14 - Sympy

November 30, 2015

Figure 1: BY-SA

*Authors* : *Sonia Estrade*´

*Jose M. G* ´ *omez* ´

*Ricardo Graciani*

*F ranc Guell*

*M anuel Lopez* ´

*Xavier Luri*

*Josep Sabater*

1 Llibreria sympy

La llibreria sympy proporciona eines per al c`alcul simb`olic. En altres paraules, permet fer amb l’ordinador manipulacions de s´ımbols algebraics (f´ormules, equacions) de forma similar a com ho fan els humans. Amb sympy podrem:

*•* resoldre equacions,

*•* derivar funcions,

*•* integrar funcions,

*•* . . .

Amb sympy, les variables de Python no fan referencia a valors num`erics sin´o a funcions i ens permet operar amb elles. Pot trobar-se una descripci´o completa de sympy i les seves capacitats en aquest tutorial: Sympy tutorial

sympy dona a Python una funcionalitat similar a

Per a usar la llibreria cal fer la seg¨uent importaci´o:

import sympy

Tot i que es preferible fer-ho amb l’alias sp:

import sympy as sp

1.1 Funcionalitat b`asica: s´ımbols i expressions

El primer pas per a utilitzar sympy ´es definir els s´ımbols que volem manipular. Cal identificar algunes variables de Python com a objectes que representen s´ımbols (o variables en el sentit matem`atic). Per fer-ho es poden fer servir les funcions de sympy:

1

*•* var("x") defineix un ´unic s´ımbol

*•* symbols("x y z") defineix com a s´ımbols les paraules contingudes a la cadena Veiem-ho en un exemple:

In [1]: import sympy as sp

sp.init\_printing()

*# Definici´o individual. Fem que "x" sigui un s´ımbol*

sp.var("x")

*# Definici´o m´ultiple. Fem que "a", "b" i "c" siguin s´ımbols* a,b,c = sp.symbols("a b c")

*# A partir d’aquests s´ımbols definim una equaci´o de segon grau # Noteu que assignem la combinaci´o de s´ımbols a una nova variable* equacio = a\*x\*\*2 + b\*x + c

*# La resolem simb`olicament amb sympy*

equacio, sp.solve(equacio,x)

Out[1]:

*ax*2 + *bx* + *c,*

1 2*a*

*−b* +p*−*4*ac* + *b*2 *, −*12*a**b* +p*−*4*ac* + *b*2

En ambd´os casos, el que Phython est`a fent ´es:

1. Crear un objecte de tipus Symbol associat al nom que li donem

2. Associar el objecte creat a una variable

Hi ha una petita difer`encia entre var() i symbols(). var() crea d’una manera autom`atica una variable de Python amb el mateix nom i symbols() nom´es crea els objectes que hem d’assignar necess`ariament a una o m´es variables per poder utilitzar-les.

Podem fer que el nom de la variable no coincideixi amb el nom del s´ımbol, per`o cal anar amb compte per que aix`o pot portar a confusions.

In [2]: y, z = sp.symbols(’z y’)

(z, y)

Out[2]:

(*y, z*)

Noteu com al imprimir (x, y) Python ens mostra els noms dels s´ımbols que representen, en aquest cas amb y, x = sp.symbols(’x y’) hem fet que la variable x representi al s´ımbol de nom y (o variable matem`atica y) i la variable y al s´ımbol de nom x (o variable matem`atica x).

Una vegada que hem definit els nostres s´ımbols els podem combinar mitjan¸cant els operadors matem`atics habituals per definir expressions algebraiques:

equacio = a\*x\*\*2 + b\*x + c

Finalment podem fer que sympy resolgui la equaci´o que resulta al igualar aquesta expressi´o a cero. sp.solve(equacio)

2

sense m´es arguments, la funci´o solve() considera la equaci´o resultant de igualar a cero el seu argument i tractar`a de trobar la seva soluci´o. Si sympy es capa¸c de trobar la soluci´o (o solucions) ens tornar`a el resultat en forma de llista.

In [3]: import sympy as sp

sp.init\_printing()

*# Associem a la variable "incognita" el simbol "x"*

incognita = sp.symbols("x")

*# Creem una equaci´o usant la variable. Veurem que es genera amb el simbol x* equacio = incognita\*\*2 - 1

equacio

Out[3]:

*x*2 *−* 1

In [4]: sp.solve(equacio)

Out[4]:

[*−*1*,* 1]

Nota: la funci`o sp.init printing() configura sympy per que la impressi´o dels s´ımbols i de les expres sions es faci de la forma m´es eficient possible en l’entorn on s’executa Python. En el entorn Notebook sympy utilitza el format LATEX per un millor resultat.

1.1.1 Creant expressions a partir de cadenes de text

Una altra possibilitat que ofereix sympy ´es la creaci´o d’expressions algebraiques directament a partir d’una cadena de text. En aquest cas es creen els s´ımbols necessaris i es construeix l’expressi´o en un sol pas, de manera semblant al que fem amb var().

Per fer-ho cal usar la funci´o sympify().

In [5]: import sympy as sp

sp.init\_printing()

sp.var("x")

*# Creem una expressi´o, una equaci´o de segon grau*

equacio = sp.sympify("A\*x\*\*2+B\*x+C")

*# Exemple 1*

equacio,sp.solve(equacio,x)

Out[5]:

*Ax*2 + *Bx* + *C,*

1 2*A*

*−B* +p*−*4*AC* + *B*2 *, −*12*A**B* +p*−*4*AC* + *B*2

En aquest cas la nostra equaci´o *Ax*2 + *Bx* + *C* = 0 t´e diversas variables (els symbols *A*, *B*, *C* i *x*), per tant hem de indicar a sympy cual de aquestas variables s’han de considerar com a constant. Per aix`o hem utilitzat la funci´o solve() amb un segon argument, x. Es posible no fer-ho i deixar que sympy seleccione alguna.

In [6]: *# Exemple 2*

sp.solve(equacio)

3

Out[6]:

O b´e indicar-la directament.

In [7]: *# Exemple 3*

sp.var(’A’)

sp.var(’B’)

sp.var(’C’)

*A* : *−*1*x*2(*Bx* + *C*)

solutionA = sp.solve(equacio, A)

solutionB = sp.solve(equacio, B)

solutionC = sp.solve(equacio, C)

{ A: solutionA, B: solutionB, C: solutionC } Out[7]:

*A* :

*−*1*x*2(*Bx* + *C*)*, B* :*−Ax −Cx**, C* : [*−x* (*Ax* + *B*)]

Cal destacar que sympy no necessita que les variables A, B i C est´en definides que per poder resoldre la equaci´o (Exemples 1 i 2). Nom´es ens fa falta definir-les quan volem indicar-li que les utilitzi com a variables independents per resoldre la nostre equaci´o (Exemple 3). Per aix`o necessitem variables de Python que facin referencia als s´ımbols corresponents:

sp.var(’A’)

sp.var(’B’)

sp.var(’C’)

Si volem fer servir lletres gregues a les nostres expressions cal que creem els s´ımbols utilitzant els seus noms en angl`es. De la mateixa manera sympy i sympify() reconeixen la major part de las funcions matem`atiques habituals.

In [8]: sp.var(’theta’) *# crea un simbol i l’asigna a una variable amb el mateix nom* sp.sin( theta )

Out[8]:

sin (*θ*)

In [9]: expressio = sp.sympify(’A\*cos(phi)’)

expressio

Out[9]:

*A* cos (*φ*)

Com representa sympy internament les expressions.

In [10]: expressio.args

Out[10]:

(*A,* cos (*φ*))

4

In [11]: print( expressio, type(expressio) )

print( expressio.args[0], type(expressio.args[0]))

print( expressio.args[1], type(expressio.args[1]))

print( expressio.args[1].args[0], type(expressio.args[1].args[0]))

A\*cos(phi) <class ’sympy.core.mul.Mul’>

A <class ’sympy.core.symbol.Symbol’>

cos(phi) cos

phi <class ’sympy.core.symbol.Symbol’>

Nota: quan fem servir la funci´o print() per mostrar un s´ımbol o una expressi´o Python ens mostra la seva representaci´o com a cadena de text, sense fer servir la notaci´o *L*A*T*E*X*.

1.1.2 Com representar gr`aficament una expressi´o

Considerem la equaci´o d’una recta:

In [12]: import sympy as sp

sp.var(’x’)

sp.var(’a’)

sp.var(’b’)

y = a\*x + b

y

Out[12]:

*ax* + *b*

Podem assignar valors als s´ımbols *a* i *b* i representar gr´aficament el resultat

In [13]: y\_plot = y.subs(a, 2).subs(b, 1)

y\_plot

Out[13]:

2*x* + 1

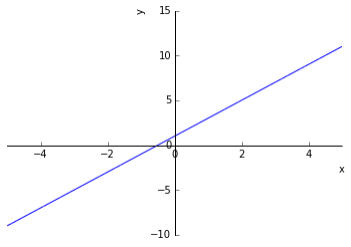
In [14]: %matplotlib inline

import sympy.plotting as symplot

*# per obtenir la grafique en el mateix notebook*

drawing = symplot.plot(y\_plot, (x, -5, 5), xlabel=’x’, ylabel=’y’)

5

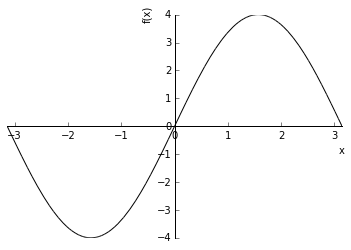


Ara podem afegir m´es funcions a la mateixa gr`afica.

In [15]: %matplotlib inline

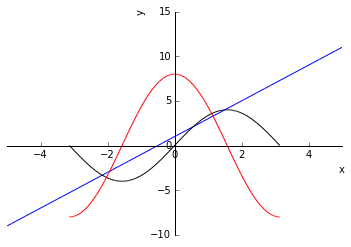
y\_sin = 4\*sp.sin(x)

y\_cos = 8\*sp.cos(x)

drawing.extend( symplot.plot( y\_sin, (x,-sp.pi,sp.pi), line\_color=’black’) ) 6

In [16]: %matplotlib inline

drawing.extend( symplot.plot( y\_cos, (x,-sp.pi,sp.pi), line\_color=’red’, show=False) ) drawing.show()



1.2 Manipulant expressions simb`oliques

En els exemples anteriors ja hem vist alguna manipulaci´o simb`olica amb sympy. Hem definit quatre s´ımbols i amb ells hem construit una equaci´o de segon grau, que despr´es hem resolt algebraicament. Usant sympy podem realitzar una gran varietat de manipulacions simb`oliques. Aqu´ı veurem alguns exemples, per`o remeteu-vos al tutorial de sympy per una descripci´o m´es completa.

1.2.1 Simplificant expressions

La funci´o simplify() permet la simplificaci´o d’expressions algebraiques. Aplica diverses t`ecniques per a intentar redu¨ır l’expressi´o donada a una forma m´es senzilla. Podeu trobar m´es detalls sobre la simplificaci´o a:

Sympy tutorial: simplification

Podem veure algunes aplicacions en els exemples seg¨uents.

In [17]: import sympy as sp

*# Simplificaci´o usant igualtats trigonom`etriques*

expr = sp.sympify("sin(x)\*\*2 - cos(x)\*\*2")

print("Expressi´o: ", expr)

print("Simplificaci´o: ", sp.trigsimp(expr))

print("Simplificaci´o: ", sp.simplify(expr))

print(10\*’-’)

*# Simplificaci´o de quocients de polinomis*

7

expr = sp.sympify("(x\*\*3 + x\*\*2 - x - 1)/(x\*\*2 + 2\*x + 1)")

print("Expressi´o: ", expr)

print("Simplificaci´o: ", sp.ratsimp(expr))

print("Simplificaci´o: ", sp.simplify(expr))

print(10\*’-’)

*# Simplificaci´o usant propietats de funcions*

expr = sp.sympify("gamma(x)/gamma(x - 2)")

print("Expressi´o: ", expr)

print("Simplificaci´o: ", sp.combsimp(expr))

print("Simplificaci´o: ", sp.simplify(expr))

Expressi´o: sin(x)\*\*2 - cos(x)\*\*2

Simplificaci´o: -cos(2\*x)

Simplificaci´o: -cos(2\*x)

----------

Expressi´o: (x\*\*3 + x\*\*2 - x - 1)/(x\*\*2 + 2\*x + 1)

Simplificaci´o: x - 1

Simplificaci´o: x - 1

----------

Expressi´o: gamma(x)/gamma(x - 2)

Simplificaci´o: (x - 2)\*(x - 1)

Simplificaci´o: (x - 2)\*(x - 1)

Noteu que a m´es de simplify() hem fet servir trigsimp(), ratsimp() o combsimp(). Quan utilitzem directament simplify() Python tractar`a de utilitzar el m`etodo m´es apropiat per a la expressi´o. A continuaci´o veurem algunes aplicacions de les t`ecniques de simplificaci´o en el cas de polinomis.

1.2.2 Factoritzaci´o de polinomis

La factoritzaci´o de polinomis es pot fer usant la funci´o factor():

In [18]: import sympy as sp

*# Factoritzaci´o d’un polinomi*

p = sp.sympify("x\*\*2 + 2\*x + 1")

print("Polinomi: ", p)

print("Factoritzaci´o: ", sp.factor(p))

print(10\*’-’)

*# Tamb´e funciona amb polinomis de m´es d’una variable*

p = sp.sympify("x\*\*2\*z + 4\*x\*y\*z + 4\*y\*\*2\*z")

print("Polinomi: ", p)

print("Factoritzaci´o: ", sp.factor(p))

Polinomi: x\*\*2 + 2\*x + 1

Factoritzaci´o: (x + 1)\*\*2

----------

Polinomi: x\*\*2\*z + 4\*x\*y\*z + 4\*y\*\*2\*z

Factoritzaci´o: z\*(x + 2\*y)\*\*2

Per polinomis m´es complexos, amb diverses variables, ´es possible indicar el ordre de preced`encia dels s´ımbols per a la factoritzaci´o:

In [19]: sp.var(’x,y,z,t’)

p = sp.sympify("x\*\*2\*z + 4\*x\*y\*z + 4\*y\*\*2\*z + x\*y + t\*z\*x")

8

print( "Polinomi: ", p )

print( "Factoritzacio: ", sp.factor(p) )

print( "Factoritzatio (x, t): ", sp.factor(p, [x, t]) )

print( "Factoritzatio (y, t): ", sp.factor(p, [y, t]) )

print( "Factoritzatio (z, t): ", sp.factor(p, [z, t]) )

Polinomi: t\*x\*z + x\*\*2\*z + 4\*x\*y\*z + x\*y + 4\*y\*\*2\*z

Factoritzacio: t\*x\*z + x\*\*2\*z + 4\*x\*y\*z + x\*y + 4\*y\*\*2\*z

Factoritzatio (x, t): t\*x\*z + x\*\*2\*z + x\*(4\*y\*z + y) + 4\*y\*\*2\*z

Factoritzatio (y, t): t\*x\*z + x\*\*2\*z + 4\*y\*\*2\*z + y\*(4\*x\*z + x)

Factoritzatio (z, t): t\*x\*z + x\*y + z\*(x\*\*2 + 4\*x\*y + 4\*y\*\*2)

1.2.3 Expansi´o de polinomis

Tamb´e podem realitzar el proc´es invers a la factoritzaci´o. Podem convertir un producte de polinomis en el polinomi equivalent usant la funci´o expand():

In [20]: import sympy as sp

sp.var("x")

*# Expansi´o d’un producte de polinomis*

p = sp.sympify("(x + 1)\*\*2")

print("Producte: ", p)

print("Polinomi: ", sp.expand(p))

print(10\*’-’)

*# En alguns casos expand pot simplificar l’expressi´o si es cancel·len termes* p = sp.sympify("(x + 1)\*(x - 2) - (x - 1)\*x")

print("Producte: ", p)

print("Polinomi: ", sp.expand(p))

Producte: (x + 1)\*\*2

Polinomi: x\*\*2 + 2\*x + 1

----------

Producte: -x\*(x - 1) + (x - 2)\*(x + 1)

Polinomi: -2

1.3 Substituci´o de s´ımbols: subs()

Quan tenim una expressi´o algebr`aica podem substituir els simbols que la composen per altres s´ımbols o per valors num´erics usant la funci´o subs().

In [21]: import sympy as sp

*# Definim un polinomi i substituim la x per cos(x)*

p = sp.sympify("(x + 1)\*\*2")

print("p = ", p)

print("p subs. = ", p.subs(x,sp.cos(x)))

print(10\*’-’)

*# Tamb´e podem fer una substituci´o per un valor num`eric*

q = sp.sympify("y\*(x + 1)\*\*2")

print("q = ", q)

print("q subs. = ", q.subs(x,3.))

9

p = (x + 1)\*\*2

p subs. = (cos(x) + 1)\*\*2

----------

q = y\*(x + 1)\*\*2

q subs. = 16.0\*y

1.4 Avaluaci´o num`erica d’expressions: evalf()

Donada una expressi´o algebraica podem avaluar-la donant valors num`erics als seus s´ımbols usant la funci´o evalf():

In [22]: import sympy as sp

*# Definim una expressi´o i l’avaluem per a un valor concret de x*

sp.var("x")

expr = 2\*x\*\*2+2\*x-1

print("Expressi´o: ", expr)

print("Valor per a x=2: ", expr.evalf(subs={x:2}))

Expressi´o: 2\*x\*\*2 + 2\*x - 1

Valor per a x=2: 11.0000000000000

In [23]: *# Tamb´e es pot fer si hi ha m´es d’un s´ımbol*

sp.var("y")

expr = y + 2.\*x\*\*2

print("Expressi´o: ", expr)

print("Valor per a x=2 y=5: ",expr.evalf(subs={x:2,y:5}))

Expressi´o: 2.0\*x\*\*2 + y

Valor per a x=2 y=5: 13.0000000000000

La funci´o evalf() permet a m´es que s’especifiqui el n´umero de xifres significatives del c`alcul, donat que sympy admet c`alculs en coma flotant de precisi´o arbitr`aria.

In [24]: import sympy as sp

*# Avaluaci´o d’arrels quadrades*

sp.var("x")

expr = sp.sqrt(x)

print("Expressi´o: ",expr)

print(10\*’-’)

print("Avaluaci´o sqrt(2) amb 100 xifres: ",expr.evalf(100,subs={x:2}))

print(10\*’-’)

print("Avaluaci´o sqrt(3) amb 100 xifres: ",expr.evalf(100,subs={x:3}))

Expressi´o: sqrt(x)

----------

Avaluaci´o sqrt(2) amb 100 xifres: 1.414213562373095048801688724209698078569671875376948073176679737990----------

Avaluaci´o sqrt(3) amb 100 xifres: 1.732050807568877293527446341505872366942805253810380628055806979451In [25]: print(type(expr.evalf(100,subs={x:3})))

<class ’sympy.core.numbers.Float’>

Noteu que podem usar evalf() per avaluar constants matem`atiques com *π* o *e* usant els objectes sympy que representen aquests objectes.

10

In [26]: import sympy as sp

*# Podem usar l’objecte sympy que representa pi per obtenir*

*# un numero arbitrari de decimals de pi*

print("pi =", sp.pi.evalf(1000))

print(10\*’-’)

*# El mateix per a e*

print("e =", sp.exp(1).evalf(1000))

pi = 3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117----------

e = 2.7182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664In [27]: help(sp.evalf)

Help on module sympy.core.evalf in sympy.core:

NAME

sympy.core.evalf

DESCRIPTION

Adaptive numerical evaluation of SymPy expressions, using mpmath

for mathematical functions.

CLASSES

builtins.ArithmeticError(builtins.Exception)

PrecisionExhausted

builtins.object

EvalfMixin

class EvalfMixin(builtins.object)

| Mixin class adding evalf capabililty.

|

| Methods defined here:

|

| evalf(self, n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False) | Evaluate the given formula to an accuracy of n digits.

| Optional keyword arguments:

|

| subs=<dict>

| Substitute numerical values for symbols, e.g.

| subs=*{*x:3, y:1+pi*}*. The substitutions must be given as a

| dictionary.

|

| maxn=<integer>

| Allow a maximum temporary working precision of maxn digits

| (default=100)

|

| chop=<bool>

| Replace tiny real or imaginary parts in subresults

| by exact zeros (default=False)

|

| strict=<bool>

| Raise PrecisionExhausted if any subresult fails to evaluate

11

| to full accuracy, given the available maxprec

| (default=False)

|

| quad=<str>

| Choose algorithm for numerical quadrature. By default,

| tanh-sinh quadrature is used. For oscillatory

| integrals on an infinite interval, try quad=’osc’.

|

| verbose=<bool>

| Print debug information (default=False)

|

| n = evalf(self, n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False)

class PrecisionExhausted(builtins.ArithmeticError)

| Method resolution order:

| PrecisionExhausted

| builtins.ArithmeticError

| builtins.Exception

| builtins.BaseException

| builtins.object

|

| Data descriptors defined here:

|

| weakref

| list of weak references to the object (if defined)

|

| ----------------------------------------------------------------------

| Methods inherited from builtins.ArithmeticError:

|

| init (self, /, \*args, \*\*kwargs)

| Initialize self. See help(type(self)) for accurate signature.

|

| new (\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature. |

| ----------------------------------------------------------------------

| Methods inherited from builtins.BaseException:

|

| delattr (self, name, /)

| Implement delattr(self, name).

|

| getattribute (self, name, /)

| Return getattr(self, name).

|

| reduce (...)

|

| repr (self, /)

| Return repr(self).

|

| setattr (self, name, value, /)

| Implement setattr(self, name, value).

|

| setstate (...)

|

12

| str (self, /)

| Return str(self).

|

| with traceback(...)

| Exception.with traceback(tb) --

| set self. traceback to tb and return self.

|

| ---------------------------------------------------------------------- | Data descriptors inherited from builtins.BaseException:

|

| cause

| exception cause

|

| context

| exception context

|

| dict

|

| suppress context

|

| traceback

|

| args

FUNCTIONS

N(x, n=15, \*\*options)

Calls x.evalf(n, *\*\**\*\*options).

Both .n() and N() are equivalent to .evalf(); use the one that you like better. See also the docstring of .evalf() for information on the options.

Examples

========

>>> from sympy import Sum, oo, N

>>> from sympy.abc import k

>>> Sum(1/k\*\*k, (k, 1, oo))

Sum(k\*\*(-k), (k, 1, oo))

>>> N( , 4)

1.291

add terms(terms, prec, target prec)

Helper for evalf add. Adds a list of (mpfval, accuracy) terms.

Returns

-------

- None, None if there are no non-zero terms;

- terms[0] if there is only 1 term;

- scaled zero if the sum of the terms produces a zero by cancellation e.g. mpfs representing 1 and -1 would produce a scaled zero which need special handling since they are not actually zero and they are purposely malformed to ensure that they can’t be used in anything but accuracy calculations;

13

- a tuple that is scaled to target prec that corresponds to the sum of the terms.

The returned mpf tuple will be normalized to target prec; the input prec is used to define the working precision.

XXX explain why this is needed and why one can’t just loop using mpf add as mpmath(x, prec, options)

bitcount(n)

check convergence(numer, denom, n)

Returns (h, g, p) where

-- h is:

> 0 for convergence of rate 1/factorial(n)\*\*h

< 0 for divergence of rate factorial(n)\*\*(-h)

= 0 for geometric or polynomial convergence or divergence

-- abs(g) is:

> 1 for geometric convergence of rate 1/h\*\*n

< 1 for geometric divergence of rate h\*\*n

= 1 for polynomial convergence or divergence

(g < 0 indicates an alternating series)

-- p is:

> 1 for polynomial convergence of rate 1/n\*\*h

<= 1 for polynomial divergence of rate n\*\*(-h)

check target(expr, result, prec)

chop parts(value, prec)

Chop off tiny real or complex parts.

complex accuracy(result)

Returns relative accuracy of a complex number with given accuracies for the real and imaginary parts. The relative accuracy is defined in the complex norm sense as ||z|+|error|| / |z| where error is equal to (real absolute error) + (imag absolute error)\*i.

The full expression for the (logarithmic) error can be approximated easily by using the max norm to approximate the complex norm.

In the worst case (re and im equal), this is wrong by a factor sqrt(2), or by log2(sqrt(2)) = 0.5 bit.

do integral(expr, prec, options)

evalf(x, prec, options)

evalf abs(expr, prec, options)

evalf add(v, prec, options)

14

evalf atan(v, prec, options)

evalf bernoulli(expr, prec, options)

evalf ceiling(expr, prec, options)

evalf floor(expr, prec, options)

evalf im(expr, prec, options)

evalf integral(expr, prec, options)

evalf log(expr, prec, options)

evalf mul(v, prec, options)

evalf piecewise(expr, prec, options)

evalf pow(v, prec, options)

evalf prod(expr, prec, options)

evalf re(expr, prec, options)

evalf subs(prec, subs)

Change all Float entries in ‘subs‘ to have precision prec. evalf sum(expr, prec, options)

evalf symbol(x, prec, options)

evalf trig(v, prec, options)

This function handles sin and cos of complex arguments. TODO: should also handle tan of complex arguments.

fastlog(x)

Fast approximation of log2(x) for an mpf value tuple x.

Notes: Calculated as exponent + width of mantissa. This is an approximation for two reasons: 1) it gives the ceil(log2(abs(x))) value and 2) it is too high by 1 in the case that x is an exact power of 2. Although this is easy to remedy by testing to see if the odd mpf mantissa is 1 (indicating that one was dealing with an exact power of 2) that would decrease the speed and is not necessary as this is only being used as an approximation for the number of bits in x. The correct return value could be written as "x[2] + (x[3] if x[1] != 1 else 0)".

Since mpf tuples always have an odd mantissa, no check is done to see if the mantissa is a multiple of 2 (in which case the result would be too large by 1).

Examples

15

========

>>> from sympy import log

>>> from sympy.core.evalf import fastlog, bitcount

>>> s, m, e = 0, 5, 1

>>> bc = bitcount(m)

>>> n = [1, -1][s]\*m\*2\*\*e

>>> n, (log(n)/log(2)).evalf(2), fastlog((s, m, e, bc))

(10, 3.3, 4)

finalize complex(re, im, prec)

get abs(expr, prec, options)

get complex part(expr, no, prec, options)

no = 0 for real part, no = 1 for imaginary part

get integer part(expr, no, options, return ints=False)

With no = 1, computes ceiling(expr)

With no = -1, computes floor(expr)

Note: this function either gives the exact result or signals failure.

hypsum(expr, n, start, prec)

Sum a rapidly convergent infinite hypergeometric series with given general term, e.g. e = hypsum(1/factorial(n), n). The quotient between successive terms must be a quotient of integer polynomials.

iszero(mpf, scaled=False)

pure complex(v)

Return a and b if v matches a + I\*b where b is not zero and a and b are Numbers, else None.

>>> from sympy.core.evalf import pure complex

>>> from sympy import Tuple, I

>>> a, b = Tuple(2, 3)

>>> pure complex(a)

>>> pure complex(a + b\*I)

(2, 3)

>>> pure complex(I)

(0, 1)

scaled zero(mag, sign=1)

Return an mpf representing a power of two with magnitude ‘‘mag‘‘ and -1 for precision. Or, if ‘‘mag‘‘ is a scaled zero tuple, then just remove the sign from within the list that it was initially wrapped in.

Examples

========

>>> from sympy.core.evalf import scaled zero

16

>>> from sympy import Float

>>> z, p = scaled zero(100)

>>> z, p

(([0], 1, 100, 1), -1)

>>> ok = scaled zero(z)

>>> ok

(0, 1, 100, 1)

>>> Float(ok)

1.26765060022823e+30

>>> Float(ok, p)

0.e+30

>>> ok, p = scaled zero(100, -1)

>>> Float(scaled zero(ok), p)

-0.e+30

DATA

C = <sympy.core.core.ClassRegistry object>

DEFAULT MAXPREC = 333

INF = inf

LG10 = 3.3219280948873626

MINUS INF = -inf

S = S

SYMPY INTS = (<class ’int’>,)

division = Feature((2, 2, 0, ’alpha’, 2), (3, 0, 0, ’alpha’, 0), 8192... evalf table = *{*<class ’sympy.core.numbers.Zero’>: <function create ev... fhalf = (0, 1, -1, 1)

fnan = (0, 0, -123, -1)

fnone = (1, 1, 0, 1)

fone = (0, 1, 0, 1)

fzero = (0, 0, 0, 0)

mp = <sympy.mpmath.ctx mp.MPContext object>

mpmath inf = mpf(’+inf’)

print function = Feature((2, 6, 0, ’alpha’, 2), (3, 0, 0, ’alpha’, 0)... rnd = ’n’

round nearest = ’n’

FILE

/Users/ricardo/anaconda3/lib/python3.4/site-packages/sympy/core/evalf.py

1.5 Resoluci´o d’equacions: solve()

Sympy implementa la resoluci´o algebraica d’equacions mitjan¸cant la funci´o solve(). Nota: la funci´o solve() assumeix per defecte que l’expressi´o que reb s’iguala a zero per a plantejar l’equaci´o.

Exemple 1: equaci´o polin`omica

*x*3 + 2*x −* 2 = 0

In [28]: import sympy as sp

sp.init\_printing()

*# Definim el polinomi*

p = sp.sympify("x\*\*3 + 2\*x - 2")

17

*# Resolem*

solucio = sp.solve(p, x)

print("Polinomi: ", p)

print("Soluci´o: ", solucio)

solucio

Polinomi: x\*\*3 + 2\*x - 2

Soluci´o: [(-1/2 - sqrt(3)\*I/2)\*(1 + sqrt(105)/9)\*\*(1/3) - 2/(3\*(-1/2 - sqrt(3)\*I/2)\*(1 + sqrt(105)/9)Out[28]:

 

*−*12*−√*3*i*2!3s1 +*~~√~~*105

9*−*2

~~q~~

*, −*2

~~q~~

+

*−*12+*√*3*i*2!3s1 +*~~√~~*105 9*, −*

3

*−*12 *−~~√~~*3*i* 2

3

1 +*~~√~~*105 9

3

*−*12 +*~~√~~*3*i* 2

3

1 +*~~√~~*105 9

A vegadas pot ser ´util fer servir la funci´o simplify(): In [29]: [ i.simplify() for i in solucio ] Out[29]:

*√*338*√*33 *−*1 + *√*3*i* 2 9 + *√*105 23





61 + *~~√~~*3*i*~~p~~39 + *~~√~~*105*,*

*√*338*√*33 *−*1 *−√*3*i* 2 9 + *√*105 23 61 *−~~√~~*3*i*~~p~~39 + *~~√~~*105*,*

*√*33*−*2*√*33 + 9 + *√*105 23 3~~p~~39 + *~~√~~*105

 

Tamb´e podem fer servir evalf() per obtenir un resultat num`eric.

In [30]: [ i.evalf() for i in solucio ]

Out[30]:

[*−*0*.*385458498529624 *−* 1*.*56388451052696*i, −*0*.*385458498529624 + 1*.*56388451052696*i,* 0*.*770916997059248] Exemple 2: equaci´o trigonom`etrica

cos(*x*) + sin(*x*) = 0

In [31]: import sympy as sp

sp.init\_printing()

*# Definim l’equaci´o trigonom`etrica*

e = sp.sympify("cos(x)+sin(x)")

*# Resolem*

solucio = sp.solve(e, x)

print("Expressi´o: ",e )

print("Soluci´o: ", solucio)

solucio

Expressi´o: sin(x) + cos(x)

Soluci´o: [-pi/4, 3\*pi/4]

18

Out[31]:

Exemple 3: sistema d’equacions

*−π*4*,*3*π*4

Si es vol resoldre un sistema d’equacions (lineal o no) les diverses equacions s’han de passar com arguments a la funci´o solve() en forma de llista, tant les equacions com els s´ımbols a resoldre.

In [32]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.var("y")

*# Resolem dos sistemes, un de lineal i l’altre no lineal*

sol1 = sp.solve([x + y - 3, x - y - 1], [x, y])

sol2 = sp.solve([sp.sin(x) + y , sp.cos(x) - y - 1], [x, y])

sol1,sol2

Out[32]:

*{x* : 2*, y* : 1*} ,*

h

(0*,* 0)*,*

*π*

2*, −*1

*,*

*π*

2*, −*1

i

Equacions suportades actualment per sympy s´on :

- Polinomis d’una variable,

- Transcendentals, - Combinacions a trossos de les anteriors,

- Sistemes d’equacions polin`omiques lineals, i

- Sistemes que continguin expressions relacionals .

Nota: Quan solve() retorna [] o un NotImplementedError no vol dir que la equaci´o no tingui soluci´o. Solament vol dir que no ha pogut trobar cap. Frequentment aixo vol dir que la soluci´o no es por repre sentar d’una manera simbolica. Per exemple, l’equaci´o *x* = *cos*(*x*) t´e soluci´o, per`o no es pot representar simb`olicament mitjan¸cant funcions est`andard.

In [33]: import sympy as sp

sp.init\_printing()

sp.var(’x’)

try:

sp.solve(x - sp.cos(x), ’x’)

except Exception as x:

print(type(x))

print(x)

<class ’NotImplementedError’>

multiple generators [x, cos(x)]

No algorithms are implemented to solve equation x - cos(x)

1.6 Algebra lineal amb ` sympy

Sympy implementa operacions d’`algebra lineal, ´es a dir, operacions amb vectors i matrius. De forma similar a numpy inclou una classe espec´ıfica per a representar matriu, la classe Matrix que pot incloure valors simb`olics.

19

1.7 Creaci´o de matrius

1.7.1 A partir d’una llista

La funci´o Matrix() pot crear una matriu a partir d’una llista, o una llista de llistes:

In [34]: import sympy as sp

sp.init\_printing()

matriu = sp.Matrix([[1, -1], [3, 4], [0, 2]])

print( ’Matriu:’, matriu )

print( ’Matriu shape:’, matriu.shape )

matriu

Matriu: Matrix([[1, -1], [3, 4], [0, 2]])

Matriu shape: (3, 2)

Out[34]:



1 *−*1

3 4 0 2

 

Si es crea un objecte Matrix a partir d’una llista unidimensional sympy ho interpreta com un vector i el representa en forma de matriu columna, per facilitar les operacions amb matrius. Com els ndarray de numpy, les objectes de tipus Matrix de sympy es podem cambiar de forma.

In [35]: import sympy as sp

sp.init\_printing()

matrix = sp.Matrix([1,2,3])

matrix

Out[35]:

 

1 2 3

 

In [36]: matrix = sp.Matrix( range(9)).reshape(3,3) matrix

Out[36]:



0 1 2

3 4 5 6 7 8

 

Com ´es natural en sympy, les components d’un objecte Matrix poden ser s´ımbols.

In [37]: import sympy as sp

sp.init\_printing()

matrix = sp.Matrix( sp.var(’x y z’) )

matrix

Out[37]:

 

*x y z*

 

20

1.7.2 Les funcions zeros(), ones(), eye(), ‘diag()“

Com en numpy, crean matrius de les mides donades amb totes les components iguals a zero o a u, unitaries o diagonals.

In [38]: import sympy as sp

sp.init\_printing()

matriu0 = sp.zeros(2, 3)

matriu1 = sp.ones(3, 2)

matriuI = sp.eye(3)

matriuD = sp.diag(range(5))

matriu0, matriu1, matriuI, matriuD

Out[38]:



0 0 0 0 0 0

*,*



1 1

1 1 1 1



 *,*



1 0 0

0 1 0 0 0 1



 *,*



0123 4

 

 

La funci`o diag() admet arguments de tipus Matrix i les organitza de forma diagonal:

In [39]: import sympy as sp

sp.init\_printing()

matriu = sp.diag(-1, sp.ones(2, 2), sp.Matrix([5, 7, 5])) matriu

Out[39]:





1.7.3 A partir d’una funci´o



*−*1 0 0 0

0 1 1 0 0 1 1 0 0 0 0 5 0 0 0 7 0 0 0 5

Es pot crear un objecte Matrix usant una funci´o. La funci´o ha de rebre com a par`ametres els ´ındexs d’una component i retornar el valor de la component que correspongui a aquests ´ındexs. Per crear la funci´o s’especifiquen les seves mides i la funci´o a usar.

In [40]: import sympy as sp

*# Definim la funci´o*

def funcio(i1,i2):

*"""Reb com a par`ametres els dos ´ındex i1,i2 i assigna com*

*a valor de la component la s´uma dels dos"""*

return i1+i2

*# Creem una matriu a partir de la funci´o*

A = sp.Matrix(4,4,funcio)

print("Matriu i1+i2:")

A

21

Matriu i1+i2:

Out[40]:





0 1 2 3



1 2 3 4

2 3 4 5

3 4 5 6

1.8 Acc´es als elements d’una Matriu 1.8.1 Elements

Els elements es poden accedir de la forma habitual donant [fila,columna] o rangs de [files,columnes] similar al slicing dels listes.

In [41]: import sympy as sp

sp.init\_printing()

matrix = sp.Matrix([[1, -1, 0], [2, 3, 4], [0, 2, 7]])

matrix

Out[41]:





1 *−*1 0



2 3 4

0 2 7

In [42]: print( ’Element 1,2:’, matrix[1,2]) matrix[0:2,0:3]

Element 1,2: 4

Out[42]:

1 *−*1 0

2 3 4

1.8.2 Files i columnes

Es pot accedir a les files i columnes d’una matriu usant els m`etodes row() i col(). Aquests m`etodes retornen un objecte Matrix que cont´e la fila o columna requerida.

In [43]: import sympy as sp

sp.init\_printing()

matriu = sp.Matrix([[1, -1, 0], [2, 3, 4], [0, 2, 7]])

matriu

Out[43]:





In [44]: matriu.row(1)



1 *−*1 0

2 3 4 0 2 7

22

Out[44]:

2 3 4

In [45]: matriu.col(2)

Out[45]:



04 7

 

Noteu que retornen una matriu de la forma apropiada.

La matriu transposada tamb´e ´es accessible f`acilment usant el membre T de l’objecte Matrix: In [46]: matriu, matriu.T

Out[46]:













1.9 Operacions b`asiques

1 *−*1 0 2 3 4 0 2 7



 *,*

 

1 2 0 *−*1 3 2 0 4 7

 

Els objectes de tipus Matrix es poden operar usant els operadors habituals +, -, \*, /, \*\*.

In [47]: import sympy as sp

sp.init\_printing()

*# Definim les matrius*

M = sp.Matrix([[1, -1, 0], [2, 3, 4], [0, 2, 7]])

N = sp.Matrix([[5, 6, 2], [8, 7, 7], [5, 1, 1]])

print( ’M, N’ )

M,N

M, N

Out[47]:













In [48]: *# Suma, resta*

print(’M+N, M-N’)

M+N,M-N

M+N, M-N

Out[48]:

1 *−*1 0 2 3 4 0 2 7



 *,*

 

5 6 2 8 7 7 5 1 1

 

 



6 5 2

10 10 11 5 3 8



 *,*





23



*−*4 *−*7 *−*2

*−*6 *−*4 *−*3 *−*5 1 6

 

In [49]: *# Producte de matrius*

print(’M\*N’)

M\*N

M\*N

Out[49]:





*−*3 *−*1 *−*5



54 37 29

51 21 21

In [50]: *# Divisio o Producte per la inversa* print(’M/N, M\*N^-1’)

M/N,M\*N\*\*-1

M/N, M\*N^-1

Out[50]:





1 *−*1 0







5 6 2





*−*1



*−*14110847





















*−*1477

108





2 3 4

8 7 7

108 *−*53

*,*

*−*5455

108

0 2 7

In [51]: *# Exponenciaci´o* print(’M, M\*\*2, M\*\*3’)

M, M\*\*2, M\*\*3

M, M\*\*2, M\*\*3

Out[51]:

5 1 1

36 *−*43 36

 



1 *−*1 0

2 3 4 0 2 7



 *,*



*−*1 *−*4 *−*4

8 15 40 4 20 57



 *,*



*−*9 *−*19 *−*44

38 117 340 44 170 479

 

 

In [52]: *# Operacions amb escalars* print(’M, M\*2, M/2’)

M, M\*2, M/2

M, M\*2, M/2

Out[52]:

 

 

1 *−*1 0 2 3 4 0 2 7



 *,*

 

2 *−*2 0 4 6 8 0 4 14



 *,*

 

1

2 *−*120 1322 0 1 72

 

 

Noteu que Matrix no admet operacions de suma o resta amb valors num`erics (escalars). Cal fer-ho usant la matriu unitaria auxiliar ones():

In [53]: *# Aquestes expressions donen error*

*#M+2*

*#M-2*

print(’M, M+2, M-2’)

*# Cal fer*

M, M+2\*sp.ones(M.rows,M.cols), M-2\*sp.ones(M.rows,M.cols)

24

M, M+2, M-2

Out[53]:







1 *−*1 0

2 3 4 0 2 7



 *,*



3 1 2

4 5 6 2 4 9



 *,*



*−*1 *−*3 *−*2

0 1 2 *−*2 0 5

 

 

I obviament, amb sympy totes aquestes operacions es poden fer amb matrius simb`oliques:

In [54]: import sympy as sp

sp.init\_printing()

*# Definim una matriu simb`olica*

sp.var("x")

M = sp.Matrix([[x, 2\*x, x], [x\*x, 3, 4], [x-1, x, x\*\*3]])

print("Matriu M")

M

Matriu M

Out[54]:









In [55]: M\*\*2

Out[55]:



*x* 2*x x*

*x*2 3 4

*x −* 1 *x x*3



2*x*3 + *x*2 + *x* (*x −* 1) 3*x*2 + 6*x x*4 + *x*2 + 8*x* 

*x*3 + 3*x*2 + 4*x −* 4 2*x*3 + 4*x* + 9 5*x*3 + 12 *x*3(*x −* 1) + *x*3 + *x* (*x −* 1) *x*4 + 2*x* (*x −* 1) + 3*x x*6 + *x* (*x −* 1) + 4*x*

Per defecte sympy no intenta simplificar l’expressi´o resultant, pot ser instru¨ıt per fer-ho utilitzant el m`etode simplify( ):

In [56]: N = M\*\*2

N.simplify()

N

Out[56]:





In [57]: M\*\*-1

Out[57]:

*x*2*x*2 + 2*x −* 13*x* (*x* + 2) *xx*3 + *x* + 8



*x*3 + 3*x*2 + 4*x −* 4 2*x*3 + 4*x* + 9 5*x*3 + 12 *xx*3 + *x −* 1*xx*3 + 2*x* + 1*xx*5 + *x* + 3



2*x*2 *−* 3  *x*(*−x*+2)

1*− −*2*x*2+8

(*−x*+2)(2*x*2*−*3)

1*− −*2*−*2

*−*2*x*~~2~~+3 *−*1

1 *−−*2*x*2+8

*−*2*x*~~2~~+3 *−*1*x*(*x −* 1)

+1*x*

(*−*2*x*~~2~~+3)(2*x*~~5~~*−*4*x*~~3~~*−x*+5) *−*2

2*x*

2*x*~~5~~*−*4*x*~~3~~*−x*+5

*−*2*x*~~2~~+3

*−*2*x*~~2~~+3

*−*2*x*~~2~~+3 *−*

2*x*5

*−*2*x*~~2~~+3 *−*(*−x*2+4)(2*x*2*−*3)*x*(*−x*+2)

(*−x*+2)(*−x*2+4)(2*x*2*−*3)

*−*2*x*~~2~~+3 *−*(*−x*2(*−*2*x*2+3

*−x*

*−*2*x*~~2~~+3*−* 1*x*(*x−*1)

(*−*2*x*~~2~~+3)(2*x*~~5~~*−*4*x*~~3~~*−x*+5)

(*−*2*x*2+3)~~2~~(2*x*5*−*4*x*3*−x*+5) +1

(2*x*2*−*3)*x*(*−x*+2)

2*x*~~5~~*−*4*x*~~3~~*−x*+5 *−*(*−x*+2)(2*x*2*−*3)

*−*2*x*~~2~~+3*−* 1*x*(*x−*1)

(*−*2*x*~~2~~+3)(2*x*~~5~~*−*4*x*~~3~~*−x*+5)2*x*5

25

In [58]: N = M\*\*-1

N.simplify()

N

Out[58]:



*−*3*x*2+4

2*x*~~5~~*−*4*x*~~3~~*−x*+5

*x*(2*x*2*−*1)

2*x*~~5~~*−*4*x*~~3~~*−x*+5 *−*5 2*x*~~5~~*−*4*x*~~3~~*−x*+5

 

*x*5*−*4*x*+4

*x*(2*x*~~5~~*−*4*x*~~3~~*−x*+5)

*−x*3+3*x−*3

*x*(2*x*~~5~~*−*4*x*~~3~~*−x*+5)

1.10 Operacions aven¸cades

*−x*3+*x−*1

2*x*~~5~~*−*4*x*~~3~~*−x*+5 *−x*+2

2*x*~~5~~*−*4*x*~~3~~*−x*+5

*−x*2+4

2*x*~~5~~*−*4*x*~~3~~*−x*+5 2*x*2*−*3

2*x*~~5~~*−*4*x*~~3~~*−x*+5

A m´es de les operacions anteriors sympy proporciona tamb´e diverses operacions aven¸cades d’`algebra lineal.

1.10.1 Transposici´o

Es realitza amb el membre T de qualsevol objecte de tipus Matrix.

In [59]: import sympy as sp

sp.init\_printing()

sp.var("x")

M = sp.Matrix([[x, 2\*x, 0], [2, x-1, 4], [x+1, 2, 7]])

print("M, Transposta de M")

M,M.T

M, Transposta de M

Out[59]:













1.10.2 Determinants

*x* 2*x* 0 2 *x −* 1 4 *x* + 1 2 7



 *,*



*x* 2 *x* + 1 

2*x x −* 1 2 0 4 7

 

Es calculen amb el m`etodo det()

In [60]: import sympy as sp

sp.init\_printing()

sp.var("x")

M = sp.Matrix([[x, 2\*x, 0], [2, x-1, 4], [x+1, 2, 7]])

print("M, Determinant de M")

M,M.det()

M, Determinant de M

Out[60]:

 



*x* 2*x* 0 

2 *x −* 1 4 *x* + 1 2 7



 *,* 15*x*2 *−* 35*x*

 

26

1.10.3 Vectors generadors del nucli (kernel)

El m`etode nullspace() permet obtenir una base (vectors generadors) del nucli (subespai vectorial que t´e per imatge el vector zero) de l’aplicaci´o lineal representada per la matriu.

*M · v* = 0

In [61]: import sympy as sp

sp.init\_printing()

*# Definim la matriu*

M = sp.Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])

print(’M, kernel de M’)

M,M.nullspace()

M, kernel de M

Out[61]:







1 2 3 0 0

*,*

4 10 0 0 1

In [62]: for v in M.nullspace():

print( M \* v )

Matrix([[0], [0]])

Matrix([[0], [0]])

Matrix([[0], [0]])

1.10.4 Valors i vectors propis



*−*15

6

1

0

0



*,*



0001 0



*,*



1*−*12

0

0

1

 

 

 

El m`etode eigenvals() permet trobar els vectors propis d’una matriu. Retorna un diccionari que cont´e els valors propis com a claus i la seva multiplicitat com a elements.

In [63]: import sympy as sp

sp.init\_printing()

*# Definim la matriu*

M = sp.Matrix([[3, -2, 4, -2], [5, 3, -3, -2], [5, -2, 2, -2], [5, -2, -3, 3]]) print(’M, valors propis de M’)

M,M.eigenvals()

M, valors propis de M

Out[63]:

 



3 *−*2 4 *−*2

5 3 *−*3 *−*2 5 *−*2 2 *−*2 5 *−*2 *−*3 3



*, {−*2 : 1*,* 3 : 1*,* 5 : 2*}*

 

De forma similar el m`etode eigenvects() permet calcular els vectors propis de la matriu. La funci´o retorna els valors propis, la seva multiplicitat i els vectors propis.

27

In [64]: M.eigenvects() Out[64]:

 



*−*2*,* 1*,*

 



011 1

 

 



 *,*



3*,* 1*,*

 



111 1

 

 



 *,*



5*,* 2*,*

 



111 0



*,*



0*−*10 1

 

 

 

 

1.10.5 Diagonalitzaci´o

La funci´o diagonalize() permet diagonalitzar una matriu. Donada una matriu *M* retorna una tupla (*P, D*) de manera que *M* = *P DP −*1.

In [65]: print("M, (P,D) Noteu que la diagonal de D cont´e els valors propis") M,M.diagonalize()

M, (P,D) Noteu que la diagonal de D cont´e els valors propis

Out[65]:

 



3 *−*2 4 *−*2

5 3 *−*3 *−*2 5 *−*2 2 *−*2 5 *−*2 *−*3 3



*,*

 



0 1 1 0

1 1 1 *−*1 1 1 1 0 1 1 0 1



*,*



*−*2 0 0 0

0 3 0 0 0 0 5 0 0 0 0 5

 

 

 

1.10.6 Resolucio de sistemes d’equacions

Es poden resoldre a partir d’una matriu mitjan¸cant el m`etode de descomposici´o LU incl`os a la classe Matrix o b´e simplement a partir de la matriu inversa del sistema.

Si considerem el sistema de equacions (el mateix que amb numpy):

*x − y* + *z* = 4

2*x* + *y −* 3*z* = 1

7*x − y −* 3*z* = 14

Hi ha quatre possibilitats per resoldre-ho:

In [66]: import sympy as sp

sp.init\_printing()

*# Definim la matriu del sistema*

A = sp.Matrix([[1, -1, 1], [2, 1, -3], [7, -1, -3]])

*# Definim el vector de termes independents*

b = sp.Matrix([4, 1, 14])

print("Sistema d’equacions")

try:

A,b,A.LUsolve(b),A.inv()\*b

except Exception as x:

print(type(x))

print(x)

Sistema d’equacions

<class ’ValueError’>

No nonzero pivot found; inversion failed.

28

In [67]: import sympy as sp

sp.init\_printing()

sp.var("x, y, z")

expre1 = x - y + z - 4

expre2 = 2\*x + y - 3\*z - 1

expre3 = 7\*x - y - 3\*z - 14

sp.solve( [expre1, expre2, expre3] ) Out[67]:

*x* :2*z*3+53*, y* :5*z*3*−*73

In [68]: eq1 = sp.Eq( x - y + z, 4 ) eq2 = sp.Eq( 2\*x + y - 3\*z, 1 ) eq3 = sp.Eq( 7\*x - y - 3\*z, 14 )

sp.solve( [eq1, eq2, eq3] )

Out[68]:

*x* :2*z*3+53*, y* :5*z*3*−*73

In [69]: M = sp.Matrix( [[1, -1, 1], [2, 1, -3], [7, -1, -3]] ) eq = sp.Eq( M \* sp.Matrix([x,y,z]), sp.Matrix([4, 1, 14]) ) eq, sp.solve(eq )

Out[69]:

 

 

*x − y* + *z* 2*x* + *y −* 3*z* 7*x − y −* 3*z*



 =

 

4

1

14



 *,*

*x* :2*z*3+53*, y* :5*z*3*−*73 

Ara una equaci´o amb determinant no nulo.

In [70]: import sympy as sp

sp.init\_printing()

*# Definim la matriu del sistema*

A = sp.Matrix([[1, -1, 0], [2, 3, 4], [0, 2, 7]])

*# Definim el vector de termes independents*

b = sp.Matrix([1, 1, 1])

print("Sistema d’equacions")

A,b,A.LUsolve(b),A.inv()\*b

Sistema d’equacions

Out[70]:

 

 

1 *−*1 0 2 3 4 0 2 7



 *,*

 

1 1 1



 *,* 29

 

16

27

*−*1127 7 27



 *,*

 

16

27

*−*1127 7 27

 

 

In [71]: import sympy as sp

sp.init\_printing()

sp.var("x, y, z")

expre1 = x - y - 1

expre2 = 2\*x + 3\*y + 4\*z - 1

expre3 = 2\*y + 7\*z - 1

sp.solve( [expre1, expre2, expre3] ) Out[71]:

*x* :1627*, y* : *−*1127*, z* :727

In [72]: eq1 = sp.Eq( x - y , 1 ) eq2 = sp.Eq( 2\*x + 3\*y + 4\*z, 1 ) eq3 = sp.Eq( + 2\*y + 7\*z, 1 )

sp.solve( [eq1, eq2, eq3] )

Out[72]:

*x* :1627*, y* : *−*1127*, z* :727

In [73]: M = sp.Matrix( [[1, -1, 0], [2, 3, 4], [0, 2, 7]] ) eq = sp.Eq( M \* sp.Matrix([x,y,z]), sp.Matrix([1, 1, 1]) ) eq, sp.solve(eq )

Out[73]:

 

 

*x − y*

2*x* + 3*y* + 4*z* 2*y* + 7*z*



 =

 

1 1 1



 *,*

*x* :1627*, y* : *−*1127*, z* :727 

1.11 Integraci´o amb simpy

1.11.1 Intregrals indefinides

sympy permet fer integraci´o simb`olica de funcions (integrals indefinides) mitjan¸cant la funci´o integrate(), que reb com a argument una expressi´o simb`olica.

In [74]: import sympy as sp

sp.init\_printing()

sp.var("x")

print( "Integrant sin(x)\*exp(x)" )

sp.integrate(sp.sin(x)\*sp.exp(x),x)

Integrant sin(x)\*exp(x)

Out[74]:

2sin (*x*) *−ex*2cos (*x*)

*ex*

Aquesta mateixa funci´o permet fer integrals m´ultiples:

30

In [75]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.var("y")

print( "Integrant exp(-x\*\*2-y\*\*2)" )

sp.integrate(sp.exp(-x\*\*2 - y\*\*2),x,y)

Integrant exp(-x\*\*2-y\*\*2)

Out[75]:

*π*

4erf (*x*) erf (*y*)

1.11.2 Integrals definides

La mateixa funci´o integrate() permet tamb´e calcular integrals definides indicant els l´ımits d’integraci´o. Per exemple la integral

R *π*

0*sin*(*x*)*dx* = *−cos*(*x*)*|π*0 = 2

s’implementa com:

In [76]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.integrate(sp.sin(x),(x,0,sp.pi))

Out[76]:

2

Es pot indicar un l´ımit d’integraci´o infinit amb el s´ımbol simpy.oo. L’exemple seg¨uent implementa la integral:

R *∞*

0*e−xdx* = *−e−x|∞*

0 = 1

In [77]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.integrate(sp.exp(-x),(x,0,sp.oo))

Out[77]:

1

De forma similar es poden calcular integrals dobles definides. Per exemple, la integral:

R *∞* 0

R *∞*

0*e−x*2*−y*2*dxdy* = *π*

In [78]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.var("y")

sp.integrate(sp.exp(-x\*\*2 - y\*\*2), (x, -sp.oo, sp.oo), (y, -sp.oo, sp.oo)) 31

Out[78]:

*π*

Els l´ımits d’integracio poden ser s´ımbols, i el resultat s’expressa en funci´o d’ells. Per exemple R *a*

0*sin*(*x*)*dx* = *−cos*(*a*) + 1

In [79]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.var("a")

sp.integrate(sp.sin(x),(x,0,a))

Out[79]:

*−* cos (*a*) + 1

Noteu que cal definir el s´ımbol que utilitzem com a l´ımit de integraci´on.

En l’exemple seg¨uent el resultat indica dues possibilitats per qu`e la integral no convergeix si la part real de *a* no ´es *>* 1

In [80]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.var("a")

sp.integrate(x\*\*a\*sp.exp(-x), (x, 0, sp.oo))

Out[80]:

(

Γ(*a* + 1) for *− <a <* 1

R *∞*

0*xae−x dx* otherwise

1.11.3 Integrals com a s´ımbols

En cas que la integral no es vulgui avaluar, obtenint la funci´o o el valor num`eric corresponent, es pot usar Integral() que retorna la integral com una expressi´o simb`olica de sympy. Posteriorment la integral es pot avaluar usant doit()

Per exemple, en el cas d’integraci´o definida:

In [81]: import sympy as sp

sp.init\_printing()

sp.var("x")

resultat = sp.Integral(sp.exp(-x\*\*2 - y\*\*2), (x, -sp.oo, sp.oo), (y, -sp.oo, sp.oo)) resultat, resultat.doit()

Out[81]:

Z *∞ −∞*

Z *∞ −∞*

*e−x*2*−y*2*dx dy, π*

I un exemple en el cas d’integraci´o simb`olica: 32

In [82]: import sympy as sp

sp.init\_printing()

sp.var("x")

resultat = sp.Integral((x\*\*4 + x\*\*2\*sp.exp(x) - x\*\*2 - 2\*x\*sp.exp(x) - 2\*x - sp.exp(x))\*sp.expresultat, resultat.doit()

Out[82]:

Z *x*4 + *x*2*ex − x*2 *−* 2*xex −* 2*x − ex**ex*

!

(*x −* 1)2(*x* + 1)2(*ex* + 1)*dx,* log (*ex* + 1) + *ex x*2 *−* 1

Tamb´e podem usar la funci´o evalf() per obtenir un resultat num`eric amb precisi´o arbitr`aria

In [83]: import sympy as sp

sp.init\_printing()

sp.var("x")

resultat = sp.Integral(sp.exp(-x\*\*2), (x, -sp.oo, sp.oo))

print( resultat.doit().evalf(100)\*\*2 )

print( sp.pi.evalf(100) )

resultat = sp.Integral(sp.exp(-x\*\*2), (x, -1, 1))

resultat, resultat.doit(), resultat.doit().evalf(100)

3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117068 3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117068

Out[83]:

Z 1

*e−x*2*dx, √~~π~~* erf (1)*,* 1*.*493648265624854050798934872263706010708999373625212658055308997917210655123545662*−*1

1.12 Derivaci´o amb sympy

1.12.1 Funci´o diff()

sympy permet obternir la derivada d’una expressi´o simb`olica mitjan¸cant la funci´o diff(). La forma m´es senzilla d’aplicar-la ´es per a calcular una derivada simple respecte una ´unica variable:

In [84]: import sympy as sp

sp.init\_printing()

sp.var("x")

print( "Derivant sin(x)\*exp(x)" )

sp.diff(sp.sin(x)\*sp.exp(x),x)

Derivant sin(x)\*exp(x)

Out[84]:

*ex*sin (*x*) + *ex*cos (*x*)

La funci´o diff() permet tamb´e fer derivades de graus superiors; nom´es cal indicar el grau al cridar la funci´o.

33

In [85]: import sympy as sp

sp.init\_printing()

sp.var("x")

print( "Derivada tercera de x\*\*4" )

sp.diff(x\*\*4,x,3)

Derivada tercera de x\*\*4

Out[85]:

24*x*

De la mateixa manera, tamb´e permet fer derivades parcials respecte varies variables, per exemple: *∂*

*∂x∂y∂zex,y,x*

In [86]: import sympy as sp

sp.init\_printing()

sp.var("x")

sp.var("y")

sp.var("z")

print( "Derivada parcial de exp(x\*y\*z) respecte x,y,z" )

sp.diff(sp.exp(x\*y\*z),x,y,z)

Derivada parcial de exp(x\*y\*z) respecte x,y,z

Out[86]:

*x*2*y*2*z*2 + 3*xyz* + 1*exyz*

1.12.2 Derivades com a s´ımbols

En cas que la derivada no es vulgui calcular expl´ıcitament es pot usar Derivative() que retorna la derivada com una expressi´o simb`olica de sympy. Posteriorment la derivada es pot calcular usant doit().

In [87]: import sympy as sp

sp.init\_printing()

sp.var("x")

print( "Derivant sin(x)\*exp(x)" )

D = sp.Derivative(sp.sin(x)\*sp.exp(x),x,3)

D, D.doit()

Derivant sin(x)\*exp(x)

Out[87]:

*d*3

*dx*3(*ex*sin (*x*))*,* 2 (*−* sin (*x*) + cos (*x*)) *ex* 34