Esenciais USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

O CÁLCULO CON *SAGE*

comandos relevantes para o cálculo en ciencia e enxeñaría

Francisco Pena (Departamento de Matemática Aplicada)

A linguaxe *Sage*

Sage é unha linguaxe de cálculo baseada en *Python* coa que podemos usar tanto o cálculo simbólico coma o numérico.

O cálculo simbólico permite, por exemplo, coñecer todas as solucións dunha ecuación:

```
show(solve(x^4 == 2*x^2, x)
```

Out: [x = – √2; x = √2; x = 0]

O cálculo numérico permite obter a aproximación numérica dunha delas:

```
find_root(x^4 == 2*x^2, 1, 2)
```

Out: 1.414213562373095

O cálculo simbólico achégase máis ao razoamento humano, mentres que o cálculo numérico achégase máis á arquitectura dun ordenador. En xeral, usarase o numérico cando non exista a solución simbólica ou, cando existindo, o sistema de cálculo simbólico non sexa quen de achala ou o tempo de cálculo non sexa asumible.

Algúns cálculos básicos

Podemos traballar con variables ou constantes simbólicas:

```
var('y')
```

A variable *x* é simbólica mentres non se redefina.

Unha expresión simbólica depende de variables simbólicas:

```
f(x,y) = sin(x+pi*y)
```

Pódese avaliar simbolicamente ou obter un valor aproximado co método .n:

```
f(2,1)
```

Out: sin(pi + 2)

```
f(2,1).n()
```

Out: -0.909297426825682

As expresións simbólicas pódense usar para calcular límites:

```
limit(sin(1/x), x=oo)
```

Out: 0

Para calcular límites laterais:

```
limit(1/x, x=0, dir='+')
```

Out: +Infinity

Para calcular derivadas:

```
diff(sin(1/x), x)
```

Out: -cos(1/x)/x^2

```
diff(sin(1/x), x, 2) #segunda derivada
```

*Out: 2*cos(1/x)/x^3 - sin(1/x)/x^4*

Para calcular o polinomio de Taylor, por exemplo, de sen(*x*) de grao 3 arredor do cero:

```
taylor(sin(x), x, 0, 3)
```

*Out: -1/6*x^3 + x*

Para calcular primitivas e integrais definidas:

```
integral(tan(x), x)
```

Out: log(sec(x))

```
integral(tan(x), x, 0, pi/4)
```

*Out: 1/2*log(2)*

Tamén podemos aproximar numericamente unha integral definida:

```
numerical_integral(tan(x), 0, pi/4)
```

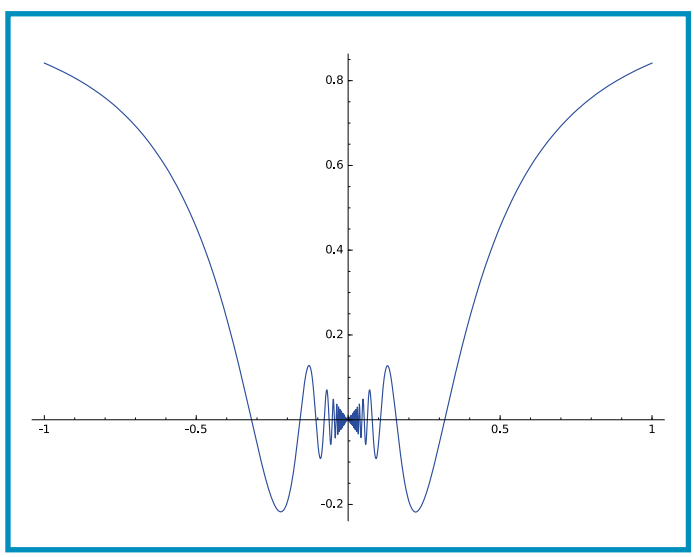
Out: (0.34657359027997264, 3.84773979655831e-15)

O resultado indica a aproximación da integral e a cota de erro.

As gráficas dunha variable

Vexamos como xerar gráficas de expresións escalares dunha variable:

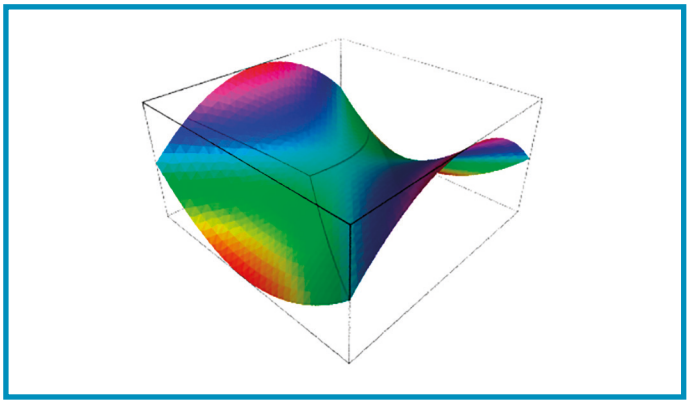
```
plot(x*sin(1/x), -1,1)
```



As gráficas de dúas variables

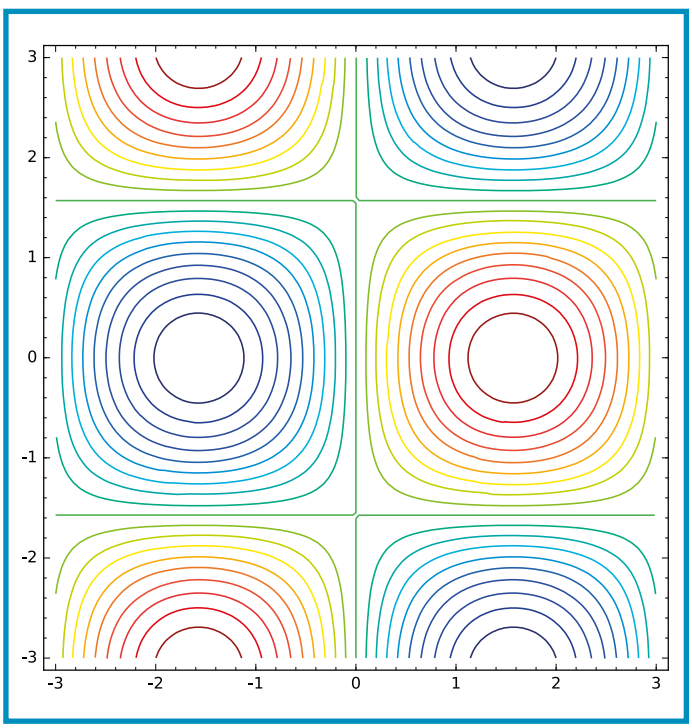
Vexamos gráficas de expresións escalares de dúas variables:

```
plot3d(x^2-y^2, (x,-1,1), (y,-1,1), \
adaptive=True)
```



Vexamos como debuxar curvas de nivel:

```
contour_plot(sin(x)*cos(y), (x,-3,3), \
(y,-3,3), fill=False, cmap='jet', \
contours=20)
```



A interpolación e axuste

Podemos calcular o polinomio de interpolación que pasa por uns nodos dados:

```
nodos = [(0,1),(2,2),(3,-2),(-4,9)]
pol = PolynomialRing(RDF,'x').\
lagrange_polynomial(nodos)
```

Tamén podemos axustar uns datos cun modelo dado. Por exemplo, uns datos que presentan un comportamento linear do tipo $y = 3x + 2$:

```
x = [0,0.1..1]
y = [3*(xi+0.1*random())+2 for xi in x]
datos = zip(x,y)
```

Sería razoable tratar de axustalos cun modelo do mesmo tipo:

```
a,b = var('a','b')
modelo(x) = a*x+b
d = find_fit(datos, modelo)
show(modelo.subs(d))
```

Out: 3.08598632229x+2.07545100447

A resolución de sistemas lineares

Un sistema linear precisa dunha matriz e un vector para ser definido:

```
A = matrix(RDF, 3, 3, [1..8, 0])
```

Out: *[1. 2. 3.]*
[4. 5. 6.]
[7. 8. 0.]

```
b = vector(RDF, 3*[1])
```

Out: (1., 1., 1.)

É preferible usar a clase RDF para a aritmética en coma flotante de 64 bits. A solución do sistema obtense co operador "\":

```
x = A\b
```

Out: (-1, 1, 0)

É posible calcular diversas factorizacións da matriz coma, por exemplo, LU e Cholesky:

```
P,L,U = A.LU()
if A.is_symmetric() and A.is_positive_definite():
  L = A.cholesky()
```

A resolución de ecuacións e sistemas non lineares

A resolución de ecuacións non lineares exemplifícase no primeiro apartado deste documento. No tocante a sistemas non lineares, pódense resolver con cálculo simbólico:

```
var('y')
solve([x-y^2+1==0, x^2+y^2-3==0], x, y)
```

Out: [[x == 1, y == -sqrt(2)], [x == 1, y == sqrt(2)], [x == -2, y == -I], [x == -2, y == I]]

Tamén se poden resolver con cálculo numérico:

```
f = lambda x,y: [x-y^2+1, x^2+y^2-3]
from mpmath import fp
fp.findroot(f, (0.1,0.1))
```

O comando `findroot` non admite expresións simbólicas, senón funcións *lambda*, coma a indicada, ou funcións *Python* coma a seguinte:

```
def f(x,y):
  return [x-y^2+1, x^2+y^2-3]
```

A optimización

Sage permite a busca de mínimos locais, non necesariamente globais, de expresións dunha variable:

```
find_local_minimum(-x*sin(x^2), 0, pi)
```

Devolve o valor mínimo e a abscisa, nesa orde:

Out: (-1.307619412991444, 1.3552111288348385)

O comando `minimize` engloba varios algoritmos numéricos de busca de mínimos para funcións de varias variables:

```
var('x y z')
f(x,y,z) = 100*(y-x^2)^2+(1-x)^2+100* \
(z-y^2)^2+(1-y)^2
minimize(f,[0.1,.3,.4])
```

Out: (1.00000000342, 1.00000000613, 1.00000001208)

As ecuacións diferenciais ordinarias

Podemos resolver ecuacións, coma $x + x - 1 = 0$, con cálculo simbólico:

```
var('t')
x = function('x',t)
DE = diff(x, t) + x - 1
desolve(DE, x)
```

*Out: _C + e^t)*e^(-t)*

É posible indicar unha condición inicial, por exemplo, $x(10) = 2$:

```
desolve(DE, x, [10,2])
```

*Out: (e^10 + e^t)*e^(-t)*

O mesmo comando permite resolver ecuacións de segunda orde:

```
desolve(diff(x,t,2)-x == t, x)
```

*Out: _K2*e^(-t) + _K1*e^t - t*

Agora, coas condicións $x(10) = 2$, $x(10) = 1$:

```
desolve(diff(x,t,2)-x == t, x, [10,2,1])
```

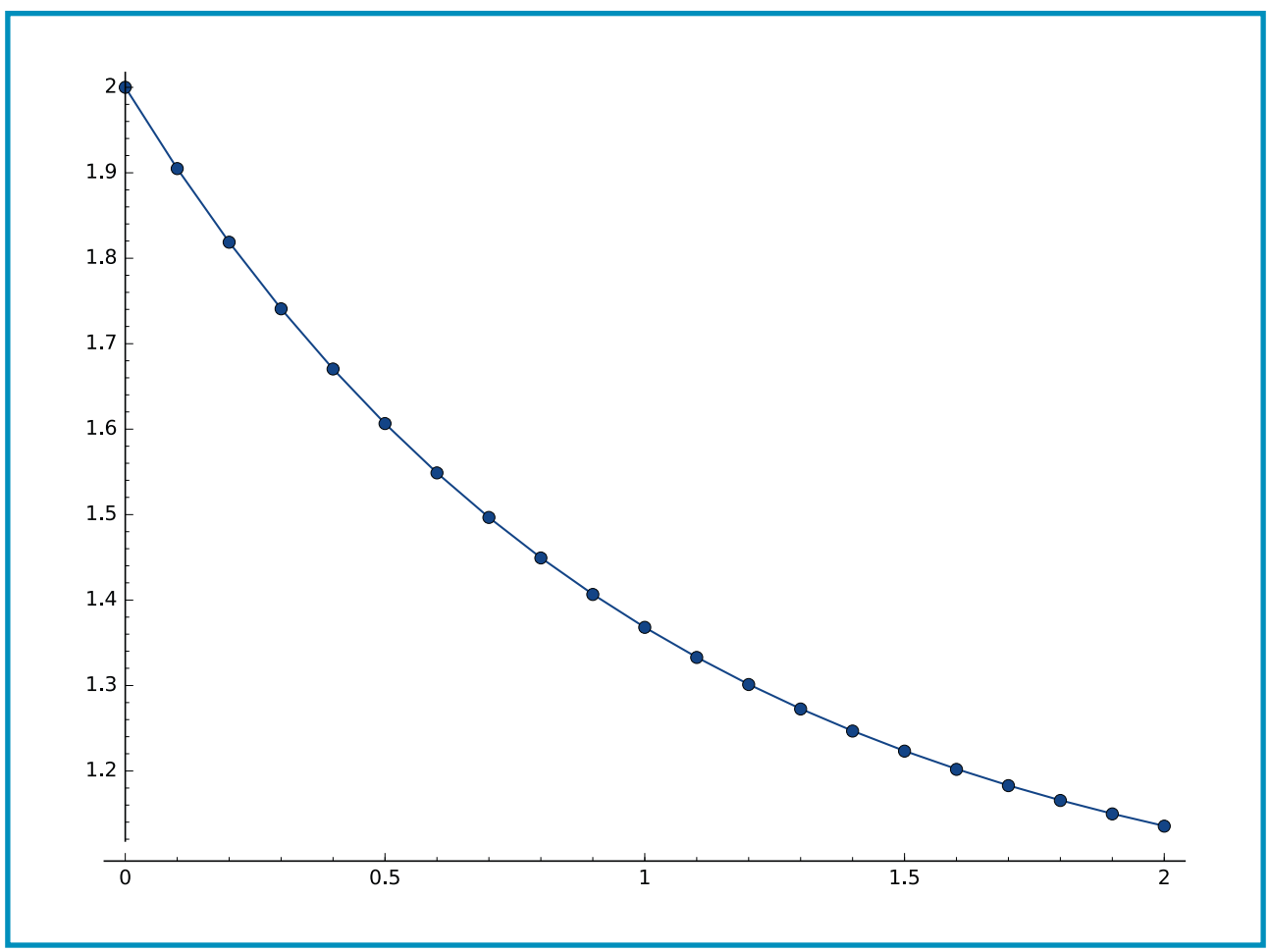
*Out: -t + 7*e^(t - 10) + 5*e^(-t + 10)*

As ecuacións de primeira orde pódense resolver numericamente. Deben estar escritas en forma explícita. Por exemplo, no caso de $x + x - 1 = 0$, debemos escribir:

```
var('x')
tt = [0,0.1..2]
s = desolve_odeint(1-x, 2, tt, x)
```

Vexamos a solución graficamente:

```
line(zip(tt,s), marker='o')
```



Os sistemas de ecuacións diferenciais ordinarias

Vexamos como resolver simbolicamente sistemas de primeira orde:

```
var('t')
x = function('x', t)
y = function('y', t)
de1 = diff(x,t) + y - 1 == 0
de2 = diff(y,t) - x + 1 == 0
desolve_system([de1, de2], [x,y])
```

*Out: [x(t) == (x(0) - 1)*cos(t) - (y(0) - 1)*sin(t) + 1, y(t) == (y(0) - 1)*cos(t) + (x(0) - 1)*sin(t) + 1]*

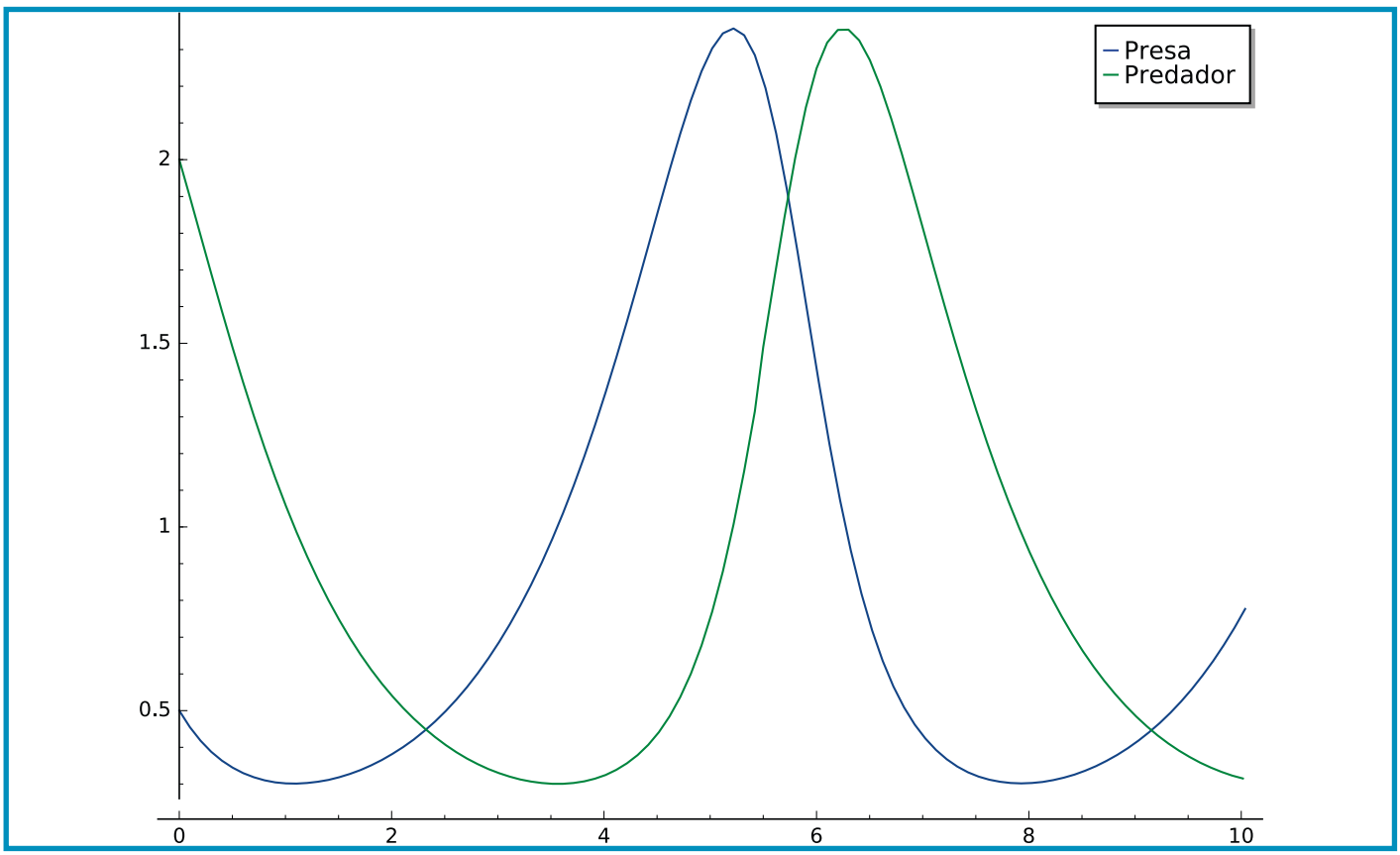
Agora, coas condicións $x(0) = 1$, $y(0) = 2$:

```
desolve_system([de1, de2], [x,y], [0,1,2])
```

Out: [x(t) == -sin(t) + 1, y(t) == cos(t) + 1]

Vexamos como resolver numericamente o sistema de primeira orde de Lotka-Volterra:

```
var('x y')
tt = [0,0.1..10]
s = desolve_odeint([x*(1-y), -y*(1-x)], \
[0.5,2], tt, [x,y])
line(zip(tt, s[:,0]), \
legend_label='Presa') + \
line(zip(tt, s[:,1]), \
legend_label='Predador', color='green')
```



A instalación e uso

Sage pódese descargar desde www.sagemath.org e instalar en Windows, Mac OSX e Linux.

Tamén se pode executar na nube desde cocalc.com.

Referencias

[1] Sage Development Team. Sage tutorial v9.0. http://doc.sagemath.org/html/en/tutorial/index.html.

[2] C. Finch. *Sage Beginner's Guide*. Packt Publishing, 2011.

[3] W. Stein. Sage quick reference. https://wiki.sagemath.org/quickref, 2009.

[4] J. L. Tábara. Matemáticas elementales con Sage. http://www.sagemath.org/es/Introduccion_a_SAGE.pdf, 2009.

[5] A. M. Vieites, F. Aguado, F. Gago, M. Ladra, G. Pérez, and C. Vidal. *Teoría de Grafos: ejercicios resueltos y propuestos. Laboratorio con Sage*. Thomson Paraninfo, 2014.