TP 1 : Introduction à SAGE

Le but de ce TP est de se familiariser avec le logiciel `SAGE` puis de commencer à l'utiliser pour illustrer l'algorithme d'Euclide vu en cours.
AIDE. On obtient l'aide de `SAGE` en faisant suivre la commande sur laquelle on cherche de l'aide d'un point d'interrogation. Par exemple : `expand?`.

1 Symboles, expressions et variables

`SAGE` peut en premier lieu être vu comme une calculatrice. Il utilise les symboles usuels ($+, -, *, /$, etc...) pour les opérations usuelles correspondantes. `SAGE` connait un certain nombre de fonctions prédéfinies (`exp, ln, cos...`) et de constantes (`e, pi, i, infinity,...`).

`SAGE` est un outil de *calcul formel* : ses possibilités vont donc bien au-delà de la manipulation de nombres ou de fonctions. Plusieurs fonctions `SAGE` permettent en particulier de calculer des dérivées (`diff`), des intégrales ou primitives (`integrate`), des sommes ou séries (`sum`), faisant intervenir des expressions dans lesquelles certains paramètres sont non-assignés. Pour prévenir `SAGE` que, disons la lettre t, va être utilisée comme symbole non-assigné, on tape `var('t')`. Il ne faut pas confondre ces symboles avec des variables (au sens informatique du terme) qui sont des objets auxquels on a donné un nom à l'aide de l'opérateur "=". Ce dernier opérateur ne doit pas non plus être confondu avec l'opérateur de comparaison "==" (qui renvoie "vrai" ou "faux" suivant que l'égalité testée est exacte ou non). Essayer, dans l'ordre, les commandes suivantes :

$$\texttt{a} = \texttt{10}\,, \qquad \texttt{a} + \texttt{1}\,, \qquad \texttt{a} == \texttt{3}\,.$$

D'autres manipulations courantes d'expressions possibles avec `SAGE` sont le développement, la factorisation, la substitution de variables, la simplification (`expand, factor, subs, simplify`).

À titre d'entrainement, essayer par exemple de :
— calculer la dérivée des fonctions usuelles cos, arctan, ln, $x \mapsto \exp(-x^2)$.
— calculer des primitives de quelques fonctions usuelles (en commençant par les fonctions du point précédent),
— calculer $\int_{-\infty}^{\infty} e^{-x^2}$,
— factoriser $4x + 4y + 2x^2y + 9x^2 + 2x^3 + 9xy$ en utilisant `factor`,
— afficher 200 décimales de π (voir l'aide sur `N`),
— calculer $\sum_{n \geqslant 1} \frac{1}{n^2}$.

2 Types

`SAGE` classifie ses objets suivant leur type. La fonction `type` renvoie le type d'un objet (essayer avec certains des objets rencontrés jusqu'ici).

Par exemple, si a et $n \geqslant 1$ sont des entiers, la classe de a modulo n peut être représentée par `IntegerModRing(n)(a)`. Il est possible de stocker un type dans une variable : on peut ainsi définir l'anneau des entiers modulo 7 via `Z7 = IntegerModRing(7)`. On peut alors "appliquer" `Z7` à un entier donné : la sortie obtenue est la classe de cet entier modulo 7. L'anneau des entiers relatifs est lui noté `ZZ`. Pour obtenir un représentant d'une classe $a \in \mathbf{Z}/7\mathbf{Z}$, on peut donc utiliser la commande `ZZ(a)`.

— Que se passe-t-il si l'on fait calculer l'entier $7^{5^{5^5}} (= 7^{\left(5^{\left(5^5\right)}\right)})$ dans $\mathbf{Z}$?
— Essayer de contourner la difficulté apparaissant à la question précédente pour calculer avec `SAGE` la valeur de la classe de $7^{5^{5^5}}$ modulo 13.

Une suite ordonnée d'expressions s'appelle une *liste*. Par exemple `s = [4, 3, 3, 6]`. La commande `s[i]` appelle la i-ème élément de la liste `s`, sachant que le premier est numéroté *zéro*. Ainsi, dans l'exemple, `s[1]` est 3. La longueur de `s` s'obtient par `len(s)`. Pour obtenir la liste des entiers entre 1 et n on utilise `[1..n]`. Pour construire une séquence (ou une liste) obtenue en faisant varier un paramètre, l'opérateur `for` est très utile. Essayer par exemple la commande `[2^i for i in [1..10]]`.

— Faire la liste `s` des entiers naturels multiples de 3 et inférieurs à 1000. Tester les commandes `s + s`, `4*s` et enfin `s.append(pi)` suivi de l'affichage de s.

— Créer la liste des six premières dérivées de xe^x sous forme factorisée (`diff`, `factor`).

3 Programmer une fonction

On peut créer de nouvelles fonctions (à ne pas confondre avec les expressions symboliques rencontrées jusqu'à présent) en utilisant la commande `def`. Par exemple, la définition suivante :

```
def sommeEntiers(n):
    l = [1..n]
    return sum(l)
```

définit une fonction renvoyant la somme des entiers de 1 à n. Notez les deux points qui marquent le début du corps de la fonction et notez que toutes les lignes qui sont dans le corps de la fonction sont décalées de 4 espaces pqr rqpport à `def`. On retrouve la syntaxe `Python`. Le même principe s'applique pour les boucles définies par `for` ou `while`. Essayer :

```
for i in [1..3]:
    j = 0
    while j < 4:
        print "i =", i, "j =", j
        j = j + 1
```

L'instruction `if` sert à exécuter une portion de programme seulement si une certaine condition est vérifiée. Pour exprimer cette condition on peut utiliser les opérateurs de comparaison `==`, `>`, `<`, `<=`, `>=`, `<>` signifiant respectivement $=, >, <, \leq, \geq, \neq$. On peut également combiner logiquement ces opérateurs en utilisant `or` et `and`. Essayer par exemple :

```
if (2 > 1):
    print "Tout va bien."
```

On peut aussi définir à l'aide du mot `else` une portion à exécuter seulement si la condition n'est pas vérifiée. Pour voir le rôle crucial de l'indention, on comparera le résultat de

```
if (2 > 1):
    print "Tout va bien."
else:
    print "Etonnant !"
    print "Il faut reprendre le calcul."
```

avec

```
if (2 > 1):
    print "Tout va bien."
else:
    print "Etonnant !"
print "Il faut reprendre le calcul."
```

— Implanter le calcul de la fonction factorielle :
  — en utilisant une boucle `for`.
  — en utilisant un appel récursif
— Écrire une procédure `listePremiers(N)` donnant la liste des nombres premiers plus petits qu'un entier N donné (voir `next_prime`).
— Donner la liste des nombres premiers jumeaux (i.e. des couples de nombres premiers de la forme $(p, p+2)$) inférieurs à 1000.
— Écrire une procédure récursive `Fib(n)` donnant le n-ième terme de la suite de Fibonacci dont on rappelle la définition :

$$F_0 = F_1 = 1\,, F_{n+2} = F_{n+1} + F_n\,,\ n \geqslant 0\,.$$

(i) Que se passe-t-il si l'on demande la valeur `Fib(n)` pour n supérieur à 40 ? Insérer la commande `@cached_function` dans le même cadre mais avant la première ligne `def`... de la procédure `Fib`. Essayer de calculer `Fib(100)`.

(ii) Donner une implantation itérative `Fib2` du calcul du n-ème nombre de Fibonacci (i.e. le code consistera cette fois essentiellement en une boucle `for`).

(iii) Calculer les valeurs approchées (utiliser `float`) de F_{n+1}/F_n pour $n = 10, 50, 100, ...$ (en utilisant au choix `Fib` ou `Fib2`). Que constate-t-on ?

(iv) En notant que la récurrence définissant (F_n) est linéaire donnant une troisième implantation `Fib3` de la suite (F_n) consistant en le calcul des puissances successives d'une certaine matrice à préciser.

— On considère la suite suivante (appelée *suite de Syracuse*) définie pour un paramètre entier $A \geq 1$ par

$$u_0 \in \mathbf{N}^* \text{ donné}, \qquad u_{n+1} = u_n/2 \text{ si } u_n \text{ est pair }, u_{n+1} = Au_n + 1 \text{ sinon }.$$

On note $f(u_0, A)$ le minimum (éventuellement ∞) des $n \geq 0$ tel que $u_n = 1$.

1. Calculer avec `SAGE` $f(u_0, 3)$ pour $u_0 \in \{2, \ldots, 100\}$. Que peut-on conjecturer ?
2. Le même phénomène se produit-il si lorsque $A = 5$?

4 Algorithme d'Euclide

Exercice 1 (*Euclide*)

1. Tester les commandes `//` et % liées à la division euclidienne pour un couple d'entiers a et b.
2. Implanter l'algorithme d'Euclide pour les entiers (on ne demande pas pour l'instant que l'algorithme renvoie des coefficients de Bézout mais simplement le pgcd du couple d'entiers donné en entrée) sous forme d'une fonction `euclide(a,b)`. Comparer avec `gcd`.
3. On conserve la notation $(F_n)_{n\geq 1}$ pour la suite de Fibonacci. En modifiant légèrement la fonction `euclide(a,b)` pour qu'apparaisse en sortie le nombre de divisions euclidiennes nécessaires au calcul du pgcd, vérifier que le calcul de $\text{pgcd}(F_{n+1}, F_{n+2})$ nécessite n itérations dans l'algorithme d'Euclide, pour $1 \leqslant n \leqslant 100$.

Exercice 2 (*Euclide étendu*)

1. Écrire l'algorithme d'Euclide étendu pour les entiers sous forme d'une fonction `euclideEtendu(a,b)`.
2. Vérifier que la procédure ci-dessus fonctionne toujours dans $\mathbf{Q}[X]$.
   [La commande `X=polygen(QQ,'X')` permet de travailler dans l'anneau de polynômes $\mathbf{Q}[X]$.]
3. Comparer `euclideEtendu` avec `xgcd` pour les entiers et les polynômes.
4. Sans recours à la factorisation dans $\mathbf{Q}[X]$, écrire une fonction `SansCarre(P)` donnant la partie sans facteur carré d'un polynôme de $\mathbf{Q}[X]$ (la *partie sans facteur carré d'un polynôme* P est le produit des facteurs irréductibles 2 à 2 distincts de P).
5. Soit $\zeta_{15} = \exp(2i\pi/15) \in \mathbb{C}$. Écrire l'inverse de $1 + \zeta_{15}$ comme un polynôme en ζ_{15} à coefficients rationnels.