I – 2)

Pour `X`, on peut calculer N_h, créer un tableau `numpy` plein de zéros et le remplir à l'aide d'une boucle. La commande `np.arange(t0,t0+T+h,h)` fait cela directement... Penser à ajouter h à la borne supérieure, car elle est exclue comme toujours avec Python ! Noter qu'il vaut mieux ici utiliser `arange` que `linspace`, pour éviter les surprises sur la valeur du pas de la subdivision...

Pour `Y`, je crée d'abord le tableau `t` contenant les t_k (à l'aide de la fonction `X` précédente), puis je remplis de proche en proche le tableau `y`, initialisé à 0 et contenant les y_k

I – 6)

La fonction `Y` ci-dessus a été programmée en utilisant la fonction Φ générique. Il suffit maintenant d'écrire les différentes fonctions Φ correspondant aux trois méthodes proposées, lesdites fonctions appelant bien sûr la fonction F définissant l'"équation différentielle $y' = F(t, y)$. Selon ce principe de programmation modulaire, il suffira de modifier la fonction F (et les valeurs de t_0, T, h et y_0) pour changer d'exemple. Voici le code ayant produit la figure de l'énoncé :

```
import numpy as np
import matplotlib.pyplot as plt

def X(t0,T,h):
    return np.arange(t0,t0+T+h,h)

def Y(t0,T,h,y0,Phi):
    t=X(t0,T,h)
    y=np.zeros(len(t))
    y[0]=y0
    for k in range(len(t)-1):
        y[k+1]=y[k]+h*Phi(t[k],y[k],h)
    return y

def Euler(t,y,h):
    return F(t,y)

def Heun(t,y,h):
    return (F(t,y)+F(t+h,y+h*F(t,y)))/2

def RK4(t,y,h):
    a=F(t,y)
    b=F(t+h/2,y+h*a/2)
    c=F(t+h/2,y+h*b/2)
    d=F(t+h,y+h*c)
    return (a+2*b+2*c+d)/6

def F(t,y):
    return y

t0, T, h = 0, 5, 0.5
y0=1

plt.grid()
plt.title('Comparaison des 3 méthodes avec $h='+str(h)+'$')
plt.xlabel('Valeurs de $t$')
plt.ylabel('Valeurs de $y$')
t=X(t0,T,h)
plt.plot(t,Y(t0,T,h,y0,Euler),'b:',linewidth=2,label='Euler')
plt.plot(t,Y(t0,T,h,y0,Heun),'y--',linewidth=2,label='Heun')
plt.plot(t,Y(t0,T,h,y0,RK4),'r-',linewidth=2,label='RK4')
plt.plot(t,np.exp(t),'k-.',linewidth=3,label='Sol. exacte')
plt.legend(loc=0)
plt.savefig('Un_pas.eps')
plt.show()
```

II

Les fonctions Φ correspondant aux différentes méthodes sont inchangées, mais il faut adapter la fonction F, qui devient une fonction à valeurs vectorielles, la fonction générique décrite dans l'énoncé.

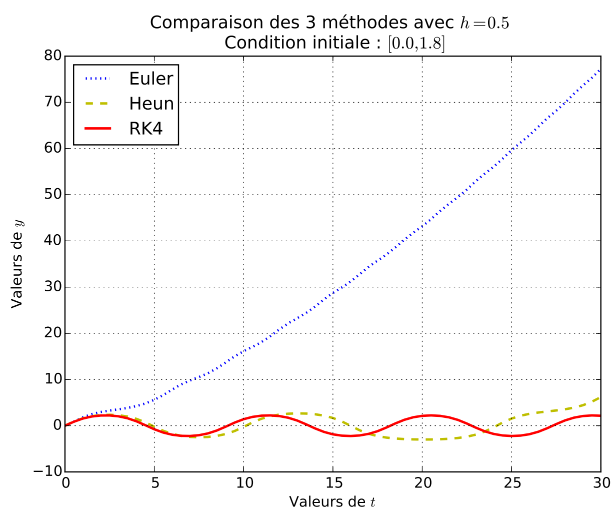
C'est la fonction scalaire G qui est à définir selon l'équation différentielle étudiée, ainsi que le *vecteur* y_0 correspondant à la condition initiale $\begin{pmatrix} u(t_0) \\ u'(t_0) \end{pmatrix}$.

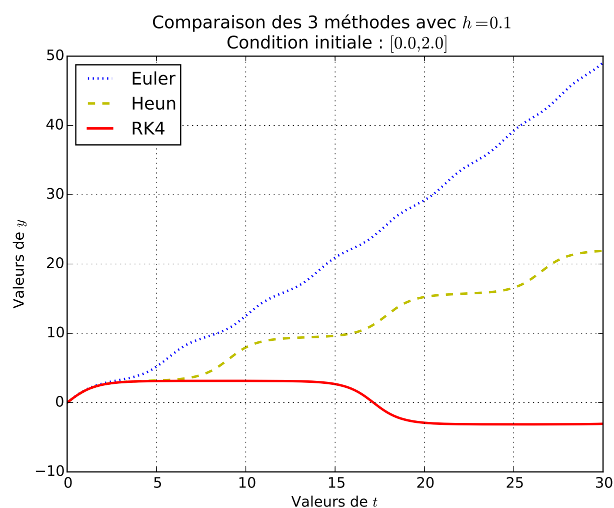
Enfin, il faut aussi adapter la fonction Y, puisque les y_k sont ici des vecteurs, d'où la création d'un tableau de type $(2, n+1)$, dont la k-ième **colonne** contiendra y_k. Suivant la syntaxe Python, cette colonne est obtenue par `y[:,k]`.
Il faut aussi penser à faire renvoyer par la fonction `Y` la **première ligne** `y[0]` (ou `y[0,:]`).
Pour tracer des portraits de phase on peut aussi utiliser les vitesses contenues dans la **seconde ligne** `y[1]`.

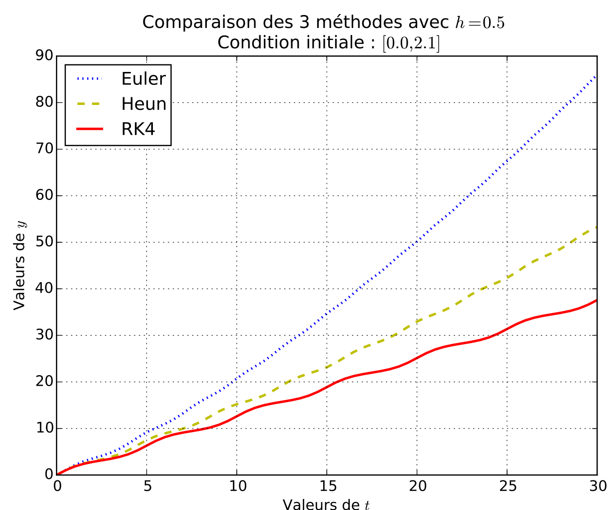
Noter la souplesse de `numpy` qui permet de stocker dans la colonne `y[:,0]` les deux valeurs initialement contenues dans une liste, voire un tableau `numpy` de type $(1, 2)$!

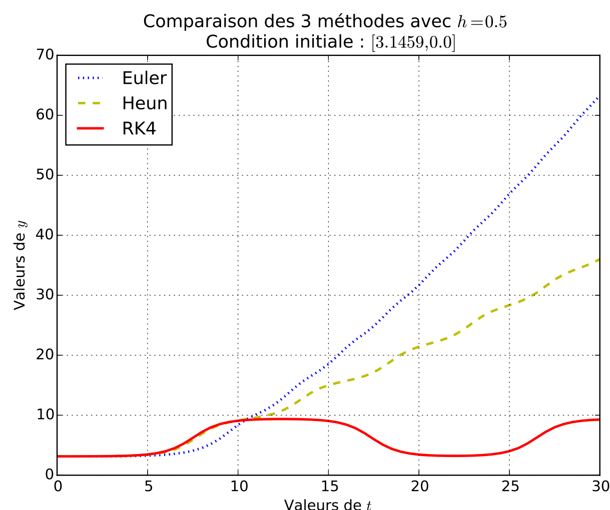
Voir page suivante le programme pour l'équation du pendule (avec $g = 1$!).

Et ci-dessous les 4 exemples suggérés (la vitesse initiale 2 correspondant au cas où le pendule monte à la verticale sans jamais l'atteindre... en théorie !).









```
from math import sin,pi
import numpy as np
import matplotlib.pyplot as plt

def X(t0,T,h):
    return np.arange(t0,t0+T+h,h)

def Y(t0,T,h,y0,Phi):
    t=X(t0,T,h)
    y=np.zeros((2,len(t)))
    y[:,0]=y0
    for k in range(len(t)-1):
        y[:,k+1]=y[:,k]+h*Phi(t[k],y[:,k],h)
    return y[0]

def F(t,y):
    return np.array([y[1],G(t,y[0],y[1])])

def Euler(t,y,h):
    return F(t,y)

def Heun(t,y,h):
    return (F(t,y)+F(t+h,y+h*F(t,y)))/2

def RK4(t,y,h):
    a=F(t,y)
    b=F(t+h/2,y+h*a/2)
    c=F(t+h/2,y+h*b/2)
    d=F(t+h,y+h*c)
    return (a+2*b+2*c+d)/6

def G(t,u,v):
    return -sin(u)

t0, T, h = 0, 30, 0.1
u0, v0 = 0.0, 2.01
y0=np.array([u0,v0])

plt.grid()
plt.title('Comparaison des 3 méthodes avec $h='+str(h)+
'$ \n Condition initiale : $['+str(u0)+','+str(v0)+']$')
plt.xlabel('Valeurs de $t$')
plt.ylabel('Valeurs de $y$')
t=X(t0,T,h)
plt.plot(t,Y(t0,T,h,y0,Euler),'b:',linewidth=2,label='Euler')
plt.plot(t,Y(t0,T,h,y0,Heun),'y--',linewidth=2,label='Heun')
plt.plot(t,Y(t0,T,h,y0,RK4),'r-',linewidth=2,label='RK4')
plt.legend(loc=0)
plt.savefig('Pendule1.eps')
plt.show()
```

III – 1)a)

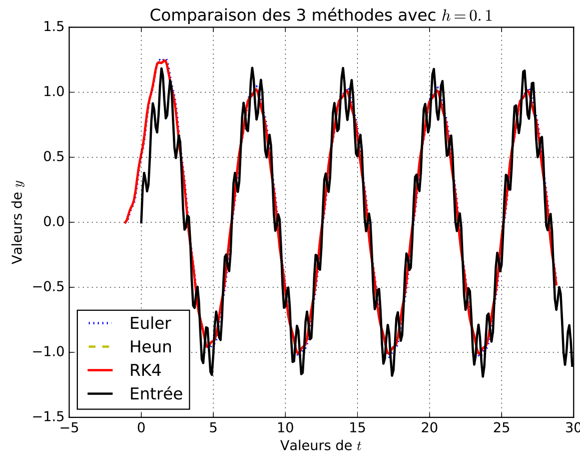
Il suffit de reprendre le code précédent, en changeant `F` (après avoir défini le signal d'entrée `e` !) et en ajoutant le calcul de δ et A qui servent à traiter la sortie :

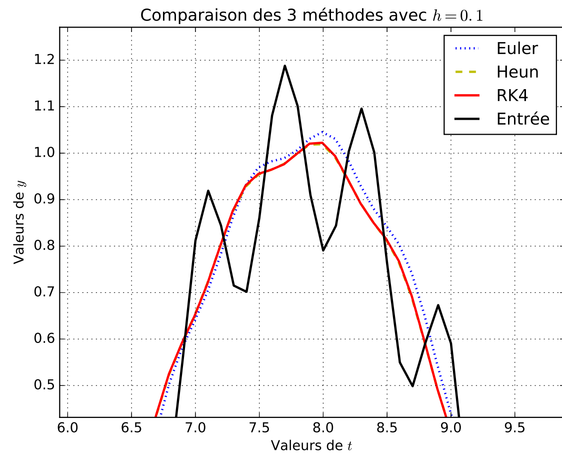
```
from math import exp,sin,atan,sqrt
import numpy as np
import matplotlib.pyplot as plt

def X(t0,T,h):
    return np.arange(t0,t0+T,h)

def Y(t0,T,h,y0,Phi):
    t=X(t0,T,h)
    y=np.zeros(len(t))
    y[0]=y0
    for k in range(len(t)-1):
        y[k+1]=y[k]+h*Phi(t[k],y[k],h)
    return y

def Euler(t,y,h):
    return F(t,y)

def Heun(t,y,h):
    return (F(t,y)+F(t+h,y+h*F(t,y)))/2

def RK4(t,y,h):
    a=F(t,y)
    b=F(t+h/2,y+h*a/2)
    c=F(t+h/2,y+h*b/2)
    d=F(t+h,y+h*c)
    return (a+2*b+2*c+d)/6

def e(t):
    return np.sin(omega*t)+a*np.sin(r*omega*t)

def F(t,y):
    return b*(e(t)-y)

t0, T, h = 0, 30, 0.1
y0=0
b, omega, a, r = 0.5, 1, 0.2, 10
phi=-atan(omega/b)
delta=phi/omega
A=sqrt(1+(omega/b)**2)

plt.grid()
plt.title('Comparaison des 3 méthodes avec $h='+str(h)+'$')
plt.xlabel('Valeurs de $t$')
plt.ylabel('Valeurs de $y$')
t=X(t0,T,h)
plt.plot(t+delta,A*Y(t0,T,h,y0,Euler),'b:',linewidth=2,label='Euler')
plt.plot(t+delta,A*Y(t0,T,h,y0,Heun),'y--',linewidth=2,label='Heun')
plt.plot(t+delta,A*Y(t0,T,h,y0,RK4),'r-',linewidth=2,label='RK4')
plt.plot(t,e(t),'k-',linewidth=2,label='Entrée')
plt.legend(loc=0)
plt.show()
```

Noter dans la définition de la fonction `e` les versions "vectorisées" des fonctions mathématiques proposées par `numpy`. Cela permet d'afficher facilement le graphe du signal d'entrée, puisque `e` s'applique ainsi à un tableau `numpy`.

Pour les fonctions définies de façon plus compliquée, la commande `np.vectorize(e)(t)` permet d'appliquer la fonction `e` à tous les éléments de `t`, tableau `numpy`. On peut aussi bien sûr le faire à l'aide d'une boucle !

Ci-dessous un exemple de sortie avec un zoom.





III – 1)b)

Idem, mais ici pas de formule pour calculer le traînage, on ajuste "à la main" la valeur de `delta`.

III – 2)

Comme indiqué, j'importe les données dans le tableau `donnees` et je récupère ses dimensions grâce à `np.shape`.

Par ailleurs, je dois adapter les fonctions Y, F en ajoutant le paramètre k, qui permet de récupérer une valeur dans une case $e[k]$ de tableau, puisque je n'ai plus de formule permettant de calculer $e(t_k)$. En conséquence, la fonction Φ doit aussi recevoir la valeur de k pour la transmettre à F.

Autre adaptation nécessaire : calculer la valeur de h à partir du nombre de valeurs dans le tableau (alors que précédemment, on pouvait choisir h). On peut récupérer ladite valeur grâce à l'option `retstep` de la commande `np.linspace`, qui permet de définir le tableau `t` à partir des bornes de l'intervalle et du nombre de points souhaités dans la subdivision (contrairement à `np.arange` à qui l'on fournit les bornes et le pas h).

Une autre solution consisterait à *programmer une fonction* e par interpolation linéaire entre les valeurs contenues dans le tableau.

Note pour la méthode RK4 : le fait qu'elle utilise des valeurs en $t + h/2$ complique les choses... Deux solutions : diviser par deux le nombre d'intervalles de la subdivision (pour que les $e(t_k + h/2)$ se retrouvent dans les données) ou bien utiliser l'interpolation évoquée ci-dessus.

```
import numpy as np
import matplotlib.pyplot as plt

def X(t0,T,h):
    return np.arange(t0,t0+T,h)

def Y(t0,T,h,y0,Phi):
    t=X(t0,T,h)
    y=np.zeros(len(t))
    y[0]=y0
    for k in range(len(t)-1):
        y[k+1]=y[k]+h*Phi(k,t[k],y[k],h)
    return y

def F(k,t,y):
    return b*(eb[k]-y)

def Euler(k,t,y,h):
    return F(k,t,y)

def Heun(k,t,y,h):
    return (F(k,t,y)+F(k+1,t+h,y+h*F(k,t,y)))/2

donnees=np.loadtxt('Filtrage_fort.csv',delimiter=';')
n, p = np.shape(donnees)
t0, T = 0, 1
y0=0
t , h = np.linspace(t0,t0+T,num=n+1,retstep=True)
b=1000

eb=donnees[:,0] #données brutes
ef=donnees[:,1] #données filtrées

plt.grid()
plt.title('Comparaison des 2 méthodes à partir de données discrètes')
plt.xlabel('Valeurs de $t$')
plt.ylabel('Valeurs de $y$')
t=X(t0,T,h)
plt.plot(t,Y(t0,T,h,y0,Euler),'b:',linewidth=2,label='Euler')
plt.plot(t,Y(t0,T,h,y0,Heun),'y--',linewidth=2,label='Heun')
plt.plot(t,eb,'k-.',linewidth=3,label='Entrée brute')
plt.plot(t,ef,'k-',linewidth=1,label='Entrée filtrée')
plt.legend(loc=0)
plt.show()
```