Generalized Method of Moments with R

Pierre Chaussé

June 20, 2018

Abstract

This vignette presents the gmm4 package, which is an attempt to rebuild the gmm package using S4 classes and methods. The goal is to facilitate the development of new functionalities.

1 Single Equation

1.1 An S4 class object for GMM models

In general, GMM models are based on the moment conditions:

$$\mathrm{E}[g_i(\theta)] = 0$$

The GMM estimator is defined as

$$\hat{\theta}(W) = \arg\min_{\theta} \bar{g}(\theta)' W \bar{g}(\theta)$$

Under some regularity conditions (see Hansen, 1982), we have the following result:

$$\sqrt{n}\Big(\hat{\theta}(W) - \theta\Big) \overset{d}{\to} N\Big(0, (G'WG)^{-1}G'WVWG(G'WG)^{-1}\Big),$$

where $G = \mathrm{E}[dg_i(\theta)/d\theta]$ and V is the asymptotic variance of $\sqrt{n}\bar{g}(\theta)$. We can therefore use the following approximation for inference:

$$\hat{\theta}(W) \approx N\Big(\theta, (\hat{G}'W\hat{G})^{-1}\hat{G}'W\hat{V}W\hat{G}(\hat{G}'W\hat{G})^{-1}/n\Big)$$

with $\hat{G} = \frac{1}{n}\sum_{i=1}^{n} dg_i(\hat{\theta}(W))/d\theta$ and $\hat{V}$ is some consistent estimate V. Therefore, the property depends on the method, which in this case is simply characterized by the choice of the weighting matrix W, and on the statistical properties of $g_i(\theta)$. The GMM model class will only include the definition of $g_i(\theta)$ and its assumed statistical properties, which is basically represented by its variance.

We want to distinguish three types of $g_i(\theta)$:

1. The linear model:
$$Y_i = X_i'\theta + \varepsilon_i,$$
with the moment condition $\mathrm{E}[\varepsilon_i(\theta)Z_i] = 0$, where X_i is $k \times 1$ and Z_i is $q \times 1$ with $q \geq k$. We consider three possibilities for the asymptotic variance of $\sqrt{n}\bar{g}_i(\theta)$:
    a) "iid": Here we assume no autocorrelation and homoscedastic error with $\mathrm{Var}(\varepsilon_i|Z_i) = \sigma^2$, which implies that the asymptotic variance V is $\sigma^2\mathrm{E}[Z_iZ_i']$ and can be estimated by:
    $$\hat{V} = \hat{\sigma}^2\left(\frac{1}{n}\sum_{i=1}^{n} Z_iZ_i'\right),$$
    where $\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}\hat{\varepsilon}_i^2$, and $\hat{\varepsilon}_i = Y_i - X_i'\hat{\theta}(W)$.

b) "MDS": We assume that $g_i(\theta) \equiv (\varepsilon_i Z_i)$ is a martingale difference sequence with no additional assumption on the conditional variance of the error term. Heteroscedasticity is therefore allowed. The asymptotic variance is therefore $V = \mathrm{E}(\varepsilon_i^2 Z_i Z_i')$, and can be estimated by:

$$\hat{V} = \frac{1}{n}\sum_{i=1}^n \hat{\varepsilon}_i^2 Z_i Z_i',$$

which represents the HC0 version of the heteroscedasticity consistent covariance matrix (HCCM) estimator.

c) "HAC": If we assume that $g_t(\theta)$ (t is used when we have time series) is weakly dependent, the asymptotic covariance matrix is $V = \Gamma_0 + \sum_{i=1}^\infty (\Gamma_i + \Gamma_i')$, with $\Gamma_i = \mathrm{E}(\varepsilon_t \varepsilon_{t-i} Z_t Z_{t-i}')$. It can be estimated using a kernel estimator:

$$\hat{V} = \sum_{i=-M}^{M} K_h(i)\hat{\Gamma}_i,$$

where $K_h(i)$ is a kernel that depends on the bandwidth h, and $\hat{\Gamma}_i$ is an estimator of Γ_i.

2. The nonlinear model:

$$y_i(\theta) = x_i(\theta) + \varepsilon_i,$$

with the moment condition $\mathrm{E}[\varepsilon_i(\theta) Z_i] = 0.$, where X_i is $k \times 1$ and Z_i is $q \times 1$ with $q \geq k$. The only difference is that $\varepsilon_i(\theta)$ is a nonlinear function of the coefficient vector θ. For this case, the same three possibilities exist for the asymptotic variance.

3. The functional case: Is we cannot represent the model in a regression format with instruments, we simply write the moment conditions as $\mathrm{E}[g_i(\theta)]$ with $g_i(\theta)$ being a continuous and differentiable function from $\mathbb{R}^k$ to $\mathbb{R}^q$, with $q \geq k$. Here, we do not distinguish "iid" from "MDS". We therefore have two possible cases:

   a) "iid" or "MDS": The asymptotic variance is $V = E[g_i(\theta)g_i(\theta)']$ and can be estimated by its sample counterpart.

   b) "HAC": Same as for the linear case with $\Gamma_i = E[g_t(\theta)g_{t-i}(\theta)']$.

Since the moment conditions are defined differently, we have three difference classes to represent the three models. Their common slots are all the arguments that specify V, which include the specifications of th HAC estimator if needed, the names of the coefficients, the names of the moment conditions, k, q, n, and the argument "isEndo", a k logical vector that indicates which regressors in X_i is considered endogenous. It is considered endogenous is it is not part of Z_i. Of course, it makes no sense when $g_i(\theta)$ is a general function.

The main difference is the slots that define $g_i(\theta)$. For "linearGmm" class, the slots "modelF" and "instF" are model.frame's that define the regression model and the instruments. For "nonlinearGmm", we have the following slots: "modelF" is a data.frame for the nonlinear regression, "instF" is as for the linear case, and "fRHS" and "fLHS" are expressions to compute the right and left hand sides of the nonlinear regression. The function D() can be used to obtain analytical derivatives. Finally, the "functionGmm" class contains the slot "fct", which is a function of two arguments, the first being θ, and returns a $n \times q$ matrix with the i^{th} row being $g_i(\theta)'$. The slot "dfct" is an optional function with the same two arguments which returns the $q \times k$ matrix of first derivatives of $\bar{g}(\theta)$. The slot "X" is whatever is needed as second argument of "fct" and "dfct". The last two classes also contain the slot "theta0", which is mainly used to validate the object. It is also used latter as starting values for "optim" if no other starting values are provided. For the nonlinear regression, it must be a named vector.

Consider the following model:

$$y = \theta_0 + \theta_1 x_{1i} + \theta_2 x_{2i} + \varepsilon_i$$

with the instruments $Z_i = \{1, x_{2i}, z_{1i}, z_{2i}\}'$ and iid errors. We could create an object of class "linarGmm" as follows:

```r
library(gmm4)
data(simData)
modelF <- model.frame(y~x1+x2, simData)
instF <- model.frame(~x2+z1+z2, simData)
mod1 <- new("linearGmm", modelF=modelF, instF=instF, k=3L, q=4L, vcov="iid",
             parNames=c("(Intercept)", "x1","x2"), n=50L,
             momNames=c("(Intercept)", "x2", "z1", "z2"),
             isEndo=c(FALSE, TRUE, FALSE, FALSE))
```

The print method describes the model.

```
mod1

## GMM Model
## *********
## Moment type: linear
## Covariance matrix: iid
## Number of regressors: 3
## Number of moment conditions: 4
## Number of Endogenous Variables: 1
## Sample size:  50
```

Although there is a validity procedure when the object is created, it is not recommended to create if this way. Small error not detected by the validity method could result in estimation problems. The constructor is the method “gmmModel”, with the signature “gmmModels”, which is the union class of all above types. The above model can be created as follows:

```
mod1 <- gmmModel(y~x1+x2, ~x2+z1+z2, data=simData, vcov="iid")
mod1

## GMM Model
## *********
## Moment type: linear
## Covariance matrix: iid
## Number of regressors: 3
## Number of moment conditions: 4
## Number of Endogenous Variables: 1
## Sample size:  50
```

The two other classes of object can be created the same way. Consider the following model:

$$y_i = e^{\theta_0+\theta_1 x_{1i}+\theta_2 x_{2i}} + \varepsilon_i$$

using the same instruments. The nonlinear model can be created as follows:

```
theta0 <- c(theta0=1, theta1=1, theta2=2)
mod2 <- gmmModel(y~exp(theta0+theta1*x1+theta2*x2), ~x2+z1+z2, theta0,
                 data=simData, vcov="iid")
mod2

## GMM Model
## *********
## Moment type: nonlinear
## Covariance matrix: iid
## Number of regressors: 3
## Number of moment conditions: 4
## Number of Endogenous Variables: 2
## Sample size:  50
```

(Wrong number of endogenous variables. Needs to be fixed. The problem is that the variable names "(Intercept)" is in Z_i but not in the right hand side expression)

For the functional case, suppose we want to estimate the mean and variance of a normal distribution using the following moment condition:

$$E\begin{pmatrix} x_i-\mu \\ (x_i-\mu)^2-\sigma^2 \\ (x_i-\mu)^3 \\ (x_i-\mu)^4-3\sigma^4 \end{pmatrix} = 0$$

The functions "fct" and "dfct" would be

```
fct <- function(theta, x)
    cbind(x-theta[1], (x-theta[1])^2-theta[2],
          (x-theta[1])^3, (x-theta[1])^4-3*theta[2]^2)
dfct <- function(theta, x)
    {
        m1 <- mean(x-theta[1])
        m2 <- mean((x-theta[1])^2)
        m3 <- mean((x-theta[1])^3)
        matrix(c(-1, -2*m1, -3*m2, -4*m3,
                 0, -1, 0, -6*theta[2]), 4, 2)
    }
```

The object can than be created:

```
theta0=c(mu=1,sig2=1)
x <- simData$x3
mod3 <- gmmModel(fct, x, theta0, grad=dfct, vcov="iid")
mod3

## GMM Model
## *********
## Moment type: function
## Covariance matrix: iid
## Number of regressors: 2
## Number of moment conditions: 4
## Sample size:  50
```

1.2 Methods for gmmModels Classes

- *residuals*: Only for linearGMM and nonlinearGMM, it returns $\varepsilon(\theta)$:

  ```
  theta0 <- c(theta0=1, theta1=1, theta2=2)
  e1 <- residuals(mod1, c(1,2,3))
  e2 <- residuals(mod2, theta0)
  ```

- *Dresiduals*: Only for linearGMM and nonlinearGMM, it returns the $n\times k$ matrix $d\varepsilon(\theta)/d\theta$:

  ```
  theta0 <- c(theta0=1, theta1=1, theta2=2)
  e1 <- Dresiduals(mod1)
  e2 <- Dresiduals(mod2, theta0)
  ```

  Notice that the coefficient θ is not required for linear models, for no error is returned if it is. It is just not used. For nonlinear regressions, the derivatives are obtained analytically using D() from the *utils* package.

- *model.matrix*: For linearGMM and nonlinearGmm only. For both classes, it ca be used to get the matrix of instruments:

```
Z <- model.matrix(mod1, type="instruments")
```

For linearGMM only, it can be used to get the matrix of regressors X

```
X <- model.matrix(mod1)
```

- *modelResponse*: For linear model only, it returns the vector of response. It is not defined for nonlinearGMM classes because the left hand side is not always defined.

```
Y <- modelResponse(mod1)
```

- "]": It creates a new object of the same class with a subset of moment conditions:

```
mod1[1:3]

## GMM Model
## *********
## Moment type: linear
## Covariance matrix: iid
## Number of regressors: 3
## Number of moment conditions: 3
## Number of Endogenous Variables: 1
## Sample size:  50

mod2[c(1,2,4)]

## GMM Model
## *********
## Moment type: nonlinear
## Covariance matrix: iid
## Number of regressors: 3
## Number of moment conditions: 3
## Number of Endogenous Variables: 2
## Sample size:  50

mod3[-1]

## GMM Model
## *********
## Moment type: function
## Covariance matrix: iid
## Number of regressors: 2
## Number of moment conditions: 3
## Sample size:  50
```

- *as*: linearGmm can be converted into a nonlinearGmm or functionGmm. The former is userful when we impose nonlinear restrictions on the coefficients.

```
mod4 <- as(mod1, "nonlinearGmm")
```

Notice, however, that coefficient names and the variable names in modelF change in this case. It is done to avoid invalid variable and parameter names in the expressions. It will happens with the intercept or if there are interactions or transformations using the identity function I().

```
mod4@parNames

## [1] "theta1" "theta2" "theta3"

mod4@fLHS
```

```
## expression(Y)

mod4@fRHS

## expression(theta1*X1+theta2*X2+theta3*X3)
```

- *subset*: As for the S3 method, it creates the same class of object with a subset of the sample:

```
subset(mod1, simData$x1>4)

## GMM Model
## *********
## Moment type: linear
## Covariance matrix: iid
## Number of regressors: 3
## Number of moment conditions: 4
## Number of Endogenous Variables: 1
## Sample size:  31
```

- *evalMoment*: It computes the $n \times q$ matrix of moments, with the i^{th} row being $g_i(\theta)'$:

```
gt <- evalMoment(mod1, 1:3)
```

- *evalDMoment*: It computes the $p \times k$ matrix of derivatives of the sample mean of $g_i(\theta)$ (the matrix G above):

```
theta0 <- c(theta0=.1, theta1=1, theta2=-2)
evalDMoment(mod2, theta0)

##                 theta0    theta1    theta2
## (Intercept) -471.3350 -5128.568 -245.9807
## x2          -245.9807 -2651.763 -161.5198
## z1          -554.1436 -6026.748 -293.1144
## z2          -180.9458 -1964.310 -103.2112
```

- *momentVcov*: It computes $\hat{V}$ using the specification of the model as described in the previous section. For example, if the model is linear with MDS error, it computes $\hat{V} = \frac{1}{n}\sum_{i=1}^{n} \hat{\varepsilon}_i^2 Z_i Z_i'$.

```
momentVcov(mod1, theta=1:3)

##             (Intercept)        x2        z1        z2
## (Intercept)    85.76003  520.3819  88.88919 119.33071
## x2            520.38190 3981.2274 509.27177 919.50037
## z1             88.88919  509.2718 167.01913  91.37298
## z2            119.33071  919.5004  91.37298 273.68146
```

- *momentStrength*: For linearGmm only (for now), it computes the first stage F-test to measure the strength of the instruments:

```
momentStrength(mod1)

## $strength
##       Stats df1 df2         pv
## x1 4.113798   2  46 0.02271759
##
## $mess
## [1] "Instrument strength based on the F-Statistics of the first stage OLS"
```

Other methods will be presented below as they require to define other classes.

1.3 Restricted models

We can create objects of class “rlinearGmm”, “rnonlinearGMM” or “rfunctionGMM” using the method *restGmmModel* and print the restrictions using the *printRestrict* method.

Lets first create a new model with more regressors:

```
UR.mod1 <- gmmModel(y~x1+x2+x3+z1, ~x1+x2+z1+z2+z3+z4, data=simData)
```

We can impose restrictions in two ways. Using $R\theta = q$ format:

```
R1 <- matrix(c(1,1,0,0,0,0,0,2,0,0,0,0,0,1,-1),3,5, byrow=TRUE)
q1 <- c(0,1,3)
R1.mod1 <- restGmmModel(UR.mod1, R1, q1)
R1.mod1

## GMM Model
## *********
## Moment type: rlinear
## Covariance matrix: HAC with  Quadratic Spectral  kernel and Andrews bandwidth
## Number of regressors: 2
## Number of moment conditions: 7
## Number of Endogenous Variables: 1
## Sample size:  50
## Constraints:
##   (Intercept) + x1 = 0
##   2 x2 = 1
##   x3 - z1 = 3
## Restricted regression:
##  (y-0.5x2-3x3) = (-(Intercept)+x1)+(x3+z1)
```

Or using character vectors. As long as it uses the parameter names, it will work fine.

```
R2 <- c("x1","2*x2+z1=2", "4+x3*5=3")
R2.mod1 <- restGmmModel(UR.mod1, R2)
printRestrict(R2.mod1)

## Constraints:
##   x1 = 0
##   2 x2 + z1 = 2
##   5 x3 = -1
## Restricted regression:
##  (y-x2+0.2x3) = (Intercept)+(-0.5x2+z1)
```

If parameters have special names because of the way the regression is defined, it will also w ork fine:

```
UR.mod2 <- gmmModel(y~x1*x2+exp(x3)+I(z1^2), ~x1+x2+z1+z2+z3+z4, data=simData)
R3 <- c("x1","exp(x3)+2*x1:x2", "I(z1^2)=3")
R3.mod2 <- restGmmModel(UR.mod2, R3)
printRestrict(R3.mod2)

## Constraints:
##   x1 = 0
##   exp(x3) + 2x1:x2 = 0
##   I(z1^2) = 3
## Restricted regression:
##  (y-3I(z1^2)) = (Intercept)+x2+(-2exp(x3)+x1:x2)
```

For nonlinearGmm, only character vector or list of formulas are allowed. The restriction must also be written as one coefficient as a function of the others.

```
R1 <- c("theta1=theta2^2")
restGmmModel(mod2, R1)

## GMM Model
## *********
## Moment type: rnonlinear
## Covariance matrix: iid
## Number of regressors: 2
## Number of moment conditions: 4
## Number of Endogenous Variables: 2
## Sample size:  50
## Constraints:
##  theta1 ~ theta2^2

printRestrict(restGmmModel(mod2, theta1~theta2))

## Constraints:
##  theta1 ~ theta2
```

Restrictions can also be imposed on functionGmm:

```
restGmmModel(mod3, "mu=0.5")

## GMM Model
## *********
## Moment type: rfunction
## Covariance matrix: iid
## Number of regressors: 1
## Number of moment conditions: 4
## Sample size:  50
## Constraints:
##  mu ~ 0.5
```

All methods described in the previous subsections also apply to restricted models. However, when θ is need, it must be of the right length, which is k minus the number of restrictions. Many of these methods use the *coef* method to obtain the unrestricted version of the coefficients and call the method for unrestricted models.

For example, in the following model

```
printRestrict(R2.mod1)

## Constraints:
##   x1 = 0
##   2 x2 + z1 = 2
##   5 x3 = -1
## Restricted regression:
##  (y-x2+0.2x3) = (Intercept)+(-0.5x2+z1)
```

There are only 2 restricted coefficients, the intercept and the coefficient of $(-0.5x_2 + z_1)$. Suppose there are respectively equal to 1.5 and 0.5, then the unrestricted version is

```
coef(R2.mod1, c(1.5,.5))

## (Intercept)          x1          x2          x3          z1
##        1.50        0.00        0.75       -0.20        0.50
```

Notice that any restricted class object contains its unrestricted version. For example, rlinearGmm is a class that contains a linearGMM class object plus a few additional slots. We can therefore use the *as* method directly to convert a restricted model to its unrestricted counterpart. We can therefore compute the residuals from the restricted model as follows:

```
e1 <- residuals(as(R2.mod1, "linearGmm"),
               coef(R2.mod1, c(1.5,.5)))
```

It is identical to use the “rlinearGmm” method directly:

```
e2 <- residuals(R2.mod1, c(1.5,.5))
all.equal(e1,e2)

## [1] TRUE
```

Other methods that behave in the same way include *evalMoment* and *momentVcov*. The methods that will produce different results include *Dresiduals*, *evalDMoment*, *model.matrix*, and *modelResponse*. Restrictions affect derivatives and the left and right hand sides of regression models. Fo example:

```
R1 <- c("theta1=theta2^2")
R1.mod2 <- restGmmModel(mod2, R1)
evalDMoment(mod2, c(theta0=1, theta1=1, theta2=1))

##                 theta0      theta1      theta2
## (Intercept)  -81045584  -879800997  -763376712
## x2          -763376712 -8146652309 -7316573837
## z1           -67269892  -726185991  -616728543
## z2          -215480202 -2340842161 -2071011945

evalDMoment(R1.mod2, c(theta0=1, theta2=1))

##                 theta0       theta2
## (Intercept)  -81045584  -2522978706
## x2          -763376712 -23609878455
## z1           -67269892  -2069100525
## z2          -215480202  -6752696267
```

Every method uses the method *modelDims* to extract the information for a model. For example, the slot “parNames” of mod2 and R1.mod2 are the same even if $theta1$ is not present in the restricted model.

```
mod2@parNames

## [1] "theta0" "theta1" "theta2"

R1.mod2@parNames

## [1] "theta0" "theta1" "theta2"
```

When we need the right specifications of the model, we need to extract that information using *modelDims*.

```
modelDims(mod2)$parNames

## [1] "theta0" "theta1" "theta2"

modelDims(mod2)$k

## [1] 3

modelDims(R1.mod2)$parNames

## [1] "theta0" "theta2"

modelDims(R1.mod2)$k

## [1] 2
```

1.4 A class object for GMM Weights

Now that we have our model classes well defined, we need a way to construct a weighting matrix. We could simply define W as a matrix and move on to the estimation section, but in an attempt to make the estimation more computationally efficient and more numerically stable, we construct the weights in a particular way depending on its structure. There is in fact an optimal choice for W that minimizes the asymptotic variance of the GMM estimator. If $W = V^{-1}$, the above property becomes:

$$\sqrt{n}\Big(\hat{\theta}(V^{-1}) - \theta\Big) \xrightarrow{d} N\Big(0, [G'V^{-1}G]^{-1}\Big),$$

The new covariance matrix $[G'V^{-1}G]^{-1}$ is smaller than the one based on other W in the sense that the difference (the second minust the first) is negative definite. The inverse V^{-1} may have to be computed several times for inference or simply for estimation if we use iterative GMM of CUE. It is therefore worth finding a way to reduce the number of potentially unstable operations. For example, in the linear or nonlinear model with iid errors, $V^{-1} = [\sigma^2 \mathrm{E}(Z_iZ_i')]^{-1}$, and can be estimated by

$$\hat{V} = \frac{1}{\hat{\sigma}^2}\left(\frac{1}{n}\sum_{i=1}^{n} Z_iZ_i'\right)^{-1}$$

Therefore two $\hat{V}$'s differ only by their estimates of σ^2. It is therefore not necessary to recompute the second term each time. In fact, it is even not necessary to compute the sum. A more stable way would be to store the QR decomposition of the $n \times q$ matrix Z. The “gmmWeights” class store only what is needed. It can be created by the *evalWeights* method. It is a method for the union class “gmmModels”, which includes all restricted models.The method has three arguments, the “gmmModels”, the vector of coefficients, and the type of weights. The third argument can be a matrix, if we want to provide our own fixed one, the character ”ident”, to create an identity matrix or, which is the default, the character ”optimal”. In the latter case, the efficient weighting matrix is computed based on the characteristics of the “gmmModels” specified when the object was created.

There are two ways of creating an identity. The first way is to use the character ”ident”. In this case, it is not necessary to provide a vector of coefficients.

```
model <- gmmModel(y~x1, ~z1+z2, data=simData, vcov="iid") ## lets create a simple model
wObj <- evalWeights(model, w="ident")
```

The *show* method for the “gmmWeights” object prints the matrix as it should look like. If it is the efficient matrix, the inverse is computed and printed. It is not too efficient but when do we really need to see it? For the one we just created, we get

```
wObj

## GMM weights matrix object
## [1] "Identity"
```

Only a character string is printed because the identity is not actually created. After all, why should we? If we need to compute $G'IG$, we do not want to create I and do the operation, but rather compute $G'G$. That's how things are done in the package. For this reason, the second way of creating an identity weighting matrix is not recommended:

```
evalWeights(model, w=diag(3))

## GMM weights matrix object
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

The optimal matrix at θ can be obtained without specifying w.

```
wObj <- evalWeights(model, theta=c(1,2))
```

The type slot indicates how the weighting matrix is stored.

```
wObj@type

## [1] "qr"
```

Here the QR decomposition is store because vcov="iid". For any "gmmModels" including "functionGmm" classes, with vcov="MDS", the QR decomposition of the $n \times q$ matrix of moment conditions is stored. It avoids having to compute $g(\theta)'g(\theta)$. For HAC, there is no gain in storing the QR decomposition. The type is then "chol", which indicates that the Cholesky upper triangular matrix is stored:

```
model2 <- gmmModel(y~x1, ~z1+z2, data=simData, vcov="HAC")
evalWeights(model2, c(1,2))@type

## [1] "chol"
```

When the matrix is provided, the type is "weights", which indicates that no inversion is needed

```
evalWeights(model, w=diag(3))@type

## [1] "weights"
```

The weights matrix is used to compute the vector of estimates, its covariance matrix and to do inference. Most operations ar in the form $A'WB$ for matrices A and B. How do we compute those knowing that it depends on how W is stored in the object. The method *quadra* does it for us. Consider the following optimal weighting matrix, which is stored as a QR decomposition:

```
wObj <- evalWeights(model, theta=1:2)
```

Let compute G and $g(\bar{\theta})$

```
G <- evalDMoment(model, theta=1:2)
gbar <- colMeans(evalMoment(model, theta=1:2))
```

If we need to compute $\bar{g}'W\bar{g}$, which is the objective function that we want to minimize, we do the following:

```
quadra(wObj, gbar)

## [1] 0.8478471
```

To compute $G'W\bar{g}$, which is the first order condition of the minimization problem, we proceed as follows:

```
quadra(wObj, G, gbar)

##           [,1]
## [1,] 0.1316962
## [2,] 0.7043764
```

If we only want W, we only use the weights as argument.

```
quadra(wObj)

##             [,1]         [,2]         [,3]
## [1,]  0.11425728 -0.041052353 -0.036112517
## [2,] -0.04105235  0.028230539  0.008474443
## [3,] -0.03611252  0.008474443  0.019640578
```

It is what the *print* method calls before printing the object. Finally, the "[" method can be used to create another "gmmWeights" object with a subset of the moment conditions. Only one argument is needed, and the slot "type" of the object is converted into "weights".

```
wObj[1:2]

## GMM weights matrix object
##             [,1]        [,2]
## [1,]  0.11425728 -0.04105235
## [2,] -0.04105235  0.02823054
```

We just saw a way of computing the objective function using *quadra*, but is can also be done using the *evalObjective* method. In this case, the weights is not necessarily based on the same coefficient as $\bar{g}$, which is often the case in GMM estimations:

```
theta0 <- 1:2
wObj <- evalWeights(model, theta0)
theta1 <- 3:4
evalObjective(model, theta1, wObj)

## [1] 374.6209
```

Notive that the method returns $n\bar{g}'W\bar{g}$.

1.5 The *solveGmm* Method

We now have all we need to estimate our models. The main method to estimate a model for a given W is *solveGmm*. The available signatures are:

```
showMethods("solveGmm")

## Function: solveGmm (package gmm4)
## object="allNLGmm", wObj="gmmWeights"
## object="functionGmm", wObj="gmmWeights"
##     (inherited from: object="allNLGmm", wObj="gmmWeights")
## object="linearGmm", wObj="gmmWeights"
## object="nonlinearGmm", wObj="gmmWeights"
##     (inherited from: object="allNLGmm", wObj="gmmWeights")
## object="rlinearGmm", wObj="gmmWeights"
##     (inherited from: object="linearGmm", wObj="gmmWeights")
## object="rnonlinearGmm", wObj="gmmWeights"
##     (inherited from: object="allNLGmm", wObj="gmmWeights")
## object="rslinearGmm", wObj="sysGmmWeights"
## object="slinearGmm", wObj="sysGmmWeights"
## object="snonlinearGmm", wObj="sysGmmWeights"
```

The last three are for systems of equations that we will cover later. The first is for "nonlinearGmm" and "functionGmm", and the second for "linearGmm". The methods require a gmmWeights object as second argument. For "nonlinearGmm" and "functionGmm" classes, there is a third optional argument, "theta0", which is the starting value to pass to *optim*. If not provided, the one stored in the GMM model object is used.

The method simply minimizes $\bar{g}(\theta)'W\bar{g}(\theta)$ for a given W. For "linearGmm" classes, the analytical solution is used. It is therefore the prefered class to use when it is possible. For all other classes, the solution is obtained by *optim*, and the argument "..." is used to pass options to it. For "nonlinearGmm", the gradian of the objective function, $2nG'W\bar{g}$ is passed to *optim* using the analytical derivative of the moment conditions (the *evalDMoment* method). For "functionGMM" classes, G is computed numerically using *numericDeriv* unless $dfct$ was provided when the object was created. The *solveGmm* method returns a vector of coefficients and a convergence code. The latter is null for linear models and is the code from *optim* otherwise.

Consider the following linear model:

```
mod <- gmmModel(y~x1, ~z1+z2, data=simData, vcov="MDS")
```

We can estimate the model using the identity matrix as weights as follows:

```
wObj0 <- evalWeights(mod, w="ident")
res0 <- solveGmm(mod, wObj0)
res0$theta

## (Intercept)          x1
##   0.1049242   0.9553511
```

For two-step GMM, we just need to recompute the weighting matrix and call the method again.

```
wObj1 <- evalWeights(mod, res0$theta)
res1 <- solveGmm(mod, wObj1)
res1$theta

## (Intercept)          x1
##   0.1505614   0.9503860
```

We could iterate and get the iterative GMM estimator. The result may be different if we express the linear model in a nonlinear way or using a function, which is not recommended.

```
solveGmm(as(mod, "nonlinearGmm"), wObj1)$theta

##    theta1    theta2
## 0.1505604 0.9503862

solveGmm(as(mod, "functionGmm"), wObj1)$theta

##    theta1    theta2
## 0.1505614 0.9503860
```

Consider now the above nonlinear model that we repeat here.

```
theta0 <- c(theta0=0, theta1=0, theta2=0)
mod2 <- gmmModel(y~exp(theta0+theta1*x1+theta2*x2), ~x2+z1+z2, theta0,
                 data=simData, vcov="MDS")
wObj0 <- evalWeights(mod2, w="ident")
res1 <- solveGmm(mod2, wObj0, control=list(maxit=2000))
res1

## $theta
##      theta0      theta1      theta2
##  0.43293969  0.20638573 -0.01283577
##
## $convergence
## [1] 0
```

Notice that there is no signature for restricted models. However, it is not needed since they inherit from their unrestricted counterpart and the same procedure is needed to estimate them. Suppose, for example, that we want to impose the restriction $\theta_1 = \theta_2^2$.

```
R1 <- c("theta1=theta2^2")
rmod2 <- restGmmModel(mod2, R1)
res2 <- solveGmm(rmod2, wObj0, control=list(maxit=2000))
res2

## $theta
##    theta0    theta2
##  2.269389 -0.118783
##
## $convergence
## [1] 0
```

The unrestricted version can be extracted using *coef*.

```
coef(rmod2, res2$theta)

##      theta0      theta1      theta2
##  2.26938888  0.01410941 -0.11878305
```

1.6 GMM Estimation: the *gmmFit* method

For most users, what we presented above will rarely be used. What they want is a way to estimate their models without worrying about how it is done. The *gmmFit* method is the main method to estimate models. The only requirement is to first create a "gmmModels". Before going into all the details, the most important arguments to set is the object, which is a "gmmModels" class, and a type of GMM. The different types are: (1) "twostep" for two-step GMM, which is the default, (2) "iter" for iterative GMM, (3) "cue" for continuously updated GMM , or (4) "onestep" for th one-step GMM.

In this package, the one-step GMM means the estimation using the identity matrix as W. It is therefore not an efficient GMM. The two-step GMM, without any other argument is computed as follows:

1. Define W_0 as being the identity matrix.
2. Get $\hat{\theta}_1 \equiv \hat{\theta}(W_0)$
3. Compute $W_1 = [\hat{V}(\hat{\theta}_1)]^{-1}$.
4. Get $\hat{\theta}_2 \equiv \hat{\theta}(W_1)$.

For the iterative GMM we proceed as follows:

1. Define W_0 as being the identity matrix.
2. Get $\hat{\theta}_1 \equiv \hat{\theta}(W_0)$
3. Compute $W_1 = [\hat{V}(\hat{\theta}_1)]^{-1}$.
4. Get $\hat{\theta}_2 \equiv \hat{\theta}(W_1)$.
5. If $\|\hat{\theta}_1 - \hat{\theta}_2\|/(1+\|\hat{\theta}_1\|) < itertol$, where *itertol* is a user defined tolerance level, stop. Otherwise, set $\hat{\theta}_1 = \hat{\theta}_2$ and go back to 3. By default, $itertol = 10^{-7}$.

CUE is a one step efficient GMM method in which $W = \hat{V}(\theta)$. The solution is obtained by minimizing $n\bar{g}(\theta)[\hat{V}(\theta)]^{-1}\bar{g}(\theta)$.

There are two special cases that are worth mentioning. The first case applies to all "gmmModels". If $q = k$, the model is just-identified. In that case, the choice of W has no effect on the solution. Therefore, *gmmFit* will automatically set W to the identity and return the one-step GMM solution. Setting the argument type to another value will therefore have no effect on the result.

Second, when "vcov" is set to "iid" in either a linearGMM or a "nonlinearGmm" model, the matrices W_1 and W_2 are proportional to each other. They therefore lead to the same solution. As a result, the two-step GMM, iterative GMM and CUE produce identical solution. In particular, if the model is linear, the solution corresponds to the two-stage least squares solution. In fact, *gmmFit* calls the method *tsls* in that case. We will look at the method below.

The *gmmFit* method returns the S4 class object "gmmfit". The object contains the vector of coefficient estimates, the "gmmWeights" used to obtain it, the model object and other information about the method and convergence. We will cover its methods in the next section. The only one we introduce now is the *show* method which prints the model info, the estimation method and the coefficient estimates. To avoid printing the model, we can set the argument “model” of print to FALSE.

```
mod <- gmmModel(y~x1, ~z1+z2, data=simData, vcov="MDS")
gmmFit(mod, type="onestep")

## GMM Model
## *********
## Moment type: linear
## Covariance matrix: MDS
## Number of regressors: 2
## Number of moment conditions: 3
## Number of Endogenous Variables: 1
## Sample size:  50
##
## Estimation:  One-Step GMM with fixed weights
## coefficients:
## (Intercept)          x1
##   0.1049242   0.9553511

print(gmmFit(mod, type="twostep"), model=FALSE)

##
## Estimation:  Two-Step GMM
## coefficients:
## (Intercept)          x1
##   0.1505614   0.9503860

print(gmmFit(mod, type="iter"), model=FALSE)

##
## Estimation:  Iterated GMM
## Convergence Iteration:  0
## coefficients:
## (Intercept)          x1
##   0.1604345   0.9487049
```

For nonlinear models, it is possible to pass arguments to *optim* and to set a different starting value with the argument “start”.

```
theta0 <- c(theta0=0, theta1=0, theta2=0)
mod2 <- gmmModel(y~exp(theta0+theta1*x1+theta2*x2), ~x2+z1+z2, theta0,
                 data=simData, vcov="MDS")
res1 <- gmmFit(mod2)
print(res1, model=FALSE)

##
## Estimation:  Two-Step GMM
## Convergence Optim:  0
## coefficients:
##      theta0      theta1      theta2
##  0.61713001  0.18549461 -0.01975427

theta0 <- c(theta0=0.5, theta1=0.5, theta2=-0.5)
res2 <- gmmFit(mod2, start=theta0, control=list(reltol=1e-8))
print(res2, model=FALSE)

##
## Estimation:  Two-Step GMM
## Convergence Optim:  0
## coefficients:
```

```
##      theta0      theta1      theta2
##  0.61712992  0.18549462 -0.01975426
```

For the iterative GMM, we can control the tolerance level and the maximum number of iterations with the arguments "itertol" and "itermaxit". The argument "weights" is equal to the character string "optimal", which implies that by default W is set to the estimate of V^{-1}. If "weights" is set to "ident", *gmmFit* returns the one-step GMM. Alternatively, we can provide *gmmFit* with a fixed weighting matrix. It could be a matrix or a "gmmWeights" object. When the weighting matrix is provided, it returns a one-step GMM based on that matrix. The "gmmfit" object contains a slot "efficientGmm" of type logical. It is TRUE if the model has been estimated by efficient GMM. By default it is TRUE, since "weights" is set to "optimal". If "weights" takes any other value or if "type" is set to "onestep", it is set to FALSE. There is one exception. It is set to TRUE if we provide the method with a weighting matrix and we set the argument "efficientWeights" to TRUE. For example, the optimal weighting matrix of the minimum distance method does not depend on any coefficient. It is probably a good idea in this case to compute it before and pass it to the *gmmFit* method. The value of the "efficientGmm" slot will be used by the *vcov* method to determine whether it should return the robust (or sandwich) covariance matrix.

1.7 Methods for "gmmfit" classes

- *meatGmm*: It returns the meat of the sandwich covariance matrix. The only other argument is "robust". A non robust meat assumes that $W = V^{-1}$, which is true if the model has been estimated by efficient GMM. Since W is usually a first step weighting matrix, it is not numerically identical to the estimate of V^{-1} based on the final estimate. However, it is a common practice to ignore it. The meat will in this case be equal to $(G'\hat{V}^{-1}G)$. If "robust" is TRUE, we do not assume that $W = V^{-1}$ and the meat becomes $(G'W\hat{V}WG)$.
- *bread*: It returns the bread of the sandwich covariance matrix, $(G'WG)^{-1}$, where W is the weighting matrix used to get the final estimate..
- *vcov*: It returns the covariance matrix of the coefficient. By default, it returns a sandwich matrix if the argument "efficienGmm" of the object is FALSE or if the model is just identified, and a non sandwich estimator otherwise. Here are all the possibilities:
  - Efficient and over-identified GMM: $(G'\hat{V}^{-1}G)^{-1}/n$
  - Just-identified GMM: $G^{-1}\hat{V}G^{-1'}/n$
  - Any other sandwich estimator: $(G'WG)^{-1}G'W\hat{V}WG(G'WG)^{-1}/n$.
  - The argument "breadonly" is set to TRUE: $(G'WG)^{-1}/n$. For efficient GMM, it is asymptotically equivalent to $(G'\hat{V}^{-1}G)^{-1}/n$. It is particularly useful for efficient and fixed weighting matrices.

  The method is flexible enough that you may hand up with a non-valid covariance matrix if not careful. For example, setting "sandwich" to FALSE would lead to non valid covariance matrix if the model was not estimated by efficient GMM. It is important to understand that we assume here that the specifications of the model are valid. If you set "vcov" to iid and that the errors are heteroscedastic, there is nothing you can do to get valid standard errors. You need to recreate a new gmmModels object. An example will be given below using two-stage least squares.

  The argument "df.adj" can be set to TRUE if degrees of freedom adjustment is needed. In that case, the covariance matrix is multiplied by $n/(n-k)$. It is only included in the package to reproduce textbook examples. This adjustment is not really justified in the GMM context.
- *specTest*: It tests the null hypothesis $\mathrm{E}[g_i(\theta)] = 0$ using the J-test. The statistics is $n\bar{g}'\hat{V}^{-1}\bar{g}$ and it is asymptotically distributed as a χ^2_{q-k} under the null. The model must have been estimated by efficient GMM for this test to be valid. The method returns an S4 class object.

```
mod <- gmmModel(y~x1, ~z1+z2, data=simData, vcov="MDS")
res <- gmmFit(mod)
specTest(res)
```

```
##
##  J-Test
##                   Statistics  df   pvalue
## Test E(g)=0:        1.0349   1  0.30902
```

- *summary*: It computes important information about the estimated model. It is an S4 class object with a *print* method that shows the results in the usual way.

```
summary(res)

## GMM Model
## *********
## Moment type: linear
## Covariance matrix: MDS
## Number of regressors: 2
## Number of moment conditions: 3
## Number of Endogenous Variables: 1
## Sample size:  50
##
## Estimation:  Two-Step GMM
## Sandwich vcov: FALSE
## coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.15056    0.52608  0.2862   0.7747
## x1           0.95039    0.10207  9.3112   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##  J-Test
##                   Statistics  df   pvalue
## Test E(g)=0:        1.0349   1  0.30902
##
##
## Instrument strength based on the F-Statistics of the first stage OLS
## x1 : F( 2 ,  47 ) =  11.04543  (P-Vavue =  0.0001169271 )
```

The argument "..." can be used to pass options to the *vcov* method. For example, we can used the bread only to compute the standard errors:

```
summary(res, breadOnly=TRUE)@coef

##              Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) 0.1505614   0.531185 0.2834445 7.768362e-01
## x1          0.9503860   0.103044 9.2231070 2.886135e-20
```

- *hypothesisTest*: Method to perform hypothesis tests on the coefficients. Consider the following unrestricted model:

```
mod <- gmmModel(y~x1+x2+x3+z1, ~x1+x2+z1+z2+z3+z4, data=simData, vcov="iid")
res <- gmmFit(mod)
```

We want to test the hypothesis

```
R <- c("x1=1", "x2=x3", "z1=-0.7")
rmod <- restGmmModel(mod, R)
printRestrict(rmod)

## Constraints:
##   x1 = 1
##   x2 - x3 = 0
##   z1 = -0.7
## Restricted regression:
##  (y-x1+0.7z1) = (Intercept)+(x2+x3)
```

There are three ways to do it. The Wald test only requires us to estimate the unrestricted model. It is performed as follows:

```
hypothesisTest(object.u=res, R=R)

## Wald Test
## ***********
## The Null Hypothesis:
##   x1 = 1
##   x2 - x3 = 0
##   z1 = -0.7
## Distribution: Chi-square with 3 degrees of freedom
##   Statistics      Pvalue
## 1   15.97411 0.001147931
```

The statistics is $(R\hat{\theta}-q)'[R\hat{\Omega}R']^{-1}(R\hat{\theta}-q)$, where $\hat{\Omega}$ is the covariance matrix of $\hat{\theta}$, and is distributed as a chi-square with degrees of freedom equal to the number of restrictions. Here R and q are given in the restricted model:

```
rmod@cstLHS

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    1    0    0    0
## [2,]    0    0    1   -1    0
## [3,]    0    0    0    0    1

rmod@cstRHS

## [1]  1.0  0.0 -0.7
```

We can also test it using the LM test, which test if the score of the GMM objective is close enough to zero when evaluated at the restricted coefficient estimates. The statistics is

$$n\bar{g}(\tilde{\theta})'\hat{V}^{-1}\tilde{G}\hat{\Omega}\tilde{G}'\hat{V}^{-1}\bar{g}(\tilde{\theta}),$$

where the tilde implies that it is evaluated at the restricted coefficient estimates. The asymptotic distribution is the same as the Wald test. To perform the test, we need to estimate the restricted model.

```
res.r <- gmmFit(rmod)
```

Then, we perform the test

```
hypothesisTest(object.r=res.r)

## LM Test
## ***********
## The Null Hypothesis:
##   x1 = 1
##   x2 - x3 = 0
##   z1 = -0.7
## Distribution: Chi-square with 3 degrees of freedom
##   Statistics      Pvalue
## 1   11.58521 0.008947939
```

The LR test, compares the values of the GMM objective function at the restricted and unrestriced coefficient estimates. It is in fact the restricted minus the unrestricted one. The distribution is also the same in large samples. We therefore need both the restricted and unrestricted model:

```
hypothesisTest(object.r=res.r, object.u=res)

## Wald Test
## ***********
## The Null Hypothesis:
##   x1 = 1
##   x2 - x3 = 0
##   z1 = -0.7
## Distribution: Chi-square with 3 degrees of freedom
##   Statistics      Pvalue
## 1   15.97411 0.001147931
```

Alternatively, we can give both model and specify the test.

```
hypothesisTest(object.r=res.r, object.u=res, type="LM")
hypothesisTest(object.r=res.r, object.u=res, type="Wald")
hypothesisTest(object.r=res.r, object.u=res, type="LR")
```

- *coef*: Returns the coefficient estimate.

```
coef(res.r)

## (Intercept)     (x2+x3)
##  1.24288790 -0.09512986
```

- *residuals*: Returns the residuals. Only for "linearGmm" and "nonlinearGmm".

```
e <- residuals(res)
e.r <- residuals(res.r)
```

References

L. P. Hansen. Large sample properties of generalized method of moments estimators. *Econometrica*, 50:1029–1054, 1982.