L1 - Calcul Scientifique

Youssef Chahir
youssef.chahir@unicaen.fr

Université de Caen Normandie

Chapitre I : SymPy
une bibliothèque Python
pour le calcul formel/symbolique

Expérimenter les mathématiques avec Python

Plan

- Rappel :
  - Les Symboles Sympy
  - Les expressions de la bibliothèque Sympy
- Opérations sur les expressions
- Traitement des nombres rationnels
- Résolution d'équations
- Calcul intégrale
- Calcul des dérivées
- Calcul matriciel

Symboles Sympy

from sympy import *

- Pour définir un **symbol x** , on utilise la commande: **Symbol**
  x = Symbol('x') # notez le S en majuscule
  ou
  import sympy as sy
  x = sy.Symbol('x')

- Plusieurs symbols sympy en même temps, on utilise la commande: **symbols**
  x, y, z = symbols('x y z') # notez le 's' en minuscule

- **dir(x)** : commande python qui vous donne un aperçu des __méthodes__ que possède l'objet (ici le symbole) x.

Numérique

- Déclarer ces variables

```
In[2] : t
(...)
NameError : name 't ' is not defined

In[3] : t = symbols('t') ; type(t)
Out[4] : sympy.core.symbol.Symbol

In[5] : x, y = Symbols('x y') ; expr = x**2 + y**2
```

- Valeur numérique

```
In[7] : math.sqrt(20)
Out[8] : 4.47213595499958

In[9] : float(sqrt(20))
Out[10] : 4.47213595499958
```

- Valeur symbolique

```
In[11] : import sympy
In[12] : sympy.sqrt(20)
Out[13] : 2√(5)
```


Variables et expressions symboliques

```
import sympy as sy

# Définir une seule variable.
x0 = sy.symbols('x0')

# Définir plusieurs variables symboliques simultanément.
x2, x3 = sy.symbols('x2, x3')  # Séparer les symboles par des
virgules,
m, a = sy.symbols('mass acceleration')  # par des espaces,
x, y, z = sy.symbols('x:z')  # ou par des deux-points.
x4, x5, x6 = sy.symbols('x4:7')

# Combinez des variables symboliques pour former des expressions.
expr = x**2 + x*y + 3*x*y + 4*y**3
force = m * a

# Afficher les expressions.
print(expr, force, sep='\n')
```


Sommes et Produits

$$\sum_{i=1}^{4} x + iy \qquad \prod_{i=0}^{5} x + iy$$

```
x, y, i = sy.symbols('x y i')
sy.summation(x + i*y, (i, 1, 4))  # Somme sur i=1,2,3,4.
# Résultat : 4*x + 10*y


sy.product(x + i*y, (i, 0, 5))  # Produit sur i=0,1,2,3,4,5.
# Résultat : x*(x + y)*(x + 2*y)*(x + 3*y)*(x + 4*y)*(x + 5*y)
```


Evaluation d’une expression

- # Expression à développer
- expr = sy.expand((x + y)**3)
- print(expr)

- # Remplacer la variable symbolique y par l'expression 2x
- expr_substituted = expr.subs(y, 2*x)
- print(expr_substituted)

- # Remplacer x par pi et y par 1
- new_expr = expr.subs({x: sy.pi, y: 1})
- print(new_expr)

- # Évaluation numérique de l'expression
- numerical_result = new_expr.evalf()
- print(numerical_result)

- # Évaluation numérique de l'expression en fournissant des valeurs pour chaque variable
- numerical_result_subs = expr.evalf(subs={x: 1, y: 2})
- print(numerical_result_subs)

Expressions sympy

- **On peut aussi faire des opérations sur les expressions comme factorisation, développement, simplification …**

1. **Développer une expression à l'aide de la méthode expand()**

```
expr = (x - y)**3
print(expr.expand()) #affiche: x**3 - 3*x**2*y + 3*x*y**2 - y**3
```

2. **Simplification via la méthode simplify()**

```
expr1 = (x + 1)**2
expr2 = x**2 + 2*x + 1
expr3 = sy.simplify( expr2 - expr1 )
print("expr2 - expr1 = " , expr3 ) # affiche: expr2 - expr1 =  0
```

3. **Factoriser une expression à l'aide de la méthode factor()**

```
P = x**2 - 2*x + 1
fact = sy.factor(P)
print(fact) # affiche: (x - 1)**2
```


Décortiquer une expression

- Affichage conforme à nos habitudes mathématiques : la fonction `**srepr**` renvoie une chaîne de caractères.
  – expr=(x + y) * (z + t)
  – srepr(expr) # affiche "Mul(Add(Symbol('t'), Symbol('z')), Add(Symbol('x'), Symbol('y')))"
- 5 opérations de l'arithmétique :
  » + c'est la fonction `**Add**`.
  » x c'est la fonction `**Mul**`.
  » L'exponentiation c'est la fonction `**Pow**`.
  » - et/sont transformées en sommes, produits et puissances.
- une expression est :
  – un **Symbole** ou Un **Entier** ou Un **Rationnel** ou **Add**(expression, Expression, ..., Expression) ou **Mul**(Expression, Expression, ..., Expression) ou **Pow**(Expression, Expression) ou bien d'autres choses, qu'il est hors de question d'examiner ici de façon exhaustive.

Analyse syntaxique des expressions

- Chaque expression SymPy possède deux **champs** qui permettent d'analyser cette expression.
  1. Un champ **`func`** qui est le "type" de l'expression.

     expr = (x + y) * (z + 2)

     expr.func # affiche sympy.core.mul.**Mul**

     # c'est un produit de deux expressions
  2. Un champ **args** est le *n*-uplet des sous-expressions de `l'expression`.

     expr.args # affiche (*z*+2, *x*+*y*)

Simplification

```
1 expr=(y*x**4+6*y*2)*x*y**2
2 factor(expr)
```

$$xy^3\left(x^4+12\right)$$

```
1 expand(expr)
```

$$x^5y^3+12xy^3$$

```
1 collect(expr,x)
```

$$xy^2\left(x^4y+12y\right)$$

La fonction collect rassemble les puissances communes d'un terme dans une expression.

Polynômes

```
P=x**3+4*x**2+5*x+2
display(P)
solve(P,x)
```

$x^3 + 4x^2 + 5x + 2$

$[-2, \ -1]$

```
factor(P)
```

$(x+1)^2 (x+2)$

```
f=(x**2+2*x+3)/(x**3+4*x**2+5*x+2)
display(f)
apart(f)
```

$$\frac{x^2 + 2x + 3}{x^3 + 4x^2 + 5x + 2}$$

$$\frac{3}{x+2} - \frac{2}{x+1} + \frac{2}{(x+1)^2}$$

On peut la décomposer en somme de fractions rationnelles à l'aide de la fonction `apart` de SymPy

Expressions sympy

4. Evaluation d'une expression à l'aide de la méthode evalf()

```
# Default value of pi
print(pi.evalf()) # affiche: 3.14159265358979
# get value of pi with 3 decimal numbers
print(pi.evalf(3))# affiche: 3.14
```

5. Transformer une expression en une fonction à l'aide de la méthode lambdify()

```
expr = x**2 + 3*x + 5
f = sy.lambdify(x , expr)
print("f(0) = " , f(0)) # affiche: f(0) =  5
```

6. Substitution des symbols dans sympy grâce à la **méthode subs**().

```
expr = x**2 + x + 3
# substitution de x par 1
print(expr.subs(x , 1)) # affiche 5
expr = x**2 + x*y + 3*x + y
# substitution de x par y ( remplacement de x par y)
print(expr.subs(x , y)) # affiche: 2*y**2 + 4*y
```

7. Code LaTeX d'une expression à l'aide de la méthode latex()

```
I = Integral(sin(x) , x)
print(latex(I)) # affiche: \int \sin{\left(x \right)}\, dx
```


Impression en mode Latex

```
In[42] : from sympy import integral, latex
In[43] : from sympy.abc import *

In[44] : latex(x**3-x*1)
Out[45] : 'x^3 - x'

In[46] : latex(x**3-x*1), mode='inline')
Out[47] :'$x^3 - x$'

In[48] : latex(x**3-x*1), mode='equation')

In[50] : latex(Integral(x**3-x*1))
```


Expressions sympy

8. Traitement des nombres rationnels grâce à la classe Rational

```
r = sy.Rational(3, 7)
print("la valeur de r est r = " , r) # affiche:  r = 3/7
r1 = sy.Rational(1, 3)
r2 = sy.Rational(1, 2)
s = r1 + r2
p = r1*r2
print("La somme est r1 + r2 = " , s) # affiche: La somme est r1 + r2 =  5/6
print("La produit est r1 x r2 = " , p) # affiche: La produit est r1 x r2 =  1/6
#Transformation d'un nombre décimal en fraction
r = 1.25
print(sy.Rational(r)) # affiche 5/4
```

9. Résoudre les équations en Python avec la méthode sympy.solve()

```
# resoudre l'équation:  x**2 - 3 = 0
print(sy.solve(sy.Eq(x**2 , 3))) # affiche: [-sqrt(3), sqrt(3)]
# définir le système d'équations
eq1 = sy.Eq(2*x + y  , 3)
eq2 = sy.Eq(x - 3*y  , -2)
# resoudre le système d'équation
solution = sy.solve( (eq1 , eq2) , (x,y) )
print( solution ) # affiche: {x: 1, y: 1}
```

Expressions sympy

10. Calcul Intégrale avec la méthode integrate()

```
expr = x**2 + 3*x + 5
# calcul de la primitive
primitive = sy.integrate(expr , x)
print(primitive) # affiche: x**3/3 + 3*x**2/2 + 5*x
```

11. Differentiation des fonctions à l'aide de la méthode diff()

```
expr = x**5
# fonction derivee
derivee1 = sy.diff(expr , x)
print(derivee1) # affiche: 5*x**4

# derivee seconde
derivee2 = sy.diff(expr , x , 2)
print(derivee 2) # affiche: 20*x**3

#derivee d'ordre 3
derivee3 = sy.diff(expr , x , 3)
print(derivee3) # affiche: 60*x**2
```


Définir une fonction

Une expression mathématique comme $g(x) = 1 - x^2$ est représentée par un objet Lambda défini par

A)
```
>>> x= sy.symbols('x')
>>> g= sy.Lambda([x],1-x**2)
>>> g
Lambda(_x, -_x**2 + 1)
>>> g(2)
-3
```

B) `>>> g= -x**2+1`

C) `>>> def  g(x) : return (-x**2+1)`

Fonction numérique

- Limite :

In[23] : limit(exp(1/x)/1-sin(x), x, 0)
Out[24] : ∞

- Dérivation :

In[25] : diff((x**y+y**2),x, y)
Out[26] : $\frac{x^y(y.\log(x)+1))}{x}$

- Intégration :

In[27] : integrate((exp(x)/1-x)
Out[28] :
$-\frac{x^2}{2} + \exp(x)$

- Développement en série :

In[29] : (exp(1/x)/1-sin(x)).series(x, 0, 7)
In[30] : $e^{\frac{1}{x}} - x + \frac{x^3}{6} - \frac{x^5}{120} + o(x^3)$

Fonctions

```
import sympy as sp
from IPython.display import display, Math
from sympy import latex
```

$f(x) = 4x^2 + 3x + 1$

1. Définir une fonction f(x) et appeler f(10) ?
   - x=sp.symbols('x')
   - Plusieurs solutions :
     - A)
       - f=4*(x**2)+3*x+1
       - f.subs(x,10)
       - display(Math(f'f = {latex(f)}'))
     - B)
       - def f(x) :
         return (4*(x**2)+3*x+1)
       - f(10) et f(x).subs(x,10)
     - C)
       - f = sp.Lambda([x], (4*(x**2)+3*x+1)
       - f(10) et f(x).subs(x,10)
       - display(Math(f'f (x)= {latex(f)}'))

Zéros d'une fonction

- Le zéro d'une fonction f = une solution de f(x)=0
  - sympy.solve(f,x)
  - sympy.solve(sp.Eq(g,0))

Convertir une expression Python en SymPy et Réciproquement

- La commande **sympify** traduit une expression Python vers une expression SymPy

  # Transformer l'expression sin(x)^2 en une fonction avec x comme variable.

  - f = sy.lambdify(x, sy.sin(x)**2)
  - print(f(0), f(np.pi/2), f(np.pi), end=' ')

  0.0 1.0 1.4997597826618576e-32

- La commande **lambdify** traduit une expression SymPy en une fonction python

  # Lambdifier une fonction de plusieurs variables.

  - f = sy.lambdify((x, y), sy.sin(x)**2 + sy.cos(y)**2)
  - print(f(0, 1), f(1, 0), f(np.pi, np.pi), end=' ')

  0.2919265817264289 1.708073418273571 1.0

Transformation des fonctions littérales -> Python

- Les fonctions crées avec sympy sont littérales
- Possible d'évaluer la fonction numériquement avec méthode 'subs' mais appels à la fonction python impossible
- Il est possible de construire automatiquement des fonctions python correspondant aux fonctions sympy :

```
x, y,z = sympy.symbols('x,y,z')
f=x**2+y**2+z**2
fp=sympy.lambdify('x,y,z',f)
fp(1,1,1)
```

```
3
```

- La fonction est ensuite connue de python :

```
fp?
```

```
Signature: fp(x, y, z)
Docstring:
Created with lambdify. Signature:

func(arg_0)

Expression:

x**2 + y**2 + z**2

Source code:

def _lambdifygenerated(x,y,z):
    return (x**2 + y**2 + z**2)


Imported modules:
File:      ~/Downloads/Youssef Chahir /<lambdifygenerated-6>
Type:      function
```


Dérivées

- La fonction diff permet de calculer les dérivées de manière littérale

```
x = sympy.symbols('x')
sympy.diff(sympy.sin(x),x)
```

```
cos(x)
```

```
x = sympy.symbols('x')
sympy.diff(sympy.sin(x)*sympy.exp(x)/x,x)
```

```
exp(x)*sin(x)/x + exp(x)*cos(x)/x - exp(x)*sin(x)/x**2
```

- Possibilité de calculer la dérivée nième :

```
x = sympy.symbols('x')
sympy.diff(sympy.sin(x),x,2)
```

```
-sin(x)
```

- Ou encore les dérivées partielles :

```
x,y = sympy.symbols('x,y')
f = x**2+y**2
sympy.diff(f,x)
```

```
2*x
```

Intégration

- La fonction integrate permet le calcul des intégrales :

```
x = sympy.symbols('x')
sympy.integrate(sympy.sin(x)*sympy.cos(x))
```

```
sin(x)**2/2
```

- Il est également possible de calculer la valeur de l'intégrale entre deux bornes (attention à la manière de définir les bornes) :

```
x = sympy.symbols('x')
sympy.integrate(sympy.sin(x),(x,0,sympy.pi/2))
```

```
1
```

- Intégrales multiples :

```
x,y = sympy.symbols('x,y')
sympy.integrate(x**2+y**2,x,y)
```

```
x**3*y/3 + x*y**3/3
```

```
sympy.integrate(x**2+y**2,(x,0,1),(y,0,1))
```

```
2/3
```


Développement limité

- Le développement limité d'une fonction, au voisinage d'un point à un ordre n, donne la fonction polynomiale de degré inférieur ou égal à n qui approxime le mieux la fonction au voisinage du point considéré.

- La fonction series permet d'obtenir ces développements limités de manière littérale :

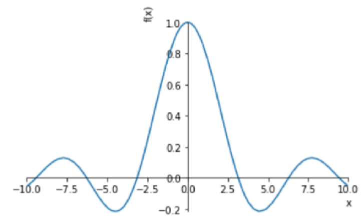
```
x = sympy.symbols('x')
sympy.series(sympy.cos(x),n=4,x0=0)
```

```
1 - x**2/2 + O(x**4)
```


Limites d'une fonction

- La fonction limit de sympy permet de trouver les limites des fonctions
- Par exemple, La fonction sinus cardinal définie par : $\mathrm{sinc}(x) = \dfrac{\sin(x)}{x}$

```
%matplotlib inline
import matplotlib.pyplot as plt
x = sympy.symbols('x')
e = sympy.sin(x)/x
sympy.plot(e)
```



```
sympy.limit(e,x,0)
```

```
1
```

```
sympy.limit(e,x,sympy.oo)
```

```
0
```

- Possibilité de calculer les limites à droite et à gauche

```
sympy.limit(1/x,x,0, dir = "+")
```

```
oo
```

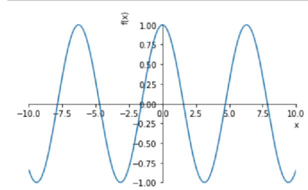
```
sympy.limit(1/x,x,0, dir = "-")
```

```
-oo
```


Tracé de courbes avec sympy

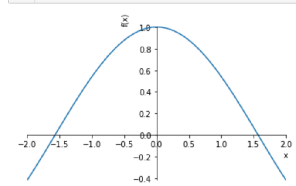
- La fonction sympy plot permet de tracer des graphiques à partir de fonctions littérales

```
x = sympy.symbols('x')
sympy.plot(sympy.cos(x))
```



- Les plages de valeurs des variables littérales peuvent être spécifiées :

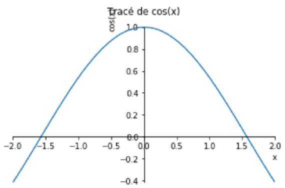
```
x = sympy.symbols('x')
sympy.plot(sympy.cos(x),(x,-2,2))
```



Tracé de courbes avec sympy (suite)

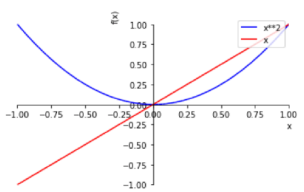
- Possibilité ‘d’annoter’ les courbes et de les sauvegarder

```
ymbols('x')
sympy.cos(x),(x,-2,2),title='Tracé de cos(x)',xlabel='x',ylabel=
```



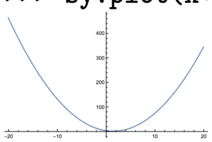
- Possibilité de trace ... er les propriétés de chaque courbe

```
p = sympy.plot(x**2,x,(x,-1,1),legend=True,show=False)
p[0].line_color='b'
p[1].line_color='r'
p.show()
```

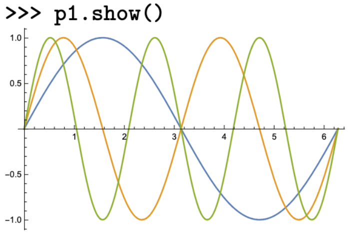


Graphique avec Sympa

```
>>> sy.plot(x**2-3*x+2,(x,-20,30))
```



```
>>> sy.plot(sin(x),sin(2*x),sin(3*x),(x,0,2*pi)) ou
>>> p1=sy.plot(sin(x),(x,0,2*pi),show=False,line_color='b')
>>> p2=sy.plot(sin(2*x),(x,0,2*pi),show=False,line_color='r')
>>> p3=sy.plot(sin(3*x),(x,0,2*pi),show=False,line_color='g')
>>> p1.extend(p2) ; p1.extend(p3) ; print(p1)
>>> p1.show()
```

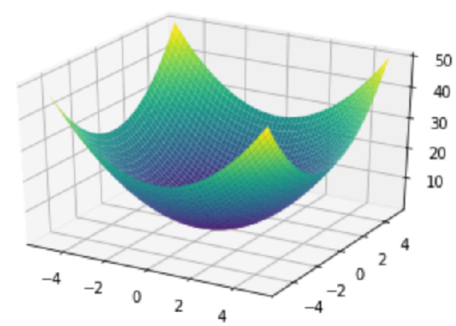


Graphique avec Sympa

- Tracer la courbe que pour y compris entre a et b avec l’option ylim
  - sp.plot(f,ylim=(a,b))
- Tracer la courbe aux abscisses comprises entre a et b
  - sp.plot(f,(x,a,b))
- Plusieurs courbes sur un même graphique
  - p1 = sy.plot(f,(x,-2,2),line_color=‘b',show=False, legend=True)
  - p2 = sy.plot(g,(x,-2,2),line_color=‘r',show=False)
  - p1[0].label='f'
  - p2[0].label='g'
  - p1.append(p2[0])
  - p1.show()

Tracé de surfaces en 3D

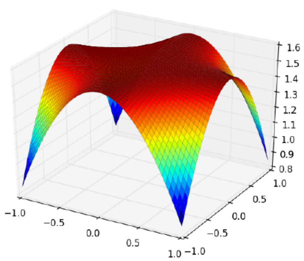
```
x,y = sympy.symbols('x,y')
sympy.plotting.plot3d(x**2+y**2,(x,-5,5),(y,-5,5))
```



Graphe 3D

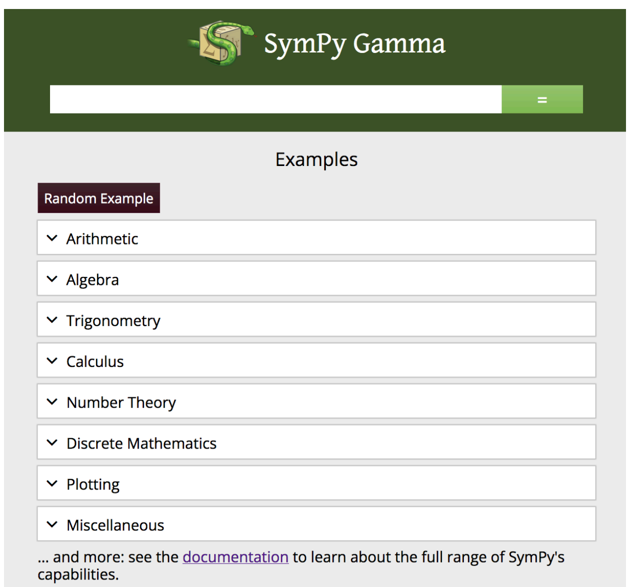
- Les plots avec SymPy dépend de deux bibliothèque Pyglet et matplolib

```
In[10] : from sympy import exp, pi, I
In[11] : from sympy.plotting import plot3d
In[12] : plot3d((pi/2)*exp(I*x*y), (x, -1, 1), (y, -1, 1), nb_of_points_x=100,
nb_of_points_y=100)
```



L'interface SymPy Gamma

https://gamma.sympy.org/



Algèbre de Boole

- Opérations logiques: ET & , OU | et NON, ~
- Représentation d'une fonction Booléenne :
  - $f = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$

```
In [1]: from sympy import *
        init_printing()
        x,y,z=symbols('x,y,z')
        f=(x&y)|(y&z)|(z&x)
        f
```

Out[1]: $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$

- Une fois f définie, on peut chercher quelles sont les variables de f par la méthode **free_symbols** :

```
In [7]: f.free_symbols
```

Out[7]: $\{x, y, z\}$

```
In [2]: f.subs({x:True,y:True,z:False})
```

Out[2]: True

Algèbre de Boole

- La fonction peut aussi être définie comme une Lambda-expression Sympy :

```
In [9]: g=Lambda([x,y,z],(x&y)|(y&z)|(z&x))
In [10]: g(True,True,False)
```

Out[10]: True

- Avec cette définition, les variables de g ne sont plus libres (elles sont liées par Lambda)

```
In [11]: g.free_symbols
```

Out[11]: {}

- Construire une table de vérité:

```
In [18]: for i in cartes([False,True],repeat=3):
             a,b,c=i
             print(i,'\t',g(a,b,c))

(False, False, False)    False
(False, False, True)     False
(False, True, False)     False
(False, True, True)      True
(True, False, False)     False
(True, False, True)      True
(True, True, False)      True
(True, True, True)       True
```

la fonction cartes de sympy nous permet de calculer directement le produit Cartésien qui nous engendre l'ensemble des valuations possibles des variables de la fonction

Algèbre de Boole

- Vérifier les identités remarquables :
  - **Equivalent(A,B)** renvoie vraie si et seulement si A et B sont soit tous deux vrais soit tous deux faux.

Un problème inverse

- Comment faire si on connait la table de vérité et qu'on veut en déduire une fonction Booléenne ?
- On fait appel à la fonction **SOPform** qui cherche une fonction Booléenne qui a la même valeur de vérité :
  - On définit d'abord les valuations qui rendent vraies la table de vérité (on les appelle des **mintermes**)
  - Ensuite, les valuations pour lesquelles les valeurs de vérité ne sont pas précisées (elles sont vraies ou fausses), on les appelle des **dontcares**
  - Toutes les autres valeur sont supposées à faux.
  - **Rq:** On n'est pas obligé de définir les don't care.

Un problème inverse

```
In [21]: x,y,z=symbols('x,y,z')

In [22]: minterms=[[0,1,0]]

In [24]: dontcares=[[0,0,0],[0,0,1]]

In [28]: SOPform([x,y,z],minterms)
```

Out[28]: $y \wedge \neg x \wedge \neg z$

x	y	z	f
0	0	0	*
0	0	1	*
0	1	0	1