PRIMERA VISITA RÁPIDA A SAGE:

Adaptación del original de Juan Luis Varona (septiembre - 2020)
Sage Version 8.7
(GNU Free Document License)

Sage (`http://www.sagemath.org`) es un entorno de cálculos matemáticos de código abierto que, gracias a los diversos programas que incorpora, permite llevar a cabo cálculos algebraicos, simbólicos y numéricos. El objetivo de Sage es crear una alternativa libre y viable a Magma, Maple, Mathematica y Matlab, todos ellos potentes (y muy caros) programas comerciales.

Sage sirve como calculadora simbólica de precisión arbitraria, pero también puede efectuar cálculos y resolver problemas usando métodos numéricos (es decir, de manera aproximada). Para todo ello emplea algoritmos que tiene implementados él mismo o que toma prestados de alguno de los programas que incorpora, como Maxima, NTL, GAP, Pari/gp, R y Singular. Y para llevar a cabo algunas tareas puede utilizar paquetes especializados opcionales. Incluye un lenguaje de programación propio, que es una extensión de Python (Sage mismo está escrito en Python).

Sage no sólo consta del programa en sí mismo, que efectúa los cálculos, y con el que podemos comunicarnos a través de terminal, sino que incorpora un interfaz gráfico de usuario a través de cualquier navegador web. ¡Echemos un vistazo a su sintaxis y su funcionamiento!

1. Uso como calculadora:

   ```
   5+4/3
   ```

2. Sage utiliza paréntesis `( )` para agrupar:

   ```
   (5+4)/3
   ```

3. Y también los usa para delimitar los argumentos de funciones:

   ```
   cos(pi) + 3
   ```

   Sage reconoce constantes matemáticas habituales: `i` o `I` es la unidad imaginaria, `e`, `pi`, ...:

   ```
   (3+4*I)^10
   e^(i*pi)      # Da igual usar e o E
   ```

   (la parte que sigue al `#` es un comentario).

4. Como calculadora, Sage proporciona resultados exactos:

   ```
   3^100             # Se usa ** o ^ para elevar a una potencia
   factorial(1000)
   ```

5. Sin embargo, no ocurre así si alguno de los números involucrados en el cálculo tiene decimales:

   ```
   3.0^100      # 3.0 es un número real, no un entero.
   ```

6. También efectúa cálculos exactos cuando aparecen funciones:

   ```
   arctan(1)
   ```

7. Con los comandos `n` o `N` conseguimos aproximaciones numéricas (ambos comandos son alias de `numerical_approx`). El símbolo `_` alude al último resultado obtenido:

   ```
   N(_)
   ```

8. Estas aproximaciones pueden tener la precisión que deseemos. Por ejemplo, evaluemos $\sqrt{10}$ con 50 cifras exactas:

   ```
   N(sqrt(10), digits=50)
   sqrt(10).n(digits=50)
   N(sqrt(10), 170)      # Significa bits de precisión, no dígitos
   ```

9. Para enteros (y elementos de otros anillos con división entera), se tienen las operaciones *división entera*, `x//y`, y *resto de la división entera*, `x%y`. Así, y puesto que $32 = 6 \cdot 5 + 2$,

   ```
   print 32//5; print 32%5
   ```

   devolverá `6` y `2`.

10. Asignación de variables (Sage tiene *tipado dinámico*, no es necesario especificar el tipo de variable), tras evaluar:

    ```
    a=1; b=1.; c=3/4; d=b+c
    ```

    `a` contiene el valor del entero 1; `b`, el *real* 1.0; `c` el racional $\frac{3}{4}$. Al sumarse un racional y un real, el resultado, por el tipado dinámico, se toma como *real* y así en `d` se tiene el *real* 1.75.

    Con `type(a)` se averigua el *tipo* del contenido de la variable `a`.

11. Asignación simultánea: si `a` contenía el valor 32 y `b`, el valor 5, entonces la asignación simultánea

    ```
    a,b=b,a%b
    ```

    produce que en `a` se cambie al valor 5 y a 2 el de `b`. Esto se debe a que, en las asignaciones, Sage primero *evalúa* la parte derecha, y luego asigna.

12. Si a tiene ya un valor asignado, `a+=` , `a-=` , `a*=` y `a/=` son *atajos* para `a=a+` , `a=a-` , `a=a*` y `a=a/` , respectivamente. Así, si `a` tuviera asignado el valor 3, tras

    ```
    a*=2; a+=1; a-=3; a/=2
    ```

    su asignación iría variando de 3 a 6, 7, 4 y, finalmente, 2.

13. Podemos librarnos de una asignación o definición previa mediante

    ```
    reset("a")
    reset()     # Reinicia todo Sage
    ```

14. Así se define la función $f(x) = \frac{1}{1+x^2}$:

    ```
    f(x) = 1/(1+x^2)
    ```

15. Y así se usa:

    ```
    var("r"); [f(x), f(x+1), f(3), f(r)]
    ```

16. La orden `diff` permite obtener la derivada (o derivadas parciales) de una función:

    ```
    var("x,y")
    diff(f(x))               # f la función definida antes
    diff(sin(x^2), x, 4)     # Derivada cuarta
    diff(x^2 + 17*y^2, y)    # primera derivada respecto y
    ```

    También se puede usar derivative: `derivative(x^2 + 17*y^2, y)` devolvería `34*y`.

17. Así calcularíamos una primitiva de f:

    ```
    integrate(f(x),x)        # Da igual usar integral o integrate
    ```

18. La integral definida $\int_0^1 f(x)\,dx$ se puede evaluar exactamente (mediante la regla de Barrow, por ejemplo):

    ```
    integral(x*sin(x^2), x)
    show(integrate(x/(1-x^3),x))
    integral(x/(x^2+1), x, 0, 1)
    ```

19. También existe integración numérica, pero su sintaxis es diferente. En la respuesta que se obtiene, el primer elemento es el resultado, y el segundo una cota del error:

    ```
    integral(x*tan(x), x)
    integral(x*tan(x), x,0,1)    # Lo devuelve exacto, sin aproximar
    numerical_integral(x*tan(x), 0,1) # Da una aproximación numérica
    ```

20. Cálculo de límites:

```
limit(sin(x)/abs(x), x=0)    # Se da cuenta de que no existe
limit(sin(x)/abs(x), x=0, dir="minus")
limit(sin(x)/abs(x), x=0, dir="plus")
```

21. Las funciones se pueden definir a trozos:

```
g = piecewise([[(-5,1),(1-x)/2], [(1,8),sqrt(x-1)]]); g
```

22. Para representar funciones disponemos del comando `plot`:

```
plot(g,-1, 3)    #  o g.plot(-1, 3)
plot(cos(x^2), -5, 5, thickness=5, rgbcolor=(0.5,1,0.5), fill = 'axis')
plot(bessel_J(2,x), 0, 20)
```

23. Así se guarda un gráfico en el disco duro:

```
save(plot(sin(x)/x, -5, 5), "ruta/dibujo.pdf")   # o plot(...).save("...")
```

24. Con `+` se superponen gráficos:

```
plot(2*t^2/3+t, 0, 6) + plot(3*t+20, 0, 6, rgbcolor='red')
  + line([(0, 10), (6, 10)], rgbcolor='green')
```

25. Podemos hacer animaciones:

```
onda = animate([sin(x+k) for k in srange(0,10,0.5)], xmin=0, xmax=8*pi)
onda.show(delay=30, iterations=1)
```

26. Para buscar ayuda sobre un comando (especialmente, su sintaxis y ejemplos de uso), basta poner `?` tras el nombre del comando; con `??` se obtiene información más técnica (sobre el código fuente):

```
plot?
numerical_integral??
```

27. También podemos buscar en la documentación (si está incorporada en el sistema):

```
search_doc("rgbcolor")
```

28. La orden `solve` sirve para resolver ecuaciones (obsérvese que se emplea `==`) o sistemas:

```
solve(x^2-2 == 0, x)
f = x^4 + 2*x^3 - 4*x^2 - 2*x + 3
solve(f == 0, x, multiplicities=true)
soluciones = solve([9*x - y == 2, x^2 + 2*x*y + y == 7], x, y)
soluciones[0][0].rhs()   # Componente x de la primera solución
```

29. Las matrices y vectores se crean así:

```
A = matrix([[-4,1,0],[3,5,-2],[6,8,3]]);
B = identity_matrix(3)
v = vector([3,-2,8]); w = vector([-1,1,1])
H = matrix([[1/(i+j+1) for i in [0..2]] for j in [0..2]])
```

30. Y con ellos se opera como sigue:

```
T = A^2*transpose(A) - 5*B - (1/20)*det(A)*exp(B)
v.dot_product(w)    # Producto escalar
H.inverse()         # También se puede usar ~H o H^(-1)
```

31. El sistema de ecuaciones lineales $Ax = w$ se resuelve con (si se hace simbólico con parámetros, no estudia casos)

```
x = A\w
```

32. Usando `simplify`, Sage simplifica expresiones (suele ser muy cuidadoso):

```
var("x"); sqrt(x^2)
sqrt(x^4)
simplify(_)    # Sigue sin hacer nada
assume(x>0); simplify(sqrt(x^2))   # Ya simplifica
sqrt(4, all=True)   # da 2 y -2
sqrt(16, extend=True)    # da 4
```

33. También con expresiones trigonométricas:

```
sin(asin(y))  # Devuelve y
asin(sin(x))  # Lo devuelve "sin hacer"
simplify(asin(sin(x)))   # Sigue sin hacer nada
```

Podemos delimitar el entorno de una variable:

```
assume(-pi/2 <= x <= pi/2); simplify(asin(sin(x)))
var('k t'); assume(k, 'integer'); simplify(sin(t+2*k*pi))
```

que se mantendrá, en la sesión de Sage, hasta que se encuentre con un `forget()`.

34. Un ejemplo que muestra un programita hecho en Python (con `"""..."""` ponemos la información que aparecerá al usar `letraDelDNI?`):

```
def letraDelDNI(n):
    u"""
    Esta funcion calcula la letra de un DNI español
    """
    letras = "TRWAGMYFPDXBNJZSQVHLCKE"
    return letras[n%23]
letraDelDNI(12345678)
```

35. Se pueden asignar valores por defecto a los argumentos de las funciones:

```
def add_n(x, n=1):
    return x + n

add_n(4)            # Devuelve 5
add_n(4, n=100)     # Devuelve 104
add_n(4, 1000)      # Devuelve 1004
```

36. Una función puede devolver varios valores:

```
def g(x):
    return x, x*x

g(2)                # Devuelve (2, 4)
resultado=g(2)
resultado           # Devuelve (2, 4)
```

La salida se puede "desempaquetar":

```
a,b = g(100)
a                   # Devuelve 100
b                   # Devuelve 10000
```