Exercices de simulations sous SAS et R

Gilbert Colletaz

12 septembre 2019

Résumé

On présente ici les commandes principales permettant sous SAS ou R de simuler des processus ARMA, de calculer leurs autocorrélations, de les estimer et de réaliser les tests d'adéquation. On ne fait état que des commandes et options principales en vous invitant à explorer les aides offertes, aisément accessibles que ce soit sous SAS ou R, pour vous familiariser avec ces deux produits et leurs possibilités en matière de modélisation des séries temporelles univariées. On ne couvrira que ce qui a été vu jusqu'ici ce qui signifie notamment que nous ne traiterons que de séries stationnaires et non saisonnières. N'hésitez pas à reprendre les exemples présentés, à les modifier et à anticiper leurs résultats. Demandez-vous par exemple quelles sont a priori les évolutions attendues des fonctions d'autocorrélation pour différents processus simulés avant qu'elles ne soient graphées par votre logiciel.

Table des matières

1 Les fonctions d'autocorrélation théoriques des processus ARMA sous R

Étant donné un processus ARMA(p,q), $\phi(L)x_t = \theta(L)u_t$ où $\phi(L)$ et $\theta(L)$ sont des polynômes connus en L, il est possible sous R de réclamer les valeurs de ses autocorrélations. Il suffit pour cela d'utiliser la commande `ARMAacf(ar=`*var*`, ma=`*vma*`, lag.max=`*K*`, pacf=`*T/[F]*`)`, où :

- *var* est un vecteur contenant les coefficients des valeurs retardées de x dans l'équation $x_t = \ldots$. Le degré de l'opérateur L correspondant à un coefficient est donné par le rang de ce coefficient dans le vecteur *var*.
- *vma* est un vecteur contenant les coefficients des valeurs retardées de l'innovation u. Comme pour *var*, le degré de L correspondant à un coefficient est celui du rang de ce coefficient dans *vma*. Le coefficient de l'innovation courante u_t étant toujours égal à l'unité, on n'a pas à le préciser dans *vma*.
- K est un entier positif spécifiant l'ordre maximal des autocorrélations que l'on désire calculer. Avec `max.lag=12` on récupérera $\rho_1, \rho_2, \ldots, \rho_{12}$ et/ou $\phi_{11}, \phi_{22}, \ldots, \phi_{12,12}$
- `PACF=` renseigne une valeur logique True ou False. Lorsque `pacf=T` la commande renvoie les valeurs des autocorrélations partielles. Sa valeur par défaut est `F` : en son absence ce sont les autocorrélations qui sont calculées. Notez que lorsque `pacf=T`, la fonction renvoie K valeurs : $\phi_{11}, \ldots, \phi_{KK}$ mais qu'en son absence, ou avec `pacf=F`, elle renvoie $K+1$ valeurs : $\rho_0, \ldots, \rho_K$, ce qui peut impliquer des aménagements dans les commandes des graphes de ces fonctions comme on le verra dans les exemples qui suivent.

ARMAacf est une commande native de R. À ma connaissance il n'existe pas d'équivalent sous SAS. Voici quelques exemples d'utilisation de cette fonction :

— calcul et sauvegarde dans le vecteur acf des 8 premières autocorrélations du processus $(1-0.8L)x_t = u_t$:
```
K=8
var <- c(0.8)
acf <- ARMAacf(ar=var, lag.max=K)
```
ou, plus concisément :
```
acf <- ARMAacf(ar=c(0.8), lag.max=8)
```
— calcul et sauvegarde dans le vecteur pacf des 8 premières autocorrélations partielles du processus $x_t = (1-0.8L+0.15L^2)u_t$:
```
pacf <- ARMAacf(ma=c(-0.8, 0.15), lag.max=8, pacf=T)
```
— sauvegarde des 12 premières autocorrélations du processus ARMA(2,2) : $(1-L+0.24L^2)x_t = (1-0.8L+0.15L^2)u_t$:
```
acf <- ARMAacf(ar=c(1, -0.24), ma=c(-0.8, 0.15), lag.max=12)
```
— calcul et sauvegarde dans le vecteur iacf des 8 premières autocorrélations inverses du processus $x_t = (1-0.8L+0.15L^2)u_t$:
```
iacf <- ARMAacf(ar=c(0.8, -0.15),lag.max=8)
```

Les valeurs des fonctions d'autocorrélation ayant été sauvegardées, elles peuvent être graphées par exemple au moyen de la commande
`barplot(`*vecteur*`, names.arg= , xlab="...", ylab="...")`
où :
- *vecteur* = nom du vecteur contenant les valeurs à représenter,
- *names.arg*= vecteur contenant les valeurs à mettre sur l'axe des abscisses,
- *xlab*= précise le titre de l'axe des abscisses,
- *ylab*= précise le titre de l'axe des ordonnées.

Par exemple, on pourra faire :
— pour représenter les autocorrélations calculées comme vu précédemment,
```
barplot(acf, names.arg=seq(0,K,1),xlab="lags",ylab="ACF")
```
ici, `seq(0,K,1)` génère un vecteur contenant la séquence de nombres : $0, 1, 2 \ldots, K$. Naturellement la valeur de K doit être compatible avec celle utilisée dans ARMAacf.
— comme on sait que les corrélations sont comprises entre -1 et +1, on peut éventuellement limiter l'axe des ordonnées à cette plage avec :
```
barplot(acf, names.arg=seq(0,K,1),xlab="lags",ylab="ACF", ylim=c(-1,1))
```
— enfin, comme l'axe des abscisses indique K valeurs qui vont de 0 à K, on peut également limiter la représentation à cette plage au moyen de
```
barplot(acf, names.arg=seq(0,K,1),xlab="lags",ylab="ACF", ylim=c(-1,1), xlim=c(0,K+2))
```
attention : dans la définition des limites de l'axe des abscisses, on doit ajouter 2 à la valeur de K car une première abscisse est utilisée pour placer les valeurs de l'axe des ordonnées, et une deuxième est ajoutée pour représenter l'axe des ordonnées lui-même. Ce n'est qu'ensuite que sont placées les valeurs $0, 1, \ldots, K$ créées par l'option `seq(0,K,1)`.

On illustre quelques-unes des commandes précédentes en les appliquant au processus $x_t = (1-0.8L+0.15L^2)u_t$. Les 8 premières autocorrélations partielles, $\phi_{1,1}, \ldots, \phi_{88}$ sont donc obtenues avec :

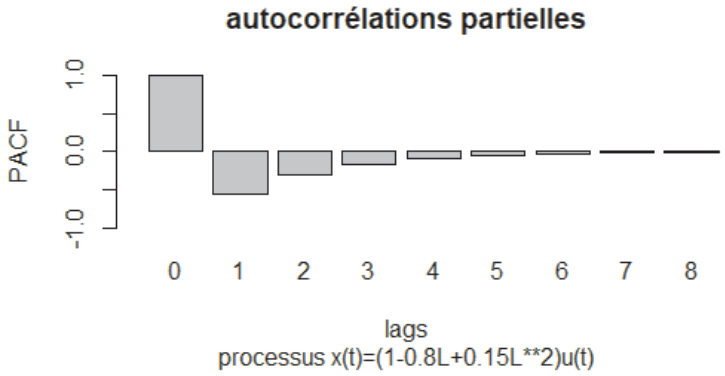
```
pacf <- ARMAacf(ma=c(-0.8, 0.15), lag.max=8, pacf=T)
```

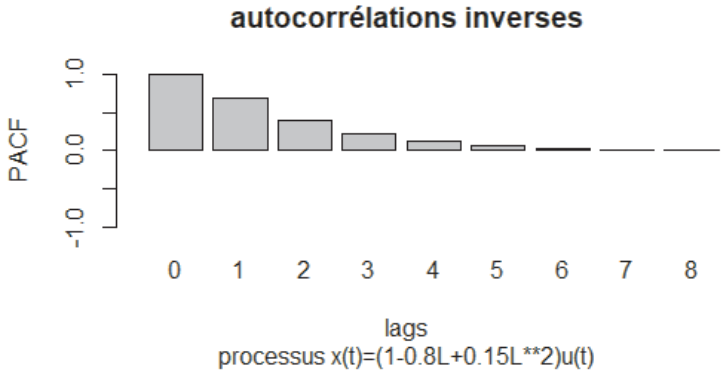
Afin d'avoir les mêmes valeurs sur l'axe des abscisses pour tous les graphiques, on met en premier élément de ce vecteur $\phi_{00} = 1$:
```
pacf <- c(1,pacf)
```

Ses 8 premières autocorrélations inverses sont données par :

```
iacf <- ARMAacf(ar=c(0.8, -0.15), lag.max=8)
```

Les graphes représentés ici sont obtenus en exécutant les deux commandes :






```
barplot(pacf,names.arg=seq(0,8,1),
     xlab="lags", xlim=c(0.0,10.0),
     ylab="PACF", ylim=c(-1,1),
     main="autocorrélations partielles",
     sub="processus x(t)=(1-0.8L+0.15L**2)u(t)")
```

et,

```
barplot(iacf,names.arg=seq(0,8,1),
   xlab="lags", xlim=c(0.0,10.0),
   ylab="IACF", ylim=c(-1,1),
   main="autocorrélations inverses",
   sub="processus x(t)=(1-0.8L+0.15L**2)u(t)")
```

2 Simulations de processus ARMA avec SAS et R

2.1 simulation d'une trajectoire sous SAS

Au moins deux possibilités vous sont offertes, la plus simple utilisant la fonction Armasim(). Il est cependant utile pour continuer à vous familiariser avec SAS et ses générateurs de nombres au hasard de vous forcer à programmer de A à Z la simulation d'une suite de réalisations d'un ARMA(p,q).

2.1.1 dans une étape data avec boucle `do ... end`; et générateur de nombres au hasard

— **Quelques précisions sur les générateurs de nombre au hasard**
Sans entrer dans les détails des algorithmes des générateurs de nombre au hasard, on peut savoir qu'ils produisent des nombres ayant une distribution uniforme et sont de deux types, déterministe ou aléatoire. Les premiers sont initialisés par une valeur entière, le *seed*, qui définit l'état initial du générateur à partir duquel il va produire une

suite de nombres ayant des caractéristiques telles qu'on ne peut la distinguer d'une suite de nombres uniformes réellement aléatoires. Ces générateurs déterministes se caractérisent notamment par leur période, c'est-à-dire le nombre de valeurs générées à partir duquel la suite va se répéter à l'identique. Un de leurs avantages est qu'en contrôlant le seed, on peut reproduire la même séquence de pseudo nombres aléatoires. A contrario, les générateurs vraiment aléatoires définissent leur état initial à partir de caractéristiques matérielles, par exemple l'état thermique du microprocesseur sur lequel ils sont implémentés à l'instant où on réclame leur exécution. En conséquence il n'est pas nécessaire de spécifier un seed, mais les séquences de nombres aléatoires qu'ils génèrent ne sont pas reproductibles.
Depuis SAS 9.4M5, il est conseillé d'utiliser la commande `streaminit` pour choisir le type de générateur et la fonction `rand()` pour spécifier la distribution désirée des aléatoires générées.
`Streaminit` offre le choix entre 8 générateurs déterministes et un générateur réellement aléatoire propre à Intel. A moins d'avoir une excellente raison pour opérer une sélection, le mieux est de prendre celui qui est offert par défaut : on n'a pas à se souvenir de son nom [1] et seule la valeur du seed doit alors être précisée.
La fonction `rand` va quant à elle assurer la transformation des nombres uniformément distribués issus du générateur vers des pseudo-nombres aléatoires ayant la distribution désirée. Sa syntaxe, `rand("distribution",liste de paramètres)`, est particulièrement simple et est partagée par d'autres fonctions traitant des distributions d'aléatoires comme `pdf()`, pour la densité, `cdf()` pour la répartition, ou encore `quantiles()`.

— **Exemple de simulation d'un ARMA(1,1) dans une étape data**
On veut simuler 100 réalisations du processus $x_t = 10 + 0.8x_{t-1} + u_t + 0.5u_{t-1}$ où u est un processus en bruit blanc gaussien de variance $\sigma_u^2 = 4$.

```
data arma11(drop=i u u0);
* définition du seed;
streaminit=123;
* initialisations sur x ;
x0=0;
* initialisation sur u;
* syntaxe : rand("normal",mu,sigma), par défaut N(0,1);
u0=rand("normal",0,2);
do i=-50 to 5000;
u=rand("normal",0,2);
x=10+0.8*x0+u+0.5*u0;
* la valeur courante de la variable, x,
* devient la valeur retardée pour l'itération suivante;
x0=x;
* l'innovation courante, u, de l'itération I
* devient l'innovation retardée pour l'itération suivante;
u0=u;
* sauvegarde de la pseudo-réalisation du ARMA(1,1)
* effectuée seulement après un certain nombre d'itérations
* afin d'éliminer l'impact des conditions initiales arbitrairement fixées;
if i>0 then output;
end;
run;
```

Sur cette base écrivez un programme simulant un AR(2) ou un MA(2). La seule petite difficulté est la mise à jour à chaque itération des deux valeurs retardées de la variable ou de l'innovation. Faites attention notamment à l'ordre dans lequel doivent être effectuées ces mises à jour.

2.1.2 avec la proc IML et la fonction armasim

Cette méthode est évidemment plus simple que la précédente. Un inconvénient est qu'elle oblige à passer par la proc iml dans laquelle on crée le vecteur des observations simulées qui peut ensuite être ramené dans une table standard.

1. Pour les curieux, il s'agit d'une version améliorée de l'algorithme déterministe Mersenne-Twister, il a une période d'environ 4×10^{6001} et comme l'indique R. Wincklin à titre de comparaison, le nombre d'atomes dans l'univers observable est de l'ordre de 10^{80} et le nombre de secondes écoulées depuis le *Big Bang* est approximativement 4×10^{17} : quelque soit la complexité de votre problème, vous ne réussirez pas à dépasser sa période.

La syntaxe de cette fonction est :
`armasim(phi, theta, mu, sigma, n, <, seed>)`
où :

- `phi` est un vecteur contenant les coefficients de la partie AR. Sa première composante est supposée valoir 1,
- `theta` est un vecteur contenant les coefficients de la partie MA. Sa première composante est supposée valoir 1,
- `mu` est un réel égal à l'espérance du processus à simuler,
- `sigma` est un réel positif égal à l'écart-type de l'innovation du processus,
- `n` est un entier donnant le nombre d'observations de la série simulée,
- `seed` est un nombre affectant le comportement du générateur de nombre au hasard de façon un peu complexe :
  - à la première exécution de la fonction :
    si seed > 0 : le nombre en question est utilisé pour définir l'état initial du générateur,
    si seed=0 : l'horloge du système est utilisée pour définir l'état initial du générateur (la simulation est alors non reproductible). C'est la valeur par défaut prise par seed si elle n'est pas indiquée par l'utilisateur.
    si seed<0 : c'est la valeur positive égale à -seed qui définit cet état initial.
  - lors des appels subséquents,
    si seed > 0 : la valeur de seed est inchangée
    si seed≤ 0 : la valeur de seed est modifiée arbitrairement.

Notez enfin que pour la génération des nombres au hasard cette fonction, de même que ranuni et autres fonctions ranxxx, fait appel à des générateurs de nombres au hasard anciens dont les propriétés sont nettement moins bonnes que celles des nouveaux générateurs associés à la commande `streaminit` et à la fonction `rand()` vus dans la section précédente. Elle reste cependant satisfaisante pour des exercices de simulations tels que ceux utilisés dans notre cours [2].
Ainsi, pour simuler, de manière non reproductible, le même processus qu'auparavant : $x_t = 10 + 0.8x_{t-1} + u_t + 0.5u_{t-1}$ où u est un processus en bruit blanc gaussien de variance $\sigma_u^2 = 4$, il suffira de faire :

```
proc iml;
phi = {1 -0.8};
theta = {1 0.5};
mu=50;
sigma=2;
x = armasim(phi, theta, mu, sigma, 100);
* création de la table arma11 contenant la variable de nom "x";
create arma11 var {"x"};
* écriture des observations de x dans la nouvelle table;
append;
* fermeture de la nouvelle table;
close arma11;
*sortie de la proc iml;
quit;
```

2.2 simulation d'une trajectoire sous R : arima.sim

La fonction arima.sim permet de simuler aisément des processus arma(p,q) avec R. Elle ne pose aucune difficulté particulière. Par exemple, pour simuler le processus considéré ci-dessus avec SAS, il suffira d'exécuter :

```
x <- arima.sim(n=100, list(ar = c(0.8), ma = c(0.5)), sd=2)+50
```

Notez également qu'il est possible aussi d'éliminer l'impact des conditions initiales au moyen de l'option n.start qui ne va pas sauvegarder les premières observations créées. Ainsi, pour reproduire ce qui a été fait ci-dessus sous SAS avec la boucle do, on fera :

```
x <- arima.sim(n=100, n.start=50, list(ar = c(0.8), ma = c(0.5)), sd=2)+50
```

Par ailleurs, il est possible d'appeler un générateur de nombres au hasard pour créer les innovations avec l'option

2. Les curieux pourront consulter le papier de Rick Wicklin, "Six reasons you should stop using the RANUNI function to generate random numbers"

`rand.gen=` qui doit faire référence à une fonction. Par défaut, `rand.gen=rnorm` et renvoie alors des pseudo-gaussiennes centrées réduites. Naturellement l'option `sd=` va normer les observations afin qu'elles aient l'écart-type désiré. Avec les deux commandes précédentes le processus d'innovation causal de x est donc gaussien.

La commande arima.sim vérifie que le modèle simulé est stationnaire. Si ce n'est pas le cas, un message d'erreur est retourné et elle n'est pas exécutée.

3 Des fonctions utiles : la division polynomiale sous SAS et R

Vous savez bien évidemment diviser un polynôme par un autre et vous savez que, dans le cadre de ce cours, cette opération permet de récupérer les coefficients des écritures moyenne mobile infinie et autorégressive infinie de tout ARMA(p,q) stationnaire et inversible puisque, partant de $\phi(L)x_t = \theta(L)u_t$, il vient :

- soit $x_t = \frac{\theta(L)}{\phi(L)}u_t \Rightarrow MA(\infty) : x_t = \psi(L)u_t$
- soit $\frac{\phi(L)}{\theta(L)}x_t = u_t \Rightarrow AR(\infty) : \pi(L)x_t = u_t$

La première de ces deux équations correspond naturellement à l'écriture de Wold, la seconde est quelquefois nommée fonction de prévision car elle exprime la valeur de x à une date t en fonction de valeurs de x aux instants antérieurs. Elle permet donc aisément de construire des prévisions itératives à partir des données passées à condition naturellement de tronquer ce polynôme infini, *i.e.* de considérer qu'au-delà d'un certain retard les coefficients π sont nuls. Nous verrons que sous certaines conditions, SAS utilise cette technique de construction de prévisions.

3.1 sous SAS, la fonction ratio

On accède à la fonction ratio via la proc IML. Sa syntaxe est `ratio(ar=var,ma=vma,`K`)` où, dans l'écriture de l'ARMA(p,q) $\phi(L)x_t = \theta(L)u_t$,

— `var` est le vecteur des coefficients du polynôme $\phi(L)$,
— `vma` est le vecteur des coefficients du polynôme $\theta(L)$,
— K est le nombre désiré de coefficients de l'écriture infinie, K

et elle retourne les coefficients de la division de vma par var.
Par exemple,

— les 4 premiers coefficients de l'écriture de Wold de $x_t = 0.8x_{t-1} + u_t$ sont obtenus par la suite de commandes :

```
proc iml;
ar={1 -0.8};
ma={1};
psi=ratio(ar,ma,4);
print psi;
```

qui affiche : $\psi = (1,\ 0.8,\ 0.64,\ 0.512)$, i.e. donne comme résultat le polynôme $(1 + 0.8L + 0.64L^2 + 0.512L^3)$

— si on veut obtenir la fonction de prévision du processus $x_t = u_t + 0.8u_{t-1} + 0.3u_{t-2}$, soit $(1 + 0.8L + 0.3L^2)^{-1}$ alors sachant que la fonction divise `vma` par `var`, il suffit d'intervertir les polynômes initiaux et de faire :

```
proc iml;
ar={1 0.8 0.3};
ma={1};
psi=ratio(ar,ma,4);
print psi;
```

ici, $\psi = (1,\ -0.8,\ 0.34,\ -0.032)$, soit en résultat le polynôme $(1 - 0.8L + 0.34L^2 - 0.032L^3)$.

3.2 sous R, la fonction ARMAtoMA

Il suffit de préciser 3 arguments :

— `var`, le vecteur des coefficients des valeurs retardées de la variable,
— `vma`, le vecteur des coefficients des innovations retardées,
— K, le nombre désiré de coefficients de l'écriture infinie, K

Sa syntaxe est : `ARMAtoMA(ar=var, ma=vma, lag.max=K)` et elle renvoie le résultat de la division de `(1,vma)` par `(1,var)`. En reprenant les exemples vus sous SAS et sa commande ratio, on aura ici

— la commande donnant les coefficients de l'écriture MA(∞) de $x_t = 0.8x_{t-1}+u_t$ est `ARMAtoMA(ar=c(0.8),ma=c(0),4)` qui renvoie 0.8000, 0.6400, 0.5120, 0.4096.

— pour obtenir la fonction de prévision, il suffit d'intervertir les polynômes dans l'appel de la fonction, en faisant attention au changement de signe, les termes autorégressifs passant dans l'opération à gauche du signe "=". Ainsi, la fonction de prévision de $x_t = u_t+0.8u_{t-1}+0.3u_{t-2}$ est obtenue par `ARMAtoMA(ar=c(-0.8,-0.3),ma=c(0),4)` ce qui donne les 4 coefficients : -0.8000 0.3400 -0.0320 -0.0764, et donc le polynôme $(1-0.8L+0.34L^2-0.032L^3-0.0764L^4)$ heureusement compatible avec ce que produit SAS.