INFO-H-100 - Programmation
TP 7 - Calcul itératif et numérique

Lorsque l'exercice demande d'écrire une fonction, écrivez la fonction demandée et testez-la avec plusieurs valeurs pertinentes.

Cette séance portant sur les boucles `while`, veillez à ne pas utiliser de boucle `for`.

Ex. 1. Nous supposons que `range` n'existe pas. Ecrire une fonction `my_range(a,b)` qui produit une liste ordonnée contenant tous les entiers dans $[a, b[$.

Ex. 2. Ecrire une fonction `my_range_step(a,b,step)` qui produit une liste ordonnée contenant tous les entiers dans $[a, b[$ séparés de `step`. On considère que $a < b$ et que `step` est strictement positif.

Ex. 3. Ecrire une fonction `linspace` qui renvoie une liste des n réels équidistants entre $begin$ et end **inclus**.

```
>>> linspace(2,6,21) #begin = 2, end = 6, n = 21
[2.0, 2.2, 2.4, 2.6, ..., 5.4, 5.6, 5.8, 6.0]
```

Ex. 4. Écrire une fonction qui calcule la valeur approchée de π sur base de la série suivante (due à Euler). Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à 10^{-6} ou lorsque le nombre de termes considérés atteint 10000.

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + ...$$

Ex. 5. Écrire une fonction qui calcule la valeur approchée de π sur base de la série suivante. Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ϵ donné (c'est à dire en variable globale) ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi}{8} = \frac{1}{1 \times 3} + \frac{1}{5 \times 7} + \frac{1}{9 \times 11} + ...$$

Ex. 6. Écrire une fonction qui calcule la valeur approchée de π sur base de la série suivante (due à Leibniz). Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ϵ donné (c'est à dire en variable globale) ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...$$

Ex. 7. Écrire une fonction qui calcule la valeur approchée de π sur base de la série suivante. Les calculs s'arrêtent lorsque la valeur du dernier terme ajouté est inférieure à un ε donné, ou lorsque le nombre de termes considérés atteint une borne également donnée.

$$\frac{\pi}{4} = 4 \cdot arctan\left(\frac{1}{5}\right) - arctan\left(\frac{1}{239}\right)$$

avec la série :

$$arctan(x) = \frac{x}{1+x^2}\left(1 + \frac{2}{3} \cdot \frac{x^2}{1+x^2} + \frac{2}{3} \cdot \frac{4}{5} \cdot (\frac{x^2}{1+x^2})^2 + \cdots\right) = \frac{x}{1+x^2} \sum_{i=0}^{\infty} c_i (\frac{x^2}{1+x^2})^i$$

où $c_0 = 1$
et pour $i > 0, c_i = \frac{2 \cdot i}{2 \cdot i + 1} c_{i-1}$

Ex. 8. Deux suites (x_i) et (y_i) sont définies par

$$\begin{aligned} y_0 &= 2 \\ x_0 &= 1 \\ x_{i+1} &= \frac{2}{y_{i+1}} \\ y_{i+1} &= \frac{y_i + x_i}{2} \end{aligned}$$

Nous admettons que les suites convergent toutes les deux vers $\sqrt{2}$, avec de plus $x_i < \sqrt{2} < y_i$. Écrire une fonction `rac2` qui calcule deux approximations supérieure et inférieure de $\sqrt{2}$, de différence inférieure à ε (prévoyez aussi d'arrêter les calculs si le nombre d'étapes dépasse une borne fixée).

Ex. 9. Soit r un nombre réel positif. La suite de nombres réels x_n est spécifiée par :

$x_1 = 1$,
$x_{n+1} = \frac{1}{2}\left(x_n + \frac{r}{x_n}\right)$,

Cette suite, obtenue par une méthode générale, due à Newton, converge très rapidement vers $\sqrt{r}$.

Écrire une fonction qui calcule la valeur approchée de $\sqrt{r}$. L'exécution s'arrêtera lorsque la différence entre deux approximations successives sera inférieure à un ε donné.

Ex. 10. Soit r un nombre réel positif. Deux suites de nombres réels x_n et y_n sont spécifiées par :

$x_1 = 1$,
$y_1 = 1$,

et pour $n \geq 1$:

$$\begin{cases} x_{n+1} = x_n + r.y_n \\ y_{n+1} = x_n + y_n \end{cases}$$

Il n'est pas difficile de prouver $y_n \neq 0$ pour $n \geq 1$, et aussi que le quotient $\frac{x_n}{y_n}$ converge vers $\sqrt{r}$.

Écrire une fonction qui calcule la valeur approchée $\frac{x_n}{y_n}$ de $\sqrt{r}$, où n est la plus petite valeur pour laquelle la condition suivante est vraie :

$(x_n)^2 < (r+\varepsilon)(y_n)^2$ **et** $(x_n)^2 > (r-\varepsilon)(y_n)^2$.

(ε étant comme d'habitude une valeur donnée). L'exécution sera interrompue si le nombre d'étapes devient trop grand.

Ex. 11. La méthode d'Euler est une procédure numérique pour résoudre par approximation des équations différentielles du premier ordre avec condition initiale. [1]

Soit une équation différentielle ordinaire de la forme

$$u'(t) = f(u(t)), \;\; u(0) = u_0$$

où u est l'inconnue, f est une fonction donnée et u_0 la condition initiale également donnée. Supposons que nous voulons calculer une solution approchée de l'équation différentielle pour les $n >= 1$

1. Une équation différentielle est une équation formulée en terme de relations entre des fonctions, typiquement une fonction et ses dérivées. Le but est de déterminer la fonction. Par exemple, pour l'équation différentielle $u''(t) = -u(t)$ où l'inconnue est la fonction u, les solutions possibles sont sin et cos (avec des coefficients dépendant des conditions initiales). En calcul numérique, la solution d'une équation différentielle ne sera pas une expression analytique de la fonction recherchée, mais sa valeur en un certain nombre de points.

instants $t_1 \dots t_n$ de 0 à T où $T > 0$ est donné. Soit $\Delta t = T/n$ et soit u_k l'approximation de la solution exacte $u(t_k)$, la méthode d'Euler propose de calculer chaque u_k en fonction de u_{k-1} à l'aide de la formule $u_k = u_{k-1} + f(u_{k-1})\Delta t$.

Écrire une fonction `euler` qui prend comme paramètres la fonction f, les entiers T et n et le réel u_0 et qui renvoie une paire formée de la liste des instants t_k et de la liste des valeurs u_k aux instants correspondants. Utilisez la fonction `linspace` de l'exercice 3.

```
>>> def f(u): return (2*u) - 1

>>> euler(f, 6, 12, 2.0)
([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0],
 [2.0, 3.5, 6.5, 12.5, 24.5, 48.5, 96.5, 192.5, 384.5, 768.5, 1536.5, 3072.5, 6144.5])
```

Par exemple, le réel `u[2]` est égal à `u[1]+(2*u[1]-1)*t/n` et donc à $3.5+(7-1)*0.5$ ce qui donne 6.5.

INFO-H-100 - Programmation
TP 7 - Calcul itératif et numérique
Corrections

Solution de l'exercice 1:

```
def my_range(a,b):
    ls = []
    while a < b:
        ls.append(a)
        a += 1
    return ls
```

Solution de l'exercice 2:

```
def my_range_step(a,b,step):
    ls = []
    while a < b:
        ls.append(a)
        a += step
    return ls
```

Solution de l'exercice 3:

```
def linspace(begin, end, n):
    step = (end - begin) / (n - 1)
    ls = []
    x = 0
    while x < n:
        ls.append(begin + x * step)
        x += 1
    return ls
```

Notes :

- Vous serez probablement surpris, en affichant le résultat de cette fonction, que certains éléments de la liste soient affichés avec un grand nombre de décimales et présenteront une erreur d'arrondi.

  ```
  >>> print(linspace(2,6,21))
  [2.0, 2.2, 2.4, 2.6, 2.8, 3.0, 3.2, 3.4000000000000004, 3.6, 3.8, 4.0, 4.2, ...
     ... 4.4, 4.6, 4.800000000000001, 5.0, 5.2, 5.4, 5.6, 5.800000000000001, 6.0]
  ```

  Cette erreur n'est pas liée à votre code ni à Python, mais à la représentation des nombres en virgule flottante au niveau hardware (pour plus d'informations, voir `http://docs.python.org/3.2/tutorial/floatingpoint.html`). Pour cet exercice, nous nous contenterons de ce résultat.
- Une autre manière, théoriquement équivalente, de résoudre cet exercice est la suivante :

  ```
  def linspace(begin, end, n):
      step = (end - begin) / (n - 1)
      ls = []
      val = begin
      while val<= end:
          ls.append(val)
          val += step
      return ls

  print(linspace(2,6,21))
  ```

  Cette fois-ci, on ne teste plus un indice (un entier), mais un nombre rationnel (le `step`) qui est représenté en virgule flottante en mémoire. En raison de cette représentation et de l'approximation qu'elle induit, la valeur de `val` pour le *dernier* élément ne sera généralement pas égal à `end`, mais légèrement supérieur. Il en résultera que la valeur `end` ne sera pas reprise dans la liste.

Solution de l'exercice 4:

```
EPS    = 1E-6
NB_MAX = 10000
```

```
import math

def pi_euler():
    nb_terms = 1
    term = 1
    res_sum = 1
    while term >= EPS and nb_terms < NB_MAX:
        nb_terms += 1
        term = 1.0 / nb_terms**2
        res_sum += term
    return math.sqrt(6*res_sum)
```

Solution de l'exercice 5:

```
def pi11():
    i = 1
    term = 1.0/3
    res_sum = term
    while term >= EPS and i < NB_MAX:
        i += 1
        term = 1.0/(4*i-3)/(4*i-1)
        res_sum += term
    return 8*res_sum
```

Solution de l'exercice 6:

```
def pi_leibnitz():
    nb_terms = 1
    sign = 1
    term = 1
    res_sum = 1
    while term >= EPS and nb_terms < NB_MAX:
        nb_terms += 1
        sign = -sign
        term = 1.0 / (2*nb_terms - 1)
        res_sum += sign*term
    return 4*res_sum
```

Solution de l'exercice 7:

```
def arctan(x):
    factor0 = x**2 / (1 + x**2)
    i = 1
    factor1 = 1
    factor2 = 1
    term = 1
    res_sum = 1
    while term >= EPS and i < NB_MAX:
        factor1 *= 2.0*i/(2*i + 1)
        i += 1
        factor2 *= factor0
        term = factor1*factor2
        res_sum += term
    return factor0/x*res_sum

def pi_arctan():
    return 4*(4*arctan(1.0/5) - arctan(1.0/239))
```

Solution de l'exercice 8:

```
def sqrt2():
    (x, y) = (1, 2)
    i = 0
    while y - x >= EPS and i < NB_MAX:
        y = (x + y)/2.0
        x = 2.0/y
        i += 1
    return (x, y)
```

Solution de l'exercice 9:

```
def sqrt_newton(r):
    old = 0
    x = 1.0
    while abs(x - old) >= EPS:
        old = x
        x = (x + r/x)/2.0
    return x
```

Solution de l'exercice 10:

```
def is_close_enough(x, y, r):
    x2 = x**2
    y2 = y**2
    return x2 < ((r + EPS)*y2) and x2 > ((r - EPS)*y2)

def sqrt_approx(r):
    i = 1
    (x, y) = (1, 1)
    while not is_close_enough(x, y, r) and i < NB_MAX:
        (x, y) = (x + r*y, x + y)
        i += 1
    return float(x)/y
```

Solution de l'exercice 11:

```
def linspace(begin, end, n):
    step = (end - begin) / (n - 1)
    ls = []
    x = 0
    while x < n:
        ls.append(begin + x * step)
        x += 1
    return ls

def f(u):
    return (2*u) - 1

def ode_euler(f, T, n, u0):
    dt = T/float(n)
    t = linspace(0,T,n+1)
    u = []
    u.append(u0)
    for k in range(n):
        u.append(u[k] + dt*f(u[k]))
    return t, u
```