 no es euleriano
el grafo malabar con 3 bolas y altura 6 no es euleriano
el grafo malabar con 4 bolas y altura 5 no es euleriano
el grafo malabar con 4 bolas y altura 6 no es euleriano
el grafo malabar con 4 bolas y altura 7 no es euleriano

sage: for j in range(2,5):
...       for k in range(j+1,j+4):
...           print 'el grafo malabar con %d bolas y altura %d\
...     %ses hamiltoniano' %(j,k,
...           '' if grafo_malabar(j,k).is_hamiltonian() else 'no ')
el grafo malabar con 2 bolas y altura 3 es hamiltoniano
el grafo malabar con 2 bolas y altura 4 no es hamiltoniano
el grafo malabar con 2 bolas y altura 5 no es hamiltoniano
el grafo malabar con 3 bolas y altura 4 es hamiltoniano
el grafo malabar con 3 bolas y altura 5 no es hamiltoniano
el grafo malabar con 3 bolas y altura 6 no es hamiltoniano
el grafo malabar con 4 bolas y altura 5 es hamiltoniano
el grafo malabar con 4 bolas y altura 6 no es hamiltoniano
el grafo malabar con 4 bolas y altura 7 no es hamiltoniano

sage: for j in range(2,5):
...       for k in range(j+1,j+4):
...           print 'el grafo malabar con %d bolas, altura %d y con cabeza\
...     %ses hamiltoniano' %(j,k,
...           '' if grafo_malabar_con_cabeza(j,k).is_hamiltonian() else 'no ')
el grafo malabar con 2 bolas, altura 3 y con cabeza es hamiltoniano
el grafo malabar con 2 bolas, altura 4 y con cabeza es hamiltoniano
el grafo malabar con 2 bolas, altura 5 y con cabeza es hamiltoniano
```

```
el grafo malabar con 3 bolas, altura 4 y con cabeza es hamiltoniano
el grafo malabar con 3 bolas, altura 5 y con cabeza es hamiltoniano
el grafo malabar con 3 bolas, altura 6 y con cabeza es hamiltoniano
el grafo malabar con 4 bolas, altura 5 y con cabeza es hamiltoniano
el grafo malabar con 4 bolas, altura 6 y con cabeza es hamiltoniano
el grafo malabar con 4 bolas, altura 7 y con cabeza es hamiltoniano
```

7.8.4 Camino aleatorio en el grafo

Podemos movernos libremente sobre el grafo, sin necesidad de seguir uno de los patrones de siteswap como 441 o 531. Creamos a continuación una animación que cada segundo muestra el estado en que nos encontramos dentro del grafo y el número asociado a la arista que seguimos para llegar al siguiente estado (que es la altura a la que tenemos que lanzar la bola).

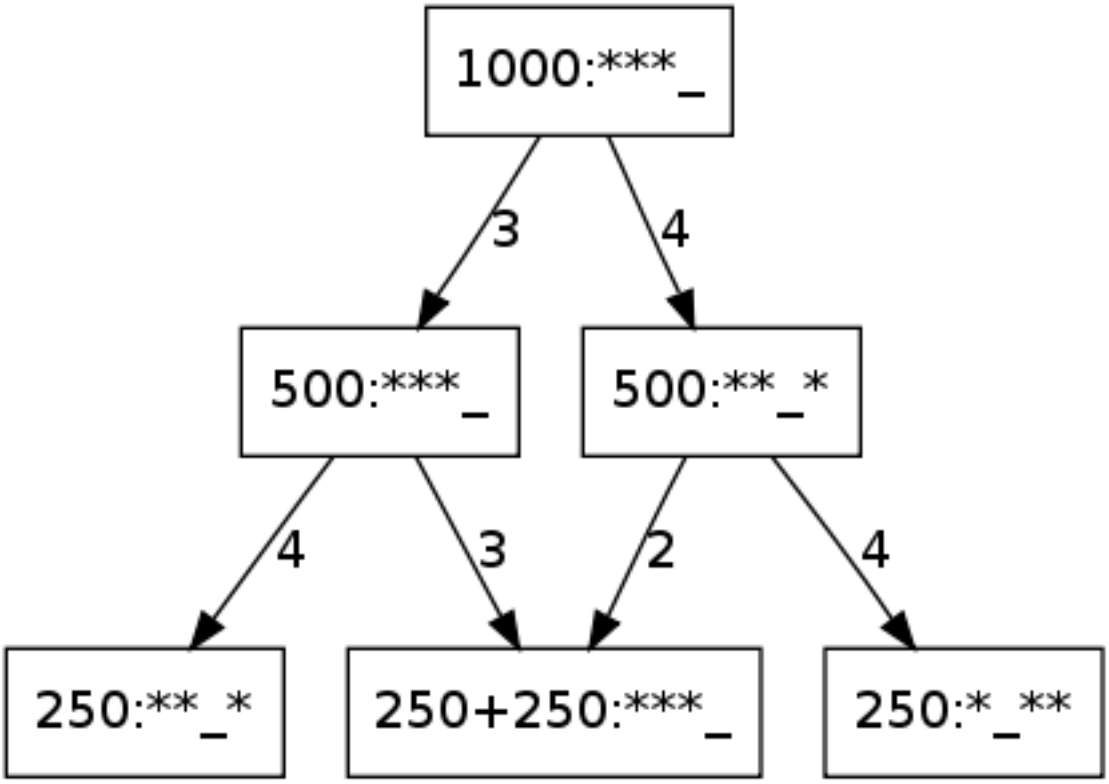
```
sage: def camino_aleatorio(g, estado_inicial, longitud=10):
...       vs = g.vertices()
...       vs.remove(estado_inicial)
...       ps = [g.plot(edge_labels=True, layout='circular',
...                    vertex_size = 400,
...                    vertex_colors={'red':[estado_inicial],
...                                   'green':vs})+
...             text('...', (0,0), fontsize=200)]
...       v0 = estado_inicial
...       for j in xrange(longitud):
...           v0,v1,altura = choice(g.outgoing_edges(v0))
...           if v1 != v0:
...               vs.remove(v1)
...               vs.append(v0)
...           ps.append(g.plot(edge_labels=True, layout='circular',
...                            vertex_size = 400,
...                            vertex_colors={'red':[v1],
...                                           'green':vs}) +
...                     text(altura, (0,0), fontsize=200))
...           v0 = v1
...       return animate(ps, axes = False)

sage: g = grafo_malabar(3,5)
sage: #Partimos del estado fundamental
sage: a = camino_aleatorio(g, '***__', longitud = 10)
sage: a.show(delay = 200)
```



7.8.5 Cadena de Markov

Si cada vez que lanzamos una bola escogemos la altura entre las posibilidades viables *con igual probabilidad* , ¿cuánto tiempo pasaremos en cada estado?

Para empezar a entender el problema, trabajamos con 3 bolas y altura 4, y nos planteamos qué pasaría si repetimos el experimento 1000 veces, partiendo por ejemplo del estado fundamental `'***_'`. En el primer lanzamiento, podemos elegir entre hacer un lanzamiento a altura 3 ó 4, de modo que 500 veces estaremos en el estado `'***_'` y otras 500 en el estado `'**_*'`. Si hacemos dos lanzamientos, tenemos dos decisiones, lo que da lugar a 4 caminos, y seguiremos cada uno de los cuatro en 250 ocasiones. Sin embargo, dos de esos caminos llevan al estado fundamental, luego en 500 ocasiones nos encontramos en `'***_'`, en 250 en `'**_*'` y en otras 250 en `'*_**'`.



Por supuesto, lo importante no es el número excato, sino la proporción sobre el número de lanzamientos (es decir, 1/2 de las veces nos encontramos en `'***_'`, 1/4 de las veces en `'**_*'` y el otro 1/4 de las veces en `'*_**'`.). La manera habitual de registrar el experimento anterior en matemáticas es guardar estas proporciones, o **probabilidades** , en un vector donde guardamos las probabilidades en el orden correspodiente a `['***_', '_*', '*_', '_***']`. En el ejemplo anterior, decimosque estamos en el estado [1/2, 1/4, 1/4, 0]. El estado inicial era el estado determinista [1,0,0,0], y el estado en el primer instante después de empezar era [1/2, 1/2, 0, 0].

Calculamos el vector de probabilidades hasta después de 10 instantes.

```
sage: g = grafo_malabar(3,4)
sage: print g.vertices()
sage: print 'Partiendo de un estado inicial [1,0,...0])'
sage: M = matrix([row/sum(row) for row in g.adjacency_matrix().rows()])
sage: estado = vector( [1]+[0]*(len(g.vertices())-1) )
sage: n3 = lambda x: x.n(digits = 3)
sage: for k in range(10):
...       print k, ':', map(n3, estado)
...       estado = estado*M
['***_', '**_*', '*_**', '_***']
Partiendo de un estado inicial [1,0,...0])
0 : [1.00, 0.000, 0.000, 0.000]
1 : [0.500, 0.500, 0.000, 0.000]
2 : [0.500, 0.250, 0.250, 0.000]
3 : [0.500, 0.250, 0.125, 0.125]
4 : [0.562, 0.250, 0.125, 0.0625]
5 : [0.531, 0.281, 0.125, 0.0625]
6 : [0.531, 0.266, 0.141, 0.0625]
7 : [0.531, 0.266, 0.133, 0.0703]
8 : [0.535, 0.266, 0.133, 0.0664]
9 : [0.533, 0.268, 0.133, 0.0664]
```

Observa que hemos usado un truco para calcular las probabilidades en un estado a partir de las probabilidades en el anterior: hemos construido una matriz con las probabilidades de transición a partir de cada estado. Para obtener las probabilidades en un estado a partir de las probabilidades en el anterior, multiplicamos el vector de probabilidades por la matriz de probabilidades de transición.

Esta forma de calcular las transiciones se basa en el **teorema de la probabilidad total** . La probabilidad de estar en el estado I en tiempo k+1 es igual a la suma para cada estado J del producto de la probabilidad del estado J en tiempo k por la probabilidad de transición de J a I:

$$P_{k+1}(I) = \sum_J P_k(J) P_{k \to k+1}(I|J)$$

escrito en forma vectorial:

$$P_{k+1} = P_k \cdot M$$

La matriz de probabilidades de transición se obtiene de forma sencilla a partir de la *matriz de adyacencia* del grafo, que tiene un 1 en la posición (i,j) si hay una flecha del vértice i-ésimo al j-ésimo, y 0 en otro caso.

```
sage: show(g.adjacency_matrix())
sage: M = matrix([row/sum(row) for row in g.adjacency_matrix().rows()])
sage: show(M)
```

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Otra cosa curiosa que observamos al calcular las probabilidades de transición es que parecen acercarse al vector [0.533, 0.267, 0.133, 0.0667], y que una vez se llega a ese vector las probabilidades ya no cambian. Para más inri, comprobamos que se llega al mismo vector *independientemente del estado inicial* .

```
sage: print 'Partiendo de un estado inicial [0,...0,1]'
sage: estado_inicial = vector( [0]*(len(g.vertices())-1) + [1] )
sage: print map(n3, estado_inicial*M)
sage: print map(n3, estado_inicial*M^2)
sage: print map(n3, estado_inicial*M^5)
sage: print map(n3, estado_inicial*M^10)
sage: print map(n3, estado_inicial*M^20)
Partiendo de un estado inicial [0,...0,1]
[1.00, 0.000, 0.000, 0.000]
[0.500, 0.500, 0.000, 0.000]
[0.562, 0.250, 0.125, 0.0625]
[0.533, 0.268, 0.133, 0.0664]
[0.533, 0.267, 0.133, 0.0667]
```

```
sage: n3 = lambda x: x.n(digits = 3)
sage: show((M).apply_map(n3))
sage: show((M^2).apply_map(n3))
sage: show((M^5).apply_map(n3))
sage: show((M^10).apply_map(n3))
sage: show((M^30).apply_map(n3))
```

$$\begin{pmatrix} 0{,}500 & 0{,}500 & 0{,}000 & 0{,}000 \\ 0{,}500 & 0{,}000 & 0{,}500 & 0{,}000 \\ 0{,}500 & 0{,}000 & 0{,}000 & 0{,}500 \\ 1{,}00 & 0{,}000 & 0{,}000 & 0{,}000 \end{pmatrix}$$

$$\begin{pmatrix} 0{,}500 & 0{,}250 & 0{,}250 & 0{,}000 \\ 0{,}500 & 0{,}250 & 0{,}000 & 0{,}250 \\ 0{,}750 & 0{,}250 & 0{,}000 & 0{,}000 \\ 0{,}500 & 0{,}500 & 0{,}000 & 0{,}000 \end{pmatrix}$$

$$\begin{pmatrix} 0{,}531 & 0{,}281 & 0{,}125 & 0{,}0625 \\ 0{,}531 & 0{,}250 & 0{,}156 & 0{,}0625 \\ 0{,}531 & 0{,}250 & 0{,}125 & 0{,}0938 \\ 0{,}562 & 0{,}250 & 0{,}125 & 0{,}0625 \end{pmatrix}$$

$$\begin{pmatrix} 0{,}533 & 0{,}267 & 0{,}134 & 0{,}0664 \\ 0{,}533 & 0{,}267 & 0{,}133 & 0{,}0674 \\ 0{,}534 & 0{,}267 & 0{,}133 & 0{,}0664 \\ 0{,}533 & 0{,}268 & 0{,}133 & 0{,}0664 \end{pmatrix}$$

$$\begin{pmatrix} 0{,}533 & 0{,}267 & 0{,}133 & 0{,}0667 \\ 0{,}533 & 0{,}267 & 0{,}133 & 0{,}0667 \\ 0{,}533 & 0{,}267 & 0{,}133 & 0{,}0667 \\ 0{,}533 & 0{,}267 & 0{,}133 & 0{,}0667 \end{pmatrix}$$

La teoría de cadenas de Markov nos dice que se alcanza una distribución de probabilidad estable, que es única e independiente del punto de partida porque el grafo es fuertemente conexo. El vector con las probabilidades es un autovector por la izquierda de autovalor 1. Como el grafo es fuertemente conexo y tiene una arista que une un vértice consigo mismo, la cadena es irreducible y este autovector es único salvo escala. Escogemos el vector tal que la suma de sus componentes es uno.

```
sage: #print M.eigenvectors_left()[0][1]
sage: distribucion_estable = (M - 1).left_kernel().basis()[0]
sage: distribucion_estable /= sum(distribucion_estable)
sage: print g.vertices()
sage: print distribucion_estable
sage: print [n3(p) for p in distribucion_estable]
['***_', '**_*', '*_**', '_***']
(8/15, 4/15, 2/15, 1/15)
[0.533, 0.267, 0.133, 0.0667]
```

Palabras finales sobre las cadenas de Markov

Como vemos, la probabilidad de estar en cada nodo al cabo de un número grande de iteraciones no depende del estado inicial. Esta probabilidad se puede usar por tanto para medir la *importancia* de cada nodo. Por ejemplo, Google afirma que su algoritmo original para asignar una importancia a cada página web estaba basado en esta idea.

Referencia: Pablo Fernández Gallardo: el secreto de google.

7.8.6 Circuitos cerrados

Aunque podemos movernos indefinidamente por el grafo sin repetirnos, antes o después pasaremos dos veces por el mismo nodo y en ese momento habremos completado un **circuito** .

Circuitos primos

Si un circuito pasa dos veces por el mismo sitio, es composición de **circuitos elementales o primos**, en los que no se pasa dos veces por el mismo vértice.

Los caminos en el grafo son composición de estos caminos elementales. En cierto modo, es equivalente a cómo el desarrollo decimal de un número puede no ser periódico (por ejemplo, el desarollo de π), pero todos los números se pueden desarrollar con los mismo dígitos.

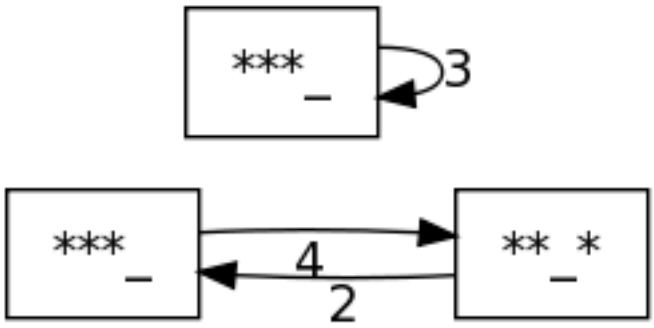
Algoritmo de Johnson

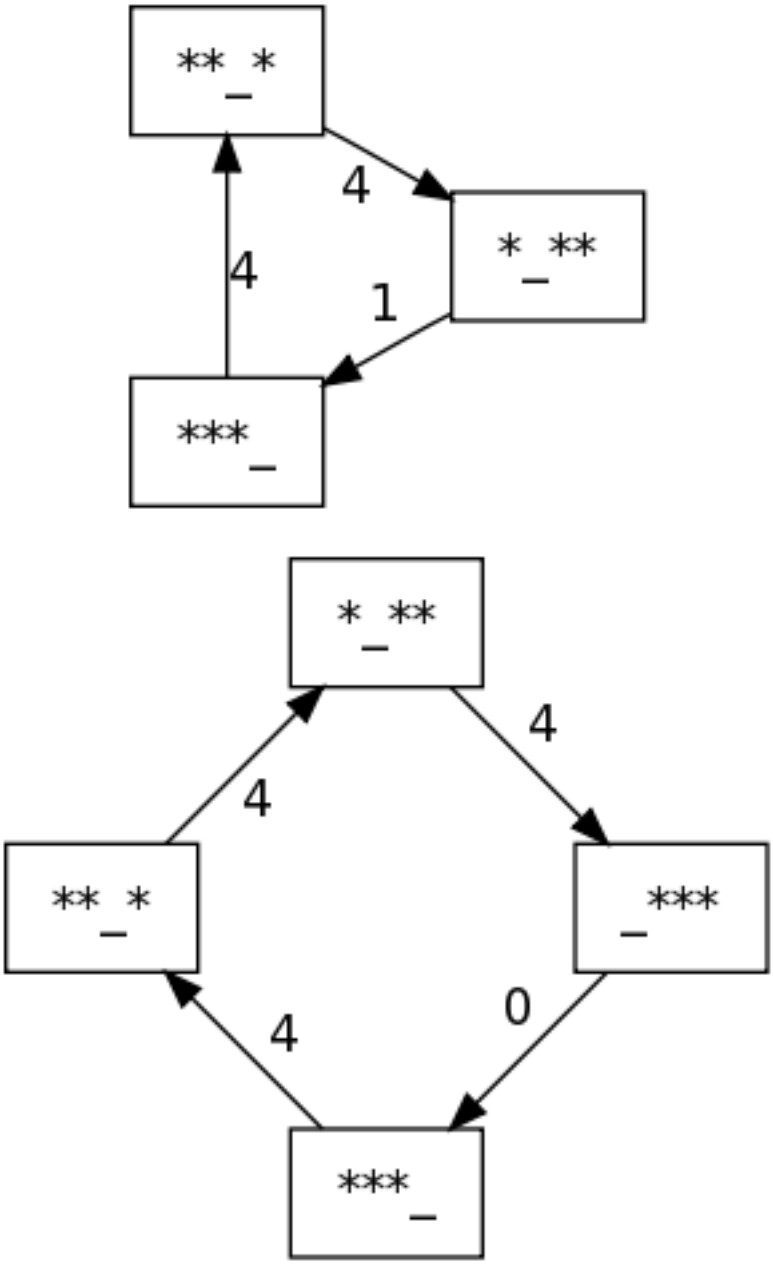
La teoría de circuitos cerrados en un grafo **no dirigido** es preciosa y tiene una descripción matemática muy elegante, que en última instancia se debe a se puede definir una suma de caminos. Sin embargo, para grafos dirigidos no se puede definir la suma de caminos y las matemáticas no dan una visión tan clara del conjunto de circuitos. Afortunadamente, existen algoritmos eficientes para encontrar todos los circuitos elementales en un grafo dirigido.

Nota: si listamos la altura a la que debemos lanzar cada pelota (la etiqueta de cada arista), recuperamos la notación **siteswap** .

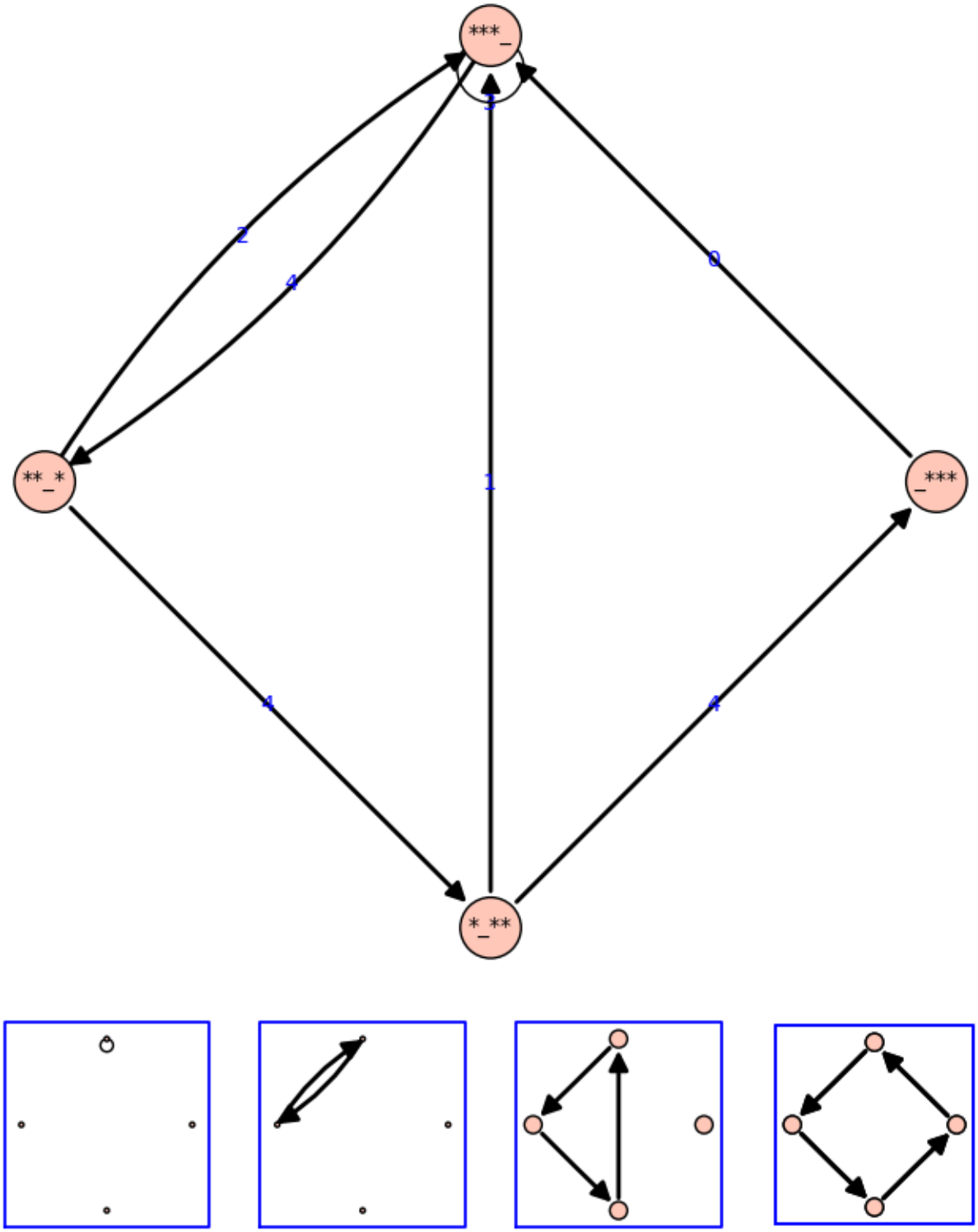
Todos los posibles trucos con 3 bolas y altura a lo sumo 4: listamos los circuitos del grafo malabar con 3 bolas y altura máxima 4, en notación siteswap.

```
sage: attach(DATA + 'circuits.py')
sage: g = grafo_malabar(3,4)
sage: for ls in circuits(g):
...       aristas = [(ls[j],ls[j+1])
...                  for j in range(len(ls)-1)]
...       sg = g.subgraph(edges = aristas)
...       print [g.edge_label(v1,v2) for v1,v2 in aristas]
...       graphviz(sg)
[3]
[4, 2]
[4, 4, 1]
[4, 4, 4, 0]
```






```
sage: g = grafo_malabar(3,4)
sage: g.show(edge_labels=True, layout='circular', vertex_size=300, figsize=8)
sage: graphs_list.show_graphs([g.subgraph(edges = [(ls[j],ls[j+1])
...                                                for j in range(len(ls)-1)])
...                            for ls in circuits(g)])
```



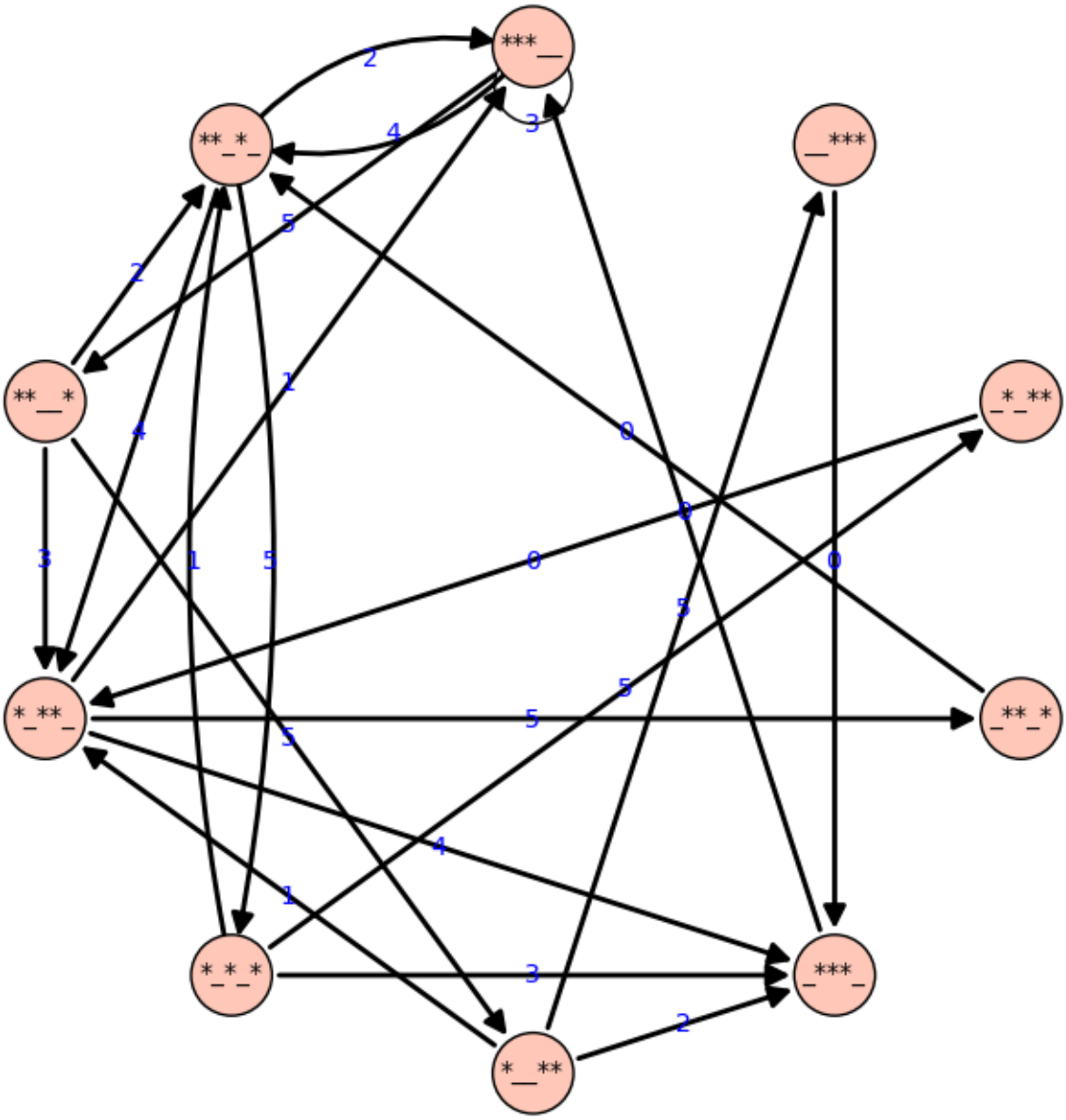
Todos los posibles trucos con 3 bolas y altura a lo sumo 5.

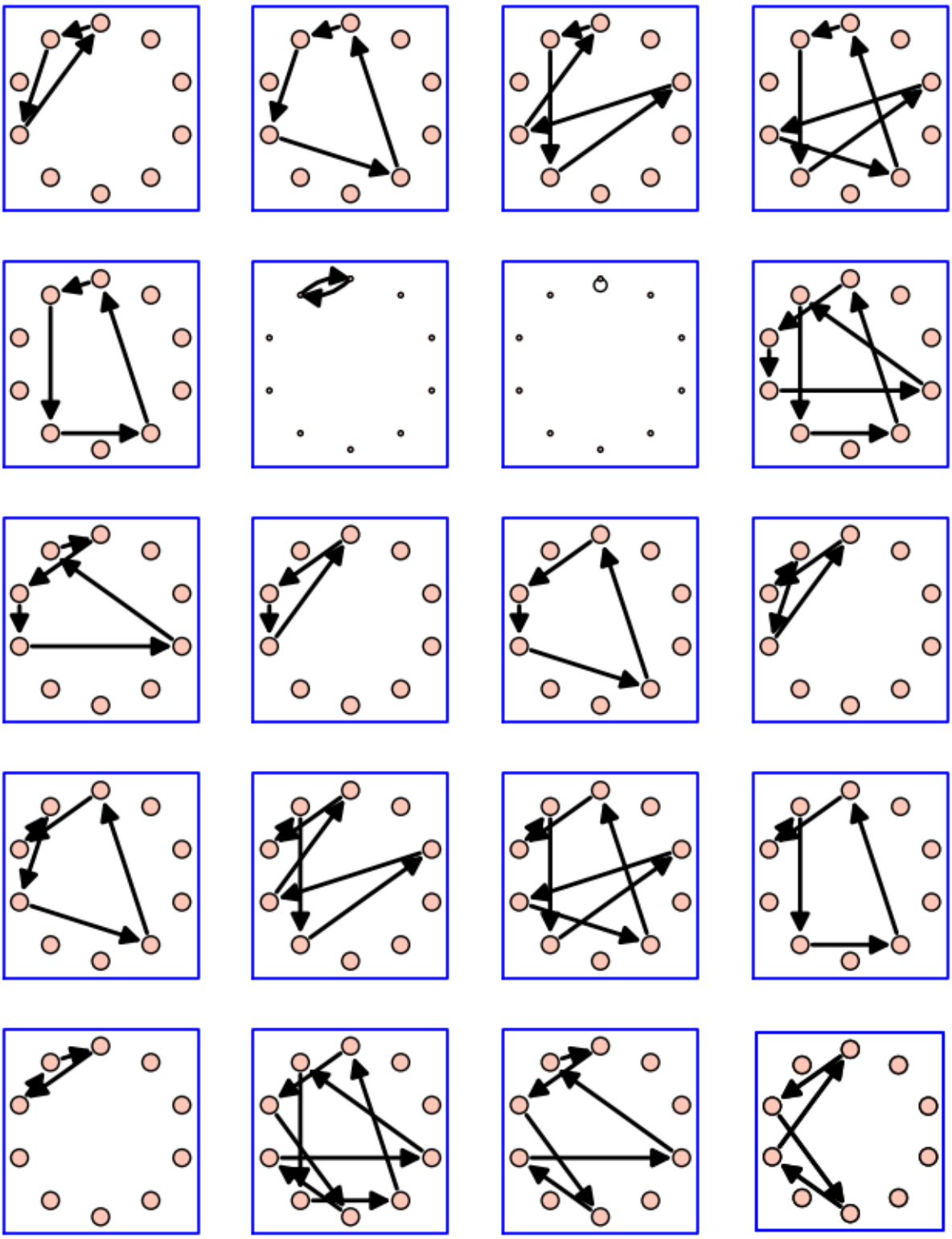
```
sage: attach(DATA + 'circuits.py')
sage: g = grafo_malabar(3,5)
sage: for ls in circuits(g):
```

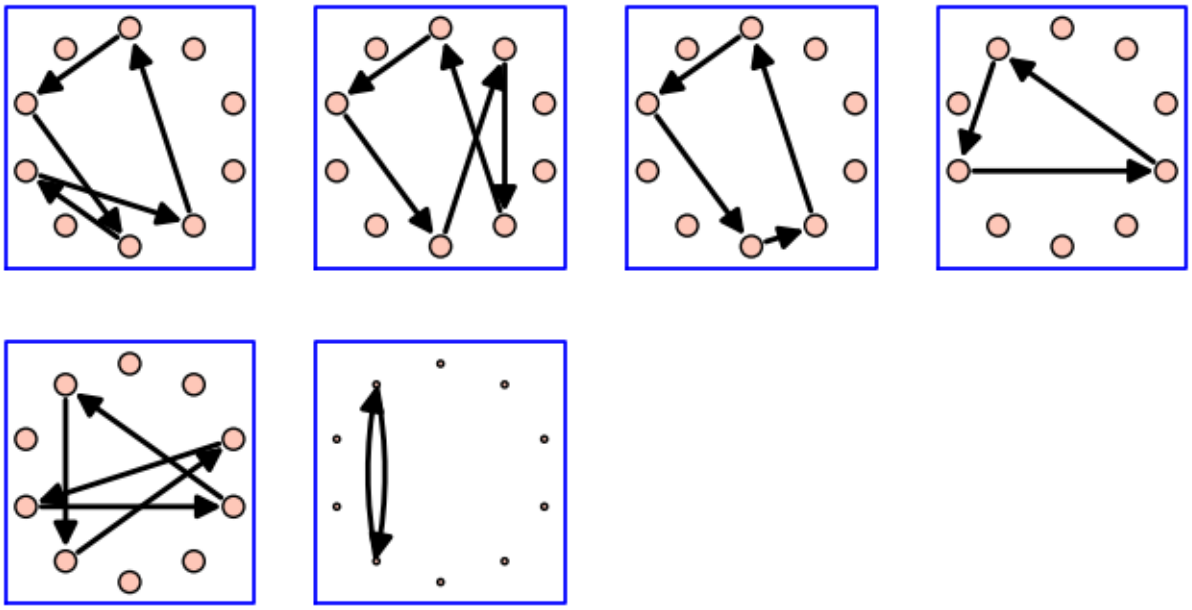
```
...       aristas = [(ls[j],ls[j+1])
...                  for j in range(len(ls)-1)]
...       print ''.join([str(g.edge_label(v1,v2)) for v1,v2 in aristas])
441
4440
45501
455040
4530
42
3
5350530
53502
531
5340
5241
52440
525501
5255040
52530
522
55150530
551502
5511
55140
55500
5520
450
55050
51
```

La misma información, de forma gráfica.

```
sage: g = grafo_malabar(3,5)
sage: show(g, edge_labels=True, layout='circular', figsize = 8, vertex_size=400)
sage: graphs_list.show_graphs([g.subgraph(edges = [(ls[j],ls[j+1])
...                                                for j in range(len(ls)-1)])
...                            for ls in circuits(g)])
```



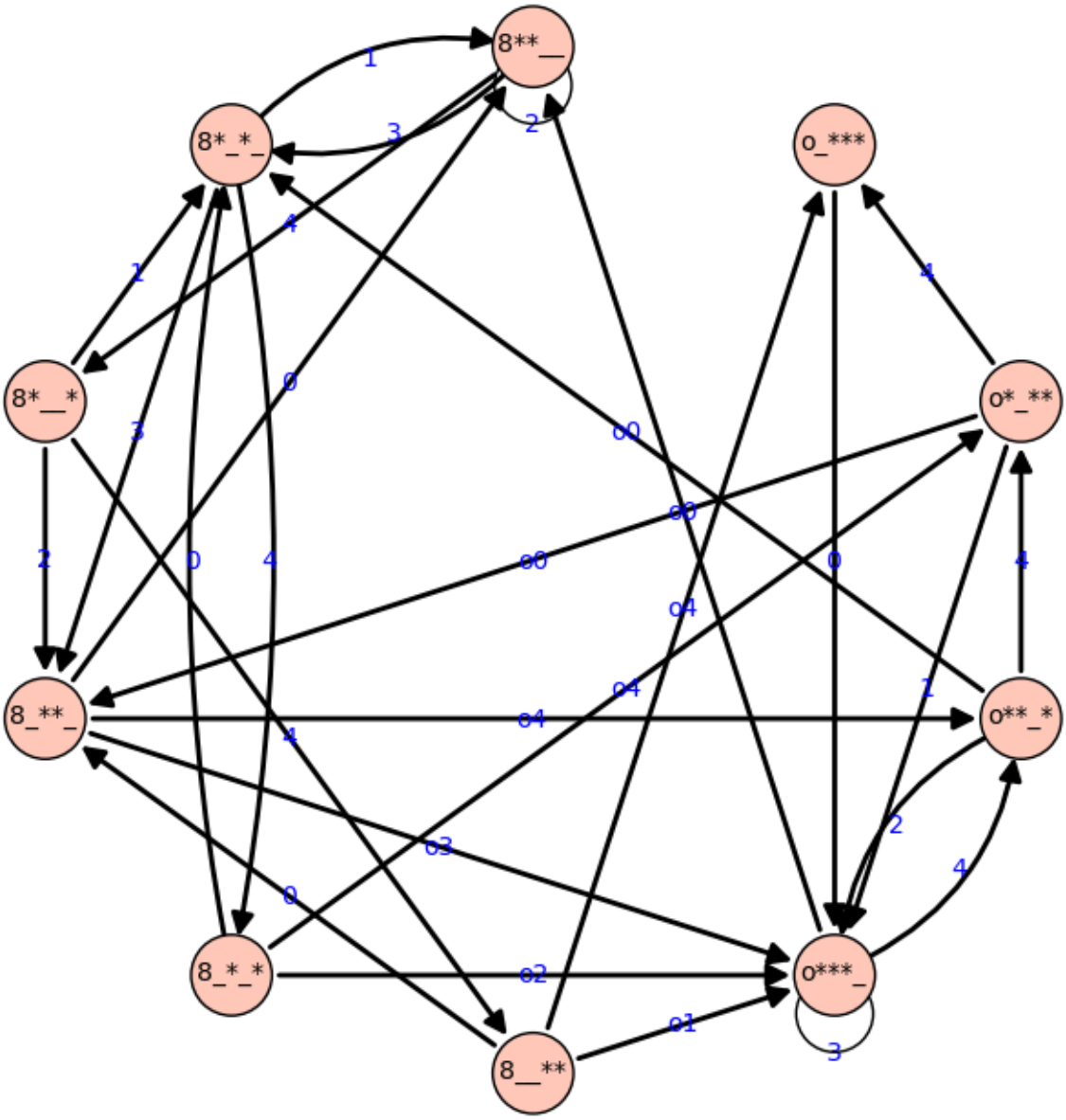


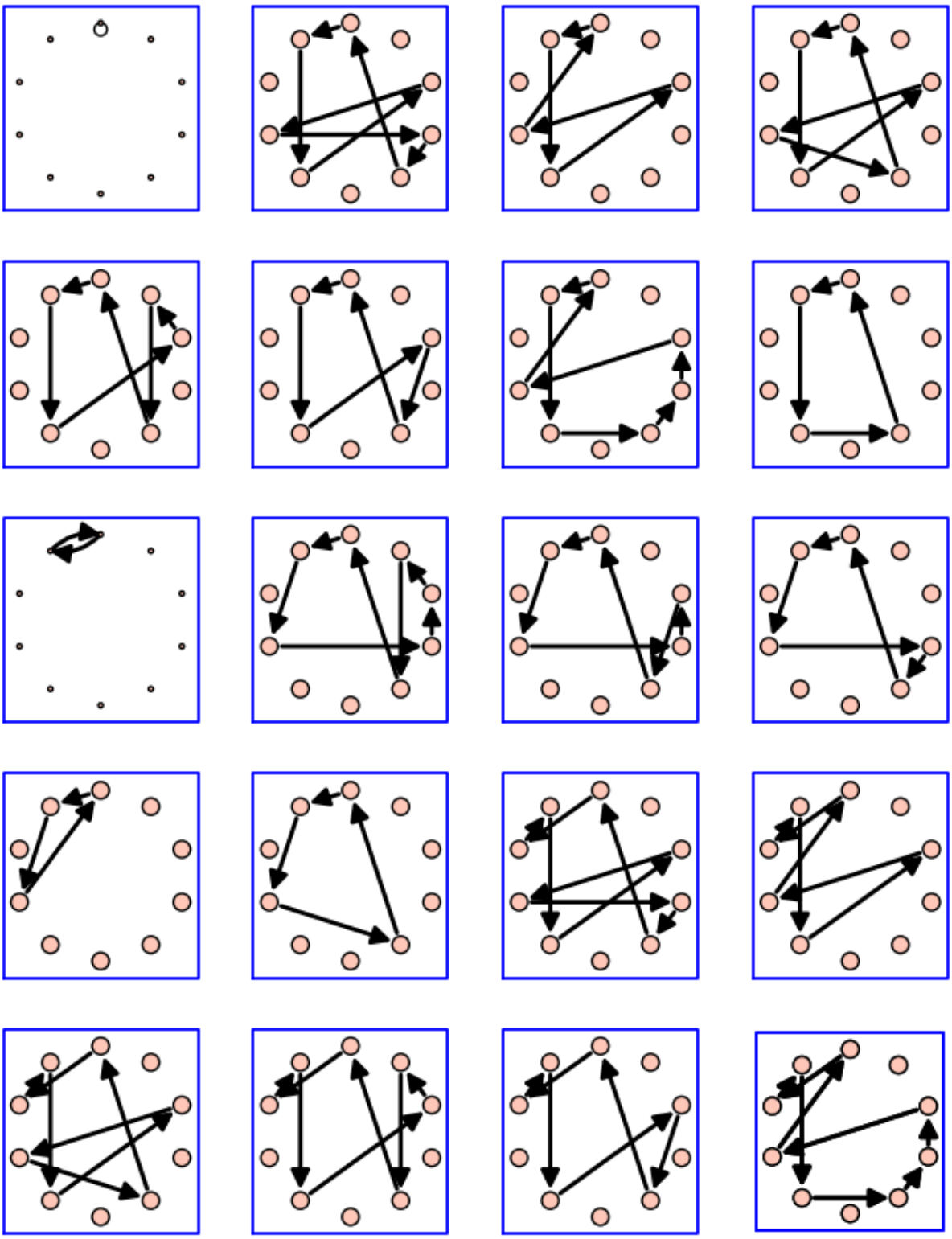


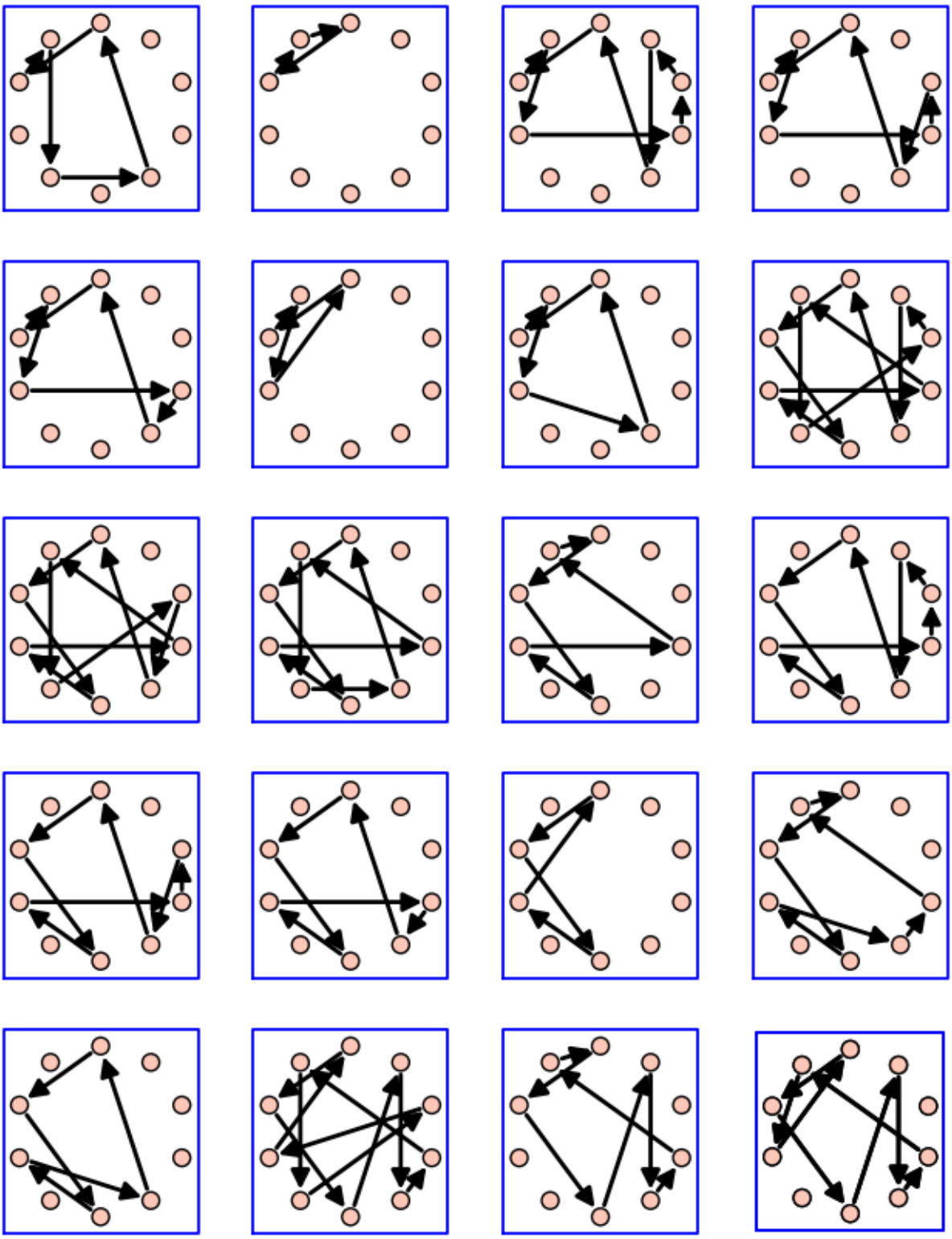
Finalmente, la misma información para el grafo de malabares con cabeza

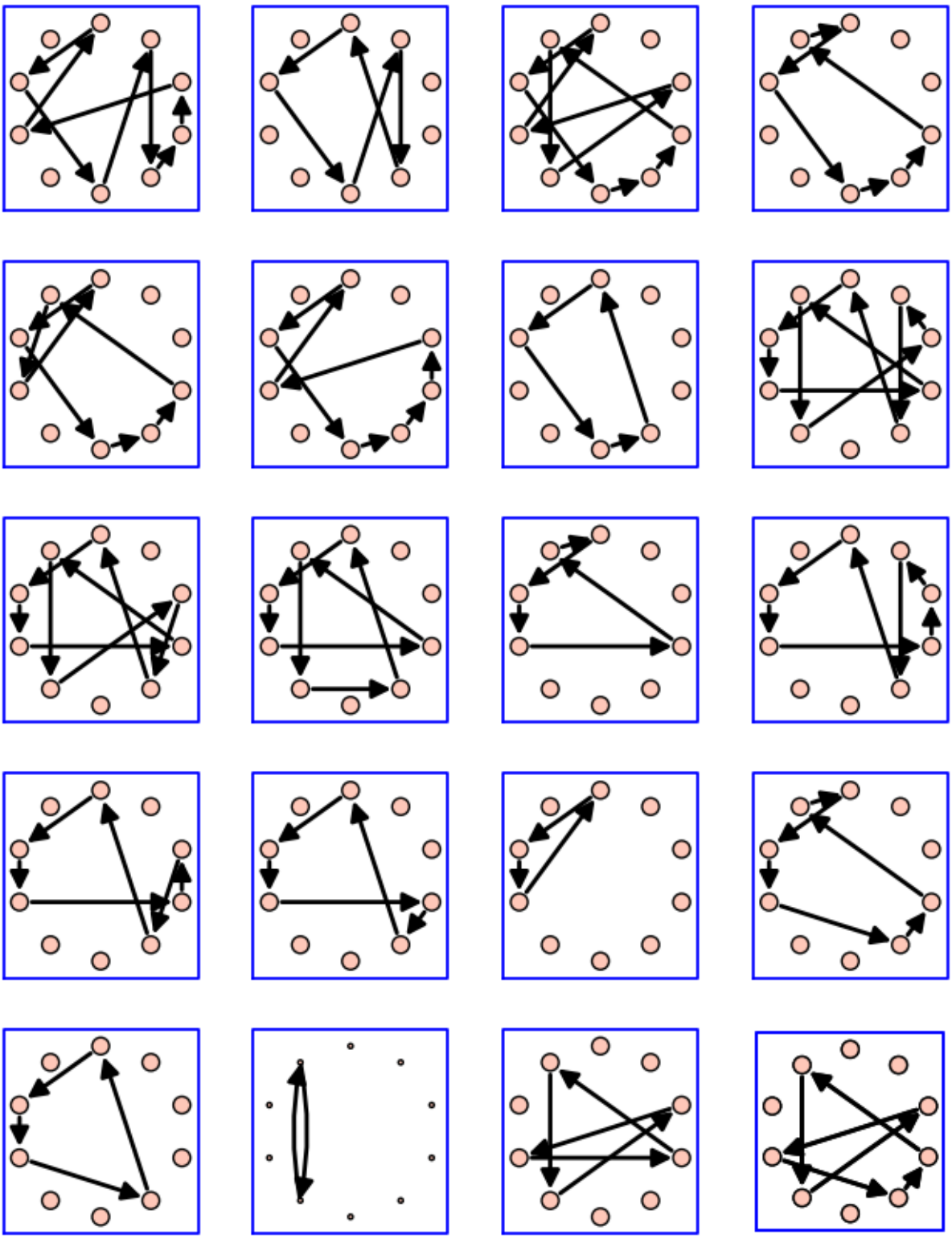
```
sage: g = grafo_malabar_con_cabeza(3,4)
sage: for ls in circuits(g):
...       aristas = [(ls[j],ls[j+1])
...                  for j in range(len(ls)-1)]
...       print ','.join([str(g.edge_label(v1,v2)) for v1,v2 in aristas])
...
sage: show(g, edge_labels=True, layout='circular',
...           figsize = 8, vertex_size=400)
sage: subgrafos = [g.subgraph(edges = [(ls[j],ls[j+1])
...                                    for j in range(len(ls)-1)])
...                for ls in circuits(g)]
sage: graphs_list.show_graphs(subgrafos)
2
3,4,o4,o0,o4,2,o0
3,4,o4,o0,0
3,4,o4,o0,o3,o0
3,4,o4,4,0,o0
3,4,o4,1,o0
3,4,o2,4,4,o0,0
3,4,o2,o0
3,1
3,3,o4,4,4,0,o0
3,3,o4,4,1,o0
3,3,o4,2,o0
3,3,0
3,3,o3,o0
4,1,4,o4,o0,o4,2,o0
4,1,4,o4,o0,0
4,1,4,o4,o0,o3,o0
4,1,4,o4,4,0,o0
4,1,4,o4,1,o0
4,1,4,o2,4,4,o0,0
4,1,4,o2,o0
4,1,1
```

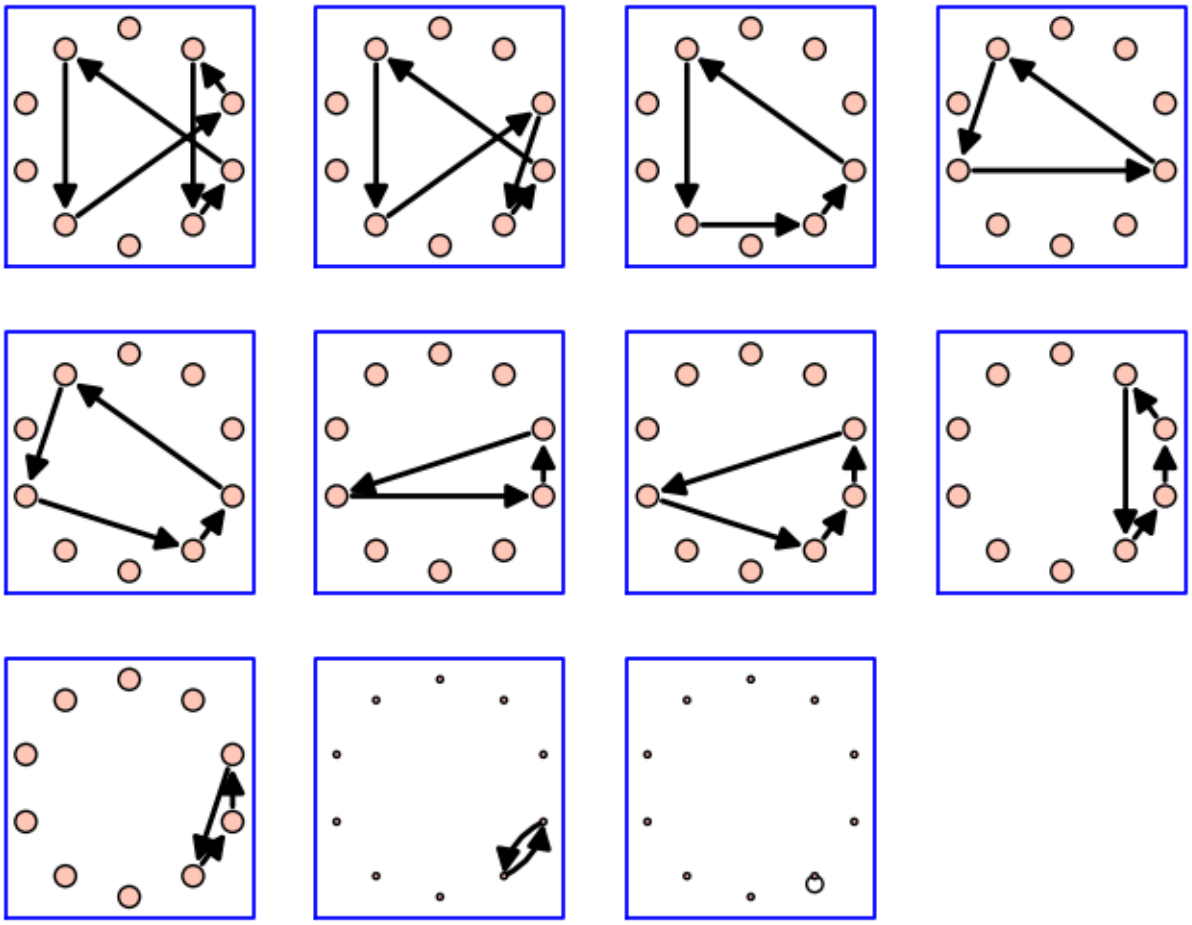
```
4,1,3,o4,4,4,0,o0
4,1,3,o4,4,1,o0
4,1,3,o4,2,o0
4,1,3,0
4,1,3,o3,o0
4,4,0,o4,o0,4,o4,4,0,o0
4,4,0,o4,o0,4,o4,1,o0
4,4,0,o4,o0,4,o2,o0
4,4,0,o4,o0,1
4,4,0,o4,4,4,0,o0
4,4,0,o4,4,1,o0
4,4,0,o4,2,o0
4,4,0,0
4,4,0,o3,4,o0,1
4,4,0,o3,o0
4,4,o4,0,4,o0,4,o4,o0,0
4,4,o4,0,4,o0,1
4,4,o4,0,4,o0,3,0
4,4,o4,0,4,4,o0,0
4,4,o4,0,o0
4,4,o1,4,o0,4,o4,o0,0
4,4,o1,4,o0,1
4,4,o1,4,o0,3,0
4,4,o1,4,4,o0,0
4,4,o1,o0
4,2,o4,o0,4,o4,4,0,o0
4,2,o4,o0,4,o4,1,o0
4,2,o4,o0,4,o2,o0
4,2,o4,o0,1
4,2,o4,4,4,0,o0
4,2,o4,4,1,o0
4,2,o4,2,o0
4,2,0
4,2,o3,4,o0,1
4,2,o3,o0
4,0
4,o4,o0,o4,o0
4,o4,o0,o3,4,o0
4,o4,4,0,4,o0
4,o4,1,4,o0
4,o2,4,o0
3,o4,o0
3,o3,4,o0
o4,4,o0
o3,4,4,o0
4,4,4,0
4,4,1
4,2
3
```


8**__
8*_*_
o_***
8*__*
o*_**
8_**_
o**_*
8_*_*
o***_
8__**









7.8.7 Ejercicios

- Escribe el grafo de malabares con dos pelotas de un color y una tercera de otro color y altura máxima 4. Si la tercera pelota es una manzana, identifica los estados en los que puedes pegarle un mordisco.
- Pepito está aprendiendo a hacer malabares con cuchillos. Se ve capaz de lanzar y recibir los cuchillos a altura 5 pero no quiere lanzar dos cuchillos seguidos tan alto por si se hace un lío al recogerlos. Modifica el grafo de tres objetos y altura 5 para evitar lanzar dos objetos a altura 5 seguidos.

Dos malabaristas pueden hacer malabares juntos en modo *síncrono* o *canon*. Cada malabarista puede pasarle cada pelota al otro o lanzarla para recogerla ella misma.

En modo síncrono, en cada instante, ambos malabaristas actúan a la vez (cada uno puede hacer un lanzamiento distinto, por supuesto). En modo *canon*, los tiempos de uno y otro malabarista no están sincronzados, sino que tienen un retraso de medio tiempo). Por ejemplo, con dos malabaristas, la secuencia es "Malabarista 1 mano I, Malabarista 2 mano I, Malabarista 1 mano D, Malabarista 2 mano D".

Tienes más abajo la construcción del grafo de n malabaristas en modo síncrono.

- Genera el grafo de dos malabaristas y un total de 5 bolas trabajando en modo canon, donde cada malabarista puede enviarse la pelota a sí mismo, para recogerla dentro de a lo sumo tres instantes, o al otro malabarista, con la misma limitación.

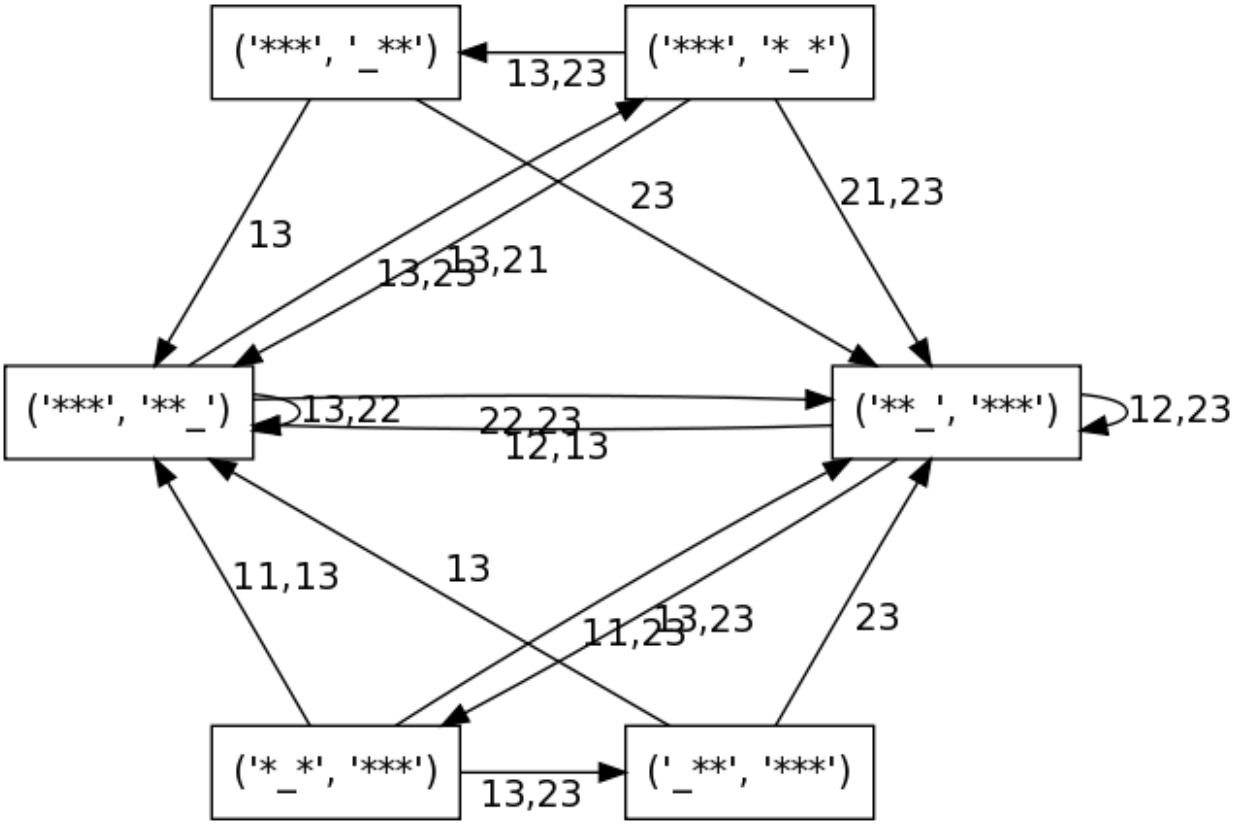
Cuestiones

- ¿Cuántos circuitos hamiltonianos tiene el grafo_malabar_con_cabeza(3,4)?
- Identifica en cada grafo el estado o estado "fundamentales", que debe verificar dos propiedades: que sea fácil alcanzar ese estado empezando con todas las bolas en la mano, y que se pueda mantener de forma indefinida.

7.8.8 Apéndice: varios malabaristas

Grafo de varios malabaristas que tienen todos la misma altura como cota y con una cantidad dada de bolas en total. Observa que si en un instante varios malabaristas deben lanzar una bola, sólo anotamos los destinos de esas bolas, sin distinguir quién hace qué lanzamiento. Es decir, que no distinguimos si cada malabarista se lanza a sí mismo o si cada uno le lanza al otro. Representamos cada estado mediante la unión de las listas de espera de todos los malabaristas, y cada arista mediante pares que dicen el malabarista que debe recibir y la altura del lanzamiento.

```
sage: def opcionesm(estado):
...       estado1 = tuple(malabarista[1:] + '_' for malabarista in estado)
...       bolas = sum(1 if malabarista[0]=='*' else 0
...                   for malabarista in estado)
...       if not bolas:
...           return {estado1:'0'}
...       huecos = [(i, j)
...                 for i,malabarista in enumerate(estado1)
...                 for j,s in enumerate(malabarista)
...                 if s=='_']
...       opciones = {}
...       for objetivos in Combinations(huecos, bolas):
...           nuevo_estado = list(estado1)
...           for i, j in objetivos:
...               cadena = nuevo_estado[i]
...               nuevo_estado[i] = cadena[:j] + '*'+ cadena[j+1:]
...           opciones[tuple(nuevo_estado)] = ','.join('%d%d'%(i+1,j+1)
...                                               for i, j in objetivos)
...       return opciones
sage: def grafo_malabarm(bolas, altura, malabaristas):
...       '''Crea el grafo de malabares para varios malabaristas
...
...       Acepta como argumento el numero de bolas y la lista de alturas
...       máximas de cada malabarista'''
...       total = altura*malabaristas
...       cadenas =  [ ''.join(('*' if j in ss else '_') for j in range(total))
...                    for ss in Subsets(range(total), bolas) ]
...       parte =lambda c: tuple(c[j:j+altura]
...                              for j in range(0, total, altura))
...       estadosm = [parte(c) for c in cadenas]
...       transiciones = dict((estado,opcionesm(estado)) for estado in estadosm)
...       return DiGraph(transiciones)
...

sage: #5 bolas, dos malabaristas a altura 3 cada uno
sage: g = grafo_malabarm(5, 3, 2)
sage: graphviz(g)
```

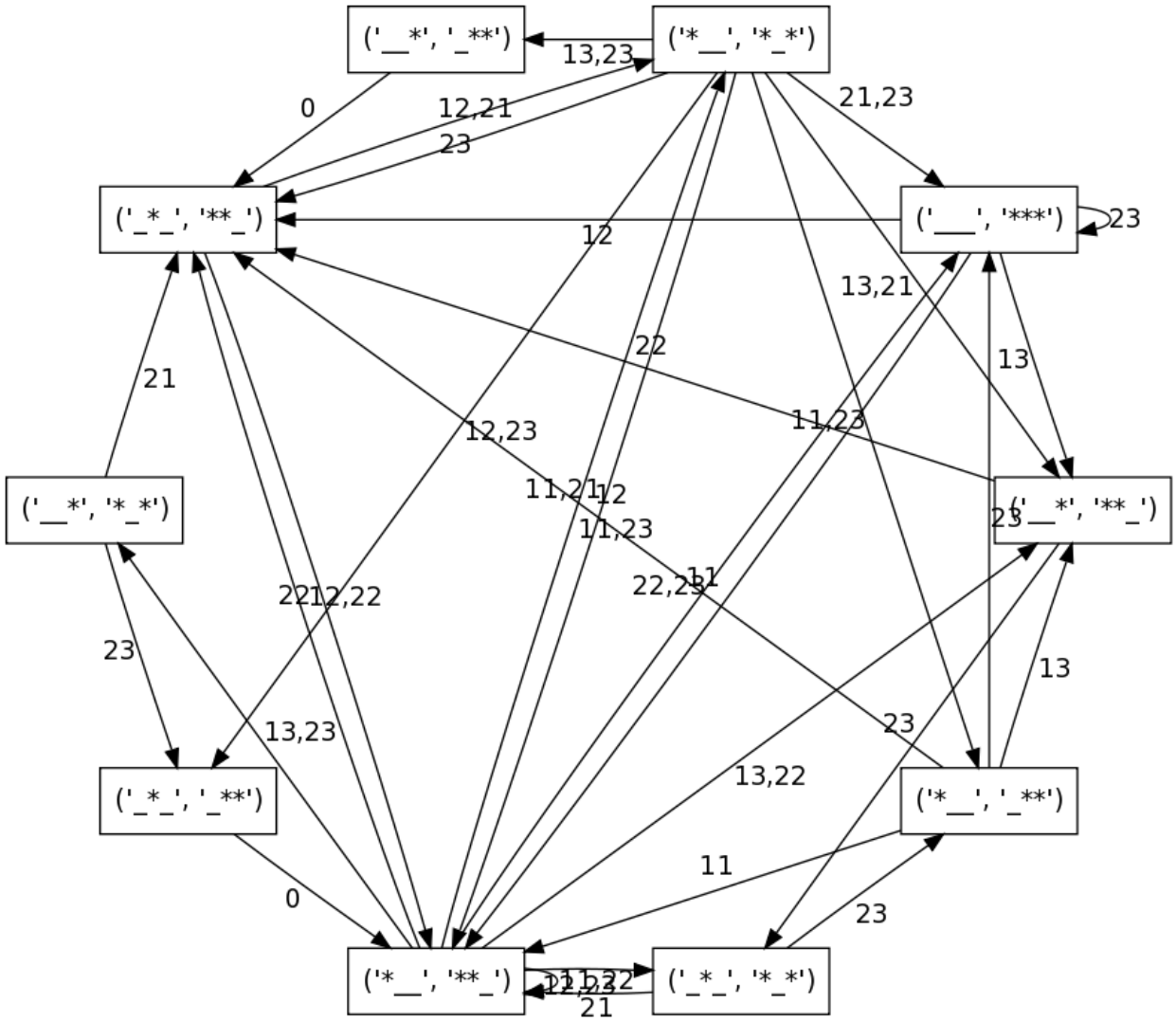


Coda: un malabarista y un inútil

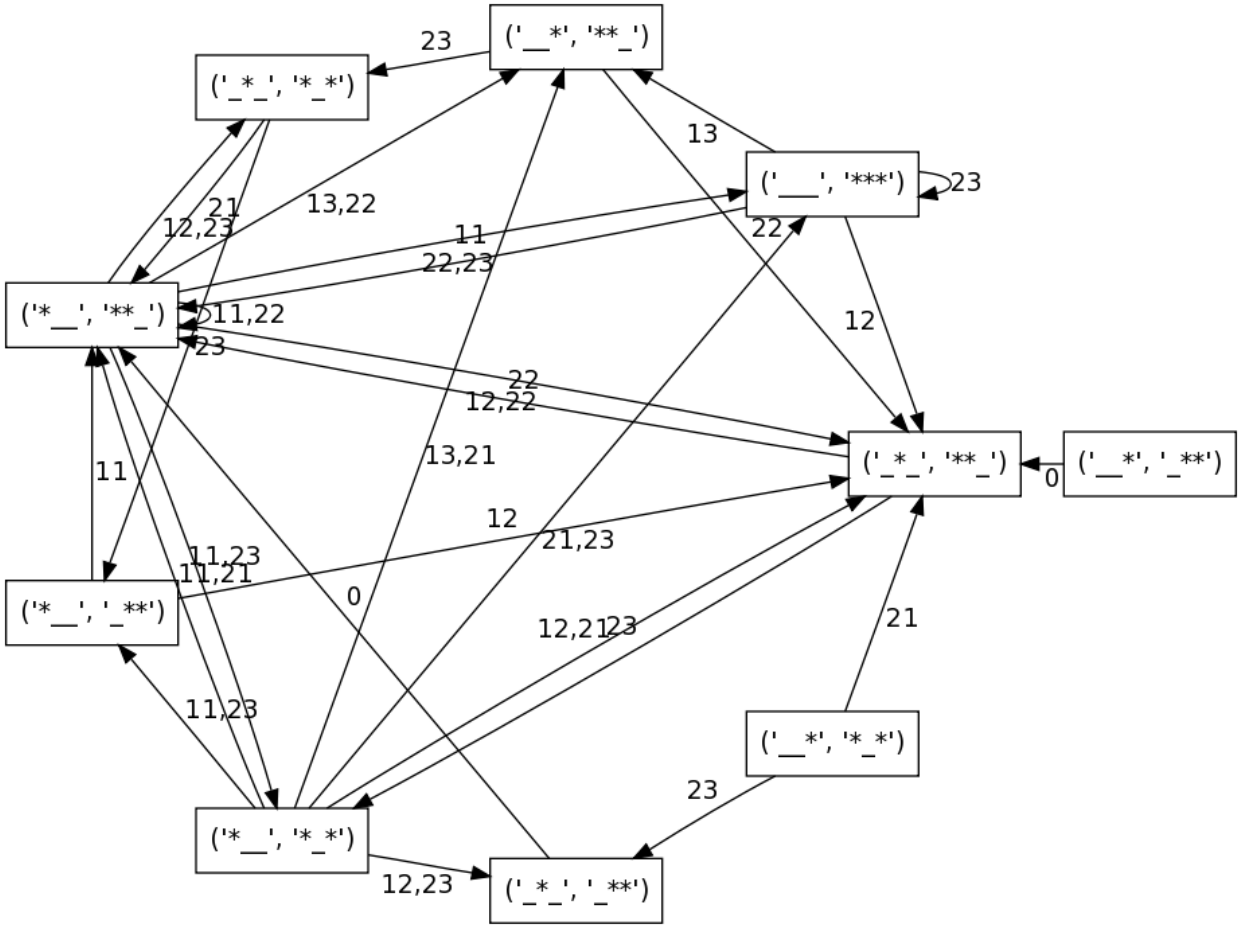
¿Podemos encontrar un truco en el que un inútil hace malabares con un malabarista?

- el inútil no puede tener más de una bola en su lista de caída
- el inútil no puede lanzar a altura mayor que 2

```
sage: #3 bolas, el malabarista trabaja a altura 3, el inutil a altura 2
sage: g = grafo_malabarm(3,3,2)
sage: #Añadimos la restricción de que el inútil
sage: #sólo puede tener una bola en la mano
sage: cota_bolas = 1
sage: vs = [v for v in g.vertices()
...           if sum(1 for c in v[0] if c=='*')<=cota_bolas ]
sage: sg = g.subgraph(vertices = vs)
sage: graphviz(sg)
```

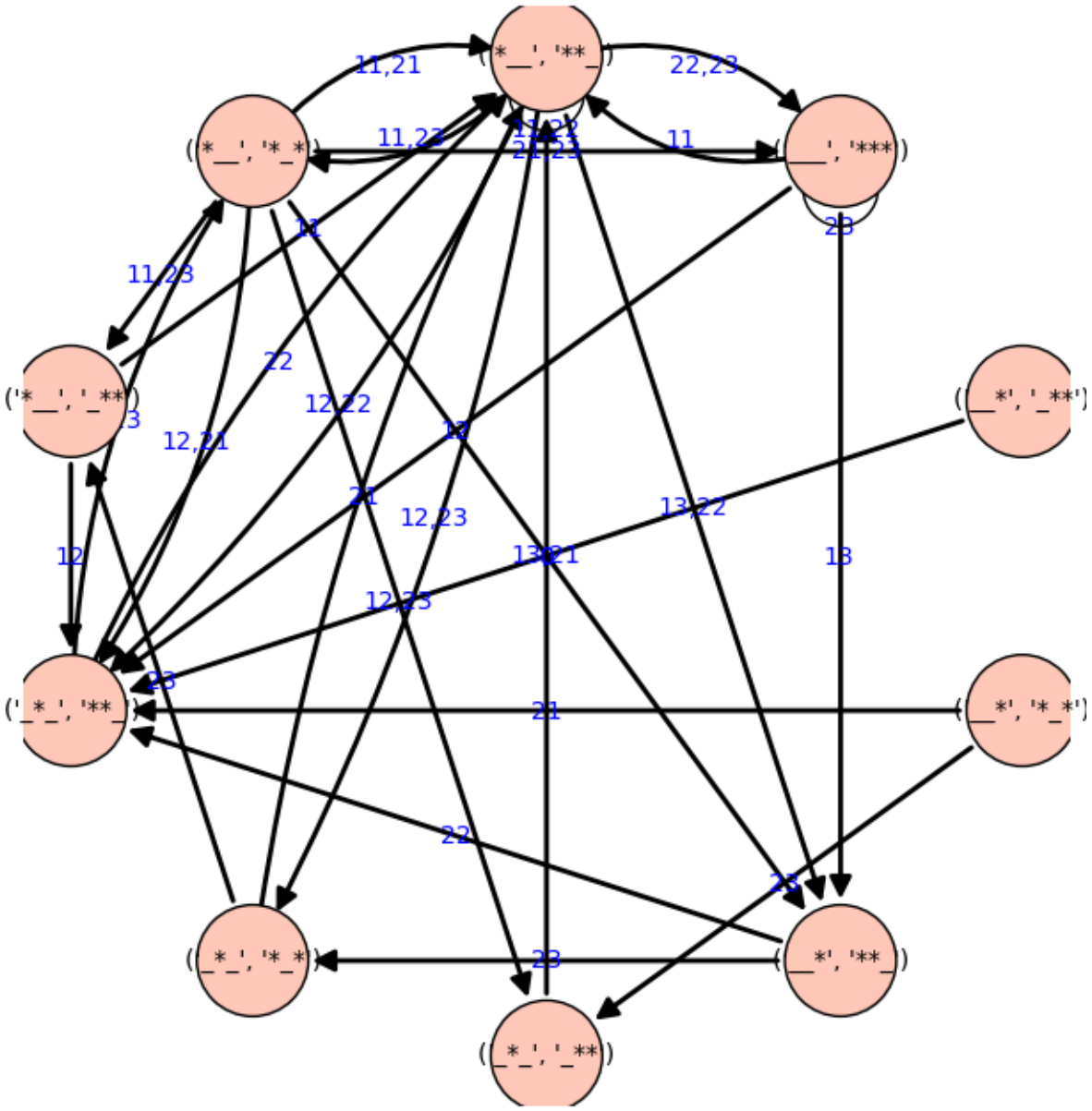


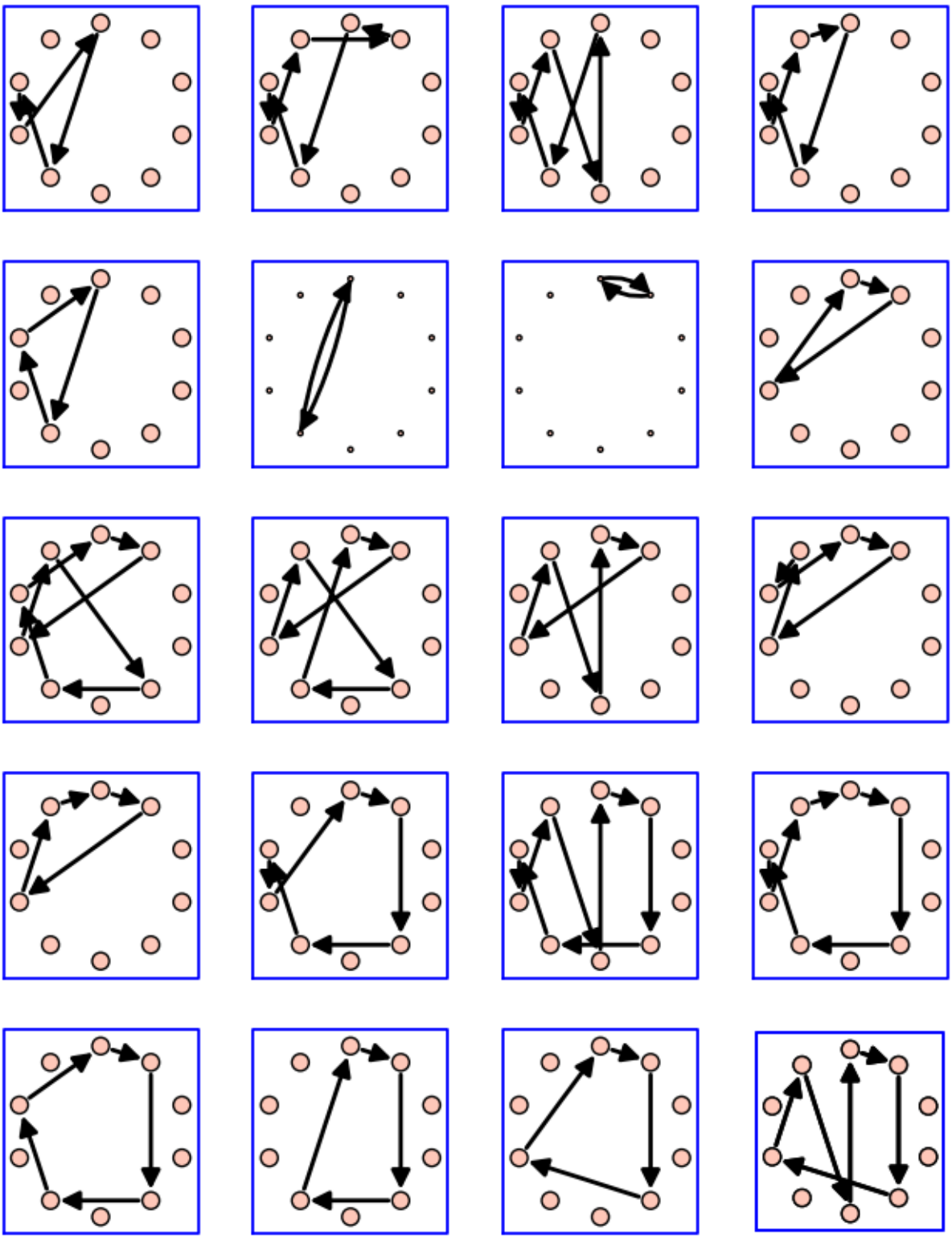

```
sage: #Añadimos la restricción de que el inútil
sage: #sólo puede lanzar a altura menor o igual que dos
sage: cota_altura = 2
sage: def criterio(arista):
...       '''Decide si una arista exige que el primer malabarista
...       tenga que lanzar mas alto que su cota
...       '''
...       e1,e2,lanzamientos = arista
...       m1, m2 = e1
...       if m1[0] == '_':
...           return True
...       if m2[0] == '_':
...           return int(lanzamientos[1]) <= cota_altura
...       return (int(lanzamientos[1]) <= cota_altura or
...               int(lanzamientos[4]) <= cota_altura)
sage: aristas = [arista for arista in g.edges()
...                     if criterio(arista) ]
sage: sg2 = sg.subgraph(edges = aristas)
sage: graphviz(sg2)
```

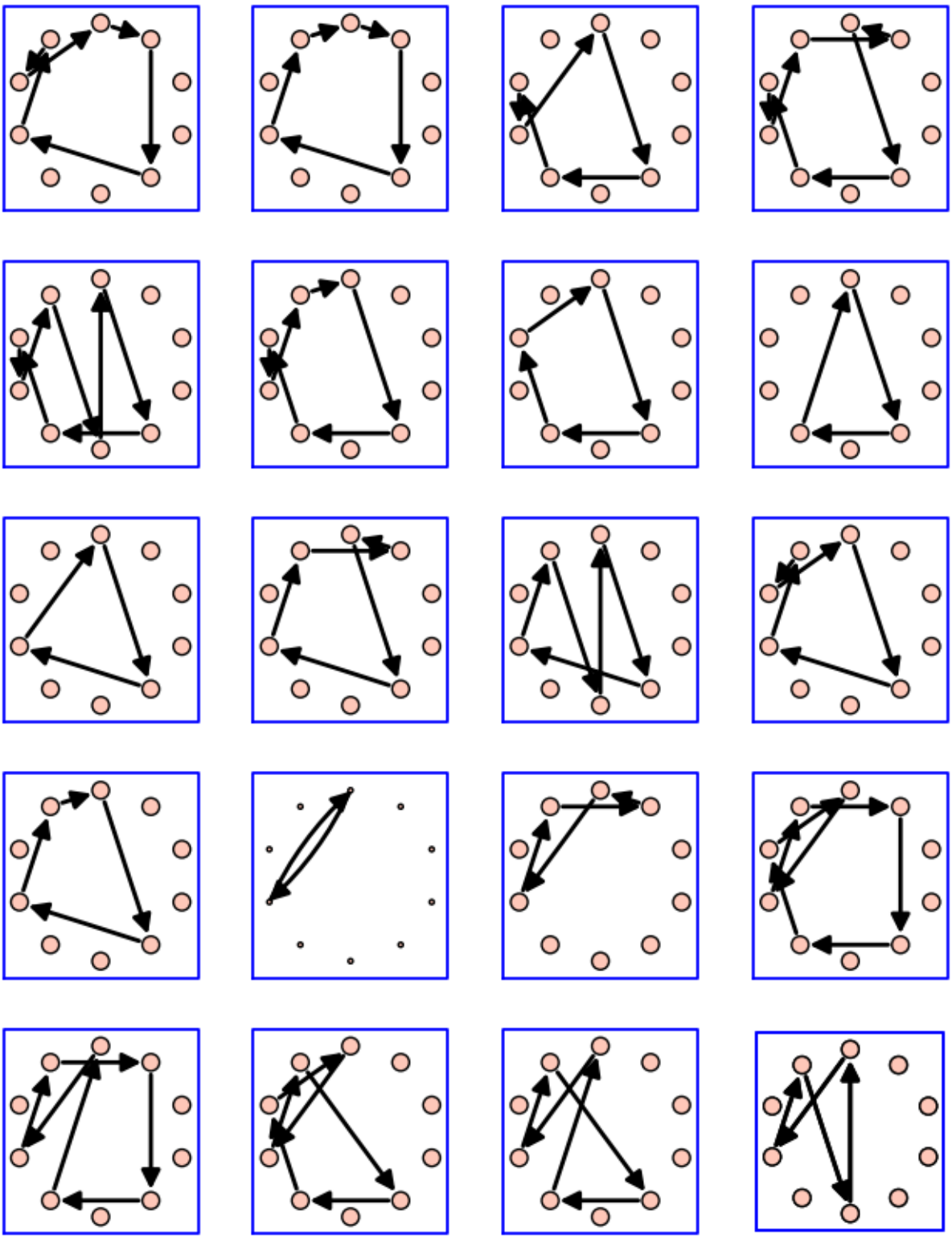


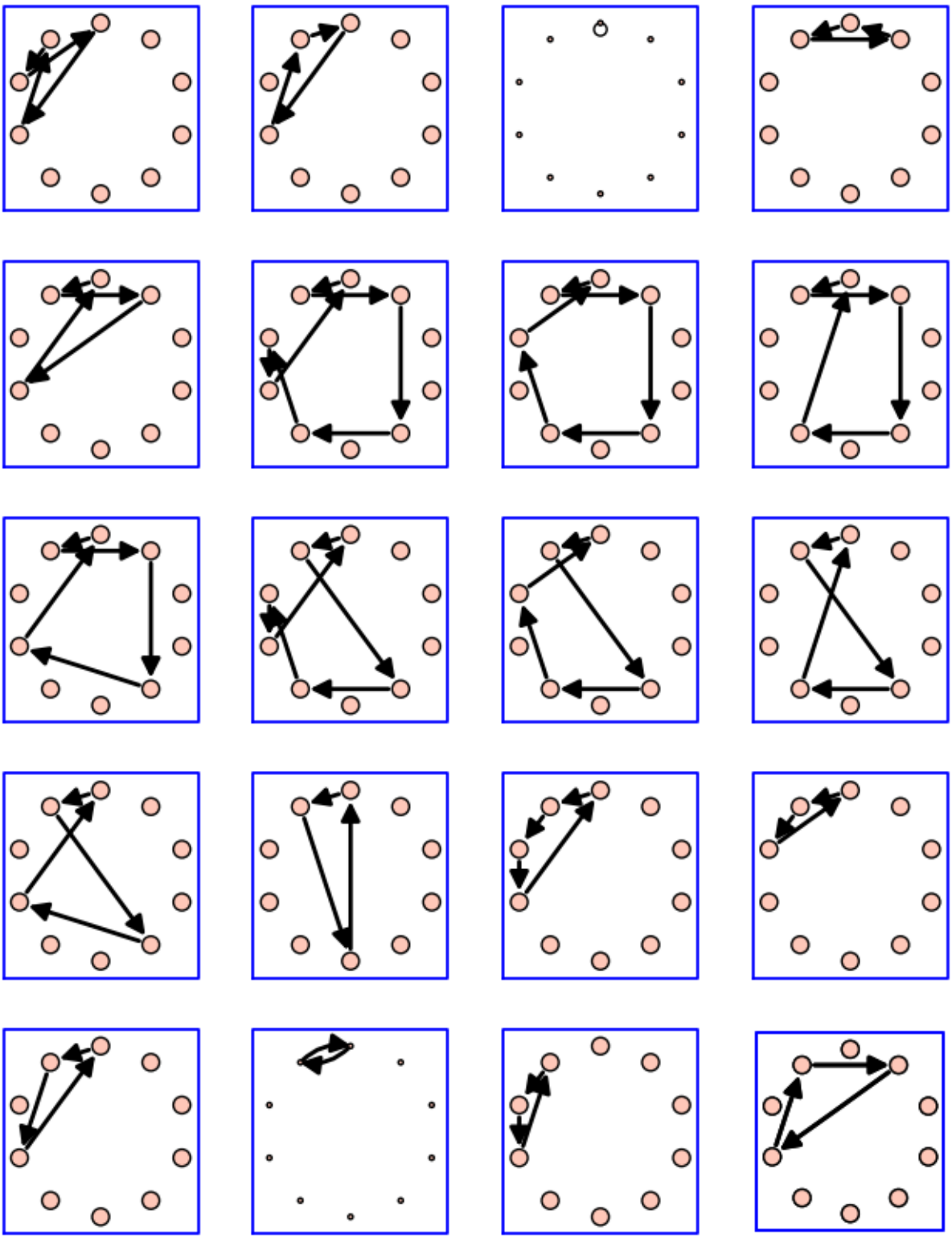

```
sage: #Todos los circuitos
sage: g = sg2
sage: for ls in circuits(g):
...       aristas = [(ls[j],ls[j+1])
...                  for j in range(len(ls)-1)]
...       print ';'.join([str(g.edge_label(v1,v2)) for v1,v2 in aristas])
...
sage: show(g, edge_labels=True, layout='circular',
...           figsize = 8, vertex_size=800)
sage: subgrafos = [g.subgraph(edges = [(ls[j],ls[j+1])
...                                    for j in range(len(ls)-1)])
...                for ls in circuits(g)]
sage: graphs_list.show_graphs(subgrafos)
12,23;23;12;22
12,23;23;12;23;21,23;11
12,23;23;12;23;12,23;0
12,23;23;12;23;11,21
12,23;23;11
12,23;21
22,23;11
22,23;12;22
22,23;12;23;13,21;23;23;11
22,23;12;23;13,21;23;21
22,23;12;23;12,23;0
22,23;12;23;11,23;11
22,23;12;23;11,21
```

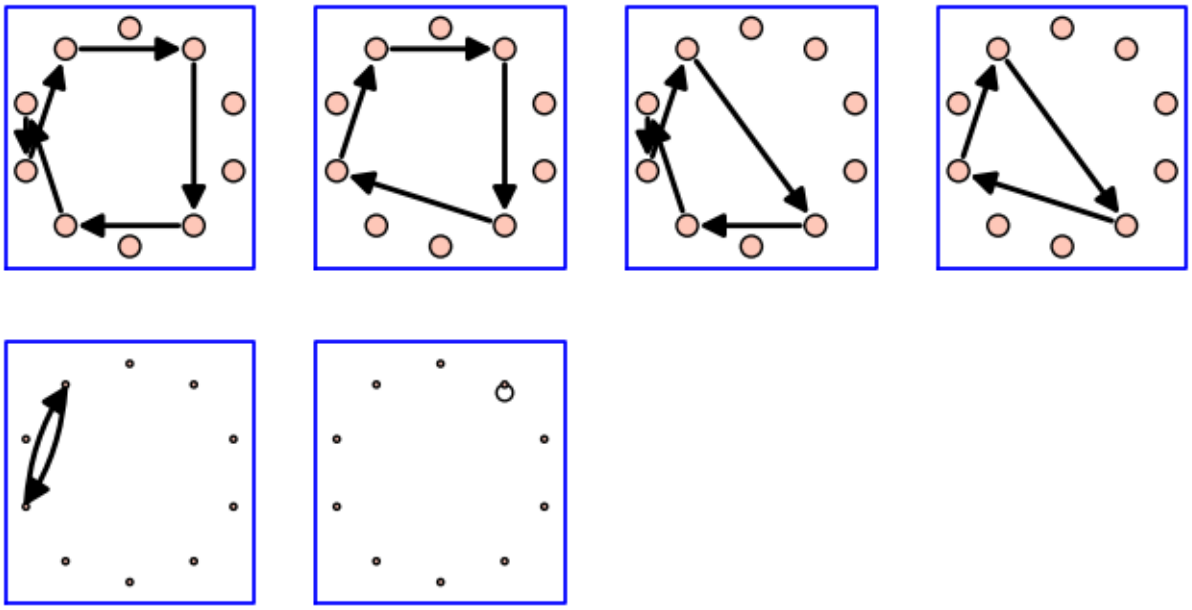
```
22,23;13;23;23;12;22
22,23;13;23;23;12;23;12,23;0
22,23;13;23;23;12;23;11,21
22,23;13;23;23;11
22,23;13;23;21
22,23;13;22;22
22,23;13;22;23;12,23;0
22,23;13;22;23;11,23;11
22,23;13;22;23;11,21
13,22;23;23;12;22
13,22;23;23;12;23;21,23;11
13,22;23;23;12;23;12,23;0
13,22;23;23;12;23;11,21
13,22;23;23;11
13,22;23;21
13,22;22;22
13,22;22;23;21,23;11
13,22;22;23;12,23;0
13,22;22;23;11,23;11
13,22;22;23;11,21
12,22;22
12,22;23;21,23;11
12,22;23;21,23;13;23;23;11
12,22;23;21,23;13;23;21
12,22;23;13,21;23;23;11
12,22;23;13,21;23;21
12,22;23;12,23;0
12,22;23;11,23;11
12,22;23;11,21
11,22
11,23;21,23;11
11,23;21,23;12;22
11,23;21,23;13;23;23;12;22
11,23;21,23;13;23;23;11
11,23;21,23;13;23;21
11,23;21,23;13;22;22
11,23;13,21;23;23;12;22
11,23;13,21;23;23;11
11,23;13,21;23;21
11,23;13,21;22;22
11,23;12,23;0
11,23;11,23;12;22
11,23;11,23;11
11,23;12,21;22
11,23;11,21
11,23;12;23
21,23;12;23
21,23;13;23;23;12;23
21,23;13;22;23
13,21;23;23;12;23
13,21;22;23
12,21;23
23
```


('*__', '**_')
('*__', '*_*')
('___', '***')
('*__', '_**')
('___*', '_**')
('_*_', '**_')
('___*', '*_*')
('_*_', '*_*')
('___*', '**_')
('_*_', '_**')
11,21
22,23
11,23
11
11,23
22
12,22
12,21
12,23
13,22
12
13,21
13
21
22
23

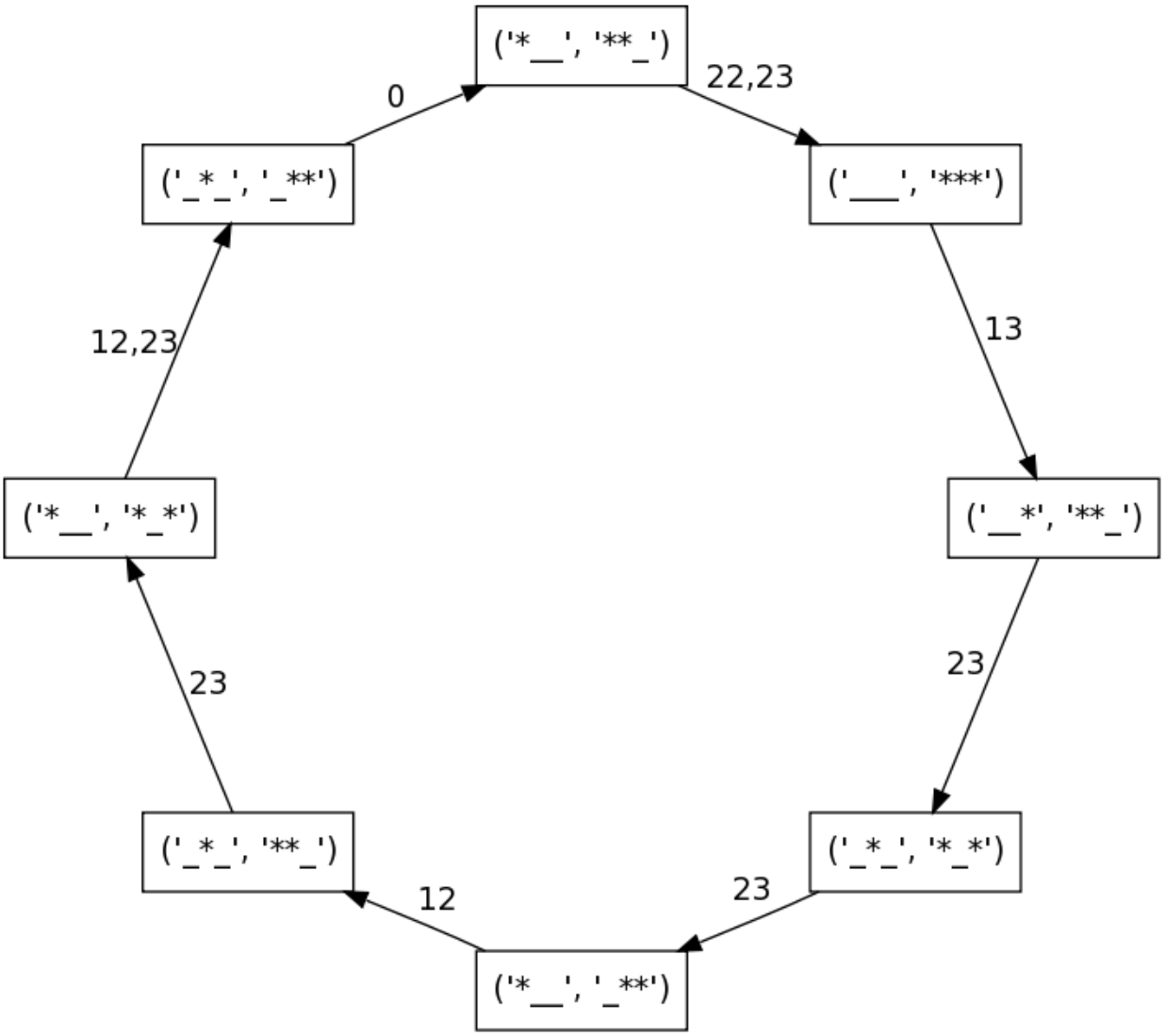








```
sage: #El circuito mas largo
sage: g = sg2
sage: L, circuito = max((len(ls), ls) for ls in circuits(g))
sage: subgrafo = g.subgraph(edges = [(circuito[j],circuito[j+1])
...                                  for j in range(len(circuito)-1)])
sage: graphviz(subgrafo)
```



7.8.9 Créditos

Esta presentación usa el programa Sage para hacer los cálculos sobre grafos y la librería graphviz para dibujar los grafos.

Muchas gracias a Daniel Sánchez , y a Nicolas M. Thiéry y Florent Hivert por sus comentarios (y a Clara, por supuesto).

7.8.10 Licencia

7.9 Inflar el PageRank controlando los links que salen de tu página

En este apéndice vamos a aprender el efecto que tienen los " link farms " o "mutual admiration societies" en el índice de google. Por supuesto hay otros métodos de *link spam* más clásicos, pero estudiamos éste método en concreto aprovechando lo que aprendimos sobre cadenas de Markov al hablar de grafos.

7.9.1 Calcular el ranking de google

El PageRank es un algoritmo abstracto que permite dar un valor numérico (**ranking**) a cada nodo de un grafo que mide de alguna forma su *conectividad* . Cuantos más links tiene una página web, mayor es su ranking.

Podemos interpretar este ranking de la siguiente forma: cada página web es un nodo de un grafo dirigido, que tiene una arista desde la página A hasta la página B sii en A hay un link que apunta a B. Comenzamos un camino aleatorio en un nodo aleatorio del grafo, escogido con igual probabilidad. Con probabilidad d (el **damping factor**), saltamos a uno de sus enlaces, con igual probabilidad. Con probabilidad 1-d, comenzamos desde cero. Hemos construido una **cadena de Markov** , que a largo plazo tiene una *distribución de probabilidad estable* que tomamos como ranking de cada nodo: el ranking es la proporción del tiempo que pasamos en ese nodo en el camino aleatorio descrito arriba.

Para más información podéis leer:

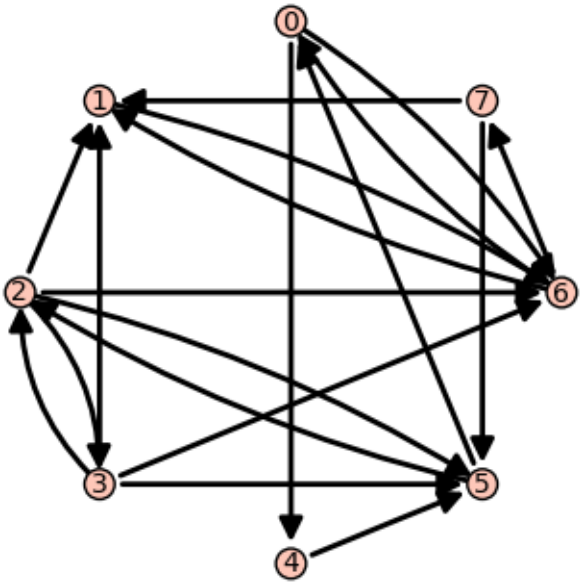
http://www.uam.es/personal_pdi/ciencias/gallardo/google.htm

```
sage: def ranking(g, damping = 0.85):
...       #M es la matriz estocastica asociada al grafo
...       M = matrix(RDF, [row/sum(row) for row in g.adjacency_matrix().rows()])
...       r = M.nrows()
...       #T tiene en cuenta el factor de "damping"
...       #o la probabilidad de abandonar el camino actual
...       #y volver a comenzar desde un nodo aleatorio
...       T = damping*M + matrix(RDF, r, [(1 - damping)/r]*(r^2))
...
...       #Una aproximacion practica al ranking: tomamos una fila
...       #cualquiera de una potencia grande de la matriz
...       distribucion_estable = (T^64).rows()[0]
...       return dict(zip(g.vertices(),distribucion_estable))
```

7.9.2 Caso práctico

Partimos de un grafo dirigido bastante arbitrario, calculamos el ranking de google de cada nodo, usando el damping por defecto de 0.15.

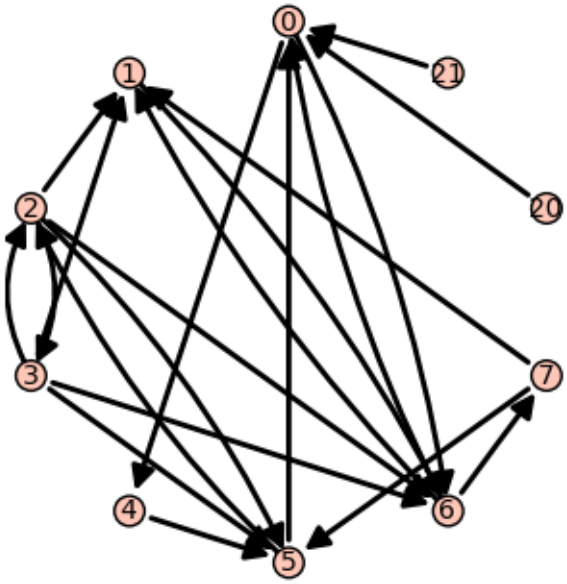
```
sage: edges = [(0, 4), (0, 6), (1, 6), (2, 1), (2, 3), (2, 5),
...            (2, 6), (3, 1), (3, 2), (3, 5), (3, 6), (4, 5),
...            (5, 0), (5, 2), (6, 0), (6, 1), (6, 7), (7, 1),
...            (7, 5)]
sage: g=DiGraph(edges)
sage: show(g, layout='circular')
sage: d = ranking(g)
sage: print [(p, v.n(digits = 3)) for p,v in d.items()]
[(0, 0.152), (1, 0.152), (2, 0.0926), (3, 0.0384), (4, 0.0835), (5, 0.154), (6, 0.240), (7, 0.0868)]
```



Crear páginas vacías que apuntan a la tuya

En un primer intento de inflar el ranking del nodo 0, creamos dos nodos, 20 y 21 que apuntan a 0. La táctica apenas es efectiva.

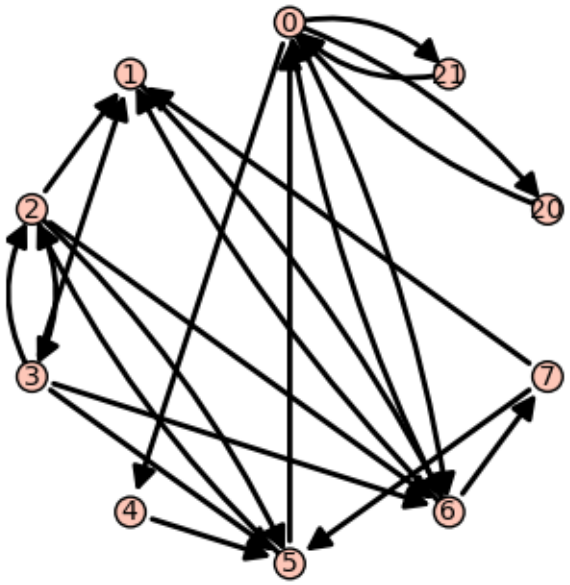
```
sage: g=DiGraph(edges)
sage: g.add_vertices([20, 21])
sage: g.add_edges([(20,0), (21,0)])
sage: g.show(layout='circular')
sage: d = ranking(g)
sage: print [(p, v.n(digits = 3)) for p,v in d.items()]
[(0, 0.168), (1, 0.139), (2, 0.0848), (3, 0.0330), (4, 0.0866), (5, 0.148), (6, 0.230), (7, 0.0802),
```



Crear circuitos cerrados que pasan por tu página

Sin embargo, si desde el vértice 0 tb apuntamos a 20 y 21, la cosa cambia. Ahora los caminos aleatorios que alcanzan 0 se quedan más tiempo orbitando no muy lejos de este nodo.

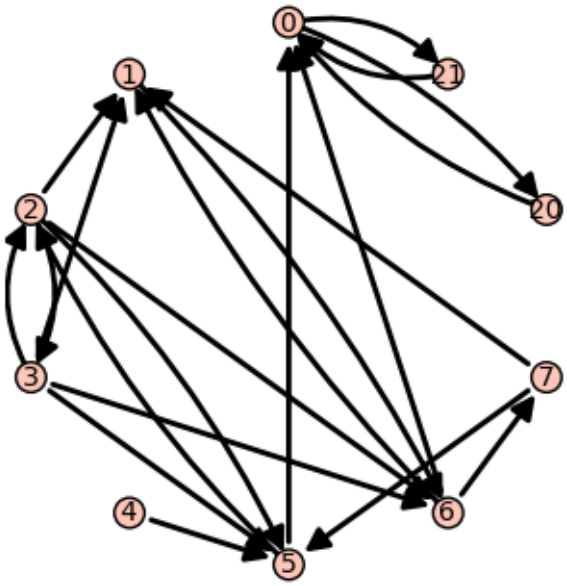
```
sage: g=DiGraph(edges)
sage: g.add_vertices([20, 21])
sage: g.add_edges([(0,20), (20,0), (0,21), (21,0)])
sage: g.show(layout='circular')
sage: d = ranking(g)
sage: print [(p, v.n(digits = 3)) for p,v in d.items()]
[(0, 0.224), (1, 0.117), (2, 0.0717), (3, 0.0302), (4, 0.0625), (5, 0.118), (6, 0.184), (7, 0.0671),
```



Secuestrar los caminos aleatorios que pasan por tu página

Siguiendo esta misma idea, *eliminamos los links "legítimos"* del nodo 0. Ahora todos los caminos aleatorios que llegan a 0 pasan su tiempo en circuitos muy cortos que contienen a 0. Los links del nodo 0 a los nodos 4 y 6 permitían que los caminos aleatorios *escapasen* de la órbita del 0, pero ahora la única manera de salir es que intervenga el factor de *damping* .

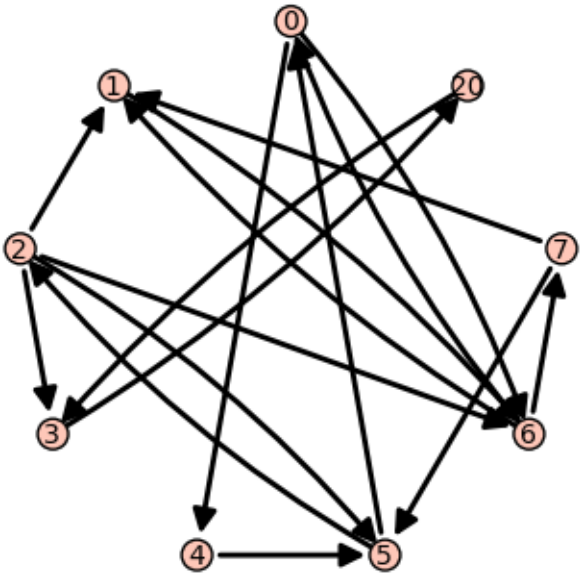
```
sage: g=DiGraph(edges)
sage: g.add_vertices([20, 21])
sage: g.add_edges([(0,20), (20,0), (0,21), (21,0)])
sage: g.delete_edges([(0,4), (0,6)])
sage: g.show(layout='circular')
sage: d = ranking(g)
sage: print [(p, v.n(digits = 3)) for p,v in d.items()]
[(0, 0.333), (1, 0.0738), (2, 0.0459), (3, 0.0248), (4, 0.0150), (5, 0.0603), (6, 0.0928), (7, 0.041
```



Lo mismo pero con un nodo peor conectado

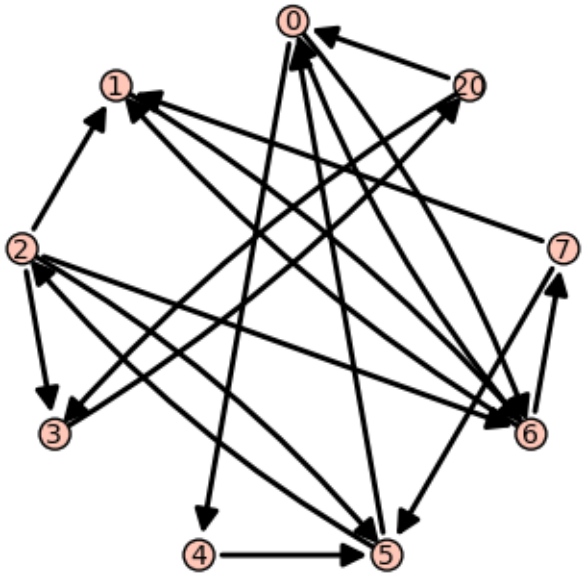
Esta táctica aumenta el ranking de cualquier nodo, pero si era un nodo muy poco conectado tampoco hace milagros. En este caso usamos sólo un nodo extra, el 20, que es más efectivo, y el ranking pasa de 0.04 a 0.16.

```
sage: g=DiGraph(edges)
sage: g.add_vertices([20])
sage: g.add_edges([(3,20), (20,3)])
sage: g.delete_edges([(3,2), (3,1), (3,5), (3,6)])
sage: g.show(layout='circular')
sage: d = ranking(g)
sage: print [(p, v.n(digits = 3)) for p,v in d.items()]
[(0, 0.111), (1, 0.105), (2, 0.0642), (3, 0.160), (4, 0.0640), (5, 0.112), (6, 0.167), (7, 0.0639),
```



Si la página falsa tiene un link auténtico que aleja el camino aleatorio de nuestro nodo, la técnica se vuelve mucho menos efectiva.

```
sage: g=DiGraph(edges)
sage: g.add_vertices([20])
sage: g.add_edges([(3,20), (20,3), (20,0)])
sage: g.delete_edges([(3,2), (3,1), (3,5), (3,6)])
sage: g.show(layout='circular')
sage: d = ranking(g)
sage: print [(p, v.n(digits = 3)) for p,v in d.items()]
[(0, 0.163), (1, 0.123), (2, 0.0753), (3, 0.0622), (4, 0.0861), (5, 0.138), (6, 0.207), (7, 0.0753),
```



7.10 Autómatas celulares

Un autómata celular consiste de:

- un conjunto de celdas, cada una de las cuales puede estar activada, o no. Las celdas se pueden disponer en una línea, una cuadrícula, un grafo.
- una regla de transformación que dice cuál será el estado de una celda en el instante $k+1$ en función de los estados de las celdas vecinas en el instante k .

Los autómatas celulares se cuentan entre los programas de ordenador más sencillos posibles. La idea de Steven Wolfram es tomar todos los posibles autómatas celulares, sin pensar en qué representan, y estudiar su comportamiento. Algunos de estos autómatas celulares evolucionarán de formas totalmente previsibles, pero otros producen patrones muy intrincados.

```
sage: %cython
sage: from numpy import zeros
sage: def cellular(rule, int N):
...       '''Returns a matrix showing the evolution of a Wolfram's cellular automaton
...
...       rule:     determines how a cell's value is updated, depending on its neighbors
...       N:        number of iterations
...       '''
```

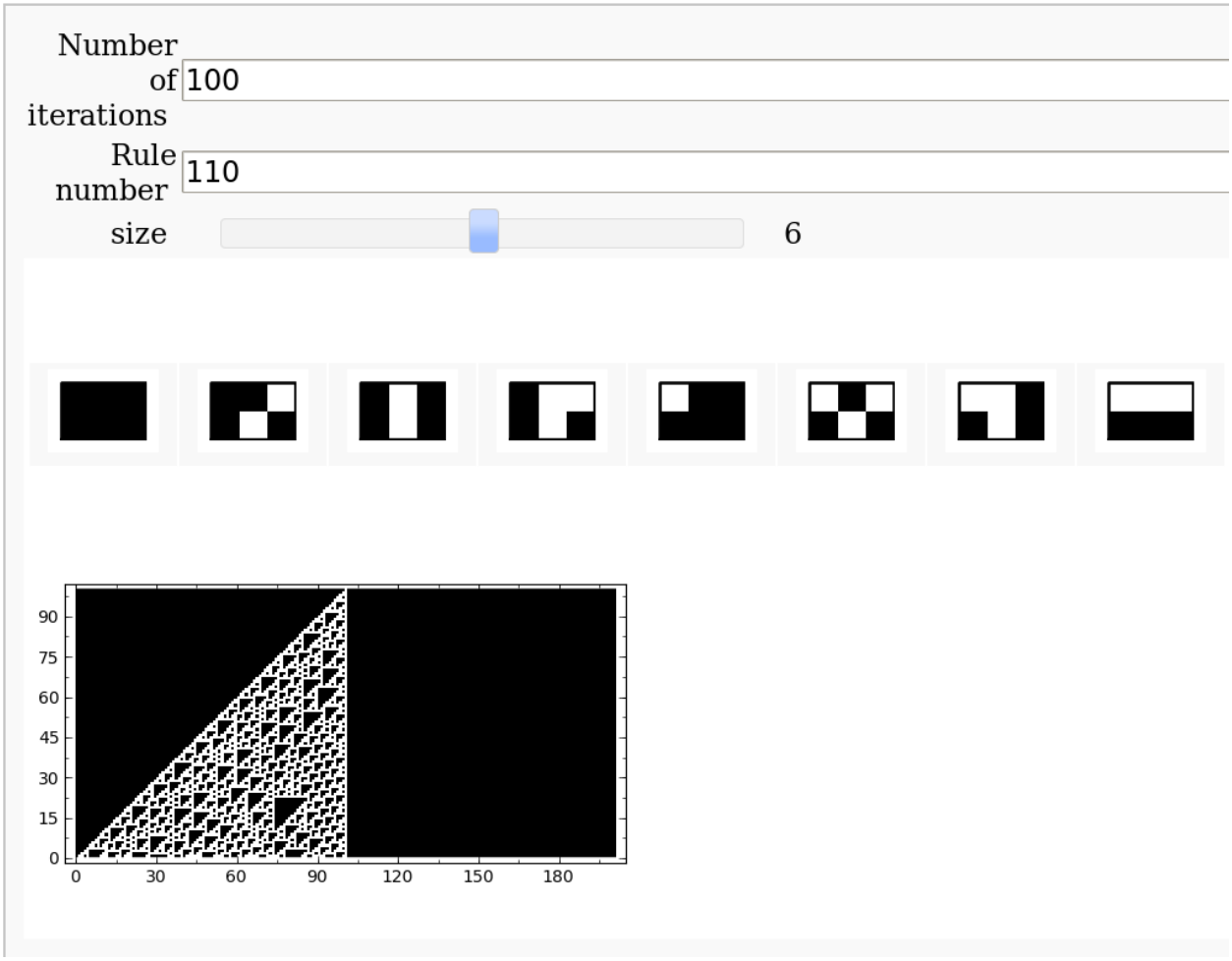
```
...       cdef int j,k,numi
...       M=zeros( (N,2*N+1), dtype=int)
...       M[0,N]=1
...
...       for j in range(1,N):
...           for k in range(N-j,N+j+1):
...               numi = 4*M[j-1,k-1] + 2*M[j-1,k] + M[j-1,k+1]
...               M[j,k]=rule[ numi ]
...       return M

sage: def plot_rule(rule):
...       '''Plots the rule
...       '''
...       plots = []
...       for j in srange(8):
...           regla = matrix(2,3)
...           digs = j.digits(base=2)
...           digs += [0]*(3-len(digs))
...           regla[0,0] = digs[2]
...           regla[0,1] = digs[1]
...           regla[0,2] = digs[0]
...           regla[1,1] = rule[j]
...           plots.append(matrix_plot(regla, frame=False, figsize=1)+
...                        line2d([(0,0),(0,2.05),(3,2.05),(3,0),(0,0)],
...                        thickness=1.5, rgbcolor=(0,0,0), axes=False)+
...                        line2d([(0,0),(0,2.05),(3,2.05),(3,0),(0,0)],
...                        thickness=1.5, rgbcolor=(0,0,0), axes=False))
...       return html.table([plots])

sage: plot_rule([0,0,0,1,1,0,0,1])
```

Un número que identifica una regla en la notación de Wolfram, se convierte mediante `num2rule` en una regla que pueda entender `cellular` .

El dibujo que realizamos es un `matrix_plot` , que pone el píxel (i,j) en blanco si la entrada (i,j) de la matriz es 1, y lo pone en negro si la entrada de la matriz es un 0.

La regla por defecto es la Regla 110 , uno de los automátas celulares más famosos.

```
sage: def num2rule(number):
...       if not (0 <= number <= 255):
...           raise Exception('Invalid rule number')
...       binary_digits = number.digits(base=2)
...       return binary_digits + [0]*(8-len(binary_digits))
sage: @interact
sage: def _( N=input_box(label='Number of iterations',default=100),
...          rule_number=input_box(label='Rule number',default=110),
...          size = slider(1, 11, step_size=1, default=6 ) ):
...       rule = num2rule(rule_number)
...       M = cellular(rule, N)
...       regla_plot = plot_rule(rule)
...       plot_M = matrix_plot(M, axes=False)
...       plot_M.show( figsize=[size,size])
```

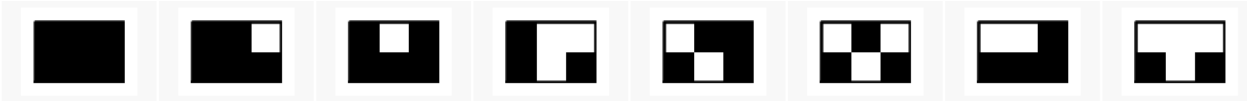


7.10.1 La regla 184 es un modelo del tráfico

http://en.wikipedia.org/wiki/Rule_184

Algunas reglas son modelos de fenómenos del mundo real. En concreto, la regla 184 puede servir como modelo del tráfico en una línea de celdas por donde se mueven los vehículos. Una celda está activa si y sólo si contiene un vehículo. Cada vehículo avanza a la derecha si no hay otro vehículo que se lo impida.

```
sage: rule184 = num2rule(184)
sage: plot_rule(rule184)
```



En el ejemplo de debajo, modelizamos dos carreteras que confluyen en una tercera, pasando por un semáforo que se abre alternativamente a una u otra carretera.

```
sage: l1=30
sage: l2=30
sage: l3=30
sage: turnos=150
sage: #Las tres carreteras
sage: c1 = zero_matrix(ZZ,turnos,l1)
```

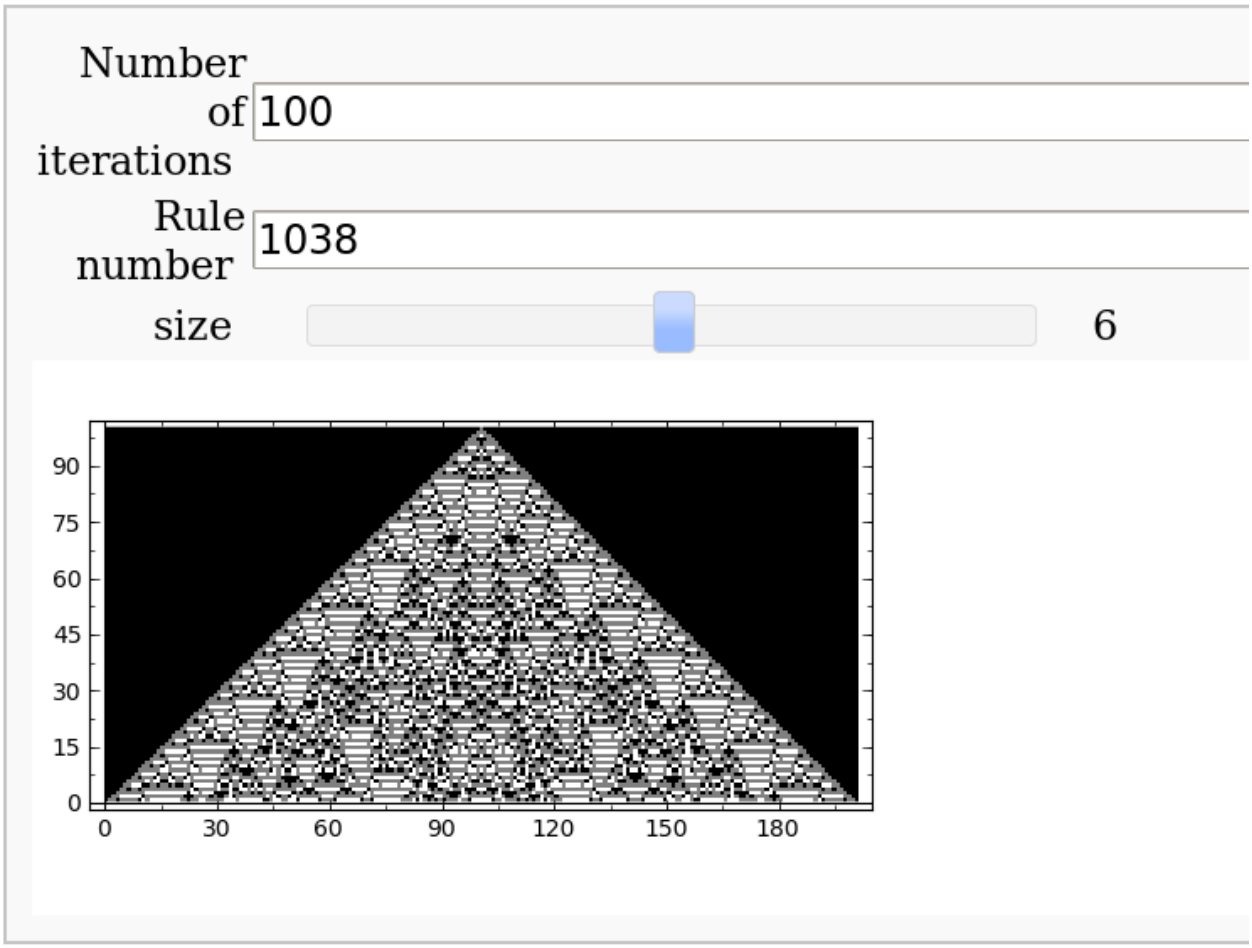
```
sage: c2 = zero_matrix(ZZ,turnos,l2)
sage: c3 = zero_matrix(ZZ,turnos,l3)
sage: densidad1=0.8
sage: densidad2=0.3
sage: semaforo={1:10,2:10}
sage: abierto=1  #semaforo abierto para c1 0 c2
sage: resto=5
sage: for j in range(turnos):
...       #actualizamos el semaforo: restamos un turno y cambiamos al llegar a cero
...       resto -= 1
...       if resto<=0:
...           #cambiamos el semaforo
...           abierto=2 if abierto==1 else 1
...           #reiniciamos el numero de turnos
...           resto = semaforo[abierto]
...
...       #actualizamos la carretera 1
...       #el primer punto es un 1 o un 0 con probabilidad = densidad1
...       c1[j, 0] = 1 if random()<densidad1 else 0
...       #el resto, excepto el último, se actualizan con la regla 184
...       for k in range(1,l1-1):
...           numi = 4*c1[j-1,k-1] + 2*c1[j-1,k] + c1[j-1,k+1]
...           c1[j,k] = rule184[numi]
...       #actualizamos la carretera 2
...       #el primer punto es un 1 o un 0 con probabilidad = densidad2
...       c2[j, 0] = 1 if random()<densidad2 else 0
...       #el resto, excepto el último, se actualizan con la regla 184
...       for k in range(1,l2-1):
...           numi = 4*c2[j-1,k-1] + 2*c2[j-1,k] + c2[j-1,k+1]
...           c2[j,k] = rule184[numi]
...       #actualizamos cerca del semaforo
...       if abierto==1:
...           numi = 4*c1[j-1,l1-2] + 2*c1[j-1,l1-1] + c3[j-1,0]
...           c1[j, l1-1] = rule184[numi]
...           numi = 4*c2[j-1,l1-2] + 2*c2[j-1,l1-1] + 1
...           c2[j, l1-1] = rule184[numi]
...
...           numi = 4*c1[j-1,l1-1] + 2*c3[j-1,0] + c3[j-1,1]
...           c3[j, 0] = rule184[numi]
...       else:
...           numi = 4*c1[j-1,l1-2] + 2*c1[j-1,l1-1] + 1
...           c1[j, l1-1] = rule184[numi]
...           numi = 4*c2[j-1,l1-2] + 2*c2[j-1,l1-1] + c3[j-1,0]
...           c2[j, l1-1] = rule184[numi]
...
...           numi = 4*c2[j-1,l1-1] + 2*c3[j-1,0] + c3[j-1,1]
...           c3[j, 0] = rule184[numi]
...       #actualizamos la carretera 3
...       #el resto, excepto el último, se actualizan con la regla 184
...       for k in range(1,l3-1):
...           numi = 4*c3[j-1,k-1] + 2*c3[j-1,k] + c3[j-1,k+1]
...           c3[j,k] = rule184[numi]
...       #el ultimo se queda a cero, queriendo decir que sale del sistema

sage: plots=[]
sage: for j in range(turnos):
...       c=max(l1,l2)
...       M=matrix(ZZ,3,c+l3)
```

```
...       M[0,(c-l1):c]=c1[j,:]
...       M[2,(c-l2):c]=c2[j,:]
...       M[1,c:]=c3[j,:]
...
...       plots.append(matrix_plot(M, axes=False))

sage: a=animate(plots, axes=False)

sage: a.show()
```

Reglas totalistas con tres colores

Exploramos ahora las reglas con tres colores (negro: 0, gris: 1 y blanco: 2) que tienen la siguiente propiedad adicional: el estado de la celda depende sólo de la suma de los colores de las celdas vecinas.

```
sage: %cython
sage: from numpy import zeros
sage: def cellular2(rule, int N):
...       '''Returns a matrix showing the evolution of the totalistic 3-color automaton
...
...       rule:     determines how a cell's value is updated, depending on its neighbors
...       N:        number of iterations
...       '''
...       cdef int j,k,numi
...       M=zeros( (N,2*N+1), dtype=int)
...       M[0,N]=1
...
...       for j in range(1,N):
...           for k in range(N-j,N+j+1):
...               #color contiene la suma de los tonos de las celdas vecinas
...               color = M[j-1,k-1] + M[j-1,k] + M[j-1,k+1]
...               M[j,k]=rule[ color ]
...       return M

sage: def num2rule2(number):
...       if not (0 <= number <= 2186):
...           raise Exception('Invalid rule number')
...       ternary_digits = number.digits(base=3)
...       return ternary_digits + [0]*(7-len(ternary_digits))
sage: @interact
sage: def _cell2( N=input_box(label='Number of iterations',default=100),
...          rule_number=input_box(label='Rule number',default=1038),
...          size = slider(1, 11, step_size=1, default=6 ) ):
...       rule = num2rule2(rule_number)
...       M = cellular2(rule, N)
...       plot_M = matrix_plot(M)
...       plot_M.show( figsize=[size,size])
```



- *genindex*
- *search*

Índice

S

T

V

W

X

Y

Z