

# EZ Bayesian Hierarchical Drift Diffusion Model

Based on Joachim's python code

Adriana F. Chavez

Last time knitted: 18 September, 2023

---

## Basic functions to generate DDM data

```
# Part 1: Simulate single trial outcome
simulate_ddm <- function(a, v, dt, max_steps){
  x <- 0
  random_dev <- rnorm(max_steps)
  # Scale step changes by dt
  noise <- random_dev * sqrt(dt)
  drift <- v * dt

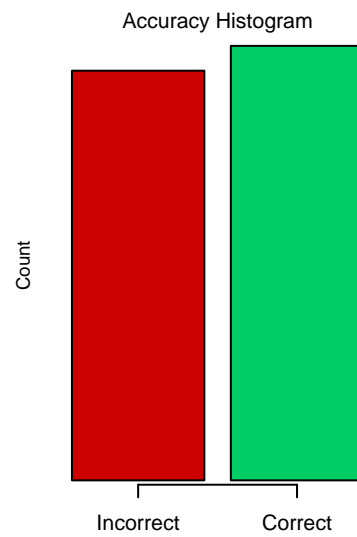
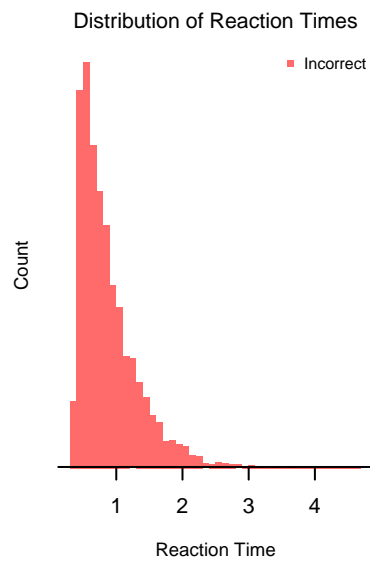
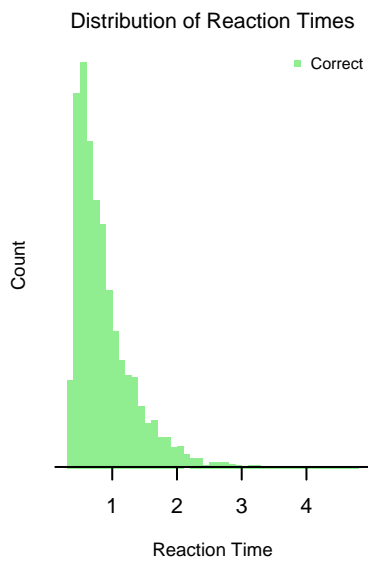
  for(i in 2:max_steps){
    this_step = drift + noise[i]
    x = x + this_step
    if(abs(x)>=(a/2)){ break }
  }
  output <- list("RT" = (i+1)*dt, "C" = x)
  return(output)
}

# Part 2: Simulate over 'n' trials
wdmrnd <- function(a,v,t,n){
  dt = 0.001
  max_steps = 10 / dt
  rt = rep(NA,n)
  accuracy = rep(NA,n)

  for(i in 1:n){
    X <- simulate_ddm(a, v, dt, max_steps)
    rt[i] <- X$RT
    if(X$C>0){ accuracy[i] <- 1
    }else{ accuracy[i] <- 0 }
  }
  output <- data.frame("RT" = rt + t, "accuracy" = accuracy)
  return(output)
}
```

## Example: Generate some data

```
a = 1.50  
v = 0.00  
t = 0.30  
n = 10000  
  
data <- wdmrnd(a, v, t, n)  
rt <- data$RT  
accuracy <- data$accuracy
```



# Simulation Study environment and variables

## Auxiliary functions

The code for the auxiliary functions is hidden from this .pdf file (but can be checked on the .Rmd file). The auxiliary functions are:

1. `design_summary`: A function to print the settings used in the simulation
2. `default_priors`: A function to load and print default prior values
3. `write_JAGSmodel`: A function to write the JAGS model using the prior values
4. `data_toJAGS`: A function to create a list with all the data objects in the JAGS model

## Core functions

```
#####  
###      Core functions      #####  
#####  
  
# Sample 'true' parameter values from the priors specified  
sample_parameters <- function(settings){  
  prior <- settings$prior  
  nP <- settings$nPart  
  bound_mean <- rnorm(1,prior$bound_mean_mean,prior$bound_mean_sdev)  
  drift_mean <- rnorm(1,prior$drift_mean_mean,prior$drift_mean_sdev)  
  nondt_mean <- rnorm(1,prior$nondt_mean_mean,prior$nondt_mean_sdev)  
  bound_sdev <- runif(1,prior$bound_sdev_lower,prior$bound_sdev_upper)  
  drift_sdev <- runif(1,prior$drift_sdev_lower,prior$drift_sdev_upper)  
  nondt_sdev <- runif(1,prior$nondt_sdev_lower,prior$nondt_sdev_upper)  
  bound <- rnorm(nP,bound_mean, bound_sdev)  
  drift <- rnorm(nP,drift_mean, drift_sdev)  
  nondt <- rnorm(nP,nondt_mean, nondt_sdev)  
  parameter_set <- list("bound_mean" = bound_mean, "drift_mean" = drift_mean,  
                        "nondt_mean" = nondt_mean, "bound_sdev" = bound_sdev,  
                        "drift_sdev" = drift_sdev, "nondt_sdev" = nondt_sdev,  
                        "bound" = bound, "drift" = drift, "nondt" = nondt)  
  return(parameter_set)  
}  
  
# Sample data using simulation settings and parameter values sampled  
sample_data <- function(settings, parameter_set){  
  nObs = settings$nPart*settings$nTrials  
  data = matrix(NA,ncol=3,nrow=nObs)  
  data[,1] = rep(1:settings$nPart, each=settings$nTrials)  
  for(i in 1:settings$nP){  
    this.sub <- which(data[,1]==i)  
    accuracy = 0  
    while(sum(accuracy)==0){  
      temp <- wdmrnd(a = parameter_set$bound[i],  
                    v = parameter_set$drift[i],  
                    t = parameter_set$nondt[i],  
                    n = settings$nTrials)  
      accuracy = temp$accuracy  
    }  
    data[this.sub,3] = accuracy  
  }
```

```

    data[this.sub,2] = temp$RT
  }
  data = as.data.frame(data)
  colnames(data) <- c("sub", "rt", "accuracy")
  return(data)
}

# Get individual statistics from full data: mean acc and correct-rt mean and var
get_Statistics <- function(data){
  if(is.null(data$accuracy)|is.null(data$rt)){
    error.msg = "Data not available."
    return(print(error.msg))
  }
  subID = unique(data$sub)
  sum_correct = tapply(data$accuracy, data$sub, sum)
  always_0 = which(sum_correct==0)
  if(length(always_0)!=0){
    bad_participants = (data$sub %in% always_0)
    data = data[-bad_participants,]
    sum_correct = tapply(data$accuracy, data$sub, sum)
  }
  mean_accuracy = tapply(data$accuracy, data$sub, mean)

  keep.correct = which(data$accuracy==1)
  correct_only = data[keep.correct,]
  mean_rt_correct = tapply(correct_only$rt, correct_only$sub, mean)
  var_rt_correct = tapply(correct_only$rt, correct_only$sub, var)

  data_statistics = cbind(subID, sum_correct, mean_accuracy, mean_rt_correct, var_rt_correct)
  data_statistics = as.data.frame(data_statistics)
  colnames(data_statistics) = c("sub", "sum_correct", "meanAccuracy", "meanRT_correct", "varRT_correct")
  return(data_statistics)
}

```

## Main functions

```

Hddm_Data <- function(settings, parameter_set){
  rawData = sample_data(settings,parameter_set)
  sumData = get_Statistics(rawData)
  jagsData = data_toJAGS()
  return(list("rawData" = rawData,
             "sumData" = sumData,
             "jagsData" = jagsData))
}

Hddm_Design = function(nParticipants, nTrials, prior, n.chains = 4,
                       save_sampleFiles = FALSE){

  settings <- list("nPart" = nParticipants,
                  "nTrials" = nTrials,
                  "prior" = prior)
  parameter_set = sample_parameters(settings)

  getData <- Hddm_Data(settings,parameter_set)
}

```

```

write_JAGSmodel(prior)

sub      <- getData$sumData$sub
correct  <- getData$sumData$sum_correct
varRT    <- getData$sumData$varRT_correct
meanRT   <- getData$sumData$meanRT_correct
nTrialsPerPerson <- as.numeric(unique(tapply(getData$rawData$accuracy,getData$rawData$sub,length)))
nParticipants <- length(getData$sumData$sub)

data <- getData$jagsData
init <- default_inits(n.chains, settings$nPart)

parameters <- c('bound_mean', 'drift_mean', 'nondt_mean', 'bound_sdev',
                'drift_sdev', 'nondt_sdev', 'bound', 'drift', 'nondt')
samples <- jags(data=data, parameters.to.save=parameters, model=modelFile,
               n.chains=n.chains, n.iter=n.iter, n.burnin=n.burnin,
               n.thin=n.thin, DIC=T, inits=myinits)
save(samples,file=fileName)
load(fileName)
}

```

```

# Step 5. Show chains obtained per parameter
#####
posterior.samples <- samples$BUGSoutput$sims.array

labels <- names(posterior.samples[1,1,])
for(i in 1:dim(posterior.samples)[3]){
  plot(posterior.samples[,1,i], type="l", main=labels[i], xlab="Iteration",
       ylab="Value sampled")
  if(n.chains>1){
    for(a in 2:n.chains){
      lines(posterior.samples[,a,i],col=a)
    }
  }
}
}

```

```

class Hddm_Parameter_Set:
  def __init__(self,
               bound_mean = None, bound_sdev = None, bound = None,
               drift_mean = None, drift_sdev = None, drift = None,
               nondt_mean = None, nondt_sdev = None, nondt = None):
    self.bound_mean = bound_mean
    self.bound_sdev = bound_sdev
    self.bound      = bound
    self.drift_mean = drift_mean
    self.drift_sdev = drift_sdev
    self.drift      = drift
    self.nondt_mean = nondt_mean
    self.nondt_sdev = nondt_sdev
    self.nondt      = nondt

  def __sub__(self, other):
    if not isinstance(other, Hddm_Parameter_Set):
      return None

    return Hddm_Parameter_Set(

```

```

        bound_mean = self.bound_mean - other.bound_mean,
        bound_sdev = self.bound_sdev - other.bound_sdev,
        drift_mean = self.drift_mean - other.drift_mean,
        drift_sdev = self.drift_sdev - other.drift_sdev,
        nondt_mean = self.nondt_mean - other.nondt_mean,
        nondt_sdev = self.nondt_sdev - other.nondt_sdev,
        bound      = self.bound - other.bound if self.bound is not None and other.bound is not None else None,
        drift      = self.drift - other.drift if self.drift is not None and other.drift is not None else None,
        nondt      = self.nondt - other.nondt if self.nondt is not None and other.nondt is not None else None
    )

def __str__(self):
    output = [
        "Hddm_Parameter_Set Details:",
        f"Bound Mean:           {self.bound_mean}",
        f"Bound Std Dev:        {self.bound_sdev}",
        f"Drift Mean:            {self.drift_mean}",
        f"Drift Std Dev:         {self.drift_sdev}",
        f"Non-decision Time Mean: {self.nondt_mean}",
        f"Non-decision Time Std: {self.nondt_sdev}",
        f"Bound:                 {self.bound}",
        f"Drift:                  {self.drift}",
        f"Non-decision Time:      {self.nondt}"
    ]
    return '\n'.join(output)

```

```

Hddm_Design = function(nParticipants, nTrials, prior){

```

```

def estimate_parameters(self):
    # This is the key bit
    code = f"""
    JAGS
    """

    data, valid_indices = self.data.to_jags()

    n_Participants_Left = data['nParticipants']

    # Initial values
    init = { "drift" : np.random.normal(0, 0.1, n_Participants_Left) }

    try:
        model = pyjags.Model(
            progress_bar = False,
            code         = code,
            data         = data,
            init         = init,
            adapt        = 100,
            chains       = 4,
            threads      = 4)
    except Exception as e:
        #error_message = str(e)
        #print(type(error_message))

```

```

        #print(error_message)
        #self.data.summary()
        #print(self.data.to_jags())
        #print(self.parameter_set)
        print('e', end='')
        return

samples = model.sample(400,
                        vars = ['bound_mean', 'drift_mean', 'nondt_mean',
                                'bound_sdev', 'drift_sdev', 'nondt_sdev',
                                'bound',      'drift',      'nondt'])

# Annoying management of sample object... First move individual parameters to their own fields
for i in np.arange(0, n_Participants_Left):
    samples.update({'bound_'+str(valid_indices[i]): samples['bound'][i,:,:],
                    'drift_'+str(valid_indices[i]): samples['drift'][i,:,:],
                    'nondt_'+str(valid_indices[i]): samples['nondt'][i,:,:], })

# ... remove the old unwieldy matrices
for s in ["bound", "drift", "nondt"]:
    samples.pop(s)

# Start a new dict with estimates only
estimate = { "bound": [np.nan] * self.n_Participants,
              "drift": [np.nan] * self.n_Participants,
              "nondt": [np.nan] * self.n_Participants
            }

for varname in ['bound_mean', 'drift_mean', 'nondt_mean',
                'bound_sdev', 'drift_sdev', 'nondt_sdev']:
    estimate.update({varname: np.mean(samples[varname])})

# ... make new, wieldy matrices
for i in valid_indices:
    estimate['bound'][i] = np.mean(samples['bound_'+str(i)])
    estimate['drift'][i] = np.mean(samples['drift_'+str(i)])
    estimate['nondt'][i] = np.mean(samples['nondt_'+str(i)])

# Copy estimate to design object
self.estimate = Hddm_Parameter_Set()
self.estimate.bound_mean = estimate['bound_mean']
self.estimate.drift_mean = estimate['drift_mean']
self.estimate.nondt_mean = estimate['nondt_mean']
self.estimate.bound_sdev = estimate['bound_sdev']
self.estimate.drift_sdev = estimate['drift_sdev']
self.estimate.nondt_sdev = estimate['nondt_sdev']
self.estimate.bound      = estimate['bound']
self.estimate.drift      = estimate['drift']
self.estimate.nondt      = estimate['nondt']

```

# Run simulations

## Simple example

```
set.seed(123)

prior <- default_priors()

write_JAGSmodel(prior)

design <- Hddm_Design(nParticipants = 20, nTrials = 50, prior = prior)

design.sample_parameters()
design.sample_data()
design.estimate_parameters()
```

## Simulation study (200 repetitions)

```
nSim <- 200
prior <- default_priors()
settings <- list("nPart"   = 50,
                 "nTrials" = 150,
                 "prior"   = prior)

tru = [Hddm_Parameter_Set()] * K
est = [Hddm_Parameter_Set()] * K
err = [Hddm_Parameter_Set()] * K

for(k in 1:nSim){
  set.seed(k)
  cat("Iteration", k+1, "of", nSim)
  design = Hddm_Design(participants=20, trials=50, prior=prior)
  design.sample_parameters()
  design.sample_data()
  #print(design.parameter_set)
  #design.data.summary()
  design.estimate_parameters()
  tru[k] = design.parameter_set
  est[k] = design.estimate
  if design.estimate is not None:
    err[k] = (design.estimate - design.parameter_set)
  else:
    err[k] = None
  if (k+1) % 100 == 0:
    print(f'. {k+1} of {K}\n', end='')
  else:
    print('.', end='')
}

def recovery_plot(x, y, parameterName, ttl):
  fontsize = 10
```



```

plt.figure(figsize=(2, 2))

plt.scatter(x, y, color='b', s=3)
plt.grid()
plt.gca().set_aspect('equal')

xax = np.linspace(min(x), max(x), 100)

plt.plot(xax, xax, '--')

plt.xlabel('Simulated value', fontsize=10)
plt.title('Group mean ' + parameterName, fontsize=10)

output_path = "ezrecovery_" + parameterName + ".pdf"
plt.savefig(output_path, format='pdf', bbox_inches='tight')

plt.show()

```

```

x = [np.nan] * K
y = [np.nan] * K
for k in range(K):
    if err[k] is not None:
        x[k] = tru[k].nondt_mean
        y[k] = est[k].nondt_mean

recovery_plot(x, y, 'nondt', 'Group mean nondt')

x = [np.nan] * K
y = [np.nan] * K
for k in range(K):
    if err[k] is not None:
        x[k] = tru[k].drift_mean
        y[k] = est[k].drift_mean

recovery_plot(x, y, 'drift', 'Group mean drift')

x = [np.nan] * K
y = [np.nan] * K
for k in range(K):
    if err[k] is not None:
        x[k] = tru[k].bound_mean
        y[k] = est[k].bound_mean

recovery_plot(x, y, 'bound', 'Group mean bound')

x = np.empty(0)
y = np.empty(0)
for k in range(K):
    if err[k] is not None:
        x = np.append(x, tru[k].drift)
        y = np.append(y, est[k].drift)

recovery_plot(x, y, 'drift', 'Individual drift rates')

```