Bayesian estimation of GARCH models using Markov Chain Monte Carlo methods

STA3431 Final Project

Name: Tian Han **Guan**

Student ID: 998978058

Department: Statistics

Program: MSc

Email: tianhan.guan@mail.utoronto.ca

Abstract

In recent years, the autoregressive conditional heteroscedasticity (ARCH) model has become an important tool for modelling financial time series data, especially when the data exhibits some time-varying volatility clustering features. This project focuses on one particular category of the ARCH model, the generalized autoregressive conditional heteroscedasticity (GARCH) model, which is a popular choice in many financial econometric contexts. In particular, the project discusses how to fit a GARCH model using various Markov Chain Monte Carlo (MCMC) methods, as well as discussing the advantages and disadvantages between them and also comparing them to traditional fitting approaches such as maximum likelihood estimation (MLE) methods.

1. Introduction of the GARCH model

As an extension of the autoregressive conditional heteroscedasticity (ARCH) model, the generalized autoregressive conditional heteroscedasticity (GARCH) model is a common choice to model a financial time series which is believed to contain time-varying volatility clustering features. For example, stock prices usually have volatility clustering features, that is, have some periods of relative low volatility and followed by some periods of high volatility. In such case, a GARCH model is usually being used to model the stock price so that people can make reasonable predictions about future stock price.

Mathematically, the $GARCH(p,q)$ model to the time series data $\{y_1, y_2, \dots, y_n\}$ is defined as the following:

$$\boldsymbol{y_t = \ \sigma_t \varepsilon_t \ (1)}$$

$$where\ \sigma_t = volatility\ and\ \varepsilon_t = error\ or\ innovation\ term$$

Furthermore, $the\ volatility\ \sigma_t\ is\ defined\ by$

$$\boldsymbol{{\sigma_t}^2 = \omega + \sum_{i=1}^{q} \alpha_i {y_{t-i}}^2 + \sum_{i=1}^{p} \beta_i {\sigma_{t-i}}^2\ (2)}$$

$where\ all\ parameters\ \omega, \alpha_{1,} \dots, \alpha_q, \beta_{1,\dots,}\beta_p$ are **strictly positive**

$For\ example, in\ a\ GARCH(1,1)\ model,$ $\boldsymbol{{\sigma_t}^2 = \ \omega + \alpha {y_{t-1}}^2 + \beta {\sigma_{t-1}}^2\ (3)}$ and in a GARCH(2,2) model, $\boldsymbol{{\sigma_t}^2 = \ \omega + \alpha_1 {y_{t-1}}^2 + \alpha_2 {y_{t-2}}^2 + \beta_1 {\sigma_{t-1}}^2 + \beta_2 {\sigma_{t-2}}^2\ (4)}$

Note: in this project, for simplicity reasons, the following assumptions have been made:

1) The error terms ε_t follows i.i.d standard normal distribution, i.e. $\boldsymbol{\varepsilon_t \sim i.i.d\ N(0,1)}$

2) $\boldsymbol{\sum_{i=1}^{q} \alpha_i + \sum_{i=1}^{p} \beta_i < 1}$ **,** this is to make sure the stationarity condition of the time series

2. Estimation of the GARCH models

2.1 Pitfalls of the maximum likelihood estimation (MLE) approach

In a GARCH model, the unknown parameters $\boldsymbol{\theta} = (\omega, \alpha_{1,} \ldots, \alpha_q, \beta_{1,\ldots,}\beta_p)$ need to be estimated, and the most traditional way of doing so is by the maximum likelihood estimation (MLE) approach, which finds the parameters that maximize the likelihood function. Although the MLE approach is being used in most cases, it has the following disadvantages:

- The fitting results are very sensitive to the *initial starting values* and are very unstable in many cases
- A *very large* amount of data is needed in order to accurately estimate the parameters (while in many cases large amount of historical data is not available)

Overall, optimization of GARCH model parameters appears to be numerically difficult and has shown to be problematic in many scenarios. Therefore, other fitting methods such as Bayesian estimation using Markov Chain Monte Carlo (MCMC) algorithms have attracted many attentions in recent years.

2.2 Bayesian estimation using various Markov Chain Monte Carlo (MCMC) algorithms

First derive the *posterior density*:

a) Prior distribution of $\boldsymbol{\theta} = (\omega, \alpha_{1,} \ldots, \alpha_q, \beta_{1,\ldots,}\beta_p)$

-In this project, the prior distribution of $\boldsymbol{\theta}$ is assumed to follow a uniform distribution (which is very common in time series models). Therefore, $the\ prior\ distribution\ p(\boldsymbol{\theta}) = K, where\ K\ is\ some\ constant$

b) Likelihood function $L(\boldsymbol{y}|\boldsymbol{\theta}), where\ \boldsymbol{y} = (y_1, y_2, \dots, y_n)$

- $L(\boldsymbol{y}|\boldsymbol{\theta}) = \prod_{t=1}^{n} \frac{1}{\sqrt{2\pi\sigma_t{}^2}} e^{-\frac{y_t^2}{2\sigma_t{}^2}}$

c) Joint density function $f(\boldsymbol{\theta}, \boldsymbol{y})$

- $f(\boldsymbol{\theta}, \boldsymbol{y}) = L(\boldsymbol{y}|\boldsymbol{\theta})p(\boldsymbol{\theta}) = \boldsymbol{K}(\boldsymbol{2\pi})^{-\frac{n}{2}} \left(\prod_{t=1}^{n} \frac{1}{\sigma_t} e^{-\frac{y_t^2}{2\sigma_t{}^2}}\right)$

d) Posterior distribution $\pi(\boldsymbol{\theta}|\boldsymbol{y})$

- $\pi(\boldsymbol{\theta}|\boldsymbol{y}) \propto f(\boldsymbol{\theta}, \boldsymbol{y}) \Rightarrow \boldsymbol{\pi}(\boldsymbol{\theta}|\boldsymbol{y}) = \boldsymbol{C} \prod_{t=1}^{n} \frac{1}{\sigma_t} e^{-\frac{y_t^2}{2\sigma_t{}^2}}$**, where C is some constant**

-log$(\pi) = \log(C) - \frac{1}{2}\sum_{t=1}^{n} \log(\boldsymbol{\sigma_t}^2) - \frac{1}{2}\sum_{t=1}^{n} \frac{y_t^2}{\sigma_t{}^2}$ (to be used in MCMC algorithms)

2.2.1. Fit a GARCH(1,1) model using the MLE approach

Before applying the MLE approach, a dataset of size N=2,100 is being simulated using a GARCH(1,1) process with parameters $\boldsymbol{\theta} = (\omega, \alpha, \beta) = (0.8, 0.2, 0.5)$. A *burn-in size of n=100* is used to throw away the first 100 data points in order to reduce variability in the simulated dataset. Graph 1 in Appendix 2 shows the time series plot of the simulated GARCH(1,1) dataset. Several volatility clustering regions can be seen from the plot which represent key features of a GARCH process.

Now, fit the simulated GARCH(1,1) dataset using the MLE approach. This is done by using the **garchFit** function from R's **fGarch package.** In order to get an accurate estimate of the parameters, 4 different runs (with different MLE algorithms) were performed. The point estimates and standard errors are summarized in Table 1 in Appendix 2.

Conclusion:

By taking the average of the 4 runs, the point estimates of $\left(\boldsymbol{\omega}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right) = (\mathbf{0.93}, \mathbf{0.19}, \mathbf{0.45})$. The largest standard errors are (0.20, 0.03, 0.09). Compared to the true value $\left(\boldsymbol{\omega}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right) = (\mathbf{0.8}, \mathbf{0.2}, \mathbf{0.5})$, the first and third parameters are not very accurate, the absolute estimate error is **(0.13, 0.01, 0.05)** . This can be due to the fact that a *very large* data size is usually needed for the MLE optimization method to work out properly.

2.2.2. Fit a GARCH(1,1) model using the random-walk Metropolis algorithm

As one of the most commonly used MCMC method, a *random-walk Metropolis algorithm* is used to estimate the parameters of a GARCH(1,1) model using the posterior density derived before.

For the *random-walk Metropolis algorithm:*

1) Since the posterior π is very complicated, the code works with *log(π)* directly instead to avoid "overflow" problems [this is applied for all MCMC algorithms in this project]

2) A standard multivariate normal distribution proposal is used, as the error terms are assumed to follow i.i.d. standard normal distribution.

$$\textbf{Proposal density}:\ \mathbf{Y_n} \sim \boldsymbol{MVN}(\boldsymbol{X_{n-1}}, \boldsymbol{\sigma^2 I})$$

$Here\ \sigma\ is\ chosen\ to\ be\ \mathbf{0.05}\ so\ that\ a$ **a reasonable acceptance rate can be achieved**

3) Since all parameters are strictly positive, any simulated negative parameter will cause the sample to be rejected (this will reduce the acceptance rate, unfortunately) [this is applied for all MCMC algorithms in this project]

4) 40,000 runs are simulated with a burn-in period of 20,000 runs (a very large burn-in period is used as the MCMC may converge slowly in time)

The point estimates and standard errors (with 4 different runs) are summarized in Table 2 in Appendix 2.

Note: se=true standard error, calculated using the “varfact” method [for all MCMC algorithms]

Comments on the graphical outputs:

1) Time series plot (Graph 2 in Appendix 2):

The simulated samples for the 2nd parameter appear to converge very quickly, while the samples for the 1st and 3rd parameters have some fluctuations in the beginning but still managed to converge as time goes on. All 3 time series plots have no trapped regions/big jumps/trends.

2) Autocorrelation plot (Graph 3 in Appendix 2):

All 3 autocorrelation plots have an exponential decay trend of the lags, which means that the generated samples should be relatively independent when the sample size is large.

<u>Conclusion:</u>

By taking the average of the 4 runs, the point estimates of $(\boldsymbol{\omega}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = (\mathbf{0.84}, \mathbf{0.21}, \mathbf{0.48})$. The largest standard errors are (0.012, 0.0016, 0.005) which is much smaller than the MLE standard errors (showing the random-walk Metropolis gives more stable results than the MLE approach). Compared to the true value, the absolute estimate error is **(0.04, 0.01, 0.02)** which can be considered as a good estimate given the limited data size and the complex nature of the GARCH(1,1) process. Compared to the MLE approach, the random-walk Metropolis algorithm gives a much more accurate estimation of the parameters with much smaller standard errors.

2.2.3. Fit a GARCH(1,1) model using the Adaptive-Metropolis algorithm

One way to improve the Metropolis algorithm is by tuning the estimated parameters as the Markov chain proceeds in time. In order to do so, the Adaptive-Metropolis algorithm considers the covariance matrix Σ in each proposal move, and this algorithm usually creates better results than the normal Metropolis algorithm.

For the *Adaptive-Metropolis algorithm:*

1) The following proposal density is being used:

$$\mathbf{Y_n} \sim \boldsymbol{MVN}(\boldsymbol{X_{n-1}}, \boldsymbol{c}\mathbf{\Sigma_n} + \boldsymbol{\varepsilon I})$$

$\boldsymbol{where\ c = 0.1, \varepsilon = 0.01, \Sigma_n = updated\ empirical\ covariance\ matrix, I = identity\ matrix}$

(The value c and $\boldsymbol{\varepsilon}$ are chosen so that a reasonable acceptance rate can be achieved)

2) Again 40,000 runs are simulated with a burn-in period of 20,000 runs

In order to get an accurate estimate of the parameters, 4 different runs were performed. The point estimates and standard errors are summarized in Table 3 in Appendix 2.

Comments on the graphical outputs:

1) Time series plot (Graph 4 in Appendix 2):

Similar to the random-walk Metropolis case, the samples for all 3 parameters converge quickly.

2) Autocorrelation plot (Graph 5 in Appendix 2):

All 3 autocorrelation plots have an exponential decay trend of the lags, which means that the generated samples should be relatively independent when the sample size is large.

Conclusion:

By taking the average of the 4 runs, the point estimates of $\left(\boldsymbol{\omega}, \boldsymbol{\alpha}, \boldsymbol{\beta}\right) = (\mathbf{0.81}, \mathbf{0.19}, \mathbf{0.51})$. The largest standard errors are (0.016, 0.0026, 0.0073) which is much smaller than the MLE standard errors but slightly larger than that of the Metropolis algorithm. Compared to the true value of the parameters, the absolute estimate error is **(0.01, 0.01, 0.01)** which is a very good estimate given the limited data size and the complex nature of the GARCH(1,1) process.

Even though the Adaptive-Metropolis algorithm seems to perform better than the Metropolis algorithm, the following items should be noted:

1) Since this algorithm is non-Markovian and does not preserve stationarity at each step, some additional checks need to be done to make sure the MCMC converges to the true stationary distribution. One simple check is to simulate a GARCH time series using the MCMC estimated parameters and plot it against the true data to see if they are considerably similar (this will be illustrated in the GARCH(2,2) example later on).

2) The speed of generating samples will decrease along the process, as the estimation of the empirical covariance-matrix gets slower and slower as more samples are accumulated. Overall,

for the same number of runs, the Adaptive-Metropolis algorithm takes much longer time than the Metropolis algorithm.

2.2.4. Fit a GARCH(1,1) model using the Parallel Tempered Metropolis algorithm

One way to attack problems where the target distribution is multi-modal is to use the Tempered MCMC. Here because the constant C_τ of the tempered density π_τ is unknown, the Parallel Tempered algorithm is used. It is hoped that by introducing a "cooling schedule" $\{\tau_n\}$, the Markov chain can move better between different modes and therefore give better estimations.

For the *Parallel Tempered Metropolis algorithm*

1) The temperature $\tau = \{1,2,3,4,5,6,7,8,9,10\}$

2) Update the Markov chain using the proposal:

$Y_n \sim MVN(X_{n-1}, \sigma^2 I), i.e. Metropolis\ algorithm$

3) Due to the increased computational burden caused by the add-in of the different temperatures, only 10,000 runs are simulated with a burn-in period of 1,000 runs

The point estimates and standard errors (with 4 different runs) are summarized in Table 4 in Appendix 2.

Comments on the graphical outputs:

1) Time series plot (Graph 6 and 7 in Appendix 2):

By comparing the time series plots of $\tau = 1 \ vs \ \tau = 5$, we can see that the Parallel Tempered Metropolis MCMC converges very quickly when temperatures cool down towards 1 compared to the other 2 MCMC methods.

2) Autocorrelation plot (Graph 8 in Appendix 2):

All 3 autocorrelation plots have very fast exponential decay trend of the lags, which means that the generated samples should be independent when the sample size is large. Also, notice that the decaying rate is faster than that of both the random-walk Metropolis and Adaptive-Metropolis methods, showing that the samples are more independent than the samples generated by the other 2 methods.

Conclusion:

By taking the average of the 4 runs, the point estimates of $(\boldsymbol{\omega}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = (\mathbf{0.83}, \mathbf{0.21}, \mathbf{0.49})$. The largest standard errors are (0.016, 0.017, 0.01) which are very small considering the relative small run size of the algorithm. Compared to the true value of the parameters, the estimation is again accurate with an absolute estimate error of **(0.03, 0.01, 0.01),** which is better than the Metropolis algorithm but worse than the Adaptive-Metropolis algorithm.

Remarks:

Even though the Parallel Tempered Metropolis converges very quickly and generates more independent samples than the other 2 MCMC algorithms, one thing to keep in mind is that this MCMC takes much longer time than the previous 2 methods as many temperatures need to be run (in this case, 10 different temperatures) in parallel. Therefore, in case where the GARCH model has more complicated structures than the simplest GARCH(1,1) case, the Parallel

Tempered MCMC will become computationally inefficient compared to the Adaptive-Metropolis algorithm which also generates very accurate fittings for the model.

2.2.5. Summary of different fitting methods for the GARCH(1,1) model

Table 5 in Appendix 2 summarizes the 4 different fitting methods for the generated GARCH(1,1) model on an artificially generated 2,000 data points. By comparing the 4 different approaches, I believe the **Adaptive-Metropolis** is the **best algorithm** to use because:

1) It gives the best point estimates of the parameters.
2) The computational speed is acceptable and the standard errors of the estimates are small.

One thing to note is that some additional checks need to be done to make sure the MCMC converges to the true stationary distribution (which will be illustrated in the next example).

2.2.6. Fit a GARCH(2,2) model using the Adaptive-Metropolis algorithm

In this section, we try to fit a GARCH(2,2) process with parameters $\boldsymbol{\theta} = (\omega, \alpha_1, \alpha_2, \beta_1, \beta_2) = (0.8, 0.1, 0.25, 0.15, 0.3)$. Here, only N=1,100 data points are being simulated artificially using the GARCH(2,2) model and a *burn-in size of n=100* is used.

A time series plot of the simulated dataset can be found in Appendix 2 (Graph 9).

For the Adaptive-Metropolis algorithm, the same proposal distribution and number of runs are used as in the GARCH(1,1) example. The point estimates and standard errors (with 4 different runs) are summarized in Table 6 in Appendix 2.

Comments on the graphical outputs:

1) Time series plot (Graph 10 in Appendix 2):

The samples for parameters 2 and 3 converge quickly in time, while the samples for parameters 1,4, and 5 have bigger fluctuations but are still reasonable to accept (given the complexity of the model and the small data size for fitting).

2) Autocorrelation plot (Graph 11 in Appendix 2):

All 5 autocorrelation plots have an exponential decay trend of the lags, which means that the generated samples should be relatively independent when the sample size is large.

<u>**Conclusion:**</u>

By taking the average of the 4 runs, the point estimates of $(\boldsymbol{\omega}, \boldsymbol{\alpha_1}, \boldsymbol{\alpha_2}, \boldsymbol{\beta_1}, \boldsymbol{\beta_2}) = (\mathbf{0.82}, \mathbf{0.12}, \mathbf{0.23}, \mathbf{0.16}, \mathbf{0.28})$. The largest standard errors are (0.038,0.0044,0.0055,0.010,0.014). Compared to the true value $(\boldsymbol{\omega}, \boldsymbol{\alpha_1}, \boldsymbol{\alpha_2}, \boldsymbol{\beta_1}, \boldsymbol{\beta_2}) = (\mathbf{0.8}, \mathbf{0.1}, \mathbf{0.25}, \mathbf{0.15}, \mathbf{0.3})$, the absolute estimate error is **(0.02,0.02,0.02,0.01,0.02)** which shows the estimate is very accurate.

Visual check of the Adaptive-Metropolis algorithm

As mentioned in the previous section, given that the Adaptive-Metropolis algorithm is non-Markovian, it is important to make sure that the MCMC actually converges to the correct target distribution. Here, 1,000 data points are generated using the estimated parameters from the Adaptive-Metropolis algorithm and are compared to the true data. From Graph 12 in Appendix 2, it can be seen that the MCMC should converge to the target stationary distribution as the simulated data has very similar pattern to the original data series. In addition, the error terms are relatively small and random, which shows the estimation should be reasonably correct.

2.2.7. Final remarks on fitting a general GARCH(p,q) model using MCMC algorithms

In summary, this project illustrates 3 different MCMC algorithms to fit an example GARCH(1,1) model, namely, 1)Metropolis algorithm 2)Adaptive-Metropolis algorithm and 3)Parallel Tempered Metropolis algorithm. By comparing the fitting results to the MLE approach, we find that all 3 MCMC algorithms perform very well and among them, the Adaptive-Metropolis is the best algorithm to use. Next, the Adaptive-Metropolis algorithm is used to fit an example GARCH(2,2) model and the result also shows that the MCMC algorithm can be used to efficiently estimate the parameters of the model. In conclusion, it is not hard to believe that MCMC algorithms should also work efficiently for general GARCH(p,q) models, and they can be good alternatives for fitting such models when the MLE method does not give good estimations.

In this project, the dataset is kept at a relative small size (N=2,000 and N=1,000) to be used in the fitting, where the MLE method generally has problems. However, it should be noted that in case where the dataset is very large (e.g. N=10,000), then the MLE method usually gives much more accurate estimations. On the other hand, a MCMC algorithm may not be a good choice to use due to their slow speed and the little benefit on the improved accuracy of the estimations. Therefore, MCMC algorithms are more useful in case where there is only a limited amount of data.

Appendix 1: R code

##The following code is used to simulate a GARCH(1,1) model with N=2,100 data points

##with a burn-in period of n=100 data points

```
library(fGarch)
library(mvtnorm)
N <- 2100  #Run size
n <- 100  #Burn-in size
a <- c(0.8, 0.2, 0.5)  #GARCH(1,1) parameters theta=(omega,alpha,beta), dim=3
# Generate the GARCH(1,1) process iteratively by its definition
garch_11 = function(params,size) {
        y <- double(N)  #Actual data
        s <-double(N) #Volatility: s=sigma^2
        #define the starting distribution for y1 and s1
        y[1] <- 0.1
        s[1] <- 0.01
        set.seed(1234)
        e <- rnorm(N)  #Assume errors~Normal(0,1)
        for(i in 2:N)  {
                s[i] <- params[1]+params[2]*y[i-1]^2+params[3]*s[i-1]
                y[i] <- sqrt(s[i])*e[i]
        }
        return(y)
}
#Simulated true data of the previously defined GARCH(1,1) process
ygarch <- garch_11(a,N)[(n+1):N]
#Plot the GARCH(1,1) time series
plot(ygarch,type='l',main="Time series plot of the simulated GARCH(1,1) process")
```

##The following code fits the true data to a GARCH(1,1) process using the MLE approach

##(using R's garchFit function)

```
mle_garch <- garchFit(data=ygarch, algorithm = "lbfgsb+nm")
```

```
mle_garch@fit$matcoef
```

##The following code fits the true data to a GARCH(1,1) process using various MCMC algorithm

#Iteratively calculate the logged posterior distribution

```
g0 = function(s,y) { return( -0.5*log(s)-0.5*(y^2/s) ) }

g =  function(x) {

        #define the starting distribution for s1

        s <- 0.01

        gg <- g0(s,ygarch[1])

        for(i in 2:length(ygarch))  {

        s <- x[1]+x[2]*ygarch[i-1]^2+x[3]*s

        gg <- gg+g0(s,ygarch[i])

        }

        return(gg)

}
```

##a)The Metropolis algorithm

```
M = 40000  # run length

B = 20000  # amount of burn-in

X = c(runif(1),runif(1),runif(1))  # overdispersed starting distribution (dim=3)

sigma = 0.05  # proposal scaling

x1list = x2list = x3list = rep(0,M)  # for keeping track of values

numaccept = 0;

for (i in 1:M) {

  Y = X + sigma * rnorm(3)      #Proposal move (dim=3)

  U = log(runif(1))  # for accept/reject

  alpha = g(Y) - g(X)  # for accept/reject

  if ( (sum(Y>0)==3) && U < alpha) { #Reject all samples where any parameter is negative

        X = Y  # accept proposal

    numaccept = numaccept + 1;
  }
  x1list[i] = X[1];
  x2list[i] = X[2];
  x3list[i] = X[3];
  # Output progress report.
  if ((i %% 1000) == 0)
    cat (" ...",i);
}
```

##Produce the outputs (acceptance rate, standard errors, time series plot, autocorrelation plot)

```
cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
cat("acceptance rate =", numaccept/M, "\n");
u = mean(x1list[(B+1):M])
cat("mean of x1 is about", u, "\n")
v = mean(x2list[(B+1):M])
cat("mean of x2 is about", v, "\n")
w = mean(x3list[(B+1):M])
cat("mean of x3 is about", w, "\n")
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
se11 =  sd(x1list[(B+1):M]) / sqrt(M-B)
thevarfact1 = varfact(x1list[(B+1):M])
se1 = se11 * sqrt( thevarfact1 )
cat("true standard error of omega is about", se1, "\n")
se12 =  sd(x2list[(B+1):M]) / sqrt(M-B)
thevarfact2 = varfact(x2list[(B+1):M])
se2 = se12 * sqrt( thevarfact2 )
cat("true standard error of alpha is about", se2, "\n")
se13 =  sd(x3list[(B+1):M]) / sqrt(M-B)
thevarfact3 = varfact(x3list[(B+1):M])
se3 = se13 * sqrt( thevarfact3 )
cat("true standard error of beta is about", se3, "\n")
```

```
par(mfrow=c(1,3))
plot(x1list,type='l',main="Time series plot of omega")
plot(x2list,type='l',main="Time series plot of alpha")
plot(x3list,type='l',main="Time series plot of beta")
par(mfrow=c(1,3))
acf(x1list,lag.max=250, main="Autocorrelation plot of omega")
acf(x2list,lag.max=250, main="Autocorrelation plot of alpha")
acf(x3list,lag.max=250, main="Autocorrelation plot of beta")
```

##b) The Adaptive-Metropolis algorithm

```
dim = 3
identity = diag(dim)
M = 40000  # run length
B = 20000  # amount of burn-in
X = c(runif(1),runif(1),runif(1))  # overdispersed starting distribution, dim=3
x1list = x2list = x3list = rep(0,M)  # for keeping track of chain values
Xveclist = matrix( rep(0,M*dim), ncol=dim)  # full chain values: contains all 3 dimensions
numaccept = 0;
epsilon = 0.01;
mult = 0.1;
for (i in 1:M) {
  #update the covariance matrix
  if (i < dim^2) {
    propSigma = identity
  }
  else {
    covsofar = cov( Xveclist[ (1:(i-1)) ,] )
    propSigma = mult * covsofar + epsilon * identity
  }
```

```
    #Metropolis algorithm
    Y = X + rmvnorm(1, sigma = propSigma)  # proposal value with dim=3
    U = log(runif(1))  # for accept/reject
    alpha = g(Y) - g(X)  # for accept/reject
    if ( (sum(Y>0)==3) && U < alpha) { #Reject samples where any parameter is negative
            X = Y  # accept proposal
        numaccept = numaccept + 1;
    }
    x1list[i] = X[1];
    x2list[i] = X[2];
    x3list[i] = X[3];
    Xveclist[i,] = X;
    # Output progress report.
    if ((i %% 1000) == 0)
      cat (" ...",i);
}
```

##Produce the outputs (acceptance rate, standard errors, time series plot, autocorrelation plot)

```
cat("acceptance rate =", numaccept/M, "\n");
u = mean(x1list[(B+1):M])
cat("mean of x1 is about", u, "\n")
v = mean(x2list[(B+1):M])
cat("mean of x2 is about", v, "\n")
w = mean(x3list[(B+1):M])
cat("mean of x3 is about", w, "\n")
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
se11 =  sd(x1list[(B+1):M]) / sqrt(M-B)
thevarfact1 = varfact(x1list[(B+1):M])
se1 = se11 * sqrt( thevarfact1 )
cat("true standard error of omega is about", se1, "\n")
```

```
se12 =  sd(x2list[(B+1):M]) / sqrt(M-B)
thevarfact2 = varfact(x2list[(B+1):M])
se2 = se12 * sqrt( thevarfact2 )
cat("true standard error of alpha is about", se2, "\n")
se13 =  sd(x3list[(B+1):M]) / sqrt(M-B)
thevarfact3 = varfact(x3list[(B+1):M])
se3 = se13 * sqrt( thevarfact3 )
cat("true standard error of beta is about", se3, "\n")
par(mfrow=c(1,3))
plot(x1list,type='l',main="Time series plot of omega")
plot(x2list,type='l',main="Time series plot of alpha")
plot(x3list,type='l',main="Time series plot of beta")
par(mfrow=c(1,3))
acf(x1list,lag.max=250,main="Autocorrelation plot of omega")
acf(x2list,lag.max=250,main="Autocorrelation plot of alpha")
acf(x3list,lag.max=250,main="Autocorrelation plot of beta")
```

##c)The Parallel Tempered Metropolis algorithm

#Define the tempered logged posterior distribution

```
g2 = function(x, thetemp) {
  if ( (thetemp<1) || (thetemp>maxtemp) )
    return(0.0)
  else
    return( (1/thetemp)*g(x) )
}
M = 10000  # run length
B = 1000 #bun-in size
sigma = 0.1  # proposal scaling
maxtemp = 5
x1list = x2list = x3list = matrix( rep(0,M*maxtemp), ncol=maxtemp )  # for chain values
```

```
numxaccept = rep(0,maxtemp)
numtempaccept = 0;
X = rbind(t(runif(maxtemp)),t(runif(maxtemp)),t(runif(maxtemp)))  # over dispersed starting distribution
for (i in 1:M) {
 for (temp in 1:maxtemp) {
   # PROPOSED X[,temp] MOVE
   Y = X[,temp] + sigma * rnorm(1)  # proposal move
   U = log(runif(1))  # for accept/reject
   A = g2(Y,temp) - g2(X[,temp],temp)  # for accept/reject
   if ( (sum(Y>0)==3) && U < A) {#Reject samples where any parameter is negative
         X[,temp] = Y  # accept proposal
     numxaccept[temp] = numxaccept[temp] + 1;
   }
 }
 # PROPOSED TEMP SWAP
   j = floor(1+runif(1,0,maxtemp))  # uniform on {1,2,...,maxtemp}
   k = floor(1+runif(1,0,maxtemp))  # uniform on {1,2,...,maxtemp}
   U = log(runif(1))  # for accept/reject
   A = g2(X[,j],k) + g2(X[,k],j) - g2(X[,j],j) - g2(X[,k],k);
   if ((sum(Y>0)==3) && U < A) { #Reject samples where any parameter is negative
         # accept proposed swap
         tmpval = X[,j];
         X[,j] = X[,k];
         X[,k] = tmpval;
     numtempaccept = numtempaccept + 1;
   }
 x1list[i,] = X[1,];
 x2list[i,] = X[2,];
 x3list[i,] = X[3,];
   # Output progress report.
```

```
  if ((i %% 500) == 0)
    cat (" ...",i);
}
```

##Produce the outputs (acceptance rate, standard errors, time series plot, autocorrelation plot)

```
valx1list = x1list[((B+1):M),1];
valx2list = x2list[((B+1):M),1];
valx3list = x3list[((B+1):M),1];
cat("x acceptance rate =", numxaccept/M, "\n");
cat("temp acceptance rate =", numtempaccept/M, "\n");
u = mean(valx1list)
cat("mean of x1 is about", u, "\n")
v = mean(valx2list)
cat("mean of x2 is about", v, "\n")
w = mean(valx3list)
cat("mean of x3 is about", w, "\n")
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
se11 =  sd(valx1list) / sqrt(length(valx1list))
thevarfact1 = varfact(valx1list)
se1 = se11 * sqrt(thevarfact1)
cat("true standard error of omega is about", se1, "\n")
se12 =  sd(valx2list) / sqrt(length(valx2list))
thevarfact2 = varfact(valx2list)
se2 = se12 * sqrt(thevarfact2)
cat("true standard error of alpha is about", se2, "\n")
se13 =  sd(valx3list) / sqrt(length(valx3list))
thevarfact3 = varfact(valx3list)
se3 = se13 * sqrt(thevarfact3)
cat("true standard error of beta is about", se3, "\n")
par(mfrow=c(1,3))
plot(x1list[,1],type='l',main="Time series plot of omega with tau=1")
plot(x2list[,1],type='l',main="Time series plot of alpha with tau=1")

plot(x3list[,1],type='l',main="Time series plot of beta with tau=1")

par(mfrow=c(1,3))

plot(x1list[,5],type='l',main="Time series plot of omega with tau=5")

plot(x2list[,5],type='l',main="Time series plot of alpha with tau=5")

plot(x3list[,5],type='l',main="Time series plot of beta with tau=5")

par(mfrow=c(1,3))

acf(x1list[,1],lag.max=250,main="Autocorrelation plot of omega with tau=1")

acf(x1list[,1],lag.max=250,main="Autocorrelation plot of alpha with tau=1")

acf(x1list[,1],lag.max=250,main="Autocorrelation plot of beta with tau=1")
```

##The following code is used to simulate a GARCH(2,2) model with N=1,000 data points

##with a burn-in of n=100 data points

```
N <- 1100  #Run size

n <- 100  #Burn-in size

a <- c(0.8, 0.1,0.25,0.15,0.3)  #GARCH(2,2) parameters theta=(omega,alpha1,alpha2,beta1,beta2), dim=5
```

# Generate the GARCH(2,2) process iteratively by its definition

```
garch_22 = function(params,size) {

            y <- double(N)  #Actual data

            s <-double(N) #Volatility: s=sigma^2

            #define the starting distribution for y1 and s1

            y[1] <- 0.1

            s[1] <- 0.01

            y[2] <- 0.1

            s[2] <- 0.01

            set.seed(12345)

            e <- rnorm(N)  #Assume errors~Normal(0,1)

            for(i in 3:N)  {

                        s[i] <- params[1]+params[2]*y[i-1]^2+params[3]*y[i-2]^2+params[4]*s[i-
1]+params[5]*s[i-2]

                        y[i] <- sqrt(s[i])*e[i]
```

```
        }
        return(y)
}
#Simulated true data of the previously defined GARCH(1,1) process
ygarch <- garch_22(a,N)[(n+1):N]
plot(ygarch,type='l',main="Time series plot of the simulated GARCH(2,2) process")
```

## Fit a GARCH(2,2) model using the Adaptive-Metropolis algorithm

#Iteratively calculate the logged posterior distribution

```
g0 = function(s,y) { return( -0.5*log(s)-0.5*(y^2/s) ) }
g =  function(x) {
        #define the starting distribution for s1 and s2
        s <- double(N)
        s[1] <- 0.01
        s[2] <- 0.01
        gg <- g0(s[1],ygarch[1]) + g0(s[2],ygarch[2])
        for(i in 3:length(ygarch))  {
        s[i] <- x[1]+x[2]*ygarch[i-1]^2+x[3]*ygarch[i-2]^2+x[4]*s[i-1]+x[5]*s[i-2]
        gg <- gg+g0(s[i],ygarch[i])
        }
        return(gg)
}
dim = 5
identity = diag(dim)
M = 40000  # run length
B = 20000  # amount of burn-in
X = c(runif(1),runif(1),runif(1),runif(1),runif(1))  # overdispersed starting distribution, dim=5
x1list = x2list = x3list = x4list = x5list = rep(0,M)  # for keeping track of chain values
Xveclist = matrix( rep(0,M*dim), ncol=dim)  # full chain values: contains all 5 dimensions
numaccept = 0;
```

```
epsilon = 0.01;
mult = 0.1;
for (i in 1:M) {
    #update the covariance matrix
    if (i < dim^2) {
      propSigma = identity
    }
    else {
      covsofar = cov( Xveclist[ (1:(i-1)) ,] )
      propSigma = mult * covsofar + epsilon * identity
    }
    #Metropolis algorithm
    Y = X + rmvnorm(1, sigma = propSigma)  # proposal value with dim=5
    U = log(runif(1))  # for accept/reject
    alpha = g(Y) - g(X)  # for accept/reject
    if ( (sum(Y>0)==5) && U < alpha) { #Reject samples where any parameter is negative
             X = Y  # accept proposal
        numaccept = numaccept + 1;
    }
    x1list[i] = X[1];
    x2list[i] = X[2];
    x3list[i] = X[3];
    x4list[i] = X[4];
    x5list[i] = X[5];
    Xveclist[i,] = X;
    # Output progress report.
    if ((i %% 1000) == 0)
      cat (" ...",i);
}
```

##Produce the outputs (acceptance rate, standard errors, time series plot, autocorrelation plot)

```
cat("acceptance rate =", numaccept/M, "\n");

u = mean(x1list[(B+1):M])

cat("mean of x1 is about", u, "\n")

v = mean(x2list[(B+1):M])

cat("mean of x2 is about", v, "\n")

w = mean(x3list[(B+1):M])

cat("mean of x3 is about", w, "\n")

y = mean(x4list[(B+1):M])

cat("mean of x4 is about", y, "\n")

z = mean(x5list[(B+1):M])

cat("mean of x5 is about", z, "\n")

varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }

se11 =  sd(x1list[(B+1):M]) / sqrt(M-B)

thevarfact1 = varfact(x1list[(B+1):M])

se1 = se11 * sqrt( thevarfact1 )

cat("true standard error of omega is about", se1, "\n")

se12 =  sd(x2list[(B+1):M]) / sqrt(M-B)

thevarfact2 = varfact(x2list[(B+1):M])

se2 = se12 * sqrt( thevarfact2 )

cat("true standard error of alpha is about", se2, "\n")

se13 =  sd(x3list[(B+1):M]) / sqrt(M-B)

thevarfact3 = varfact(x3list[(B+1):M])

se3 = se13 * sqrt( thevarfact3 )

cat("true standard error of beta is about", se3, "\n")

se14 =  sd(x4list[(B+1):M]) / sqrt(M-B)

thevarfact4 = varfact(x4list[(B+1):M])

se4 = se14 * sqrt( thevarfact4 )

cat("true standard error of beta is about", se4, "\n")

se15 =  sd(x5list[(B+1):M]) / sqrt(M-B)

thevarfact5 = varfact(x5list[(B+1):M])
```

```
se5 = se15 * sqrt( thevarfact5 )
cat("true standard error of beta is about", se5, "\n")
par(mfrow=c(2,3))
plot(x1list,type='l',main="Time series plot of omega")
plot(x2list,type='l',main="Time series plot of alpha1")
plot(x3list,type='l',main="Time series plot of alpha2")
plot(x4list,type='l',main="Time series plot of beta1")
plot(x5list,type='l',main="Time series plot of beta2")
par(mfrow=c(2,3))
acf(x1list,lag.max=550,main="Autocorrelation plot of omega")
acf(x2list,lag.max=550,main="Autocorrelation plot of alpha1")
acf(x3list,lag.max=550,main="Autocorrelation plot of alpha2")
acf(x4list,lag.max=550,main="Autocorrelation plot of beta1")
acf(x5list,lag.max=550,main="Autocorrelation plot of beta2")
```

##Visual comparison of the original data vs simulated data using the Adaptive-Metropolis algorithm

```
#Simulate the MCMC-fitted GARCH(2,2) process
mcmc <- c(0.82,0.12,0.23,0.16,0.28)
mcmcgarch <- garch_22(mcmc,N)[(n+1):N]
diff=mcmcgarch-ygarch


#Plot the time series plot of the MCMC-fitted data vs true data
windows()
par(mfrow=c(3,1))
plot(ygarch,type='l',main="GARCH(2,2) of true data")
plot(mcmcgarch,type='l',main="GARCH(2,2) of simulated data using the Adaptive-Metropolis
algorithm")
plot(diff,type='l',main="Error")
```

Appendix 2: Tables and Graphs

All the tables and graphs that are referred in the main section can be found in this appendix.

Table 1. Fit a GARCH(1,1) model using the MLE approach (sample size of 2,000)

	ω	α	β
Run #1	1.10631336 (se= 0.20086509)	0.20771032 (se= 0.03260278)	0.36780409 (se= 0.08942434)
Run #2	0.862625973 (se= 0.16573902)	0.187800621 (se= 0.03258179)	0.480477406 (se= 0.07946264)
Run #3	0.790441778 (se= 0.16956516)	0.160320902 (se=0.02784665)	0.517083266 (se= 0.08190561)
Run #4	0.94104412 (se=0.18443978)	0.19536607 (se=0.03107434)	0.44089570 (se=0.08475960)

Table 2. Fit a GARCH(1,1) model using the random-walk Metropolis algorithm

	ω	α	β
Run #1 (acceptance rate = 0.18875)	0.8514435 (se= 0.01044993)	0.18875 (se=0.00111812)	0.4762199 (se= 0.004783638)
Run #2 (acceptance rate = 0.194525)	0.7976029 (se=0.007805497)	0.238634 (se=0.001586949)	0.4807624 (se=0.003490197)
Run #3 (acceptance rate = 0.194075)	0.920548 (se=0.01182069)	0.192264 (se=0.001104906)	0.4740603 (se=0.00500694)
Run #4 (acceptance rate= 0.19795)	0.8095317 (se=0.007218881)	0.2394162 (se=0.001171542)	0.4944903 (se= 0.003057131)

Table 3. Fit a GARCH(1,1) model using the Adaptive-Metropolis algorithm

	ω	α	β
Run #1 (acceptance rate = 0.07)	0.8444606 (se= 0.01150428)	0.1897836 (se= 0.002026295)	0.504239 (se= 0.005337938)
Run #2 (acceptance rate = 0.060175)	0.809481 (se= 0.01588055)	0.184765 (se= 0.002570107)	0.5195663 (se= 0.007280295)
Run #3 (acceptance rate = 0.0613)	0.7799155 (se= 0.01354061)	0.1822626 (se= 0.002253097)	0.5216622 (se= 0.006449901)
Run #4 (acceptance rate= 0.071)	0.8226512 (se= 0.01483221)	0.1976596 (se= 0.002220627)	0.4806246 (se= 0.00650116)

Table 4. Fit a GARCH(1,1) model using the Parallel Tempered Metropolis algorithm

	ω	α	β

Run #1 (temp acceptance rate = 0.2482)	0.8660317 (se= 0.008567103)	0.2199919 (se= 0.006085087)	0.4518535 (se= 0.007961508)
Run #2 (temp acceptance rate = 0.1584)	0.7336208 (se= 0.01588055)	0.218299 (se= 0.002570107)	0.5166585 (se= 0.007280295)
Run #3 (temp acceptance rate = 0.1469)	0.8166021 (se= 0.0061563311)	0.1625378 (se= 0.004187944)	0.4616362 (se= 0.005807854)
Run #4 (temp acceptance rate = 0.3014)	0.8965856 (se= 0.002866034)	0.2420393 (se= 0.01675526)	0.51986 (se= 0.009736792)

Table 5. Compare the 4 different methods for fitting the GARCH(1,1) model on a data set of 2,000 values

Method	**Absolute estimate error**	**Largest standard error**	**Convergence rate**	**Independence of generated samples**	**Time needed**
1)MLE	**(0.13, 0.01, 0.05)**	(0.20, 0.03, 0.09)	Very fast	Not being observed	Very short
2)Random-walk Metropolis	**(0.04, 0.01, 0.02)**	(0.012, 0.0016, 0.005)**	Medium	Independent when sample size is large	Short
3)Adaptive-Metropolis	**(0.01, 0.01, 0.01)**	(0.016, 0.0026, 0.0073)**	Medium	Independent when sample size is large	Medium
4)Parallel Tempered Metropolis	**(0.03, 0.01, 0.01)**	(0.016, 0.017, 0.01)*	Fast	Independent when sample size is large	Long

*10,000 samples are simulated

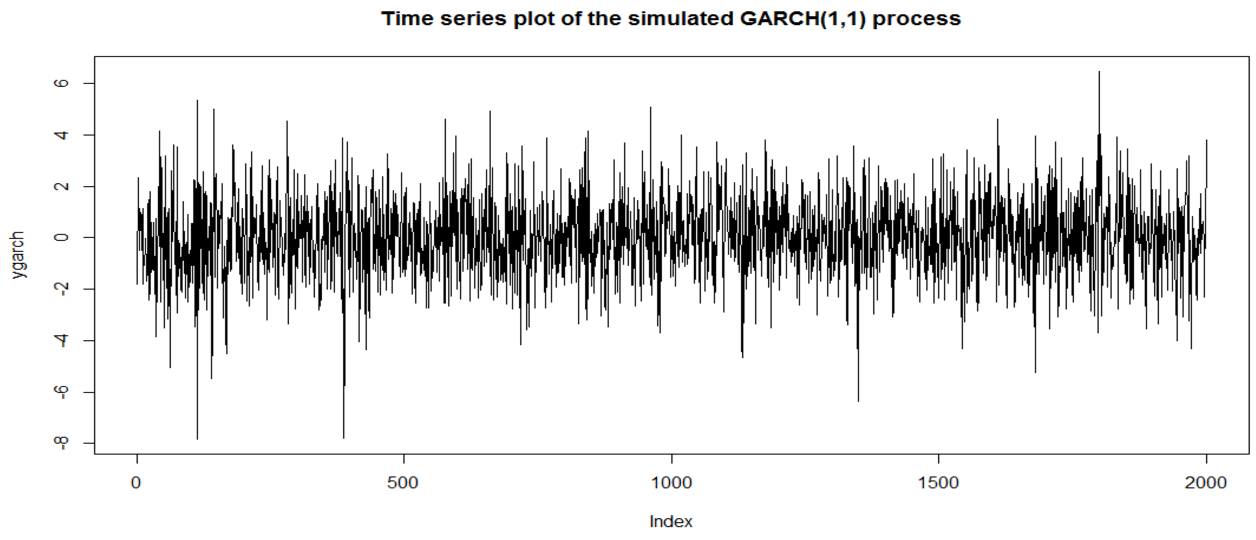
**40,000 samples are simulated

Table 6. Fit a GARCH(2,2) model using the Adaptive-Metropolis algorithm (sample size of 1,000)

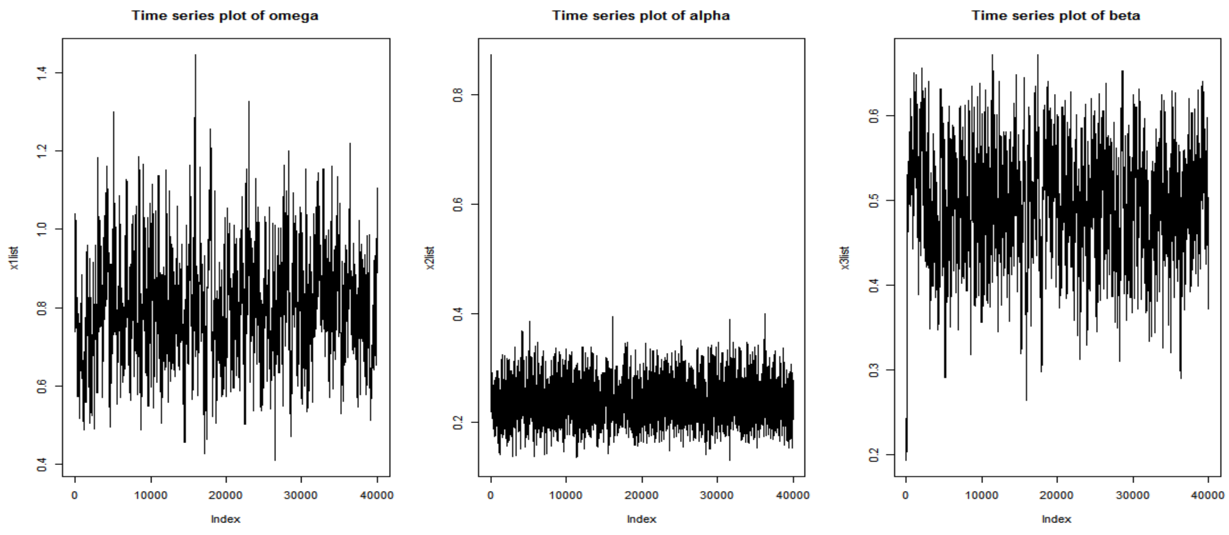
	$\boldsymbol{\omega}$	$\boldsymbol{\alpha_1}$	$\boldsymbol{\alpha_2}$	$\boldsymbol{\beta_1}$	$\boldsymbol{\beta_2}$
Run #1 acceptance rate= 0.040175	0.8335222 (se=0.01636)	0.1253866 (se= 0.002818)	0.2349355 (se= 0.00368)	0.1192728 (se=0.007559)	0.3096259 (se=0.0073355)
Run #2 acceptance rate = 0.03815	0.6384618 (se=0.01143)	0.1464818 (se= 0.002287)	0.1828733 (se= 0.003481)	0.2183379 (se=0.010100 1)	0.3005217 (se=0.007590585)
Run #3	0.8862938	0.09709199	0.2728931	0.1836288	0.2073998

acceptance rate = 0.043825	(se=0.01305)	(se= 0.001593)	(se= 0.00261)	(se=0.0069040)	(se=0.00662388)
Run #4 acceptance rate = 0.03675	0.908406 (se=0.03781)	0.129374 (se= 0.004414)	0.2409197 (se= 0.005549)	0.1090718 (se=0.0098663)	0.2898838 (se=0.01439529)

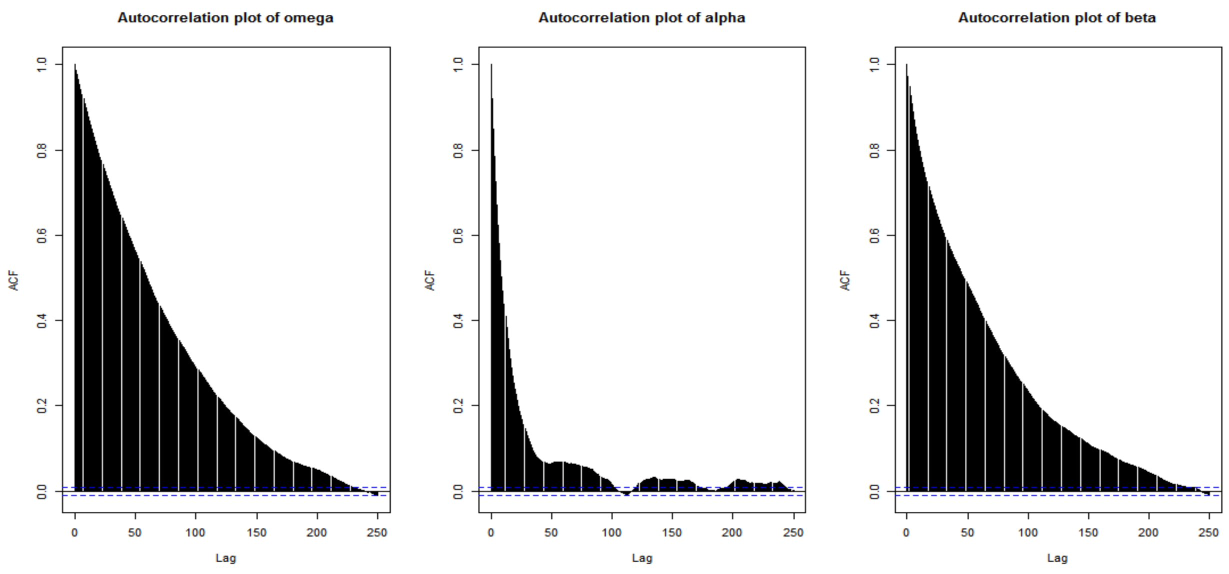
Graph 1: Time series plot of 2,000 simulated values using a GARCH(1,1) process with parameters $(\boldsymbol{\omega}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = (\mathbf{0.8}, \mathbf{0.2}, \mathbf{0.5})$



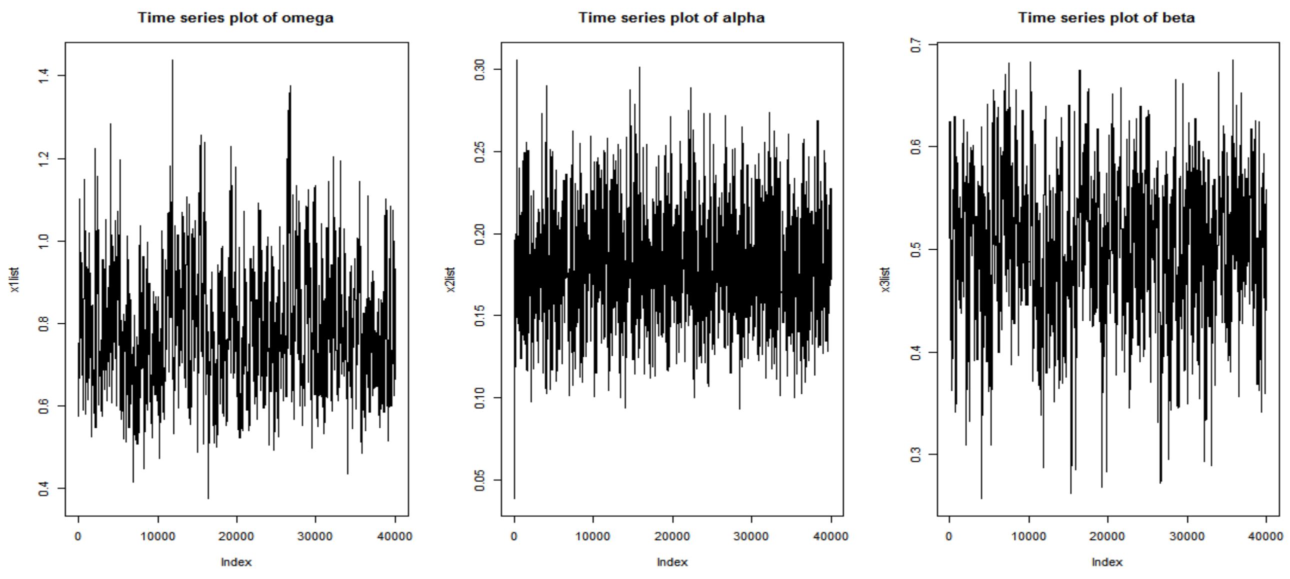
Graph 2: Time series plot of samples from the random-walk Metropolis algorithm for the GARCH(1,1) model



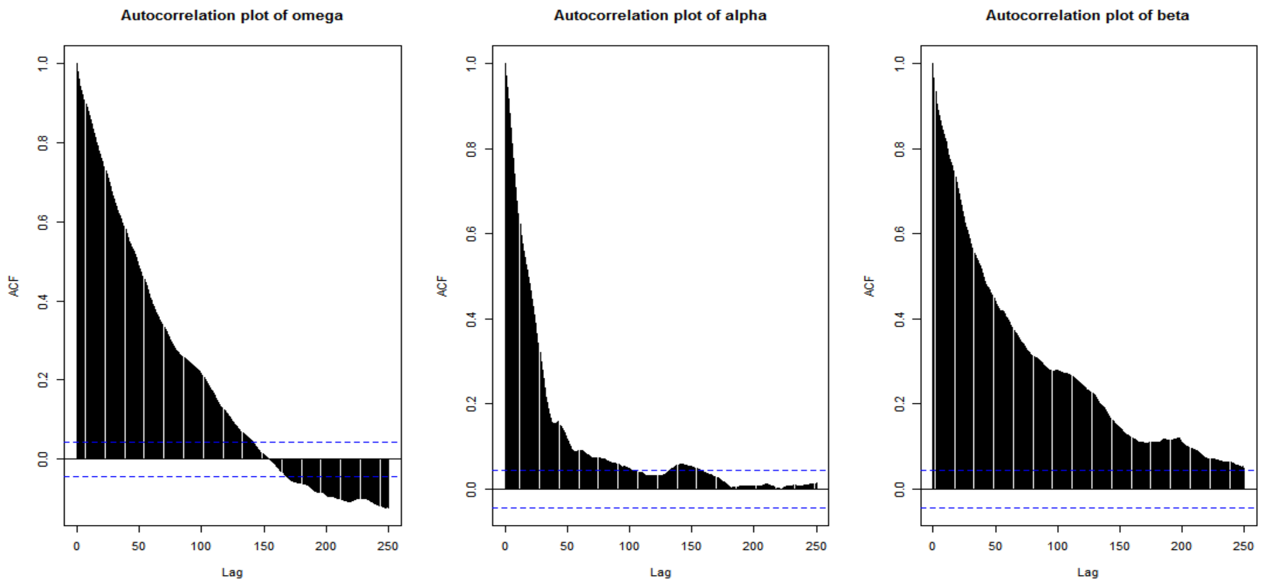
Graph 3: Autocorrelation plots of samples from the random-walk Metropolis algorithm for the GARCH(1,1) model



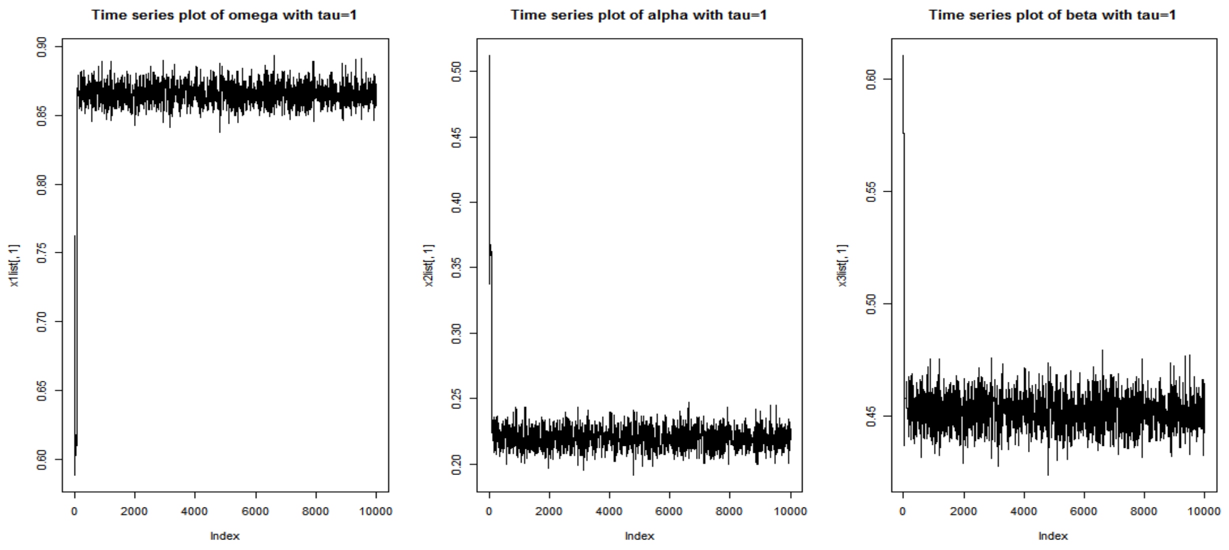
Graph 4: Time series plot of samples from the Adaptive-Metropolis algorithm for the GARCH(1,1) model



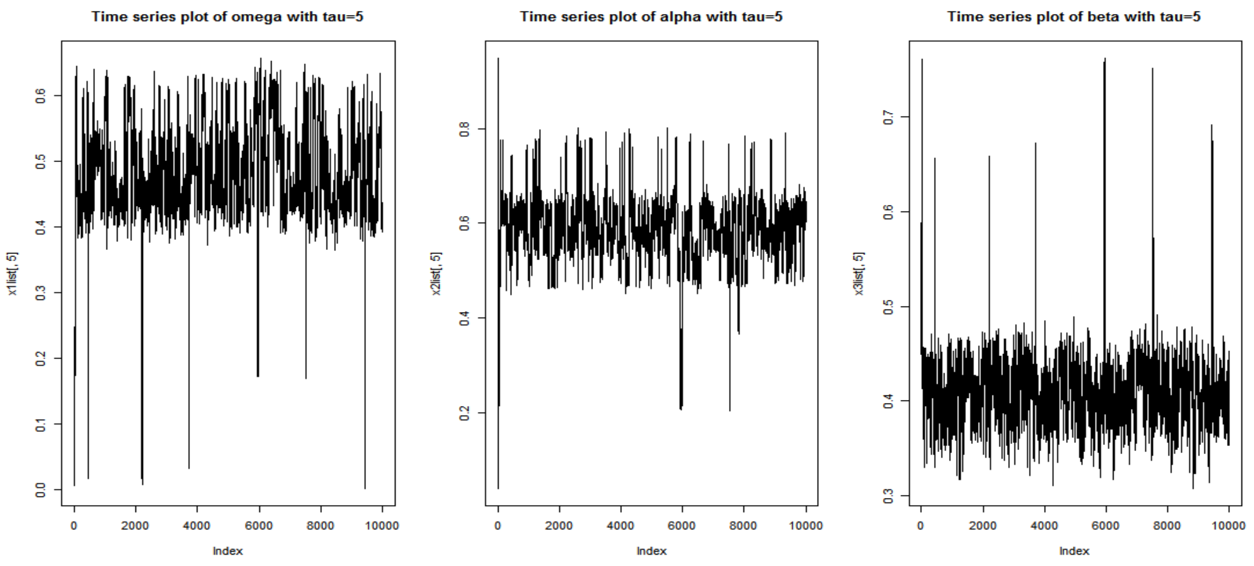
Graph 5: Autocorrelation plots of samples from the Adaptive-Metropolis algorithm for the GARCH(1,1) model



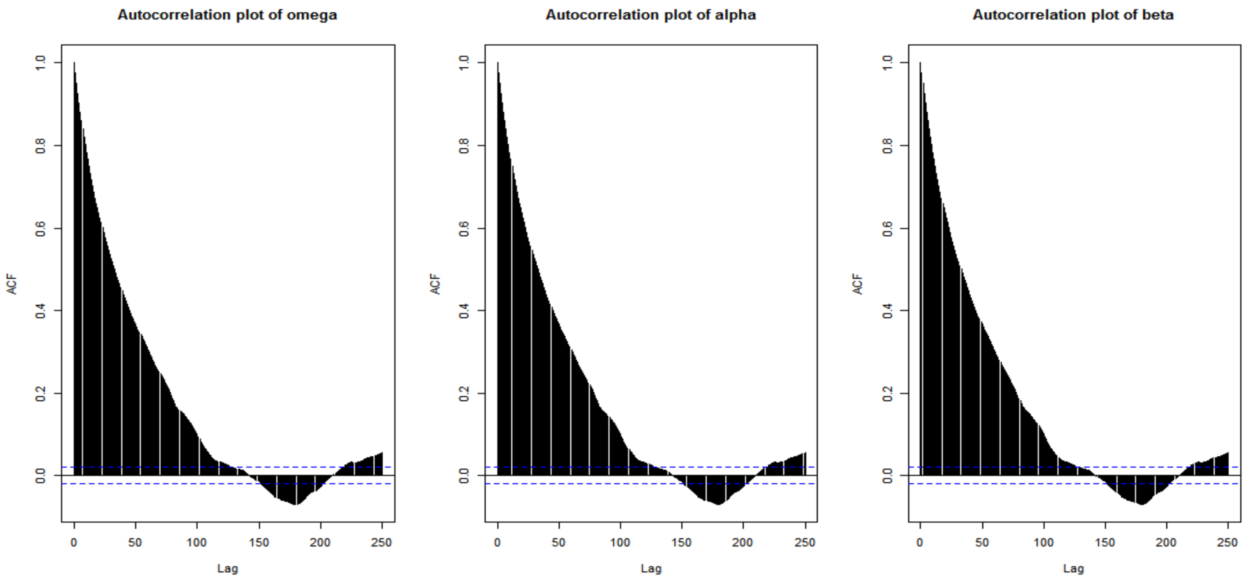
Graph 6: Time series plot of samples from the Parallel Tempered Metropolis algorithm (tau=1) for the GARCH(1,1) model



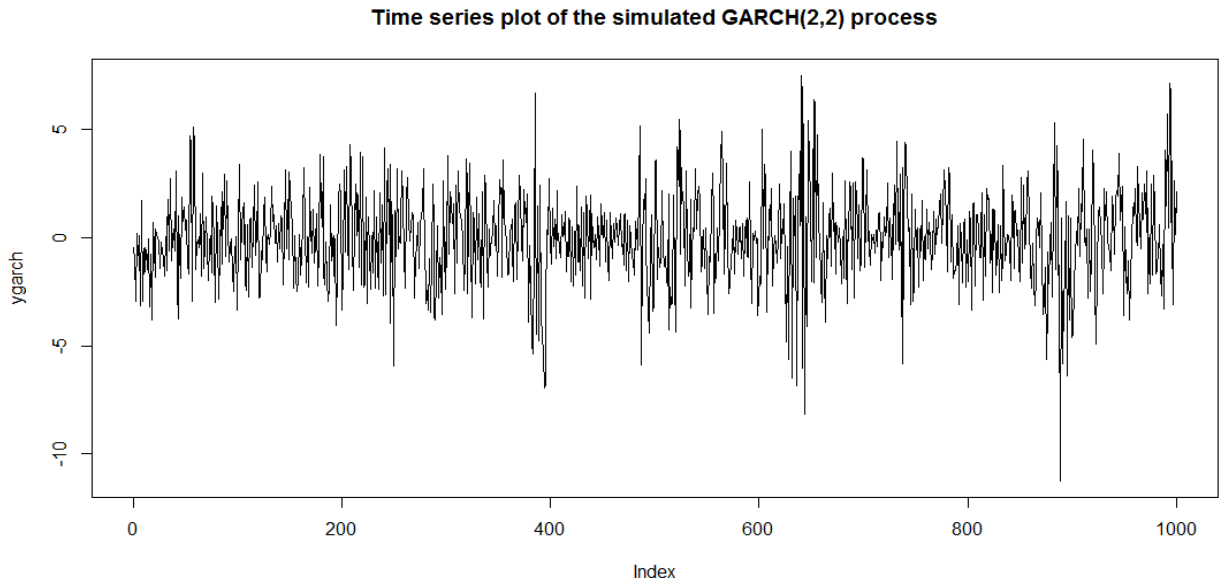
Graph 7: Time series plot of samples from the Parallel Tempered Metropolis algorithm (tau=5) for the GARCH(1,1) model



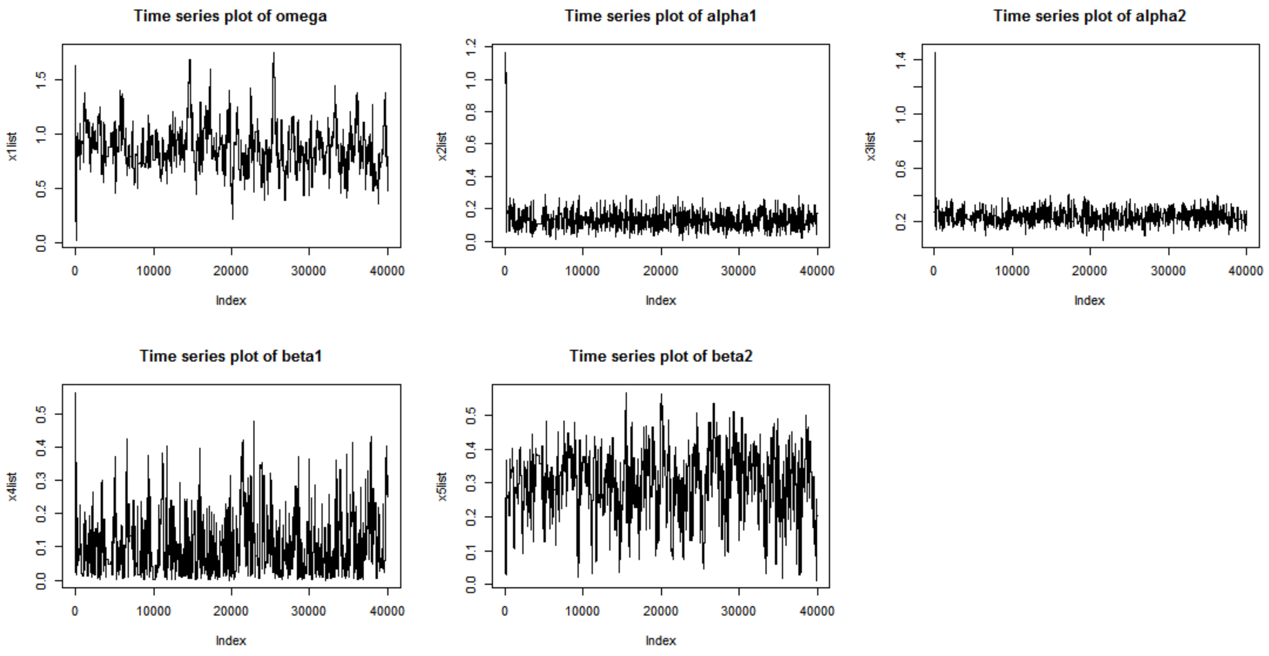
Graph 8: Autocorrelation plots of samples from the Parallel Tempered Metropolis algorithm for the GARCH(1,1) model



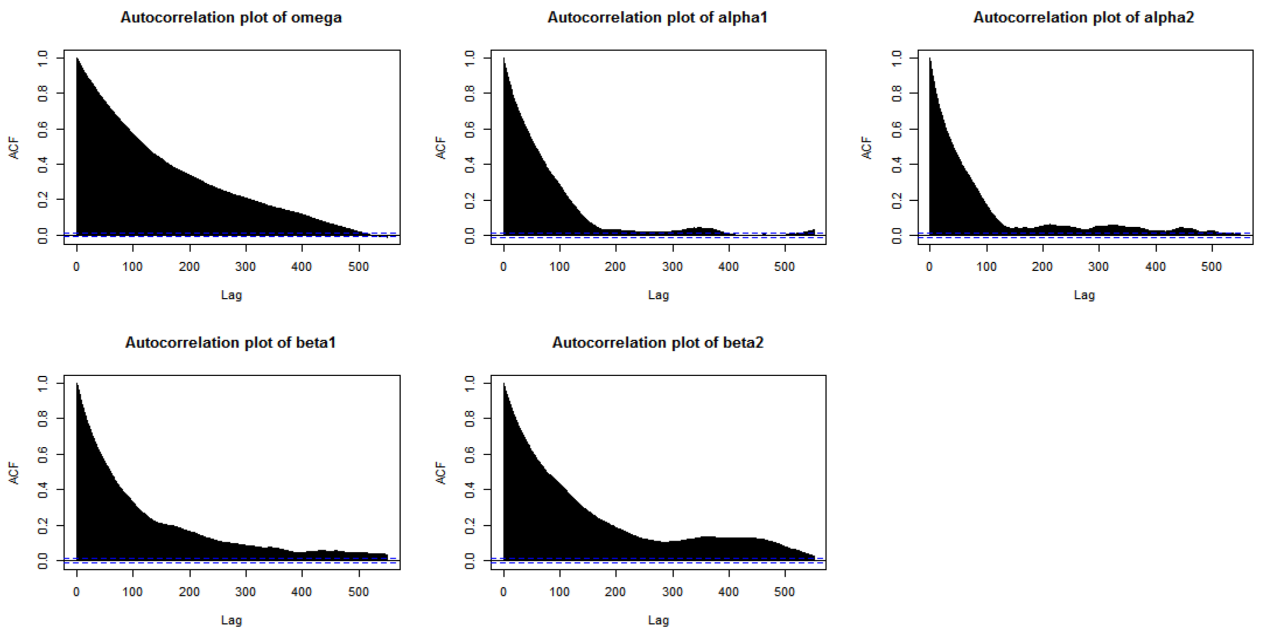
Graph 9: Time series plot of 1,000 simulated values using a GARCH(2,2) process with parameters $(\omega, \alpha_1, \alpha_2, \beta_1, \beta_2) = (0.8, 0.1, 0.25, 0.15, 0.3)$



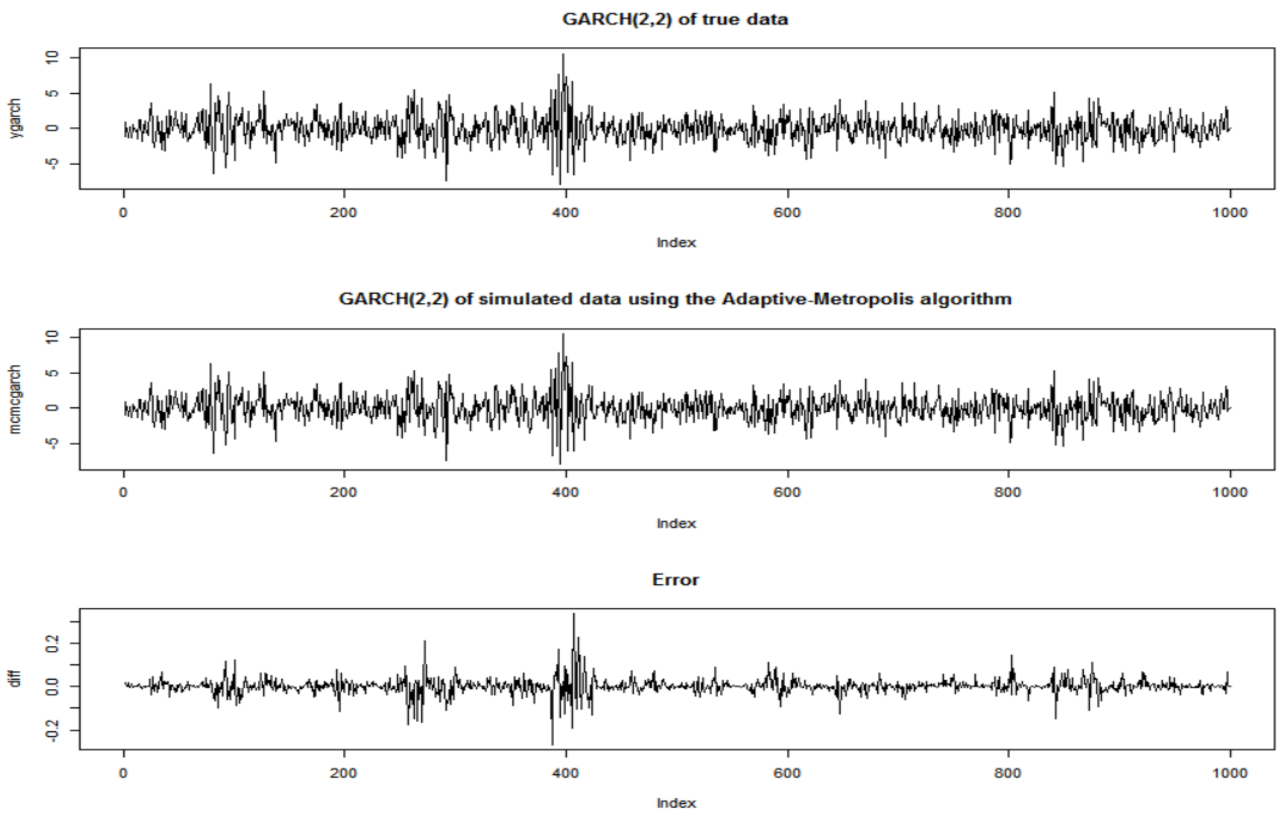
Graph 10: Time series plot of samples from the Adaptive-Metropolis algorithm for the GARCH(2,2) model



Graph 11: Autocorrelation plots of samples from the Adaptive-Metropolis algorithm for the GARCH(2,2) model



Graph 12: Compare the simulated data using the Adaptive-Metropolis fitted GARCH(2,2) model vs the true data



References

A practical introduction to garch modeling. (2012, July 05). Retrieved November 26, 2017, from https://www.r-bloggers.com/a-practical-introduction-to-garch-modeling/

Autoregressive conditional heteroskedasticity. (2017, November 22). Retrieved November 26, 2017, from https://en.wikipedia.org/wiki/Autoregressive_conditional_heteroskedasticity

Markov Chain Monte Carlo with Adaptive Proposals. (2010). *Advanced Markov Chain Monte Carlo Methods*, 305-325. doi:10.1002/9780470669723.ch8

Nakatsuma, T. (1998). A Markov-Chain Sampling Algorithm for GARCH Models. *Studies in Nonlinear Dynamics & Econometrics*, *3*(2). doi:10.2202/1558-3708.1043

Volatility Forecasting I: GARCH Models. (2009). Time Series Analysis and Statistical Arbitrage G63.2707, Fall 2009