EZ Bayesian Hierarchical Drift Diffusion Model

Based on Joachim's python code

Adriana F. Chavez

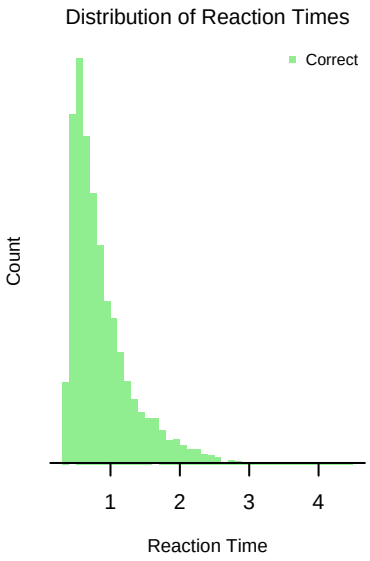
Last time knitted: 18 September, 2023

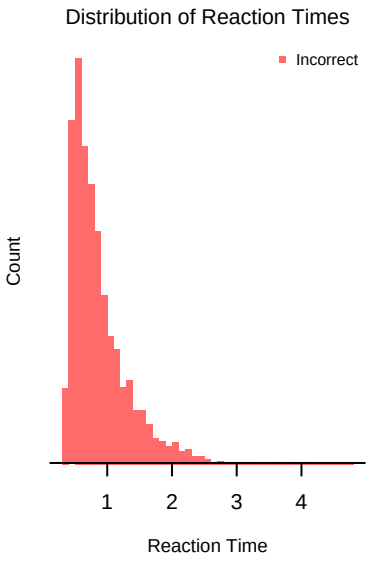
Basic functions to generate DDM data

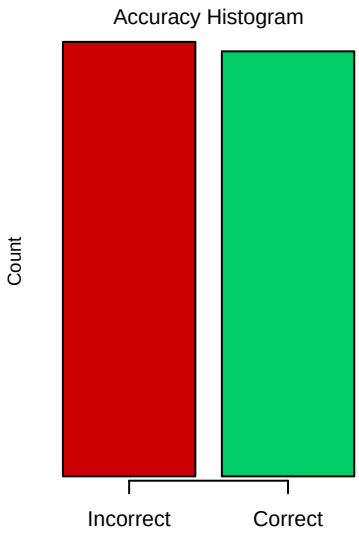
```
# Part 1: Simulate single trial outcome
simulate_ddm <- function(a, v, dt, max_steps){
     x <- 0
     random_dev <- rnorm(max_steps)
     # Scale step changes by dt
     noise <- random_dev * sqrt(dt)
     drift <- v * dt

     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

Core functions

```
#####################################################################
####    Basic functions       ###################################
#####################################################################

# Sample 'true' parameter values from the priors specified
sample_parameters <- function(settings){
    prior <- settings$prior
    nP <- settings$nP
    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$nP){
      this.sub <- which(data[,1]==i)
      temp <- wdmrnd(a = parameter_set$bound[i],
                     v = parameter_set$drift[i],
                     t = parameter_set$nondt[i],
                     n = settings$nT[i])
      data[this.sub,2] = temp$C
      data[this.sub,3] = temp$RT
  }
  colnames(data) <- c("sub", "choice", "rt")
  return(data)
}
```

Main functions

```
Hddm_Data <- function(settings, parameter_set){

}
```

```
    def __init__(self, person = None, rt = None, accuracy = None, n_TrialsPerPerson = None):
        self.person            = person
        self.rt                = rt
        self.accuracy          = accuracy
        self.n_TrialsPerPerson = n_TrialsPerPerson

    @staticmethod
    def sample(design):
        T = design.n_TrialsPerPerson
        P = design.n_Participants
        parameters = design.parameter_set

        person_list   = []
        rt_list       = []
        accuracy_list = []

        for p in range(P):
            accuracy = 0
            while np.sum(accuracy) == 0:
                rt, accuracy = wdmrnd(parameters.bound[p], parameters.drift[p], parameters.nondt[p], T)
            person_list.extend([p] * T)  # Repeat the participant ID for T trials
            rt_list.extend(rt)
            accuracy_list.extend(accuracy)

        # Convert lists to NumPy arrays for consistency and potential performance benefits
        person   = np.array(person_list)
        rt       = np.array(rt_list)
        accuracy = np.array(accuracy_list)

        return Hddm_Data(person, rt, accuracy, T)

    def summary(self):
        if self.person is None or self.rt is None or self.accuracy is None:
            print("Data not available.")
            return

        unique_persons = np.unique(np.array(self.person))
        print("{:<10} {:<20} {:<20} {:<20}".format("Person", "Mean Accuracy", "Mean RT (Correct)", "Variance RT

        for person_id in unique_persons:
            # Filter data for current person
            person_indices  = np.where(self.person == person_id)
            person_rts      = np.array(self.rt)[person_indices]
            person_accuracy = np.array(self.accuracy)[person_indices]

            # Compute the metrics
            mean_accuracy       = np.mean(person_accuracy)
            correct_rts         = person_rts[person_accuracy == 1]  # only accurate responses
            mean_rt_correct     = np.mean(correct_rts) if len(correct_rts) > 0 else np.nan
            variance_rt_correct = np.var(correct_rts) if len(correct_rts) > 0 else np.nan
```

```
        print("{:<10} {:<20.3f} {:<20.3f} {:<20.3f}".format(person_id, mean_accuracy, mean_rt_correct, varia

def to_jags(self):
    if self.person is None or self.rt is None or self.accuracy is None:
        return None

    unique_persons = np.unique(np.array(self.person)).astype(int)
    nParticipants  = len(unique_persons)

    # Initialize arrays to NaN for storing metrics
    sum_accuracy        = np.zeros(nParticipants, dtype=int)
    mean_rt_correct     = np.full(nParticipants, np.nan)
    variance_rt_correct = np.full(nParticipants, np.nan)

    # Loop over unique persons and compute metrics
    for person_id in unique_persons:
        # Filter data for the current person
        person_indices  = self.person == person_id
        person_rts      = self.rt[person_indices]
        person_accuracy = self.accuracy[person_indices]

        # Update metrics
        sum_accuracy[person_id] = np.sum(person_accuracy)
        correct_rts = person_rts[person_accuracy == 1]  # only accurate responses

        if correct_rts.size > 1:
            mean_rt_correct[person_id]     = np.mean(correct_rts)
            variance_rt_correct[person_id] = np.var(correct_rts)

    # Filter out participants with NaN values in any metric
    valid_indices = ~(
        np.isnan(mean_rt_correct) |
        np.isnan(variance_rt_correct)
    )

    # Extract valid metrics
    sum_accuracy        = sum_accuracy[valid_indices].tolist()
    mean_rt_correct     = mean_rt_correct[valid_indices].tolist()
    variance_rt_correct = variance_rt_correct[valid_indices].tolist()
    nParticipants       = len(sum_accuracy)  # Update nParticipants after filtering

    return {
        "nTrialsPerPerson": int(self.n_TrialsPerPerson),
        "nParticipants": nParticipants,
        "meanRT": mean_rt_correct,
        "varRT": variance_rt_correct,
        "correct": sum_accuracy,
    }, unique_persons[valid_indices]

def __str__(self):
    output = [
        "Hddm_Data Details:",
        f"Person:    {self.person}",
        f"RT:        {self.rt}",
        f"Accuracy:  {self.accuracy}"
```

```
        ]
        return '\n'.join(output)
```

```
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 Nor
            drift      = self.drift - other.drift if self.drift is not None and other.drift is not None else Nor
            nondt      = self.nondt - other.nondt if self.nondt is not None and other.nondt is not None else Nor
        )

    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){

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

  parameter_set = sample_parameters(settings)
```

```
  getData = sample_data(settings, parameter_set)

}
```

```
    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')
```