TP Info 8
corrigé

I.1

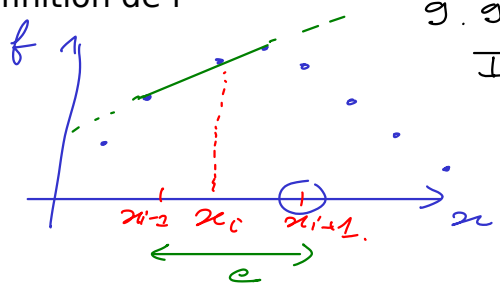
```
def fonction_1(f,x_0,e):
    """
    f: fonction
    x_0: flottant
    e: flottant, plus e est petit plus le resultat est precis
    """
    df=(f(x_0+e)-f(x_0))/e
    return df
```

calcule le taux d'accroissement de f au voisinage de x_0.
= valeur approchée de la dérivée $f'(x_0)$

```
def fonction_2(f,x):
    """
    f: fonction
    x: liste de flottants=intervalle de definition de f
    """
    derivee=[]
    for i in range(1,len(x)-1):
        x0=x[i]
        e=x[i+1]-x[i-1]
        df=fonction_1(f,x0,e)
        derivee.append(df)

    return derivee
```

ensemble discret (x = linspace (0, 10, 100))
[0, 0.1, 0.2 ----- 9.9, 10] 100 pts
Ici e = 0,2

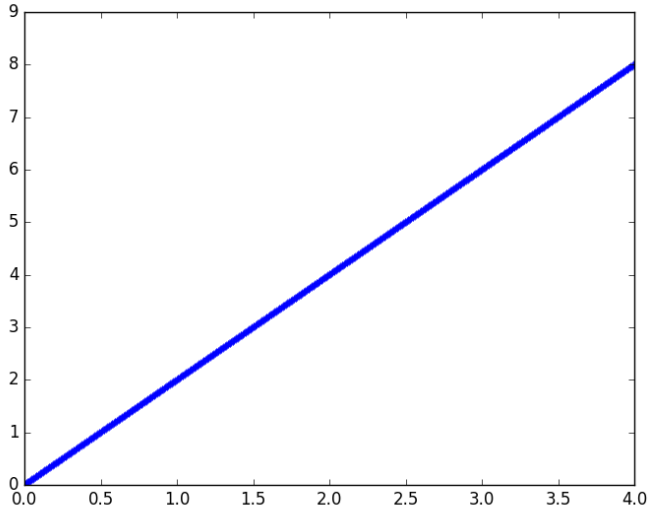


$f'(x_0)$

évalue, de façon approchée les dérivée de f sur l'intervalle x.

```
#Appel fonction
x=np.linspace(0,4,1000)#[0,4]
def carre(x):
    return x**2

df=fonction_2(carre,x);print(df)
x=x[1:len(x)-1] # dérivée non calculée aux bords (en 0 et en 4) : on exclut ces 2 valeurs.
plt.plot(x,df,'+')
```



On reconnaît la courbe représentative de la fonction $f : x \to 2x$ dérivée de $f : x \to x^2$

I2 – Produit scalaire

$$\vec{u}\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} \cdot \vec{v}\begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} = \sum_{i=0}^{n} u_i v_i$$

↑ boucle for

↑ structure de donnée pour représenter un vecteur : un tableau de flottants.

```
def scalaire(u,v):
    """
    calcule le produit scalaire de u et v
    u,v: list, de meme dimension
    scal : float, scal=u.v
    """
    scal=0
    for i in range(len(u)):
        scal=scal+u[i]*v[i]
    return scal
```

fonctionne quelque soit la dimension de $\vec{u}$ et $\vec{v}$

I3 – Suite de Collatz (ou de Syracuse)

```
def f_collatz(u):
    """
    Calcule un+1 à partir de un suivant la relation de recurrence suivante:
    si un pair, un+1=un/2 sinon un+1=3un+1
    u: int
    v:int

    """
    if u%2==0:
        v=u/2
    else:
        v=3*u+1
    return v
```

direct.

```
def temps_vol(u0):
    """
    Calcule le temps de vol de la suite de Collatz i.e l'indice n du premier
    terme égal à 1
    u0 : germe de la suite de Collatz, int
    n : int
    """
    n=0
    u=u0
    while u!=1: # définition du temps de vol
        u=f_collatz(u) # un ← un+1
        n=n+1
    return n
```

```
def liste_suite(u0,f,N):
    """
    Renvoie la liste des N premiers de la suite définie par la
    relation de reccurence f et le terme initial u0.
    u0: terme intial, float ou int
    f: fonction
    N: int

    """
    l_u=[u0]
    u=u0
    for i in range(1,N):
        u=f(u)#un<-un+1
        l_u.append(u)
    return l_u
```

Très semblable à la fonction précédente à 2 différences près.

(1) renvoie N termes et non un terme donc un tableau

(2) fonctionne pour n'importe quelle suite :
- le terme initial est passé en argument via u_0
- la relation de récurrence est passée en argument via la fonction f.

```
def evolution_temps_vol(u0):
    """
    Calcule et renvoie la liste des temps de vol de la suite
    de Collatz pour chaque valeur du terme initial contenue dans
    la liste u0
    u0: list
    temps_vol: list
    """
    n=0#temps de vol
    l_t=[]
    for i in range(len(u0)):
        n=temps_vol(u0[i])
        l_t.append(n)
    return l_t
```

(1) Ensemble des valeurs de u_0 à partir desquels on calcule le temps vol.

Etude la suite

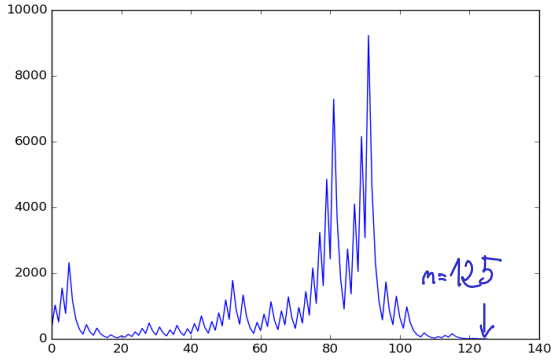
```
v=temps_vol(343);print(v)
```

Calcule du temps de vol pour $u_0 = 343$
Renvoie 125

```
collatz=liste_suite(343,f_collatz,temps_vol(343)+1)
plot(collatz)# graphe des termes de la suite.
```

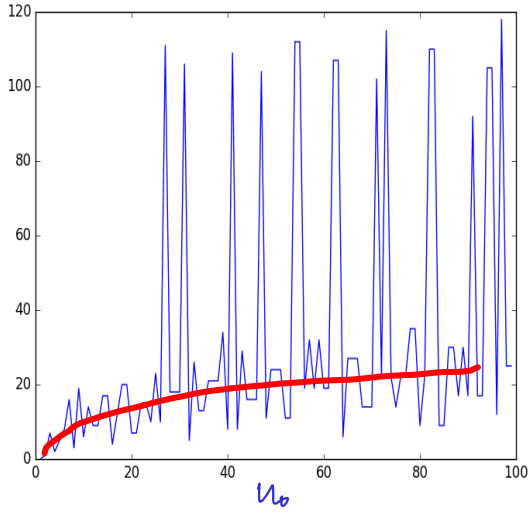
Calcule des termes de la suite sur le temps de vol.

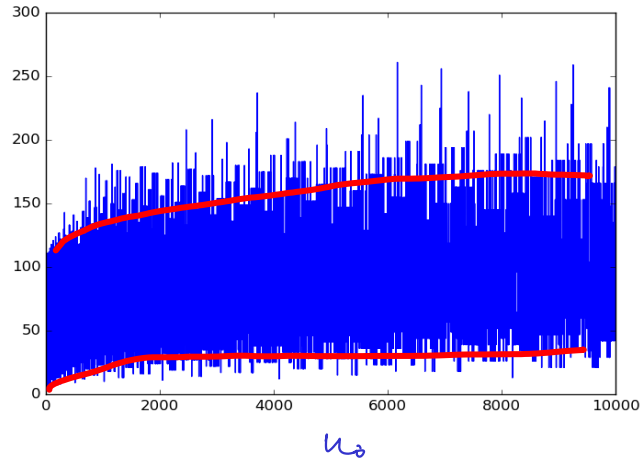
de 0 à v : termes.



La suite est très fluctuante.

Influence de u_0 sur le temps de vol v





Difficile de conjecturer un lien entre temps de vol. avec u_0 : $\langle v \rangle \sim \log(u_0)$?
↑ moyenne

I4- Autour des polynômes

1/ Evaluation naïve de $P(x)$: $P(x) = \sum_{i=0}^{n} a_i x^i$

↑ boucle for

```
def polynome(P,x):
    P_x=P[0]
    for i in range(11,len(P)):
        P_x=P_x+P[i]*x**i
    return P_x
```

2/ Evaluation de Horner : $P(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + a_n x))\cdots)$

↑ à calculer en premier car il faut suivre les règle de parenthésage

il faut parcourir la liste des coeff polynomiaux à l'envers ! ⟸

Methode 1 :

for i in range(n)

i indice décroissant

$n-i-1$ $= n-1$ si $i=0$
$= n-2$ si $i=1$
⋮
$= 1$ si $i=n-2$
$= 0$ si $i=n-1$

Methode 2 :

for i in range(n-1, 0, -1)

↑ 1ère valeur ; ↑ 1 = dernière valeur ; ↑ pas

génère la liste $[n-1, n-2, n-3 \cdots 0]$

. i parcourt les indices dans l'ordre décroissant.

```
def Horner(P,x):
    n=len(P)
    P_x=0
    for i in range(0,n):
        P_x=P[n-1-i]+P_x*x
    return P_x
```

```
def Horner(P,x):
    n=len(P)
    P_x=0
    for i in range(n-1,-1,-1):
        P_x=P[i]+P_x*x
    return P_x
```

Rem : on peut aussi écrire une fonction inverse_liste(l), l'appeler dans Horner(P,x) et parcourir la liste dans l'ordre naturel.

Appel de la fonction

Q = [1, 0, -1] # polynôme $Q : x \longrightarrow x^2 - 1$

res = Horner(Q, 1)

print(res) ⟶ affiche 0

3/ Multiplication polynômiale

Maths : $P(x) = \sum_{i=0}^{p} a_i x^i$, $Q(x) = \sum_{j=0}^{q} b_j x^j$

$R(x) = P(x)Q(x) = \sum_{i=0}^{p} a_i x^i \sum_{j=0}^{p} b_j x^j$ ⟶ degré de R : $r = q + p$.

Pour $p = 2$ et $q = 2$.

$R(x) = (a_0 + a_1 x + a_2 x^2)(b_0 + b_1 x + b_2 x^2)$

$= a_0 b_0 + a_0 b_1 x + a_0 b_2 x^2$
$+ a_1 b_0 x + a_1 b_1 x^2 + a_1 b_2 x^3$
$+ a_2 b_0 x^2 + a_2 b_1 x^3 + a_2 b_2 x^4$

$= a_0 b_0 + (a_0 b_1 + a_1 b_0) x + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2$
$+ (a_1 b_2 + a_2 b_1) x^3 + a_2 b_2 x^4$

Le coeff du terme de degré k est tel que $\sum a_i b_j$ avec $i + j = k$

En généralisant :

$$R(x) = \sum_{k=0}^{r} c_k x^k \quad \text{avec} \quad \boxed{c_k = \sum_{\substack{i,j \\ i+j=k}} a_i b_j}$$

Implémentation :

1. Entrée : tableaux P, Q avec $p = \text{len}(P) - 1$ et $q = \text{len}(Q) - 1$
2. Créer un tableau R nul. taille $r = q + p$
3. Ajouter au coeff R[k] tous les produits P[i]Q[j] tq $i + j = k$

```
for i in range(q+1):
    for j in range(p+1):
        k = i+j
        R[k] = R[k] + P[i]*Q[j]
```

```
def poly_nul(n):
    N=[]
    for i in range(n+1):
        N.append(0)
    return N

def poly_mult(P,Q):
    """
    Calcule le produit P*Q
    P,Q : liste des coefficients des deux polynomes ou P[i] est le coefficient du terme de degre i
    R : liste des coefficients du polynome R=P*Q
    """
    p=len(P)-1#degre de P
    q=len(Q)-1
    r=(p+q)#degre max de R
    R=poly_nul(r)
    for i in range(p+1):
        for j in range(q+1):
            R[i+j]=R[i+j]+P[i]*Q[j]
    return R

P=[-1,1]#P=x-1
Q=[1,1]#Q=x+1
R=poly_mult(P,Q)#On espere R=x^2-1 !
print(R)

P=[-1,0,1]
Q=[1,0,2,-1]
R=poly_mult(P,Q)#On espere R=x^2-1 !
print(R)
```

Appel de poly_mult

→ affiche [-1,0,1] ! ok

Appel de poly_mult

→ affiche [-1,0,-1,1,2,-1] : à vérifier à la main