LM203 — Laurent Koelblen — TP des 20 et 27 octobre 2006

À propos d'un algorithme de factorisation des nombres entiers

I — Un petit outil de statistique

D'une suite de nombres $L = [L_1, L_2, \ldots, L_N]$ dans un intervalle $[a, b[$ on compte combien sont dans les intervalles $\left[a, a + \frac{b-a}{n}\right[$, $\left[a + \frac{b-a}{n}, a + 2\frac{b-a}{n}\right[$,..., $\left[a + (n-1)\frac{b-a}{n}, b\right[$ et on en fait un graphique :

```
histogram(L,a,b,n):= block
(
    [L1,N1,L2,i],
    N1:length(L),
    L1:makelist( ?truncate(float((L[i]-a)/(b-a)*n)) ,i,1,N1),
    L2:[],
    for i from 0 thru n-1 do
    (
        L1:delete(i,L1),
        N2:N1-length(L1),
        L2:endcons([float(a+i*(b-a)/n),N2],L2),
        L2:endcons([float(a+(i+1)*(b-a)/n),N2],L2),
        N1:length(L1)
    ),
    L2:cons([float(a),0],L2),
    L2:cons([float(a-(b-a)/n),0],L2),
    L2:endcons([float(b),0],L2),
    L2:endcons([float(b+(b-a)/n),0],L2),
    plot2d([discrete,L2],[gnuplot_curve_styles,["with lines"]])
)$
```

Fonctionnement de la procédure :

`L,a,b,n`
: sont les paramètres d'entrée. `L` est une liste de nombres (entiers où en virgule flottante) compris entre `a` et `b`. `n` est le nombre de sous intervalles de l'intervalle $[a, b]$.

`L1,N1,L2,i`
: sont les variables dites locales : leurs valeurs sont indépendantes des variables de mêmes noms définies éventuellement hors de la procédure.

`N1:length(L)`
: on stocke la longueur de la liste `L` dans la variable `N1`

`L1:makelist( ?truncate(float((L[i]-a)/(b-a)*n)) ,i,1,N1)`

`?truncate` est la fonction « partie entière ». `L1` est donc une suite d'entier compris entre 0 et $n-1$: pour chaque terme de la liste `L` dans l'intervalle

$$\left[a+i\frac{b-a}{n}, a+(i+1)\frac{b-a}{n}\right[$$

le terme correspondant de la liste L1 vaut `i`.

`L2:[]`

on initialise la variable `L2` en lui donnant comme valeur la liste vide. (`L2` est destiné ensuite à recevoir une liste de points du plan exprimés sous la forme `[x,y]`.)

```
for i from 0 thru n-1 do
(
    L1:delete(i,L1),
    N2:N1-length(L1),
    L2:endcons([float(a+i*(b-a)/n),N2],L2),
    L2:endcons([float(a+(i+1)*(b-a)/n),N2],L2)
    N1:length(L1),
)
```

Faisons une description pas à pas de cette boucle ; pour $i=0$:

`L1:delete(0,L1)`

on efface de la liste tous les termes qui sont égaux à 0.

`N2:N1-length(L1)`

`N1` était égal à la longueur de la liste avant qu'on supprime les termes égaux à 0. `N2` reçoit donc le nombre de 0 qu'il y avait dans la liste `L1`.

`L2:endcons([float(a),N2],L2),`
`L2:endcons([float(a+(b-a)/n),N2],L2)`

on ajoute à la liste `L2` les points de coordonnées $[a, \texttt{N2}]$ et $\left[a+\frac{b-a}{n}, \texttt{N2}\right]$. On remarque que `N2` est égal au nombre d'éléments de la liste `L` compris dans l'intervalle $\left[a, a+\frac{b-a}{n}\right[$.

`N1:length(L1)`

on stocke la nouvelle longueur de la liste `L1` dans la variable `N1`

Puis pour $i=1$ on ajoutera à la liste `L2` les points de coordonnées $\left[a+\frac{b-a}{n}, \texttt{N2}\right]$ et $\left[a+2\frac{b-a}{n}, \texttt{N2}\right]$ où `N2` sera cette fois égal au nombre d'éléments de la liste `L` compris dans l'intervalle $\left[a+\frac{b-a}{n}, a+2\frac{b-a}{n}\right[$.

```
L2:cons([float(a),0],L2),
L2:cons([float(a-(b-a)/n),0],L2),
L2:endcons([float(b),0],L2),
L2:endcons([float(b+(b-a)/n),0],L2)
```

on ajoute en début et en fin de la liste `L2` des points d'ordonnée nulle. Cela uniquement pour obtenir un graphique satisfaisant à l'aide la commande suivante :

`plot2d([discrete,L2],[gnuplot_curve_styles,["with lines"]])`
fabrique un graphique à partir de la liste de points `L2` en les joignant par des lignes.

Rien ne vaut quelques exemples pour mieux illustrer l'utilisation et le fonctionnement de cette procédure.

Voici donc quelques essais sur des suites aléatoires de nombres.

1) Tout d'abord sur un tirage à « pile ou face » : on tire au hasard des suites de « 0 » et de « 1 » puis on visualise le nombre de « 0 » et de « 1 » obtenus :

```
L:makelist(random(2),i,1,100) $
histogram(L,0,2,2);

L:makelist(random(2),i,1,1000) $
histogram(L,0,2,2);

L:makelist(random(2),i,1,10000) $
histogram(L,0,2,2);
```

Que se passe-t-il lorsque la longueur de la liste aléatoire augmente ?

2) Voici ensuite des suites aléatoires de types différents.

2a) 100 nombres en virgule flottante compris dans l'intervalle $[0, 1[$, avec une division de l'intervalle en 10 :

```
L:makelist(random(1.0),i,1,100) $
histogram(L,0,1,10);
```

2b) 1000 nombres entiers dans l'intervalle $[0, 10^{10}[$, avec une division de l'intervalle en 10 :

```
L:makelist(random(10**10),i,1,1000) $
histogram(L,0,10**10,10);
```

2c) 10000 nombres en virgule flottante compris dans l'intervalle $[0, 1[$, avec une division de l'intervalle en 10, puis en 100 :

```
L:makelist(random(1.0),i,1,10000) $
histogram(L,0,1,10);
histogram(L,0,1,100);
```

Que se passe -t-il lorsque le nombre de division de l'intervalle augmente ?

3) Voici maintenant une suite de 10000 nombres entiers dans un intervalle $[0, N[$ (N étant choisi de façon aléatoire entre 0 et 10^{10}) donnés par la formule : $u_0 = 1$, $u_{i+1} \equiv u_i^2 + 1 \pmod{N}$.

```
N:random(10**10) ;
u:1 $
L:[u] $
from 1 thru 10000 do
(
    u:remainder(u**2+1,N),
    L:endcons(u,L)
) $
histogram(L,0,N,10);
histogram(L,0,N,100);
```

Au vu des histogrammes, cette suite a-t-elle l'air d'être aléatoire ?

II — Factorisation

1) Voici une procédure naïve pour trouver un diviseur (le plus petit) d'un nombre n : on teste tous les nombres i compris entre 2 et $\sqrt{n}$ pour savoir s'ils divisent n :

```
divisor_naive(n):=block
(
    [i],
    i:2,
    while ( is(i**2<n) and is(remainder(n,i)#0) ) do i:i+1,
    if remainder(n,i)=0 then return(i) else return(n)
) $
```

On va tester cette procédure sur des nombres qui sont produits de deux nombres premiers. Pour cela on va utiliser les procédures suivantes :

`prevprime(n)` cherche le plus grand nombre premier inférieur ou égal à n.

```
prevprime(n):=block
(
    while not primep(n) do n:n-1,
    n
) $
```

`nextprime(n)` cherche le plus petit nombre premier supérieur ou égal à n.

```
nextprime(n):=block
(
    while not primep(n) do n:n+1,
    n
) $
```

`randomprime(a,b)` tire au hasard un nombre premier dans l'intervalle $[a, b[$.

```
randomprime(a,b):=block
(
    [c,d],
    c:prevprime(?truncate(float(b))),
    d:?truncate(float(a+(c-a)*random(1.0))),
    if d<a then d:d+1,
    nextprime(d)
) $
```

`randomproduct(a,b)` tire au hasard un produit de `k` nombres premiers dans l'intervalle [`a`, `b`[.

```
randomproduct(a,b,k):=block
(
  [c],
  c:1,
  from 1 thru k do c:c*randomprime(a,b),
  c
) $
```

Alors que donne les tests suivants ?

```
showtime:true;

n:randomproduct(10**2,10**3,2);
divisor_naive(n);

n:randomproduct(10**3,10**4,2);
divisor_naive(n);

n:randomproduct(10**4,10**5,2);
divisor_naive(n);
```

Est-il raisonnable de pousser les tests avec des nombres plus grands ? (Estimez le temps de calcul pour un nombre produit de 2 nombres premiers de 6 chiffres, 7 chiffres, etc.)

2) On va maintenant écrire une autre procédure, basée sur un algorithme de Pollard. Soit n le nombre dont on veut trouver un diviseur.

L'idée est de tirer des nombres w_0, w_1, etc., au hasard entre 0 et $n-1$ et de tester si l'on a pour deux d'entre eux (w_i et w_j) la propriété $\text{pgcd}(w_j - w_i, n) \neq 1$. Si c'est le cas on a trouvé un diviseur de n (ce pgcd).

Pour mettre cette idée en pratique on introduit la suite pseudo aléatoire définie par $u_0 = 1$, $u_{i+1} \equiv u_i^2 + 1 \pmod{n}$.

On remarque aussi que pour cette suite on a la propriété suivante : si $\text{pgcd}(u_j - u_i, n) = d \neq 1$ alors on a $u_j \equiv u_i \pmod{d}$ et alors $u_j^2 \equiv u_i^2 \pmod{d}$ donc $u_j^2 + 1 \equiv u_i^2 + 1 \pmod{d}$ c'est à dire $u_{j+1} \equiv u_{i+1} \pmod{d}$. Il en résulte que pour tout $s > 0$ on a $u_{j+s} \equiv u_{i+s} \pmod{d}$.

Alors : si $2i < j$, posant $s = j - 2i$ on a $u_{j-i} \equiv u_{2(j-i)}$.

Et si $2i > j$, posant $s = j - i$ on a $u_i \equiv u_j \equiv u_{2j-i} \equiv u_{3j-2i} \equiv \ldots \pmod{d}$. Dans cette suite il y aura nécessairement un indice $> 2i$ et on peut se ramener au cas précedent.

Ce qui montre que s'il y a deux termes de la suite, u_i et u_j, tels que $u_i \equiv u_j \pmod{d}$ alors il existe un indice k tel que $u_k \equiv u_{2k} \pmod{d}$ ou encore, pour se ramener au problème initial, s'il y a deux termes de la suite, u_i et u_j, tels que $\text{pgcd}(u_j - u_i, n) \neq 1$, alors il existe un indice k tel que $\text{pgcd}(u_{2k} - u_k, n) \neq 1$.

On simplifie donc la méthode en cherchant simplement deux termes de la suite u_k et u_{2k} tels que $\text{pgcd}(u_{2k} - u_k, n) \neq 1$.

C'est ce qu'on met en pratique dans la procédure suivante, dans laquelle `u` désigne le terme général de la suite (u_i) et `v` le terme général de la suite (u_{2i}) :

```
divisor_pollard(n):=block
(
    [u,v],
    u:2,
    v:remainder(u**2+1,n),
    while ( gcd(v-u,n)=1 ) do
    (
        u:remainder(u**2+1,n),
        v:remainder(v**2+1,n),
        v:remainder(v**2+1,n)
    ),
    gcd(v-u,n)
) $
```

Faisons quelques tests :

```
makelist([i,divisor_pollard(i)],i,2,100);
```

On constate que la procédure ne donne pas de diviseur propre des nombres : 4, 8, 14, 16, 25, 28, 32, 56. On peut supposer qu'il y a bien d'autres nombres non premiers pour lesquels la procédure échoue à donner un diviseur propre.

On peut tenter d'éliminer ce genre de problème en modifiant le premier terme de la suite ainsi que la formule de récurrence.

On choisira donc $0 \leq u_0 < n$ de façon aléatoire et $u_{i+1} = u_i^2 + c \pmod{n}$ où $0 \leq c < n$ sera aussi choisi de façon aléatoire.

Si la procédure échoue à donner un diviseur propre en recommencera avec des nouvelles valeurs de u_0 et de c.

Il faut également tester si le nombre est premier avant de lancer la boucle. (Si n est premier, il n'a pas de diviseur propre, et la boucle ne se termine jamais.)

Ce qui donne la nouvelle procédure suivante :

```
divisor_pollard(n):=block
(
    [u,v,c,d],
    if primep(n)
    then return(n)
    else
    (
        d:n,
        while d=n do
        (
            c:random(n),
            u:random(n),
            v:remainder(u**2+c,n),
            while gcd(v-u,n)=1 do
            (
                u:remainder(u**2+c,n),
                v:remainder(v**2+c,n),
                v:remainder(v**2+c,n)
            ),
            d:gcd(v-u,n)
        ),
        return(d)
    )
) $
```

On peut tester cette procédure sur des nombres, produits de deux nombres premiers, pour lesquels la première procédure « naïve » prenait un temps trop long :

```
showtime:true;

n:randomproduct(10**7,10**8,2);
divisor_pollard(n);

n:randomproduct(10**8,10**9,2);
divisor_pollard(n);

n:randomproduct(10**9,10**10,2);
divisor_pollard(n);
```

Voici une modification de la procédure qui retourne le couple $\left[d, \frac{k}{\sqrt{d}}\right]$ où d est le diviseur trouvé et k le nombre de boucles effectuées par la procédure :

```
divisor_pollard_mod(n):=block
(
    [u,v,c,d,k],
    if primep(n)
    then return([n,0])
```

```
        else
        (
            d:n,
            while d=n do
            (
                k:0,
                c:random(n),
                u:random(n),
                v:remainder(u**2+c,n),
                while gcd(v-u,n)=1 do
                (
                    k:k+1,
                    u:remainder(u**2+c,n),
                    v:remainder(v**2+c,n),
                    v:remainder(v**2+c,n)
                ),
                d:gcd(v-u,n)
            ),
            return([d,float(k/sqrt(d))])
        )
    ) $
```

On peut s'en servir pour faire des graphiques sur lesquels apparaissent le rapport $\frac{k}{\sqrt{d}}$ en fonction de d (pour des nombres produits de 2 nombres premiers aléatoires).

On va utiliser une variante de la fonction `randomproduct(n,r,k)` :

`randomproduct2(n,r,k)` tire au hasard un produit de `k` nombres premiers dans un intervalle $\left[\frac{m}{r}, mr\right[$ où `m` est tiré au hasard entre `2` et `n`.

```
    randomproduct2(n,r,k):=block
    (
        [m],
        m:2+random(n-1),
        randomproduct(m/r,m*r,k)
    ) $

    L1:makelist( randomproduct2(10**4,2,2) ,i,1,200) $
    L2:map( divisor_pollard_mod , L1) $
    plot2d([discrete,L2],[gnuplot_curve_styles,["with points"]]);

    L1:makelist( randomproduct2(10**6,2,2) ,i,1,200) $
    L2:map( divisor_pollard_mod , L1) $
    plot2d([discrete,L2],[gnuplot_curve_styles,["with points"]]);
```

Que constatez-vous sur ces graphiques ?

Expliquez, en terme mathématique, de combien la procédure `divisor_pollard` est plus efficace que la procédure `divisor_naive`.