Programmation dans R
Ch. 3. Simulation
M2 CEE

Pr. Philippe Polomé, Université Lumière Lyon 2

2018 – 2019

GATE Lyon / St-Etienne

UNIVERSITÉ DE LYON

Sommaire

Définitions

Simulation étape 1 : générer des données

Simulation étape 2 : évaluer les quantités d'intérêt

Simulation step 3 : Itérer sur plusieurs scénarios

Simulation : Présentation des résultats

Introduction

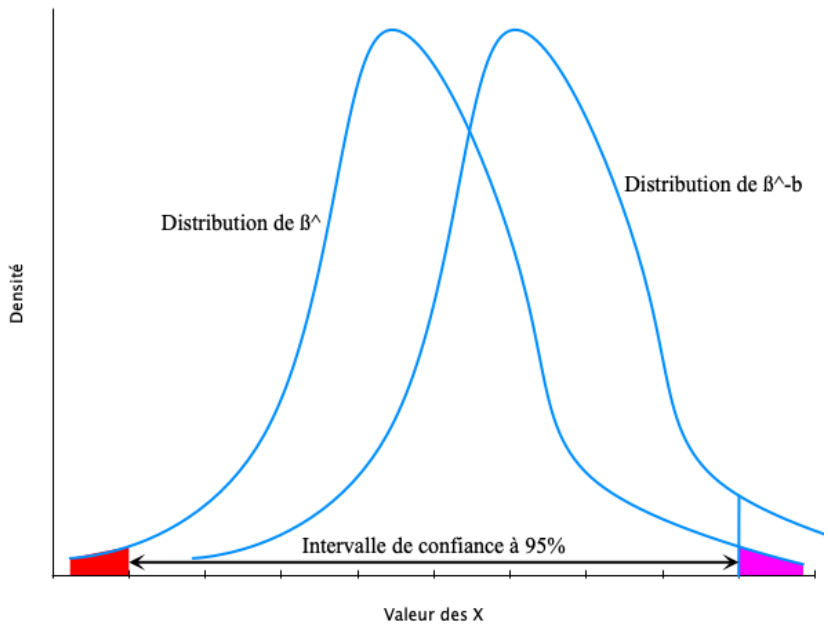
- Les procédures de simulation sont utilisées pour évaluer des algorithmes
  - c'est-à-dire : évaluer un test ou un estimateur
- Typiquement 3 étapes conceptuelles
  1. Générer des données dans l'ordinateur
     - Sur la base d'une ou pls lois (DGP : Data-Generating-Process)
  2. Évaluer les quantités d'intérêt
     - p-valeurs &/ou
     - paramètres estimés &/ou
     - prédictions d'un modèle
  3. Itérer ces 2 premiers pas sur plusieurs scénarios
- Ensuite, il faut présenter les résultats

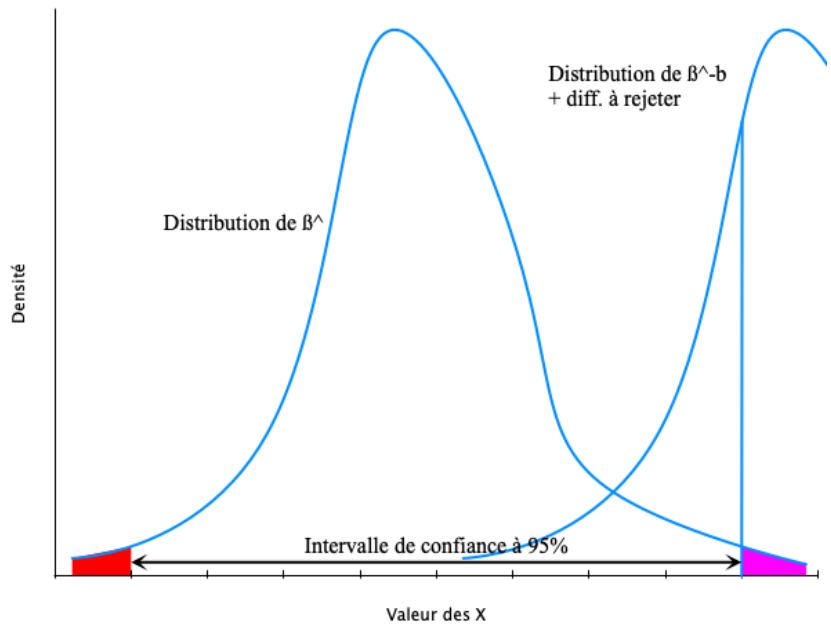
Exemples

- 2 “lessons” de SWIRL
  - R Programming
    - Lesson 13 Simulation
  - Regression Models
    - Lesson 10 Variance Inflation Factors
    - Lesson 11 Overfitting and Underfitting
- Comparer la **puissance** des tests d’autocorrélation de Durbin-Watson & Breusch-Godfrey
  - dans 2 spécifications d’une régression linéaire

Rappels

- Puissance d'un test
  - Probabilité qu'il rejette H_0 lorsque H_0 est fausse
  - $Puissance = 1 - \beta = 1 - \Pr\{Type\ II\ error\}$
  - Dépend de la taille de l'éch.
  - mais aussi de la fausseté de H_0





Rappels

- Séries AR(1) $\epsilon_t = \rho\epsilon_{t-1} + \mu_t$ est **stationnaire** si
  - μ_t bruit blanc (stationnaire, non-corrélé)
    - $|\rho| < 1$
- Les tests de Durbin-Watson & Breusch-Godfrey
  - servent tous deux à détecter un terme d'erreur AR(1) stationnaire
  - Mais le premier n'est pas valable s'il y a un retard de y dans les régresseurs

Survol du programme

- On définit 3 fonctions dans R
  - qui capturent les 3 étapes ci-dessus
    - Générer, évaluer, itérer
- Ensuite on lance la simulation
  - et on présente le résumé des résultats
    - numériquement & graphiquement
- Soient 2 DGP linéaires
  - Un modèle "trend" : $y_i = \beta_1 + \beta_2 i + \epsilon_i$
    - i indice le temps car i est un numéro de ligne
  - Un modèle "dynamic" : $y_i = \beta_1 + \beta_2 y_{i-1} + \epsilon_i$
    - Durbin-Watson n'est pas valable dans le cas de ce 2nd modèle
  - ϵ suit un AR(1) stationnaire
    - $\epsilon_i = \rho\epsilon_{i-1} + \mu_i \; i = 1...n \; |\rho| < 1$
- On fixe
  - $\beta = (.25; -.75)'$
  - $\epsilon_0 = 0$

Survol du programme

- !! On prend ces 2 DGP pour générer les données !!
  - On sait ce qui est vrai
  - On veut voir si les tests arrivent à nous le dire
  - Parce que s'ils ne peuvent pas en conditions simulées, il ne pourront pas dans la nature
- On veut évaluer la puissance des 2 tests
  - Pour les 2 DGP (trend, dynamic)
  - Pour une **taille** $\alpha = 0.05$ (size ds R)
    - α = proba de RH_0 alors que H_0 est vraie
    - = proba d'une erreur de type I
    - On pourrait regarder pls niveaux de α
  - Pour différents niveaux d'autocorrélation
    - $\rho = 0; 0.2; 0.4; 0.6; 0.8; 0.9; 0.95; 0.99$
  - Pour 3 tailles d'éch. $n = 15; 30; 50$
    - on pourrait aussi faire 5000 au lieu de 50 pour $n \to \infty$

Sommaire

Définitions

Simulation étape 1 : générer des données

Simulation étape 2 : évaluer les quantités d'intérêt

Simulation step 3 : Itérer sur plusieurs scénarios

Simulation : Présentation des résultats

Générer les données

- On implémente les 2 DGP (trend & dynamic)
  - en **créant** une fonction dgp( )
- De la manière suivante :
  - Énoncer les arguments de la dgp
    - et leurs valeurs par défaut
  - Spécifier comment ces arguments vont acquérir une valeur
    - Créer une matrice multidimensionnelle des valeurs des paramètres
    - nbr tailles éch. (3) $\times$ nbr modèles (2) $\times$ nbr corrélations (8)
  - Spécifier comment les données sont générées pour chaque jeu de valeurs des paramètres

Générer les données – créer la fonction dgp

- Les arguments qu'il faut donner à dgp sont :
  - nobs le nombre d'obs., défaut 15
  - model spécifie l'équation,
    - peut prendre 2 valeurs : "trend" et "dynamic"
  - corr autocorrélation ρ, default 0
  - coef les vraies valeurs β, défaut 0.25 et -0.75
  - sd "standard deviation" – écart-type de l'erreur ϵ
- **Créer** la fonction

```
dgp <- function(nobs = 15, model = c("trend", "dynamic"), corr
= 0, coef = c(0.25, -0.75), sd = 1)
```

- À ce stade, la fonction est un récipient vide
  - mais des arguments peuvent lui être passés
    - des valeurs par défaut sont définies
  - Ces arguments sont utilisés plus loin

Générer les données – structure de dgp

- À l'intérieur de dgp, on fabrique ϵ et y
  - Pour chaque modèle
- dgp ensuite renvoie les données dans un "data.frame"
- La structure des commandes à l'intérieur de dgp est
  { définition de l'erreur
  if(model is "trend"){create data of trend model}
  else {data of dynamic model}
  }

Générer les données – corps de dgp

{

model <- match.arg(model)

- saisit l'argument "model" passé à dgp
  - soit "trend", soit "dynamic"
  - et le place dans l'objet model

err <- as.vector(filter(rnorm(nobs, sd = sd), corr, method = "recursive"))

- filter crée une série temp.
  - de nobs valeurs d'une v.a. normale d'écart-type sd et autorégressive de coefficient "corr"
- as.vector traite la série temp. comme un vecteur

if(model == "trend") {

- Ici, on définit les valeurs de y pour le modèle "trend" (==)

x <- 1 :nobs crée une tendance (1 à nobs)

y <- coef[1] + coef[2] * x + err

Générer les données – corps de dgp

} else {

- Idem pour le modèle “dynamic”

y <- rep(NA, nobs)

- prépare la place pour y

y[1] <- coef[1] + err[1]
for(i in 2 :nobs)
y[i] <- coef[1] + coef[2] * y[i-1] + err[i]

- Définition récursive de *y*

x <- c(0, y[1 :(nobs-1)]) : x défini comme lag de y
} fin du “else”
return(data.frame(y = y, x = x))

- On est tjrs dans dgp
  - produit le dataframe de y & x (= soit trend, soit lag de y)

}

- dgp est définie
  - elle emploie bien les paramètres coef, nobs, model, sd et corr

Sommaire

Définitions

Simulation étape 1 : générer des données

Simulation étape 2 : évaluer les quantités d'intérêt

Simulation step 3 : Itérer sur plusieurs scénarios

Simulation : Présentation des résultats

Évaluation pour un seul scénario : principe

- En se basant sur cette DGP
  - Dans laquelle **toutes** les séries de donnés ont un terme d'erreur AR(1)
    - sauf lorsque $\rho = 0$ (corr)
  - On calcule la puissance du test pour une combinaison de paramètres
    - Une telle combinaison s'appelle un **scénario**
    - Les paramètres sont coef, nobs, model, sd et corr
    - Mais seules les variations de nobs, model et corr sont d'intérêt
- La puissance va être mesurée par le nombre de fois que la p-valeur est $\leq .05$
  - Sachant que H_0 est "absence d'autocorrélation"
    - pour les 2 tests
  - Ce nombre de fois est le nombre d'**itérations**
    - régi par une fonction simpower( )

Évaluation pour un seul scénario : conception de simpower

- Dans simpower( ), on va itérer
  - via un loop (boucle) for( )
    - Par défaut, avec nrep = 100 itérations
  - dans lequel on génère un jeu de données en utilisant dgp( )
- Dans ce jeu de données
  - appliquer dwtest( ) (Durbin-Watson) & bgtest( ) (Breusch-Godfrey)
    - aux résidus de la régression de y~x
  - stocker les 2 p-valeurs associées
- En terminant le loop for( )
  - Renvoyer la **proportion** de p-valeurs significatives, soit $< .05$

Évaluation pour un seul scénario : écriture de simpower

- D'abord on déclare simpower
  - nrep
  - size pour se donner la possibilité de contrôler le α dans les tests qui suivent
  - ... pour que simpower passe les paramètres de dgp

```
simpower <- function(nrep = 100, size = 0.05, ...)
{
pval <- matrix(rep(NA, 2 * nrep), ncol = 2)
```

- On crée une matrice "pval"
  - de taille nrep $\times$ 2 (car 2 tests)
  - remplie de NA – qu'on va remplir par la suite

```
colnames(pval) <- c("dwtest", "bgtest")
```

- noms des col. de pval

Évaluation pour un seul scénario : loop de simpower

for(i in 1 :nrep) loop for, sur i, de 1 à nrep

{

dat <- dgp(...)

- Exécute dgp, en renvoyant les données dans dat
  - L'arg ... fait que des arguments passent d'une fonction à une autre
    - Ça permet de mettre les arg de dgp dans une fonction ultérieure qui "enveloppera" dgp et simpower

pval[i,1] <- dwtest(y ~ x, data = dat, alternative = "two.sided")$p.value

- p-valeurs du Durbin-Watson
  - On en remplit la 1º col de la matrice pval
  - Lorsque le modèle est "dynamic", dw n'est en principe pas valable

pval[i,2] <- bgtest(y ~ x, data = dat)$p.value

- Idem pour Breush-Godfrey sur la 2º col de pval

}

Évaluation pour un seul scénario : renvoyer les résultats

return(colMeans(pval < size))

- Prend la moyenne par col du nombre de fois où pval<size
  - l'expression pval < size va retourner 1 si pval < 0.05 (la taille par défaut), zéro sinon
  - return( ) sert en programmation pour renvoyer le résultat de l'opération entre ( )
    - C'est le résultat de la fonction utilisable à l'extérieur
    - Si la fin d'une fonction est atteinte sans un return, la valeur de la dernière expression évaluée est renvoyée

} Ceci ferme simpower

- Les calculs sont essentiellement finis
  - On a donc à présent 2 "blocs" : dgp & simpower
  - le 1º crée les données, le 2º calcule la puissance
  - Il faut à présent invoquer ces blocs en passant les paramètres désirés

Sommaire

Définitions

Simulation étape 1 : générer des données

Simulation étape 2 : évaluer les quantités d'intérêt

Simulation step 3 : Itérer sur plusieurs scénarios

Simulation : Présentation des résultats

Évaluation sur tous les scénarios par itération

- On crée une 3º fonction

```
simulation <- function(corr = c(0, 0.2, 0.4, 0.6, 0.8, 0.9, 0.95,
0.99), nobs = c(15, 30, 50), model = c("trend", "dynamic"), ...)
```

- simulation va invoquer simpower
  - sur les différentes combinaisons de paramètres (scénarios) qu'on lui passe
    - autocorrelation corr, sample size nobs, model
- Comme dgp va être appelée (par simpower) sur des vecteurs de paramètres,
  - chaque i de simpower va calculer autant de proportions
    - Ici : 8 niveaux de corrélations $\times$ 3 tailles d'éch. $\times$ 2 modèles = 48 jeux de données (/itération)

Évaluation sur tous les scénarios par itération

- Dans la prochaine diapo, la fonction simulation
  - crée les scénarios dans un "data.frame"
    - par la commande expand.grid( )
  - ensuite passe ces scénarios à simpower
    - qui les repasse à dgp
    - avec les mêmes noms
  - calcule les puissances des 2 tests pour chaque scénario
    - via un loop for( ) qui reprend chaque scénario en appelant une cellule de expand.grid( )
  - réarrange les résultats et les stocke dans un "data.frame"
- simulation est plus de la programmation et moins de l'éctrie

Évaluation sur tous les scénarios par itération

```
{
prs <- expand.grid(corr = corr, nobs = nobs, model = model)
```

- Crée un data frame appelé prs contenant les scénarios qu'on a envoyés dans la fonction "simulation"
  - Ce data frame contient une ligne par scénario
    - 1º col : corr ; 2º : nobs ; 3º : model

```
nprs <- nrow(prs)
pow <- matrix(rep(NA, 2 * nprs), ncol = 2)
```

- Comme pval, une matrice pow remplie de NA avec 2 col et nprs lignes

```
for(i in 1 :nprs) pow[i,] <- simpower(corr = prs[i,1], nobs = prs[i,2],
model = as.character(prs[i,3]), ...)
```

- loop (for) pour remplir les lignes de pow
  - en appelant simpower sur le sénario correspondant (corr, nobs, model)
  - Simpower renvoie les proportions de tests DW (col 1) et BG (col 2) significatifs

Évaluation sur tous les scénarios : sorties

- La fin de la fonction simulation présente les résultats

rval <- rbind(prs, prs)

- Matrice rval : concaténation verticale de matrices prs
  - 2 prs l'une au-dessus de l'autre

rval$test <- factor(rep(1 :2, c(nprs, nprs)), labels = c("dwtest", "bgtest"))

- Ajoute une col "test" à la matrice rval
  - Cette col contient un factor à 2 niveaux : dwtest & bgtest

rval$power <- c(pow[,1], pow[,2])

- Ajoute 1 col "power" avec les éléments de col 1 & 2 de pow
  - donc, les résultats de simpower

rval$nobs <- factor(rval$nobs)

- Ajoute une col "nobs"

return(rval)

- Renvoie la matrice rval, contenant les résultats de simpower
  - dans la col power selon la matrice pow
  - en rajoutant 2 cols nobs et test, selon prs

} ferme la fonction simulation

Sommaire

Définitions

Simulation étape 1 : générer des données

Simulation étape 2 : évaluer les quantités d'intérêt

Simulation step 3 : Itérer sur plusieurs scénarios

Simulation : Présentation des résultats

Simulation steps : résumé

- Le programme de simulation est complet
  - il faut l'appeler et présenter les résultats
  - Avant ça, un résumé du programme en mots
- En Step 3, simulation crée la matrice prs dont les lignes comprennent
  - une corrélation parmi {0, 0.2, 0.4, 0.6, 0.8, 0.9, 0.95, 0.99}
  - une taille d'éch. parmi {15, 30, 50}
  - un modèle {trend, dynamic}
- Step 3 simulation appele Step 2 Simpower qui appelle Step 1 dgp
  - qui crée une jeu de données correspondant à ce scénario
    - un parmi 8x3x2 48 jeux de donnée

Simulation steps : résumé

- En Step 2, Simpower calcule 2 tests (DW & BG) sur les résidus de la régression y~x
  - Si le jeu de données correspond à un modèle "dynamic", DW n'est pas approprié
  - Step 2 répète ce calcul 100 fois (=nrep)
    - puis calcule la proportion de test donnant une pval<0.05
    - donc, RH_0 où H_0 : absence d'autocorrélation
- Un scénario est une cellule de la matrice prs
  - Step 3 simulation continue à appeler Step 2 Simpower jusqu'à arriver au bout des cellules
- Cette division en Steps (blocs)
  - n'est pas obligatoire
  - mais améliore la lisibilité du code
  - et permet de réutiliser certains steps dans d'autres simulations

Simulation : la fin

- À présent, exécuter les 3 fonctions pour les mettre en mémoire
  - Puis appeler simulation( )
    - Il faut que AER soit en mémoire pour le dwtest
  - Éventuellement on fixe la seed
    - set.seed(123)

```
psim <- simulation( )
```

- simulation ( ) renvoie rval - donc psim est rval
- Si on ne change pas les défauts, ce programme prend moins d'une minute
  - Avec 100 itérations dans Simpower (nrep=100)
    - La précision n'est pas très élevée
    - Mais passer à 10 000 ne demande que de changer un paramètre
- Pour présenter les résultats, utiliser des *tables numériques*
  - Via xtabs( ), qui transforme le "data.frame" en "table" qui classe selon les 4 paramètres (corr ect)

```
tab <- xtabs(power ~ corr + test + model + nobs, data = psim)
```

Simulation : la fin

- Pour améliorer la présentation
  - on peut utiliser ftable( ) (flat table)
    - Les valeurs de ρ dans les col et les autres paramètres sur les lignes

ftable(tab, row.vars = c("model", "nobs", "test"), col.vars = "corr")

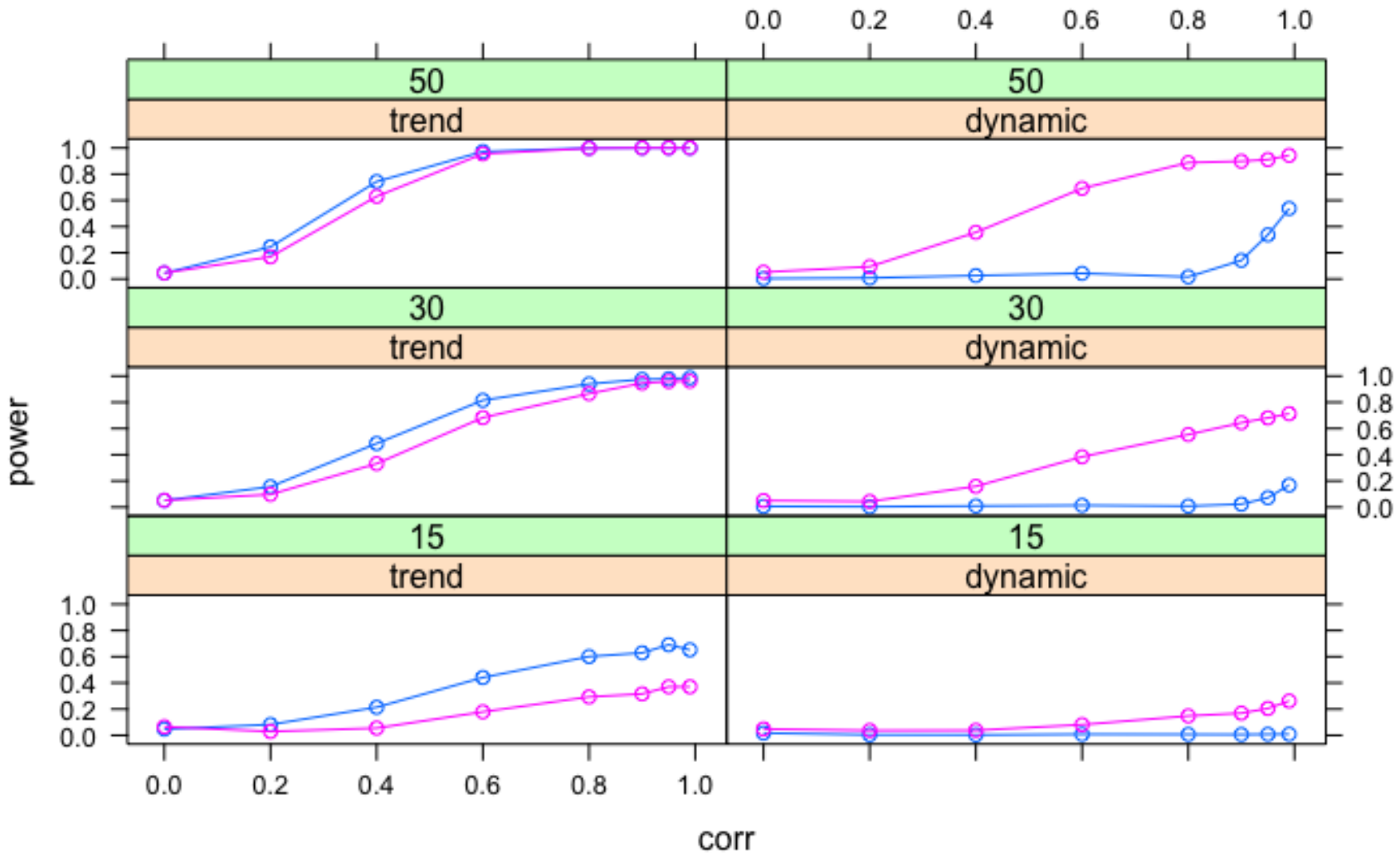
- Comme on met les résultats du test sur la dernière ligne
  - la table permet de comparer la puissance entre les 2 tests dans les mêmes conditions
    - Durbin-Watson est un peu meilleur dans le modèle trend, mais cet avantage sur Breusch-Godfrey diminue en augmentant ρ & n
    - dans le modèle dynamic, Durbin-Watson n'a pratiquement pas de puissance sauf à de très hautes corrélations, alors que Breusch-Godfrey est acceptable

Simulation : Graphiques

- On peut aussi présenter les résultats dans un graphique
  - avec la même interprétation
  - mais qui rend mieux les différences

```
library("lattice")
xyplot(power ~ corr | model + nobs, groups = ~ test, data = psim,
type = "b")
```

Avec nrep =1000 – bleu = DW : rose = BG



Devoir 3

- Répliquer la présente analyse avec 2 autres tests
  - p.e. 2 tests d'hétéroscédasticité
    - ou bien 2 tests np
  - sur données simulées évidemment
    - mais la procédure de simulation peut varier
- Lesson de SWIRL
  - Regression Models, Lesson 10 Variance Inflation Factors