

CORRIGE du TP 3

Acceleration de convergence

B. Landreau

[> restart;

– Acceleration de Richardson

– Construction des procedures

Dans un premier temps on peut ecrire une procedure **richardson_1(u,n)** qui cree un tableau V de 0 a n ,
le remplit avec u_{2^i} pour i de 0 a n (l'affiche eventuellement) et le retourne.

```
[ > richardson_1:=proc(u,n)
    local i,j,V;
    V:=array(0..n);
    # initialisation
    for j from 0 to n do
        V[j]:=evalf(u(2^j));
    od;
    # lprint(seq(V[j],j=0..n));
    RETURN(V);
end;
```

[On l'essaie.

```
[ > u:=n->sum(1/'k', 'k'=1..n) - ln(n);
```

$$u := n \rightarrow \left(\sum_{k=1}^n \frac{1}{k} \right) - \ln(n)$$

```
[ > R:=richardson_1(u,10); lprint(seq(R[j],j=0..10));
```

$R := V$

```
1., .806852819440054690582767878542, .69703897221344271449886909041, .63841
560117730692889116077849, .60814027098921199132430050740, .5927592926367935
5567367757296, .58500782034609716950249050465, .581116828669555210875022553
21, .57916751833771789353911671887, .57819190951021532426313368335, .577703
86667868073920475162294
```

[On cree maintenant une seconde procedure plus complete qui va appeler **richardson_1** puis qui realise les iterations.

On peut utiliser un seul vecteur V car si on reflechit un peu, on voit que l'on peut tout a fait faire les calculs sur place.

En effet, a l'etape i , si on commence par la gauche, on peut ecraser la valeur ancienne $V[j]$ (ligne $i-1$) en mettant a la place la combinaison lineaire de $V[j+1]$ et $V[j]$

qui correspond a la nouvelle valeur de $V[j]$ pour la ligne i et ceci pour j de 0 a $n-i$.

[On peut tres bien lui faire retourner la derniere valeur calculee i.e. $V[0]$.

```
[ > richardson_2:=proc(u,n)
    local i,j,V;
```

```

V:=richardson_1(u,n);
# boucle principale
for i to n do
  # boucle sur les elements de la ligne
  for j from 0 to n-i do
    V[j]:=(2^i*V[j+1]-V[j])/(2^i-1);
  od;
  lprint(seq(V[j],j=0..n-i));
od;
#printf("\nV[0]=%1.20f",V[0]);
RETURN(V[0]);
end:

```

```

> a:=richardson_2(u,6);evalf(a-gamma,30);
      a := .5772156700
      .5098467139393487909918 10-8

```

[Pas mal!

On peut aussi (si on n'est pas sur de soi) decider de calculer la ligne i (a partir du tableau V de la ligne $i-1$)

dans un autre tableau W puis copier W dans V avant de passer a la ligne $i+1$.

Attention pour copier les structures telles que les tableaux, vecteurs, matrices, il faut utiliser la commande **copy** (exemple $V:=\text{copy}(W)$)

En effet, la simple instruction $V:=W$ copie en fait l'adresse memoire de V sur celle de W et du coup les variables V et W pointent des lors au meme endroit memoire.

Cela peut creer pas mal de confusion ensuite. A eviter.....

Voici une version qui utilise 2 tableaux.

```

> richardson_2_bis:=proc(u,n)
  local i,j,V,W;
  Digits:=15;
  V:=richardson_1(u,n);
  W:=array(0..n);
  # boucle principale
  for i to n do
    # boucle sur les elements de la ligne
    for j from 0 to n-i do
      W[j]:=(2^i*V[j+1]-V[j])/(2^i-1);
    od;
    V:=copy(W);
    lprint(seq(V[j],j=0..n-i));
  od;
  RETURN(V[0]);
end:

```

```

> a:=richardson_2_bis(u,6);evalf(a-gamma,30);
.61370563888011, .587225124986825, .57979223014116, .57786494080112, .57737
831428437, .57725634805541

```

```
.578398287022397, .577314598525940, .577222511021107, .577216105445453, .57
7215692645757
.577159785883589, .577209355663274, .577215190363216, .577215633674373
.577212660315253, .577215579343213, .577215663228450
.577215673505403, .577215665934426
.577215665814252
```

$a := .577215665814252$

$.912719139393487909918 \cdot 10^{-9}$

Le nec plus ultra

On construit le tableau en diagonale
et on decide de s'arreter lorsque la difference de 2 termes successifs de la premiere colonne
est inferieure a ε .

```
> richardson_3:=proc(u,eps)
  local i,j,n,V,erreur;
  # on ajuste la precision en fonction de eps
  Digits:=floor(evalf(-2*ln(eps)/ln(10)));
  printf("Digits = %d\n",Digits);
  n:=100;
  V:=array(0..n);
  # initialisation
  V[0]:=evalf(u(1));
  i:=1;
  erreur:=10; # pour que l'on soit sur de rentrer dans la
  boucle
  while (erreur>eps) and (i<=100) do
    erreur:=V[0];
    V[i]:=evalf(u(2^i));
    for j from i-1 to 0 by -1 do
      V[j]:=(2^(i-j)*V[j+1]-V[j])/(2^(i-j)-1);
    od;
    #lprint(seq(V[j],j=0..i));
    erreur:=abs(erreur-V[0]);
    i:=i+1;
  od;
  printf("\nnombre d'iterations faites : %d",i-1);
  RETURN(V[0]);
end:
```

```
> a:=richardson_3(u,1e-13);evalf(a-gamma,30);
Digits = 26
```

```
nombre d'iterations faites : 10
```

$a := 2.7182818284590452290828475$

$2.14106616355751236847633540992$

>

Applications : Accelerons, Accelerons!

— Constante d'Euler

[C'est ce qu'on vient de faire pour tester nos procedures mais poussons le bouchon plus loin.

[> u:=n->sum(1/'k', 'k'=1..n)-ln(n);

$$u := n \rightarrow \left(\sum_{k=1}^n \frac{1}{k} \right) - \ln(n)$$

[> Digits:=20;a:=richardson_2(u,6);evalf(gamma-a,20);
 >

Digits := 20

.6137056388801093812, .58722512498683073842, .5797922301411711431, .5778
 649408011170538, .5773783142843751201, .5772563480554007833
 .57839828702240452417, .57731459852595127800, .57722251102109902403, .57
 721610544546114220, .57721569264574267103
 .57715978588360081426, .57720935566326298774, .57721519036322715909, .57
 721563367435431800
 .57721266031524046597, .57721557934322477051, .57721566322842946193
 .57721567350541781258, .57721566593440380681
 .57721566581422898132

a := .57721566581422898132

-.91269612071 10⁻⁹

[Avec 6 accelerations, on obtient 8 chiffres significatifs.

En fait, si l'on sait que dans le developpement asymptotique, apres le terme en $\frac{1}{n}$, il n'y a que des termes avec des exposants pairs,

on peut (doit) modifier la procedure **richardson_2** en remplaçant 2^i par $4^{(i-1)}$ des la deuxieme acceleration.

[> richardson_2_gamma:=proc(u,n)
 local i,j,V;
 V:=richardson_1(u,n);
 # boucle principale
 # le cas i=1
 for j from 0 to n-1 do
 V[j]:=(2*V[j+1]-V[j])/(2-1);
 od;
 # le cas i>1
 for i from 2 to n do
 # boucle sur les elements de la ligne
 for j from 0 to n-i do
 V[j]:=(4^(i-1)*V[j+1]-V[j])/(4^(i-1)-1);
 od;
 #lprint(seq(V[j],j=0..n-i));
 od;
 #printf("\nV[0]=%1.20f",V[0]);
 RETURN(V[0]);
end:

[Voyons si c'est mieux.

[> a:=richardson_2_gamma(u,6);evalf(gamma-a,20);

a := .57721566490388239757500280041595770384517608840196

-234953697 10⁻¹¹

On obtient cette fois 11 chiffres significatifs avec 6 accelerations.

```
> Digits:=50:
essai:=proc(n,m)
  local i,a;
  for i from n to m do
    a:=richardson_2_gamma(u,i):
    printf("\n i=%d erreur=%g",i,evalf(a-gamma,50));
  od;
end:
> essai(5,12);
```

```
i=5 erreur=1.353120e-09
i=6 erreur=2.349537e-12
i=7 erreur=1.445690e-15
i=8 erreur=3.008859e-19
i=9 erreur=2.044599e-23
i=10 erreur=4.411386e-28
i=11 erreur=2.954325e-33
i=12 erreur=6.026802e-39
```

On peut se faire un petit tableau

	A	B	C	D
1	"nombre d'iterations"	5	6	7

Combien d'iterations pour avoir 20 chiffres significatifs ?

Par tatonnement, on trouve **9**.

```
> Digits:=30;a:=richardson_2_gamma(u,9);evalf(gamma-a,30);
      Digits := 30
      a := .577215664901532860606532536092
      -20446010 10-22
```

>

Pi

On utilise la suite $n \sin\left(\frac{\pi}{n}\right)$

En fait le calcul de la suite $2^n \sin\left(\frac{\pi}{2^n}\right)$ qui sert pour la preliere ligne du tableau de

Richarson

se fait a part par recurrence.

```
> w:=proc(n)
  local a,b,i;
  if n=0 then RETURN(0);fi;
  if n=1 then RETURN(2);fi;
  a:=2;b:=0;
  for i from 2 to n
  do b:=sqrt((1+b)/2); a:=a/b; od;
  RETURN (evalf(a));
end:
```

```
> seq(w(i), i=0..4);
0, 2, 2.82842712474619009760337744842, 3.06146745892071817382767987225,
3.12144515225805228557255789563
```

```
>
```

```
2.82842712474619009760337744842
```

Il faut alors localement modifier la procedure **richardson_1** :

on remplace $u(2^i)$ par $u(i)$ sachant que l'on va l'appliquer directement a la suite w.

```
> richardson_1:=proc(u,n)
  local i,j,V;
  V:=array(0..n);
  # initialisation
  for j from 0 to n do
    V[j]:=evalf(u(j));
  od;
  lprint(seq(V[j], j=0..n));
  RETURN(V);
end;
```

```
>
```

```
> Digits:=30;a:=richardson_2(w,6);evalf(Pi-a,30);
```

```
Digits := 30
```

```
0., 2., 2.82842712474619009760337744842, 3.06146745892071817382767987225
, 3.12144515225805228557255789563, 3.13654849054593926381425804443, 3.14
033115695475291231711852432
4., 3.65685424949238019520675489684, 3.29450779309524625005198229608, 3.
18142284559538639731743591901, 3.15165182883382624205595819323, 3.144113
82336356656081997900421
3.54247233265650692694233986247, 3.17372564096286826833372476249, 3.1437
2786309543311307258712664, 3.14172815657997285696879895130, 3.1416011548
7348000040798594119
3.12104754214949131710392260534, 3.13944246625722809089242460723, 3.1414
4248422062139181111492626, 3.14158301177255244947072693974
3.14066879453107720914499140736, 3.14157581875151427853902761420, 3.1415
9238027601451998136774063
3.14160507759733482916464168537, 3.14159291451874033422144322857
3.14159272145400073906361468164
```

```
a := 3.14159272145400073906361468164
```

```
-.6786420750060097129836 10-7
```

Pas terrible! On obtient seulement 6 decimales exactes avec 6 accelerations.

En fait, si on calcul le developpement asymptotique de $n \sin\left(\frac{\pi}{n}\right)$ on s'aperçoit qu'il n'y a que des termes

avec des exposants pairs. Il faut en tenir compte et remplacer dans les formules

d'acceleration 2^i par 4^i .

Cela conduit a une autre procedure : **richardson_2_pi**.

```
> richardson_2_pi:=proc(u,n)
  local i,j,V;
  V:=richardson_1(u,n);
  # boucle principale
  for i to n do
    # boucle sur les elements de la ligne
    for j from 0 to n-i do
```


Il faut et il suffit donc de 7 accelerations pour avoir nos 20 chiffres significatifs..

$$e$$

Le développement asymptotique est de la forme qui va bien pour pouvoir appliquer Richardson.

$$u := n \rightarrow \left(1 + \frac{1}{n}\right)^n$$
$$Digits := 30$$
$$a := 2.71828171504738678902277205395$$

Utilisons notre `procedure richardson_3` pour degrossir.

$$Digits := 30$$
$$a := 2.7182818284590452353602867312040085758$$
$$.74015 \cdot 10^{-24}$$

2.71802983463829927301087906693, 2.71826670291244640244612721187, 2.7182

```

8113434204245816566950911, 2.71828180164089392473847907405, 2.7182818275
2147899953257118395, 2.71828182842800301068359102698, 2.7182818284580462
7278664173111, 2.71828182845901355260817241721
2.71827434382451566468597392622, 2.71828159987202942770500958321, 2.7182
8182316666332688598906002, 2.71828182835633658259044512298, 2.7182818284
5724572072072069934, 2.71828182845901541027383691511, 2.7182818284590447
5518306050386
2.71828171504738678902277205395, 2.71828182671102259512695698822, 2.7182
8182843871234855400791764, 2.71828182845884745307199491484, 2.7182818284
5904350058420383918, 2.71828182845904522097527040210
2.71828182759026382194667497983, 2.71828182845231620488020516906, 2.7182
8182845900599720205780458, 2.71828182845904504426540233465, 2.7182818284
5904523452165675299
2.71828182845569680246033666000, 2.71828182845903223168175134433, 2.7182
8182845904519739114094065, 2.71828182845904523526775971149
2.71828182845903875894050166661, 2.71828182845904522276434913947, 2.7182
8182845904523534188225311
2.71828182845904522908284752410, 2.71828182845904523535417700688
2.71828182845904523535724067541

```

$a := 2.71828182845904523535724067541$

$.304679594 \cdot 10^{-20}$

En fait, **11** accelerations c'est ce qu'il faut pour avoir nos 20 chiffres significatifs.

– Acceleration de la convergence d'une serie alternee

On considere la serie

$$\sum_{k=1}^{\infty} \frac{(-1)^k}{k},$$

et on veut en tirer une approximation de $\ln(2)$.

Soit S la somme de cette serie, on $S = -\ln(2)$.

> restart;

a)

En utilisant la suite des sommes partielles

$$S_n = \sum_{k=1}^n \frac{(-1)^k}{k}$$

calculer une approximation de $\ln(2)$ a $10^{(-4)}$ pres.

> s:=n->evalf(sum((-1)^k/'k', 'k'=1..n));

$$s := n \rightarrow \text{evalf}\left(\sum_{k'=1}^n \frac{(-1)^{k'}}{k'}\right)$$

On sait que le terme reste R_n pour une serie alternee est majore en valeur aboslue

par lavaleur absolue du premier terme soit ici $\frac{1}{n+1}$.

Donc en prenant $n = 10^4$, on aura $\ln(2)$ a $10^{(-4)}$ pres.

```

> a:=-s(10^4);evalf(ln(2)-a);
a := .6930971831
      .0000499975

```

[Ca marche mais c'est long!

b)

Soient les suites $M_1(n) = \frac{S_n + S_{n+1}}{2}$ et pour k supérieur

ou égal a 2 : $M_k(n) = \frac{M_{k-1}(n) + M_{k-1}(n+1)}{2}$.

Construire une procédure Maple **altern(n)** permettant de calculer

de manière triangulaire, en première ligne $S_l, 1 \leq l \leq n$

puis les différentes moyennes $M_k(n)$.

Calculer $M_8(1)$ et comparer à $\ln(2)$.

En fait plutôt que de faire appel à chaque fois à la suite S_n ,

qui recalcule la somme de 1 à n ,

pour remplir la première ligne, il vaut mieux le faire par récurrence.

```

> altern:=proc(n)
  local i,k,x,v;
  v:=array(1..n);
  #première ligne
  x:=0.;
  for k from 1 to n do
    x:=x+(-1)^k/k;
    v[k]:=x;
  od;
  lprint(seq(evalf(v[k]),k=1..n));
  # lignes suivantes, (n-1) moyennes
  for i from 1 to n-1 do
    for k to n-i do
      v[k]:=(v[k+1]+v[k])/2
    od;
    lprint(seq(evalf(v[k]),k=1..n-i));
  od;
  printf("M_%d(%d)=%.15f",i-1,1,evalf(v[1]));
  RETURN(evalf(v[1]));
end:

```

[Pour pouvoir faire 8 moyennes, il faut 9 valeurs au départ.

```

> Digits:=20;a:=altern(9);evalf(a+ln(2));

```

Digits := 20

$$\frac{3}{4} \left| \frac{1}{(n+1)(n+2)(n+3)(n+4)} \right|$$

$$\frac{3}{2} \left| \frac{1}{(n+1)(n+2)(n+3)(n+4)(n+5)} \right|$$

$$\frac{45}{4} \left| \frac{1}{(n+1)(n+2)(n+3)(n+4)(n+5)(n+6)(n+7)} \right|$$

On observe en fait que le terme

$r_k(n)$ est equivalent a $C_k \frac{1}{n^{(k+1)}}$.

La convergence est de type lineaire mais de plus en plus rapide.

Verifions si notre majoration est bonne, pour $n=4$ et $k=6$ on a :

> `reste:=evalf(r(6,4));`

reste := .6764069264 10⁻⁵

> `Digits:=10;a:=altern(10);`

Digits := 10

-1., -.5000000000, -.8333333333, -.5833333333, -.7833333333, -.6166666666,
-.7595238095, -.6345238095, -.7456349206, -.6456349206
-.7500000000, -.6666666666, -.7083333332, -.6833333332, -.6999999999, -.688
0952381, -.6970238096, -.6900793651, -.6956349206
-.7083333333, -.6874999999, -.6958333332, -.6916666666, -.6940476190, -.692
5595238, -.6935515874, -.6928571429
-.6979166666, -.6916666666, -.6937499999, -.6928571428, -.6933035714, -.693
0555556, -.6932043651
-.6947916666, -.6927083333, -.6933035714, -.6930803571, -.6931795635, -.693
1299604
-.6937499999, -.6930059523, -.6931919643, -.6931299604, -.6931547620
-.6933779762, -.6930989584, -.6931609624, -.6931423612
-.6932384673, -.6931299604, -.6931516618
-.6931842138, -.6931408111
-.6931625125
M_9(1)=-.693162512500000

a := -.6931625125

1

La valeur $M_6(4)$ est d'apres le tableau precedent -.6931423612.

> `a:=-.6931423612;b:=evalf(abs(a+ln(2)));`
`reste:=evalf(r(6,4));`
`evalb(b<reste);`

a := -.6931423612

b := .48194 10⁻⁵

reste := .6764069264 10⁻⁵

true

C'est OK

On peut regarder la convergence sur la premiere colonne.

> `r(k,1);`

$$\left| \frac{1}{2^k (k+1)(k+2)} \right|$$

Cette fois, on constate que la convergence est essentiellement en $\frac{1}{2^k}$, toujours lineaire

mais avec un coefficient d'erreur asymptotique $\frac{1}{2}$. C'est beaucoup rapide que n'importe quelle suite ligne $M_k(n)$.

>

c) Procédé d'Aitken

On peut montrer que

le reste $S - S_n$ est équivalent en valeur absolue à $\frac{1}{2n}$ (par comparaison à une intégrale).

La suite S_n satisfait-elle les

hypothèses du théorème test pour le procédé d'Aitken?

Tester ce procédé sur cette suite.

A priori on a une convergence du type linéaire mais avec un facteur

$k=-1$ donc on **n'est pas** dans les conditions du théorème vu en cours.

Ceci dit, si on regarde la preuve du **procédé d'Aitken**, c'est le cas $k=1$ qui pose problème mais pas celui où $k=-1$. Donc ça devrait marcher.

Essayons.

Pour une série $S_n = \sum_{k=1}^n u_k$ on doit remplacer

S_n par la suite

$$T_n = S_n - \frac{u_{n+1}^2}{u_{n+2} - u_{n+1}}.$$

Testons d'abord l'équivalence du reste.

Calculons $S_{10000} - S_{1000}$ qui doit être grosso modo égal au reste de rang 1000 et comparons le avec $1/2000$.

```
> evalf((s(10^4)-s(10^3))*2000);
-.899505
```

Comme on connaît la limite, on peut regarder aussi directement $S - S_n$.

```
> evalf(abs(s(10^2)+ln(2))*2*10^2);
.9950002
.999500
.99995
```

Ça a l'air de marcher.

Attaquons le Δ^2 d'Aitken.

```
> aitken:=proc(n)
  local k,x,y,u,v,w;
  x:=0.;y:=0.;
  for k to n do
```

```

    u:=(-1)^k/k;
    v:=(-1)^(k+1)/(k+1);
    w:=(-1)^(k+2)/(k+2);
    x:=x+u;# c'est la somme s_k
    y:=x-v^2/(w-v);# c'est la somme t_k
    print(evalf(x),evalf(y));
  od;
  RETURN(evalf(y));
end:
> a:=aitken(10);evalf(a+ln(2));

-1., -.7000000000
-.5000000000, -.6904761905
-.8333333333, -.6944444444
-.5833333333, -.6924242424
-.7833333333, -.6935897436
-.6166666666, -.6928571428
-.7595238095, -.6933473389
-.6345238095, -.6930033417
-.7456349206, -.6932539682
-.6456349206, -.6930657506
a := -.6930657506
.0000814300

```

Effectivement, ça accélère, la suite d'Aitken est la suite de la seconde colonne.

Avec 10 termes de la suite d'Aitken, on a $\ln(2)$ à $10^{(-4)}$ près.

[>

[>

d) Calcul de Pi

[On adapte à Pi la procédure **altern** précédente.

[>

```

> altern_pi:=proc(n)
  local i,k,x,v;
  v:=array(0..n);
  #premiere ligne
  x:=0.;
  for k from 0 to n do
    x:=x+(-1)^k/(2*k+1);
    v[k]:=x;
  od;
  lprint(seq(evalf(v[k]),k=1..n));
  # lignes suivantes, (n-1) moyennes
  for i from 1 to n-1 do
    for k to n-i do
      v[k]:=(v[k+1]+v[k])/2
    od
  od
end:

```

```

        od;
        lprint(seq(evalf(v[k]),k=1..n-i));
    od;
    printf("M_%d(%d)=%.15f",i-1,1,evalf(v[1]));
    RETURN(evalf(4*v[1]));
end:
> altern_pi(10);
.6666666667, .8666666667, .7238095238, .8349206349, .7440115440, .820934620
9, .7542679542, .8130914836, .7604599046, .8080789522
.7666666668, .7952380953, .7793650793, .7894660894, .7824730824, .787601287
5, .7836797189, .7867756941, .7842694284
.7809523810, .7873015872, .7844155843, .7859695859, .7850371850, .785640503
2, .7852277064, .7855225612
.7841269841, .7858585858, .7851925852, .7855033855, .7853388441, .785434104
8, .7853751338
.7849927849, .7855255855, .7853479854, .7854211148, .7853864744, .785404619
3
.7852591852, .7854367855, .7853845501, .7854037946, .7853955468
.7853479854, .7854106678, .7853941723, .7853996707
.7853793266, .7854024201, .7853969216
.7853908733, .7853996708
.7853952720
M_9(1)=.785395272000000
3.141581088
Ca accelere la convergence mais toute de meme ce n'est certainement pas la bonne serie pour
calculer Pi.
>

```

FIN