Suites récurrentes et preuves formelles

Calcul formel – TP 3

1. Nombres bis de Fibonacci

1. Voici une fonction qui pour $n = 0, 1, 2$, retourne les valeurs P_0, P_1, P_2, et qui pour $n > 2$ calcule P_n à l'aide d'une boucle.

```
def fibonacci_bis(n):
    if n == 0: return 3
    if n == 1: return 0
    if n == 2: return 2
    P_n_3 = 3
    P_n_2 = 0
    P_n_1 = 2
    for k in range(n-2):
        P_n = P_n_2 + P_n_3
        P_n_3 = P_n_2
        P_n_2 = P_n_1
        P_n_1 = P_n
    return P_n
```

   On peut ensuite, dans une deuxième boucle, utiliser la fonction qui vient d'être définie pour afficher la valeur de n et celle de P_n, et ce pour $0 \leqslant n < 20$:

```
for n in range(20):
    print n,'␣',fibonacci_bis(n)
```

2. On résout l'équation à l'aide de `solutions = solve(x^3-x-1==0,x)`. Comme on le voit avec `print(solutions)`, cette commande renvoie une liste de trois égalités, par exemple la première commence par :

   `x == -1/2*(1/18*sqrt(23)*sqrt(3) + 1/2)^(1/3)*(I*sqrt(3) + 1) - ...`

   Pour en extraire ce qui nous intéresse, à savoir les trois membres de droite, on peut alors définir :

```
alpha = solutions[2].rhs()
beta = solutions[0].rhs()
gamma = solutions[1].rhs()
```

3. (a) Grâce aux solutions que nous venons de nommer, nous pouvons définir une fonction qui retourne Q_n par :

```
def conjec(n):
    return alpha^n+beta^n+gamma^n
```

On peut ensuite utiliser une nouvelle boucle pour comparer les premières valeurs de P_n et Q_n et voir qu'elles sont égales (en calculant pas exemple leur différence) :

```
for n in range(10):
    valeur = fibonacci_bis(n)-conjec(n)
    print expand(valeur).full_simplify()
```

(b) On vient de voir que les deux suites ont les mêmes premiers termes. Il suffit donc de montrer que Q_n vérifie la même relation de récurrence que celle définissant P_n. Ce qu'on fait par :

```
var('n')
test = conjec(n)-conjec(n-2)-conjec(n-3)
print(test.canonicalize_radical())
```

4. (a) On modifie facilement la fonction précédente :

```
def fibonacci_bis_modulo(n,m):
    if n == 0: return 3
    if n == 1: return 0
    if n == 2: return 2
    P_n_3 = 3
    P_n_2 = 0
    P_n_1 = 2
    for k in range(n-2):
        P_n = (P_n_2 + P_n_3) % m
        P_n_3 = P_n_2
        P_n_2 = P_n_1
        P_n_1 = P_n
    return P_n
```

(b) **Énigme.** On peut se servir d'une boucle du type

```
for n in range(270000,272000):
    if not(is_prime(n)) and fibonacci_bis_modulo(n,n)==0:
      print n
```

Les calculs sont assez longs ! Mais tous ces calculs sont-ils nécessaires ?

Réponse attendue : 33150.

2. Une suite récurrente

1. On procède exactement comme à la première question de l'exercice précédent.
2. (a) On trouve immédiatement que $v_n = 3v_{n-1} + 4v_{n-2}$.

   (b) L'équation associée à cette récurrence double est donc $x^2 = 3x + 4$, qu'on résout formellement sans difficulté. On trouve 4 et -1 comme solutions, ce qui montre que la suite s'écrit sous la forme $v_n = a4^n + b(-1)^n$. Avec les conditions initiales, on arrive à $v_n = \frac{2\cdot 4^n + 3\cdot(-1)^n}{5}$ et $u_n = \frac{5}{2\cdot 4^n + 3\cdot(-1)^n}$.
3. Même méthode qu'à la troisième question de l'exercice précédent.

4. **Énigme.** On pose `epsilon = 10^(-20)`.

   Sage n'est pas très bon pour résoudre formellement des équations avec des inconnues en exposants. Ainsi, utiliser directement `soln = solve( 5/(2*4^n+3*(-1)^n) <= epsilon, n)` n'est pas convaincant.

   On distingue les cas pair et impair :

   ```
   var('k')
   solk_pair = find_root( 5/(2*4^(2*k)+3) == epsilon, 0,100)
   print 'sol␣pair', solk_pair
   ```

   On trouve pour le cas pair $16,94\ldots$, et pour le cas impair $16,44\ldots$ On peut faire la synthèse des deux cas avec `enigme = min(2*ceil(solk_pair),2*ceil(solk_impair)+1)`.

   Réponse attendue : 34.

3. La méthode de Halley

1. La méthode est la même que dans les exercices précédents. On définit une fonction contenant une boucle permettant de calculer les termes de la suite récurrente de proche en proche :

   ```
   def newton(g,x0,n):
       gg = diff(g,x)
       phi = x - g/gg
       u = x0
       for i in range(n):
           u = phi(x=u)
       return u
   ```

   La commande `print(newton(x^3-5,2,2))` retourne bien la valeur $\frac{503}{294}$.

2. Plutôt que $\frac{g(x)}{g'(x)}$, il est avantageux de calculer l'inverse qui est la dérivée logarithmique de $g(x)$:

$$\frac{g'(x)}{g(x)} = \frac{f'(x)}{f(x)} - \frac{1}{2}\cdot\frac{f''(x)}{f'(x)} = \frac{2\left(f'(x)\right)^2 - f(x)f''(x)}{2f(x)f'(x)}$$

   ce qui est bien ce qu'il fallait vérifier.

3. **Énigme.** On peut suivre exactement la démarche de la première question. On trouve $v_2 = \frac{61896}{36197}$.

   Réponse attendue : 61896.

4. Factorielle

1. Les multiples de p divisant n sont $\{p, 2p, 3p\ldots\}$. Il y en a $E\left(\frac{n}{p}\right)$. Les multiples de p^2 sont bien dans la liste précédente mais doivent compter deux fois, il y en a $E\left(\frac{n}{p^2}\right)$, on rajoute ce nombre à notre total. Les multiples de p^3 doivent compter trois fois, mais n'ont été comptés que deux fois dans nos listes précédentes, on rajoute donc $E\left(\frac{n}{p^3}\right)$... D'où la formule.

   La définition d'une fonction calculant cette somme se fait suivant le modèle habituel :

   ```
   def legendre1(n,p):
       puiss = p
       somme = 0
       while n/puiss >= 1:
   ```

```
        somme  = somme + floor( n/puiss )
        puiss = puiss * p
    return somme
```

2. On retourne la liste des chiffres de la décomposition de n en base p grâce à la fonction :

```
def decomposition(n,p):
    dec = []
    m = n
    while m>0:
        dec = [m % p] + dec
        m = m // p
    return dec
```

3. Compte tenu de la question précédente, il suffit de savoir calculer la somme des éléments d'une liste, ce qui se fait par exemple de la manière suivante :

```
def legendre2(n,p):
    dec = decomposition(n,p)
    somme = sum(dec)
    val = (n-somme)/(p-1)
    return val
```

4. **Énigme.** On calcule grâce aux fonctions ci-dessus $v_2(123456789) = 123456773$ et $v_5(123456789) = 30864192$.

   Réponse attendue : 30864192**.**