Modélisation financière à l'aide de la programmation orientée objet avec Rcpp

Denis-Alexandre Trottier

Université Laval, Québec (Québec), Canada

26 Mai 2017

Résumé

Objectif : Coder un grand nombre de modèles économétriques.

Approche : programmation en C++ via Rcpp :

- Boucles beaucoup plus rapides qu'en R.
- Programmation orientée objet : permet de coder plusieurs modèles de façon très compacte et efficace.

Présentation d'un exemple simplifié avec étude de performance.

Outline

Exemple de modélisation financière

$\{y_t\}_{t\in\mathcal{T}}$: série de rendements financiers.

$\mathcal{T} = \{1, \ldots, T\}$: discrétisation du temps.

Forme générale du modèle :

$$y_t = \mu_t + \sigma_t z_t,$$

- $\{\mu_t\}_{t\in\mathcal{T}}$: processus de moyenne conditionnelle.
- $\{\sigma_t\}_{t\in\mathcal{T}}$: processus de volatilité conditionnelle.
- $\{z_t\}_{t\in\mathcal{T}}$: bruit blanc fort de variance unitaire.

Problématique de programmation

Supposons qu'on considère :

- 10 modèles différents pour $\{\mu_t\}_{t\in\mathcal{T}}$.
- 10 modèles différents pour $\{\sigma_t\}_{t\in\mathcal{T}}$.
- 10 modèles différents pour $\{z_t\}_{t\in\mathcal{T}}$.

$\Rightarrow$ Il y a donc $10 \times 10 \times 10 =$ 1000 modèles à programmer ☹

De plus, on pourrait également vouloir programmer :

- des modèles multivariés,
- des modèles (multivariés) à changement de régime.

$\Rightarrow$ Le nombre de modèles à coder devient excessivement grand.

La solution : programmation orienté objet en C++

Cette solution fut utilisée pour écrire la librairie R MSGARCH.

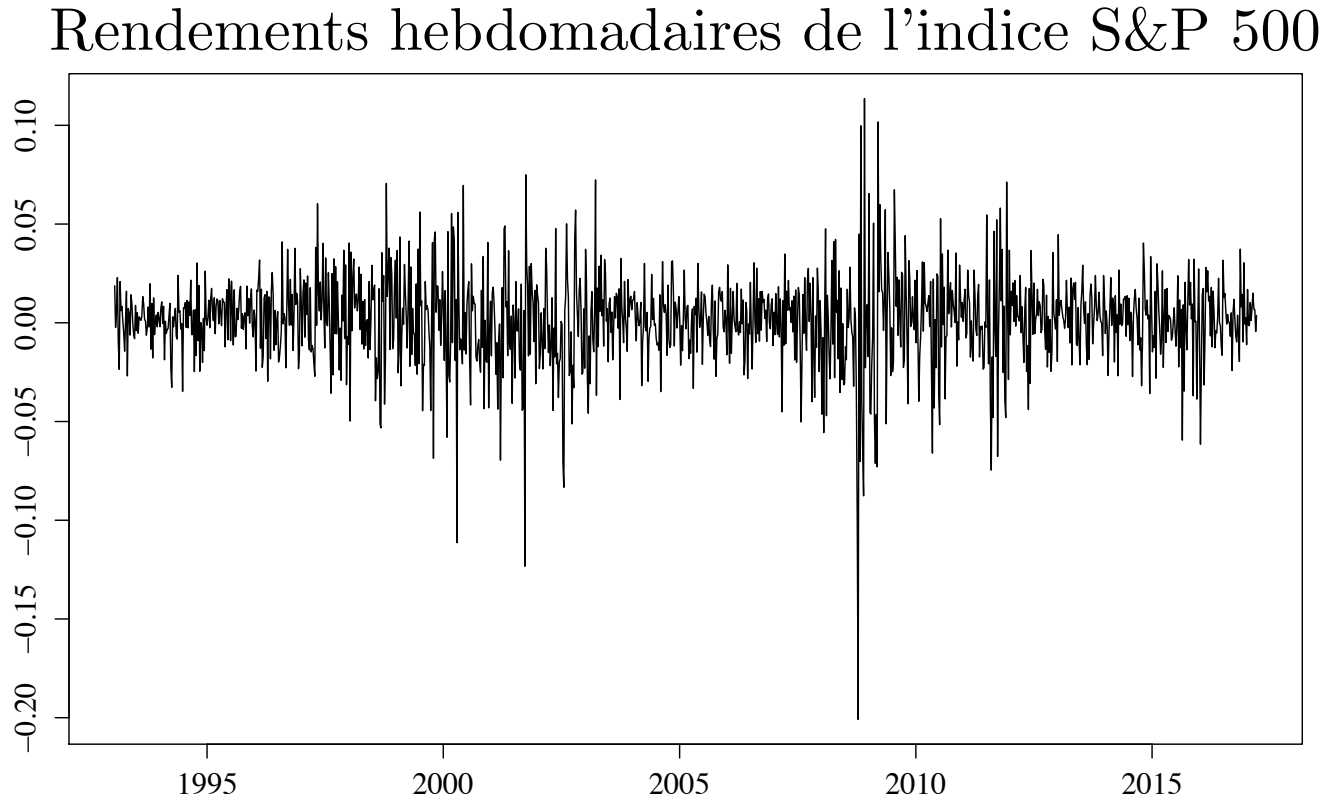
Programmation orientée objet en C++ : permet de coder une large classe de modèles de façon efficace et compacte.

Vitesse : boucles C++ beaucoup plus rapides qu'en R.

Rcpp (librairie de Eddelbuettel et al., 2017) facilite l'usage combiné de R et C++.

Outline

Motivation empirique des modèles GARCH



Les "grappes" de volatilité dans la série $\{y_t\}$ ci-haut peuvent être modélisés par des processus de la famille GARCH.

Notre objectif

Soit le **modèle GARCH(1,1)** suivant :

$$y_t = \sigma_t z_t, \qquad \sigma_t^2 = \omega + \alpha y_{t-1}^2 + \beta \sigma_{t-1}^2,$$

où

- les $\{z_t\}_{t=1}^T$ sont i.i.d. selon une fonction de densité $g(\cdot)$,
- le processus de volatilité est initialisé avec $\sigma_1^2 \equiv \omega/(1-\alpha-\beta)$.

On désire coder la **fonction de log-vraisemblance totale** :

$$\log f(y_1, \ldots, y_T) = \sum_{t=1}^{T} \left[\log g\left(\frac{y_t}{\sigma_t}\right) - \log \sigma_t \right].$$

R_garchNormal.R : approche standard en R

Supposons le modèle $\{z_t\}_{t=1}^{T} \overset{i.i.d.}{\sim} N(0,1)$, i.e., $g(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}}$

```
R_garchNormal = function(omega, alpha, beta, y)
{
  CST = -0.5 * log(2 * pi) # we compute the constant -1/2 log(2π)
  h = omega / (1 - alpha - beta) # initialize volatility: σ_1^2 ≡ ω/(1-α-β)
  llh = 0 # initialize the total log-likelihood
  for (t in 1:length(y)) {
    yt = y[t]
    zt = yt / sqrt(h) # we compute z_t = y_t/σ_t
    # we add log g(z_t) - log σ_t = -1/2 log(2π) - 1/2 z_t^2 - 1/2 log σ_t^2
    llh = llh + CST - 0.5 * zt * zt - 0.5 * log(h)
    h = omega + alpha * yt * yt + beta * h # update volatility
  }
  return(llh)
}
```

Rcpp_garchNormal.cpp : approche standard en Rcpp

La même fonction en C++ s'écrit ainsi :

```
double Rcpp_garchNormal(const double& omega, const double& alpha,
      const double& beta, const NumericVector& y)
{
  double CST = -0.5 * log(2 * M_PI);
  double h = omega / (1.0 - alpha - beta);
  double llh = 0.0;
  double yt, zt;
  for (int t = 0; t < y.size(); ++t) {
    yt = y[t];
    zt = yt / sqrt(h);
    llh += CST - 0.5 * zt * zt - 0.5 * log(h);
    h = omega + alpha * yt * yt + beta * h;
  }
  return llh;
}
```

Extension du problème de programmation

Supposons qu'on désire également programmer le modèle GARCH sous la distribution Laplace :

$$g(z) = \frac{1}{\sqrt{2}} e^{-\sqrt{2}|z|}, \qquad z \in \mathbb{R}.$$

On a maintenant deux modèles à programmer. On aimerait cependant programmer une seule fonction pour calculer la log-vraisemblance. I.e., on désire adopter une approche applicable lorsque le nombre de modèles est grand.

R_garchAny.R : approche naïve flexible en R

On crée une fonction pour calculer la log-densité $N(0,1)$:

```
## Normal innovations
CST.normal = -0.5 * log(2 * pi)
kernel.normal = function(z) CST.normal - 0.5 * z * z
```

On fait de même pour la loi Laplace :

```
## Laplace innovations
CST.laplace = -0.5 * log(2)
CST2.laplace = sqrt(2)
kernel.laplace = function(z) CST.laplace - CST2.laplace * abs(z)
```

R_garchAny.R (suite)

On programme ensuite la fonction de log-vraisemblance GARCH. Celle-ci possède l'argument **f.kernel** qui correspond à l'une des fonctions ci-haut.

```
R_garchAny = function(omega, alpha, beta, y, f.kernel)
{
  h = omega / (1 - alpha - beta) # initialize volatility
  llh = 0
  for (t in 1:length(y)) {
    yt = y[t]
    z = yt / sqrt(h) # compute innovation z_t = y_t/σ_t
    llh = llh + f.kernel(z) - 0.5 * log(h) # add log g(z_t) - 1/2 log(σ_t^2)
    h = omega + alpha * yt * yt + beta * h # update volatility
  }
  return(llh)
}
```

R_garchAny.R (discussion)

Dans cette approche, la fonction à utiliser (**kernel.normal** ou **kernel.laplace**) est inconnue lors de la compilation du code.

La fonction à utiliser est déterminée lors de l'exécution. Cela engendre des coûts supplémentaires à chaque appel de celle-ci. (Nous verrons plus tard l'impact de ces coûts.)

Une approche de programmation orientée objet permet de régler ce genre de problème.

Outline

Rcpp_garchAny.cpp : approche orientée objet en Rcpp

Dans cette approche, on crée des classes qui sont utilisées pour en créer d'autres.

On peut par exemple créer plusieurs variantes d'une même classe à l'aide d'arguments *templates*.

Cela permet de sauver beaucoup de lignes de code sans perdre en vitesse d'exécution.

Classe NormalKernel

On crée premièrement une classe qui contient une fonction pour calculer la log-densité $N(0,1)$.

```
class NormalKernel
{
  const double lncst_; // kernel constant defined as  -1/2 log(2π)

public:

   // constructor: set kernel constant
  NormalKernel() : lncst_(- 0.5 * log(2 * M_PI)) {};

  // returns -1/2 log(2π) - 1/2 z^2
  double computeKernel(const double& z) const
  {
    return lncst_ - 0.5 * z * z;
  }
};
```

Classe LaplaceKernel

On fait de même pour la loi Laplace.

```
class LaplaceKernel
{
  const double cst_; // kernel constant defined as √2
  const double lncst_; // kernel constant defined as −1/2 log(2)

public:
   // constructor: set kernel constants
  LaplaceKernel() : cst_(sqrt(2.0)), lncst_(− 0.5 ∗ log(2.0)) {};

  // returns −1/2 log(2) − √2|z|
  double computeKernel(const double& z) const
  {
    return lncst_ − cst_ ∗ fabs(z);
  }
};
```

Classe GarchAny

On déclare ensuite une classe pour calculer la log-vraisemblance du modèle GARCH(1,1). Celle-ci possède un argument template qui correspond au type d'une des deux classes ci-haut.

```
template <typename T>
class GarchAny
{
  T distribution_; // conditional distribution (templated)

public:
  GarchAny() {};

  // computes log-likelihood (defined next slide)
  double computeKernel(const double&, const double&, const double
      &, const NumericVector&);
};
```

Fonction computeKernel

Calcule la log-vraisemblance d'une série chronologique **y** sous le modèle GARCH avec paramètres **omega, alpha, beta**.

```
template <typename T>
double GarchAny<T>::computeKernel(const double& omega, const
    double& alpha, const double& beta, const NumericVector& y)
{
  double h = omega / (1.0 - alpha - beta); // initialize volatility
  double llh = 0.0;
  double yt, zt;
  for (int t = 0; t < y.size(); ++t) {
    yt = y[t]; // extract y_t
    zt = yt / sqrt(h); // compute z_t = y_t/σ_t
    // add: log g(z_t) - 1/2 log(σ_t^2)
    llh += distribution_.computeKernel(zt) - 0.5 * log(h);
    h = omega + alpha * yt * yt + beta * h; // update volatility
  }
  return llh;
}
```

On crée l'interface R avec les modules Rcpp

```
RCPP_MODULE(GarchAny)
{
  // GARCH-Normal
  class_< GarchAny <NormalKernel> >("GarchAnyNormal")
  .constructor()
  .method("computeKernel",
    &GarchAny<NormalKernel>::computeKernel)
  ;

  // GARCH-Laplace
  class_< GarchAny <LaplaceKernel> >("GarchAnyLaplace")
    .constructor()
    .method("computeKernel",
      &GarchAny<LaplaceKernel>::computeKernel)
  ;
}
```

Utilisation en R

Une fois le code **Rcpp_garchAny.cpp** compilé, on peut créer des fonctions R qui servent d'interface :

Rcpp_garchAnyNormal = new(GarchAnyNormal)$computeKernel

Rcpp_garchAnyLaplace = new(GarchAnyLaplace)$computeKernel

Étude de performance

On peut comparer la vitesse d'exécution des quatre approches.

Le temps est mesuré en microsecondes pour des séries de différentes longueurs T.

T	100	250	500	1000
R_garchNormal	277.9	655.1	1262.8	2814.6
R_garchAnyNormal	365.4	789.8	1452.4	3318.0
Rcpp_garchNormal	6.9	9.7	14.2	27.7
Rcpp_garchAnyNormal	5.4	9.9	14.1	27.8

Rcpp / C++ est beaucoup rapide que R ici.

L'approche flexible naïve engendre une perte de vitesse.

L'approche objet orientée n'engendre pas de perte de vitesse.

Librairie MSGARCH

Dans notre librairie, les modèles économétriques sont programmés grâce à une combinaison de plusieurs techniques de programmation orientée objet (e.g., *templates*, polymorphisme).

Cela permet de programmer cette très large famille de modèles sans perte d'efficacité et avec un code compact.

Application dans nos projets de recherche

Projet 1 : couverture de fonds distincts en présence de risque de base *(avec Frédéric Godin et Emmanuel Hamel)*[1]

Les codes furent utilisés pour estimer un modèle bivarié à changement de régime. Les actifs sont le TSX60 *futures* et un fonds distinct canadien.

Projet 2 : stratégies d'investissement pour CAT funds *(avec Van Son Lai et Frédéric Godin)*

Les codes sont utilisés pour estimer un modèle économétrique contenant des actifs financiers traditionnels ainsi que des fonds spécialisés dans les *insurance-linked securities*.

1. Projet en collaboration avec l'Autorité des Marchés Financiers et financé par le Fonds pour l'éducation et la saine gouvernance.

Outline

Conclusion

Rcpp facilite l'utilisation combinée de C++ et R.

Les boucles en C++ sont beaucoup plus rapides qu'en R.

La programmation orientée objet C++ permet de coder un grand nombre de modèles en simultané et sans perte d'efficacité.

Ces techniques sont utilisées dans la librairie MSGARCH (disponible sur le CRAN).

Références

Eddelbuettel, D., François, R., Allaire, J., Ushey, K., Kou, Q., Bates, D., and Chambers, J. (2017). *Rcpp : Seamless R and C++ Integration*. R package version 0.12.10.

Ardia, D., Bluteau, K., Boudt, K., Peterson, B., Trottier, D. -A. (2017). *MSGARCH : Markov-Switching GARCH Models*. R package version 0.17.7.