

TP2 : Equations Diophantiennes linéaires

N'oubliez pas d'exécuter (valider avec la touche Entrée) les commandes Maple (texte en rouge) avant de les utiliser.

Les exercices en petits caractères sont facultatifs.

– Calcul des coefficients de Bézout

On utilisera la fonction **igcdex** de Maple :

```
[ > igcdex(456,123,'u','v');  
                                     3
```

qui rend le pgcd mais en plus

```
[ > u,v,u*456+v*123;  
                                     17, -63, 3
```

met des coefficients de Bézout dans les variables dont les noms sont donnés entre apostrophes.

Nous allons utiliser ces calculs pour résoudre différentes équations.

– Equation $ax+by=c$

Exercice 1 : écrire une fonction qui prenne en entrée trois entiers a , b et c et rende les entiers x et y tels que $ax+by=c$.

```
[ > sol1(456,123,21);  
                                     {x = 119 + 41 k, y = -441 - 152 k}
```

Comparez avec les solutions données par `isolve` (?isolve). La façon dont les résultats sont rendus permet de les utiliser dans la fonction `subs` (?subs).

– Solution

On va utiliser la fonction `igcdex` de Maple qui calcule le pgcd et une paire de coefficients de Bézout.

Posons $d = \text{pgcd}(a, b)$ alors :

- soit d ne divise pas c et il n'y a pas de solution,
- soit d divise c , posons alors $a1, b1, c1$ les quotients de a, b, c par d .

On a $a d = a u + b v$ et donc $d c1 = a u c1 + b v c1$, or $d c1 = c$, c'est une solution particulière.

La famille de solutions est donnée par $d c1 = a (u c1 + k b1) + b (v c1 - k a1)$.

```
[ > sol1:=proc(a,b,c)  
    local d,c1,u,v;  
    d:=igcdex(a,b,'u','v');  
    if irem(c,d)=0  
    then  
        c1:=iquo(c,d); { 'x'=u*c1+'k'*iquo(b,d), 'y'=v*c1-'k'*iquo(a,  
        d) }  
    else {} fi  
    end:  
[ > sols:=sol1(456,123,21);
```

```
sols := {x = 119 + 41 k, y = -441 - 152 k}
```

Remarquez qu'on obtient une vérification en substituant les solutions trouvées dans l'expression $456x + 123y$:

```
> subs(sols, 456*x+123*y);  
21  
> solve(456*x+123*y=21);  
{x = 37 + 41 _NI, y = -137 - 152 _NI}
```

et que les solutions particulières ne sont pas les mêmes dans notre fonction et dans solve. Celles de solve sont plus simples. Pour faire correspondre les deux familles, il faut faire $k = _NI - 2$.

D'autres tests :

```
> sol1(456, 123, 8); solve(456*x+123*y=8);  
{ }
```

Il n'y a pas de solution. Notre fonction rend {}, solve rien ce qui se dit NULL en Maple. Que préférez-vous ?

```
> sol1(456, 121, 2); solve(456*x+121*y=2);  
{x = -26 + 121 k, y = 98 - 456 k}  
{x = 95 + 121 _NI, y = -358 - 456 _NI}
```

Comme précédemment, quelle correspondance y-a-t'il entre k et $_NI$?

Exercice 2 : écrire une fonction qui prenne en entrée trois entiers a , b et m et rende les x tels que $ax \equiv b \pmod{m}$.

```
> sol2(123, 21, 456);  
{15, 167, 319}
```

Comparez avec les solutions données par msolve (?msolve).

Solution

C'est très proche de l'exercice précédent : on va résoudre l'équation $ax = b + my$ en nombres entiers, mais on ne s'intéresse qu'aux valeurs distinctes de $x \pmod{m}$. En remarquant que $ub + dm = ub \pmod{m}$, on voit que $\text{seq}((u*b1 + k*m1) \pmod{m}, k=0..d-1)$ va nous donner ces valeurs.

```
> sol2:=proc(a,b,m)  
  local d,b1,m1,u,v,k;  
  d:=igcdex(a,m,'u','v');m1:=iquo(m,d);  
  if irem(b,d)=0  
  then b1:=iquo(b,d);{seq((u*b1+k*m1) mod m,k=0..d-1)} else  
  {} fi  
end;
```

Quelques tests :

```
> sol2(123, 21, 456); msolve(123*x=21, 456);  
{15, 167, 319}  
{x = 319}, {x = 167}, {x = 15}  
> sol2(123, 8, 456); msolve(123*x=8, 456);  
{ }
```

msolve ne rend rien de visible, c'est à dire l'objet NULL.

```
[ > sol2(121, 2, 456);msolve(121*x=2, 456);
      {98}
      {x=98}
```

Exercice 3 : écrire une fonction qui prenne en entrée deux entiers a et m et rende l'inverse de a (mod m) s'il existe (pourquoi est-il alors unique ?) et FAIL sinon. Comparez avec les solutions données par mod (?mod, ?FAIL). Comprenez-vous l'intérêt de FAIL ?

— **Solution**

C'est très proche de l'exercice précédent : on va résoudre l'équation $ax = 1 \pmod{m}$. Il n'y a de solution que si $\text{pgcd}(a, m) = 1$ et elle est alors unique, on la tire de la relation de Bézout $au + mv = 1$ et donc $au = 1 \pmod{m}$.

```
[ > inverse:=proc(a,m)
      local d,u,v;
      d:=igcdex(a,m,'u','v');
      if d=1 then u else FAIL fi
      end;
      inverse := proc(a, m)
      local d, u, v;
      d := igcdex(a, m, 'u', 'v'); if d = 1 then u else FAIL end if
      end proc
```

Les tests :

```
[ > inverse(121, 456);1/121 mod 456;
      49
      49
[ > inverse(123, 456);1/123 mod 456;
      FAIL
[ Error, the modular inverse does not exist
```

L'intérêt de FAIL est de ne pas interrompre la suite du programme, alors qu'une erreur interrompt tout :

```
[ > test:=proc() local a;
      a:=inverse(123, 456);print('a='.a);end;
      test := proc() local a; a := inverse(123, 456); print('a='||a) end proc
[ > test();
      a=FAIL
```

L'affectation a été faite puis le print .

```
[ > test:=proc() local a; a:=1/123 mod 456;print('a='.a);end;
      test := proc() local a; a := 1 / 123 mod 456; print('a='||a) end proc
[ > test();
      Error, (in test) the modular inverse does not exist
```

Rien n'est fait : le déclenchement d'une erreur ramène au niveau global de Maple.

Exercice 4 : dans les livres de casse-têtes mathématiques on peut trouver ce genre de problèmes : 5 marins se retrouvent sur une île déserte. Ils ramassent le maximum de noix de coco. La nuit venue le premier décide de dissimuler sa part : il divise le tas en 5 parts égales, il reste une noix de coco qu'il jette, il cache sa part. Ainsi fait le second un peu plus tard, puis le troisième, le quatrième et le cinquième. Le lendemain ils se partagent le tas restant en 5 parts égales. Combien y'avait-il de noix de coco au départ ?

— **Solution**

Ecrivons l'action de chaque marin comme une fonction : il part d'un tas de x noix, en jette une, puis cache le cinquième du tas restant. Il reste donc un tas de $\frac{4(x-1)}{5}$ noix.

```
[ > f:=x->4/5*(x-1);
```

$$f := x \rightarrow \frac{4}{5}x - \frac{4}{5}$$

Le lendemain, après que chacun des cinq marins ait fait cette action, il reste un nombre de noix multiple de 5. L'équation est donc :

```
[ > equa:=f(f(f(f(f(x)))))=5*y;
```

$$equa := \frac{1024}{3125}x - \frac{8404}{3125} = 5y$$

dont on cherche les solutions en nombres entiers :

```
[ > solve(equa);
```

$$\{y = 204 + 1024_NI, x = 3121 + 15625_NI\}$$

Il y avait donc 3121 noix au départ (ou 3121+15625, 3121+2*15625, ... mais ça fait beaucoup).

— Théorème des restes chinois

Exercice 5 : écrire une fonction qui prenne en entrée quatre entiers a , m , b et n et rende, si m et n sont premiers entre eux, l'unique $x \pmod{mn}$ tel que $x \equiv a \pmod{m}$ et $x \equiv b \pmod{n}$, FAIL sinon.

— Solution

Si m , n sont premiers entre eux on a alors des coefficients de Bézout u , v tels que $mu + nv = 1$ et donc $mu = 1 \pmod{n}$ et $nv = 1 \pmod{m}$. On a donc $anv + bmu = a \pmod{m}$ et $anv + bmu = b \pmod{n}$.

```
[ > resteschinois:=proc(a,m,b,n)
  local d,u,v;
  d:=igcdex(m,n,'u','v');
  if d=1 then (a*n*v+b*m*u) mod (m*n) else FAIL fi
end;
```

```
resteschinois := proc(a, m, b, n)
```

```
local d, u, v;
```

```
  d := igcdex(m, n, 'u', 'v');
```

```
  if d = 1 then (a*n*v + b*m*u) mod m*n else FAIL end if
```

```
end proc
```

Les tests :

```
[ > resteschinois(3,17,4,11);
```

37

en effet

```
[ > 37 mod 17; 37 mod 11;
```

3

4

Si m , n ne sont pas premiers entre eux :

```
[ > resteschinois(3,16,4,10);
```

Exercice 6 : écrire une fonction qui prenne en entrée deux listes d'entiers $[a_1 .. a_n]$ et $[m_1 .. m_n]$ et rende, si les m_i sont premiers entre eux deux à deux, l'unique x modulo le produit des m_i tel que $x = a_i \bmod m_i$, FAIL sinon.

— Solution

Attention, il y a dans la solution présentée ici une part de virtuosité technique pour aficionados.

On commence par construire une fonction qui va tester la validité des arguments, c'est à dire :

- la liste $[a_1 .. a_n]$ est formée d'entiers et la liste $[m_1 .. m_n]$ d'entiers positifs non nuls (on utilise le typage des arguments),
- les deux listes ont la même longueur (on utilise la fonction **nops**),
- cette longueur est au moins 1.

Si ces conditions sont remplies on renvoie une liste de couples $[[a_1, m_1] .. [a_n, m_n]]$ (on utilise la fonction **zip**) sinon on déclenche une erreur (on utilise la fonction **ERROR**).

```
> validargs:=proc(la::list(integer),lm::list(posint))
  local n;
  n:=nops(la);
  if n<>nops(lm) then ERROR('longueurs différentes',la,lm)
  elif n=0 then ERROR('liste vide')
  else zip((x,y)->[x,y],la,lm) fi
end;

validargs := proc(la::list(integer), lm::list(posint))
local n;
  n := nops(la);
  if n ≠ nops(lm) then ERROR('longueurs différentes' , la, lm)
  elif n = 0 then ERROR('liste vide')
  else zip((x, y) → [x, y], la, lm)
  end if
end proc
```

Quelques tests :

```
> validargs([1,2,x],[1,2]);
Error, validargs expects its 1st argument, la, to be of type
list(integer), but received [1, 2, x]
> validargs([1,2,3.0],[1,2]);
Error, validargs expects its 1st argument, la, to be of type
list(integer), but received [1, 2, 3.0]
```

Erreur de type dans la liste la.

```
> validargs([1,2,3],[1,0,2]);
Error, validargs expects its 2nd argument, lm, to be of type
list(posint), but received [1, 0, 2]
```

Erreur de type dans la liste lm.

```
> validargs([1,2,3],[1,2]);
Error, (in validargs) longueurs différentes, [1, 2, 3], [1, 2]
```

```
> validargs([],[]);
```

└ Error, (in validargs) liste vide

Et quand même un exemple qui marche :

```
> validargs([1,2,3],[4,5,6]);  
[[1, 4], [2, 5], [3, 6]]
```

On modifie la solution de l'exercice 5 pour s'adapter au format des arguments : la fonction **chinois2** reçoit deux arguments $am1$ qui est le couple $[a_1, m_1]$ et $am2$ qui est le couple $[a_2, m_2]$ et rend la solution $[x, m_1 m_2]$ telle que $x = a_1 \pmod{m_1}$ et $x = a_2 \pmod{m_2}$, ou bien $[FAIL, m_1 m_2]$ si celle-ci n'existe pas.

```
> chinois2:=proc(am1, am2)  
  local d, u, v;  
  d:=igcdex(am1[2], am2[2], 'u', 'v');  
  if d=1 then [(am1[1]*am2[2]*v+am2[1]*am1[2]*u) mod  
    (am1[2]*am2[2]), am1[2]*am2[2]]  
  else [FAIL, am1[2]*am2[2]] fi  
end;  
  
chinois2 := proc(am1, am2)  
  local d, u, v;  
  d := igcdex(am1[2], am2[2], 'u', 'v');  
  if d = 1 then [(am1[1]*am2[2]*v + am2[1]*am1[2]*u) mod am1[2]*am2[2],  
    am1[2]*am2[2]]  
  else [FAIL, am1[2]*am2[2]]  
  end if  
end proc
```

On construit alors une solution itérative (on pourrait chercher une solution récursive) :

```
> chinoisn:=proc(la, lm)  
  local lam, am1, am2;  
  lam:=validargs(la, lm);  
  am1:= [0, 1];  
  for am2 in lam do  
    am1:=chinois2(am1, am2);  
    if am1[1]=FAIL then RETURN(FAIL) fi  
  od;  
  am1[1]  
end;  
  
chinoisn := proc(la, lm)  
  local lam, am1, am2;  
  lam := validargs(la, lm);  
  am1 := [0, 1];  
  for am2 in lam do  
    am1 := chinois2(am1, am2); if am1[1] = FAIL then RETURN(FAIL) end if  
  end do;  
  am1[1]  
end proc
```

Une batterie de tests :

on vérifie d'abord que validargs fait toujours son travail.

```
[ > chinoisn([1,2,x],[1,2]);
Error, (in chinoisn) validargs expects its 1st argument, la, to be of
type list(integer), but received [1, 2, x]
[ > chinoisn([1,2,3.0],[1,2]);
Error, (in chinoisn) validargs expects its 1st argument, la, to be of
type list(integer), but received [1, 2, 3.0]
[ > chinoisn([1,2,3],[1,0,2]);
Error, (in chinoisn) validargs expects its 2nd argument, lm, to be of
type list(posint), but received [1, 0, 2]
[ > chinoisn([1,2,3],[1,2]);
Error, (in validargs) longueurs différentes, [1, 2, 3], [1, 2]
[ > chinoisn([],[]);
Error, (in validargs) liste vide
```

le cas limite d'une seule congruence,

```
[ > chinoisn([3],[5]);
3
[ > chinoisn([5],[3]);
2
```

les exemples de l'exercice 5,

```
[ > chinoisn([3,4],[17,11]);
37
[ > chinoisn([3,4],[16,10]);
FAIL
```

enfin quelques exemples plus sérieux.

```
[ > chinoisn([1,3,6,4],[17,11,8,9]);
3622
[ > 3622 mod 17,3622 mod 11,3622 mod 8,3622 mod 9;17*11*8*9;
1, 3, 6, 4
13464
[ > chinoisn([1,3,6,4],[17,22,8,9]);
FAIL
```