Using NIMBLE to implement Markov Chain Monte Carlo with Integrated Nested Laplace approximation

Kwaku Peprah Adjei[1,2], Robert B. O'Hara[1,2]

[1]Department of Mathematical Sciences, Norwegian University of Science and Technology

[2]Center for Biodiversity Dynamics, Norwegian University of Science and Technology

1 Build INLA function

```
inlaStepVirtual <- nimbleFunctionVirtual(
  run = function(beta = double(1)
                 ) {
    returnType(double(0))
  }
)


# Bootstrap filter as specified in Doucet & Johnasen '08,
# uses weights from previous time point to calculate likelihood estimate.
inlaFStep <- nimbleFunction(
  name = 'inlaFStep',
  contains = inlaStepVirtual,
  setup = function(model,
                   mvEWSamples,
                   fixedVals,
                  # inlaModel,
                   x,
                   y,
                  # interInModel,
                   fam
                   ) {
```

```
    N <- length(fixedVals)

  },
  run = function(beta = double(1)
                 ) {
    returnType(double(0))

    #vals <- numeric(N, init=FALSE)
   # vals <-
    #ll <-
    #copy(mvEWSamples, model, nodes = fixedVals, row = 1)
    res <- nimbleINLA(x, y, beta= beta, fixedVals,  family = fam)

    lll <- res[1,1]

   if(lll == -Inf){
     copy(mvEWSamples, model, nodes = fixedVals, row = 1)
   }else{
     saveResults(fixedVals, res)
     copy( model, mvEWSamples, nodes = fixedVals, row = 1)
   }

      out <- lll

    return(out)
  },
methods = list(
saveResults = function(fixedVals = character(0),
                       res = double(2)){
  #n <- length(fixedVals)
  vals <- res[1, 2]
  values(model, fixedVals) <<- c(vals)

  return(vals)
  returnType(double(0))
```

```
}
)
)



# Bootstrap filter as specified in Doucet & Johnasen '08,
# uses weights from previous time point to calculate likelihood estimate.
inlaFStepMultiple <- nimbleFunction(
  name = 'inlaFStepMultiple',
  contains = inlaStepVirtual,
  setup = function(model,
                   mvEWSamples,
                   fixedVals,
                   # inlaModel,
                   x,
                   y,
                   # interInModel,
                   fam
  ) {
    #res <- inlaModel(x, y, beta, fixedVals, interInModel, family = fam)
    #copy(model, mvEWSamples, nodes = fixedVals, row = 1)
    N <- length(fixedVals)

    #     if(N >1){
    #       mult <- TRUE
    #       #vals <- rep(0, N)
    #     }else{
    #       mult <- FALSE
    #      # vals <- 0
    #     }
    #     print(mult)
    # print(N)

    # mult <- TRUE
    # mult1 <- FALSE
```

```
  # if(length(fixedVals) == 1){
  #   mult <- FALSE
  #   mult1 <- TRUE}
  # print(mult)
  #
  # if(N ==1){
  #   mult <- FALSE
  # }else{
  #   mult <- TRUE
  # }
  #
  # if(N == 1){
  #   vals <- 0
  # }else{
  #   vals <- rep(0, N)
  # }
},
run = function(beta = double(1) # beta is a vector

) {
  returnType(double(0))

  res <- nimbleINLA(x, y, beta= beta, fixedVals,  family = fam)



  lll <- res[1,1]

  if(lll == -Inf){
    copy(mvEWSamples, model, nodes = fixedVals, row = 1)
  }else{
    saveResults(fixedVals, res)
    copy( model, mvEWSamples, nodes = fixedVals, row = 1)
  }
```

```
      out <- lll

    return(out)
  },
  methods = list(
    saveResults = function(fixedVals = character(1),
                           res = double(2)){
      n <- length(fixedVals)
      vals <- numeric(n, init = FALSE)

      for(i in seq_along(fixedVals)){

        vals[i] <- res[1, i + 1]

      }

      values(model, fixedVals) <<- c(vals)

      return(vals)
      returnType(double(1))
    }
  )
)


buildINLAmodel <- nimbleFunction(
  name = 'buildINLAmodel',
  setup = function(model, fam, x, y, control) {
    inlaModel <- extractControlElement(control, 'fit.inla',  NULL)
    fixedVals <- extractControlElement(control, 'fixedVals',  double())

    yExpand <- model$expandNodeNames(y, returnScalarComponents = TRUE)
    y <- model[[y]]

    #save posterior samples
```

```
#modelVals = modelValues(model, m = 1)
vars <- model$getVarNames(nodes = fixedVals)
modelSymbolObjects <- model$getSymbolTable()$getSymbolObjects()[vars]

  names <- sapply(modelSymbolObjects, function(x)return(x$name))
  type <- sapply(modelSymbolObjects, function(x)return(x$type))
  size <- lapply(modelSymbolObjects, function(x)return(x$size))
  size <- lapply(size, function(x){
    if(length(x) == 0){

      return(1)
    }else(
      x
    )
  } )

  mvEWSamples <- modelValues(modelValuesConf(vars = names,
                                            types = type,
                                            sizes = size))

  fixedVals <- model$expandNodeNames(fixedVals)

  multiple <- TRUE
  if(length(model$expandNodeNames(fixedVals)) == 1) multiple = FALSE

  inlaStepFunctions <- nimbleFunctionList(inlaStepVirtual)

  if(multiple == TRUE){
    inlaStepFunctions[[1]] <- inlaFStepMultiple(model,
                                       mvEWSamples,
                                       fixedVals,
                                       # inlaModel,
                                       x,
                                       y,
                                      # interInModel,
```

```
                                                     fam)
     }else{
       inlaStepFunctions[[1]] <- inlaFStep(model,
                                                      mvEWSamples,
                                                      fixedVals,
                                                      # inlaModel,
                                                      x,
                                                      y,
                                                      # interInModel,
                                                      fam)
     }
   #}
   #essVals <- rep(0, length(nodes))

  lastLogLik <- -Inf
},
run = function(beta=double(1)#, # beta is a vector

  ) {
  returnType(double())
  #need this to retuen the saved mvEWSamples
  resize(mvEWSamples, 1)
  # for(i in seq_along(fixedVals)){
  #   mvEWSamples[[fixedVals[i]]][1] <<- vals[i]
  # }
  out <- inlaStepFunctions[[1]]$run(beta)#, interInModel)
  #rr <- inlaStepFunctions[[1]]$mvEWSamples
  logL <- out
  if(logL == -Inf) {lastLogLik <<- logL; return(logL)}
  if(is.nan(logL)) {lastLogLik <<- -Inf; return(-Inf)}
  if(logL == Inf)  {lastLogLik <<- -Inf; return(-Inf)}
  lastLogLik <<-  logL
  return(logL)
},
methods = list(
```

```
    getLastLogLik = function() {
      return(lastLogLik)
      returnType(double())
    },
    setLastLogLik = function(lll = double()) {
      lastLogLik <<- lll
    }
  )
)

# Bootstrap filter as specified in Doucet & Johnasen '08,
# uses weights from previous time point to calculate likelihood estimate.
inlaStepVirtualV2 <- nimbleFunctionVirtual(
  run = function(beta = double(1),
                 extraVars = double(1)
  ) {
    returnType(double(0))
  }
)


inlaFStepV2 <- nimbleFunction(
  name = 'inlaFStepV2',
  contains = inlaStepVirtualV2,
  setup = function(model,
                   mvEWSamples,
                   fixedVals,
                   x,
                   y,
                   fam
  ) {
    N <- length(fixedVals)
  },
  run = function(beta = double(1),
                 extraVars = double(1)
```

```
  ) {
    returnType(double(0))

    res <- nimbleINLA(x, y, beta= beta, extraVars = extraVars,fixedVals,  family = fam)

    lll <- res[1,1]

    if(lll == -Inf){
      copy(mvEWSamples, model, nodes = fixedVals, row = 1)
    }else{
      saveResults(fixedVals, res)
      copy( model, mvEWSamples, nodes = fixedVals, row = 1)
    }

    out <- lll
    return(out)
  },
  methods = list(
    saveResults = function(fixedVals = character(0),
                           res = double(2)){
      #n <- length(fixedVals)
      vals <- res[1, 2]
      values(model, fixedVals) <<- c(vals)

      return(vals)
      returnType(double(0))
    }
  )
)


inlaFStepMultipleV2 <- nimbleFunction(
  name = 'inlaFStepMultipleV2',
  contains = inlaStepVirtualV2,
```

```
setup = function(model,
                 mvEWSamples,
                 fixedVals,
                 # inlaModel,
                 x,
                 y,
                 # interInModel,
                 fam
) {
  N <- length(fixedVals)
},
run = function(beta = double(1), # beta is a vector
                extraVars = double(1)
) {
  returnType(double(0))

  res <- nimbleINLA(x, y, beta= beta, extraVars = extraVars,fixedVals,  family = fam)

  lll <- res[1,1]

  if(lll == -Inf){
    copy(mvEWSamples, model, nodes = fixedVals, row = 1)
  }else{
    saveResults(fixedVals, res)
    copy( model, mvEWSamples, nodes = fixedVals, row = 1)
  }
  out <- lll

  return(out)
},
methods = list(
  saveResults = function(fixedVals = character(1),
                         res = double(2)){
    n <- length(fixedVals)
    vals <- numeric(n, init = FALSE)
```

```
      #r <- character(0)
      #if(n > 1){
      for(i in seq_along(fixedVals)){
        # r <- fixedVals[i]
        vals[i] <- res[1, i + 1]
        #model[[r]] <<- vals[i]
      }

      values(model, fixedVals) <<- c(vals)

      return(vals)
      returnType(double(1))
    }
  )
)
```

2 Customised random-walk block sampler

```
sampler_RW_INLA_block <- nimbleFunction(
  name = 'sampler_RW_INLA_block',
  contains = sampler_BASE,
  setup = function(model, mvSaved, target,  control) {
    ## control list extraction
    adaptive             <- extractControlElement(control, 'adaptive',             FALSE)
    adaptScaleOnly       <- extractControlElement(control, 'adaptScaleOnly',       FALSE)
    adaptInterval        <- extractControlElement(control, 'adaptInterval',        200)
    adaptFactorExponent <- extractControlElement(control, 'adaptFactorExponent',  0.8)
    x                    <- extractControlElement(control, 'x',  double())
    y                    <- extractControlElement(control, 'y',  character())
    targetMCMC                <- extractControlElement(control, 'targetMCMC',               NULL)
    mu                   <- extractControlElement(control, 'mu',  NULL)
    #obsVars <- extractControlElement(control, 'obsVar',  character())
    fixedVals            <- extractControlElement(control, 'fixedVals',  double())
    fam                  <- extractControlElement(control, 'fam',  "gaussian")
    interVal             <- extractControlElement(control, 'interInModel',  1)
```

```
scale               <- extractControlElement(control, 'scale',                1)
propCov             <- extractControlElement(control, 'propCov',              'identity')
existingINLA        <- extractControlElement(control, 'fit.inla',                   NULL)
m                   <- extractControlElement(control, 'pfNparticles',         1000)
filterType          <- extractControlElement(control, 'pfType',               'bootstrap')
filterControl       <- extractControlElement(control, 'pfControl',            list())
optimizeM           <- extractControlElement(control, 'pfOptimizeNparticles', FALSE)

## node list generation
targetAsScalar      <- model$expandNodeNames(target, returnScalarComponents = TRUE)
calcNodes           <- model$getDependencies(target)
if(length(fixedVals) > 0){
latentSamp <- TRUE
}else{
  latentSamp <- FALSE
}
fixedValsDep <- model$getDependencies(fixedVals)
MCMCmonitors <- tryCatch(parent.frame(2)$conf$monitors, error = function(e) e)

topParams <- model$getNodeNames(stochOnly=TRUE, includeData=FALSE, topOnly=TRUE)
target <- model$expandNodeNames(target)
## numeric value generation
optimizeM     <- as.integer(optimizeM)
scaleOriginal <- scale
timesRan      <- 0
timesAccepted <- 0
timesAdapted  <- 0
prevLL        <- 0
nVarEsts      <- 0
itCount       <- 0
d <- length(targetAsScalar)
if(is.character(propCov) && propCov == 'identity')     propCov <- diag(d)
propCovOriginal <- propCov
chol_propCov <- chol(propCov)
chol_propCov_scale <- scale * chol_propCov
```

```
    empirSamp <- matrix(0, nrow=adaptInterval, ncol=d)

    # if(is.null(mu)){
    #   muTarget <- nimble::values(model,target)
    # }else{
    #     muTarget <- mu
    #   }
    storeParticleLP <- -Inf
    storeLLVar  <- 0
    nVarReps <- 7     ## number of LL estimates to compute to get each LL variance estimate for
    mBurnIn  <- 15    ## number of LL variance estimates to compute before deciding optimal m
    if(optimizeM)   m <- 3000
    ## nested function and function list definitions
    my_setAndCalculate <- setAndCalculate(model, target)
    my_decideAndJump <- decideAndJump(model, mvSaved, target, calcNodes)
    my_calcAdaptationFactor <- calcAdaptationFactor(d, adaptFactorExponent)

    #yVals <- values(model, obsData)
    my_particleFilter <- buildINLAmodel(model,
                                        fam,
                                        x,
                                        y = y,
                                        control = list(fit.inla = existingINLA,
                                                       fixedVals = fixedVals))


                                                       #Target values for inla
if(is.null(targetMCMC)){
  targetVal <- nimble::values(model, targetAsScalar)
}else{
targetMCMCasScalar <- model$expandNodeNames(targetMCMC, returnScalarComponents = TRUE)

targetVal <- nimble::values(model, c(targetMCMCasScalar, targetAsScalar))
   }

    particleMV <- my_particleFilter$mvEWSamples
```

```
    print(latentSamp)

    ## checks
    if(!inherits(propCov, 'matrix'))                    stop('propCov must be a matrix\n')
    if(!inherits(propCov[1,1], 'numeric'))              stop('propCov matrix must be numeric\n')
    if(!all(dim(propCov) == d))                         stop('propCov matrix must have dimensior
    if(!isSymmetric(propCov))                           stop('propCov matrix must be symmetric')
    if(length(targetAsScalar) < 2)                      stop('less than two top-level targets; c
   # if(any(target%in%model$expandNodeNames(latents)))   stop('PMCMC \'target\' argument cannot
  },
  run = function() {
    storeParticleLP <<- my_particleFilter$getLastLogLik()
    modelLP0 <- storeParticleLP + getLogProb(model, target)
    propValueVector <- generateProposalVector()
    my_setAndCalculate$run(propValueVector)
    targetVal <<- values(model, targetAsScalar)
    particleLP <- my_particleFilter$run(beta = targetVal)#,interInModel = interVal)
    modelLP1 <- particleLP + getLogProb(model, target)
    jump <- my_decideAndJump$run(modelLP1, modelLP0, 0, 0)
    if(!jump) {
      my_particleFilter$setLastLogLik(storeParticleLP)
    }
     if(jump & latentSamp) {
       ## if we jump, randomly sample latent nodes from pf output and put
       ## into model so that they can be monitored
    #   index <- ceiling(runif(1, 0, m))
       copy(particleMV, model, fixedVals, fixedVals, row = 1)
       calculate(model, fixedValsDep)
       copy(from = model, to = mvSaved, nodes = fixedValsDep, row = 1, logProb = TRUE)
    }
     else if(!jump & latentSamp) {
      ## if we don't jump, replace model latent nodes with saved latent nodes
       copy(from = mvSaved, to = model, nodes = fixedValsDep, row = 1, logProb = TRUE)
     }
    ##if(jump & !resample)  storeParticleLP <<- particleLP
```

```
    if(jump & optimizeM) optimM()
    if(adaptive)     adaptiveProcedure(jump)
  },
  methods = list(
    optimM = function() {
      tempM <- 15000
      declare(LLEst, double(1, nVarReps))
      if(nVarEsts < mBurnIn) {  # checks whether we have enough var estimates to get good approx
        for(i in 1:nVarReps)
          LLEst[i] <- my_particleFilter$run(beta = targetVal)#, interInModel = interVal)
        ## next, store average of var estimates
        if(nVarEsts == 1)
          storeLLVar <<- var(LLEst)/mBurnIn
        else {
          LLVar <- storeLLVar
          LLVar <- LLVar + var(LLEst)/mBurnIn
          storeLLVar <<- LLVar
        }
        nVarEsts <<- nVarEsts + 1
      }
      else {  # once enough var estimates have been taken, use their average to compute m
        m <<- m*storeLLVar/(0.92^2)
        m <<- ceiling(m)
        storeParticleLP <<- my_particleFilter$run(targetVal)
        optimizeM <<- 0
      }
    },
    generateProposalVector = function() {
      propValueVector <- rmnorm_chol(1, values(model, targetAsScalar), chol_propCov_scale, 0)  #
      returnType(double(1))
      return(propValueVector)
    },
    adaptiveProcedure = function(jump = logical()) {
      timesRan <<- timesRan + 1
      if(jump)     timesAccepted <<- timesAccepted + 1
```

```
        if(!adaptScaleOnly)     empirSamp[timesRan, 1:d] <<- values(model, target)
        if(timesRan %% adaptInterval == 0) {
          acceptanceRate <- timesAccepted / timesRan
          timesAdapted <<- timesAdapted + 1
          adaptFactor <- my_calcAdaptationFactor$run(acceptanceRate)
          scale <<- scale * adaptFactor
          ## calculate empirical covariance, and adapt proposal covariance
          if(!adaptScaleOnly) {
            gamma1 <- my_calcAdaptationFactor$getGamma1()
            for(i in 1:d)     empirSamp[, i] <<- empirSamp[, i] - mean(empirSamp[, i])
            empirCov <- (t(empirSamp) %*% empirSamp) / (timesRan-1)
            propCov <<- propCov + gamma1 * (empirCov - propCov)
            chol_propCov <<- chol(propCov)
          }
          chol_propCov_scale <<- chol_propCov * scale
          timesRan <<- 0
          timesAccepted <<- 0
        }
      },
      reset = function() {
        scale   <<- scaleOriginal
        propCov <<- propCovOriginal
        chol_propCov <<- chol(propCov)
        chol_propCov_scale <<- chol_propCov * scale
        storeParticleLP <<- -Inf
        timesRan      <<- 0
        timesAccepted <<- 0
        timesAdapted  <<- 0
        my_calcAdaptationFactor$reset()
      }
  )
)
```

3 Fitting models with Alternative One

```
INLAWiNimDataGenerating <- function(data,
                                    covariate,
                                    code,
                                    family,
                                    modelData,
                                    modelConstants,
                                    modelInits,
                                    nimbleINLA,
                                    inlaMCMC = c("inla", "mcmc", "inlamcmc"),
                                    inlaMCsampler = "RW_INLA_block",
                                    samplerControl = list(),
                                    parametersToMonitor = list(mcmc = c("mcmc"),
                                                               inla = c("inla"),
                                                               additionalPars = NULL),
                                    mcmcConfiguration = list(n.chains = 1,
                                                             n.iterations = 10,
                                                             n.burnin = 0,
                                                             n.thin = 1,
                                                             setSeed = TRUE,
                                                             samples=TRUE,
                                                             samplesAsCodaMCMC = TRUE,
                                                             summary = TRUE,
                                                             WAIC = FALSE)){



  #extract necessary variables
  x <- covariate # must be a matrix
  y <- data # must be a vector
  family <- family
  fixedVals <- parametersToMonitor$inla
  target <- parametersToMonitor$mcmc

  #set up initial value
```

```r
initsList <- modelInits()

#Create the model in nimble
mwtc <- nimble::nimbleModel(code,
                            data = modelData,
                            constants = modelConstants,
                            inits = initsList)

# Create the model in C
Cmwtc <- nimble::compileNimble(mwtc,
                               showCompilerOutput = FALSE) #Have issues compiling

#create list to return
retList <- list()
# Fit INLAMCMC

if(inlaMCMC %in% "inlamcmc"){
  mcmcconf <- nimble::configureMCMC(Cmwtc,
                                    nodes = NULL)

  # setting sampler controls
  samplerControl$fit.inla = nimbleINLA
  samplerControl$x = x
  samplerControl$y = y
  samplerControl$fixedVals = fixedVals
  samplerControl$fam = family

  # mcmc configuration
  mcmcconf$addSampler(target = target,
                      type = inlaMCsampler,
                      control = samplerControl)

  mcmcconf$printSamplers(executionOrder = TRUE)
  mcmcconf$addMonitors(target)
```

```r
    if(!is.null(parametersToMonitor$additionalPars)){
      mcmcconf$addMonitors(parametersToMonitor$additionalPars)
    }

    #build model
    Rmcmc <- nimble::buildMCMC(mcmcconf)

    # Compile
    cmcmc <- nimble::compileNimble(Rmcmc,
                                   project = Cmwtc,
                                   resetFunctions = TRUE)
    startTime <- Sys.time()
    mcmc.out <- nimble::runMCMC(cmcmc,
                                niter = mcmcConfiguration[["n.iterations"]],
                                nchains = mcmcConfiguration[["n.chains"]],
                                nburnin = mcmcConfiguration[["n.burnin"]],
                                #inits = initsList,
                                thin = mcmcConfiguration[["n.thin"]],
                                setSeed = mcmcConfiguration[["setSeed"]],
                                samples = mcmcConfiguration[["samples"]],
                                samplesAsCodaMCMC = mcmcConfiguration[["samplesAsCodaMCMC"]],
                                summary = mcmcConfiguration[["summary"]],
                                WAIC = mcmcConfiguration[["WAIC"]])
    endTime <- Sys.time()
    timeTaken <- difftime(endTime, startTime, units = "secs")
      #as.numeric(endTime - startTime)

    ret <- list(mcmc.out = mcmc.out,
                timeTaken = timeTaken)

    #save inlamcmc results
    retList$inlamcmc <- ret
  }

  if(inlaMCMC %in% "mcmc"){
```

```
mcmcconf <- nimble::configureMCMC(Cmwtc,
                                  monitors = c(target, fixedVals))

mcmcconf$printSamplers()

# Add new samplers
mcmcconf$removeSampler(target)
# mcmcconf$removeSampler("beta")
#mcmcconf$addSampler("beta", type = inlaMCsampler)
mcmcconf$addSampler(target, type = inlaMCsampler,
                    control = samplerControl)

mcmcconf$printSamplers(executionOrder = TRUE)

if(!is.null(parametersToMonitor$additionalPars)){
  mcmcconf$addMonitors(parametersToMonitor$additionalPars)
}
#build model
Rmcmc <- nimble::buildMCMC(mcmcconf, useConjugacy = FALSE)

# Compile
cmcmc <- nimble::compileNimble(Rmcmc,
                               project = Cmwtc,
                               resetFunctions = TRUE)
startTime <- Sys.time()
mcmc.out <- nimble::runMCMC(cmcmc,
                            niter = mcmcConfiguration[["n.iterations"]],
                            nchains = mcmcConfiguration[["n.chains"]],
                            nburnin = mcmcConfiguration[["n.burnin"]],
                            #inits = initsList,
                            thin = mcmcConfiguration[["n.thin"]],
                            setSeed = mcmcConfiguration[["setSeed"]],
                            samples = mcmcConfiguration[["samples"]],
                            samplesAsCodaMCMC = mcmcConfiguration[["samplesAsCodaMCMC"]],
                            summary = mcmcConfiguration[["summary"]],
```

```r
                                WAIC = mcmcConfiguration[["WAIC"]])
    endTime <- Sys.time()
    timeTaken <- difftime(endTime, startTime, units = "secs")

    ret <- list(mcmc.out = mcmc.out,
                timeTaken = timeTaken)

    #save results for MCMC
    retList$mcmc <-  ret
  }
  return(retList)
}
```

4 Fitting models with Alternative Two

```r
INLAWiNim <- function(data,
                      code,
                      fam,
                      modelData,
                      modelConstants,
                      modelInits,
                      parametersToMonitor = c("beta", "sigma", "intercept"),
                      mcmcControl = NULL,
                      mcmcSamplerChange = FALSE,
                      parametersForSamplerChange = NULL,
                      newSampler = NULL,
                      newSamplerControl = NULL,
                      mcmcConfiguration = list(n.chains = 1,
                                               n.iterations = 10,
                                               n.burnin = 0,
                                               n.thin = 1,
                                               setSeed = TRUE,
                                               samples=TRUE,
                                               samplesAsCodaMCMC = TRUE,
                                               summary = TRUE,
```

```
                                                WAIC = TRUE)){




initsList <- modelInits()

#initsList <- idm_inits()



#Create the model in nimble

mwtc <- nimble::nimbleModel(code,

                            data = modelData,

                            constants = modelConstants,

                            inits = initsList)


# Create the model in C

Cmwtc <- nimble::compileNimble(mwtc,

                               showCompilerOutput = FALSE) #Have issues compiling


if(!is.null(mcmcControl)){

  mcmcconf <- nimble::configureMCMC(Cmwtc,

                                    monitors = parametersToMonitor,

                                    control = mcmcControl,

                                    enableWAIC = FALSE)

}else{

  mcmcconf <- nimble::configureMCMC(Cmwtc,

                                    monitors = parametersToMonitor,

                                    enableWAIC = FALSE

  )

}


if(mcmcSamplerChange == TRUE){

  mcmcconf$removeSamplers(parametersForSamplerChange)

  mcmcconf$addSampler(target = parametersForSamplerChange,

                      type = newSampler,
```

```
                          control = newSamplerControl)
  mcmcconf$printSamplers()
}

Rmcmc <- nimble::buildMCMC(mcmcconf)

# Compile
cmcmc <- nimble::compileNimble(Rmcmc,
                               project = Cmwtc,
                               resetFunctions = TRUE)

#MCMC Configurations


# Run the MCMC
startTime <- Sys.time()
mcmc.out <- nimble::runMCMC(cmcmc,
                            niter = mcmcConfiguration[["n.iterations"]],
                            nchains = mcmcConfiguration[["n.chains"]],
                            nburnin = mcmcConfiguration[["n.burnin"]],
                            #inits = initsList,
                            thin = mcmcConfiguration[["n.thin"]],
                            setSeed = mcmcConfiguration[["setSeed"]],
                            samples = mcmcConfiguration[["samples"]],
                            samplesAsCodaMCMC = mcmcConfiguration[["samplesAsCodaMCMC"]],
                            summary = mcmcConfiguration[["summary"]],
                            WAIC = mcmcConfiguration[["WAIC"]])

endTime <- Sys.time()
timeTaken <- difftime(endTime, startTime, units = "secs")
#Output from the MCMC
output <- mcmc.out$summary
output
```

```r
  if(fam == "gaussian"){

    scales <- NA
    accept <- NA
    prop_history <- NA

  }else{
    scales <- NA
    accept <- NA
    prop_history <- NA
  }


  returnList = list(output=output,
                    scales=scales,
                    accept=accept,
                    prop_history=prop_history,
                    mcmc.out=mcmc.out,
                    timeTaken = timeTaken)


  return(returnList)
}
```

References