14 - Sympy

November 30, 2015

Figure 1: BY-SA

Authors : *Sonia Estradé*
José M. Gómez
Ricardo Graciani
Franc Guell
Manuel López
Xavier Luri
Josep Sabater

1 Llibreria sympy

La llibreria `sympy` proporciona eines per al **càlcul simbòlic**. En altres paraules, permet fer amb l'ordinador manipulacions de símbols algebraics (fórmules, 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 numèrics sinó a funcions i ens permet operar amb elles. Pot trobar-se una descripció 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 següent importació:

```
import sympy
```

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

```
import sympy as sp
```

1.1 Funcionalitat bàsica: símbols i expressions

El primer pas per a utilitzar sympy és definir els símbols que volem manipular. Cal identificar algunes variables de Python com a objectes que representen símbols (o variables en el sentit matemàtic).

Per fer-ho es poden fer servir les funcions de `sympy`:

- `var("x")` defineix un únic símbol
- `symbols("x y z")` defineix com a símbols les paraules contingudes a la cadena

Veiem-ho en un exemple:

In [1]:
```
import sympy as sp
sp.init_printing()

# Definició individual. Fem que "x" sigui un símbol
sp.var("x")

# Definició múltiple. Fem que "a", "b" i "c" siguin símbols
a,b,c = sp.symbols("a b c")


# A partir d'aquests símbols definim una equació de segon grau
# Noteu que assignem la combinació de símbols a una nova variable
equacio = a*x**2 + b*x + c

# La resolem simbòlicament amb sympy
equacio, sp.solve(equacio,x)
```

Out[1]:

$$\left(ax^2 + bx + c, \quad \left[\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right), \quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right]\right)$$

En ambdós casos, el que Phython està fent és:

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 diferència entre `var()` i `symbols()`. `var()` crea d'una manera automàtica una variable de Python amb el mateix nom i `symbols()` només crea els objectes que hem d'assignar necessàriament a una o més variables per poder utilitzar-les.

Podem fer que el nom de la variable no coincideixi amb el nom del símbol, però cal anar amb compte per que això pot portar a confusions.

In [2]:
```
y, z = sp.symbols('z y')
(z, y)
```

Out[2]:

$$(y, \quad z)$$

Noteu com al imprimir `(x, y)` Python ens mostra els noms dels símbols que representen, en aquest cas amb `y, x = sp.symbols('x y')` hem fet que la variable `x` representi al símbol de nom **y** (o variable matemàtica **y**) i la variable `y` al símbol de nom **x** (o variable matemàtica **x**).

Una vegada que hem definit els nostres símbols els podem combinar mitjançant els operadors matemàtics habituals per definir expressions algebraiques:

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

Finalment podem fer que `sympy` resolgui la equació que resulta al igualar aquesta expressió a cero.

```
sp.solve(equacio)
```

sense més arguments, la funció `solve()` considera la equació resultant de igualar a cero el seu argument i tractarà de trobar la seva solució. Si `sympy` es capaç de trobar la solució (o solucions) ens tornarà 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 equació 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]:

$$[-1, \quad 1]$$

Nota: la funciò `sp.init_printing()` configura `sympy` per que la impressió dels símbols i de les expressions es faci de la forma més 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` és la creació d'expressions algebraiques directament a partir d'una cadena de text. En aquest cas es creen els símbols necessaris i es construeix l'expressió en un sol pas, de manera semblant al que fem amb `var()`.

Per fer-ho cal usar la funció `sympify()`.

```
In [5]: import sympy as sp
        sp.init_printing()

        sp.var("x")

        # Creem una expressió, una equació de segon grau
        equacio = sp.sympify("A*x**2+B*x+C")

        # Exemple 1
        equacio,sp.solve(equacio,x)
```

Out[5]:

$$\left(Ax^2 + Bx + C, \quad \left[\frac{1}{2A}\left(-B + \sqrt{-4AC + B^2}\right), \quad -\frac{1}{2A}\left(B + \sqrt{-4AC + B^2}\right)\right]\right)$$

En aquest cas la nostra equació $Ax^2 + Bx + C = 0$ té 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 això hem utilitzat la funció `solve()` amb un segon argument, `x`. Es posible no fer-ho i deixar que `sympy` seleccione alguna.

```
In [6]: # Exemple 2
        sp.solve(equacio)
```

Out[6]:

$$\left[\left\{ A : -\frac{1}{x^2} (Bx + C) \right\} \right]$$

O bé indicar-la directament.

In [7]:
```
# Exemple 3

sp.var('A')
sp.var('B')
sp.var('C')

solutionA = sp.solve(equacio, A)
solutionB = sp.solve(equacio, B)
solutionC = sp.solve(equacio, C)
{ A: solutionA, B: solutionB, C: solutionC }
```

Out[7]:

$$\left\{ A : \left[-\frac{1}{x^2} (Bx + C) \right], \quad B : \left[-Ax - \frac{C}{x} \right], \quad C : [-x (Ax + B)] \right\}$$

Cal destacar que `sympy` no necessita que les variables `A`, `B` i `C` estén definides que per poder resoldre la equació (Exemples 1 i 2). Només ens fa falta definir-les quan volem indicar-li que les utilitzi com a variables independents per resoldre la nostre equació (Exemple 3). Per això necessitem variables de Python que facin referencia als símbols corresponents:

```
sp.var('A')
sp.var('B')
sp.var('C')
```

Si volem fer servir lletres gregues a les nostres expressions cal que creem els símbols utilitzant els seus noms en anglès. De la mateixa manera `sympy` i `sympify()` reconeixen la major part de las funcions matemàtiques habituals.

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

Out[8]:

$$\sin(\theta)$$

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

Out[9]:

$$A \cos(\phi)$$

Com representa `sympy` internament les expressions.

In [10]:
```
expressio.args
```

Out[10]:

$$(A, \quad \cos(\phi))$$

```
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 funció `print()` per mostrar un símbol o una expressió Python ens mostra la seva representació com a cadena de text, sense fer servir la notació LaTeX.

1.1.2 Com representar gràficament una expressió

Considerem la equació 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]:

$$ax + b$$

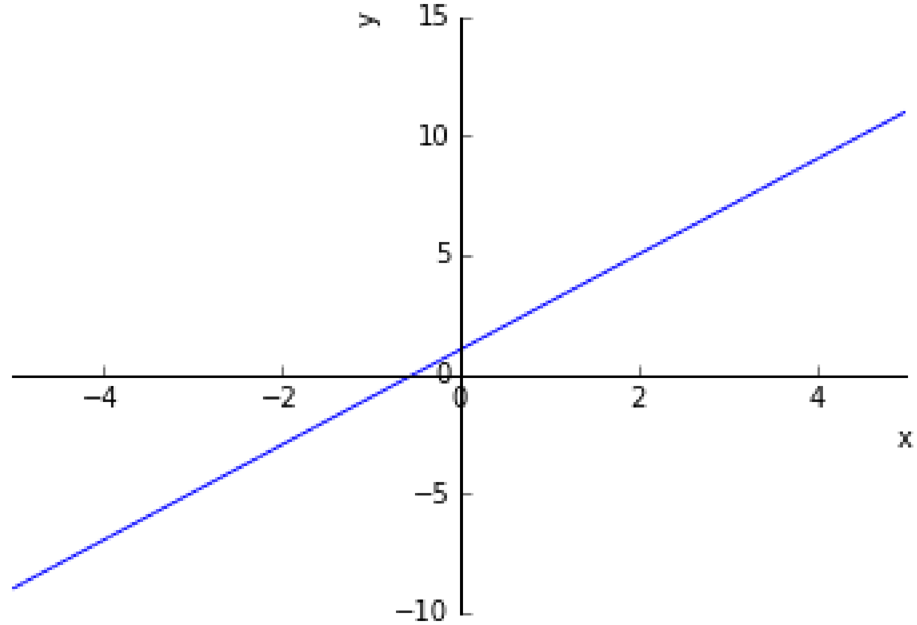
Podem assignar valors als símbols a i b i representar gráficament el resultat

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

Out[13]:

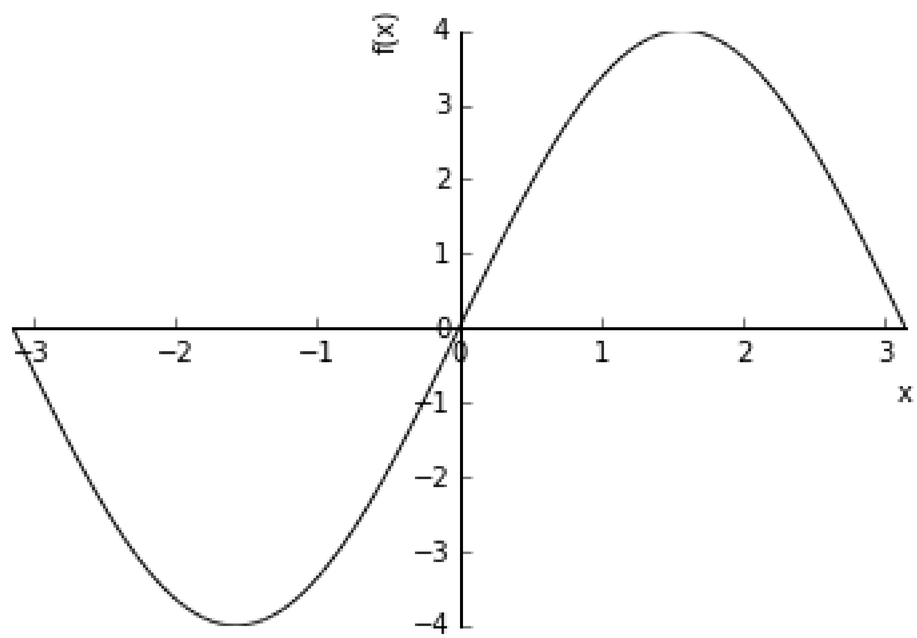
$$2x + 1$$

```
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')
```

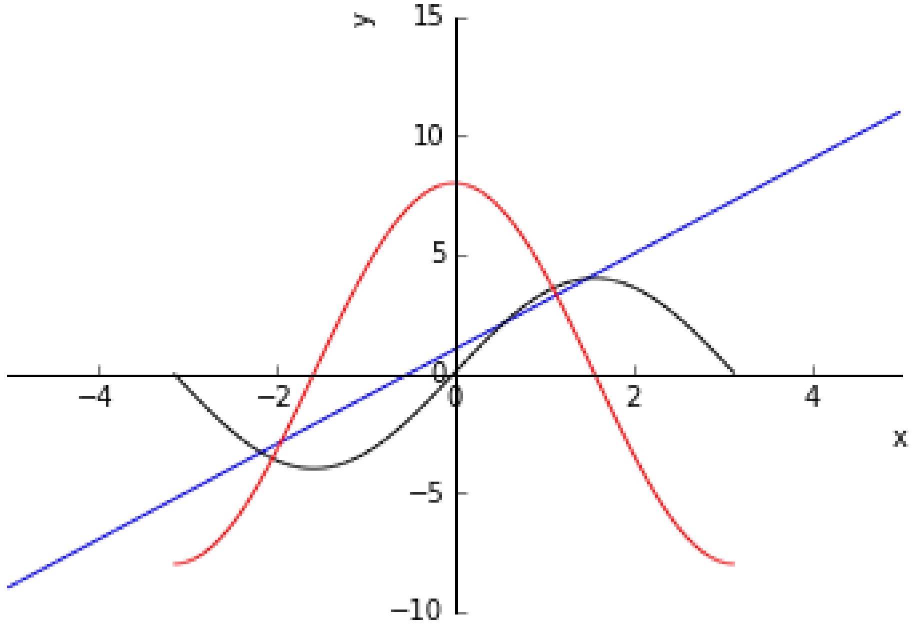


Ara podem afegir més funcions a la mateixa gràfica.

```
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') )
```



```
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 simbòliques

En els exemples anteriors ja hem vist alguna manipulació simbòlica amb `sympy`. Hem definit quatre símbols i amb ells hem construit una equació de segon grau, que després hem resolt algebraicament.

Usant `sympy` podem realitzar una gran varietat de manipulacions simbòliques. Aquí veurem alguns exemples, però remeteu-vos al tutorial de `sympy` per una descripció més completa.

1.2.1 Simplificant expressions

La funció `simplify()` permet la simplificació d'expressions algebraiques. Aplica diverses tècniques per a intentar reduïr l'expressió donada a una forma més senzilla. Podeu trobar més detalls sobre la simplificació a:

Sympy tutorial: simplification

Podem veure algunes aplicacions en els exemples següents.

```
In [17]: import sympy as sp

         # Simplificació usant igualtats trigonomètriques
         expr = sp.sympify("sin(x)**2 - cos(x)**2")
         print("Expressió: ", expr)
         print("Simplificació: ", sp.trigsimp(expr))
         print("Simplificació: ", sp.simplify(expr))
         print(10*'-')

         # Simplificació de quocients de polinomis
```

```
        expr = sp.sympify("(x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)")
        print("Expressió: ", expr)
        print("Simplificació: ", sp.ratsimp(expr))
        print("Simplificació: ", sp.simplify(expr))

        print(10*'-')
        # Simplificació usant propietats de funcions
        expr = sp.sympify("gamma(x)/gamma(x - 2)")
        print("Expressió: ", expr)
        print("Simplificació: ", sp.combsimp(expr))
        print("Simplificació: ", sp.simplify(expr))
```

```
Expressió:  sin(x)**2 - cos(x)**2
Simplificació:  -cos(2*x)
Simplificació:  -cos(2*x)
----------
Expressió:  (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
Simplificació:  x - 1
Simplificació:  x - 1
----------
Expressió:  gamma(x)/gamma(x - 2)
Simplificació:  (x - 2)*(x - 1)
Simplificació:  (x - 2)*(x - 1)
```

Noteu que a més de `simplify()` hem fet servir `trigsimp()`, `ratsimp()` o `combsimp()`. Quan utilitzem directament `simplify()` Python tractarà de utilitzar el mètodo més apropiat per a la expressió.

A continuació veurem algunes aplicacions de les tècniques de simplificació en el cas de polinomis.

1.2.2 Factorització de polinomis

La factorització de polinomis es pot fer usant la funció `factor()`:

```
In [18]: import sympy as sp

        # Factorització d'un polinomi
        p = sp.sympify("x**2 + 2*x + 1")
        print("Polinomi: ", p)
        print("Factorització: ", sp.factor(p))

        print(10*'-')
        # També funciona amb polinomis de més d'una variable
        p = sp.sympify("x**2*z + 4*x*y*z + 4*y**2*z")
        print("Polinomi: ", p)
        print("Factorització: ", sp.factor(p))
```

```
Polinomi:  x**2 + 2*x + 1
Factorització:  (x + 1)**2
----------
Polinomi:  x**2*z + 4*x*y*z + 4*y**2*z
Factorització:  z*(x + 2*y)**2
```

Per polinomis més complexos, amb diverses variables, és possible indicar el ordre de precedència dels símbols per a la factorització:

```
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")
```

```
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 Expansió de polinomis

També podem realitzar el procés invers a la factorització. Podem convertir un producte de polinomis en el polinomi equivalent usant la funció `expand()`:

```
In [20]: import sympy as sp

         sp.var("x")

         # Expansió 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'expressió 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 Substitució de símbols: `subs()`

Quan tenim una expressió algebràica podem substituir els simbols que la composen per altres símbols o per valors numérics usant la funció `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*'-')
         # També podem fer una substitució per un valor numèric
         q = sp.sympify("y*(x + 1)**2")
         print("q = ", q)
         print("q subs. = ", q.subs(x,3.))
```

```
p =  (x + 1)**2
p subs. =  (cos(x) + 1)**2
----------
q =  y*(x + 1)**2
q subs. =  16.0*y
```

1.4 Avaluació numèrica d'expressions: `evalf()`

Donada una expressió algebraica podem avaluar-la donant valors numèrics als seus símbols usant la funció `evalf()`:

```
In [22]: import sympy as sp

         # Definim una expressió i l'avaluem per a un valor concret de x
         sp.var("x")
         expr = 2*x**2+2*x-1
         print("Expressió: ", expr)
         print("Valor per a x=2: ", expr.evalf(subs={x:2}))
```

```
Expressió:  2*x**2 + 2*x - 1
Valor per a x=2:  11.0000000000000
```

```
In [23]: # També es pot fer si hi ha més d'un símbol
         sp.var("y")
         expr = y + 2.*x**2
         print("Expressió: ", expr)
         print("Valor per a x=2 y=5: ",expr.evalf(subs={x:2,y:5}))
```

```
Expressió:  2.0*x**2 + y
Valor per a x=2 y=5:  13.0000000000000
```

La funció `evalf()` permet a més que s'especifiqui el número de xifres significatives del càlcul, donat que `sympy` admet càlculs en coma flotant de precisió arbitrària.

```
In [24]: import sympy as sp

         # Avaluació d'arrels quadrades
         sp.var("x")
         expr = sp.sqrt(x)
         print("Expressió: ",expr)
         print(10*'-')
         print("Avaluació sqrt(2) amb 100 xifres: ",expr.evalf(100,subs={x:2}))
         print(10*'-')
         print("Avaluació sqrt(3) amb 100 xifres: ",expr.evalf(100,subs={x:3}))
```

```
Expressió:  sqrt(x)
----------
Avaluació sqrt(2) amb 100 xifres:  1.414213562373095048801688724209698078569671875376948073176679737990
----------
Avaluació sqrt(3) amb 100 xifres:  1.7320508075688772935274463415058723669428052538103806280558069794
```

```
In [25]: print(type(expr.evalf(100,subs={x:3})))
```

```
<class 'sympy.core.numbers.Float'>
```

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

```
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 = 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211
----------
e = 2.7182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664
```

```
In [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
```

```
 |              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__(...)
 |
```

```
 |  __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;
```

```
    - 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)
```

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

```
    ========

    >>> 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
```

```
    >>> 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)
    1.2676506002282e+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 Resolució d'equacions: `solve()`

Sympy implementa la resolució algebraica d'equacions mitjançant la funció `solve()`.

Nota: la funció `solve()` assumeix per defecte que l'expressió que reb s'iguala a zero per a plantejar l'equació.

Exemple 1: equació polinòmica

$$x^3 + 2x - 2 = 0$$

In [28]:
```python
import sympy as sp
sp.init_printing()

# Definim el polinomi
p = sp.sympify("x**3 + 2*x - 2")
```

```
# Resolem
solucio = sp.solve(p, x)
print("Polinomi: ", p)
print("Solució:  ", solucio)
solucio
```

```
Polinomi:  x**3 + 2*x - 2
Solució:   [(-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]:

$$\left[\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\frac{\sqrt{105}}{9}}-\frac{2}{3\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\frac{\sqrt{105}}{9}}},\quad -\frac{2}{3\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\frac{\sqrt{105}}{9}}}+\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{1+\frac{\sqrt{105}}{9}},\quad -\right.$$

A vegadas pot ser útil fer servir la funció `simplify()`:

In [29]:
```
[ i.simplify() for i in solucio ]
```

Out[29]:

$$\left[\frac{\sqrt[3]{3}\left(8\sqrt[3]{3}-\left(1+\sqrt{3}i\right)^{2}\left(9+\sqrt{105}\right)^{\frac{2}{3}}\right)}{6\left(1+\sqrt{3}i\right)\sqrt[3]{9+\sqrt{105}}},\quad \frac{\sqrt[3]{3}\left(8\sqrt[3]{3}-\left(1-\sqrt{3}i\right)^{2}\left(9+\sqrt{105}\right)^{\frac{2}{3}}\right)}{6\left(1-\sqrt{3}i\right)\sqrt[3]{9+\sqrt{105}}},\quad \frac{\sqrt[3]{3}\left(-2\sqrt[3]{3}+\left(9+\sqrt{105}\right)^{\frac{2}{3}}\right)}{3\sqrt[3]{9+\sqrt{105}}}\right]$$

També podem fer servir `evalf()` per obtenir un resultat numèric.

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

Out[30]:

$$[-0.385458498529624-1.56388451052696i,\quad -0.385458498529624+1.56388451052696i,\quad 0.770916997059248]$$

Exemple 2: equació trigonomètrica

$$\cos(x)+\sin(x)=0$$

In [31]:
```
import sympy as sp
sp.init_printing()

# Definim l'equació trigonomètrica
e = sp.sympify("cos(x)+sin(x)")

# Resolem
solucio = sp.solve(e, x)
print("Expressió: ",e )
print("Solució: ", solucio)
solucio
```

```
Expressió:  sin(x) + cos(x)
Solució:  [-pi/4, 3*pi/4]
```

Out[31]:

$$\left[-\frac{\pi}{4}, \quad \frac{3\pi}{4}\right]$$

Exemple 3: sistema d'equacions

Si es vol resoldre un sistema d'equacions (lineal o no) les diverses equacions s'han de passar com arguments a la funció `solve()` en forma de llista, tant les equacions com els símbols 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]:

$$\left(\{x : 2, \quad y : 1\}, \quad \left[(0, \quad 0), \quad \left(\frac{\pi}{2}, \quad -1\right), \quad \left(\frac{\pi}{2}, \quad -1\right)\right]\right)$$

Equacions suportades actualment per `sympy` són :
- Polinomis d'una variable,
- Transcendentals, - Combinacions a trossos de les anteriors,
- Sistemes d'equacions polinòmiques lineals, i
- Sistemes que continguin expressions relacionals .

Nota: Quan `solve()` retorna `[]` o un `NotImplementedError` no vol dir que la equació no tingui solució. Solament vol dir que no ha pogut trobar cap. Frequentment aixo vol dir que la solució no es por representar d'una manera simbolica. Per exemple, l'equació $x = cos(x)$ té solució, però no es pot representar simbòlicament mitjançant funcions estàndard.

```
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 Àlgebra lineal amb sympy

Sympy implementa operacions d'àlgebra lineal, és a dir, operacions amb vectors i matrius. De forma similar a numpy inclou una classe específica per a representar matriu, la classe `Matrix` que pot incloure valors simbòlics.

1.7 Creació de matrius

1.7.1 A partir d'una llista

La funció `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]:

$$\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 0 & 2 \end{bmatrix}$$

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]:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

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

Out[36]:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Com és natural en `sympy`, les components d'un objecte `Matrix` poden ser símbols.

```
In [37]: import sympy as sp
         sp.init_printing()

         matrix = sp.Matrix( sp.var('x y z') )
         matrix
```

Out[37]:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

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]:
```python
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]:

$$\left(\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \right)$$

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

In [39]:
```python
import sympy as sp
sp.init_printing()

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

Out[39]:

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

1.7.3 A partir d'una funció

Es pot crear un objecte `Matrix` usant una funció. La funció ha de rebre com a paràmetres els índexs d'una component i retornar el valor de la component que correspongui a aquests índexs. Per crear la funció s'especifiquen les seves mides i la funció a usar.

In [40]:
```python
import sympy as sp

# Definim la funció
def funcio(i1,i2):
    """Reb com a paràmetres els dos índex i1,i2 i assigna com
       a valor de la component la súma dels dos"""
    return i1+i2

# Creem una matriu a partir de la funció
A = sp.Matrix(4,4,funcio)
print("Matriu i1+i2:")
A
```

```
Matriu i1+i2:
```

Out[40]:

$$\begin{bmatrix}0 & 1 & 2 & 3\\1 & 2 & 3 & 4\\2 & 3 & 4 & 5\\3 & 4 & 5 & 6\end{bmatrix}$$

1.8 Accés 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]:

$$\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}$$

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

```
Element 1,2: 4
```

Out[42]:

$$\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\end{bmatrix}$$

1.8.2 Files i columnes

Es pot accedir a les files i columnes d'una matriu usant els mètodes `row()` i `col()`. Aquests mètodes retornen un objecte `Matrix` que conté 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]:

$$\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}$$

In [44]:
```
matriu.row(1)
```

Out[44]:

$$\begin{bmatrix}2 & 3 & 4\end{bmatrix}$$

```
In [45]: matriu.col(2)
```

Out[45]:

$$\begin{bmatrix}0\\4\\7\end{bmatrix}$$

Noteu que retornen una matriu de la forma apropiada.

La matriu transposada també és accessible fàcilment usant el membre `T` de l'objecte `Matrix`:

```
In [46]: matriu, matriu.T
```

Out[46]:

$$\left(\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}, \begin{bmatrix}1 & 2 & 0\\-1 & 3 & 2\\0 & 4 & 7\end{bmatrix}\right)$$

1.9 Operacions bàsiques

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]:

$$\left(\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}, \begin{bmatrix}5 & 6 & 2\\8 & 7 & 7\\5 & 1 & 1\end{bmatrix}\right)$$

```
In [48]: # Suma, resta
         print('M+N, M-N')
         M+N,M-N
```

```
M+N, M-N
```

Out[48]:

$$\left(\begin{bmatrix}6 & 5 & 2\\10 & 10 & 11\\5 & 3 & 8\end{bmatrix}, \begin{bmatrix}-4 & -7 & -2\\-6 & -4 & -3\\-5 & 1 & 6\end{bmatrix}\right)$$

```
In [49]: # Producte de matrius
         print('M*N')
         M*N
```

```
M*N
```

Out[49]:

$$\begin{bmatrix}-3 & -1 & -5\\54 & 37 & 29\\51 & 21 & 21\end{bmatrix}$$

```
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]:

$$\left(\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}\left(\begin{bmatrix}5 & 6 & 2\\8 & 7 & 7\\5 & 1 & 1\end{bmatrix}\right)^{-1}, \begin{bmatrix}-\frac{1}{4} & \frac{1}{108} & \frac{47}{108}\\-\frac{1}{4} & \frac{77}{108} & -\frac{53}{108}\\-\frac{5}{4} & \frac{55}{36} & -\frac{43}{36}\end{bmatrix}\right)$$

```
In [51]: # Exponenciació
         print('M, M**2, M**3')
         M, M**2, M**3
```

```
M, M**2, M**3
```

Out[51]:

$$\left(\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}, \begin{bmatrix}-1 & -4 & -4\\8 & 15 & 40\\4 & 20 & 57\end{bmatrix}, \begin{bmatrix}-9 & -19 & -44\\38 & 117 & 340\\44 & 170 & 479\end{bmatrix}\right)$$

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

```
M, M*2, M/2
```

Out[52]:

$$\left(\begin{bmatrix}1 & -1 & 0\\2 & 3 & 4\\0 & 2 & 7\end{bmatrix}, \begin{bmatrix}2 & -2 & 0\\4 & 6 & 8\\0 & 4 & 14\end{bmatrix}, \begin{bmatrix}\frac{1}{2} & -\frac{1}{2} & 0\\1 & \frac{3}{2} & 2\\0 & 1 & \frac{7}{2}\end{bmatrix}\right)$$

Noteu que `Matrix` no admet operacions de suma o resta amb valors numèrics (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)
```

```
M, M+2, M-2
```

Out[53]:

$$\left(\begin{bmatrix} 1 & -1 & 0 \\ 2 & 3 & 4 \\ 0 & 2 & 7 \end{bmatrix}, \quad \begin{bmatrix} 3 & 1 & 2 \\ 4 & 5 & 6 \\ 2 & 4 & 9 \end{bmatrix}, \quad \begin{bmatrix} -1 & -3 & -2 \\ 0 & 1 & 2 \\ -2 & 0 & 5 \end{bmatrix} \right)$$

I obviament, amb **sympy** totes aquestes operacions es poden fer amb matrius simbòliques:

```
In [54]: import sympy as sp
         sp.init_printing()

         # Definim una matriu simbòlica
         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]:

$$\begin{bmatrix} x & 2x & x \\ x^2 & 3 & 4 \\ x-1 & x & x^3 \end{bmatrix}$$

```
In [55]: M**2
```

Out[55]:

$$\begin{bmatrix} 2x^3 + x^2 + x(x-1) & 3x^2 + 6x & x^4 + x^2 + 8x \\ x^3 + 3x^2 + 4x - 4 & 2x^3 + 4x + 9 & 5x^3 + 12 \\ x^3(x-1) + x^3 + x(x-1) & x^4 + 2x(x-1) + 3x & x^6 + x(x-1) + 4x \end{bmatrix}$$

Per defecte **sympy** no intenta simplificar l'expressió resultant, pot ser instruït per fer-ho utilitzant el mètode **simplify( )**:

```
In [56]: N = M**2
         N.simplify()
         N
```

Out[56]:

$$\begin{bmatrix} x\left(2x^2 + 2x - 1\right) & 3x(x+2) & x\left(x^3 + x + 8\right) \\ x^3 + 3x^2 + 4x - 4 & 2x^3 + 4x + 9 & 5x^3 + 12 \\ x\left(x^3 + x - 1\right) & x\left(x^3 + 2x + 1\right) & x\left(x^5 + x + 3\right) \end{bmatrix}$$

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

Out[57]:

$$\left[\begin{matrix} \frac{2x}{-2x^2+3} - \frac{1}{2x^5-4x^3-x+5}\left(1 - \frac{-2x^2+8}{-2x^2+3}\right)\left(2x^2-3\right)\left(\frac{x(-x+2)}{-2x^2+3} - \frac{1}{x}(x-1)\right) + \frac{1}{x} & \frac{\left(1-\frac{-2x^2+8}{-2x^2+3}\right)(-x+2)\left(2x^2-3\right)}{(-2x^2+3)(2x^5-4x^3-x+5)} - \frac{2}{-2x^2+3} \\ -\frac{x}{-2x^2+3} - \frac{\left(-x^2+4\right)\left(2x^2-3\right)\left(\frac{x(-x+2)}{-2x^2+3} - \frac{1}{x}(x-1)\right)}{(-2x^2+3)(2x^5-4x^3-x+5)} & \frac{(-x+2)\left(-x^2+4\right)\left(2x^2-3\right)}{(-2x^2+3)^2(2x^5-4x^3-x+5)} + \frac{1}{-2x^2+3} \\ \frac{\left(2x^2-3\right)\left(\frac{x(-x+2)}{-2x^2+3} - \frac{1}{x}(x-1)\right)}{2x^5-4x^3-x+5} & -\frac{(-x+2)\left(2x^2-3\right)}{(-2x^2+3)(2x^5-4x^3-x+5)} \end{matrix} \right.$$

In [58]:
```
N = M**-1
N.simplify()
N
```

Out[58]:

$$\left[\begin{matrix}\frac{-3x^{2}+4}{2x^{5}-4x^{3}-x+5} & \frac{x\left(2x^{2}-1\right)}{2x^{5}-4x^{3}-x+5} & -\frac{5}{2x^{5}-4x^{3}-x+5}\\\frac{x^{5}-4x+4}{x(2x^{5}-4x^{3}-x+5)} & \frac{-x^{3}+x-1}{2x^{5}-4x^{3}-x+5} & \frac{-x^{2}+4}{2x^{5}-4x^{3}-x+5}\\\frac{-x^{3}+3x-3}{x(2x^{5}-4x^{3}-x+5)} & \frac{-x+2}{2x^{5}-4x^{3}-x+5} & \frac{2x^{2}-3}{2x^{5}-4x^{3}-x+5}\end{matrix}\right]$$

1.10 Operacions avençades

A més de les operacions anteriors `sympy` proporciona també diverses operacions avençades d'àlgebra lineal.

1.10.1 Transposició

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]:

$$\left(\left[\begin{matrix}x & 2x & 0\\2 & x-1 & 4\\x+1 & 2 & 7\end{matrix}\right], \quad \left[\begin{matrix}x & 2 & x+1\\2x & x-1 & 2\\0 & 4 & 7\end{matrix}\right]\right)$$

1.10.2 Determinants

Es calculen amb el mètodo `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]:

$$\left(\left[\begin{matrix}x & 2x & 0\\2 & x-1 & 4\\x+1 & 2 & 7\end{matrix}\right], \quad 15x^{2}-35x\right)$$

1.10.3 Vectors generadors del nucli (kernel)

El mètode `nullspace()` permet obtenir una base (vectors generadors) del nucli (subespai vectorial que té per imatge el vector zero) de l'aplicació lineal representada per la matriu.

$$M \cdot v = 0$$

```
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]:

$$\left(\begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 10 & 0 & 0 & 1 \end{bmatrix}, \quad \left[\begin{bmatrix} -15 \\ 6 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ -\frac{1}{2} \\ 0 \\ 0 \\ 1 \end{bmatrix} \right] \right)$$

```
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

El mètode `eigenvals()` permet trobar els vectors propis d'una matriu. Retorna un diccionari que conté 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]:

$$\left(\begin{bmatrix} 3 & -2 & 4 & -2 \\ 5 & 3 & -3 & -2 \\ 5 & -2 & 2 & -2 \\ 5 & -2 & -3 & 3 \end{bmatrix}, \quad \{-2 : 1, \quad 3 : 1, \quad 5 : 2\} \right)$$

De forma similar el mètode `eigenvects()` permet calcular els vectors propis de la matriu. La funció retorna els valors propis, la seva multiplicitat i els vectors propis.

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

Out[64]:

$$\left[\left(-2, \quad 1, \quad \left[\begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right] \right), \quad \left(3, \quad 1, \quad \left[\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right] \right), \quad \left(5, \quad 2, \quad \left[\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix} \right] \right) \right]$$

1.10.5 Diagonalització

La funció `diagonalize()` permet diagonalitzar una matriu. Donada una matriu M retorna una tupla (P, D) de manera que $M = PDP^{-1}$.

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

```
M, (P,D)     Noteu que la diagonal de D conté els valors propis
```

Out[65]:

$$\left(\begin{bmatrix} 3 & -2 & 4 & -2 \\ 5 & 3 & -3 & -2 \\ 5 & -2 & 2 & -2 \\ 5 & -2 & -3 & 3 \end{bmatrix}, \quad \left(\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \quad \begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix} \right) \right)$$

1.10.6 Resolucio de sistemes d'equacions

Es poden resoldre a partir d'una matriu mitjançant el mètode de descomposició LU inclòs a la classe `Matrix` o bé simplement a partir de la matriu inversa del sistema.

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

$$\begin{array}{rcrcrcl} x & - & y & + & z & = & 4 \\ 2x & + & y & - & 3z & = & 1 \\ 7x & - & y & - & 3z & = & 14 \end{array}$$

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.
```

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]:

$$\left\{x : \frac{2z}{3} + \frac{5}{3}, \quad y : \frac{5z}{3} - \frac{7}{3}\right\}$$

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]:

$$\left\{x : \frac{2z}{3} + \frac{5}{3}, \quad y : \frac{5z}{3} - \frac{7}{3}\right\}$$

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]:

$$\left(\begin{bmatrix} x - y + z \\ 2x + y - 3z \\ 7x - y - 3z \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 14 \end{bmatrix}, \quad \left[\left\{x : \frac{2z}{3} + \frac{5}{3}, \quad y : \frac{5z}{3} - \frac{7}{3}\right\}\right]\right)$$

Ara una equació 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]:

$$\left(\begin{bmatrix} 1 & -1 & 0 \\ 2 & 3 & 4 \\ 0 & 2 & 7 \end{bmatrix}, \quad \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} \frac{16}{27} \\ -\frac{11}{27} \\ \frac{7}{27} \end{bmatrix}, \quad \begin{bmatrix} \frac{16}{27} \\ -\frac{11}{27} \\ \frac{7}{27} \end{bmatrix}\right)$$

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]:

$$\left\{x : \frac{16}{27}, \quad y : -\frac{11}{27}, \quad z : \frac{7}{27}\right\}$$

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]:

$$\left\{x : \frac{16}{27}, \quad y : -\frac{11}{27}, \quad z : \frac{7}{27}\right\}$$

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]:

$$\left(\begin{bmatrix} x - y \\ 2x + 3y + 4z \\ 2y + 7z \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad \left[\left\{x : \frac{16}{27}, \quad y : -\frac{11}{27}, \quad z : \frac{7}{27}\right\}\right]\right)$$

1.11 Integració amb simpy

1.11.1 Intregrals indefinides

sympy permet fer integració simbòlica de funcions (integrals indefinides) mitjançant la funció integrate(), que reb com a argument una expressió simbòlica.

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]:

$$\frac{e^x}{2} \sin(x) - \frac{e^x}{2} \cos(x)$$

Aquesta mateixa funció permet fer integrals múltiples:

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]:

$$\frac{\pi}{4} \operatorname{erf}\left(x\right) \operatorname{erf}\left(y\right)$$

1.11.2 Integrals definides

La mateixa funció `integrate()` permet també calcular integrals definides indicant els límits d'integració. Per exemple la integral

$\int_0^\pi sin(x)dx = -cos(x)|_0^\pi = 2$

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 límit d'integració infinit amb el símbol `simpy.oo`. L'exemple següent implementa la integral:

$\int_0^\infty e^{-x}dx = -e^{-x}|_0^\infty = 1$

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:

$\int_0^\infty \int_0^\infty e^{-x^2-y^2}dxdy = \pi$

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))
```

Out[78]:

$$\pi$$

Els límits d'integracio poden ser símbols, i el resultat s'expressa en funció d'ells. Per exemple $\int_0^a sin(x)dx = -cos(a) + 1$

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 símbol que utilitzem com a límit de integración.

En l'exemple següent el resultat indica dues possibilitats per què la integral no convergeix si la part real de a no és > 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]:

$$\begin{cases} \Gamma(a+1) & \text{for } -\Re a < 1 \\ \int_0^\infty x^a e^{-x}\, dx & \text{otherwise} \end{cases}$$

1.11.3 Integrals com a símbols

En cas que la integral no es vulgui avaluar, obtenint la funció o el valor numèric corresponent, es pot usar `Integral()` que retorna la integral com una expressió simbòlica de `sympy`. Posteriorment la integral es pot avaluar usant `doit()`

Per exemple, en el cas d'integració 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]:

$$\left(\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-x^2-y^2}\, dx\, dy, \quad \pi\right)$$

I un exemple en el cas d'integració simbòlica:

```
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.exp
         resultat, resultat.doit()
```

Out[82]:

$$\left(\int \frac{\left(x^{4} + x^{2} e^{x} - x^{2} - 2 x e^{x} - 2 x - e^{x}\right) e^{x}}{\left(x - 1\right)^{2} \left(x + 1\right)^{2} \left(e^{x} + 1\right)}\, dx, \quad \log{\left(e^{x} + 1 \right)} + \frac{e^{x}}{x^{2} - 1}\right)$$

També podem usar la funció `evalf()` per obtenir un resultat numèric amb precisió arbitrària

```
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]:

$$\left(\int\limits_{-1}^{1} e^{- x^{2}}\, dx, \quad \sqrt{\pi} \operatorname{erf}{\left(1 \right)}, \quad 1.493648265624854050798934872263706010708999373625212658055308997917210655123545662$$

1.12 Derivació amb sympy

1.12.1 Funció `diff()`

`sympy` permet obternir la derivada d'una expressió simbòlica mitjançant la funció `diff()`. La forma més senzilla d'aplicar-la és per a calcular una derivada simple respecte una única 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]:

$$e^{x} \sin{\left(x \right)} + e^{x} \cos{\left(x \right)}$$

La funció `diff()` permet també fer derivades de graus superiors; només cal indicar el grau al cridar la funció.

```
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]:

$$24x$$

De la mateixa manera, també permet fer derivades parcials respecte varies variables, per exemple: $\frac{\partial}{\partial_x \partial_y \partial_z} e^{x,y,x}$

```
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]:

$$\left(x^{2} y^{2} z^{2} + 3 x y z + 1\right) e^{x y z}$$

1.12.2 Derivades com a símbols

En cas que la derivada no es vulgui calcular explícitament es pot usar `Derivative()` que retorna la derivada com una expressió simbòlica 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]:

$$\left(\frac{d^{3}}{d x^{3}} \left(e^{x} \sin{\left(x \right)}\right), \quad 2 \left(- \sin{\left(x \right)} + \cos{\left(x \right)}\right) e^{x}\right)$$