

[illegible]

**NC STATE**  
UNIVERSITY



# OBJECTIVES

- By the end of this session, you should be able to:
  - Understand the basic procedure and key considerations for designing model simulations
  - Understand the tradeoffs among model complexity, simulation complication, and computational demands
  - Understand basic design principles for constructing and running simulations in R
  - Construct code for replicating simulations in Dionne et al.'s (2010) computational model of mental model convergence

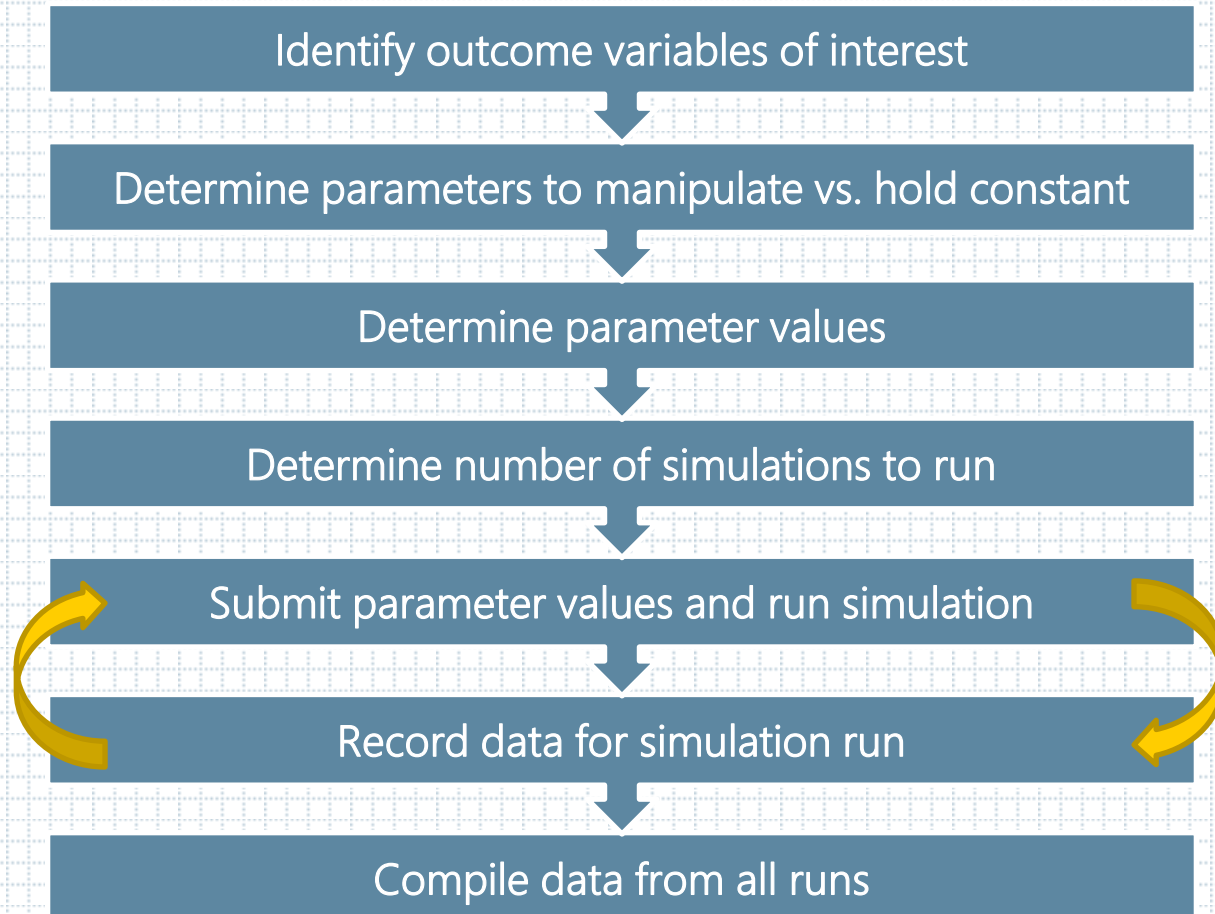
# SIMULATION DESIGN CONSIDERATIONS

- Simulation = conducting “virtual experiments”
  - Running computational model to examine questions, identify hypotheses, evaluate generative sufficiency, etc.
  - Different from *model fitting* or *parameter estimation*
    - » Using optimization procedures to statistically identify parameter values that account for existing/observed data

Computational Model	Simulation
<ul style="list-style-type: none"><li>• Algorithmic description of a dynamic process</li></ul>	<ul style="list-style-type: none"><li>• Systematic (i.e., experimental) investigation using computational model as data generator</li></ul>
<ul style="list-style-type: none"><li>• Contains mathematical and/or logical (e.g., if-then) statements of how system changes from one time point to the next</li></ul>	<ul style="list-style-type: none"><li>• Purposeful manipulation of model inputs and/or processes</li></ul>
<ul style="list-style-type: none"><li>• “Rules” describe how inputs become outputs over time</li></ul>	<ul style="list-style-type: none"><li>• Many purposes – prediction, exploration, explanation, demonstration, sufficiency, prescription</li></ul>

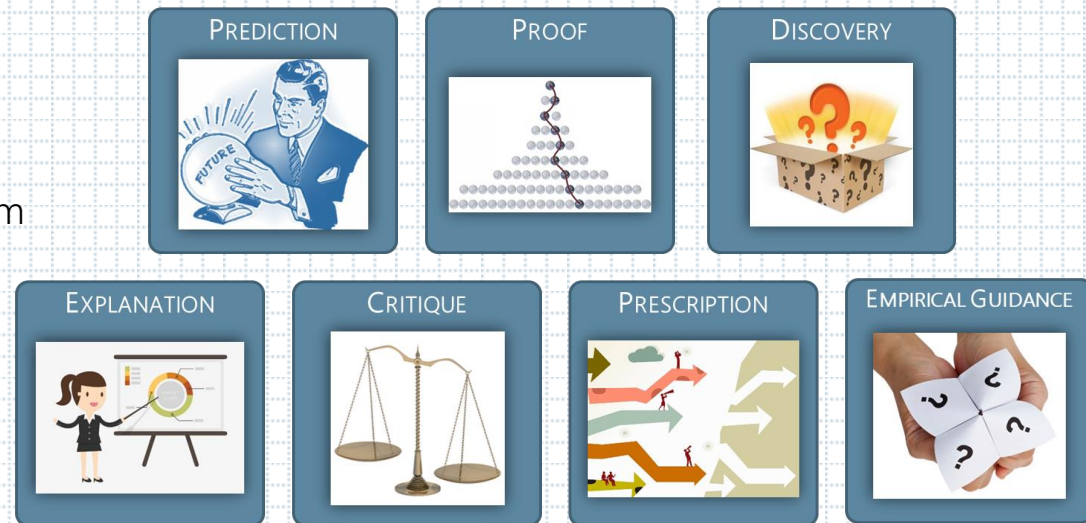
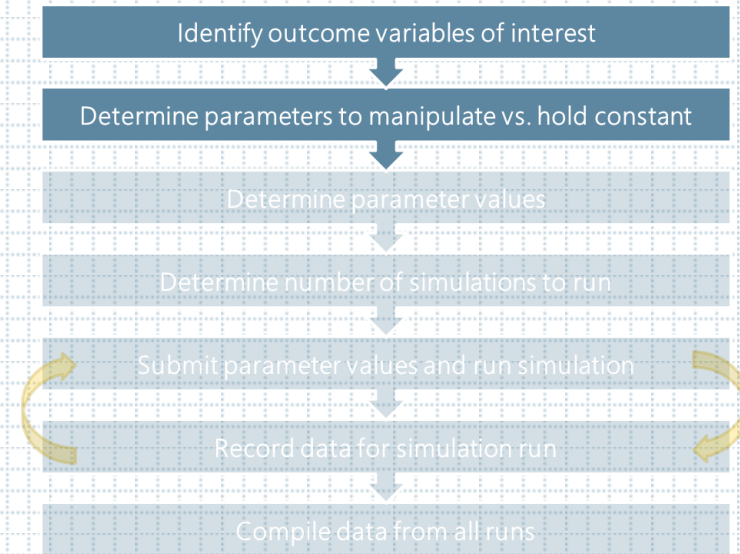
# SIMULATION DESIGN CONSIDERATIONS

- Overview of simulation procedure:



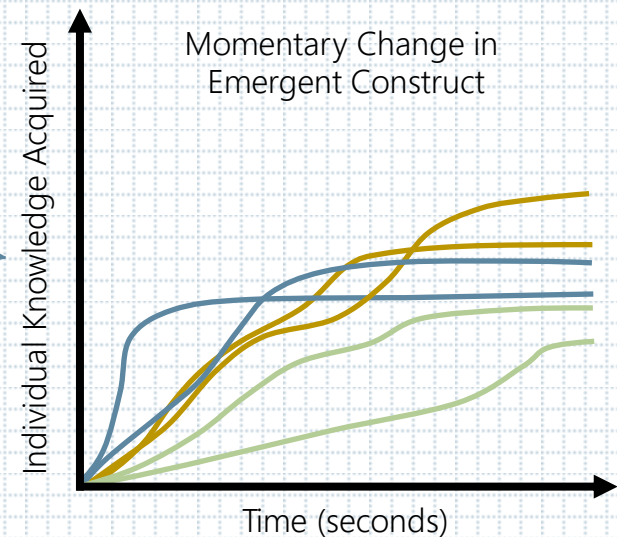
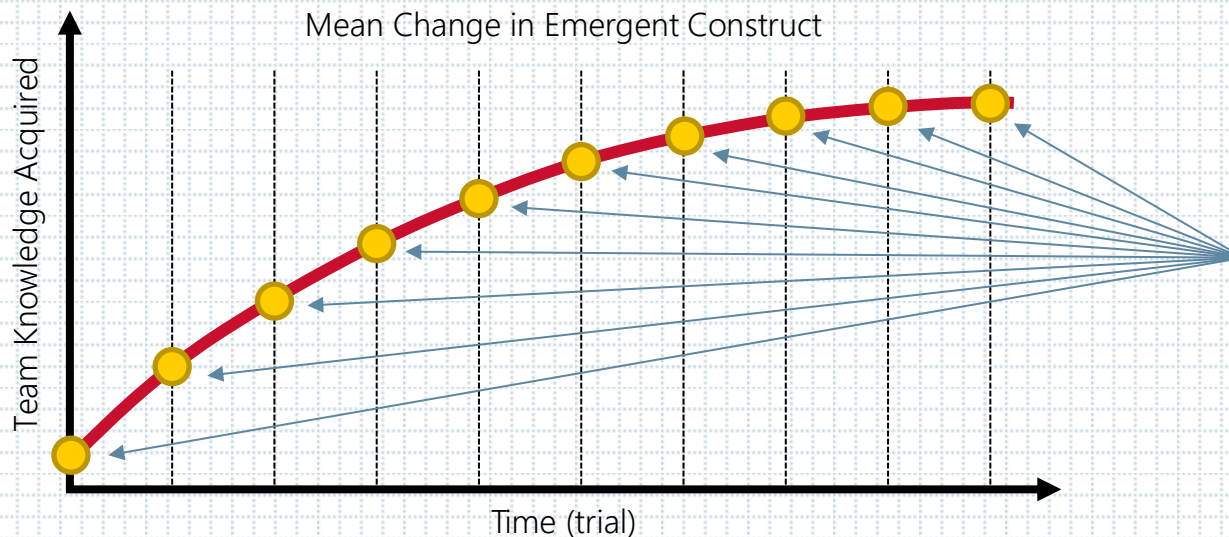
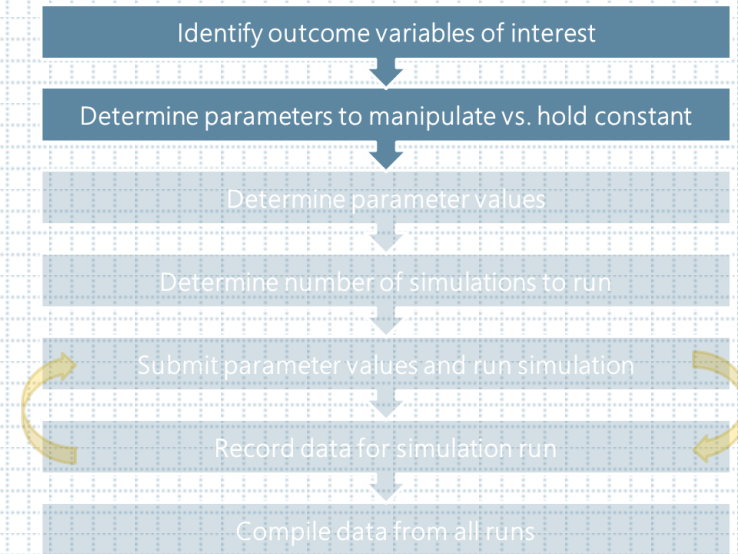
# SIMULATION DESIGN CONSIDERATIONS

- Like collecting human data, *purpose* of simulation will guide selection of outcomes and causal variables
  - Different purposes may have different needs for variable selection
  - For example:
    - » Prediction, explanation, & prescription → well-defined constraints
      - E.g., Which teams will perform best under adverse conditions?
    - » Proof, discovery, & critique → explore more broadly
      - E.g., What are the factors that contribute most strongly to team performance?



# SIMULATION DESIGN CONSIDERATIONS

- Considerations for variable and parameter selection
  - Outcome variables
    - » Level of analysis
    - » Means or dynamics?
    - 1. May want both → dynamic patterns help explain mean patterns





# SIMULATION DESIGN CONSIDERATIONS

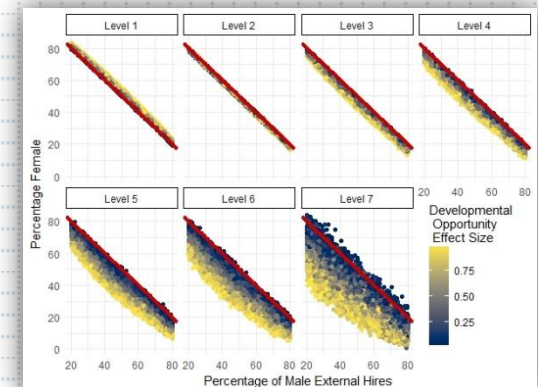
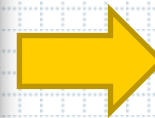
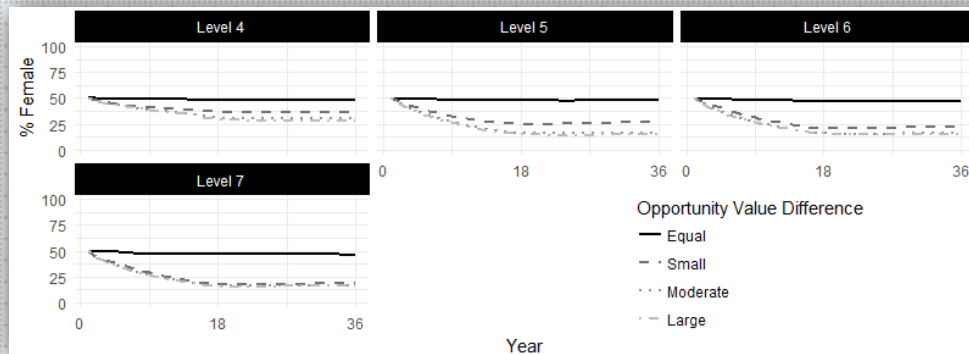
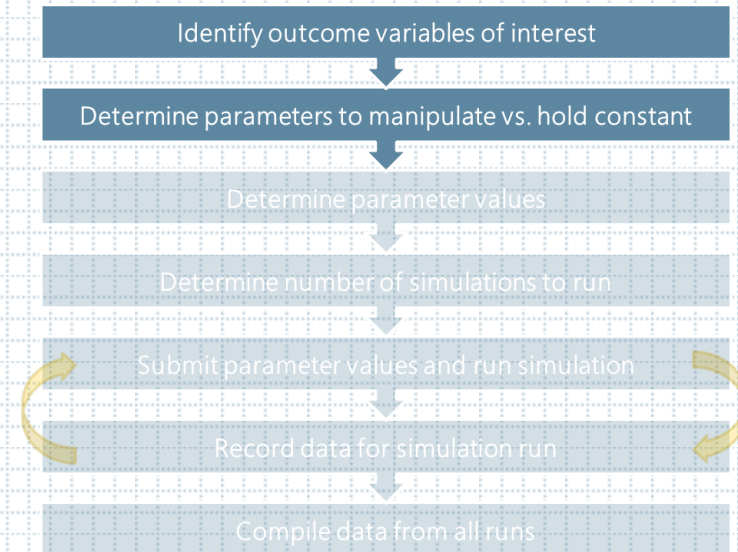
- Considerations for variable and parameter selection

- Causal variables

- » Models can easily end up with many more parameters than you'd feasibly want to manipulate...

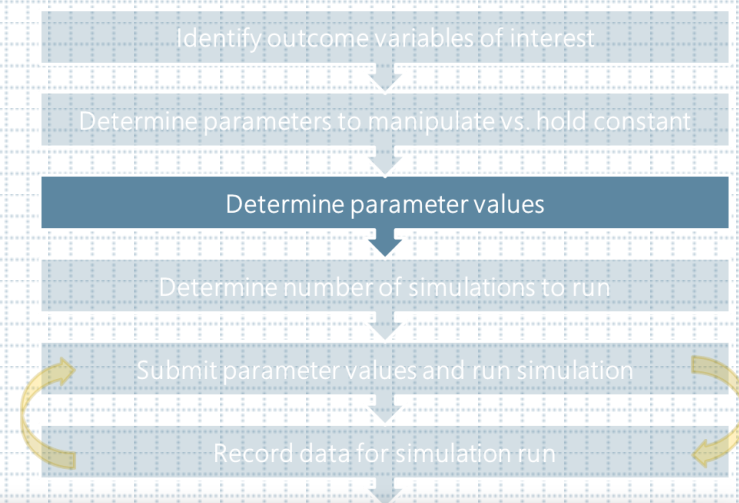
- » Sensitivity analyses

1. Testing robustness of interpretations to parameter & parameter value selection
2. May identify boundary conditions, inflection points, operationalization impacts



# SIMULATION DESIGN CONSIDERATIONS

- Parameter value selection also likely to depend on simulation purpose
  - E.g., critique/empirical guidance → choose simulation inputs/conditions equal to published/empirical values



## *The Availability Heuristic*

Tversky and Kahneman (1973) argued that people judge the frequency of events on the basis of the ease (or speed) with which relevant instances of those events are recalled or brought to mind. In one experiment, Tversky and Kahneman showed participants a list of 19 famous and 20 nonfamous names. Approximately 81% of the participants indicated that the famous names were more frequent, despite the fact they occurred slightly less frequently on the lists. This result was taken as evidence that the judged frequency of the “famous names” category is related to how easily these names could be brought to mind. A recall task showed a positive correlation between the number of names recalled and the judged frequency.

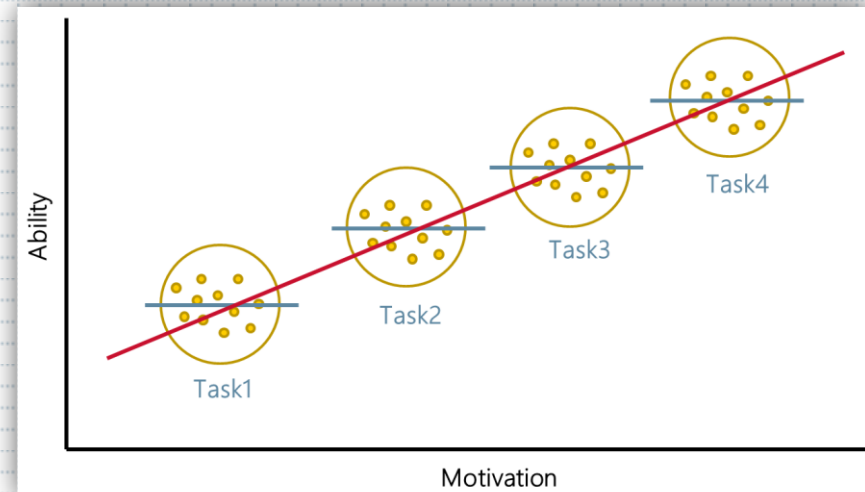
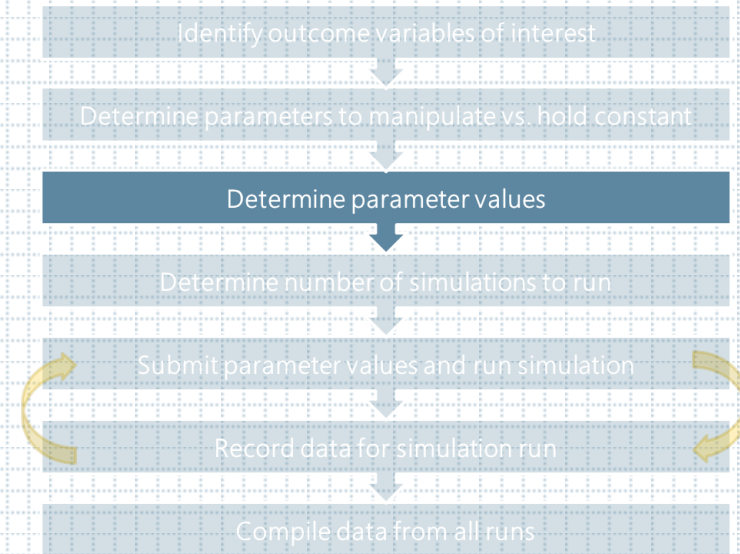
We generated four lists of famous and less famous male and female actors. These names were then used as search cues in a news database containing citations to these actors in major newspapers such as the *New York Times*, the *Chicago Tribune*, the *Washington Post*, and so on. The citation frequency for famous male actors was 46 times higher than less famous male actors, and the corresponding ratio for female actors was 26:1. These values were entered into MDM as the number of extraexperimental memory traces to be produced for every famous experimental trace.

**Simulation method.** Three simulations were conducted. Each simulation used 39 memory traces corresponding to the 19 famous names and the 20 less famous names. Either 0, 26, or 46 extraexperimental traces were created for every famous intraexperimental trace, whereas 0, 1, or 1 extraexperimental traces were used for every less famous trace. These numbers capture the 26:1 and 46:1 ratios discussed earlier. The first



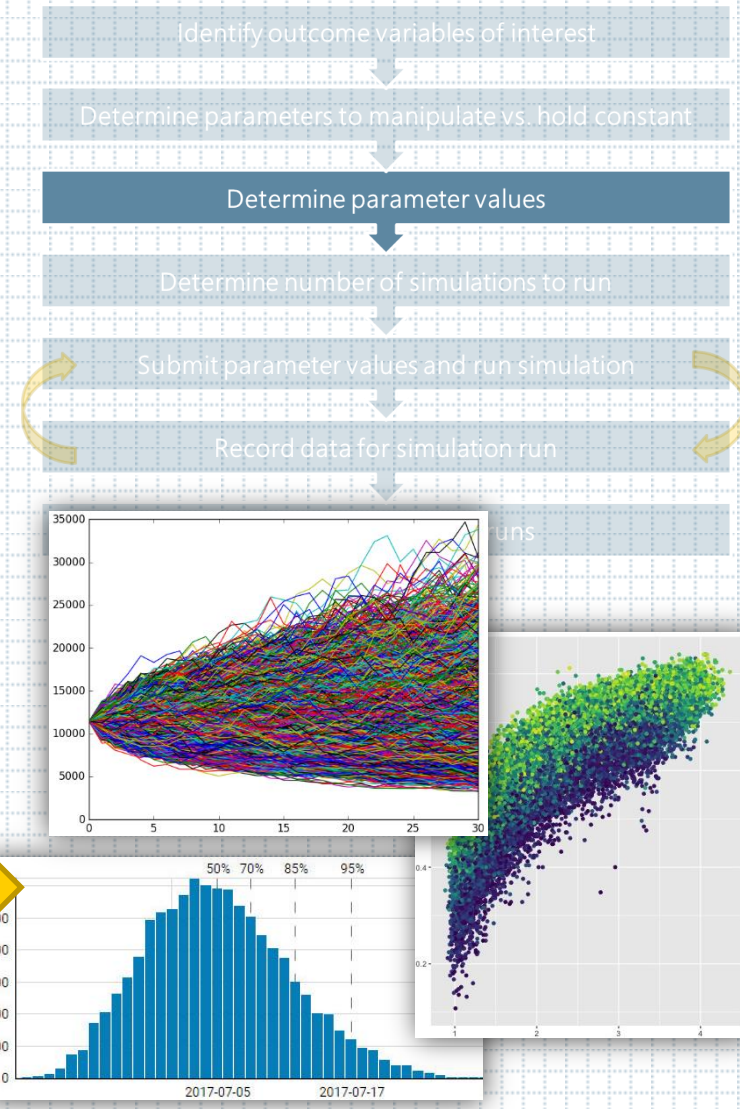
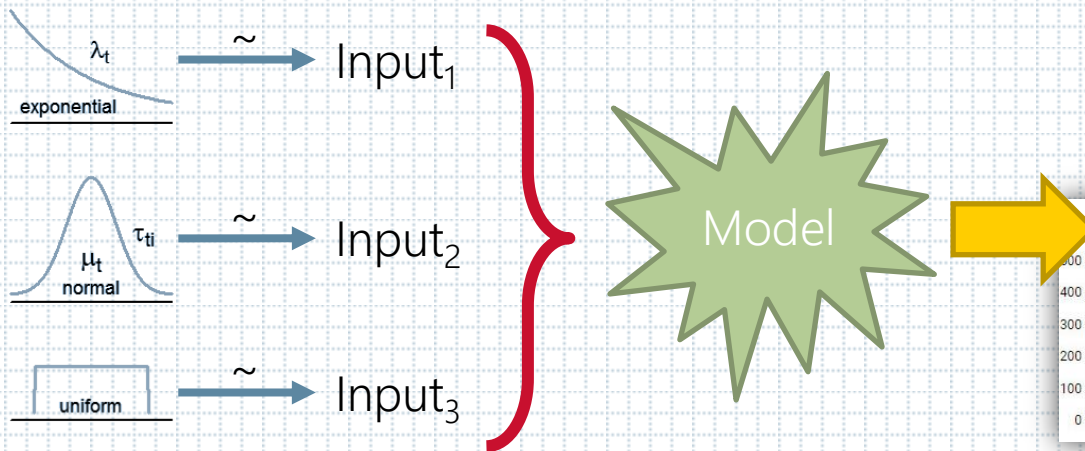
# SIMULATION DESIGN CONSIDERATIONS

- Considerations for selecting parameter values to simulate
  - Is model “attuned” to accept empirical estimates?
    - » E.g., Does “time” in the simulation correspond to time in reality?
  - Are empirical estimates appropriate for model?
    - » E.g., Does correlation between observed ability and motivation hold within and between different tasks?
  - In many cases, *relative* parameter values/differences are acceptable
    - » E.g., As member ability increases, team performance increases



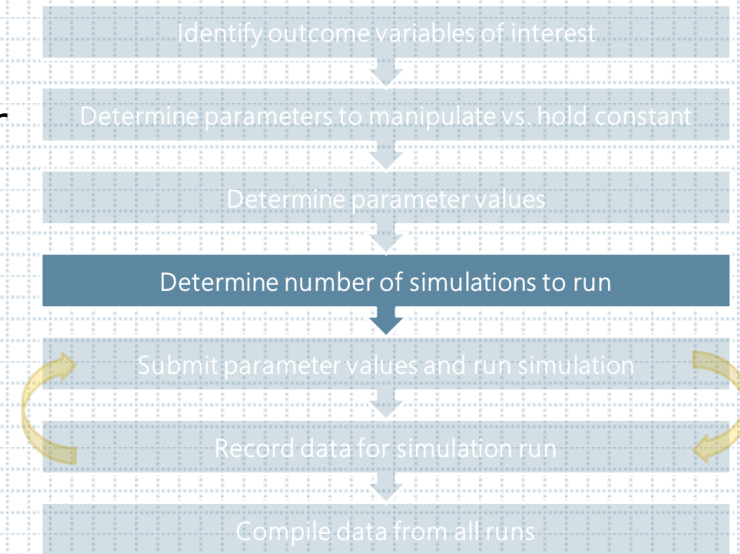
# SIMULATION DESIGN CONSIDERATIONS

- Considerations for selecting parameter values to simulate
  - What to do if you don't know or only have range of possible values?
  - Monte Carlo simulation methods
    - » Probability distribution over parameters
      1. Sensible, plausible, and/or defensible
    - » Sample values and run model many times
    - » Evaluate outcome distributions

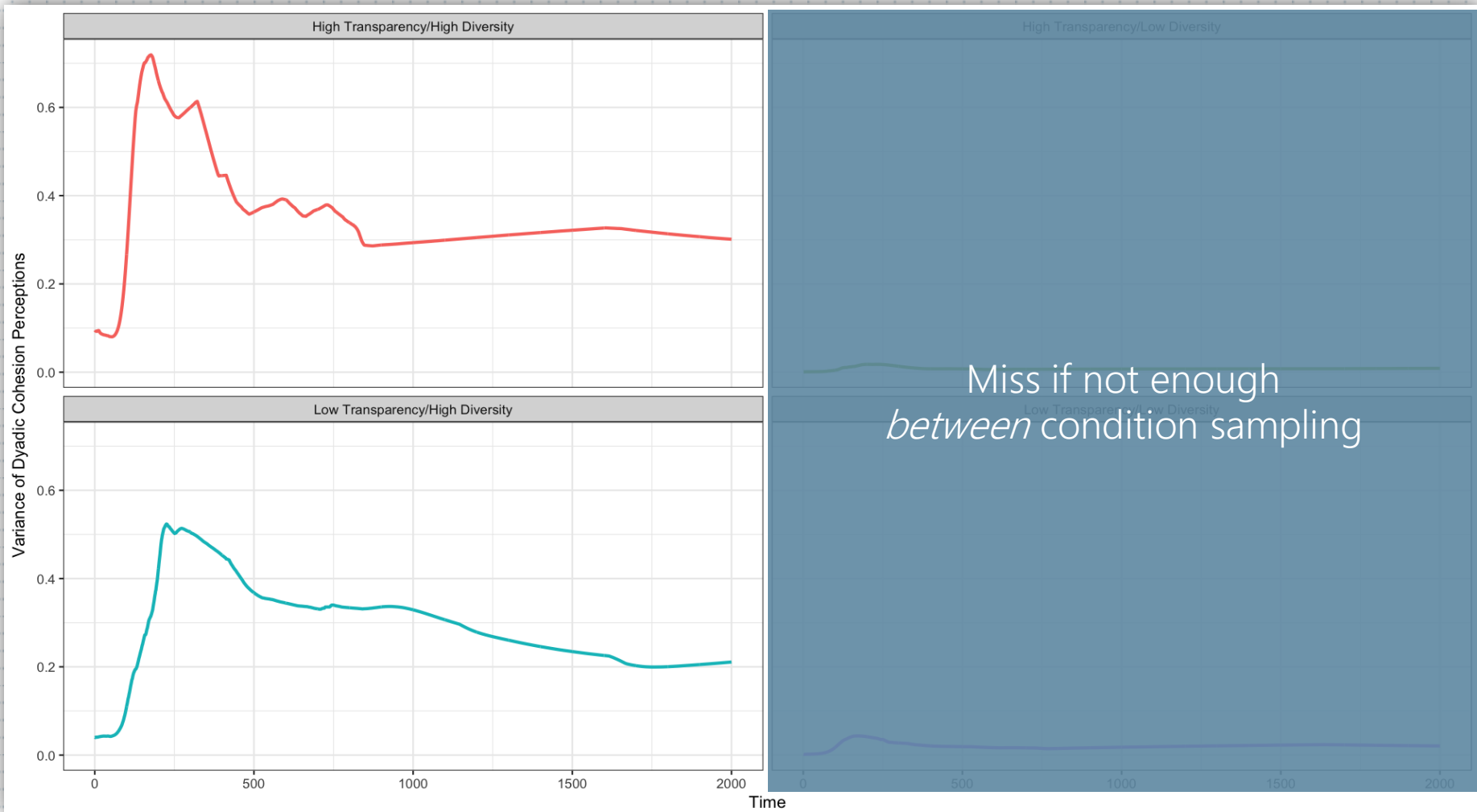


# SIMULATION DESIGN CONSIDERATIONS

- Simulation run = complete run of model under a given set of parameter values
  - Same as sample size in experiments → more runs = more confidence/stability
- Considerations for simulation runs
  - Number of observations *between* parameter value combinations...
    - » Monte Carlo methods need many runs to map parameter space

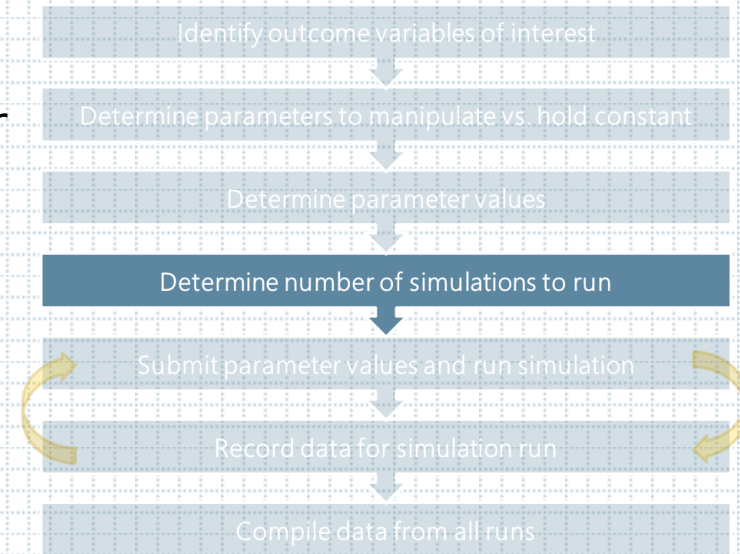


# SIMULATION DESIGN CONSIDERATIONS



# SIMULATION DESIGN CONSIDERATIONS

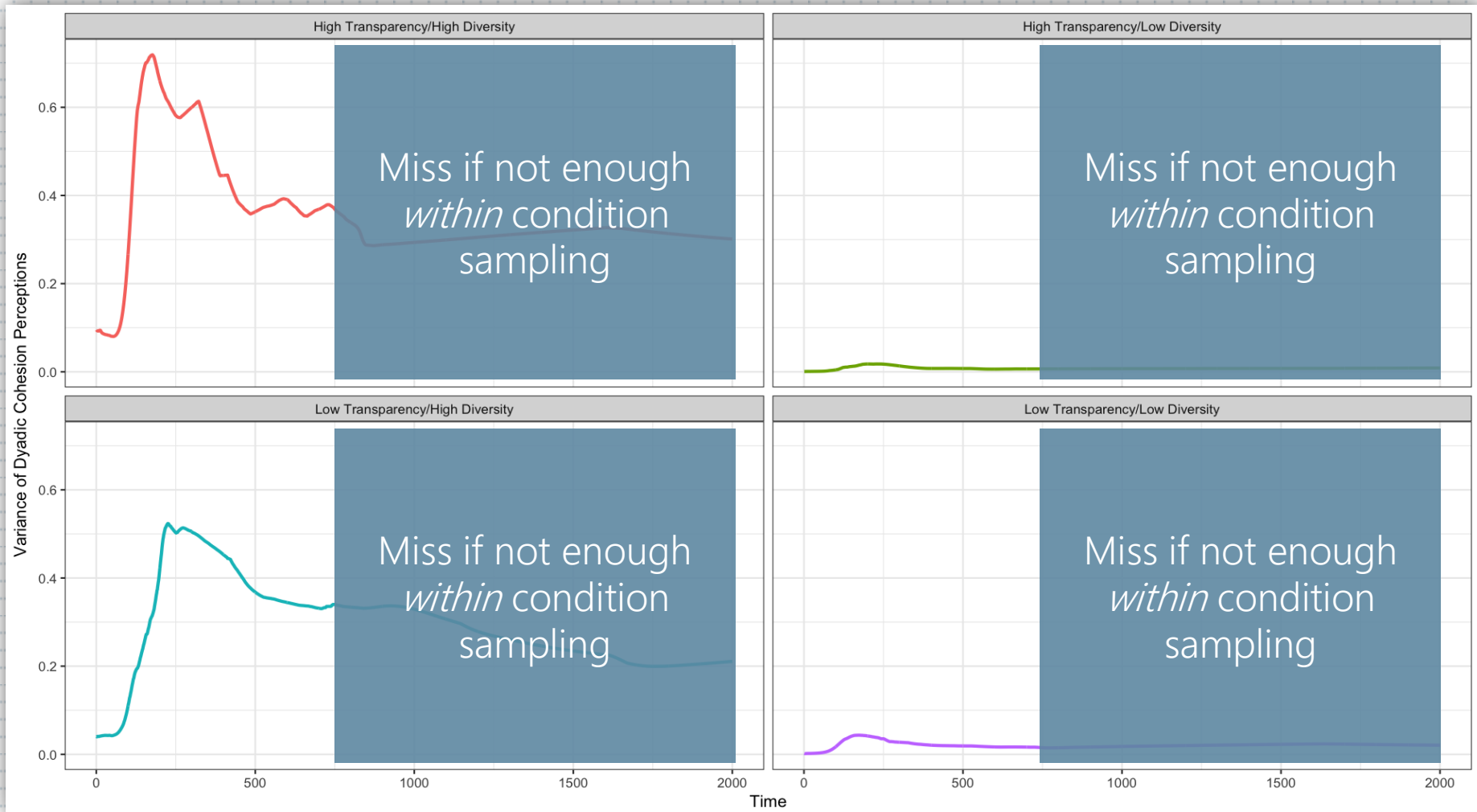
- Simulation run = complete run of model under a given set of parameter values
  - Same as sample size in experiments → more runs = more confidence/stability



- Considerations for simulation runs
  - Number of observations *between* parameter value combinations...
    - » Monte Carlo methods need many runs to map parameter space
  - ...and number of observations *within* parameter value combinations
    - » How many time points/ticks/iterations should model run?
    - » How many samples per parameter value combination to run?
    - » Depends on simulation purposes...

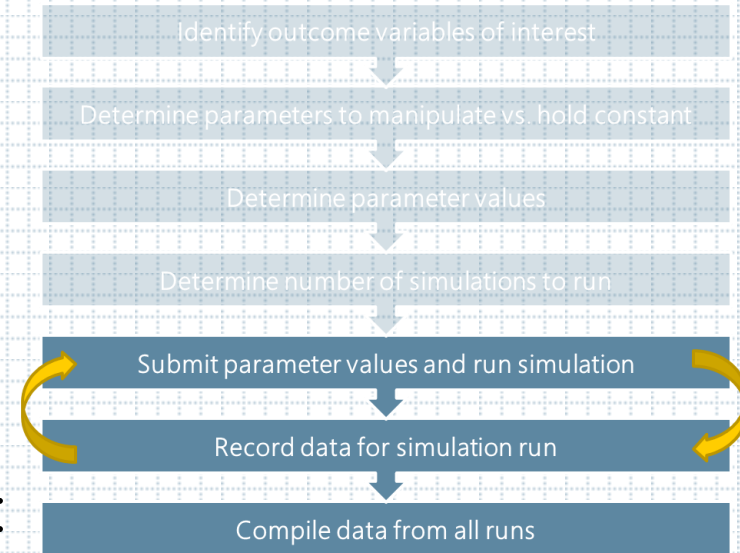


# SIMULATION DESIGN CONSIDERATIONS

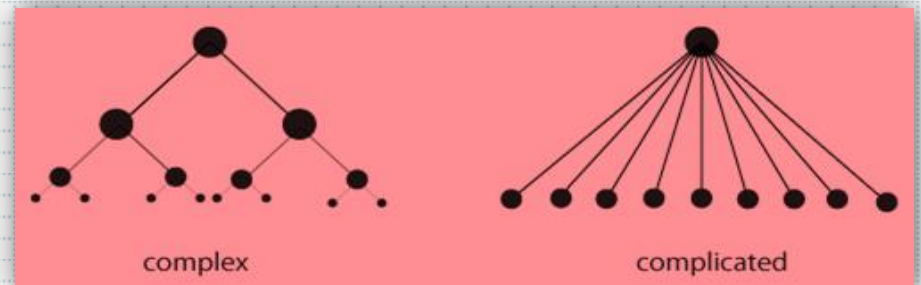
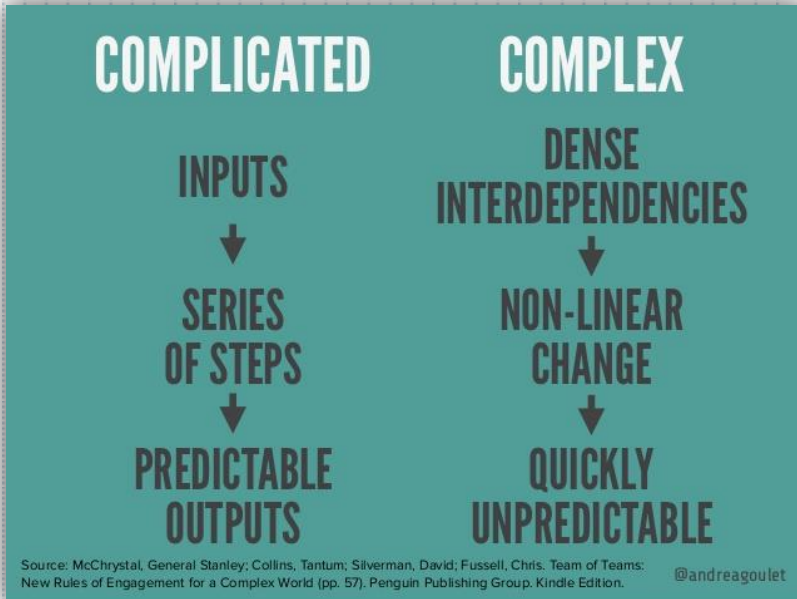
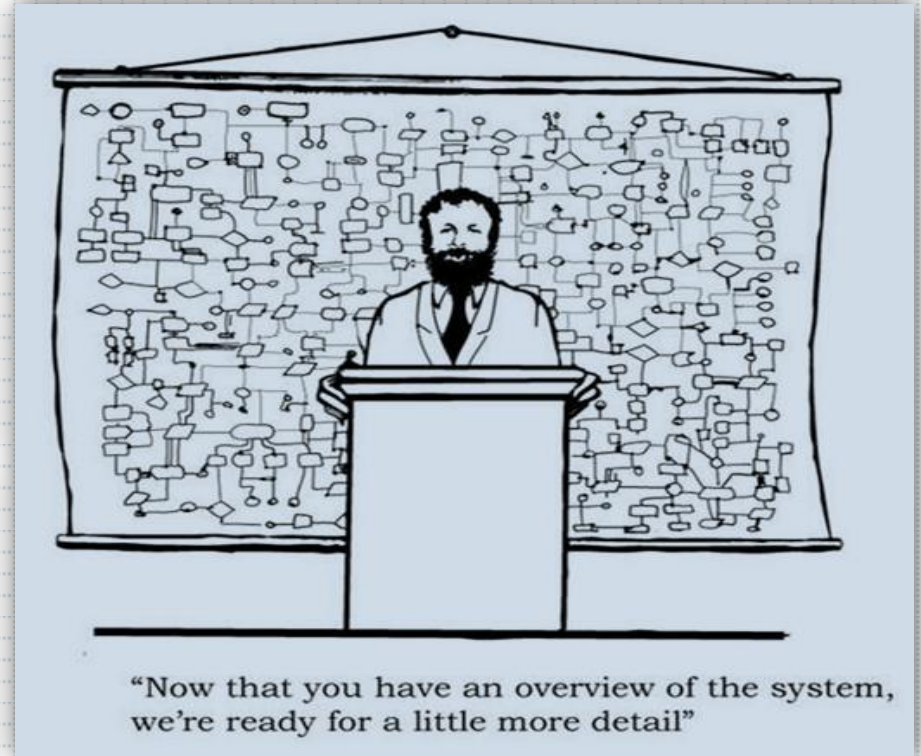
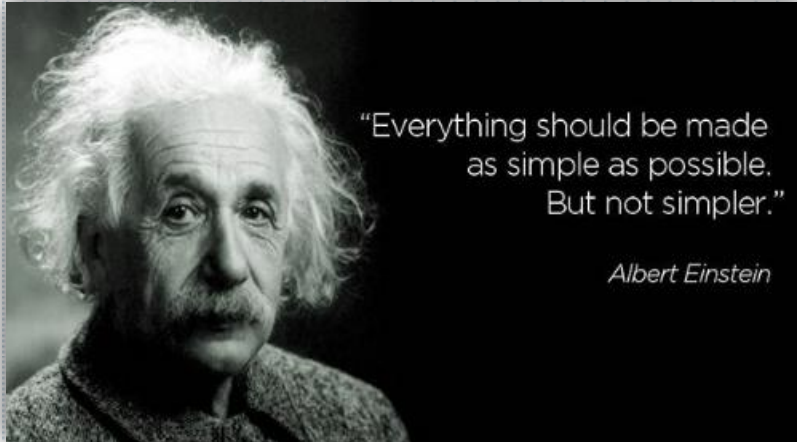


# SIMULATION DESIGN CONSIDERATIONS

- Remaining steps are operational
  - Depend on model code, simulation software, and hardware implementation
- Some general considerations for data:
  - What data should be recorded?
    - » What data is needed from a *single* simulation run?
      1. “Diagnostic” vs. “substantive” data
    - » How should data from a *single* simulation be stored/structured?
    - » Usually good to save simulation inputs as well!
  - What data needs to be aggregated/compiled?
    - » More efficient to compute metrics during vs. after simulation runs?



# COMPLEX VS. COMPLICATED



# COMPLEX VS. COMPLICATED

- Complicated

- Many (usually additive) inputs
- Many (potentially) difficult computations/calculations
- Output can be determined with sufficient compute time
- Improvements achieved through *efficiency*

- Complex

- High degree of interdependence among inputs
- Many (potentially) simple computations/calculations
- Outputs are emergent & difficult (impossible) to compute
- Improvements (usually) achieved through *simplification*

# COMPLEX VS. COMPLICATED

- Computational models can vary in terms of complexity and complication
- Simulations vary only in terms of complication
  - What makes simulations complicated?
    - » Simulating complicated (i.e., inefficient) models
    - » Manipulating many parameters simultaneously
- Helpful to have basic understanding of tradeoffs among complexity, complication, & computational demands





# COMPLEX VS. COMPLICATED

- Simulating complicated (i.e., inefficient) models
  - When initially coding model, don't worry too much about efficiency...
  - ...before preparing to run simulations, worth \*trying\* to improve code:
    - » Perform fewer computations
    - » Perform less complex computations
    - » Write code to take advantage of software strengths; e.g., in R:
      1. Use data structures that use less memory (e.g., lists instead of arrays, `data.table` > `matrix` > `data frames`)
      2. Vectorization (e.g., apply statements instead of for loops)
      3. Pre-allocate output (i.e., don't grow matrices)
      4. Use built-in/primitive functions instead of custom/package functions
  - Don't need to go crazy...but surprising how small changes can drastically cut back on computational time and cost!

# COMPLEX VS. COMPLICATED

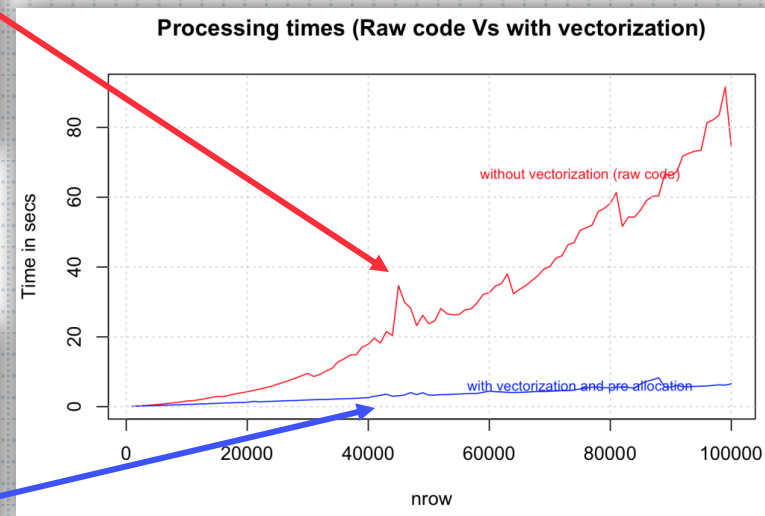
- “For every row in data frame (df), check if sum of all values is greater than 4. If yes, a new 5th variable gets the value “greater\_than\_4”, else, it gets “lesser\_than\_4”.”

```
1 # Original R code: Before vectorization and pre-allocation
2 system.time({
3   for (i in 1:nrow(df)) { # for every row
4     if ((df[i, "col1"] + df[i, "col2"] + df[i, "col3"] + df[i, "col4"]) > 4) { # check if > 4
5       df[i, 5] <- "greater_than_4" # assign 5th column
6     } else {
7       df[i, 5] <- "lesser_than_4" # assign 5th column
8     }
9   }
10 })
```

- No vectorization
- Add new row to data frame each step

```
1 # after vectorization and pre-allocation
2 output <- character(nrow(df)) # initialize output vector
3 system.time({
4   for (i in 1:nrow(df)) {
5     if ((df[i, "col1"] + df[i, "col2"] + df[i, "col3"] + df[i, "col4"]) > 4) {
6       output[i] <- "greater_than_4"
7     } else {
8       output[i] <- "lesser_than_4"
9     }
10   }
11   df$output <- output
12 })
```

- Pre-allocate output
- Add output vector to column all at once



# COMPLEX VS. COMPLICATED

- Manipulating many parameters simultaneously in simulations
  - Increasing number of parameters or number of levels per parameter increases computational demands *exponentially*
    - » Dichotomous parameters:
      1. Fully crossed  $\rightarrow 2^n$  conditions, where  $n$  = number of parameters
    - » Continuous parameters
      1. "Parameter sweeps"/Monte Carlo approaches  $\rightarrow$  require *many* runs to ensure parameter combinations encountered multiple times

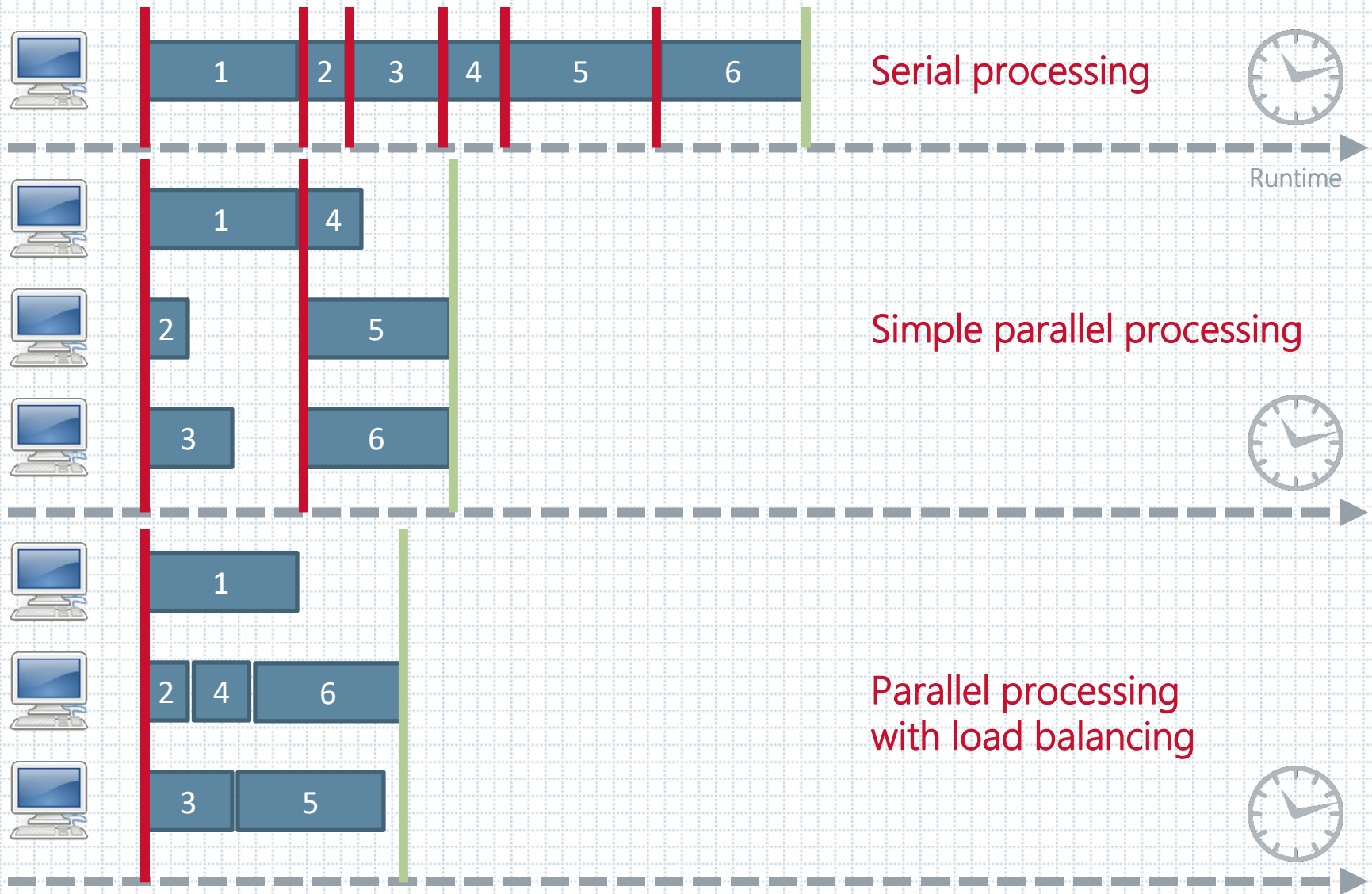
Order of growth

order of growth		$N$			
description	function	1	10	100	1000
constant	1	1	1	1	1
logarithmic	$\log N$	0	1	2	3
linear	$N$	1	10	100	1000
linearithmic	$N \log N$	0	10	200	3000
quadratic	$N^2$	1	100	10,000	1,000,000
cubic	$N^3$	1	$10^3$	$100^3$	$1000^3$
exponential	$2^N$	2	$2^{10}$	$2^{100}$	$2^{1000}$
Commonly encountered order-of-growth functions					

# COMPLEX VS. COMPLICATED

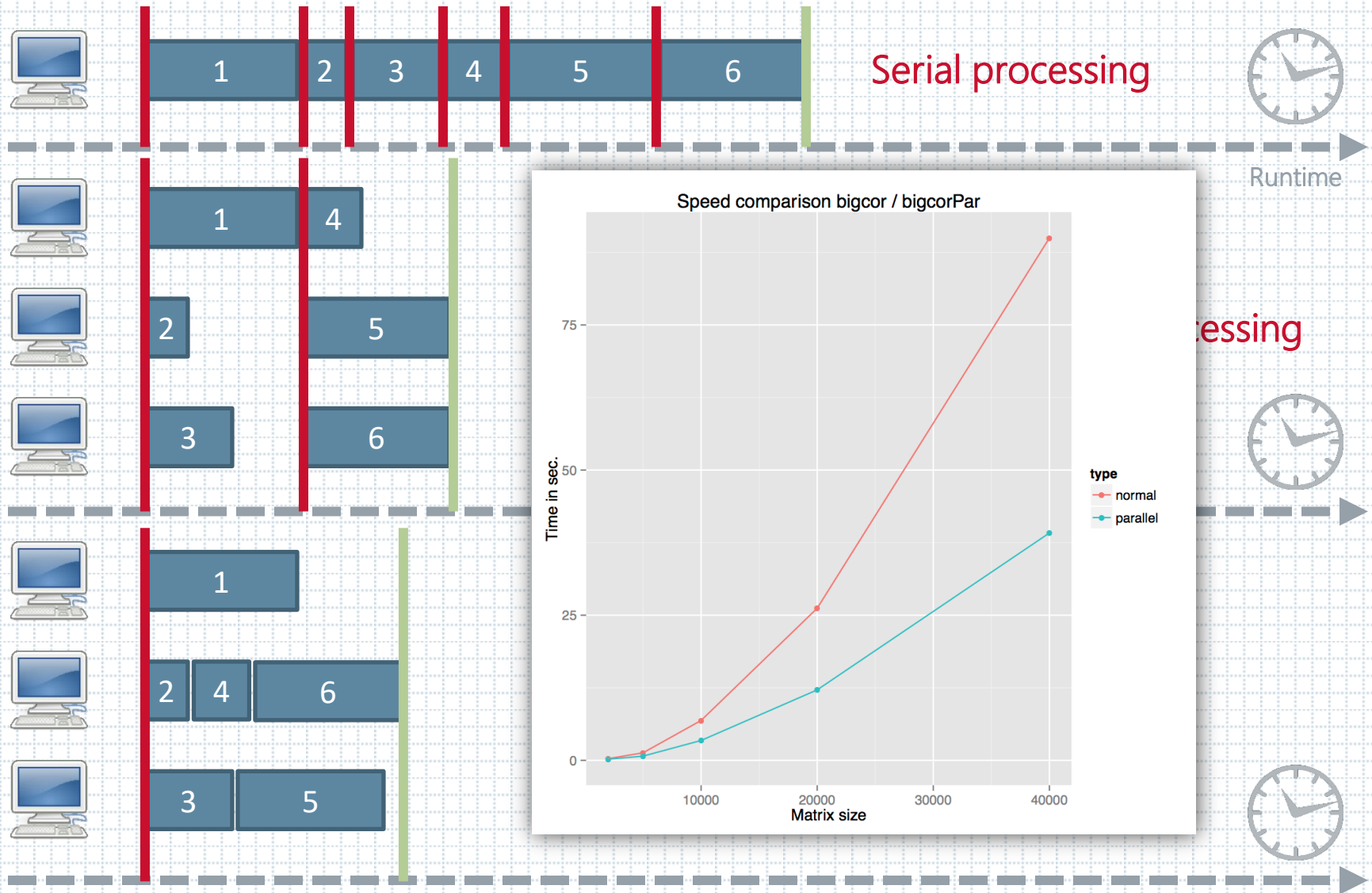
- Hardware offers “brute force” method for reducing compute time
  - CPU speed → faster CPU, **faster** runs
  - Number of CPUs → more CPUs, **more** runs
    - » Parallel processing & load balancing
    - » Note: R performs computations on single processor by default!

# COMPLEX VS. COMPLICATED





# COMPLEX VS. COMPLICATED



# COMPLEX VS. COMPLICATED

- Hardware offers “brute force” method for reducing compute time
  - CPU speed → faster CPU, **faster** runs
  - Number of CPUs → more CPUs, **more** runs
    - » Parallel processing & load balancing
    - » Note: R performs computations on single processor by default!
  - RAM → more RAM, **larger** runs
    - » How much data can be “active” and operated on at any given time

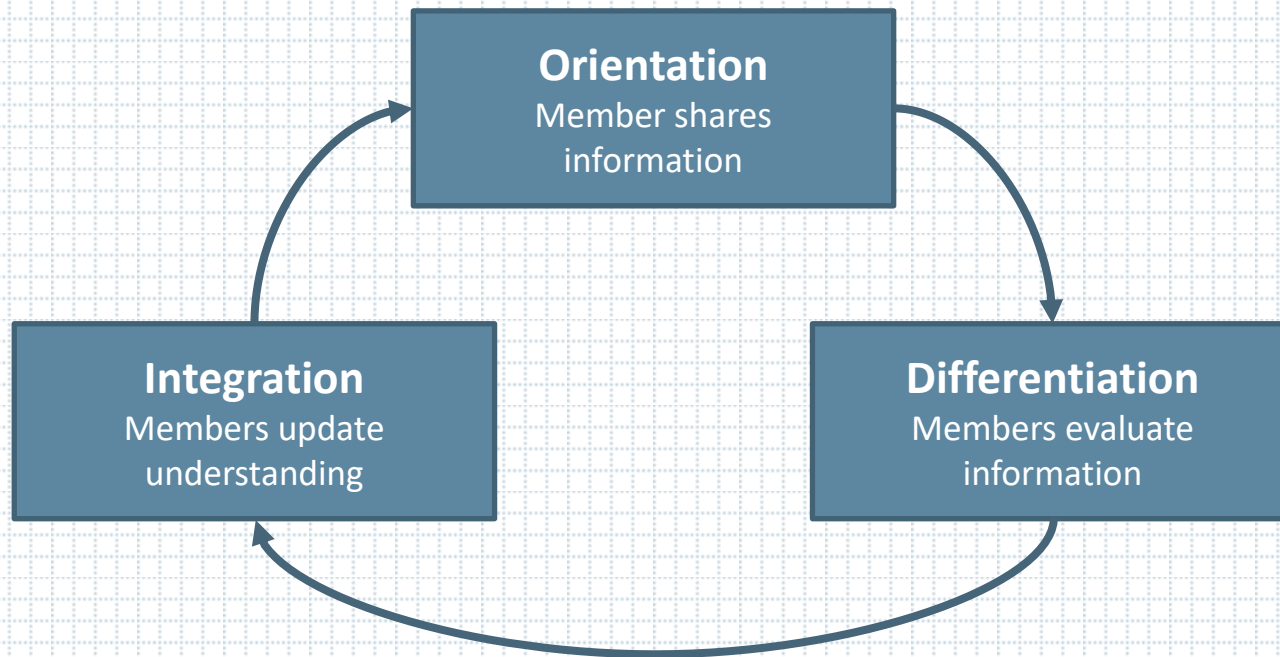
# BUILDING A SIMULATION

Any questions at this point?

Ready to start simulating!?

# BUILDING A SIMULATION

- Dionne, S.D., Sayama, H., Hao, C., & Bush, B.J. (2010). The role of leadership in shared mental model convergence and team performance improvement: An agent-based computational model. *Leadership quarterly*, 21, 1035-1049.
- Translates theory of **mental model convergence** into computational model:

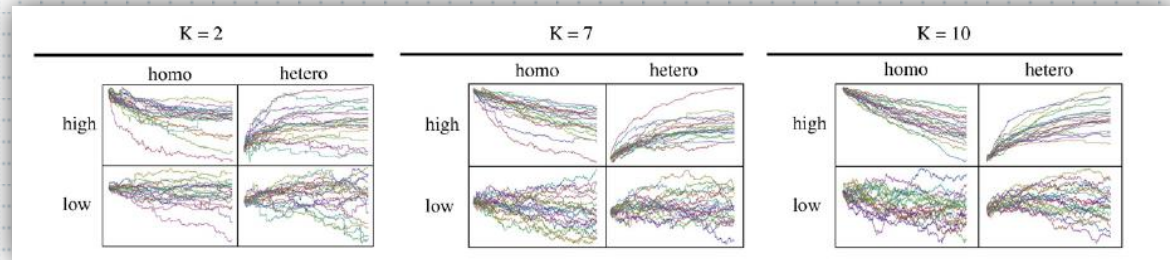


# BUILDING A SIMULATION

- Dionne, S.D., Sayama, H., Hao, C., & Bush, B.J. (2010). The role of leadership in shared mental model convergence and team performance improvement: An agent-based computational model. *Leadership quarterly*, 21, 1035-1049.
- Will use full computational model code (available on GitHub for download)
  - MMConverge.R
  - Completed process code from yesterday's exercises (Steps 3 and 4)
  - Also includes primary outcome variables of interest:

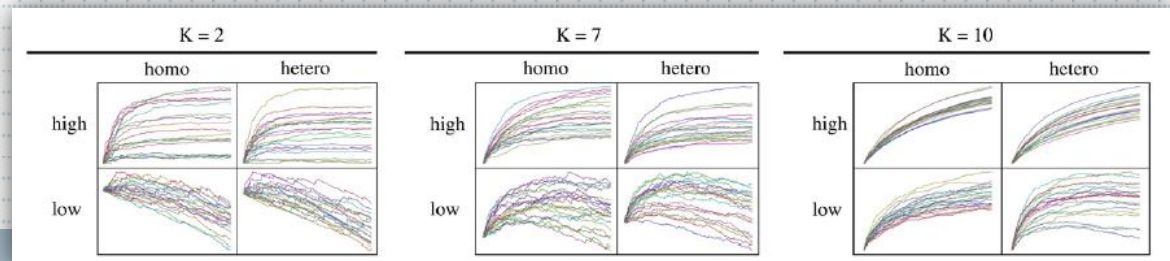
## Mental model accuracy

Sum of squared difference between GPF and TPF



## Mental model convergence

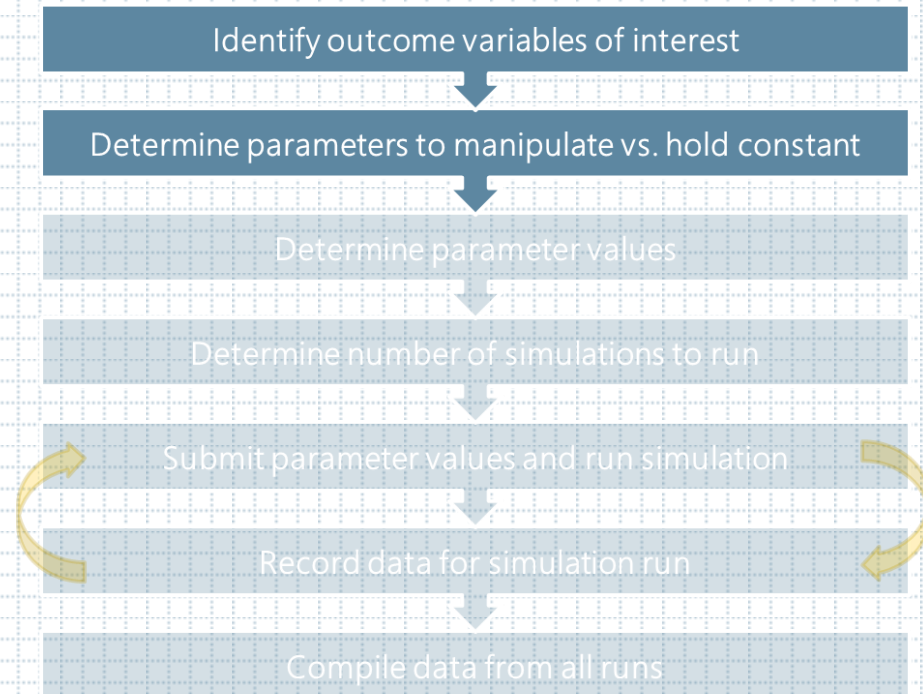
Sum of Euclidean distance between each pair of agents' confidence perceptions





# BUILDING A SIMULATION

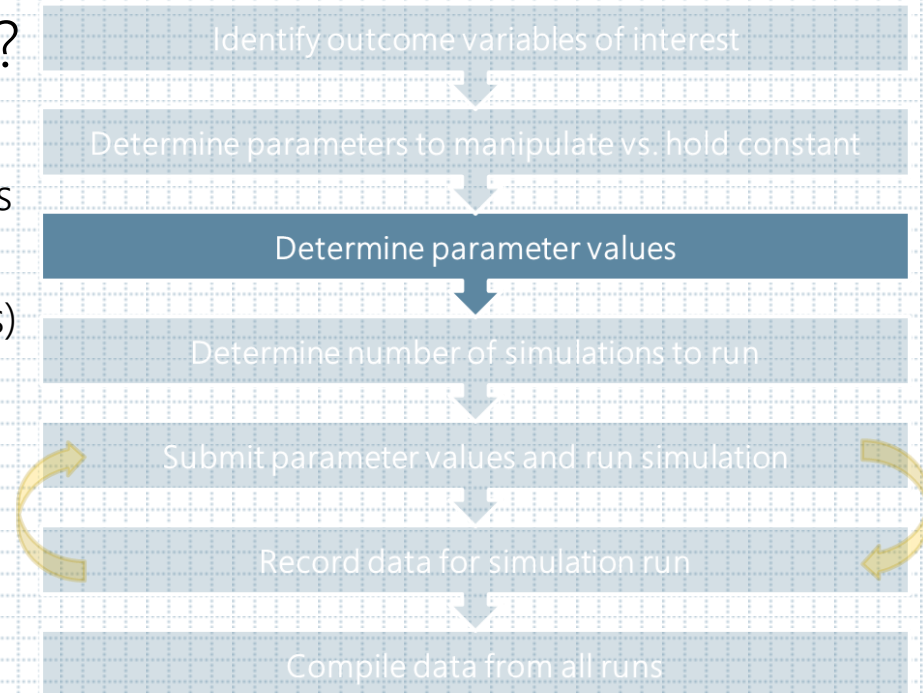
- What are parameters to be manipulated?
  - H: group heterogeneity in expertise domains
  - M: level of mutual interest
  - K: number of members in leader's in-group



- Question: What are other parameter values we could manipulate? What other questions could we look at?

# BUILDING A SIMULATION

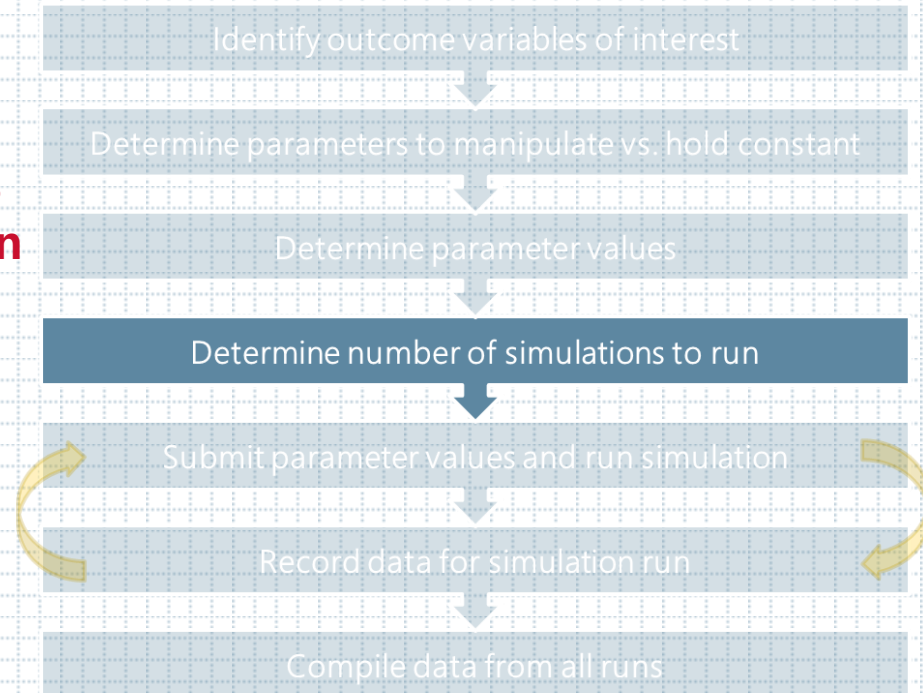
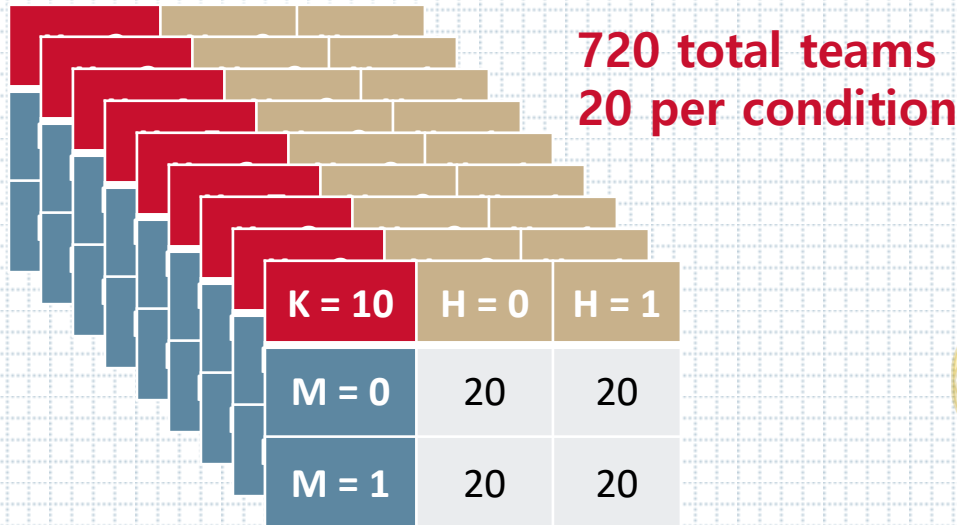
- What are the parameter values?
  - Manipulated:
    - » H: 0 = homogenous, 1 = heterogeneous
    - » M: 0 = self-interest, 1 = mutual interest
    - » K = 2-10 (number of in-group members)
  - Constant:
    - »  $n = 10$
    - »  $n_{\text{iter}} = 500$
    - »  $\lambda = .001$  (learning rate)



All data inputs are from synthetic data, generated from parameters guided by theory and established specifically for this simulation. The parameter settings we systematically experimented were:  $H = 0$  or  $1$ ,  $M = 0$  or  $1$ , and  $K = 2, 3, \dots 10$ . Thus, the total number of simulation runs conducted was  $2 \times 2 \times 9 \times 20 = 720$ . Although simulation runs can be any size, one can quickly reach a point where additional runs provide no additional significant information. We were more than satisfied that 20 simulation runs revealed enough similarity to indicate that no additional runs would be productive. More information on use of Monte Carlo simulations within team research can be found in [Dionne and Dionne \(2008\)](#) and [Sayama et al. \(unpublished\)](#).

# BUILDING A SIMULATION

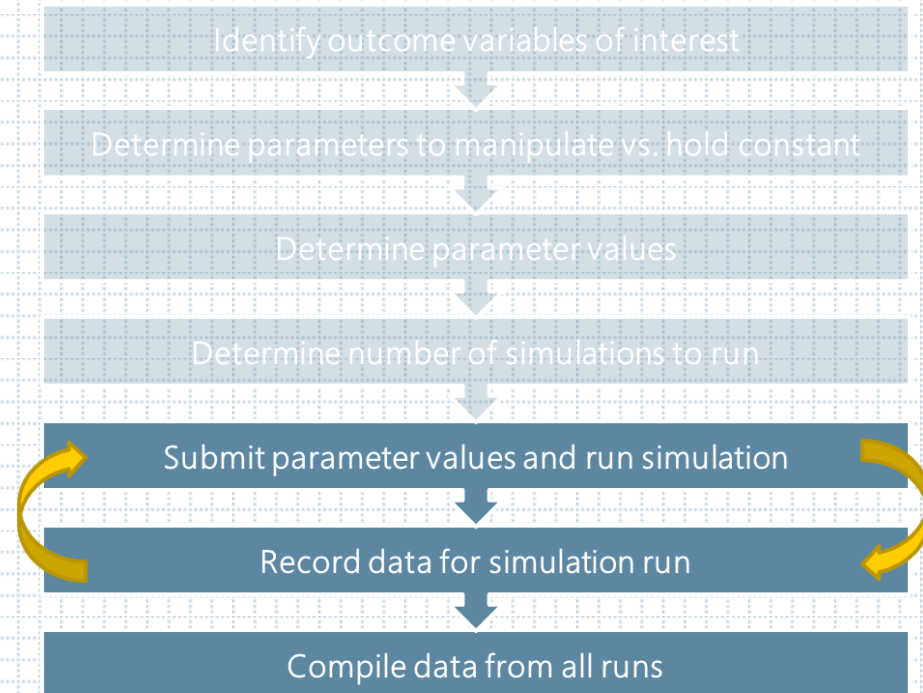
- How many simulations to run?



All data inputs are from synthetic data, generated from parameters guided by theory and established specifically for this simulation. The parameter settings we systematically experimented were:  $H = 0$  or  $1$ ,  $M = 0$  or  $1$ , and  $K = 2, 3, \dots, 10$ . Thus, the total number of simulation runs conducted was  $2 \times 2 \times 9 \times 20 = 720$ . Although simulation runs can be any size, one can quickly reach a point where additional runs provide no additional significant information. We were more than satisfied that 20 simulation runs revealed enough similarity to indicate that no additional runs would be productive. More information on use of Monte Carlo simulations within team research can be found in [Dionne and Dionne \(2008\)](#) and [Sayama et al. \(unpublished\)](#).

# BUILDING A SIMULATION

- Submit parameter values and run simulation
- Record data for simulation run
- Compile data from all runs
- Code we have is currently set up to run 1 team...
  - Need a procedure in R that allows us to:
    - » Run this code 720 times
    - » Systematically change input parameters
  - How could we accomplish this?



# BUILDING A SIMULATION

- Basic procedure for creating a simulation in R
  - Transform model code into a custom function
  - Make parameters to be manipulated arguments of function
    - » Update model code to use arguments!
  - Create new simulation script
    - » Specify number of simulations to run
    - » Specify design matrix of parameter values for simulation runs
    - » Source model function
    - » Pass parameter values to model function and run until desired number of simulations completed
    - » Format and save output data
  - (We will also set up our simulation to take advantage of parallel processing)

# BUILDING A SIMULATION

- Exercise: Work with partner to transform model code into a function called “mmconverge”

— Hint:

» Remember how creating a function in R works:

Save function to an object

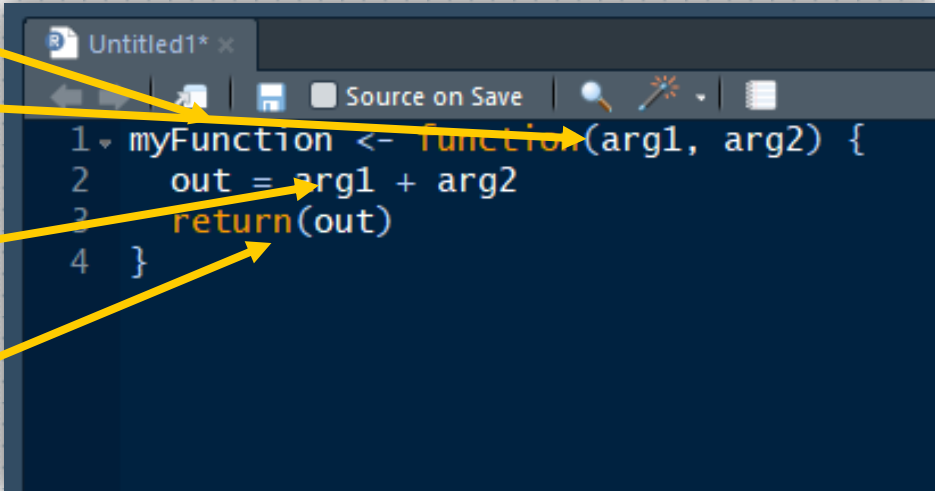
Name arguments to function

- Don't set these variables in the model code!

Place operations in { }

Specify what to return in a list

- mmStats
- GPF
- IPF
- TPF
- speaker
- teamNet



```
1 myFunction <- function(arg1, arg2) {  
2   out = arg1 + arg2  
3   return(out)  
4 }
```



# BUILDING A SIMULATION

- Basic procedure for creating a simulation in R
  - ~~Transform model code into a custom function~~
  - ~~Make parameters to be manipulated arguments of function~~
    - » ~~Update model code to use arguments!~~
  - Create new simulation script
    - » Specify number of simulations to run
    - » Specify design matrix of parameter values for simulation runs
    - » Source model function
    - » Pass parameter values to model function and run until desired number of simulations completed
    - » Format and save output data

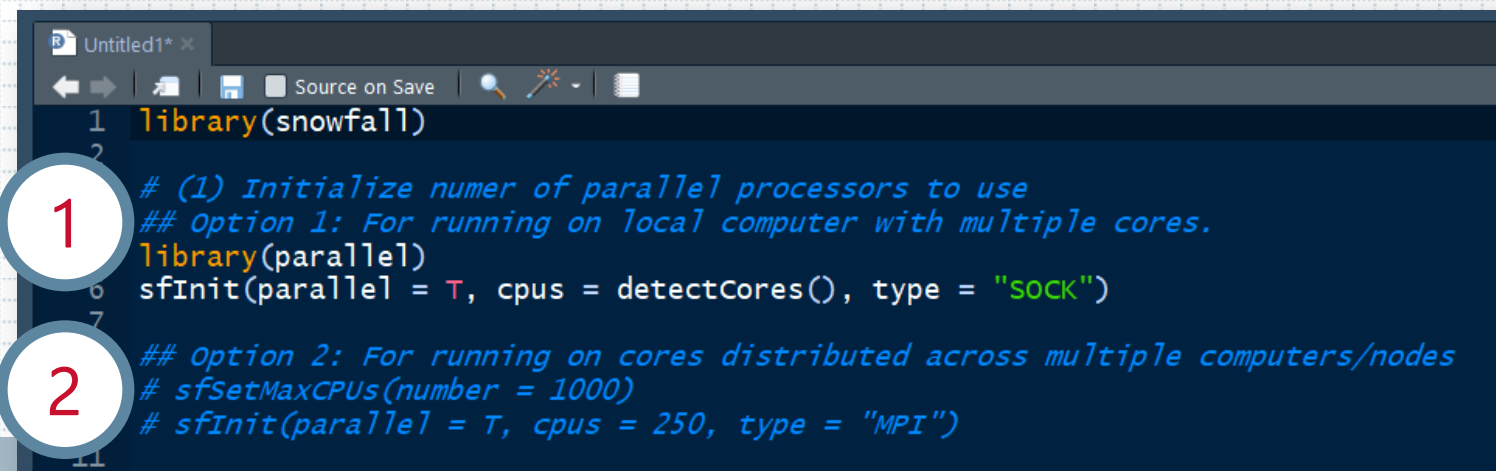
# BUILDING A SIMULATION

- Structure of simulation file for parallel computing

```
MMConvergeSim_template.R* x
1 library(snowfall)
2
3 # (1) Initialize number of parallel processors to use
4 ## Option 1: For running on local computer with multiple cores
5 library(parallel)
6 sfInit(parallel = T, cpus = detectCores(), type = "sock")
7
8 ## Option 2: For running on cores distributed across multiple computers/nodes
9 # sfSetMaxCPUs(number = 1000)
10 # sfInit(parallel = T, cpus = 250, type = "MPI")
11
12 ## (2) Create experimental conditions to run
13 ## nTeams PER condition to run
14
15 ## Design matrix
16
17 # (3) Load model scripts
18
19 # (4) Export all data and packages necessary to run the model
20 sfLibrary()
21 sfExportAll()
22 sfClusterSetupRNG()
23
24 # (5) Run simulation
25
26 # (6) Stop cluster and return R to running on single-core
27 sfStop()
28
29 ### (7) Save raw data
30
31 ## (8) Save formatted data
```

# BUILDING A SIMULATION

- Note on parallel processing in R
  - snowfall: convenience wrapper for accessing the snow package in R (simple network of workstations)
    - » parallel package has similar functionality...I just prefer snowfall
  - Two options for parallel computing:
    - » Local → utilizes multiple cores on single computer
    - » Distributed → utilizes multiple cores on different computers
  - “Embarrassingly parallel” → no/little communication across cores
    - » Each core runs *entire* model independently



```
1 library(snowfall)
2
3 # (1) Initialize number of parallel processors to use
4 ## Option 1: For running on local computer with multiple cores.
5 library(parallel)
6 sfInit(parallel = T, cpus = detectCores(), type = "sock")
7
8 ## Option 2: For running on cores distributed across multiple computers/nodes
9 # sfSetMaxCPUs(number = 1000)
10 # sfInit(parallel = T, cpus = 250, type = "MPI")
11
```

# BUILDING A SIMULATION

- Note on parallel processing in R
  - What needs to happen to run parallel computations in R:
    - » Initialize cluster and number of cores to enlist
      1. `sfInit(parallel = T, cpus = xxx, type = xxx)`
    - » Export ALL packages and data to each enlisted core
      1. `sfLibrary(package)` → one of these for each library to export
      2. `sfExportAll()` → exports everything in global environment to each spawn
        - A. Use `sfExport()` for more control
      3. `sfClusterSetupRNG()` → sets RNG sampler for each spawn
    - » Execute model in parallel
      1. Use `sfApply` family of functions to run model on each core
    - » Stop cluster
      1. `sfStop()`
      2. IMPORTANT to stop!! Many R functions won't work and R will crash if try to run things while cores are enlisted

# BUILDING A SIMULATION

- Creating design matrix to use in parallel processing
  - I prefer to create a run x parameter matrix:
    - »  $\text{length}(\text{run}) = \text{total}$  number of simulations to run
    - »  $\text{length}(\text{parameter}) = \text{total}$  number of parameters to manipulate
  - Lots of ways to create design matrix in R...just make sure to end up with a run x parameter data.frame (or matrix)

H	M	K
0	0	2
0	0	3
...	...	...
0	1	2
0	1	3
...	...	...
1	0	2
1	0	3
...	...	...

# BUILDING A SIMULATION

- Creating design matrix to use in parallel processing
  - Some helpful tips:
    - » `expand.grid()` → crosses values of all given vectors together
      1. If all parameters categorical, use `expand.grid()` and `rep()` matrix number of times needed
    - » Monte Carlo → sample as many values as needed from desired distribution and combine into single data frame

H	M	K
0	0	2
0	0	3
...	...	...
0	1	2
0	1	3
...	...	...
1	0	2
1	0	3
...	...	...



# BUILDING A SIMULATION

- Exercise: Work with partner to create design matrix called “conds” to replicate Dionne et al. simulations
  - Hints:
    - » Try `expand.grid()`...
    - » If you use `expand.grid()`, how could you replicate each of these rows?
      1. Try `conds[rep(1, 3) ,]`...What does this do? How could you use this logic to control the number of simulations to run?
    - » Make your code flexible to be able to specify as many runs per parameter combination as you want!

All data inputs are from synthetic data, generated from parameters guided by theory and established specifically for this simulation. The parameter settings we systematically experimented were:  $H = 0$  or  $1$ ,  $M = 0$  or  $1$ , and  $K = 2, 3, \dots, 10$ . Thus, the total number of simulation runs conducted was  $2 \times 2 \times 9 \times 20 = 720$ . Although simulation runs can be any size, one can quickly reach a point where additional runs provide no additional significant information. We were more than satisfied that 20 simulation runs revealed enough similarity to indicate that no additional runs would be productive. More information on use of Monte Carlo simulations within team research can be found in [Dionne and Dionne \(2008\)](#) and [Sayama et al. \(unpublished\)](#).

# BUILDING A SIMULATION

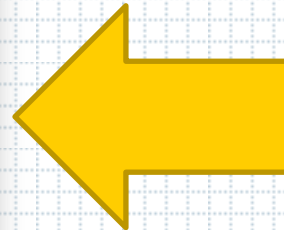
- Basic procedure for creating a simulation in R
  - ~~Transform model code into a custom function~~
  - ~~Make parameters to be manipulated arguments of function~~
    - » ~~Update model code to use arguments!~~
  - Create new simulation script
    - » Specify number of simulations to run
    - » Specify design matrix of parameter values for simulation runs
    - » Source model function
    - » Pass parameter values to model function and run until desired number of simulations completed
    - » Format and save output data

# BUILDING A SIMULATION

- Source model function
  - At this point, we have:
    - » File with model script transformed into function
    - » File with simulation script that contains design matrix
  - Simulation script needs “access” to the model function

## Simulation file

```
MMConvergeSim.R
1 # ----- #
2 # Run Simulation #
3 # ----- #
4 library(snowfall)
5
6 # (1) Initialize number of parallel processors to use
7 ## Option 1: For running on local computer with multiple cores. warning: This will
8 library(parallel)
9 sfInit(parallel = T, cpus = detectCores(), type = "sock")
10
11 ## Option 2: For running on cores distributed across multiple computers/nodes
12 # sfSetMaxCPUs(number = 1000)
13 # sfInit(parallel = T, cpus = 250, type = "MPI")
14
15 ## (2) Create experimental conditions to run. To make the call to use parallel pro
16 ## nTeams PER condition to run
17 nTeams = 20
18 ## H is group heterogeneity in domains of expertise [0,1]; H = 1 means agents have
19 ## K is number of members in the in-group network; K = 2:n-1
20 ## M is level of mutual interest [0,1]; M = 0 means agents determine whether to up
21 conds = expand.grid(H = 0:1, M = 0:1, K = 2:10)
22 conds = conds[rep(seq(nrow(conds)), nTeams),]
23 conds = conds[order(conds$H, conds$M, conds$K),]
24 rownames(conds) <- 1:nrow(conds)
25 condsList <- split(conds, seq(nrow(conds))) # Turns condition data into list so th
```

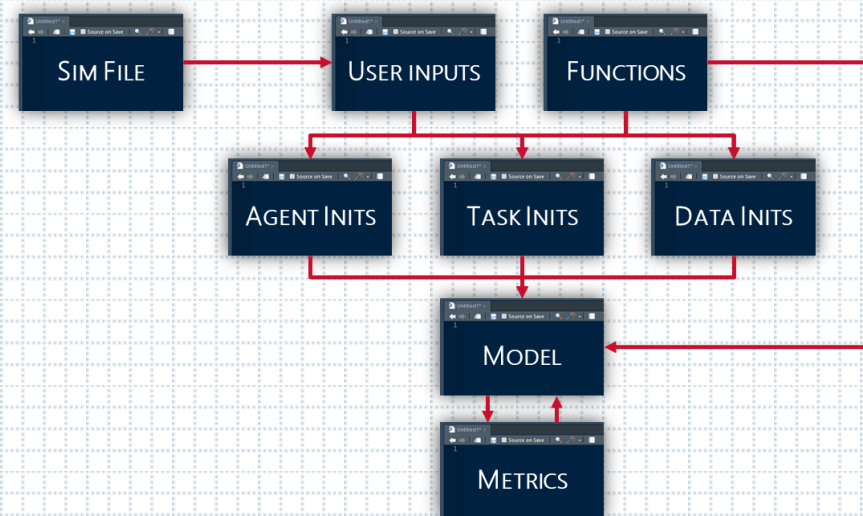


## Model file

```
MMConvergeModel.R
1 mmconverge = function(H, M, K) {
2   #####
3   # MODEL PARAMETERS #
4   #####
5   n = 10 # number of members in a team
6   niter = 500 # number of time points to let simulation run
7   lambda = .001 # learning rate; factor by which differences between an agent's
8
9   netType = 1 # parameter for creating different network configurations; 1 = Di
10  domSize = 100 # size of problem domain (must be integer value). NOTE: IN THEO
11  rndmDom = domSize*.5 # number of random points to select, between which value
12  domExptMean = 30 # size of expertise domain for each agent; higher numbers =
13  domExptSD = 5 # variability in expertise domain size for each agent; higher n
14  ldrndx = sample(1:n, 1) # randomly sample leader
15  confHeight = .1 # height of triangular function for changing confidence evalu
16  confWidth = 10 # width of triangular function for changing confidence evaluat
17  confChg = c(head((0:(confWidth*10/2))^(confHeight/(confWidth*10/2))), -1), ((c
18
19  #####
20  # INITIALIZE INPUT DATA AND MODEL OUTPUT #
21  #####
22  # Initialize the true problem function (TPF)
23  TPFObj = createTPF(rndmDom = rndmDom, domSize = domSize)
24  TPF = TPFObj$tpf
25  # Initialize agents' individual problem functions (IPF); IPF is a list of dat
```

# BUILDING A SIMULATION

- Source model function
  - source() → reads R code from an existing file into active environment
  - Useful method for creating “modular” code
    - » Create separate R files
      1. Organize each file by common purpose
    - » Use source() function in simulation file to load/run R code so it is available
  - **IMPORTANT!!** Order in which you source things matters
    - » Load dependencies for a file first



# BUILDING A SIMULATION

- Exercise: Work with partner to “modularize” our Dionne et al. model code and source it in the simulation file
  - Break model code into two separate files:
    - » MMConvergeFuncs.R → File containing all our custom functions
    - » MMConvergeModel.R → File containing both the initialization and model code
  - source() both files in simulation file

# BUILDING A SIMULATION

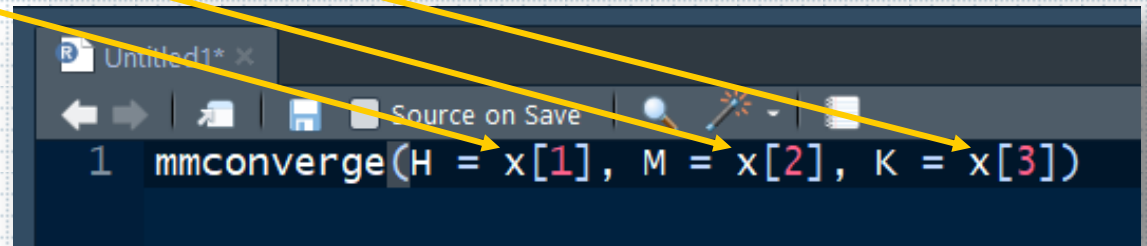
- Basic procedure for creating a simulation in R
  - ~~Transform model code into a custom function~~
  - ~~Make parameters to be manipulated arguments of function~~
    - » ~~Update model code to use arguments!~~
  - Create new simulation script
    - » Specify number of simulations to run
    - » Specify design matrix of parameter values for simulation runs
    - » Source model function
    - » Pass parameter values to model function and run until desired number of simulations completed
    - » Format and save output data



# BUILDING A SIMULATION

- Pass parameter values to model function and run until desired number of simulations completed
  - Basic logic = feed parameter values from each row of design matrix as arguments to the model function

H	M	K
0	0	2
0	0	3
...	...	...
0	1	2
0	1	3
...	...	...
1	0	2
1	0	3
...	...	...

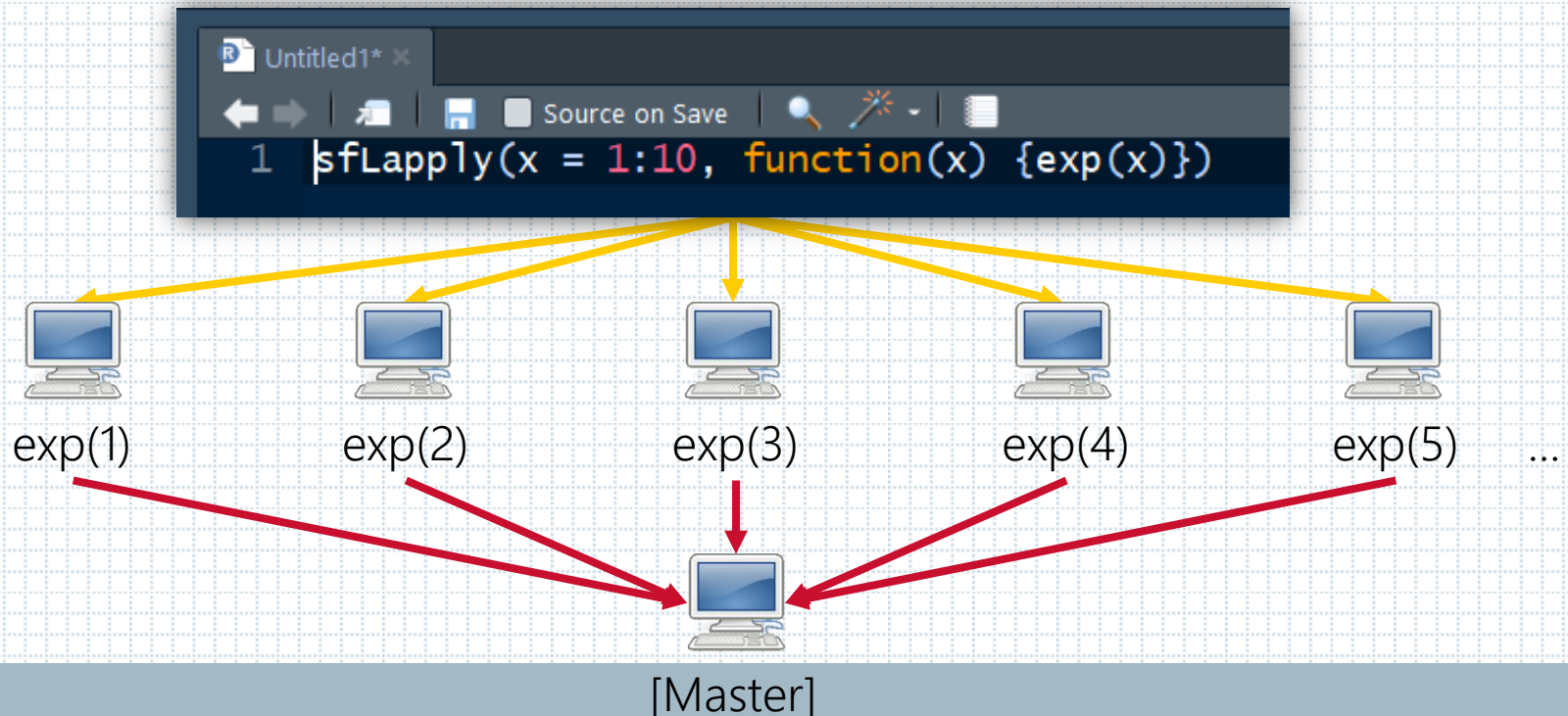


```
1 mmconverge(H = x[1], M = x[2], K = x[3])
```

The screenshot shows the R Studio interface with a code editor window titled 'Untitled1\*'. The code line is `1 mmconverge(H = x[1], M = x[2], K = x[3])`. Three yellow arrows originate from the first three rows of the design matrix table and point to the arguments in the function call: the first row (0, 0, 2) points to `x[1]`, the second row (0, 0, 3) points to `x[2]`, and the third row (..., ..., ...) points to `x[3]`.

# BUILDING A SIMULATION

- Pass parameter values to model function and run until desired number of simulations completed
  - Use family of `sfApply()` functions to run simulation in parallel
    - » Similar to base `apply()` functions, except that each computation is sent to and performed on separate cores



# BUILDING A SIMULATION

- Pass parameter values to model function and run until desired number of simulations completed
  - By default, sfApply family does not use load balancing
    - » In many cases, load balancing gains are minimal...
    - » Most beneficial when:
      1. Using distributed processing across cores with different speeds
      2. Speed of runs differs notably across parameter configurations
  - To use load balancing, we need to do two things:
    - » Turn design matrix into a list → each element = data frame of parameter values for single run
    - » Use sfClusterApplyLB() function
    - » **IMPORTANT!!** Don't forget to assign sfClusterApplyLB() to an object
      1. If don't assign, results of simulation will be output to R console and can't be saved!

# BUILDING A SIMULATION

- Pass parameter values to model function and run until desired number of simulations completed
  - Let's set it up and run a small simulation in R!

# BUILDING A SIMULATION

- Basic procedure for creating a simulation in R
  - ~~Transform model code into a custom function~~
  - ~~Make parameters to be manipulated arguments of function~~
    - » ~~Update model code to use arguments!~~
  - Create new simulation script
    - » Specify number of simulations to run
    - » Specify design matrix of parameter values for simulation runs
    - » Source model function
    - » Pass parameter values to model function and run until desired number of simulations completed
    - » Format and save output data

# BUILDING A SIMULATION

- Format and save data
  - `sfClusterApplyLB()` returns a list when complete
    - » Each list element = data saved from one simulation run
    - » Structure of data within each list element will correspond to what was returned by model function
  - Typically will want to do some data wrangling to combine data from each run into singular data object
    - » Nature of these operations will depend on what your data looks like...
    - » ...but we can practice with some results to get some basic principles

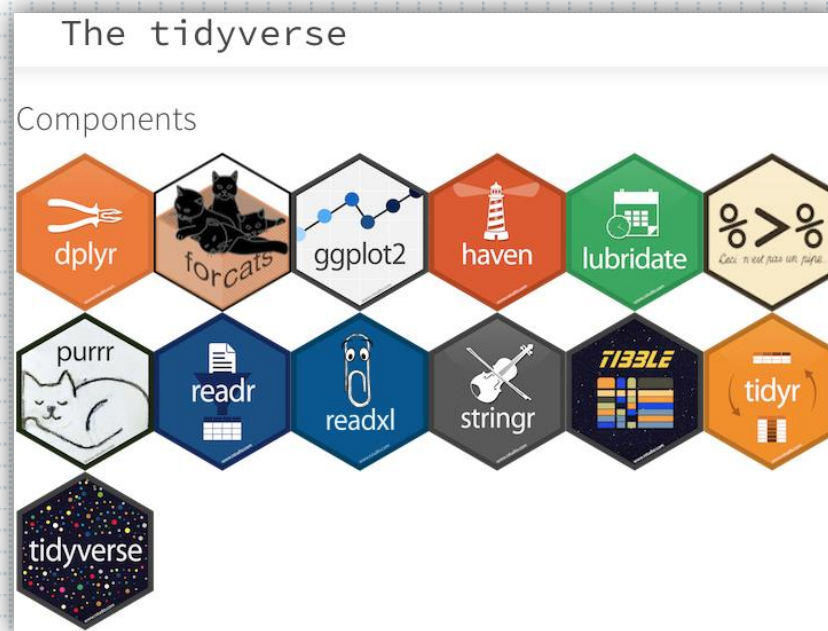


# BUILDING A SIMULATION

- Format and save data
  - **Exercise:** With a partner, examine the following:
    - » Load into R:
      1. conds.Rdata → design matrix
      2. condsDat.Rdata → raw simulation data
    - » How is condsDat structured?
      1. Hint: Take a look at what is returned from the overall model function in MMConvergeModel.R
    - » How would you extract the mmStats data.frame for the first team? For the second team? The 631<sup>st</sup> team?
      1. How is the mmStats data frame structured? What does it contain?

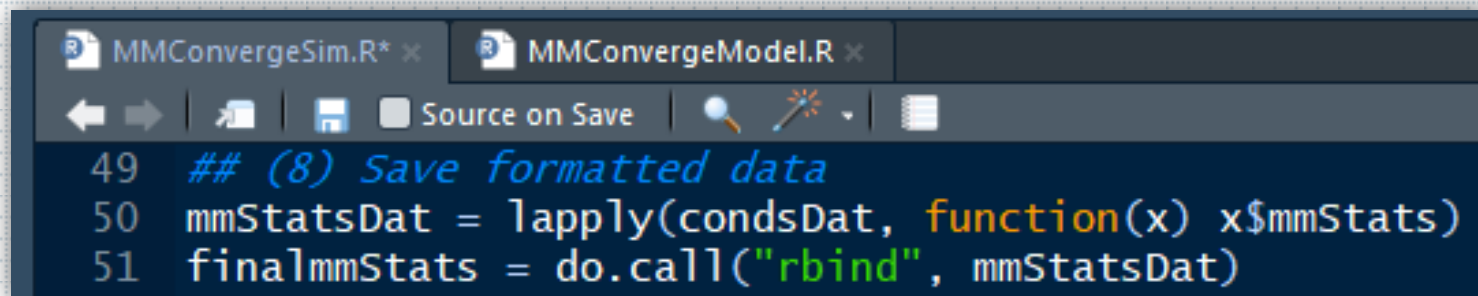
# BUILDING A SIMULATION

- Format and save data
  - But I don't want just one team at a time...I want them all!
    - » How do you extract the mmStats for all teams and combine them into a single data.frame?
  - Starts to tread into data wrangling & munging...beyond scope of this tutorial
    - » tidyverse = set of packages for importing, tidying, exploring, and communicating data
    - » Can pretty much do all this stuff in base R...so it's just preference



# BUILDING A SIMULATION

- Format and save data
  - How do you extract the mmStats for all teams and combine them into a single data frame?
    - » Combine lapply() and do.call("rbind")



The screenshot shows an R Studio interface with two open files: MMConvergeSim.R\* and MMConvergeModel.R. The script editor displays the following R code:

```
49 ## (8) Save formatted data
50 mmStatsDat = lapply(condsDat, function(x) x$mmStats)
51 finalmmStats = do.call("rbind", mmStatsDat)
```

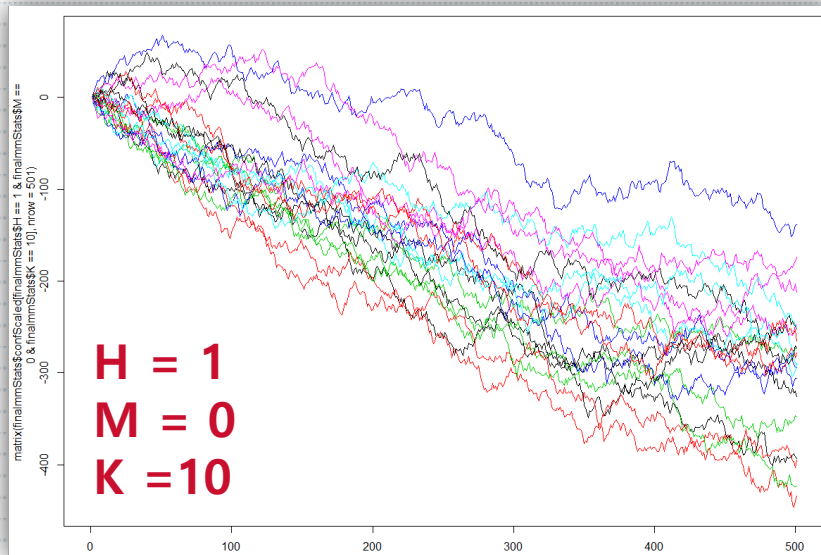
# BUILDING A SIMULATION

- Format and save data
  - **Exercise:** With a partner, create a single data frame for all the mmStats data that contains the following:
    - » New column called “accScaled” → accuracy relative to time 1
    - » New column called “confScaled” → confidence relative to time 1
      1. Hint: First extract all the mmStats into a list, then use lapply() function to operate on each mmStats data frame
    - » Columns containing the parameter values for each team’s run
      1. Hint: conds contains the parameter values for all 720 simulations, but the combined mmStats data frames contains more than 720 rows...
        - A. How many times does each row in conds need to be replicated?
    - » Column containing “teamID” that contains a unique identifier for each team’s data

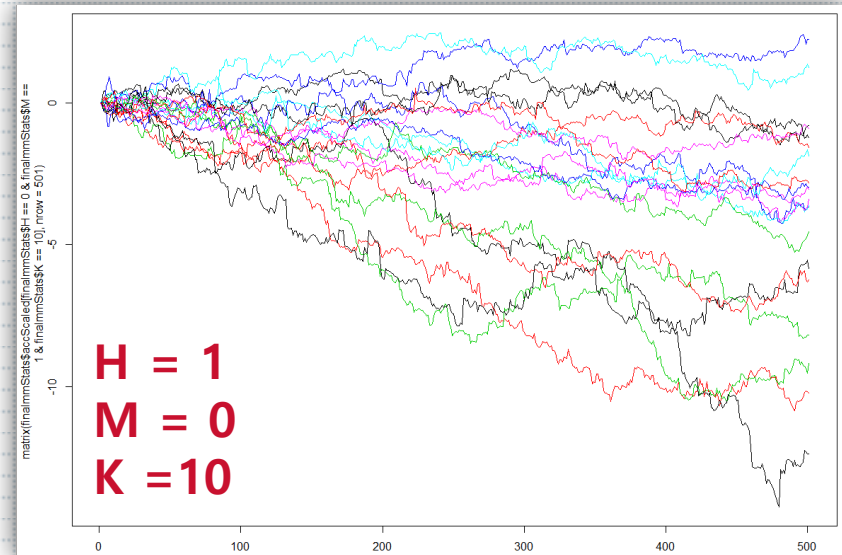
# BUILDING A SIMULATION

- Analyzing/visualizing data goes beyond scope of this tutorial...
- ...but is key to communicating, exploring, and generating model insights
  - *Patterns* often more helpful/informative than *parameters*
  - Inferential stats using simulation data often of secondary interest

Compare to Fig 4 in Dionne et al



Compare to Fig 6 in Dionne et al



# BUILDING A SIMULATION

- Takeaways from afternoon session
  - Models are *tools*, simulations are *creations*
    - » Models are jumping off points for understanding the world in a more precise and robust manner
  - Simulation results are sensitive to parameter choices...but that's okay
    - » Help to identify boundary conditions, probable parameter values, etc.
  - Have a purpose...
    - » Like experiments, simulations are conducted to help you learn
    - » Design of simulation will influence what insights you can draw
  - ...but don't feel too constrained
    - » Simulations can help identify new explanations for old findings just as easily as how old explanations account for new findings