UNA VISITA RÁPIDA A SAGE

Juan Luis Varona (8 - febrero - 2010)
Sage Version 4.3.1
`http://wiki.sagemath.org/quickref`

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); es muy recomendable conocer Python para hacer un uso avanzado de Sage.

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; para representar las fórmulas y expresiones matemáticas utiliza jsMath, una implementación de LaTeX por medio de JavaScript. Sin necesidad de descargarlo e instalarlo en nuestro ordenador, podemos utilizar Sage en `http://www.sagenb.org`. Pero no nos preocupemos de ello; simplemente, ¡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 como argumentos de funciones:
```
cos(0)
```
4. Corchetes `[ ]` para formar listas (con sus elementos separados por comas):
```
v = [3,4,-6]      # Alternativa:  v = vector([3,4,-6])
```
5. También corchetes para acceder a elementos de listas (enumera contando desde 0, como en C y en Python):
```
v[2]
```
6. Como calculadora, Sage proporciona resultados exactos:
```
3^100             # Se usa ** o ^ para elevar a una potencia
factorial(1000)
```
7. Sin embargo, no ocurre así si alguno de los números involucrados en el cálculo tiene decimales (la parte que sigue al `#` es un comentario):
```
3.0^100      # 3.0 es un número real, no un entero.
```
8. También efectúa cálculos exactos cuando aparecen funciones:
```
arctan(1)
```
9. Con los comandos `n` o `N` conseguimos aproximaciones numéricas (ambos comandos son alias de `numerical_approx`). El símbolo `_` alude al último resultado obtenido:
```
N(_)
```
10. 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
```
11. Definición y uso de variables simbólicas (se puede usar `"` o `'`, y poner comas o no ponerlas):
```
var("alpha, x, y, z")              # Definimos alpha, x, y, z
z = sqrt(7*x + y^5 - sin(alpha))   # (z no hacía falta)
show(z)     # (o jsmath(z))   ¡LaTeX se encarga de dar formato!
latex(z)    # Proporciona el código LaTeX
```
12. Sage permite operar con números complejos (`i` o `I` es la unidad imaginaria):
```
(3+4*I)^10
e^(i*pi)      # Da iguar usar e o E
```
13. Podemos definir expresiones simbólicas y manipularlas (aquí, `;` sirve para separar órdenes):
```
var('x'); p = (x+1)*(x-1)^2     # El * es importante
q = expand(p); q
```
14. En este ejemplo, el camino inverso lo recorreríamos con
```
factor(q)
```
15. Ahora, hallemos (numéricamente) una raíz de `q` que esté entre `0` y `3`:
```
find_root(q, 0, 3)
```
16. Otro ejemplo de lo mismo:
```
var("theta")
find_root(cos(theta) == sin(theta)+1/5, 0, pi/2)
```
17. Para conocer el tiempo empleado por Sage en efectuar un cálculo:
```
time is_prime(2^127-1)
time factor(2^128-1)
```
18. Podemos librarnos de una asignación o definición previa mediante
```
reset("a")
reset()     # Reinicia todo Sage
```
19. Así se define la función $f(x) = \frac{1}{1+x^2}$:
```
f(x) = 1/(1+x^2)
```
20. Y así se usa:
```
var("r"); [f(x), f(x+1), f(3), f(r)]
```
21. 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)     # También se puede usar derivative
```
22. Así calcularíamos una primitiva de f:
```
integrate(f(x),x)         # Da igual usar integral o integrate
```
23. La integral definida $\int_0^1 f(x)\,dx$ podemos evaluarla exactamente (mediante la regla de Barrow, por ejemplo) o numéricamente (mediante una fórmula de cuadratura):
```
var("x")
integral(x*sin(x^2), x)
show(integrate(x/(1-x^3)))
integral(x/(x^2+1), x, 0, 1)
```
24. 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 sin hacer
numerical_integral(x*tan(x), 0,1)
```
25. 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")
```
26. Conoce la equivalencia de Stirling:
```
lim(factorial(x)*exp(x)/x^(x+1/2), x=oo)     # oo es lo mismo que infinity
```
27. Las funciones se pueden definir a trozos:
```
g = Piecewise([[(-5,1),(1-x)/2], [(1,8),sqrt(x-1)]],x)
```
28. Para representar funciones disponemos del comando `plot`:

```
plot(g)    #  o g.plot()
plot(cos(x^2), -5, 5, thickness=5, rgbcolor=(0.5,1,0.5), fill = 'axis')
plot(bessel_J(2,x,"maxima"), 0, 20)    # Funciona pero es muuuuuy lento
```

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

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

30. También podemos representar funciones en paramétricas, gráficos en tres dimensiones, curvas de nivel...

```
automatic_names(true)    # Ya no necesitamos predefinir las variables (v. 4.3.1)
parametric_plot((cos(t),sin(t)), 0,2*pi).show(aspect_ratio=1, frame=true)
plot3d(4*x*exp(-x^2-y^2), (x,-2,2), (y,-2,2))
contour_plot(sin(x*y), (x,-3,3), (y,-3,3), contours=5, plot_points=80)
```

31. Incluso funciones en implícitas en dos y tres dimensiones:

```
implicit_plot(sin(x*y) + sin(x)*sin(y) == 1, (x,-5,5), (y,-5,5))
implicit_plot3d(x^4 + y^4 + z^4 == 16,
  (x, -2, 2), (y, -2, 2), (z, -2, 2), viewer='tachyon')
```

32. 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')
```

33. 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)
```

34. Y gráficos interactivos:

```
f = sin(x)*e^(-x)
dibujof = plot(f,-1,5, thickness=2)
punto = point((0,f(x=0)), pointsize=80, rgbcolor=(1,0,0))
@interact
def _(orden=(1..12)):          # La variable de control
  ft = f.taylor(x,0,orden)
  dibujotaylor = plot(ft,-1, 5, color="green", thickness=2)
  show(punto + dibujof + dibujotaylor, ymin = -.5, ymax = 1)
```

35. 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??
```

36. También podemos buscar en la documentación:

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

37. 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
```

38. En la versión 4.3.1, Sage aún no sabe sumar series, pero se lo podemos pedir a Maxima:

```
sum(1/n^2 for n in (1..20))    # No sabe si en vez de 20 ponemos oo
maxima("sum(1/n^2,n,1,inf), simpsum")
```

39. 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]])
```

40. 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)
```

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

```
x = A\w
```

42. Sage nos permite resolver ecuaciones diferenciales:

```
x = var("x"); y = function("y",x)
desolve(diff(y,x,2)-2*diff(y,x)-3*y == exp(x)*sin(x),y)
desolve(diff(y,x) + 2*y - 8 == 0, y, ics=[3,5])  # Condición inicial y(3) = 5
desolvers?   # Más órdenes para resolver ecuaciones diferenciales (o sistemas)
```

43. También podemos resolverlas mediante métodos numéricos (p.e., con un Runge-Kutta):

```
y = function('y',x)
sol = desolve_rk4(diff(y,x)+y*(y-2) == x-3, y, ics=[1,2], step=0.1, end_points=8)
list_plot(sol, plotjoined=True, color="purple")
```

44. 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
```

45. También con expresiones trigonométricas:

```
sin(asin(y))  # Devuelve y
asin(sin(x))  # Lo devuelve "sin hacer"
simplify(_)   # Sigue sin hacer nada
assume(-pi/2 <= x <= pi/2); simplify(asin(sin(x)))
var('k t'); assume(k, 'integer'); simplify(sin(t+2*k*pi))
```

46. Pero Sage a veces hace chapuzas:

```
find_root(x*exp(-x), 2, 100)
```

47. Obsérvese también esto:

```
t=-40.0;    # Número real
sum([t^n/factorial(n) for n in [0..300]])
t = -40     # Número entero
N(sum([t^n/factorial(n) for n in [0..300]]))
```

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

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

49. Así se define una función de manera recursiva:

```
def f(n):
    if n <= 1: return 1
    elif n%2 == 0: return 2*f(n/2)
    else: return 3*f((n-1)/2)
f(12345678)
```

50. Concluyamos con otro programita, el test de Lucas-Lehmer (como s está definido módulo $2^p - 1$, las operaciones con s también son modulares):

```
def is_prime_lucas_lehmer(p):
    s = Mod(4,2^p-1)          # ¡Definimos s como un entero modular!
    for i in range(0, p-2):
        s = s^2 - 2
    return s==0
is_prime_lucas_lehmer(127)         # Nos dice si 2^127-1 es primo (Lucas, 1876)
time is_prime_lucas_lehmer(19937)  # El mayor primo conocido en 1971
```