*

July 13, 2016

The results below are generated from an R script.

```
# gwmcmc.R
# collection of R functions for using the Goodman & Weare (2010) ensemble MCMC
# method. For details see:
# J. Goodman & J. Weare, 2010, Ensemble samplers with affine invariance,
#           CAMCOS, v5, pp65-80
# F. Hou, J. Goodman, D. Hogg, J. Weare, C. Schwab, 2012, ApJ, v745, 198
# D. Foreman-Mackey, 2013, PASJ (http://arxiv.org/abs/1202.3665)
```

```
gw.mcmc <- function(posterior,
                    theta.0,
                    nsteps = 1E4,
                    nwalkers = 100,
                    burn.in = 2E3,
                    update = 5,
                    chatter = 1,
                    thin = NULL,
                    scale.init = NULL,
                    cov.init = NULL,
                    walk.rate = 0,
                    atune = 2.0,
                    stune = NULL,
                    merge.walkers = TRUE, ...) {

# gw.mcmc - Ensemble Markov Chain Monte Carlo sampler
# Inputs:
#   posterior  - (function) name of the log(densiy) function to sampling from
#   theta.0    - (vector) initial values of the M variables
#   nsteps     - (integer) total number of samples required
#   nwalkers   - (integer) number of 'walkers' (default: 100; should be > M)
#   burn.in    - (integer) the 'burn-in' period for the walkers
#   update     - (integer) print a progress update after <update> steps
#   chatter    - (integer) how verbose is the output?
#                    (0=quiet, 1=normal, 2=verbose)
#   atune      - (float) set the scale size of the random jumps (default: 2.0)
#   thin       - (integer) keep only every <thin> sample
#   scale.init - (float) variances for randomising walkers' start positions
#   cov.init   - (matrix) specify exact covariance matrix for randomising
#                    walkers' start positions
#   merge.walkers - TRUE/FALSE combine output from all walkers into one
#   walk.rate  - Fraction of moves (0-1) to make use of the "walk move"
#                  (default: NULL)
```

*This report is automatically generated with the R package **knitr** (version 1.12.3).

```
#   ...         - (anything else) other arguments needed for posterior function
#
# Value:
#  A list with components
#   theta      - (array) <nsteps> samples from M-dimensional posterior
#                  [nsteps rows, M columns]
#   func       - (string) name of posterior function sampled
#   lpost      - (vector) nsteps values of the LogPosterior density at each
#                  sample position
#   method     - sampling method uses (=gwmcmc)
#   Nwalkers   - number of walkers used
#
# Description:
# A simple implementation of the ensemble MCMC sampler proposed by Goodman &
# Weare (2010). Given some function to compute the log of an (un-normalised)
# M-dimensional posterior density function (PDF) this will produce <nsteps>
# samples of M-dimensional vectors drawn from the PDF.
#
# It works by running a number <nwalkers> of 'walkers' through the M-dimensional
# space. At initialisation, all walkers begin near some start point (specified
# by theta.0) but have their positions randomised (using a multivariate normal
# distribution). The ensemble of walkers then updates each cycle. The updating
# is done by the 'stretch move'. The stretch move is handled by seperate
# function (stretch.move).
#
# There is an initial period (called the 'burn-in' period) of <burn.in> cycles
# that from the beginning of the chain that is discarded. This is to help remove
# memory of the start positions. After a few cycles the ensemble should have
# found regions of high posterior density even if started from a region of low
# density.
#
# The chain is then run until there are at least <nsteps> total samples. Each
# cycle produces <nwalkers> samples (one from each walker). So we run for nrows
# = ceiling(nsteps/nwalkers) cycles after the burn-in period. Any extra cycles
# can be discarded.
#
# The user has the option to 'thin' the output. This involves keeping only a
# subset of the full chain. If <thin> is equal to 5 then we keep only every 5th
# cycle. This helps remove autocorrelation between successive cycles. But modern
# MCMC practice would advice against this as it throws away perfectly good
# samples.
#
# Once we have enough samples the output from all walkers is merged into a
# single nsteps (rows) * M (columns) array.
#
# History:
#  04/04/16 - v0.1 - First working version
#  14/04/16 - v0.2 - Added walk.move as an optional update step
#  28/04/16 - v0.3 - Added saetfy checks, fixed sqrt(S) error in
#                    walk move, reflow comments
#  11/07/16 - v0.4 - Changed output to a list with additional information
#
# Simon Vaughan, University of Leicester
```

```
# Copyright (C) 2016 Simon Vaughan

  # check the input arguments
  if (missing(theta.0)) stop('Must specify theta.0 start position.')
  if (missing(posterior)) stop('Must specify name of posterior function')
  if (!exists('posterior'))
    stop('The specified log density function does not exist.')

  # dimensions of the PDF
  M <- length(theta.0)

  # number of 'walkers'
  if (nwalkers <= M) stop('Increase the number of walkers.')

  # ensure the number of steps, walkers, etc. are integers
  nsteps <- as.integer(nsteps)
  nwalkers <- as.integer(nwalkers)
  burn.in <- as.integer(burn.in)

  # number of interations needed
  nrows.keep <- ceiling(nsteps / nwalkers)
  if (nrows.keep < 10) stop('Make nsteps larger')
  nrows.burnin <- ceiling(burn.in / nwalkers)
  ncycles <- nrows.keep + nrows.burnin

  # the scale factor is used to define the variance of each variable and is only
  # used for randomising their starting values
  if (is.null(scale.init)) scale.init <- rep(1e-4, M)
  if (length(scale.init) != M) stop('scale.init has wrong length')
  if (min(scale.init <= 0)) stop('scale.init should be >0 everywhere.')

  # check the parameter that sets the size of the random jumps
  atune <- as.numeric(atune)
  if (atune <= 1) stop('Parameter atune should be >1.')

  # set the size of the complementary sample
  if (is.null(stune)) stune <- as.integer( M+1 )
  if (!is.integer(stune)) warning('stune is not an integer.')

  # initialise the array for output. There are two additional 'columns': The M+1
  # column stores the accept/reject flag (0=rejected, 1=accepted proposal at
  # each update). This is useful for tracking the acceptance rate. The M+2
  # column stores the log posterior (PDF) values at the current position of the
  # walker. This saves recomputing the posterior density at the current position
  # when evaluating the accept probability.
  theta <- array(NA, dim=c(ncycles, nwalkers, M+2))

  # initialise each walker with a slightly different position. The starting
  # positions are randomised using a M-dimensional t distribution centred on
  # theta.0. If a matrix cov.init is supplied then use this as the covariance
  # matrix. Otherwise create a diagonal covariance matrix and set the variances
  # to be a small fraction (scale.init) of the (absolute value of) the start
  # position for each variable. If the starting position is exactly zero then
```

```
# use 1E-6 instead.
vars <- scale.init * theta.0^2
vars <- pmax(vars, 1E-13)
cov <- diag(vars)
if (is.matrix(cov.init)) cov <- cov.init
theta.now <- mvtnorm::rmvnorm(nwalkers, mean=theta.0, sigma=cov)

# add two extra columns to track: the acceptances, the log posterior densities
theta.now <- cbind(theta.now, rep(NA, nwalkers), rep(NA, nwalkers))

# to get the calculation started, evaluate the posterior density at
# these starting positions
for (i in 1:nwalkers) {
  theta.now[i, M+2] <- posterior(theta.now[i, 1:M], ...)
}
if (!all(is.finite(theta.now[,M+2]))) {
  print(theta.now[,1:M])
  print(theta.now[,M+2])
  stop('Non-finite start values for target PDF.')
}

# Now, theta.now is a 2D array with dimensions nwalkers * M+2.
# each row is the current position of one walker, i.e. an
# M-dimensional vector (plus the accept/reject flag, and the log(PDF)).
if (chatter > 1) {
  cat('-- Dimensions of theta.now array:', dim(theta.now), fill=TRUE)
  cat('-- Dimensions of theta array:    ', dim(theta), fill=TRUE)
  cat('-- Number of whole cycles:       ', ncycles, fill=TRUE)
  cat('-- No. burn-in cycles:           ', nrows.burnin, fill=TRUE)
}

# record the start time of the main loop
start.time <- proc.time()

# main loop over ncycles, each iteration updates all walkers
# the first (burn.in) steps are the 'burn-in' period
for (i in 1:ncycles) {

  # should we to a "stretch" (default) or a "walk" move?
  do.walk <- FALSE
  if (walk.rate > 0) {
    do.walk <- (i %% walk.rate == 0)
  }

  # update the walkers' position to step i using their positions
  # at step i-1
  if (do.walk == TRUE) {
    theta.now <- walk.move(posterior, theta.now,
                           chatter = chatter, stune, ...)
  } else {
    theta.now <- stretch.move(posterior, theta.now,
                              chatter = chatter, atune, ...)
  }
```

```
  # safety check
  if (!all(is.finite(theta.now[, M+2]))) {
    mask <- !is.finite(theta.now[, M+2])
    cat(theta.now[mask,])
    stop('Non-finite value in theta.now.')
  }

  # save the current position of each walker
  theta[i, , ] <- theta.now

  # progress report to user if requested
    i.count <- 1
    if (chatter > 0) {
    if (i %% update == 0) {
      accept.rate <- mean( theta[i.count:i,,M+1], na.rm=TRUE )
      cat("\r-- Cycle", i, "of", ncycles, ". Acceptance rate:",
          signif(accept.rate*100, 2), "%")
    }
    if (i == ncycles) cat('', fill=TRUE)
    if (i == nrows.burnin) {
      cat(' - Finished burn-in', fill=TRUE)
      i.count <- nrows.burnin+1
    }
  }
} # end of main loop (i = 1, ncycles)

# record the end time of the main loop
end.time <- proc.time()

# strip off the burn-in period and keep only nsteps
nrows <- nrows.keep
theta <- theta[(1:nrows) + nrows.burnin, , ]

# thin the output by keeping only every few rows
if (!is.null(thin)) {
  nrow.keep <- floor(nrows / thin)
  mask <- (1:nrow.keep) * thin
  theta <- theta[mask,,]
}

# Strip off the acceptance and log(posterior) columns
accept <- theta[, , M+1]
lpost <- as.vector( theta[, , M+2])
theta <- theta[, , 1:M]

# check the acceptance rate (column M+1).
# Also strip off the log(posterior) values which are no longer needed.
accept.rate <- mean(accept, na.rm = TRUE)
if (chatter > 0) {
  print(end.time - start.time)
  cat('\n-- Final acceptance rate: ', accept.rate, fill = TRUE)
  if (accept.rate < 0.05) {
    cat('-- Low acceptance rate. Consider the following suggestions:',
        fill = TRUE)
```

```
      cat('-- 1. Increase number of walkers: nwalkers.', fill = TRUE)
      cat('-- 2. Lower the jump scale parameter: atune.', fill = TRUE)
      cat('-- 3. Adjust the start position: theta.0,', fill = TRUE)
      cat('-- 4. increasing the variances of the start point
           randomisation: scale.init or cov.init.', fill = TRUE
      )
    }
  }

  # reshape the array from [nrows, M, nwalkers] to [nrows*nwalkers, M]
  # so each column is one variable, each row is one sample from the M
  # M-dimensional distribution.
  if (merge.walkers == TRUE) {
    nrows <- dim(theta)[1]
    theta <- matrix(theta, nrow = nwalkers * nrows, ncol = M, byrow = FALSE)
  }

  # return the final array
  return(list(theta = theta,
              func = deparse(substitute(posterior)),
              lpost = lpost,
              method = "gw.mcmc",
              nwalkers = nwalkers))
}
```

```
# stretch.move - update an ensemble of 'walkers' using the 'stretch move'
#
# Inputs:
#   posterior  - (function) name of the log(posterior) function to sample from
#   theta      - (array) nwalkers (rows) * M+2 (columns)
#                  the current position of each walker
#   a          - (float) set the scale size of the random jumps
#   ...        - (anything else) any other arguments needed for posterior
#
# Value:
#   theta     - (array) nwalkers (rows) * M+2 (columns)
#                 updated position of each walker
#
# Description:
# A simple implementation of the ensemble MCMC sampler proposed by Goodman &
# Weare (2010). Given some function to compute the log of an (un-normalised)
# M-dimensional posterior density function (PDF) this will produce <nsteps>
# samples of M-dimensional vectors drawn from the PDF.
#
# At input the array <theta> gives the current position of each walker. There
# are <nwalkers> rows, one for each walker. Each row has M+2 columns. The first
# 1:M columns are the position of each walker, an M-dimensional vector. On
# output the position of each walker is updated. And the M+1 column is assigned
# 0 (rejected) or 1 (accepted) depending on whether the walkers' position is
# updated this cycle (accept/reject the proposed update position). The M+2
# column contains the log(posterior density) at the walkers' current position
# (after update).
#
```

```
# Each walkers' position is updated in turn. The position of walker j is updated
# by randomly selecting another walker k from the ensemble (the complementary
# walker) and moving along the line joining the current position of walker j to
# walker k. The jump size is random and has the distribution suggested by
# Goodman & Weare (eqn 9). This updated position for walker i is then accepted
# or rejected with a probability that depends on the ratio of the posterior
# densities at the current and the proposed new position. If the proposal is
# rejected, walker j remains at its current position. Once every walker has been
# updated (j = 1, 2, ..., nwalkers) we move to cycle i+1 and repeat the update
# step.
#
# The random numbers (the size of the jump z and the uniform variate u used to
# randomly choose accept/reject) are computed before the loop over walkers
# begins. This is to make the code a little more clear and efficient.
#
# History:
#  04/04/16 - First working version
#
# Simon Vaughan, University of Leicester
# Copyright (C) 2016 Simon Vaughan
```

```
stretch.move <- function(posterior, theta, a, chatter=0, ...) {

  # number of walkers to treat
  nwalkers <- NROW(theta)

  # number of dimensions of each walker
  M <- as.integer( NCOL(theta)-2 )

  # an index of walkers
  walkers <- 1:nwalkers

  # draw the random z values and u values
  # The z density is eqn 9 of Goodman & Weare (2010)
  z <- (1/a) * ( 1 + (a-1)*runif(nwalkers) )^2
  u <- runif(nwalkers)

  # loop over each walker, updating its position
  for (j in 1:nwalkers) {

    # for walker j, randomly select a complementary walker k
    k <- sample(walkers[-j], 1)

    # present position of the jth walker
    X.pres <- theta[j, 1:M]

    # present position of the complementary (kth) walker
    X.comp <- theta[k, 1:M]

    # proposed new position for the jth walker
    # This is eqn 7 of Goodman & Weare (2010)
    X.prop <- (1-z[j])*X.comp + z[j]*X.pres
```

```
    # evalute the log PDF at the current and proposed positions.
    # The value at the current position was evaluated in the previous
    # step, so we reuse this value. Only the value at the proposed new
    # position needs evaluating.
    p.pres <- theta[j, M+2]
    p.prop <- posterior(X.prop, ...)

    # now evaluate the acceptance probability
    logP <- (M-1)*log(z[j]) + p.prop - p.pres
    lopP <- max(logP, 0)
    if ( exp(logP) >= u[j] ) {
      theta[j, 1:M] <- X.prop     # accept
      theta[j, M+1] <- 1
      theta[j, M+2] <- p.prop
    } else {
      theta[j, 1:M] <- X.pres     # reject
      theta[j, M+1] <- 0
      theta[j, M+2] <- p.pres
    }

  } # end of loop (j = 1, nwalkers)

  if (chatter > 5) {
    cat('-- Sretch move accept rate:', mean(theta[,M+1]))
  }

  # return the updated array
  return(theta)

}
```

```
# walk - update an ensemble of 'walkers' using the 'walk move'
#
# Inputs:
#   theta      - (array) nwalkers (rows) * M+2 (columns) current position
#                  of each walker
#   S          - (integer) size of the complementary sample
#   posterior  - (function) name of the log(posterior) function we are
#                  sampling from
#   ...        - (anything else) any other arguments needed for the
#                  posterior function
#
# Value:
#   theta     - (array) nwalkers (rows) * M+2 (columns) updated position
#                 of each walker
#
# Description:
# A simple implementation of the ensemble MCMC sampler proposed by Goodman &
# Weare (2010). Given some function to compute the log of an (un-normalised)
# M-dimensional posterior density function (PDF) this will produce <nsteps>
# samples of M-dimensional vectors drawn from the PDF.
#
# At input the array <theta> gives the current position of each walker. There
```

```
# are <nwalkers> rows, one for each walker. Each row has M+2 columns. The first
# 1:M columns are the position of each walker, an M-dimensional vector. On
# output the position of each walker is updated. And the M+1 column is assigned
# 0 (rejected) or 1 (accepted) depending on whether the walkers' position is
# updated this cycle (accept/reject the proposed update position). The M+2
# column contains the log(posterior density) at the walkers' current position
# (after update).
#
# Each walkers' position is updated in turn. The position of walker j is updated
# by randomly selecting a subset of walkers from the ensemble (the complementary
# sample). From the sample we effectively compute the covariance matrix, and use
# this to define a multivariate Normal, with the mean as the current position of
# walker j. We then propose a new position for walker j by drawing from this
# multivariate Normal.
#
# In practice we randomly select M+1 walkers' positions (where M is the number
# of variables, or dimensions of the problem), from the set of all walkers
# excluding walker j. With M+1 positions we form a 'simplex'. (Although, by
# setting the S parameter one can increase the size of the complementary sample
# if desired, forming an S-polytope.) We then compute the mean position of the
# complementary sample <x> and the displacements of each of its walkers from
# this mean: delta_k = (x_k - <x>). We then produce M+1 univariate, Normal
# random numbers z_k and compute W = sum_k z_k * delta_k. So W is a random
# displacement made from a weighted sum of the displacements (of each walker in
# the complementary sample from their mean), with random weights. (See Goodman &
# Weare eqn 11.)
#
# This updated position for walker i is then accepted or rejected with a
# probability that depends on the ratio of the posterior densities at the
# current and the proposed new position. If the proposal is rejected, walker j
# remains at its current position. Once every walker has been updated (j = 1, 2,
# ..., nwalkers) we move to cycle i+1 and repeat the update step.
#
# The random numbers (the size of the jump z and the uniform variate u used to
# randomly choose accept/reject) are computed before the loop over walkers
# begins. This is to make the code a little more clear and efficient.
#
# History:
#  14/04/16 - First working version
#
# Simon Vaughan, University of Leicester
# Copyright (C) 2016 Simon Vaughan
```

```
walk.move <- function(posterior, theta, S=NULL, chatter=0, ...) {

  # number of walkers to treat
  nwalkers <- NROW(theta)

  # number of dimensions of each walker
  M <- as.integer( NCOL(theta)-2 )

  # set the size of the complementary sample
  # should be M < S < nwalkers
```

```
if (is.null(S)) S <- M+1
if (S <= M) S <- M+1
if (S >= nwalkers) S <- nwalkers - 1
S <- as.integer(S)

# an index of walkers
walkers <- 1:nwalkers

# draw the random z values and u values
# u are uniform values (0-1) and z are standard normal values
# for each walker we need S values of z.
z <- array(rnorm(nwalkers * S), dim=c(nwalkers, S))
u <- runif(nwalkers)

# loop over each walker, updating its position
for (j in 1:nwalkers) {

  # for walker j, randomly select a complementary sample of S walkers
  s <- sample(walkers[-j], S, replace=FALSE)

  # current position of the jth walker
  X.pres <- theta[j, 1:M]

  # current positions of all S of the complementary walkers
  # X.comp is a [S, M] matrix; each row is position of one walker
  X.comp <- theta[s, 1:M]

  # find mean position of the complementary walkers
  # X.mean is an M-vector
  X.mean <- apply(X.comp, 2, mean)

  # find deviation of each complementary walker from their mean
  # delta.k is a [S, M] matrix; each row is difference of a walker
  # from the mean
  delta.k <- X.comp - X.mean[col(X.comp)]

  # randomly weight each deviation
  # W.k is a matrix; each row is a randomly weighted difference
  # (Using 'recyling' we get each row (walker difference) multiplied by
  # one random deviate from z.)
  W.k <- z[j, ] * delta.k

  # now compute the random step W from the sum of all the deviations
  # (Sum over all the delta.k's to get an M-vector.)
  # This is eqn 11 of Goodman & Weare except I correct it by
  # a factor 1/sqrt(N) to ensure the covarince matrix is as required.
  W <- apply(W.k, 2, sum) / sqrt(S)

  # proposed new position for the jth walker
  X.prop <- X.pres + W

  if (chatter > 10) {
    cat('-- X.pres', X.pres, fill=TRUE)
```

```
      cat('-- X.mean', X.mean, fill=TRUE)
      cat('-- W', W, fill=TRUE)
      cat('-- X.prop', X.prop, fill=TRUE)
      cat('-- subsample', s, fill=TRUE)
#      cat('-- stune', S, fill=TRUE)
    }

    # evalute the log PDF at the existing and proposed positions.
    # The value at the current position was evaluated in the previous
    # step, so we reuse this value. Only the value at the proposed new
    # position needs evaluating.
    p.pres <- theta[j, M+2]
    p.prop <- posterior(X.prop, ...)

    # now evaluate the acceptance probability
    logP <- p.prop - p.pres
    lopP <- max(logP, 0)
    if ( exp(logP) >= u[j] ) {
      theta[j, 1:M] <- X.prop     # accept
      theta[j, M+1] <- 1
      theta[j, M+2] <- p.prop
    } else {
      theta[j, 1:M] <- X.pres     # reject
      theta[j, M+1] <- 0
      theta[j, M+2] <- p.pres
    }

  } # end of loop (j = 1, nwalkers)

  if (chatter > 5) {
    cat('-- Walk move accept rate:', mean(theta[,M+1]))
  }

  # return the updated array
  return(theta)

}
```

The R session information (including the OS info, R version and all packages used):

```
sessionInfo()

## R version 3.2.2 (2015-08-14)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 8 x64 (build 9200)
##
## locale:
## [1] LC_COLLATE=English_United Kingdom.1252  LC_CTYPE=English_United Kingdom.1252
## [3] LC_MONETARY=English_United Kingdom.1252 LC_NUMERIC=C
## [5] LC_TIME=English_United Kingdom.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
```

```
## other attached packages:
## [1] knitr_1.12.3
##
## loaded via a namespace (and not attached):
## [1] magrittr_1.5  formatR_1.2.1 tools_3.2.2   stringi_1.0-1 highr_0.5.1   stringr_1.0.0
## [7] evaluate_0.8
```

```
Sys.time()
```

```
## [1] "2016-07-13 08:43:19 BST"
```