Online Appendix A

SAS Code

PROC MCMC is general purpose MCMC routine that provides considerable flexibility in specifying models. The syntax is similar to PROC NLMIXED.

Annotated PROC MCMC code

```
proc mcmc data=one outpost=postout thin=1 nmc=500000 nbi=100000
    monitor=(_parms_);
    ods select PostSummaries PostIntervals;
    array u[17];

    parms b0 0 b1 0;
    parms u: 0;
    parms s2e_cl .39;
    parms s2e_un .27;
    parms s2u 1;

    prior b0 ~normal(3, var=2.25);
    prior b1 ~normal(0, var=1);
    prior u: ~normal(0, var=s2u);
    prior s2e_cl ~gamma(shape=13, scale=.03);
    prior s2e_un ~gamma(shape=9, scale=.03);
    prior s2u ~gamma(shape=.7, scale=.098);

    if grouped=1 then
        mu = b0 + b1*DIS + u[groupid];
    else
        mu = b0;

    if grouped=1 then
        llike = lpdfnorm(tii2, mu, sqrt(s2e_cl));
    else
        llike = lpdfnorm(tii2, mu, sqrt(s2e_un));

    model general(llike);
run;

proc mcmc data=one outpost=postout thin=1 nmc=500000 nbi=100000
    monitor=(_parms_);
```

The `proc mcmc` line specifies the input and output data with the data and outpost commands. Additionally, the `proc mcmc` line specifies the number of iterations in the MCMC chain (`nmc`), the number of burn-in iterations (`nbi`), and the amount the MCMC chain should be thinned (`thin`). Finally, the `monitor` option specifies which parameters in the model SAS should save the

draws from the posterior distribution for in the outpost file. Using the `_parms_` option indicates that the draws from the posterior for all parameters should be saved.

```
ods select PostSummaries PostIntervals;
```

The `ods select` line specifies what information should be posted to the output file. In this case we ask for summary information (`PostSummaries`) for each parameter (means, standard deviations, and quartiles) and for interval estimates (`PostIntervals`) for each parameter.

```
array u[17];
```

In this model, the cluster means (`u`) are unobserved. Thus, we need to create an array that has as many elements as there are clusters in the data. The `array u[17]` tells says to create an array called `u` with 17 elements.

```
parms b0 0 b1 0;
parms u: 0;
parms s2e_cl .39;
parms s2e_un .27;
parms s2u 1;
```

The `parms` statements name and provide starting values for the parameters in the model.

```
prior b0 ~normal(3, var=2.25);
prior b1 ~normal(0, var=1);
prior u: ~normal(0, var=s2u);
prior s2e_cl ~gamma(shape=13, scale=.03);
prior s2e_un ~gamma(shape=9, scale=.03);
prior s2u ~gamma(shape=.7, scale=.098);
```

The `prior` statements specify the prior distributions for each parameter.

```
if txcond=1 then
    mu = b0 + b1*DIS + u[groupid];
else
    mu = b0;
```

The likelihood for the data is a normal likelihood. The first `if/else` statement specifies what the mean of the likelihood is for the clustered condition (`txcond=1`) and the unclustered condition (`txcond=0`).

```
if txcond=1 then
    llike = lpdfnorm(tii2, mu, sqrt(s2e_cl));
else
    llike = lpdfnorm(tii2, mu, sqrt(s2e_un));
```

The second `if/else` statement actually specifies the full likelihood for the clustered and unclustered condition. Because we want to have unique residual errors for the clustered and unclustered condition, we have to specify the likelihood separately for each condition and then use the general likelihood in the model statement below. If we wanted a common residual variance for each condition, we could forgo this step and use the `normal` function with the `model` statement.

```
model general(llike);
```

The `model` statement specifies the model we want PROC MCMC to estimate.

JAGS

JAGS can be run as a standalone application through a terminal window. It can also be run through R using the `rjags` package. We find this workflow to be useful as we can do all our data management in R, pass the information to JAGS, and have JAGS return the results in a format that R can use. Our JAGS workflow consists of six steps:
1. Load necessary packages
2. Read in and prepare the data
3. Write the JAGS model file
4. Write the JAGS model file to a text file and prepare the data, initial values, and parameters to monitor
5. Use the `rjags` package to run the model
6. Process the results

Annotated JAGS Code

```
## Part 1: Load rjags
library(rjags)

## Part 2: Read and prepare the data
data <- read.csv("partial_cluster_example.csv")
yc <- data$y[data$grouped==1]
cid <- data$j[data$grouped==1]
yu <- data$y[data$grouped==0]
sizeclus <- length(yc)
sizeunclus <- length(yu)
numberclus <- max(cid)

## Part 3: Write the JAGS code file, data, initial values,
and parameters to monitor
jags.code <- "
    model {

        for (i in 1:sizeunclus) {
            yu[i] ~ dnorm(b0, prec.un)
```

```
        }
        for (i in 1:sizeclus) {
            yc[i] ~ dnorm(mu.c[i], prec.cl)
            mu.c[i] <- b0 + b1 + u[cid[i]]
        }

        b0 ~ dnorm(3,.44)
        b1 ~ dnorm(0,1)

        prec.cl <- 1/s2e.cl
        s2e.cl ~ dgamma(13,33.33)

        prec.un <- 1/s2e.un
        s2e.un ~ dgamma(9,33.33)

        for (i in 1:numberclus) {
            u[i] ~ dnorm(0,prec.u)
        }

        prec.u <- 1/s2u
        s2u ~ dgamma(.7,10.2)
    }
"

## Part 4: Write the JAGS model file to a text file
## and prepare the data, initial values, and parameters to monitor
writeLines(jags.code, con='jags_sim.txt')

data.jags <- list("yc"=yc, "cid"=cid, "yu"=yu, "sizeclus"=sizeclus,
                "sizeunclus"=sizeunclus, "numberclus"=numberclus)
parms.jags <- c('b0', 'b1', 's2u', 's2e.cl', 's2e.un', 'u')
inits <- function () {
    list("b0" = rnorm(1), "b1" = rnorm(1), "s2u" = runif(1),
                "s2e.cl" = runif(1), "s2e.un" = runif(1),
                "u" = rnorm(numberclus))
  }

## Part 5: Use rjags to estimate the model
jags.model <- jags.model(file="jags_sim.txt",
                           data = data.jags,
                           inits=inits,
                           n.chains=1,
                           n.adapt=1000)
update(jags.model,2000)
jags.results <- coda.samples(jags.model, variable.names=parms.jags,
```

```
                        n.iter=10000, n.thin=1)

## Part 6: Process the results
summary(jags.results)
posterior.mode(jags.results[[1]])
plot(jags.results)
```

Part 1: Load rjags

```
library(rjags)
```

The first step is to load the required `rjags` package.

Part 2: Read and prepare the data

```
data <- read.csv("partial_cluster_example.csv")
yc <- data$y[data$grouped==1]
cid <- data$j[data$grouped==1]
yu <- data$y[data$grouped==0]
sizeclus <- length(yc)
sizeunclus <- length(yu)
numberclus <- max(cid)
```

This section of the code reads the data from the csv file and create six objects.
1. `yc` = outcome variable in the clustered condition
2. `cid` = the cluster id variable in the clustered condition
3. `yu` = outcome variable in the unclustered condition
4. `sizeclus` = number of observations in the clustered condition
5. `sizeunclus` = number of observations in the unclustered condition
6. `numberclus` = number of clusters in the clustered condition

Part 3: Write the JAGS model file

The `jags.code` object is a string object containing the JAGS model statement. In JAGS the `~` indicates that a stochastic relationship whereas `<-` indicates a deterministic relationship.

```
for (i in 1:sizeunclus) {
    yu[i] ~ dnorm(b0, prec.un)
}
for (i in 1:sizeclus) {
    yc[i] ~ dnorm(mu.c[i], prec.cl)
    mu.c[i] <- b0 + b1 + u[cid[i]]
}
```

The first two `for` loops indicate the model for the data in the unclustered and clustered conditions, respectively. The data in the unclustered (`yu`) and clustered (`yc`) conditions both follow a normal distribution. Note that JAGS paramterizes the normal distribution with precisions rather than variances. The mean for the unclustered condition is `b0`. The mean for the clustered condition is the sum of `b0`, `b1`, and `u[cid[i]]` (i.e., the cluster effects).

```
b0 ~ dnorm(3,.44)
b1 ~ dnorm(0, 1)
prec.cl <- 1/s2e.cl
s2e.cl ~ dgamma(13,33.33)
prec.un <- 1/s2e.un
s2e.un ~ dgamma(9, 33.33)
```

The precisions for both conditions are unknown. Because we find it easier to state prior distributions for variances rather than precisions, we specify prior distributions for each of the residual variances using a gamma distribution (as described in the paper). For example, the prior for the residual variance in the unclustered condition is `s2e.un ~ dgamma(9, 33.33)`. We then compute the precision by taking the reciprocal of the variance `prec.un <- 1/s2e.un`. The next two lines do the same for the residual variance in the clustered condition.

```
for (i in 1:numberclus) {
    u[i] ~ dnorm(0,prec.u)
}

prec.u <- 1/s2u
s2u ~ dgamma(.7, 10.2)
```

Finally, the model specifies that the cluster effects are normally distribution with a mean of 0 and an unknown precision. As before, we specify a gamma prior on the variance of the cluster effects – `s2u ~ dgamma(.7, 10.2)` and then compute the precision by taking the reciprocal of the variance – `prec.u <- 1/s2u`.

Part 4: Write the JAGS model file to a text file and prepare the data, initial values, and parameters to monitor

```
writeLines(jags.code, con='jags_sim.txt')

data.jags <- list("yc"=yc, "cid"=cid, "yu"=yu, "sizeclus"=sizeclus,
                  "sizeunclus"=sizeunclus, "numberclus"=numberclus)
parms.jags <- c('b0', 'b1', 's2u', 's2e.cl', 's2e.un', 'u')
inits <- function () {
    list("b0" = rnorm(1), "b1" = rnorm(1), "s2u" = runif(1),
                  "s2e.cl" = runif(1), "s2e.un" = runif(1),
                  "u" = rnorm(numberclus))
  }
```

The `jags.code` object is a string of the JAGS model file and the `writeLines` function writes the contents of the `jags.code` object to a text file called `"jags_sim.txt"`. Estimation using `rjags` also requires:

- `data.jags`: a list of R objects that consitute the data
- `parms.jags`: a vector of parameters to monitor during the estimation
- `inits`: a function for computing initial or starting values for each of the parameters

Part 5: Use the rjags package to run the model

```
jags.model <- jags.model(file="jags_sim.txt",
                         data = data.jags,
                         inits=inits,
                         n.chains=1,
                         n.adapt=1000)
update(jags.model,2000)
jags.results <- coda.samples(jags.model, variable.names=parms.jags,
                       n.iter=10000, n.thin=1)
```

We first use the `jags.model` function to specify the model file (`"file=jags_sim.txt"`), the data source (`data=data.jags`), the initial values (`inits=inits`), number of chains (`n.chains=1`), and number of tuning interations (`n.adapt=1000`). The `update` function is used to specifying the number of burn-in iterations. The `coda.samples` function is used to run the final model and to specify the parameters to monitor (`variable.names=parms.jags`), the number of iterations (`n.iter=10000`), and the amount of thinning (`n.thin=1`). The `coda.samples` function will return the results in a way that allows for easy processing using the functions in the R package `coda`, which automatically loads when using the `rjags` package.

Part 6: Process the results

```
summary(jags.results)
plot(jags.results)
```

The final step is to process the MCMC results. This includes but is not limited to summarizing the results to get point and interval estimate – `summary(jags.results)` – and creating trace and density plots – `plot(jags.results)`. See the following books for discussion on processing and analyzing results from an MCMC chain:

- Jackman, S. (2009). *Bayesian analysis for the social sciences*. New York: Wiley
- Kruschke, J. K. (2011). *Doing Bayesian data analysis: A tutorial with R and BUGS*. Burlington, MA: Academic Press
- Lynch, S. M. (2007). *Introduction to applied Bayesian statistics and estimation for social scientists*. New York: Springer

Online Appendix B

Example Metropolis-Hastings Sampler

In this appendix we illustrate the creation of a Metropolis-Hastings (MH) sampler for a partially clustered multilevel model. Additionally, we include a simulated dataset so that readers can test the sampler we created or generate their own in their software of choice. Creating an MH sampler starts with specifying the likelihood for the data and prior distributions for all parameters. Then conditional distributions for all parameters are created because at each iteration the MH sampler draws values for a given parameter conditional on the current value of all other parameters. Finally, an algorithm is specified for sampling the parameters.

This appendix starts with specification of the likelihood and priors for all parameters in the model. We then multiply the likelihood and priors and construct the full conditional densities for all parameters. Then we describe the algorithm for computing the parameters. Finally, we present a MH sampler written in the open-source language Python.

In describing the likelihood, priors, and conditional distributions, we use the following notation.

Notation	Description
i	Indexes person within cluster (person in the unclustered condition)
j	Indexes cluster in the clustered condition
m_j	Cluster size for the jth cluster
c	Number of clusters in the clustered condition
n_1	Sample size in the clustered condition
n_0	Sample size in the unclustered condition
Y	Outcome variable
u_j	Cluster effect for cluster j
U	A 10×1 vector of all u_j
σ_u^2	Cluster variance
$\sigma_{e_1}^2$	Residual variance in clustered condition
$\sigma_{e_0}^2$	Residual variance in unclustered condition
θ	A 15×1 vector consisting all parameters in the model

The dataset includes four variables: (a) Y is a normally distributed outcome, (b) i is a person id, (c) j is a cluster id, and (d) X is a dummy variable coded 1 for the clustered condition and 0 for the unclustered condition.

Likelihood

The data in the clustered and unclustered conditions are both normally distributed. However, because the mean and variance of the normal distributions are unique to condition, we keep the likelihoods separate for each condition. For participants in the clustered condition, the joint likelihood is:

$$f\left(y_{ij} \mid \theta, X_{ij}=1\right)=2\pi^{-n_1/2}\left(\sigma_{e_1}^2\right)^{-n_1/2}\prod_{j=1}^{c}\prod_{i=1}^{m_j}\exp\left\{-\frac{\left(y_{ij}-b_0-b_1-u_j\right)^2}{2\sigma_{e_1}^2}\right\}.$$

For participants in the unclustered condition, the joint likelihood is:

$$f\left(y_{ij} \mid \theta, X_i=0\right)=2\pi^{-n_0/2}\left(\sigma_{e_0}^2\right)^{-n_0/2}\prod_{i=1}^{n_0}\exp\left\{-\frac{\left(y_i-b_0\right)^2}{2\sigma_{e_0}^2}\right\}.$$

Priors

The priors for each parameter are listed below. We used a normal distribution for the regression coefficients and the cluster effects. We used a gamma distribution for the variances.

$$f(b_0)=\frac{1}{\sqrt{2\pi}\sqrt{\sigma_{b_0}^2}}\exp\left\{-\frac{1}{2\sigma_{b_0}^2}\left(b_0\right)^2\right\}=N\left(0,\sigma_{b_0}^2\right)$$

$$f(b_1)=\frac{1}{\sqrt{2\pi}\sqrt{\sigma_{b_1}^2}}\exp\left\{-\frac{1}{2\sigma_{b_1}^2}\left(b_1\right)^2\right\}=N\left(0,\sigma_{b_1}^2\right)$$

$$f(u_j)=\frac{1}{\sqrt{2\pi}\sqrt{\sigma_u^2}}\exp\left\{-\frac{1}{2\sigma_u^2}\left(u_j\right)^2\right\}=N\left(0,\sigma_u^2\right)$$

$$f(\sigma_u^2)=\frac{1}{\Gamma(\alpha_u)\beta^{\alpha_u}}\left(\sigma_u^2\right)^{\alpha_u-1}\exp\left\{-\sigma_u^2/\beta_u\right\}=Gamma\left(\alpha_u,\beta_u\right)$$

$$f(\sigma_{e_1}^2)=\frac{1}{\Gamma(\alpha_{e_1})\beta^{\alpha_{e_1}}}\left(\sigma_{e_1}^2\right)^{\alpha_{e_1}-1}\exp\left\{-\sigma_{e_1}^2/\beta_{e_1}\right\}=Gamma\left(\alpha_{e_1},\beta_{e_1}\right)$$

$$f(\sigma_{e_0}^2)=\frac{1}{\Gamma(\alpha_{e_0})\beta^{\alpha_{e_0}}}\left(\sigma_{e_0}^2\right)^{\alpha_{e_0}-1}\exp\left\{-\sigma_{e_0}^2/\beta_{e_0}\right\}=Gamma\left(\alpha_{e_0},\beta_{e_0}\right)$$

Likelihood × Priors

Because we do not worry about the normalizing constant, we can remove constants and write:

$$
\begin{aligned}
f(\theta) \propto \quad & \left(\sigma_{e_1}^2\right)^{-n_1/2} \prod_{j=1}^{c} \prod_{i=1}^{m_j} \exp\left\{-\frac{\left(y_{ij}-b_0-b_1-u_j\right)^2}{2\sigma_{e_1}^2}\right\} \\
& \times\left(\sigma_{e_0}^2\right)^{-n_0/2} \prod_{i=1}^{n_0} \exp\left\{-\frac{\left(y_i-b_0\right)^2}{2\sigma_{e_0}^2}\right\} \\
& \times\frac{1}{\sqrt{\sigma^2{}_{b_o}}} \exp\left\{-\frac{1}{2\sigma^2{}_{b_o}}\left(b_0\right)^2\right\} \times\frac{1}{\sqrt{\sigma^2{}_{b_1}}} \exp\left\{-\frac{1}{2\sigma^2{}_{b_1}}\left(b_1\right)^2\right\} \\
& \times\prod_{j=1}^{c} \frac{1}{\sqrt{\sigma_u^2}} \exp\left\{-\frac{1}{2\sigma_u^2}\left(u_j\right)^2\right\} \times\left(\sigma_u^2\right)^{\alpha_u-1} \exp\left\{-\sigma_u^2/\beta_u\right\} \\
& \times\left(\sigma_{e_1}^2\right)^{\alpha_{e_1}-1} \exp\left\{-\sigma_{e_1}^2/\beta_{e_1}\right\} \times\left(\sigma_{e_0}^2\right)^{\alpha_{e_0}-1} \exp\left\{-\sigma_{e_0}^2/\beta_{e_0}\right\}
\end{aligned}
$$

Taking the log we get:

$$
\begin{aligned}
& -\frac{n_1}{2}\log\left(\sigma_{e_1}^2\right)+\sum_{j}^{c}\sum_{i}^{m_j}-\frac{1}{2\sigma_{e_1}^2}\left(y_{ij}-b_0-b_1-u_j\right)^2 \\
& -\frac{n_0}{2}\log\left(\sigma_{e_0}^2\right)+\sum_{i=1}^{n_0}-\frac{1}{2\sigma_{e_0}^2}\left(y_i-b_0\right)^2 \\
& -\frac{1}{2}\log\left(\sigma_{b_0}^2\right)-\frac{1}{2\sigma_{b_0}^2}\left(b_0\right)^2-\frac{1}{2}\log\left(\sigma_{b_1}^2\right)-\frac{1}{2\sigma_{b_1}^2}\left(b_1\right)^2 \\
& -\frac{c}{2}\log\left(\sigma_u^2\right)-\frac{1}{2\sigma_u^2}\sum_{j=1}^{c}u_j^2+\left(\alpha_u-1\right)\log\left(\sigma_u^2\right)-\frac{\sigma_u^2}{\beta_u} \\
& +\left(\alpha_{e_1}-1\right)\log\left(\sigma_{e_1}^2\right)-\frac{\sigma_{e_1}^2}{\beta_{e_1}}+\left(\alpha_{e_0}-1\right)\log\left(\sigma_{e_0}^2\right)-\frac{\sigma_{e_0}^2}{\beta_{e_0}}
\end{aligned}
$$

Complete conditionals

We now simplify $\log(f(\theta))$ to form the complete conditional density for each parameter. In forming the conditional densities, we remove any term that does not involve the parameter of interest, as those terms will be constants in the MH algorithm. Letting $[\theta]$ stand for the log complete conditional with constants removed, we get:

$$\left[b_0\right] = -\frac{1}{2\sigma_{e_1}^2}\sum_{j}^{c}\sum_{i}^{m_j}\left(y_{ij} - b_0 - b_1 - u_j\right)^2 - \frac{1}{2\sigma_{e_0}^2}\sum_{i=1}^{n_0}\left(y_i - b_0\right)^2 - \frac{1}{2\sigma_{b_0}^2}\left(b_0\right)^2$$

$$\left[b_1\right] = -\frac{1}{2\sigma_{e_1}^2}\sum_{j}^{c}\sum_{i}^{m_j}\left(y_{ij} - b_0 - b_1 - u_j\right)^2 - \frac{1}{2\sigma_{b_1}^2}\left(b_1\right)^2$$

$$\left[u_j\right] = -\frac{1}{2\sigma_{e_1}^2}\sum_{i=1}^{m_j}\left(y_{ij} - b_0 - b_1 - u_j\right)^2 - \frac{1}{2\sigma_u^2}u_j^2$$

$$\left[\sigma_u^2\right] = -\frac{c}{2}\log\left(\sigma_u^2\right) - \frac{1}{2\sigma_u^2}\sum_{j=1}^{c}u_j^2 + \left(\alpha_u - 1\right)\log\left(\sigma_u^2\right) - \frac{\sigma_u^2}{\beta_u}$$

$$\left[\sigma_{e_1}^2\right] = -\frac{n_1}{2}\log\left(\sigma_{e_1}^2\right) - \frac{1}{2\sigma_{e_1}^2}\sum_{j}^{c}\sum_{i}^{m_j}\left(y_{ij} - b_0 - b_1 - u_j\right)^2 + \left(\alpha_{e_1} - 1\right)\log\left(\sigma_{e_1}^2\right) - \frac{\sigma_{e_1}^2}{\beta_{e_1}}$$

$$\left[\sigma_{e_0}^2\right] = -\frac{n_0}{2}\log\left(\sigma_{e_0}^2\right) - \frac{1}{2\sigma_{e_0}^2}\sum_{i=1}^{n_0}\left(y_i - b_0\right)^2 + \left(\alpha_{e_0} - 1\right)\log\left(\sigma_{e_0}^2\right) - \frac{\sigma_{e_0}^2}{\beta_{e_0}}$$

Algorithm

Let θ equal a parameter vector with 15 elements (i.e., the 15 parameters from above) indexed by *k*. Let *M* equal the total number of *m* iterations of the MH sampler algorithm. This algorithm assumes a symmetric proposal distribution

1. Set *m* = 0. Select starting values for all elements of θ. Starting values can be based on maximum likelihood estimates or can be an arbitrary value.
2. Set *m* = *m* + 1.
3. Draw a candidate value $\theta_{k=1}^{c}$ from a proposal distribution. A normal distribution is commonly used as a proposal distribution.
4. Compute the difference (D) between the conditional density for $\theta_{k=1}$ at $\theta_{k=1}^{c}$ and $\theta_{k>1}^{m-1}$ and the conditional density for $\theta_{k=1}$ at $\theta_{k=1}^{m-1}$ and $\theta_{k>1}^{m-1}$.
5. Compare D to the log of a random draw ($\log(unif_r)$) from a Uniform(0,1). If D > $\log(unif_r)$ then set $\theta_{k=1}^{m} = \theta_{k=1}^{c}$. If D < $\log(unif_r)$, then $\theta_{k=1}^{m} = \theta_{k=1}^{m-1}$.

6. Draw a candidate value $\theta^{c}_{k=2}$ from a proposal distribution.
7. Compute the difference (D) between the condition density for $\theta_{k=2}$ at $\theta^{c}_{k=2}$, $\theta^{m}_{k<2}$, and $\theta^{m-1}_{k>2}$ and the conditional density for $\theta_{k=2}$ at $\theta^{m-1}_{k=2}$, $\theta^{m}_{k<2}$, and $\theta^{m-1}_{k>2}$.
8. Compare D to the log of a random draw ($\log(unif_r)$) from a Uniform(0,1). If D > $\log(unif_r)$ then set $\theta^{m}_{k=2} = \theta^{c}_{k=2}$. If D < $\log(unif_r)$, $\theta^{m}_{k=2} = \theta^{m-1}_{k=2}$.
9. Repeat steps 6-8 for elements k = 3-15 of θ.
10. Repeat steps 2-9 until $m = M$

Python Code

Any number of programming languages could be used to implement this sampler, including languages in statistics packages such as R, SAS, or Stata or general programming lanuages such as Fortran, C, or Java. We used the general programming language Python to implement this sampler. We chose Python because it is open source, is cross-platform, is reasonably fast, has excellent random number generators and linear algebra routines, can easily generate plots for examining convergence of the MH sampler, and is a relatively accessible programming language for scientists. It is also possible to weave in Fortran or C code into Python to speed up time-consuming portions of the sampler (i.e., time-consuming loops), although we did not take advantage of this feature in this sampler as it was already fast enough for our purposes.

This sampler requires the NumPy library, which is the best numerical library in Python (http://numpy.scipy.org/).

```
import numpy as np

#Number of iterations
m = 50000

#Number of clusters
c = 10

data = np.genfromtxt('run1.csv', delimiter=",", unpack = True, skip_header=1)

##Total sample size
N = len(data[0])

##Variables for all subjects
pid = data[0].reshape(N,1)
gid = data[1].reshape(N,1)
tx = data[2].reshape(N,1)
y = data[3].reshape(N,1)

##Sample size in the clustered condition
n1 = len(tx[np.where(tx==1)])
##Sample size in the unclustered condition
n0 = len(tx[np.where(tx==0)])
```

```
##data for clustered condition
Y1 = y[np.where(tx==1)].reshape(n1,1)
##data for the unclustered condition
Y0 = y[np.where(tx==0)].reshape(n0,1)

##group id for clustered condtion only
gidclus = gid[np.where(gid<11)].reshape(n1,1)

##Design matrix for cluster effect
##Used in the cluslike function below
Z = np.zeros((n1,10))
for row, j in enumerate(gid):
    if j <= c:
        Z[row,j[0]-1]=1

#Starting values for variances are 1 and for u's and reg coef are 0
s2e1 = np.ones((m,1)) #Initialize res variance for clustered cond
s2e0 = np.ones((m,1)) #Initialize res variance for unclustered cond
s2u = np.ones((m,1)) #Initialize matrix for cluster variance
U = np.zeros((m,c))  #Initialize cluster effects
b0 = np.zeros((m,1)) #Initialize regression coefficients
b1 = np.zeros((m,1))  #Initialize regression coefficients

#Matrices for holding information about the acceptance rate
b0rate = np.zeros((m,1))
b1rate = np.zeros((m,1))
urate = np.zeros((m,c))
s2urate = np.zeros((m,1))
s2e1rate = np.zeros((m,1))
s2e0rate = np.zeros((m,1))

###Functions used in the sampler
###See the docstrings for a description for each parameter

###############
def ucond(Y1, b0, b1, uj, s2e1, s2u, cid):
       """Conditional Density for u_j

       Keyword arguments:
       Y1 -- data from the clustered condition only
       b0 -- intercept
       b1 -- slope
       uj -- cluster effect for group j
       s2e1 -- residual variance clustered condition
       s2u -- cluster variance
       cid -- cluster id: loop over this value consistent with id's in data

       """
       Yu = Y1[np.where(gidclus==cid)].reshape(10,1)
       likeu = (-.5*(s2e1**-1))*np.sum((Yu-b0-b1-uj)**2)
       pr_u = (-.5*(s2u**-1))*(uj**2)
       return likeu + pr_u
```

```
############
def cluslike(Y1,b0,b1,U,s2e1):
      """Calculation of the Likelihood in the Clustered Condition

      Y1 -- data from the clustered condition only
      b0 -- intercept
      b1 -- slope
      U -- vector of cluster effects for all clusters
      s2e1 -- residual variance clustered condition

      """
      U = U.reshape(10,1)
      Uj = np.dot(Z,U)
      likeclus = (-.5*(s2e1**-1))*np.sum((Y1 - b0 - b1 - Uj)**2)
      return likeclus

############
def uncluslike(Y0, b0, s2e0):
      """Calculation of the Likelihood for the Unclusted Condition

      Y0 -- data in the unclusted condition only
      b0 -- intercept
      s2e0 -- residual variance in the unclustered condition

      """
      likeunclus = (-.5*(s2e0**-1))*np.sum((Y0-b0)**2)
      return likeunclus


################
def b0cond(Y1, Y0, b0, b1, U, s2e1, s2e0, s2b0):
      """Calculation of the full conditional for b0

      Y1 -- data from the clustered condition only
      Y0 -- data from the unclustered condition only
      b0 -- intercept
      b1 -- slope
      U -- vector of cluster effects for all clusters
      s2e1 -- residual variance for clustered condition
      s2e0 -- residual variance for unclustered condition
      s2b0 -- prior variance for the intercept (mean = 0)

      """
      pr_b0 = (-.5*(s2b0**-1))*(b0**2)
      likeclus = cluslike(Y1=Y1, b0=b0, b1=b1, U=U, s2e1=s2e1)
      likeunclus = uncluslike(Y0=Y0, b0=b0, s2e0=s2e0)
      return likeclus + likeunclus + pr_b0


##################
def b1cond(Y1,b0,b1,U,s2e1,s2b1):
      """Calculation of the full conditional for b1
```

```
        Y1 -- data from the clustered condition only
        b0 -- intercept
        b1 -- slope
        U -- vector of cluster effects for all clusters
        s2e1 -- residual variance for clustered condition
        s2b1 -- prior variance for the slope (mean = 0)

        """
        pr_b1 = (-.5*(s2b1**-1))*(b1**2)
        likeclus = cluslike(Y1=Y1, b0=b0, b1=b1, U=U, s2e1=s2e1)
        return likeclus + pr_b1

def s2ucond(c, s2u, U, alphau, betau):
        """Calculation of the full conditional for s2u

        c -- number of clusters
        s2u -- cluster variance
        U -- vector of cluster effects for all clusters
        alphau -- prior shape parameter for s2u
        betau -- prior scale parameter for s2u

        """
        part1 = ((-.5*c)*np.log(s2u))
        part2 = (-.5*(s2u**-1))*np.sum(U**2)
        part3 = (alphau-1)*np.log(s2u)
        part4 = (-1*(s2u)*(betau**-1))
        return part1 + part2 + part3 + part4

#########
def s2e1cond(n1, s2e1, Y1, b0, b1, U, alphae1, betae1):
        """Calculation of the full conditional for s2e1

        n1 -- sample size in the clustered condition
        s2e1 -- residual variance for clustered condition
        Y1 -- data from the clustered condition only
        b0 -- intercept
        b1 -- slope
        U -- vector of cluster effects for all clusters
        alphae1 -- prior shape parameter for s2e1
        betae1 -- prior scale parameter for s2e1

        """
        part1 = ((-.5*n1)*np.log(s2e1))
        part2 = cluslike(Y1=Y1, b0=b0, b1=b1, U=U, s2e1=s2e1)
        part3 = (alphae1-1)*np.log(s2e1)
        part4 = (-1*(s2e1)*(betae1**-1))
        return part1 + part2 + part3 + part4


##########
def s2e0cond(n0, s2e0, Y0, b0, alphae0, betae0):
        """Calculation of the full conditional for s2e0
```

```
	n0 -- sample size in the unclustered condition
	s2e0 -- residual variance for unclustered condition
	Y0 -- data from the unclustered condition only
	b0 -- intercept
	alphae0 -- prior shape parameter for s2e0
	betae0 -- prior scale parameter for s2e0

	"""
	part1 = ((-.5*n0)*np.log(s2e0))
	part2 = uncluslike(Y0=Y0, b0=b0, s2e0=s2e0)
	part3 = (alphae0-1)*np.log(s2e0)
	part4 = (-1*(s2e0)*(betae0**-1))
	return part1 + part2 + part3 + part4


##Prior variance for b0, b1 -- respectively
s2b0=10
s2b1=10

##shape and scale prior values for s2u
alphau=.7
betau=.098

##shape and scale prior values for s2e1
alphae1=13
betae1=.03

##shape and scale prior values for s2e0
alphae0=9
betae0=.03


###Scale of the proposal densities (i.e., standard deviation
###of the proposal distribution)
b0scale = .3
b1scale = .4
uscale = np.zeros([10,1])+1
s2uscale = .275
s2e1scale = .4
s2e0scale = .4


##begin MH sampling

for i in range(1, m):

	#update b0
	b0[i] = b0[i-1] + np.random.normal(loc=0, scale=b0scale, size=1)
	acc=1

	if (b0cond(Y1=Y1, Y0=Y0, b0=b0[i], b1=b1[i-1], U=U[i-1,], s2e1=s2e1[i-1],
s2e0=s2e0[i-1], s2b0=s2b0) - b0cond(Y1=Y1, Y0=Y0, b0=b0[i-1], b1=b1[i-1], U=U[i-1,],
```

```
s2e1=s2e1[i-1], s2e0=s2e0[i-1], s2b0=s2b0)) < np.log(np.random.uniform(low=0, high=1,
size=1)):
              b0[i] = b0[i-1]
              acc = 0
       b0rate[i] = acc

       #update b1
       b1[i] = b1[i-1] + np.random.normal(loc=0, scale=b1scale, size=1)
       acc = 1


       if (b1cond(Y1=Y1, b0=b0[i], b1=b1[i], U=U[i-1,], s2e1=s2e1[i-1], s2b1=s2b1) -
b1cond(Y1=Y1, b0=b0[i], b1=b1[i-1], U=U[i-1,], s2e1=s2e1[i-1], s2b1=s2b1)) <
np.log(np.random.uniform(low=0, high=1, size=1)):
              b1[i] = b1[i-1]
              acc = 0
       b1rate[i] = acc

       #update u's
       for j in range(0,c):
              U[i,j] = U[i-1,j] + np.random.normal(loc=0, scale=uscale[j], size=1)
              acc = 1

              if (ucond(Y1=Y1, b0=b0[i], b1=b1[i], uj=U[i,j], s2e1=s2e1[i-1],
s2u=s2u[i-1], cid=j+1) - ucond(Y1=Y1, b0=b0[i], b1=b1[i], uj=U[i-1,j], s2e1=s2e1[i-1],
s2u=s2u[i-1], cid=j+1)) < np.log(np.random.uniform(low=0, high=1, size=1)):
                     U[i,j] = U[i-1, j]
                     acc=0
              urate[i,j]=acc

       #update s2u
       s2u[i]=s2u[i-1] + np.random.normal(loc=0, scale=s2uscale, size=1)
       acc=1

       if s2u[i] < 0:
              s2u[i]=s2u[i-1]
              acc=0
       elif (s2ucond(c=c, s2u=s2u[i], U=U[i,], alphau=alphau, betau=betau) -
s2ucond(c=c, s2u=s2u[i-1], U=U[i,], alphau=alphau, betau=betau)) <
np.log(np.random.uniform(low=0, high=1, size=1)):
              s2u[i]=s2u[i-1]
              acc=0
       s2urate[i]=acc

       #update s2e1
       s2e1[i]=s2e1[i-1] + np.random.normal(loc=0, scale=s2e1scale, size=1)
       acc = 1

       if s2e1[i] < 0:
              s2e1[i]=s2e1[i-1]
              acc=0
       elif (s2e1cond(n1=n1, s2e1=s2e1[i], Y1=Y1, b0=b0[i], b1=b1[i], U=U[i,],
alphae1=alphae1, betae1=betae1) - s2e1cond(n1=n1, s2e1=s2e1[i-1], Y1=Y1, b0=b0[i],
```

```
b1=b1[i], U=U[i,], alphae1=alphae1, betae1=betae1)) < np.log(np.random.uniform(low=0,
high=1, size=1)):
              s2e1[i]=s2e1[i-1]
              acc = 0
       s2e1rate[i]=acc

       #update s2e0
       s2e0[i]=s2e0[i-1] + np.random.normal(loc=0, scale=s2e0scale, size=1)
       acc = 1

       if s2e0[i] < 0:
              s2e0[i]=s2e0[i-1]
              acc=0
       elif (s2e0cond(n0=n0, s2e0=s2e0[i], Y0=Y0, b0=b0[i], alphae0=alphae0,
betae0=betae0) - s2e0cond(n0=n0, s2e0=s2e0[i-1], Y0=Y0, b0=b0[i], alphae0=alphae0,
betae0=betae0)) < np.log(np.random.uniform(low=0, high=1, size=1)):
              s2e0[i]=s2e0[i-1]
              acc = 0
       s2e0rate[i]=acc

###End sampling
```