Calcul Symbolique
L2 SF, parcours informatique & double cursus MI

Bruno MARTIN,
Université Nice Sophia Antipolis

Section 1

Organisation

Organisation du cours

9 séances ; cf. `http://deptinfo.unice.fr/~bmartin/CS.html`
MCC :

- Rédaction des corrigés des TP (1/3) par (10) groupes
- Partiel le 10/10 (1/3)
- Examen en décembre (1/3)

Différentes thématiques :

- Graphisme
- Ensembles et chaînes de caractères
- Mesure de complexité
- Arithmétique et cryptographie
- Algèbre et codes correcteurs
- Calcul Booléen
- Récursion et programmation dynamique
- Analyse et applications

Section 2

Généralités

Calcul symbolique (ou formel)

A l'intersection des mathématiques et de l'informatique.
Étudie et propose des algorithmes qui travaillent de manière symbolique sur des expressions mathématiques qui ont une représentation finie et exacte : *Computer Algebra System* (ou CAS) en anglais.
Diffère du calcul scientifique (ou numérique) qui travaille sur des nombres qui ont une représentation numérique approchée (en virgule flottante).

Exemple (de calcul symbolique)

calculer la dérivée, la primitive d'une fonction, simplifier une expression algébrique, faire tous les calculs algébriques habituels (matriciel,...)

Bref historique

1950 algorithmes de calcul de dérivée d'une fonction

1970 premiers systèmes de calcul formel : Reduce, Macsyma écrits en LISP

1980 systèmes modernes (avec GUI) : Maple, Mathematica écrits en C

2000 calcul formel dans le libre : sage (http ://www.sagemath.org) utilise Python (et divers autres langages) ou `SymPy` un module de `Python`

Objets du calcul formel

- les nombres
  - les entiers (en précision arbitraire)
  - 100 !
  - les rationnels par un couple p/q de deux entiers
  - les entiers modulo p (un élément de l'ensemble $\{0,...,p-1\}$)
- les polynômes
- les matrices
- et bien d'autres objets de base manipulés par les algorithmes

Que fait un système de calcul formel

Résolution d'équations, factorisation, simplification d'expressions contenant des variables ; les réécrire.
Calcul symbolique dans des structures algébriques (groupes, anneaux, corps,...).
Plus généralement, travailler sur des expression de façon symbolique (plutôt que numérique).

SymPy comme système de calcul symbolique

Deux logiciels libres récents offrent les fonctionnalités du calcul symbolique :

- `SageMath` rassemble un certain nombre de systèmes de calcul symbolique au sein d'une interface standardisée avec un gros noyau
- `SymPy` est un module `Python`. Son intérêt est d'être léger et de fonctionner sur tout système capable d'exécuter `Python`. Il utilise `NumPy` pour le calcul numérique et `Matplotlib` pour le graphisme. On privilégiera l'utilisation des feuilles de calcul fournies par `Jupyter`.

Section 3

Prise en main

Structure de SymPy

Composé de :

- interface web graphique : interagit avec l'utilisateur ; gère le fichier de travail (feuille de calcul ou notebook Jupyter), permet de saisir les instructions et afficher les résultats, y compris l'affichage de graphiques.
- noyau (kernel) : interprète les instructions écrites en `SymPy` ou en `Python`, effectue les calculs et retourne le résultat

Pour lancer la session (et un navigateur), saisir la commande :

```
jupyter notebook
```

soit dans un terminal (sous `linux`, `*BSD`, `MacOS`) soit dans une dans une invite de commande sous `Windows`.
Ou lancer iPython : `C:\ProgramData\Anaconda3\Scripts\ipython3`

Structure d'un notebook

Fichier `SymPy`
Ou notebook, d'extension `.ipynb` est strucuré en cellules (cells) d'une ou plusieurs lignes.
Chaque cellule peut contenir des instructions, des résultats, du texte ou du texte mis en forme.
Une cellule est évaluée par SHIFT+ENTREE

Listing 1 – Importer SymPy

```
1 import sympy as sy
2 sy.init_printing()   # formatte la sortie selon
      l'interface
3 x = sy.symbols('x')
4 r = sqrt(x)
5 r
```

$$\sqrt{x}$$

Utiliser l'aide

Le menu Help propose des tutoriels et de l'aide. Il permet d'accéder aux pages web des différents modules (`NumPy`, `SciPy`, `SymPy`,...

On peut demander de l'aide sur une fonction spécifique par

```
?factorial
```


Calculer

```
>>> 1+1
2
```

Faire des calculs simples, les puissances, les fractions, le modulo :

```
>>> 2**10
1024
>>> sy.Rational(4,6)
2/3
>>> 23 % 12
11
```

Evaluer une valeur numérique

```
>>> sy.Rational(4,6).n(10)
0.6666666667
>>> sy.Rational(4,6).evalf(10)==2/3
True
```


Résultats exacts ou approchés

Avec `SymPy` on travaille avec des valeurs exactes :

```
>>> 3**100
515377520732011331036461129765621272702107522001
>>> sqrt(2)
√2
```

On peut obtenir une valeur approchée avec `.n()` ou `.evalf()`

```
>>>sqrt(2).n()
1.4142135623731
```

Quelques constantes
pi, e (tapé E), i (tapé I), l'infini (∞)

Définir des variables

```
>>> x = 5
>>> x**2
25
```

la valeur de la variable (`Python`) x est modifiable

```
x = 6
```

et on peut l'utiliser dans des expressions

```
>>> pi*x**2
36 pi
>>> del x       # efface la variable x
>>> x           # pour vérifier que x est vide
-------------------------------------------
NameError ...
NameError: name 'x' is not defined
```

Définir des fonctions (à la Python)

```
>>> def f(x):
        return x**2
>>> f(3)
9
>>> f(2+5*x)
-----------------------------------------------
NameError: name 'x' is not defined
>>> x = sy.symbols('x') # on stocke le symbole x dans la variable x
>>> f(2+5*x)
(2+5.x)**2
>>> del f               # efface le contenu de f
>>> def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```


Définir des fonctions (à la SymPy)

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

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

C'est plus une représentation interne. La manière classique de définir la fonction passe par la définition de expr

```
>>> x= sy.symbols('x')
>>> expr = 1-x**2
```

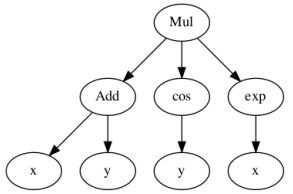
Et de manipuler ensuite cette expression

Expressions

Les fonctions, variables et même les flottants SymPy sont différents de ceux de Python.
Les symboles utilisés pour définir une variable symbolique doivent être déclarés.
`>>> x,y =sy.symbols('x y')`
Conséquence : pour évaluer une expression symbolique de SymPy, il faut la traduire en expression Python
`>>> D=(x+y)*sy.exp(x)*sy.cos(y)`
D hérite du membre droit de la déclaration et n'a pas besoin d'être déclaré comme symbole et devient une fonction de x et y.
Les symboles ne sont pas mutables.

Expressions –2–

Les expressions SymPy sont représentées par un arbre
`>>> D=(x+y)*sy.exp(x)*sy.cos(y)`



qu'on peut aussi obtenir par :

```
>>> sy.srepr(D)
"Mul(Add(Symbol('x'),Symbol('y')),exp(Symbol('x')),
cos(Symbol('y')))"
```

Convertir une expression Python en SymPy et réciproquement

Par la commande `sympify` qui traduit une expression `Python` vers une expression `SymPy`

```
>>> D_s = sy.sympify('(x+y)*exp(x)*cos(y)')
>>> D_s
(x+y)*exp(x)*cos(y)
```

`D_s` se comporte comme `D` défini avant.

On traduit une expression `Sympy` en `Python` par `lambdify`

```
>>> x=sy.symbols('x')
>>> a=12*x**3
>>> f=sy.lambdify(x,a)
>>> f(3)
324
```


Substitution

Opération essentielle du calcul symbolique qui remplace une variable symbolique d'une expression Sympy par une valeur

```
>>> x=sy.symbols('x')
>>> a=12*x**3
>>> a.subs(x,3)
324
```

Permet d'évaluer une expression sans affectation. La substitution est locale à l'expression.

Permet de renommer (ou remplacer) une variable par une autre expression (ou variable)

```
>>> a.subs(x,y)
12*y**3
>>> a
12*x**3
```


Simplification

Par la commande `simplify()` qui fournit une expression plus courte ou plus simple à comprendre.
`simplify()` utilise différentes heuristiques de simplification

```
>>> sy.simplify((x**2-1)/(x+1))
x-1
>>> sy.simplify(1/(x-1) - 1/(x+1))
2 / (x**2 - 1)
>>> sy.simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
```

`simplify()` applique les heuristiques `besselsimp`, `combsimp`, `exptrigsimp`, `hypersimp`, `nsimplify`, `powsim`, `radsimp`, `ratsimpmodprime`, `signsimp`, `simplify`, `simplify_logic`, `trigsimp`. On peut appeler directement une heuristique pour avoir le résultat plus vite (cf `http://docs.sympy.org/latest/tutorial/simplification.html`)

Développement

Par la commande `expand()`

```
>>> x,y=sy.symbols('x y')
>>> sy.expand((x+y)**3))
x**3+3*x**2y+3x*y**2+y**3
```

Ce qui permet parfois de simplifier une expression

```
>>> x,y=sy.symbols('x y')
>>> (x+y)**2-(x-y)**2
x**3+3*x**2y+3x*y**2+y**3
>>> sy.expand(_)
4*x*y
```

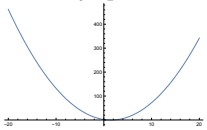
On a utilisé `_` qui rappelle le dernier résultat évalué.
Comme pour `simplify()`, `expand()` utilise des heuristiques.

Fonctions mathématiques usuelles

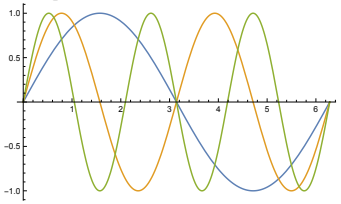
sqrt(x), *exp*(x), log(x), log(x, b), sin(x), abs(x)
Fonctions propres au calcul formel
del(x)
expand($(x-y)^2$)
simplify $((13+34*I)/(3+4*I))$
solve(*Eq*($x**2-3*x+2$,0))
factor($x**2-2*x-3$)
SymPy permet aussi de dériver (diff) d'intégrer (integrate), de calculer une limite (limit), de calculer des D.L. (series)

Faire des graphiques

```
>>> 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()
```



Listes Python

On utilisera beaucoup les listes : ensembles d'éléments groupés

```
>>> L=[1,2,"etc"]
>>> L.append(f(2+5*x))
>>> L
[1, 2, 'etc', (5*x + 2)**2]
```

On accède à l'élément i par `L[i]` (on commence à 0)

```
>>> L[0]
1
>>> M=list(range(10))
>>> L+M
[1, 2, 'etc', 12*(5*x + 2)**3, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(M)
10
```


Compréhension de listes

Les compréhensions de listes permettent de créer des listes par des *one-liners* par une syntaxe proche de la description mathématique.

```
>>> M=list(range(10))
>>> M
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On peut définir la liste des carrés :

```
>>> [i**2 for i in M]
[0,1,4,9,16,25,36,49,64,81]
```

Et filtrer les nombres impairs des carrés en ajoutant une condition

```
>>> [i**2 for i in M if i%2==1]
[1,9,25,49,81]
```

Appliquer une fonction sur une liste

Au moyen de la fonction `map` de `Python`

```
>>> Q=[i**2 for i in range(1,6)]
[1,4,9,16,25]
>>> list(map(sqrt,Q))
[1,2,3,4,5]
```

On peut appliquer une fonction `f` symbolique sur les éléments

```
>>> f=sy.Function('f')
>>> list(map(f,Q))
[f(1),f(4),f(9),f(16),f(25)]
```

Ou une fonction définie

```
>>> def f(x):
        return x+1
>>> list(map(f,Q))
[2,5,10,17,26]
```


Tests et conditions

Tester si $2+2=4$ (relations `==`, `!=`, `<`,`>`,`<=`,`>=`)

```
>>> 2+2 == 4
True
```

Conditions avec `if` sous trois formes :

- `if` seul
- `if -- else`
- `if -- elif --else`

```
>>> a,b = 200,33
    if b > a:
        print("b est supérieur à a")
    elif a == b:
        print("a et b sont égaux")
    else:
        print("a est supérieur à b")
```


Itérer en Python

Deux commandes pour itérer : `while` et `for`

while : qui exécute une suite d'instructions tant qu'une condition est vraie

```
>>> i = 1
    while i < 3:
        print(i)
        i += 1
1 2
```

for : qui parcourt tout objet énumérable

```
>>> for i in [1,2]:
        print(i)
1 2
```

Exemple pour calculer les racines *n*-ièmes de l'unité

```
>>> for i in range(1,3): # 3 est exclu; seulement 1 et 2
        print(sy.factor(x**k-1))
(x-1) (x-1)*(x+1)
```


Tutoriel video

`https://www.youtube.com/watch?v=cvHyaE_bs8s`