Chapitre 6

Un petit récapitulatif des programmes à connaitre de première année et une étude de leurs complexités. On rappelera aussi les modules à maitriser pour l'application à la physique et les mathématiques.

1 Complexité de programme classique et direct

1.1 Factoriel

Calcul de $n!$ en récursif.

```
def factoriel(n)
""" calcule factoriel n"""
       if n==0:
          return 1
       else:
          return n*factoriel(n-1)
```

Une complexité en $O(n)$, dites linéaire... super rapide.

1.2 Recherche d'un minimum, maximum

Rechercher un maximum ou un minimum dans une liste numérique `L` :

```
def minimum(L):
      min=L[0]
      for k in L:
           if k<min:
              min=k
       return min
```

2 Applications : Diviser pour régner

Nous allons voir comment utiliser la récursivité, en particulier en divisant successivement les objets sur lesquels on travaille. On verra ensuite comment réécrire des programmes déjà vu, le calcul de puissance, factoriel ou de recherche dans un tableau.

2.1 Calcul de puissance

Un moyen d'accélérer le calcul de puissance repose sur le principe " diviser pour régner" :

```
def puissance(x,n):
   if n==1:
            return x
   if n==0:
        return 1
   else:
        r=puissance(x,n//2)# n//2 est le quotien de n par 2
        if n% 2==0: # n est pair
            return r*r
        else:
            return x*r*r
```

```
>>> puissance(2,10)
1024
```

Remarquons que l'évaluation de la complexité $C(n)$, en fonction de la puissance n vérifie :

$$C(n) \leq C(\frac{n}{2}) + 3.$$

Si on pose $U_k = C(2^k)$ on obtient dans le cas $n = 2^k$:$C(n) = U_{\log_2(n)} \leq 3\log_2(n)$ et donc dans le cas général un $0(\log(n))$.

2.2 Méthode de Horner

On peut aussi simplifier la programmation d'évaluation d'un polynôme par la méthode de Horner :

$$P(x) = a_0 + x(a_1 + x(a_2...)$$

```
def evaluationpolynome(p,x):
        if len(p)==1:
            return p[0]
        else:
           q=p[1:len(p)]
           return p[0]+x*evaluationpolynome(q,x)
```

```
>>> evaluationpolynome([1,2,3],2)
17
```

Ici le programme s'écrit exactement comme avec le formalisme mathématiques. Cependant une spécificité du langage python fait que la commande `p[1:len(p)]` consiste à créer une liste de taille `len(p)-1` et réaffecter les valeurs... donc faire $len(p) - 1$ opérations. En effet curieusement il ne s'agit pas d'une lecture partielle de "l'adresse" p, comme se serait le cas si on effectuait l'opération $q = p$!!! On peut y remédier en proposant :

```
def evaluationpolynome(p,x):
        if len(p)==1:
            return p[0]
        else:
            a=p[0]
           del p[:1]
               return a+x*evaluationpolynome(p,x)
```

Remarquons qu'il en va de même avec la commande :

```
>>>L=[1,2,3]+[4]
>>>L
[1,2,3,4] # creation d'une nouvelle liste et donc $4$ affectations
                donc $4$  operations!
>>> L.append(5) # ajoute un element ( empile un element)
                             donc effectue une operation
```

2.3 Vérifier l'appartenance d'une valeur à un tableau trié

Voici une variante d'une recherche dichotomique pour un tableau trié.

```
def recherche_dichotomique(x,a):
            m=len(a)//2
            if x==a[m]:
                       return True

            if m==0:
                      return False

            if x>a[m]:
                     return recherche_dichotomique(x,a[m:len(a)])

           return recherche_dichotomique(x,a[0:m])
```

```
>>> recherche_dichotomique(2,[1,2,3,4,5])
True
```

Ici en posant n la longueur du tableau on trouve comme précédemment une complexité en $O(\log n)$. On peut aussi proposer :

```
def recherche_dichotomique(x,a):
        m=len(a)//2
        if x==a[m]:
                    return True

       if m==0:
                   return False

       if x>a[m]:
              del a[:m]
              return recherche_dichotomique(x,a)

      del a[m:]
      return recherche_dichotomique(x,a)
```

2.4 Dichotomie

Soit f continue sur $[a,b]$ si $f(a).f(b) < 0$ alors on pose $c = \frac{a+b}{2}$ et l'algorithme de dichotomie (ou de bissection) converge avec une vitesse $0(\frac{1}{2^n})$.

```
 def dichotomie(f,a,b,epsilon):
           assert f(a)*f(b)<=0
           if abs(a-b)<=epsilon:
                  return (a+b)/2
           c=(a+b)/2
           if f(c)*f(b)<0:
                      return dichotonmie(f,c,b,epsilon)

           return dichotomie(f,a,c,epsilon)
```

Toujours dans scipy : `scipy.optimize.bissect(f,a,b)`

Ici en posant n la longueur du tableau on trouve comme précédemment une complexité en $O(\log n)$. On peut aussi proposer :

```
 >>> def recherche_dichotomique(x,a):
        m=len(a)//2
            if x==a[m]:
                 return True
            if m==0:
                return False
           if x>a[m]:
                  del a[:m]
                  return recherche_dichotomique(x,a)
          del a[m:]
          return recherche_dichotomique(x,a)
```

3 Tris

3.1 Tris insertion

Le tri insertion en moyenne $O(n^2)$:

```
def insertion(a,b):
    """ a une liste triee, on insert b dans a"""
    l=len(a)
    if b>=a[l-1]:
        return a+[b]
    if b<=a[0]:
        return [b]+a
    while b<a[l-1]and l>0:
        l=l-1
```

```
        return a[0:l]+[b]+a[l:]
def tri_insertion(a):
    l=len(a)
    if l<=1:
        return a
    L=tri_insertion(a[0:l-1])
    return insertion(L,a[l-1])
```

```
def tri_insertion2(a):
        if len(a)==1:
            return a
        for i in range(len(a)):
            if     a[i]>a[i+1]:
                   a[i],a[i+1]=a[i+1],a[i]
        max=a[len(a)-1]
        tri_insertion2(a[0:len(a)-1])
        return a.append(max)
```

3.2 Tris rapides

Le tri rapide, au mieux en $0(n\ln(n))$:

```
import random
def partition(a):
    j=random.randint(0,len(a)-1)
    pivot=a[j]
    partieinf=[]
    partiesup=[]
    for k in range(len(a)):
        if  k!=j:
            if a[k]>pivot:
                partiesup=partiesup+[a[k]]
            else:
                partieinf=[a[k]]+partieinf
    return partiesup,partieinf,pivot
def tri_rapide(a):
    if len(a)<=2:
        return tri_insertion(a)
    if len(a)>2:
        sup,inf,pivot=partition(a)
        return   tri_rapide(sup)+[pivot]+tri_rapide(inf)
```

3.3 Tris fusions

Le tri fusion, en moyenne en $0n\ln(n)$) :

```

def indice(a,b):
    """ a une liste et b un nombre  l'indice de depart"""
    i=0
    while b>a[i]:
        i=i+1
    return i
def fusion(a,b):
    l=len(a)-1
    k=len(b)-1
    if a[0]>=b[k]:
        return b+a
    else:
        if b[0]>=a[l]:
            return a+b
        else:
            j=indice(a,b[0])
```

```
            return a[0:j]+[b[0]]+fusion(a[j:],b[1:])
def tri_fusion(a):
    if len(a)<4:
        return tri_insertion(a)
    else:
        n=len(a)//2
        c=tri_fusion(a[0:n])
        d=tri_fusion(a[n:])
        return fusion(c,d)
```

4 Newton

Voici un petit programme correspondant à l'algorithme de la sécante :

```
def newton(f,x0,df,n):
      def deriv(f,a,b):
            return (f(a)-f(b))/(a-b)
      if n==0:
            return x0
      if n==1:
            return x0-f(x0)/df
      x=newton(f,x0,df,n-1)
      y=newton(f,x0,df,n-2)
      return x-f(x)/deriv(f,x,y)
```

Ici la précision n'est pas explicite, on a juste imposé le nombre de boucle si on veut une précision ϵ il faut estimer n. On peut imposer la recherche de x_0 tant que $\mid f(x_k) \mid > \epsilon$ et on utilise une boucle conditionnelle while :

```
def newton(f,x0,df,epsilon):
      def deriv(f,a,b):
            return (f(a)-f(b))/(a-b)

      x1= x0-f(x0)/df
      while abs(f(x0))>epsilon:
           x0,x1=x1, x1-f(x1)/deriv(f,x0,x1)
      return x0
```

5 Euler

Pour résoudre numériquement une équation différentielle ordinaire on utilise la famille des méthodes Runge Kutta. On commencera avec le grand père... la méthode d'Euler à un pas. Attention ses méthodes n'ont rien d'exacte, on doit normalement vérifier si elle est consistante (si elle marche), stable (ne craint pas trop les erreurs de calcul ou de condition initiale) et convergente (elle approche la solution exacte). On reparlera de tout ça plus tard mais il est important de comprendre que le choix du pas est important, que par conséquent on travaille sur des temps courts.

Pour résoudre l'équation :

$$y' = f(t, y).$$

Avec la méthode d'Euler on choisira le pas tel que $\Lambda\Delta T$ soit assez petit et surtout que ΛT ne soit pas trop grand(Λ est le coefficient de Lipschitz de f, $\Delta T = t_{n+1} - t_n = \frac{T}{N}$) et on construira les points de coordonnées (t_n, y_n) vérifiants :

$$\begin{cases} y_{n+1} = y_n + \Delta T f(t_n, y_n) \\ (y_0, t_0) = (y(0), 0) \\ t_n = \frac{nT}{N} \end{cases}$$

On se place d'abord dans le cadre des fonctions de $\mathbb{R}$ dans $\mathbb{R}$ et on programmera une fonction prenant les paramètres N, T, y_0 et F retournant un couple de listes (les listes X et Y correspondants aux coordonnées x_i, y_i).

On propose un petit programme sur le mode récursif :

```
def Euler(f,y0,DeltaT,N):
  if N==0:
      return y0
  y=Euler(f,y0,DeltaT,N-1)
  return y+\DeltaT*f(t,y)
```

En directe :

```
def euler(f,y0,T,N):
    y=[None]*(N+1)
    x=[i*T/N for i in range(N+1)]
    y[0]=y0
    for i in range(N):
        y[i+1]=y[i]+(T/N)*f(x[i],y[i])
    return x,y
```

Ou bien par concaténation :

```
def euler(f,y0,t0,T,N):
    t=[t0]
    y=[y0]
    for i in range(N)
         t.append(t[i]+((T-t0)/N))
         y.append(y[i]+((T-t0)/N)*f(t[i],y[i]))
    return t,y
```

Il peut être intéressant de représenter graphiquement ces données :

```
def graph_euler(f,y0,T,N):
    t,y=euler(f,y0,t0,T,N)
    import matplotlib.pyplot as pl
    pl.plot(t,y)
    pl.show()
```

Dans le cadre des équations d'ordre supérieur la méthode doit être étendue avec précaution car (à moins d'utiliser le mode `array` de `numpy`) les additions et multiplications sur les listes (ou les uplets) sont des concaténations.
Soit

$$\begin{cases} y'' = f(t, y, y') \\ y'(0) = a \\ y(0) = b \end{cases}$$

On pose $Y = (p, q) = (y', y)$ et $F : (t, Y) \to (f(t, q, p), p)$ et l'équation se ramène alors à :

$$\begin{cases} \frac{dY}{dt} = F(t, Y) \\ Y(0) = (a, b) \end{cases}$$

On adapte alors le programme précédent ainsi :

```
def euler_ordre2(f,y0,t0,T,N):
    DT=(T-t0)/N
    t=[i*DT for i in range(N+1)]
    y=[None]*(N+1)
    y[0]=y0

    for i in range(N):

        y[i+1]=[y[i][0]+DT*f(t[i],y[i])[0],y[i][1]+DT*f(t[i],y[i])[1]]
    return t,y
```

On l'applique alors à l'équation différentielle $y'' = y$ en définissant :

```
def f(t,y):
    return [y[1],y[0]]
```

On peut ainsi obtenir facilement le portrait de phase :

```
import matplotlib.pyplot as pl
t,y= euler_ordre2(f,y0,t0,T,N)
p,q=y
pl.pyplot(p,q)
pl.show()
```

On peut bien sûr le généraliser aux ordre supérieur :

```
def euler_ordrep(f,y0,t0,T,N):
    DT=(T-t0)/N
    p=len(y0)
    t=[i*DT for i in range(N+1)]
    y=[None]*(N+1)
    y[0]=y0

    for i in range(N):
        y[i+1]=[None]*p
        for j in range(p):
            y[i+1][j]=y[i][j]+DT*f(t[i],y[i])[j]
    return t,y
```

6 Integration

Dans le cas des rectangles on a poser :

$$\int_0^1 f(t)dt \simeq g(1).$$

Ce qui nous donne une approximation en $O(\frac{1}{n})$ et se programme très simplement de la façon suivante :

```
def int_rectangle(f,n,a,b):
    """calcule l'integrale de f entre a et b avec un pas de n"""
    assert a!=b

    S=0
    for i in range(n):
        xi=a+i*(b-a)/n
        S=f(xi)*(b-a)/n+S
    return S
```

Dans le cas d'ordre supérieur, méthode du point milieux, de Simpson et autre....

```
 def approx_quad(g):
    return (g(0)+g(1))/2
def integration_quad(f,a,b,epsilon):
    n=1+int((b-a)/epsilon)
    integf=0
    for i in range(n):
        def g(t):
            return ((b-a)/n)*f(a+((t+i)*(b-a)/n)
        integf=integf+approx_quad(g)
    return integf
def approx_simpson(g):
    return (g(0)+g(1))/6+2*g(0.5)/3
def integration_simpson(f,a,b,epsilon):
    n=1+int((b-a)/epsilon)
    integf=0
    for i in range(n):
        def g(t):
            return ((b-a)/n)*f(a+((t+i)*(b-a)/n)
        integf=integf+approx_simpson(g)
    return integf
```

7 Piles

```
 >>>[] # creer une pile
 >>>p.pop() # retire  le dernier terme de la liste et le retourne
 >>>p.append(v) # ajoute v comme dernier terme dans la liste
 >>>len(p) # retourne la taille de la pile
 >>>p[-1]# retourne le sommet de la pile.
```

pour une file d'attente : premier arrivé, premier servi.

```
def depiler_gauche(P):
      Q=P[0]
      del P[0]
      return Q
```

On peut aussi dépiler et empiler dans une autre pile :

```
P2=P2.append(P1.pop())
```

Plus généralement... tout dépiler :

```
def depiler_empiler(P,Q):
""" Depile P et l'empiler dans Q"""
   while len(P)>0:
        Q=Q.appen(P.pop())
```

8 Les modules

8.1 Odeint

Pour résoudre $y' = f(y,t)$ ou f est une fonction de plusieurs variables. On doit importer la bibliothèque `scipy.integrate` :

```
>>> import scipy . integrate as *
#ou bien
>>> from spicy.integrate import odeint
```

Elle prend pour argument la fonction f (qu'on peut définir avec lambda) la condition initiale y_0 et une liste de temps $t = [t_0, ...t_i, ...t_n]$. Elle retourne alors la liste des approximation y_i :

```
>>>odeint(lambda x,t:x,1,[0,0.1,0.2,0.3])
```

On peut créer un tableau de temps en utilisant :

```
>>> [i/n for k in range(n+1)]
```

Dans le cadre de système et donc d'équation différentielle d'ordre p, la fonction retourne une liste de listes de taille $p-1$ (la liste des Y_i) et la condition initiale doit être une liste. Par exemple :

$$\begin{cases} y'' = y \\ y'(0) = 1 \\ y(0) = 1 \end{cases}$$

Se traduit par le système :

$$\begin{cases} \frac{dy_0}{dt} = y_1 \\ \frac{dy_1}{dt} = y_0 \\ y_1(0) = 1 \\ y_0(0) = 1 \end{cases}$$

```
def f(y,t):
     return y[1],y[0]
t=[0.01*i for i in range(101)]
from scipy.integrate import odeint
Y=odeint(f,[1,1],t)
y=[Y[i][0] for i in range(101)] # ou aussi dans le mode array de numpy Y[:,0]
y1=[Y[i][1] for i in range(101)]# Y[:,1]
import matplotlib.pyplot as pl
pl.plot(t,y)# tracer de la courbe de y(t)
pl.plot(y,y1)# portrait de phase
pl.show()
```

Remarquons que si on utilise directement *plot* sur Y :

```
>>>Y=odeint(f,[1,1],t)
>>>pl.plot(Y)
>>>pl.show()
```

On obtient le tracé des points (i, y_i) et (i, y_i')(et non pas (t_i, y_i)

8.2 quad

Dans la bibliothèque `scipy.integrate`, on peut utiliser : `quad(f,a,b)`, que je vous invite à tester. Par exemple $\int_0^1 xdx$ donne bien :

```
>>> import scipy.integrate as intg

>>> intg.quad(lambda x:x,0,1)
>>> (0.5, 5.551115123125783e-15)
```

8.3 Newton et Dichotomie

```
>>> Import scipy.optimize
>>>scipy.optimize.newton(f,a,f')
>>>scipy.optimize.bissect(f,a,b)
```

8.4 Numpy et matrices

```
>>> from numpy import *
>>> from scipy import linalg
>>> A=array([ [1,1,1],[1,0,2],[1,1,0] ]) # On utilise le mode array de numpy
>>> B=array([1,1,1]) # B etant definie en ligne
>>> X=linalg.solve(A,B) # resout AX=B.
>>> X #   X ressort en  en ligne

array([ 1.,  0.,  0.])
```

On peut aussi obtenir l'inverse d'une matrice et multiplier ces matrices dans le mode `array` de `numpy`.

```
from numpy import *
 M=array([ [1,1,1],[1,0,2],[1,1,0] ])
 linalg.inv(M)
linalg.det(M) # determine le determinant.
dot(A,B) # multiplie A et B.

import numpy as np
A=np.array([[1,2,3],[2,4,5],[1,2,5]])# est une matrice 3 3
2*A
array([[ 2,  4,  6],
       [ 4,  8, 10],
       [ 2,  4, 10]])
B=np.zeros((3,3)) # creer une matrice 3 3 de zeros
B
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
B[0,1]=3 # affecter 3 a ligne 1 colonne 2
B
array([[ 0.,  3.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

B+3*A
array([[  3.,   9.,   9.],
       [  6.,  12.,  15.],
       [  3.,   6.,  15.]])
A[1]
array([2, 4, 5])
A[0,1]
2
A[:,1] # colonne 2
array([2, 4, 2])
```

```
np.dpt(A,B) # produit AB
array([[ 0.,  3.,  0.],
       [ 0.,  6.,  0.],
       [ 0.,  3.,  0.]])
```

8.5 Tris

```
L.sort() # tri par ordre croissant
L.reverse() #  change l'ordre
min(L) # ressort le min
max(a) # je vous laisse deviner
L.insert(i,a)# insert le terme a en position i
del L[i] # efface le terme en position i
L[i:j] # extrait la sous liste d'indice i a j-1
L.remove(a)# retirer l'element a de L
L.count(i)# indique le nombre de terme de L de valeur i
```