Université des Sciences et Technologies de Lille
U.F.R. de Mathématiques Pures et Appliquées
Bât. M2, F-59655 Villeneuve d'Ascq Cedex

Agrégation externe **Année 2003-2004**

Corrigé multidigressif du TP LFGN en Scilab

Les fonctions Scilab `sum` et `cumsum` sont certainement parmi celles qui vous seront les plus utiles. La première retourne la somme des composantes d'un vecteur ou d'une matrice.

```
-->V=[2 5 7];sum(V)
 ans  =

    14.

-->A=[2 3; 4 5]
 A  =

!   2.    3. !
!   4.    5. !
-->sum(A)
 ans  =

    14.
-->sum(A,'col')
 ans  =

!   5. !
!   9. !
 -->sum(A,'row')
 ans  =

!   6.    8. !
```

La fonction `cumsum` appliquée à un vecteur retourne le vecteur des sommes partielles :

$$(x_1, x_2, \ldots, x_n) \mapsto (x_1, x_1 + x_2, \ldots, x_1 + \cdots + x_n).$$

```
-->cumsum(V)
 ans  =

!   2.    7.    14. !
```

```
-->cumsum(A)
 ans  =

!   2.    9.  !
!   6.    14. !
```

Elle fait la même chose pour une matrice, en la parcourant colonne par colonne.

Il est beaucoup plus rapide (à la programmation et à l'exécution) d'utiliser `sum` et `cumsum` que des boucles `for`, chaque fois que l'on a besoin de sommes ou de suite de sommes partielles.

Les fonctions `prod` et `cumprod` sont bâties sur le même modèle en remplaçant l'addition par la multiplication.

```
-->V=1:4
 V  =

!   1.    2.    3.    4. !

-->prod(V)
 ans  =

    24.

-->cumprod(V)
 ans  =

!   1.    2.    6.    24. !
```

Ceci explique pourquoi il n'y a pas de fonction factorielle préprogrammée en Scilab. Pour obtenir $n!$, il suffit de prendre[1] `prod(1 :n)`. Le seul intérêt autre que pégagogique dans l'écriture d'une fonction factorielle est la possibilité de rajouter des contrôles sur le type et la valeur du paramètre d'entrée ainsi que d'éventuels messages d'erreur (voir les fonctions `fact2`, `fact3` et `fact4` dans [1, pp. 42–43]).

Ex 1. *Coefficients du binôme*

On se propose de calculer la liste des C_n^k, $0 \leq k \leq n$ par différentes méthodes.

1) Méthode bête avec les factorielles : on se contente de traduire la formule

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

Pour calculer un C_n^k donné, il suffirait de le coder :

1. Cela marche aussi avec $n = 0$ puisque `1 :0` représente le vecteur ligne des entiers k tels que $1 \leq k \leq 0$, donc le vide et qu'un produit indexé par l'ensemble vide vaut 1 par convention (pour une somme, ce serait 0).

```
prod(1:n)/(prod(1:k)*prod(1:(n-k)))
```

Pour obtenir une fonction qui donne le vecteur des C_n^k, il suffit d'écrire avec son éditeur préféré [2] le code suivant dans un fichier `coefbin.sci` :

```
function [y]=coefbin1(n)
// fournit la liste des coefficients binomiaux C_n^k
// pour k=0 jusqu'a n
for k=0:n,
    y(k+1)=prod(1:n)/(prod(1:k)*prod(1:(n-k)));
end
```

Remarquer le `y(k+1)` au lieu de `y(k)`. Scilab indexe toujours les vecteurs en commençant par l'entier 1, toute tentative d'utiliser un indice 0 provoque un message d'erreur. Cette fonction crée un vecteur y à $n+1$ composantes avec

$$y(1) = C_n^0,\ y(2) = C_n^1, \ldots,\ y(n+1) = C_n^n.$$

Testons cette première fonction en la chargeant d'abord dans l'environnement par le menu `File -> File operations -> Getf` après sélection à la souris du fichier :

```
-->;getf("/home/charles/Enseignement/Agregation/coefbin.sci");

-->coefbin1(5)
 ans  =

!   1.  !
!   5.  !
!   10. !
!   10. !
!   5.  !
!   1.  !
```

C'est plutôt encourageant, mais avouez que vous vous attendiez à un vecteur ligne (inutile de nier, moi aussi) ! L'explication est simple. Au premier tour de la boucle `for`, Scilab ne sait pas ce qu'est `y`. Il crée donc une *matrice*, pour l'instant de taille 1×1. Au tour suivant, il choisit d'agrandir cette matrice *verticalement* en créant une deuxième ligne `y(2)`, etc. Notons au passage que Scilab accepte tout à fait de numéroter avec un seul indice les termes d'une matrice rectangulaire. Il se contente de les balayer de haut en bas, colonne après colonne.

```
-->M=[4 8 12; 3 6 9]
 M  =

!   4.    8.    12. !
```

2. Donc avec Emacs !

```
!   3.    6.    9.  !

-->[M(1) M(2) M(3) M(4) M(5)]
 ans  =

!   4.    3.    8.    6.    12. !
```

Pour obtenir un vecteur ligne avec `coefbin1`, on peut remplacer `y(k+1)` par `y(1,k+1)` pour forcer la création progressive d'un vecteur ligne s'agrandissant vers la droite. Une autre solution consiste à *initialiser* `y` *avant* la boucle `for` en un vecteur ligne de la taille voulue. On dispose pour cela des fonctions `zeros` et `ones` :

```
-->0:5
 ans  =

!   0.    1.    2.    3.    4.    5. !

-->zeros(0:5)
 ans  =

!   0.    0.    0.    0.    0.    0. !

-->ones(0:5)
 ans  =

!   1.    1.    1.    1.    1.    1. !
```

Les affectations `y(k+1)=...` utilisent alors la numérotation à un seul indice de la matrice `y`, conformément à la règle ci-dessus, colonne après colonne. Comme chaque colonne a un seul élément, cela donne bien le résultat escompté.

```
function [y]=coefbin1(n)
// fournit la liste des coefficients binomiaux C_n^k
// pour k=0 jusqu'a n
y=zeros(0:n);
for k=0:n,
    y(k+1)=prod(1:n)/(prod(1:k)*prod(1:(n-k)));
end
```

Relançons :

```
-->;getf("/home/charles/Enseignement/Agregation/coefbin.sci");
Warning :redefining function: coefbin1

-->coefbin1(5)
 ans  =
```

```
!   1.    5.    10.    10.    5.    1. !
```

Dans le code `y(k+1)=prod(1 :n)/(prod(1 :k)*prod(1 :(n-k)))`, les opérations de division `/` et de multiplication `*` sont *matricielles*. En fait elles agissent ici sur des nombres considérés comme matrices de taille 1×1. L'utilisation des opérations ponctuelles `./` et `.*` aurait donné ici le même résultat (testez le). Ces opérations ponctuelles sont utiles pour créer *directement* le vecteur des $(C_n^k; 0 \leq k \leq n)$ sans boucle `for`. Examinons d'abord la division ponctuelle en créant le vecteur des $(n!/k!; 1 \leq k \leq n)$:

```
-->n=5;prod(1:n)
 ans  =

    120.

-->cumprod(1:5)
 ans  =

!   1.    2.    6.    24.    120. !

-->prod(1:5)./cumprod(1:5)
 ans  =

!   120.    60.    20.    5.    1. !
```

Attention, l'utilisation de `/` au lieu de `./` ne déclenche pas ici de message d'erreur, mais effectue une autre opération que celle souhaitée. Pour comprendre, examinez ce qui suit :

```
-->A=prod(1:5)/cumprod(1:5)
 A  =

!   0.0079909 !
!   0.0159819 !
!   0.0479457 !
!   0.1917826 !
!   0.9589132 !

-->A*cumprod(1:5)
 ans  =

!   0.0079909    0.0159819    0.0479457    0.1917826    0.9589132 !
!   0.0159819    0.0319638    0.0958913    0.3835653    1.9178265 !
!   0.0479457    0.0958913    0.2876740    1.1506959    5.7534794 !
!   0.1917826    0.3835653    1.1506959    4.6027835    23.013918 !
!   0.9589132    1.9178265    5.7534794    23.013918    115.06959 !
```

```
-->cumprod(1:5)*A
 ans  =

    120.
```

La multiplication ponctuelle permet la création du vecteur $\big(k!(n-k)!; 0 \leq k \leq n\big)$, mais auparavant, nous avons besoin de savoir « retourner » un vecteur.

```
-->B=cumprod(1:5)
 B  =

!   1.    2.    6.    24.    120. !

-->5:-1:1
 ans  =

!   5.    4.    3.    2.    1. !

-->C=B(5:-1:1)
 C  =

!   120.    24.    6.    2.    1. !
```

D'une façon générale, si V est un vecteur à n composantes, celles-ci sont indexées par Scilab de 1 à n et si I est un vecteur d'indices pris dans $\{1, \ldots, n\}$, $V(I)$ est le vecteur des $(V_i; i \in I)$ en respectant l'ordre d'écriture de I.

```
-->M=[2 34 10 21 17]
 M  =

!   2.    34.    10.    21.    17. !

-->M(5:-1:2)
 ans  =

!   17.    21.    10.    34. !

-->I=[2 2 5 3]
 I  =

!   2.    2.    5.    3. !

-->M(I)
 ans  =
```

```
!   34.    34.    17.    10. !

-->J=[1 1 2 2 3 3 5 ];M(J)
 ans  =

!   2.    2.    34.    34.    10.    10.    17. !

-->J=1:6;M(J)
           !--error    21
invalid index
```

Le lecteur attentif aura flairé un futur bogue dans les tests ci-dessus : on a simplement oublié $k = 0$, car 0! ne peut être généré avec des `cumprod`. Il faut donc rajouter cet élément « à la main » par concaténation.

```
-->B=[1 cumprod(1:5)] //concatenation avec 0!
 B  =

!   1.    1.    2.    6.    24.    120. !

-->C=B(6:-1:1)  // maintenant dimension(B)=6
 C  =

!   120.    24.    6.    2.    1.    1. !

-->D=B.*C      // creation du vecteur des denominateurs k!(n-k)!
 D  =

!   120.    24.    12.    12.    24.    120. !

-->prod(1:5)./D         //liste des C_5^k
 ans  =

!   1.    5.    10.    10.    5.    1. !
```

Attention à bien taper `D=B.*C` et non pas `B*C`. Si cela ne vous paraît pas évident, essayez de taper successivement `B*C` puis `B*C'` et `B'*C`.

Il ne reste plus qu'à ajouter le code suivant dans le fichier[3] `coefbin.sci`.

```
function [y]=coefbin2(n)
```

3. Bien sûr vous avez déjà remarqué que le nom du fichier n'est pas forcément celui de la fonction qu'il contient. Il peut donc en contenir plusieurs, qui sont toutes connues dans l'environnement de travail dès que le fichier `coefbin.sci` est chargé par `getf`.

```
// version avec factorielles, mais sans boucle for
B=[1 cumprod(1:n)]; // ajout de 0!
C=B((n+1):-1:1);    // retournement k! --> (n-k)!
D=B.*C;             // denominateurs k!(n-k)!
y=prod(1:n)./D
```

On teste cette nouvelle version :

```
-->;getf("/home/charles/Enseignement/Agregation/coefbin.sci");

-->coefbin2(6)
 ans  =

!   1.    6.    15.    20.    15.    6.    1. !

-->y=coefbin2(170);max(y)
 ans  =

    9.145E+49

-->y(86)  // autrement dit: C_170^85
 ans  =

    9.145E+49
```

Jusque là tout va bien, mais

```
-->y=coefbin2(171);

-->max(y)
 ans  =

   Nan

-->y(85)
 ans  =

   Inf
```

La fonction se plante[4] à partir de $n = 171$. C'est dû au fait que 171! dépasse le plus grand nombre réel représenté en machine. Vérifions le :

```
-->prod(1:170)
 ans  =
```

4. Noter que la saisie de `y=coefbin2(171) ;` ne déclenche pas de message d'erreur.

```
    7.257+306

-->prod(1:171)
 ans  =

   Inf

-->2^1023
 ans  =

    8.988+307

-->2^1024
 ans  =

   Inf
```

2) On peut améliorer la situation en utilisant les simplifications de l'écriture des C_n^k avec factorielle :

$$C_n^k = \frac{n(n-1)\dots(n-k+1)}{k!}.$$

L'affaire se règle avec le quotient ponctuel de deux `cumprod` :

```
-->cumprod(5:-1:1)
 ans  =

!   5.    20.    60.    120.    120. !

-->cumprod(5:-1:1)./cumprod(1:5)
 ans  =

!   5.    10.    10.    5.    1. !
```

et en traitant à part le cas $k = 0$.

```
function [y]=coefbin3(n)
y(1)=1;
y(2:(n+1))=cumprod(n:-1:1)./cumprod(1:n);
```

Après rechargement par `getf`, voici un petit test

```
-->y=coefbin3(171);y(85:88)
 ans  =

   1.0E+50 *
```

```
!   1.776534    1.8183348    1.8183348    1.776534 !
```

C'est déjà mieux, mais hélas le calcul de $(C_{171}^k, 167 \leq k \leq 171)$ donne :

```
-->y(168:172)
 ans  =

!   34389810.   Inf   Inf   Inf   Nan !
```

alors qu'on devrait trouver le vecteur « symétrique » de $(C_{171}^k, 0 \leq k \leq 4)$:

```
-->y(1:5)
 ans  =

!   1.    171.    14535.    818805.    34389810. !
```

Il n'y a là aucun mystère. Le calcul par cette fonction de C_{171}^{168} (i.e. de `y(169)`) fait intervenir au numérateur `prod(171 :-1 :4)` qui dépasse[5] le plus grand nombre représentable en machine et est donc traité comme $+\infty$ alors que le dénominateur est 168! qui reste calculable. Le quotient est donc $+\infty$ pour la machine. Il en va de même pour C_{171}^{169} et C_{171}^{170}. Enfin pour C_{171}^{171}, Scilab doit calculer 171!/171! qui pour lui est ∞/∞ d'où le résultat `Nan` (Not a number).

```
-->prod(171:-1:5)
 ans  =

    5.171+307

-->prod(171:-1:4)
 ans  =

   Inf
-->prod(171:-1:5)/prod(1:167)
 ans  =

    34389810.
```

3) Méthode utilisant la fonction de répartition de la loi binomiale. Soit S_n une variable aléatoire binomiale de paramètres n et $1/2$ et F_n sa fonction de répartition. On a alors pour $0 \leq k \leq n$,

$$\mathbf{P}(S_n = k) = F_n(k) - F_n(k-1) = C_n^k \Big(\frac{1}{2}\Big)^k \Big(1 - \frac{1}{2}\Big)^{n-k},$$

5. Quand on veut calculer 171! en prenant les facteurs consécutifs dans l'ordre décroissant, on dépasse le « plafond » Scilab avant d'arriver au dernier facteur, contrairement à ce qui se passe avec les facteurs pris dans l'ordre croissant.

d'où la formule

$$C_n^k = 2^n\big(F_n(k) - F_n(k-1)\big).$$

L'intérêt est que la f.d.r. de la loi binomiale existe déjà en Scilab, voir le menu

```
Help -> Cumulative Distribution Functions ; inverses, grand -> cdfbin
```

Cette fonction utilise une routine Fortran basée sur la f.d.r. de la loi Beta incomplète. Il convient de la manipuler avec précautions : elle est considérée comme continue alors qu'elle devrait être en escalier, les valeurs prises aux points de saut (i.e. les entiers k) sont correctes, les autres sont interpolées (cf. documentation sur les f.d.r., à venir).

```
function [y]=coefbin4(n,k)
// calcul de C_n^k a l'aide de la fdr binomiale
if (k>floor(k))|(n>floor(n)) then
    warning('les parametres doivent etre entiers');
elseif k==0 then y=1;
else y=2.^n.*(cdfbin("PQ",k,n,0.5,0.5)-cdfbin("PQ",k-1,n,0.5,0.5));
end
```

On n'a pas fait ici de version directement « vectorisée », car la vectorisation de `cdfbin` est malaisée. Si on veut vectoriser k dans cette fonction (`k=0 :n`), alors il faut vectoriser à la même taille les trois paramètres suivants : `n.*ones(0 :n)`, `0.5.*ones(0 :n)`... À cause des puissances de 2, il semble que l'on puisse aller jusqu'à $n = 1023$. Cependant, la méthode manque de fiabilité pour les valeurs de k supérieures à $n/2$: comparons les valeurs obtenues pour C_{1000}^{350} et C_{1000}^{650}.

```
-->coefbin4(1000,650)
 ans  =

    0.

-->coefbin4(1000,350)
 ans  =

    4.025+279
```

En fait la situation est carrément désastreuse puisque ce phénomène se produit même pour des valeurs petites de n. Par exemple :

```
-->[coefbin4(59,0) coefbin4(59,1) coefbin4(59,58) coefbin4(59,59)]
 ans  =

!   1.    59.    64.    0. !

-->[coefbin4(60,0) coefbin4(60,1) coefbin4(60,59) coefbin4(60,60)]
 ans  =

!   1.    60.    0.    0. !
```

On peut proposer l'explication suivante. Pour k proche de n, $F_n(k) - F_n(k-1)$ est très petit (la loi binomiale ayant une masse très concentrée autour de $n/2$, cf. le théorème de De Moivre Laplace). Si cette valeur tombe en dessous du plus petit réel strictement positif représenté par Scilab, elle est arrondie à 0. La multiplication ultérieure par 2^n n'y peut rien changer. Vérifions

```
-->cdfbin("PQ",59,60,0.5,0.5)-cdfbin("PQ",58,60,0.5,0.5)
 ans  =

    0.
```

Même quand cette valeur n'est pas nulle, sa précision n'est pas toujours suffisante pour résister à la multiplication par 2^n, c'est ce qui explique la valeur 64 trouvée ci-dessus pour C_{59}^{58} calculé par `coefbin4(59,58)`. A propos du plus petit réel strictement positif reconnu par Scilab, il semble que ce soit 2^{-1074}, si l'on en juge par le test suivant :

```
-->2.^(-1074)
 ans  =

    4.941-324

-->2.^(-1075)
 ans  =

    0.

-->0.500000001*2.^(-1074)
 ans  =

    4.941-324
```

4) Méthode la plus économique pour la machine et pour le programmeur : utilisez le triangle de Pascal ! Ainsi aucun calcul ne fera intervenir de nombre supérieur au plus grand des C_n^k à calculer. C'est donc beaucoup plus fiable que les méthodes précédentes. Pour utiliser de façon vectorielle la formule

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1},$$

il suffit d'écrire le vecteur ligne calculé à l'étape $n-1$ et de le recopier en dessous avec décalage d'un cran puis d'additionner. Pour permettre l'addition vectorielle, on bouche les trous créés par le décalage avec un 0 comme dans l'exemple ci-dessous

$$\begin{array}{rcccc} & [1 & 0] & & \\ + & [0 & 1] & & \\ \hline = & [1 & 1] & & \end{array} \qquad \begin{array}{rcccc} & [1 & 1 & 0] & \\ + & [0 & 1 & 1] & \\ \hline = & [1 & 2 & 1] & \end{array} \qquad \begin{array}{rcccc} & [1 & 2 & 1 & 0] \\ + & [0 & 1 & 2 & 1] \\ \hline = & [1 & 3 & 3 & 1] \end{array} \qquad \begin{array}{rccccc} & [1 & 3 & 3 & 1 & 0] \\ + & [0 & 1 & 3 & 3 & 1] \\ \hline = & [1 & 4 & 6 & 4 & 1] \end{array} \quad \ldots$$

Voici le code

```
function [y]=coefbinA(n)
// "A" comme additive
// fournit le vecteur ligne des C_n^k, k=0:n
C=[1]; // initialisation n=0
for i=1:n, C=[C 0]+[0 C];end
y=C
```

Remarquons qu'à l'intérieur de la boucle `for` ci-dessus, l'instruction à réaliser ne dépend pas formellement de l'indice `i`. La variable `C` est une variable *locale* à la fonction dont la taille et la valeur change à chaque tour de la boucle `for`. Cette méthode évite de stocker en mémoire la totalité du triangle de Pascal (donc une matrice de taille n^2).

```
-->coefbinA(5)
 ans  =

!   1.    5.    10.    10.    5.    1. !

-->y=coefbinA(1029);max(y)
 ans  =

    1.430+308

-->y=coefbinA(1030);max(y)
 ans  =

   Inf

-->y=coefbinA(1000);y(351)
 ans  =

    4.025+279

-->y(651)
 ans  =

    4.025+279
```

À titre d'intermède ludique, essayons une illustration graphique.

```
-->xbasc();y=coefbinA(200);plot2d3("gnn",(0:200)',y')
```

Ceci permet de visualiser d'un coup l'ensemble des valeurs du vecteur $(C_{200}^k; 0 \leq k \leq 200)$ grâce à un diagramme en bâtons. On n'est pas très surpris de voir se dessiner une belle courbe en cloche. Les paramètres `nn` dans la chaîne ■`gnn`■ signifient (`n`=normal) que les graduations sur les deux axes sont régulièrement espacées (chacune avec son échelle propre). De ce fait, on ne voit rien[6] en dehors de l'intervalle $[70, 130]$. Pour en avoir le

6. Même avec des zooms, il est très difficile d'y détecter des bâtons de hauteur non nulle.

coeur net, essayons avec une échelle logarithmique sur l'axe des ordonnées. Il suffit pour cela de remplacer ■gnn■ par ■gnl■ (l=logarithmique).

```
-->xbasc();y=coefbinA(200);plot2d3("gnl",(0:200)',y')
```

L'enveloppe supérieure du nouveau diagramme en bâtons ressemble furieusement à un arc de parabole du type $y = a - c(x - 100)^2$. Si l'on ne connaît pas le théorème de De Moivre Laplace, ce dernier graphique permet de le conjecturer !

Pour finir voici un petit exercice de version. Expliquez et commentez le code suivant. On notera l'emploi de `find` qui aurait pu être utilisée ci-dessus pour raccourcir certains tests.

```
-->m=[1 2 4 %inf %inf];cumsum(m)
 ans  =

!   1.    3.    7.   Inf   Inf !

-->y=coefbinA(2000); y(200)
 ans  =

    7.622+279

-->n=min(find(y==%inf))
 n  =

    231.

-->u=y(1:230);v=y(2001:-1:1772);w=u-v;max(abs(w))
 ans  =

    0.
```

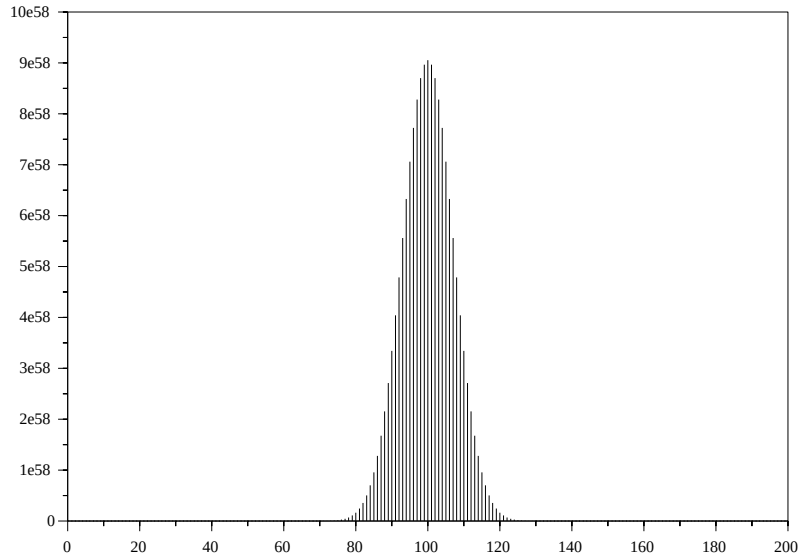

FIG. 1 – Coefficients binomiaux, $n = 200$

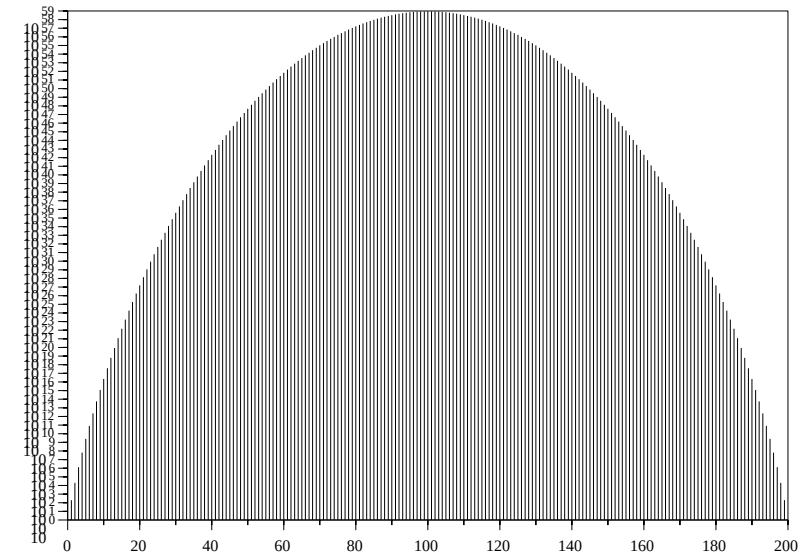

FIG. 2 – Coefficients binomiaux, $n = 200$, échelle verticale logarithmique

Ex 2. *Entonnoirs déterministes pour la LFGN*
Selon l'inégalité de Hoeffding, si les X_k sont indépendantes, identiquement distribuées et s'il existe des constantes a et b telles que $P(a \le X_1 \le b) = 1$, alors

$$\forall n \ge 1,\ \forall \varepsilon > 0, \quad \mathbf{P}\Big(\Big|\frac{S_n - \mathbb{E}\, S_n}{n}\Big| \ge \varepsilon\Big) \le 2\exp\Big(-n\frac{2\varepsilon^2}{(b-a)^2}\Big). \tag{1}$$

On peut utiliser ce type d'inégalité pour étudier quantitativement les fluctuations asymptotiques de S_n/n autour de $\mathbb{E}\, X_1$. Pour simplifier, on suppose désormais $a = 0$ et $b = 1$. On déduit alors facilement de (1) que pour tout entier $N \ge 2$ et tout $\alpha > 1/2$,

$$\mathbf{P}\left\{\forall k > N,\ \Big|\frac{S_k}{k} - \mathbb{E}\, X_1\Big| \le \sqrt{\frac{\alpha \ln k}{k}}\right\} \ge 1 - \frac{2}{2\alpha - 1}N^{1-2\alpha}. \tag{2}$$

On va illustrer graphiquement cette inégalité en faisant tracer la ligne polygonale d'interpolation des S_k/k et les deux courbes encadrantes (l'entonnoir) correspondantes. On prendra d'abord pour X_i des variables uniformes sur $[0, 1]$ puis des v.a. de Bernoulli de paramètres $p = 0,5$ puis $p = 0,1$.

Le travail à effectuer peut se décomposer ainsi
(a) Génération d'un échantillon $(X_1, \dots, X_n)$ suivant la loi prescrite ;
(b) Production des vecteurs $(S_k/k; 1 \le k \le n)$ et $(\mathbb{E}\, X_1 \pm k^{-1/2}\sqrt{\alpha \ln k}; 1 \le k \le n)$;
(c) Affichage graphique de la ligne polygonale des S_k/k et de l'entonnoir ;
(d) Assemblage de (a), (b) et (c) : script ou fonction ?

Génération d'échantillons. On fait appel au générateur aléatoire `rand`. Si on lui fournit en entrée une matrice A, il retourne une matrice aléatoire de mêmes dimensions dont les composants peuvent être issus soit d'un échantillon de la loi uniforme sur $[0, 1]$, soit de la loi normale standard. Pour fixer ce choix, il suffit d'entrer `rand('uniform')` ou `rand('normal')`.

```
-->rand('uniform');u=ones(1:5);rand(u)
 ans  =

!   0.0735908    0.9251916    0.6312055    0.3643678    0.9871317 !

-->rand('normal');u=ones(1:5);rand(u)
 ans  =

! - 0.5914395    2.0764882    0.9343553  - 0.3449410  - 1.1683503 !
```

La génération d'un échantillon uniforme de taille n se réduit ainsi à l'une des deux instructions `rand('uniform') ;rand(1 :n)` ou `rand(1 :n,'uniform')`. La nuance entre les deux est que dans la seconde, le passage au générateur uniforme est « local ».

```
-->rand('normal');rand(1:5,'uniform')
 ans  =
```

```
!   0.6195476    0.9568300    0.3985421    0.4879705    0.1285184 !

-->rand(1:5)
 ans  =

!   0.6092911  - 1.3172012    1.0871039    1.2827068  - 0.9788234 !
```

Comment utiliser ce générateur pour construire un n échantillon de la loi de Bernoulli de paramètre p ? Soit $U_1, \ldots, U_n$ des variables aléatoires indépendantes de même loi uniforme sur $[0, 1]$. Alors les

$$X_i := \mathbf{1}_{\{U_i<p\}}, \qquad i = 1, \ldots, n,$$

sont des variables de Bernoulli indépendantes de paramètre p puisque $\mathbf{P}(X_i = 1) = P(U_i < p) = p$. On est ainsi ramené au codage Scilab de la *fonction indicatrice* d'un évènement. En pratique cet évènement est défini par une condition, par exemple ici $U_i < p$. Les opérations de comparaison retournent une valeur booléenne `T` (true, vrai) ou `F` (false, faux).

```
-->[2<3, 4/2==2, 3<=2]
 ans  =

! T T F !
```

Les booléens n'étant pas des nombres, il reste à les convertir en `1` ou `0`, c'est le rôle de la fonction `bool2s` (raccourci de « Boolean to string »). La bonne nouvelle c'est que tout ceci se vectorise de la façon la plus simple qui soit.

```
-->rand('uniform');U=rand(1:5)
 U  =

!   0.6487530    0.0656285    0.8226198    0.2171544    0.4476069 !

-->U<0.5
 ans  =

! F T F T T !

-->bool2s(U<0.5)
 ans  =

!   0.    1.    0.    1.    1. !
```

Production des vecteurs. Avec ce que l'on sait déjà sur les opérations ponctuelles et `cumsum`, cette étape est quasiment évidente. La seule chose à signaler est que le logarithme

néperien se code `log` et agit ponctuellement sur les composantes d'une matrice [7], de même pour la racine carrée `sqrt` (« `square root` of »). En prenant comme paramètres d'entrée un échantillon `X`, l'espérance `EX` (un réel qu'il faudra fournir !) et `alpha`, on obtient le code suivant.

```
K=1:length(X); // vecteur des abscisses entières k=1,...,n
SPN=cumsum(X)./K; // SPN: sommes partielles normalisees
enton=sqrt(alpha.*log(K)./K); // forme de l'entonnoir
BH=EX+enton; // branche haute de l'entonnoir
BB=EX-enton; // branche basse
```

A l'exécution, on peut économiser de la place mémoire en supprimant la variable `enton` et en la remplaçant directement par sa valeur dans `BH` et `BB`. Scilab devra alors calculer deux fois plus de logarithmes et de racines carrées.

Affichage graphique. Pour afficher simultanément trois courbes, la fonction `plot2d` est toute indiquée. Ses paramètres d'entrée (dans la version minimaliste) sont deux matrices de même taille. L'affichage est fait en prenant pour vecteur des abscisses de la j-ème courbe la j-ème colonne de la première matrice et pour vecteur des ordonnées la j-ème colonne de la deuxième matrice. La fonction `plot2d` trace pour chaque paire de vecteurs colonnes les points de coordonnées correspondantes et complète par interpolation affine entre deux points consécutifs. Essayez

```
-->x=(1:10)';xx=(0.5:9.5)';y=rand(x);yy=rand(xx);xbasc();...
-->plot2d([x xx],[y yy])
```

Il y a une exception à cette règle, le cas où les vecteurs colonnes sont de dimension 1. Dans ce cas `plot2d` trace *une seule* ligne polygonale. Regardez l'effet de

```
-->xbasc();u=[1 2 5 4]; v=[1 3 4 0];plot2d(u,v)
```

On obtient la ligne polygonale joignant dans cet ordre les points $(1,1)$, $(2,3)$, $(5,4)$, $(4,0)$. Comparez avec

```
-->xbasc();u=[1 2; 5 4]; v=[1 3; 4 0];plot2d(u,v)
```

où `u` et `v` ont chacune deux colonnes de taille 2. On obtient bien deux lignes polygonales distinctes. Attention aussi à la façon de transposer pour avoir des vecteurs colonnes, examinez les 6 lignes suivantes et les sous-graphiques [8] correspondants (figure 3).

```
-->u=1:3;v=u;U=[1 1.5 3];V=v+1;xbasc()

-->xsetech([0, 0, 0.5, 0.5]);plot2d([u v], [U V])// haut-gauche
-->xsetech([0.5, 0, 0.5, 0.5]);plot2d([u v]', [U V]') // haut-droit
-->xsetech([0, 0.5, 0.5, 0.5]);plot2d([u; v]', [U; V]')// bas-gauche
-->xsetech([0.5, 0.5, 0.5, 0.5]);plot2d([u' v'], [U' V'])// bas-droit
```

7. Le logarithme d'une matrice carrée se code `logm`, de même sa racine carrée matricielle se code `sqrtm`, voir la documentation en ligne.

8. Obtenus grâce à `xsetech`.

Dans le premier cas, `[u v]` et `[U V]` constituent chacun un seul vecteur ligne, donc il n'y a qu'une ligne polygonale. La transposition effectuée dans le deuxième cas n'y change rien puisque on a maintenant un seul vecteur colonne d'abscisses et un seul vecteur colonne d'ordonnées, donc toujours une seule courbe.

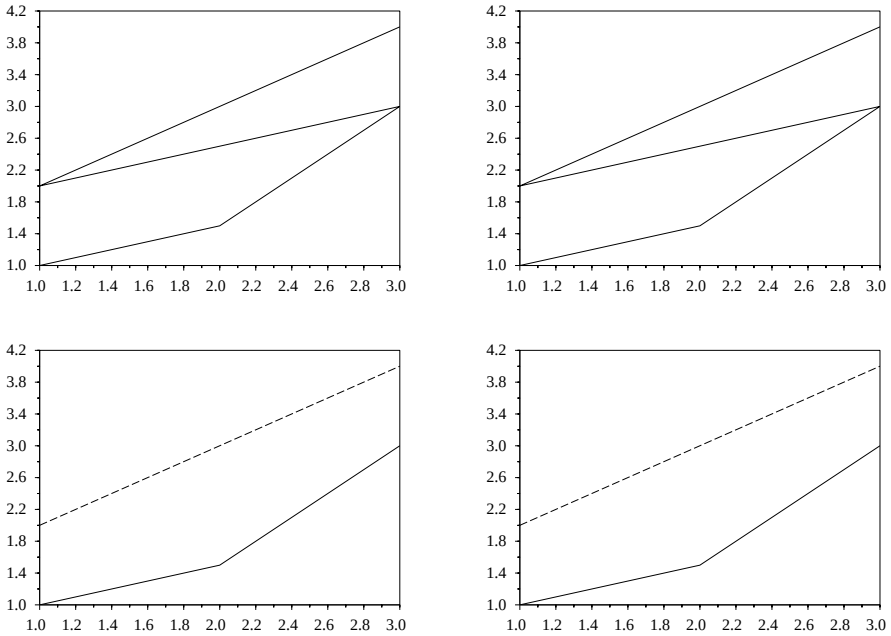

FIG. 3 – Canulars de `plot2d`

Revenons à notre entonnoir. On va regrouper l'ensemble des instructions qui aboutissent à sa création graphique dans un script. Il s'agit d'un fichier de suffixe `.sce` qui pourra être chargé par le menu

```
File ---> File operations ---> Exec
```

après bien sûr sélection à la souris du nom du fichier. Il est plus naturel d'écrire ici un script qu'une fonction, car le résultat cherché n'est pas le retour d'une variable[9], mais la création d'un graphique. D'autre part, il semble raisonnable de dissocier de ce script la création de l'échantillon, de façon à garder toute liberté de choisir la loi de X_1. Sinon il faudrait écrire un script pour chaque type de loi ou proposer un menu avec choix entre plusieurs lois. Nous devrons donc fournir de manière interactive l'échantillon `X` et son espérance `EX`. Ceci se réalise très simplement avec l'instruction `input`. Voici une première version basique.

9. Pour les accros des fonctions, on peut quand même présenter cela sous forme de fonction, il suffit de glisser sournoisement l'instruction `plot2d` dans le corps de la fonction et de lui faire retourner une variable bidon !

```
X=input('Rentrer l''échantillon (vecteur ligne)');
EX=input('Rentrer l''espérance');
alpha=input('Donner le paramètre alpha de l''entonnoir');
K=1:length(X); // vecteur des abscisses entières k=1,...,n
SPN=cumsum(X)./K; // SPN: sommes partielles normalisees
enton=sqrt(alpha.*log(K)./K); // forme de l'entonnoir
BH=EX+enton; // branche haute de l'entonnoir
BB=EX-enton; // branche basse
xbasc(); // nettoyage prealable de la fenetre
plot2d([K' K' K'],[BH' SPN' BB']);
```

Cette version a été testée et devrait marcher. Si vous l'avez saisie au clavier, vous avez peut-être eu quelques ennuis avec les `input`. Cela provient vraisemblablement des apostrophes dans les chaînes de caractères. Observez qu'elles ont été systématiquement doublées : `l■échantillon`, `l■entonnoir'`. Si vous omettez de les doubler, vous aurez des messages d'erreur. Explication : pour Scilab, les délimiteurs de chaînes de caractères sont le caractère apostrophe `'` ou le caractère guillemet du clavier `"`. Pour le voir, il est commode d'utiliser la fonction `string`.

```
-->string(ami)
         !--error     4
undefined variable : ami

-->string("ami")
 ans  =

 ami

-->ami='pote';string(ami)
 ans  =

 pote
```

Que se passe-t-il si la chaîne de caractères contient elle-même des apostrophes ou des guillemets ? Il faut lever l'ambiguité en préfixant chacun de ces caractères par une apostrophe ou un guillemet.

```
-->string("combien d'apostrophes?")
         !--error     3
waiting for right parenthesis

-->string("combien d"'apostrophes?")
 ans  =

 combien d'apostrophes?
```

```
-->string("combien d'"apostrophes?")
 ans  =

 combien d"apostrophes?

-->string("combien d''apostrophes et de guillemets du type ""?")
 ans  =

 combien d'apostrophes et de guillemets du type "?
```

On se propose maintenant d'apporter à notre script les améliorations suivantes :
- Forcer les deux branches de l'entonnoir à avoir la même couleur, disons rouge. Choisir la couleur de la courbe des S_k/k, disons bleu.
- Afficher une légende.
- Afficher un titre rappelant la valeur donnée à α.

Le fichier `Enton2.sce` suivant est une solution possible. Pour consulter la table des couleurs utilisée pour fixer le troisième paramètre de `plot2d`, tapez `xset()`. Notez aussi la façon de remplacer alpha par sa valeur numérique dans le titre[10].

```
X=input('Rentrer l''échantillon (vecteur ligne)');
EX=input('Rentrer l''espérance');
alpha=input('Donner le paramètre alpha de l''entonnoir');
K=1:length(X); // vecteur des abscisses entières k=1,...,n
SPN=cumsum(X)./K; // SPN: sommes partielles normalisees
enton=sqrt(alpha.*log(K)./K); // forme de l'entonnoir
BH=EX+enton; // branche haute de l'entonnoir
BB=EX-enton; // branche basse
xbasc(); // nettoyage prealable de la fenetre
xtitle("entonnoir pour alpha="+string(alpha));
leg="entonnoir@moyennes arithmétiques de la LFGN";
plot2d([K' K' K'],[BH' SPN' BB'], [5 2 5], "121",leg);
```

Ex 3. *Jeu de pile ou face et lois singulières*
Les X_k étant des variables aléatoires de Bernoulli indépendantes de même paramètre p, la série

$$U := \sum_{k=1}^{+\infty} 2^{-k} X_k$$

converge p.s. et on note μ_p la loi de U qui est donc une mesure de probabilité sur $[0,1]$. On a vu en cours que $\mu_{1/2}$ est la loi uniforme sur $[0,1]$, que pour $p \neq 1/2$ les mesures μ_p sont singulières (et étrangères 2 à 2) et que pour $0 < p < 1$ la fonction de répartition F_p

10. Je n'ai pas réussi à faire la même chose dans la légende...

de μ_p est continue. Le but de cet exercice est de tracer une représentation graphique de F_p. On note $F_{p,n}$ la fonction de répartition de

$$U_n = \sum_{k=1}^{n} 2^{-k} X_k.$$

1) *Calcul et représentation graphique de $F_{p,1}$ et $F_{p,2}$.*
Comme $U_1 = X_1/2$, on a immédiatement en notant $q = 1 - p = \mathbf{P}(X_1 = 0)$,

$$F_{p,1}(x) = \begin{cases} 0 & \text{si } x < 0, \\ q & \text{si } 0 \le x < 1/2, \\ q + p = 1 & \text{si } x \ge 1/2. \end{cases}$$

La variable aléatoire U_2 est discrète et n'a que 4 valeurs possibles. Grâce à l'indépendance de X_1 et X_2, on voit facilement que la loi $\mu_{p,2}$ de U_2 est fournie par le tableau

$u \in U_2(\Omega)$	0	1/4	1/2	3/4
$\mathbf{P}(U_2 = u)$	q^2	qp	pq	p^2

On en déduit la fonction de répartition

$$F_{p,2}(x) = \begin{cases} 0 & \text{si } x < 0, \\ q^2 & \text{si } 0 \le x < 1/4, \\ q^2 + qp & \text{si } 1/4 \le x < 1/2, \\ q^2 + qp + pq & \text{si } 1/2 \le x < 3/4, \\ q^2 + qp + pq + p^2 = 1 & \text{si } x \ge 3/4. \end{cases}$$

Pour afficher les fonctions en escaliers (« step functions »), on dispose de `plot2d2`. Sa forme d'utilisation la plus simple est

```
plot2d2("gnn",x,y)
```

où `x` et `y` sont des vecteurs *colonnes* ou des matrices de même taille. Dans le cas de deux vecteurs colonnes à n éléments, le résultat est l'affichage de la fonction en escalier qui vaut $y(i)$ sur le segment $[x(i), x(i+1)[$ $(1 \le i < n)$ et $y(n)$ au point $x(n)$. Si x et y sont deux matrices à d colonnes et n lignes, on obtient d fonctions en escalier, la j-ème courbe résultant de l'association de la j-ième colonne de x avec la j-ème colonne de y par la même méthode. La chaîne de caractères `"gnn"` qui figure comme premier argument de `plot2d2` signifie que les graduations sont régulièrement espacées sur chaque axe (`nn`) et que les abscisses des sauts peuvent être différentes d'une courbe à l'autre (`g` comme général). L'autre possibilité à la place de `g` est `o` (one) signifiant que les abscisses des sauts sont les mêmes pour les d courbes. Dans ce cas `x` est un vecteur colonne $n \times 1$ et `y` une matrice $n \times d$. Les autres paramètres de `plot2d2` sont optionnels et permettent comme pour `plot2d`, de contrôler la couleur et le type de trait, les légendes, le cadre et les graduations des axes. Essayons de tracer $F_{p,1}$ avec $p = 0.7$:

```
-->F1=[0.3 1 1];x=[0 0.5 1];xbasc();plot2d2("onn",x,F1)// erreur habituelle
                                                    !--error   999
 plot2d2: x has a wrong size, ( 1,1) expected
```

Ici le message d'erreur ne vient pas de l'erreur habituelle (entrée de vecteurs lignes au lieu de colonnes) mais du fait qu'en raison du paramètre `o`, `x` devrait être un seul vecteur colonne au lieu de 3 (de hauteur 1) ! Si on réessaye avec `g`, on n'a plus de message d'erreur, mais le résultat n'est pas celui attendu.

```
-->F1=[0.3 1 1];x=[0 0.5 1];xbasc();plot2d2("gnn",x,F1)// erreur habituelle

-->
```

On obtient un cadre vide ! Voici deux codages corrects (comme il y a une seule courbe, on peut utiliser aussi bien `o` que `g`).

```
-->xbasc();plot2d2("onn",x',F1')// vecteurs colonnes!

-->xbasc();plot2d2("gnn",x',F1')// vecteurs colonnes!
```

On observe enfin l'escalier escompté, mais l'affichage n'est pas terrible car tous les traits sont en noir et une partie de l'escalier est confondue avec le cadre. Pour y remédier, colorions l'escalier en bleu :

```
-->xbasc();plot2d2("gnn",x',F1',[2])// escalier bleu
```

Pour afficher $F_{0.7,2}$, on peut toujours faire quelques petits calculs à la main (ou de tête !) et adapter le codage ci-dessus. Mais pourquoi se priver des fonctionnalités de Scilab ? Voici une façon de faire qui permet de pressentir la solution générale :

```
-->p=0.7;q=1-p;G=q.*[q p];D=p.*[q p];A=[G D]
 A  =

!   0.09    0.21    0.21    0.49 !

-->F2=[cumsum(A) 1] // "1" pour la dernière marche
 F2  =

!   0.09    0.3    0.51    1.    1. !

-->x2=[0 0.25 0.5 0.75 1];xbasc();plot2d2("gnn",x2',F2',[3])// en vert
```

Sans surprise, la question suivante est : comment tracer $F_{0.7,1}$ et $F_{0.7,2}$ dans le même graphique ? Avec `"onn"`, il faut prendre comme vecteur des abscisses commun `x2'` en exploitant sournoisement le fait que tous les points de saut de $F_{0.7,1}$ sont aussi points de saut de $F_{0.7,2}$. Il faudra ensuite agrandir le vecteur `F1'` en fournissant les nouvelles valeurs correspondantes. Cela revient à *dupliquer* les composantes de `F1'`, sauf la dernière. Avec

"gnn", on peut appliquer à `x'` et `F1'` l'opération de duplication de toutes les composantes sauf la dernière. Dans les deux cas, on introduit artificiellement pour $F_{0.7,1}$ des sauts de hauteur nulle, au milieu d'une marche ou au début. Voyons la duplication[11].

```
-->xx=[x;x]
 xx  =

!   0.    0.5    1. !
!   0.    0.5    1. !

-->x1=xx(1:6)
 x1  =

!   0.  !
!   0.  !
!   0.5 !
!   0.5 !
!   1.  !
!   1.  !

-->x1=xx(1:(2*length(x)-1))
 x1  =

!   0.  !
!   0.  !
!   0.5 !
!   0.5 !
!   1.  !
-->FF=[F1;F1];FF1=FF(1:(2*length(F1)-1))
 FF1  =

!   0.3 !
!   0.3 !
!   1.  !
!   1.  !
!   1.  !
-->xbasc();plot2d2("gnn",[x1 x2'],[FF1 F2'],[2 3])
```

On obtient ainsi les deux escaliers, celui de $F_{0.7,1}$ en bleu et celui de $F_{0.7,2}$ en vert. Sur les segments horizontaux d'abscisses $[1/4, 1/2[$ et $[3/4, 1]$, on ne voit que du vert à l'affichage. C'est normal puisque ces deux segments sont communs aux deux courbes et la dernière courbe tracée a écrasé la couleur de la précédente. Pour le vérifier, essayez

```
-->xbasc();plot2d2("gnn",[x2' x1],[F2' FF1],[3 2])
```

11. Sans boucle `for`, sinon vous connaissez le tarif...

en prenant soin d'intervertir aussi l'ordre des couleurs (`[3 2]`).

Pour revenir au problème de l'agrandissement de `x1'`, il est clair que la solution de duplication n'est pas la meilleure[12] car trop liée à l'aspect dyadique du problème. Une méthode plus générale serait de répliquer autant de fois que nécessaire la dernière composante du vecteur. Voici une façon de le faire, écrite de manière portable (vous pouvez en faire une fonction qui prend en entrée un vecteur `x` et un vecteur `x2` plus long et qui retourne le vecteur `x1` de même longueur que `x2`, obtenu en agrandissant `x` par réplication de sa dernière composante autant de fois que nécessaire).

```
-->x
 x  =

!   0.    0.5    1. !

-->x2
 x2  =

!   0.    0.25    0.5    0.75    1. !

-->xx=ones(x2)
 xx  =

!   1.    1.    1.    1.    1. !
-->xx(1:length(x))=x // déjà le bon résultat mais coup de bol
 xx  =

!   0.    0.5    1.    1.    1. !
-->xx(length(x):length(x2))=0 // juste pour illustrer
 xx  =

!   0.    0.5    0.    0.    0. !
-->xx(length(x):length(x2))=x(length(x))//réplication  dernier élément de x
 xx  =

!   0.    0.5    1.    1.    1. !
-->x1=xx;
```

2) *Comment fabriquer la liste des masses ponctuelles de la loi de U_n ?*

D'abord il est clair que ces masses ponctuelles sont localisées aux points dyadiques $k2^{-n}$, $0 \leq k < 2^n$ (le réel 1 n'est jamais atteint comme somme *partielle* de la série U). Pour comprendre la construction du vecteur des masses ponctuelles, regardons leur formation jusqu'à $n = 3$ donnée par le tableau suivant dans lequel les valeurs possibles

12. Mais pourquoi laisser passer une occasion de s'amuser un peu ?

(i.e. les localisations des masses) sont notées $u = x_1 2^{-1} + x_2 2^{-2} + x_3 2^{-3}$ avec $x_i \in \{0, 1\}$.

u	0	1/8	1/4	3/8	1/2	5/8	3/4	7/8
(x_1, x_2, x_3)	$(0,0,0)$	$(0,0,1)$	$(0,1,0)$	$(0,1,1)$	$(1,0,0)$	$(1,0,1)$	$(1,1,0)$	$(1,1,1)$
$\mathbf{P}(U_1 = u)$	q				p			
$\mathbf{P}(U_2 = u)$	q^2		qp		pq		p^2	
$\mathbf{P}(U_3 = u)$	q^3	q^2p	qpq	qp^2	pq^2	pqp	p^2q	p^3

La structure d'arbre binaire saute aux yeux : on passe d'une ligne à l'autre en dédoublant le nombre de cases à remplir et chaque masse ponctuelle $\pi_{n,k}$ sur la ligne n engendre deux masses ponctuelles $\pi_{n+1,2k}$ (à la verticale) et $\pi_{n+1,2k+1}$ (à droite) sur la ligne $n+1$. On a les relations

$$\pi_{n,k} = \mathbf{P}(U_n = k2^{-n}), \quad \pi_{n+1,2k} = \pi_{n,k} \times q \quad \text{et} \quad \pi_{n+1,2k+1} = \pi_{n,k} \times p. \tag{3}$$

La justification mathématique est la suivante : pour $0 \leq j < 2^{n+1}$, notons le dyadique $u = j2^{-n-1}$ sous la forme $u = x_1 2^{-1} + \cdots + x_n 2^{-n} + x_{n+1} 2^{-n-1}$, où les x_i sont ses chiffres binaires (quand son écriture est formatée sur $n+1$ bits). On a alors $x_1 2^{-1} + \cdots + x_n 2^{-n} = k2^{-n}$ et $j = 2k$ ou $2k+1$ selon que $x_{n+1} = 0$ ou 1. De plus, k est déterminé de façon unique par u. En utilisant l'indépendance de U_n et X_{n+1}, on obtient :

$$\mathbf{P}(U_{n+1} = u) = \mathbf{P}(U_n = k2^{-j} \text{ et } X_{n+1} = x_{n+1}) = \mathbf{P}(U_n = k2^{-j})\mathbf{P}(X_{n+1} = x_{n+1}),$$

d'où les relations (3).

Les relations (3) suggèrent l'algorithme suivant pour passer d'une ligne à la suivante. Soit `M` le vecteur ligne des masses ponctuelles de la ligne n (sa longueur est 2^n). Pour obtenir celui de la ligne $n+1$, il suffit de

- Fabriquer les vecteurs `M.*q` et `M.*p`.
- Les assembler en intercalant les éléments de `M.*q` en positions impaires et ceux de `M.*p` en positions paires.

Cela ressemble à notre digression précédente sur la *duplication* d'un vecteur. Voici une solution (noter la signification de `length` pour une matrice et l'utilisation de `matrix` pour remodeler une matrice et se rappeler la règle pour la numérotation avec un seul indice de tous les termes d'une matrice) :

```
-->p=0.2;q=1-p;M=[q p]
 M  =

!   0.8    0.2 !

-->MM=[q.*M;p.*M]
 MM  =

!   0.64    0.16 !
!   0.16    0.04 !
```

```
-->M=MM(1:length(MM))
 M  =

!   0.64 !
!   0.16 !
!   0.16 !
!   0.04 !

-->M=matrix(MM,1,length(MM)) // mieux, évite de retransposer
 M  =

!   0.64    0.16    0.16    0.04 !

-->MM=[q.*M;p.*M];M=matrix(MM,1,length(MM))
 M  =

!  0.512   0.128   0.128   0.032   0.128   0.032   0.032   0.008 !

-->sum(M)
 ans  =

    1.
```

L'observation du tableau des masses ponctuelles des lois de U_1, U_2, U_3 peut donner l'idée d'un autre algorithme pour construire la ligne n de ce tableau. On remarque en effet que chaque nouvelle ligne a sa moitié gauche égale au produit par q de *toute* la ligne précédente et sa moitié droite égale au produit par p de toute la ligne précédente. D'où le code encore plus court :

```
M=[q p];
for i=2:n,
    M=[q.*M p.*M];
end
```

La justification mathématique est un peu moins immédiate que pour l'algorithme précédent. L'idée est de conditionner par X_1. Reprenant les notations $u = j2^{-n-1} = x_1 2^{-1} + \cdots + x_n 2^{-n} + x_{n+1}2^{-n-1}$, on remarque d'abord que $0 \leq u < 1/2$ équivaut à $x_1 = 0$ et que $1/2 \leq u < 1$ équivaut à $x_1 = 1$. De plus on peut écrire

$$U_{n+1} = \frac{1}{2}X_1 + \frac{1}{2}\sum_{k=2}^{n+1} \frac{1}{2^{k-1}} X_k, = \frac{1}{2}X_1 + \frac{1}{2}U'_n,$$

où U'_n est indépendante de X_1 et a *même loi* que U_n. Considérons alors la décomposition

$$\mathbf{P}(U_{n+1} = u) = \mathbf{P}(U_{n+1} = u \mid X_1 = 0)\mathbf{P}(X_1 = 0) + \mathbf{P}(U_{n+1} = u \mid X_1 = 1)\mathbf{P}(X_1 = 1).$$

Regardons d'abord le cas $0 \leq u < 1/2$ (donc $0 \leq j < 2^n$ et $x_1 = 0$). On a alors $\mathbf{P}(U_{n+1} = u \mid X_1 = 1) = 0$ et

$$\begin{aligned}
\mathbf{P}\Big(U_{n+1} = \frac{j}{2^{n+1}} \mid X_1 = 0\Big) &= \mathbf{P}\Big(\frac{1}{2}X_1 + \frac{1}{2}U'_n = \frac{j}{2^{n+1}} \mid X_1 = 0\Big) \\
&= \mathbf{P}\Big(\frac{1}{2}U'_n = \frac{j}{2^{n+1}} \mid X_1 = 0\Big) \\
&= \mathbf{P}\Big(U'_n = \frac{j}{2^n} \mid X_1 = 0\Big) \\
&= \mathbf{P}\Big(U'_n = \frac{j}{2^n}\Big) \quad \text{(indépendance de } X_1 \text{ et } U'_n) \\
&= \mathbf{P}\Big(U_n = \frac{j}{2^n}\Big) \quad (U'_n \text{ a même loi que } U_n).
\end{aligned}$$

Nous avons ainsi établi que

$$\mathbf{P}\Big(U_{n+1} = \frac{j}{2^{n+1}}\Big) = \mathbf{P}\Big(U_n = \frac{j}{2^n}\Big)\mathbf{P}(X_1 = 0) = q\mathbf{P}\Big(U_n = \frac{j}{2^n}\Big) \quad \text{pour tout } 0 \leq j < 2^n,$$

ce qui justifie la règle de construction de la moitié gauche de la ligne $n+1$.

Dans le cas $1/2 \leq u < 1$, on a $2^n \leq j < 2^{n+1}$, on pose $j = 2^n + l$ et on note que $\mathbf{P}(U_{n+1} = u \mid X_1 = 0) = 0$.

$$\begin{aligned}
\mathbf{P}\Big(U_{n+1} = \frac{j}{2^{n+1}} \mid X_1 = 1\Big) &= \mathbf{P}\Big(\frac{1}{2}X_1 + \frac{1}{2}U'_n = \frac{1}{2} + \frac{l}{2^{n+1}} \mid X_1 = 1\Big) \\
&= \mathbf{P}\Big(\frac{1}{2}U'_n = \frac{l}{2^{n+1}} \mid X_1 = 1\Big) \\
&= \mathbf{P}\Big(U'_n = \frac{l}{2^n} \mid X_1 = 1\Big) \\
&= \mathbf{P}\Big(U'_n = \frac{l}{2^n}\Big) \quad \text{(indépendance de } X_1 \text{ et } U'_n) \\
&= \mathbf{P}\Big(U_n = \frac{l}{2^n}\Big) \quad (U'_n \text{ a même loi que } U_n).
\end{aligned}$$

Ainsi

$$\mathbf{P}\Big(U_{n+1} = \frac{2^n + l}{2^{n+1}}\Big) = \mathbf{P}\Big(U_n = \frac{l}{2^n}\Big)\mathbf{P}(X_1 = 1) = p\mathbf{P}\Big(U_n = \frac{l}{2^n}\Big) \quad \text{pour tout } 0 \leq l < 2^n,$$

ce qui justifie la règle de construction de la moitié droite de la ligne $n+1$.

3) *Représentation graphique de $F_{p,n}$.*
Voici un script simple qui devrait faire l'affaire.

```
// Construction de la fdr F_n de U_n=\sum_{i=1}^n 2^{-i}X_i
// les X_i etant iid de loi Bern(p)
n=input('Rentrer le nombre n : ');
p=input('Rentrer le parametre p de la loi de Bernoulli : ');
q=1-p;
```

```
M=[q p];
for i=2:n,
    M=[M.*q M.*p];
end
F=[cumsum(M) 1];
x=0:2^{-n}:1;
xbasc(); plot2d2("onn",x',F')
```

Quand on le teste, tout marche bien pour des petites valeurs de n. On a assez vite des messages d'erreur sur la capacité mémoire dès qu'on augmente n (essayez $n = 20$). C'est bien normal puisque la taille des vecteurs manipulés est 2^n, donc pour $n = 20$, le script se voit intimer l'ordre de générer deux vecteurs de longueur supérieure à 1 million ! De toutes façons il est illusoire de rechercher une telle précision car l'affichage dépend du degré de résolution de l'écran. On peut considérer que les dimensions de la fenêtre graphique en pixels n'excèdent pas un millier. Donc comme $2^{10} = 1024$, il est un peu illusoire d'escompter une meilleure précision graphique avec $n > 10$ qu'avec $n = 10$. Si l'on veut prendre en compte un premier zoom, disons que $n = 14$ semble une valeur raisonnable.

Pour finir on vous laisse le soin (le plaisir ?)

- D'illustrer graphiquement la convergence de $F_{p,n}$ vers F_p en affichant sur un même graphique les courbes des $F_{p,n}$;
- D'écrire un script qui affiche sur un même graphique les représentations des $F_{14,p}$ pour p variant de 0.1 à 0.9 par pas de 0.1 en choisissant les couleurs de sorte que $F_{14,p}$ et $F_{14,1-p}$ aient la même couleur.

Ex 4. *L'exercice auquel vous avez échappé.*

Voici une fonction qui retourne le vecteur des valeurs du n-ième polynôme de Bernstein d'une fonction. Par rapport à tout ce que l'on vient de voir, la seule nouveauté est la façon de passer une fonction comme paramètre à une autre fonction.

```
function [Bf]=Bernstein(f,n,x)
// Retourne le n-ieme polynome de Bernstein de la fonction f
// a utiliser apres avoir introduit la fonction f dans l'environnement
// de travail, par exemple avec: deff('[y]=f(x)','y=sqrt(x)')
//
// x est un réel ou un vecteur
//
// 1) Generation de la liste C des coefficients binomiaux
//
C=[1]; // initialisation n=0
for i=1:n, C=[C 0]+[0 C];end
//
// 2) Generation de la liste Cf des C_n^k f(k/n)
//
Cf=C.*f((0:n)./n);
```

```
//
// 3) Fabrication de la liste X des x(i)^k (1-x(i))^(n-k)
//    et de Bf(i). X est une variable locale réaffectée pour chaque i
//    de façon à économiser de la place mémoire
//
for i=1:length(x),
    X=(x(i).^(0:n)).*((1-x(i)).^(n:-1:0));
    Bf(i)=sum(Cf.*X);
end

//  Exemple d'utilisation
//
/// deff('[y]=f(x)','y=sqrt(abs(2.*x-1))')
/// x=linspace(0,1,101)';
/// plot2d([x x x], [f(x) Bernstein(f,10,x)  Bernstein(f,200,x)])
```

Avec ce script, on peut s'amuser à étudier expérimentalement l'influence de la régularité de f sur la vitesse de convergence. On peut aussi vérifier expérimentalement que les polynômes de Bernstein d'une fonction monotone le sont aussi et que si f est convexe, ses polynômes de Bernstein aussi.

Références

[1] B. PINÇON. *Une introduction à Scilab*, version 0.996. E.S.I.A.L., Université H. Poincaré, Nancy.