TP7

October 25, 2022

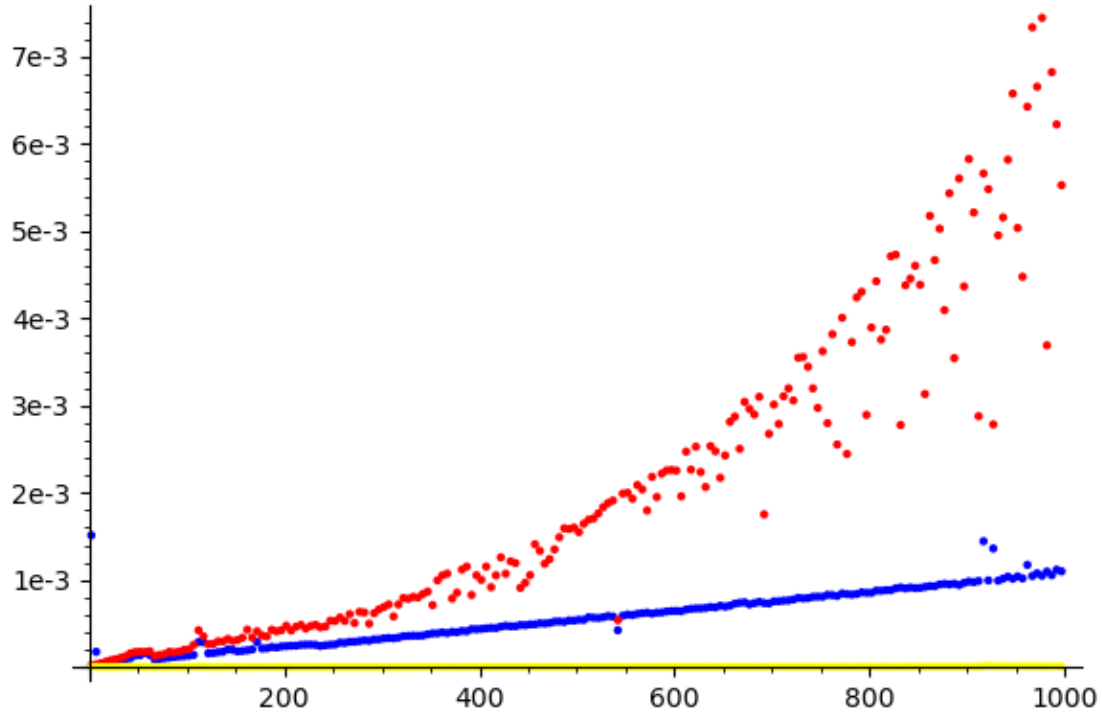
0.0.1 Exo 1 : Horner

[1]:
```
def Horner(P,a):
    d=P.degree()
    S=0
    for i in range(d,-1,-1):
        S=P[i]+a*S
    return S

def EvalueNaif(P,c):
    return sum(P[i]*c**i for i in range(P.degree()+1))
```

[2]:
```
from time import *

def TempsHorner(P,a):
    t0=time()
    Horner(P,a)
    t1=time()
    return(t1-t0)

def TempsEvalueNaif(P,a):
    t0=time()
    P=EvalueNaif(P,a)
    t1=time()
    return (t1-t0)

def TempsEvalueSage(P,a):
    t0=time()
    ev=P(a)
    t1=time()
    return(t1-t0)

N=1000; p=next_prime(N);

F.<X>=GF(p)['X']

T0=[]; T1=[]; T2=[]
```

```
for k in range(2,N+1,5):
    a=randint(1,k)
    P=F.random_element(degree=k)
    t=TempsHorner(P,a)
    T0.append([k,t])
    t=TempsEvalueNaif(P,a)
    T1.append([k,t])
    t=TempsEvalueSage(P,a)
    T2.append([k,t])


point(T0,color='blue')+point(T1,color='red')+point(T2,color='yellow')
```

[2]:



[3]:
```
def DivisionHorner(P,a):
    d=P.degree()
    S=0
    Q=0
    for i in range(d,0,-1):
        S=P[i]+a*S
        Q=Q+S*X**(i-1)
    S=P[0]+a*S
    return [Q,S]
```

```
P=F.random_element(degree=10)
print(DivisionHorner(P,5))
print(P//(X-5),P%(X-5))
```

```
[771*X^9 + 916*X^8 + 43*X^7 + 699*X^6 + 828*X^5 + 124*X^4 + 351*X^3 + 885*X^2 +
288*X + 743, 978]
771*X^9 + 916*X^8 + 43*X^7 + 699*X^6 + 828*X^5 + 124*X^4 + 351*X^3 + 885*X^2 +
288*X + 743 978
```

[4]:
```
def Taylor(P,a):
    L=[]
    for i in range(P.degree()+1):
        [P,S]=DivisionHorner(P,a)
        L.append(S)
    return L

L=Taylor(P,5)
print(L)
P==sum(L[i]*(X-5)**i for i in range(len(L)))
```

```
[978, 109, 547, 742, 579, 811, 466, 964, 67, 296, 771]
```

[4]:
```
True
```

0.0.2 Exo 2 : Evaluation multipoints

[5]:
```
def Arbre(A):  # A=[a_0,...,a_n] avec n+1=2**k
    n=len(A)-1
    k=log(n+1,2)
    B=[]
    B0=[X-A[j] for j in range(n+1)]
    B.append(B0)
    for i in range(1,k):
        Bprec=B[i-1]
        Bsuiv=[Bprec[j]*Bprec[j+1] for j in range(0,len(Bprec),2)]
        B.append(Bsuiv)
    return B

R.<X>=QQ['X']
A=[i for i in range(8)]
B=Arbre(A)
print(B)

def EvalueMultipoint(P,A):
    B=Arbre(A)
    B.append([P])
    for i in range(len(B)-2,-1,-1):
```

```
        for j in range(len(B[i+1])):
            B[i][2*j]=B[i+1][j]%B[i][2*j]
            B[i][2*j+1]=B[i+1][j]%B[i][2*j+1]
    return B[0]
```

```
[[X, X - 1, X - 2, X - 3, X - 4, X - 5, X - 6, X - 7], [X^2 - X, X^2 - 5*X + 6,
X^2 - 9*X + 20, X^2 - 13*X + 42], [X^4 - 6*X^3 + 11*X^2 - 6*X, X^4 - 22*X^3 +
179*X^2 - 638*X + 840]]
```

[6]:
```
R.<X>=QQ['X']

A=[1,2,3,4,5,6,7,8]
P=R.random_element(degree =7)
L=EvalueMultipoint(P,A)
print(P)
print(L)
```

```
-X^7 - X^6 - X^5 + X^3 - 1/14*X^2 + 3/2*X - 2
[-18/7, -1507/7, -21911/7, -150060/7, -677224/7, -2339033/7, -957651,
-16740826/7]
```

0.0.3 Exo 3 : Interpolation

[7]:
```
R.<X>=QQ['X']

def Lagrange(A,B):
    n=len(A)
    P=0
    for i in range(n):
        Li=1
        for j in range(n):
            if i!=j:
                Li=Li*(X-A[j])/(A[i]-A[j])
        P=P+B[i]*Li
    return P

def InterpolationRapide(A,B):
    r=1
    while r<len(A):
        r=2*r
    if r==1:
        return B[0]
    B1=B[:r//2]
    B2=B[r//2:]
    A1=A[:r//2]
    A2=A[r//2:]
    P1=InterpolationRapide(A1,B1)
```

```
    P2=InterpolationRapide(A2,B2)
    Q1=prod([X-A1[i] for i in range(len(A1))])
    Q2=prod([X-A2[i] for i in range(len(A2))])
    D,U,V=xgcd(Q1,Q2)
    P=P2*U*Q1 + P1*V*Q2
    return(P%(Q1*Q2))
```

[8]:
```
R.<X>=QQ['X']

A=[1,2,3,4,5,6,7,8]
B=[2,1,-5,3,-4,2,9,-7]

P1=Lagrange(A,B)
print(P1)
print(EvalueMultipoint(P1,A)==B)

P2=InterpolationRapide(A,B)
print(P2)

Noeuds=[(A[i],B[i]) for i in range(len(A))]

P3=R.lagrange_polynomial(Noeuds)
print(P3)
```

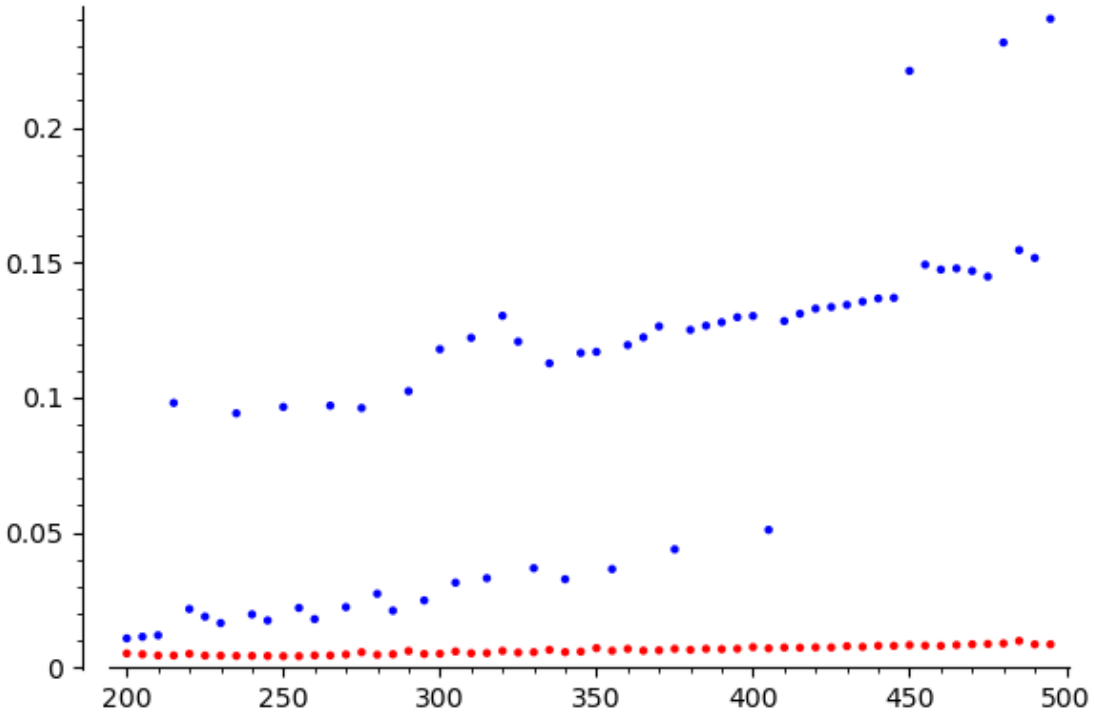
```
109/1680*X^7 - 151/72*X^6 + 3319/120*X^5 - 3445/18*X^4 + 178303/240*X^3 -
114913/72*X^2 + 181004/105*X - 703
True
109/1680*X^7 - 151/72*X^6 + 3319/120*X^5 - 3445/18*X^4 + 178303/240*X^3 -
114913/72*X^2 + 181004/105*X - 703
109/1680*X^7 - 151/72*X^6 + 3319/120*X^5 - 3445/18*X^4 + 178303/240*X^3 -
114913/72*X^2 + 181004/105*X - 703
```

[20]:
```
N=2**20
A=list(range(N))
B=[randint(0,N) for i in range(N)]
Noeuds=[(A[i],B[i]) for i in range(len(A))]

p=next_prime(N)
R.<X>=GF(p)['X']

from time import *

def TempsSage(n):
    t0=time()
    R.lagrange_polynomial(Noeuds[:n])
    t1=time()
    return(t1-t0)
```

```
def TempsRapide(n):
    t0=time()
    P=InterpolationRapide(A[:n],B[:n])
    t1=time()
    return (t1-t0)

def TempsLagrange(n):
    t0=time()
    P=Lagrange(A[:n],B[:n])
    t1=time()
    return (t1-t0)

print(TempsSage(2**10))
print(TempsRapide(2**10))
print(TempsLagrange(2**10))

T0=[]
for k in range(200,500,5):
    t=TempsSage(k)
    T0.append([k,t])

T1=[]
for k in range(200,500,5):
    t=TempsRapide(k)
    T1.append([k,t])

point(T0,color='blue')+point(T1,color='red')
```

```
0.7823255062103271
0.02080678939819336
6.9482645988464355
```

[20]:



0.0.4 Exo 4: FFT

[10]:
```
def Ev(G,a,n): # deg(G)<n, avec n puissance de 2. Et a racine n-eme primitive de 1
    if n==1:
        return [G]
    k=n//2
    R0=sum((G[i]+G[i+k])*X**i for i in range(k))
    R1t=sum((G[i]-G[i+k])*a**i*X**i for i in range(k))
    L0=Ev(R0,a**2,k)
    L1=Ev(R1t,a**2,k)
    L=[]
    for i in range(k):
        L.extend([L0[i],L1[i]])
    return L
```

[11]:
```
p=29*2**(57)+1
print(is_prime(p))
F=GF(p)
a=F.primitive_element()  # racine primitive (p-1)-ème de 1 dans F_p

def Multiplie(G,H):
    n=1
```

```
    while n<=G.degree()+H.degree():
        n=2*n
    b=a**((p-1)//n)  # racine primitive n-ème de 1 dans F_p.
    LG=Ev(G,b,n)  # evaluation multipoints de G en 1,b,...,b**(n-1)
    LH=Ev(H,b,n)  # evaluation multipoints de H en 1,b,...,b**(n-1)
    LF=[LG[i]*LH[i] for i in range(n)]   # produit points à points␣
↪G(b**i)H(b**i) pour i=0,...,n-1
    F=K(Ev(LF,b**(-1),n))//n  # interpolation
    return F


K.<X>=F['X']
G=K.random_element(degree=1000)
H=K.random_element(degree=1000)
%time F1=G*H
%time F2=Multiplie(G,H)
print(F1==F2)
```

```
True
CPU times: user 226 µs, sys: 3 µs, total: 229 µs
Wall time: 232 µs
CPU times: user 968 ms, sys: 3.92 ms, total: 972 ms
Wall time: 971 ms
True
```

0.0.5 Exo 5 : Localisation de racines

[12]:
```
def Sturm(P):
    P0=P
    P1=P.derivative()
    L=[P0,P1]
    while P1!=0:
        P0,P1=P1,-P0%P1
        L.append(P1)
    return L

def V(P,c):
    compt=0
    L=Sturm(P)
    for i in range(len(L)-1):
        if L[i](c)*L[i+1](c)<0:
            compt+=1
    return compt

def NombresRacines(P,a,b):
    return V(P,a)-V(P,b)
```

```
R.<X>=RR['X']
P=R.random_element(degree=10)
a=-1
b=1
N=NombresRacines(P,a,b)
print('Nombres racines entre a=',a,'et b=',b,':',N)
print('Racines réelles :\n',P.roots())
```

```
Nombres racines entre a= -1 et b= 1 : 1
Racines réelles :
 [(-1.35506944957858, 1), (0.335266416169948, 1), (1.15068316485421, 1),
(3.42003151011891, 1)]
```

[13]:
```
def NombreTotalRacines(P):
    n=P.degree()
    M=1+max([abs(P[i]/P[n]) for i in range(n)])
    N=V(P,-M)-V(P,M)
    print('Nombres total de racines réelles:',N)
    print('Racines réelles :\n',P.roots())
    return N

R.<X>=RR['X']
P=R.random_element(degree=10)
NombreTotalRacines(P)
```

```
Nombres total de racines réelles: 4
Racines réelles :
 [(-1.16531913031712, 1), (-0.739747891523217, 1), (0.799115937350319, 1),
(2.22856266121904, 1)]
```

[13]:
```
4
```

[14]:
```
def Racines(P,a,b,e):
    N=NombresRacines(P,a,b)
    if N==0:
        return([])
    if N==1 and b-a<e:
        return [float((a+b)/2)]
    m=float((a+b)/2)
    R1=Racines(P,a,m,e)
    R2=Racines(P,m,b,e)
    return R1+R2

def RacinesGlobal(P,e):
    n=P.degree()
    M=1+max([abs(P[i]/P[n]) for i in range(n)])
```

```
    a=-M
    b=M
    N=V(P,a)-V(P,b)
    if N==0:
        print('pas de racines réelles')
        return([])
    else:
        print('Nombres total de racines réelles:',N)
        L=Racines(P,a,b,e)
        print('Approximations à précision',e,':',L)
    return(L)

R.<X>=RR['X']
P=R.random_element(degree=10)
RacinesGlobal(P,0.1)
P.roots()
```

```
Nombres total de racines réelles: 4
Approximations à précision 0.100000000000000 : [-4.26873981759748,
-1.246623132572715, 0.6988644834119766, 0.7366409419747861]
```

[14]:
```
[(-4.24776067260813, 1),
 (-1.21780052609836, 1),
 (0.683622434412369, 1),
 (0.728087957374937, 1)]
```

[15]:
```
def Newton(P,x0,e):
    Q=P.derivative()
    c=1
    while abs(P(x0))>e and c<100:  # Introduit compteur en garde fou en cas de
 ↪divergence
        x0=x0-P(x0)/Q(x0)
        c=c+1
    return [x0,c]

def RacinesGrandePrecision(P,e,E):
    L=RacinesGlobal(P,e)
    for i in range(len(L)):
        L[i]=Newton(P,L[i],E)
    print('Racines à grande précisions',E,':',L)
    return L

R.<X>=RR['X']
P=R.random_element(degree=10)
e=1e-1; E=1e-11
%time RacinesGlobal(P,E)
print()
```

```
%time RacinesGrandePrecision(P,e,E)
P.roots()
```

```
Nombres total de racines réelles: 2
Approximations à précision 1.00000000000000e-11 : [0.9544556549126106,
2.42057095061915]
CPU times: user 34.8 ms, sys: 0 ns, total: 34.8 ms
Wall time: 34.2 ms

Nombres total de racines réelles: 2
Approximations à précision 0.100000000000000 : [0.9841518046673094,
2.409475107978585]
Racines à grande précisions 1.00000000000000e-11 : [[0.954455654915176, 5],
[2.42057095061837, 5]]
CPU times: user 5.8 ms, sys: 17 µs, total: 5.81 ms
Wall time: 5.67 ms
```

[15]: [(0.954455654915173, 1), (2.42057095061837, 1)]