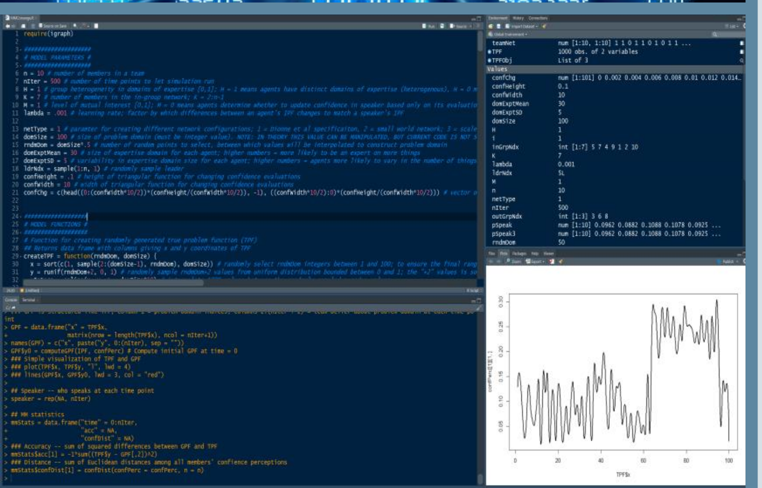


Introduction to Programming Computational Models in R

James A. Grand

University of Maryland

William Rand

NC State University

ARI Workshop

Fort Belvoir, VA

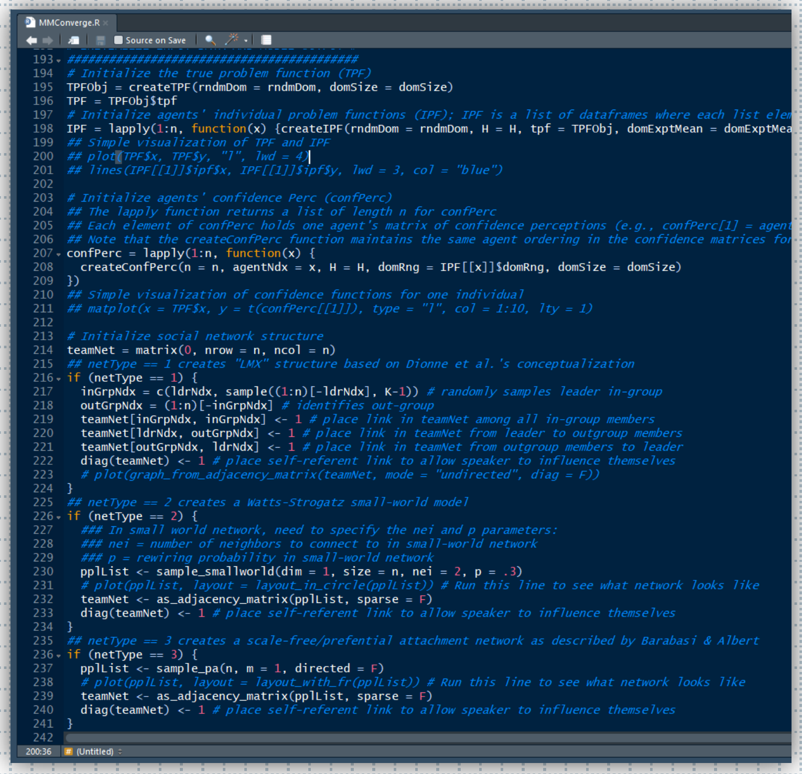
November 29, 2018

Objectives

- By the end of this session, you should be able to:
  - Identify and understand the use of control flow statements in the R programming language
  - Create your own custom functions in R
  - Understand how to create vectorized code in R
  - Identify and understand the structure computation
  - and organize computational model code into a coherent & organized structure
  - Develop initialization and process code for Dionne et al.'s (2010) computational model of mental model convergence

Programming Fundamentals in R

- Control flow statements
  - Programs executed sequentially from top to bottom
  - Control statements can be used to “break up” this flow



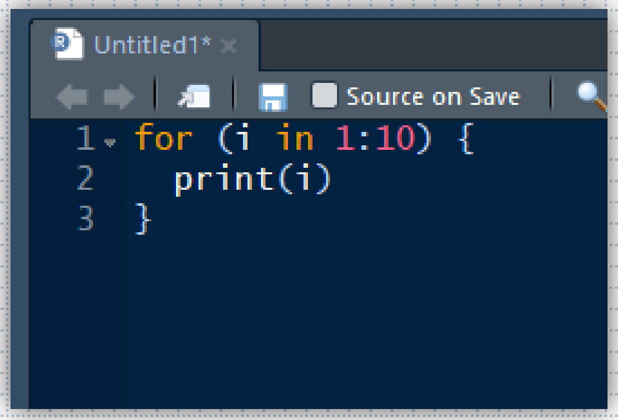
Programming Fundamentals in R

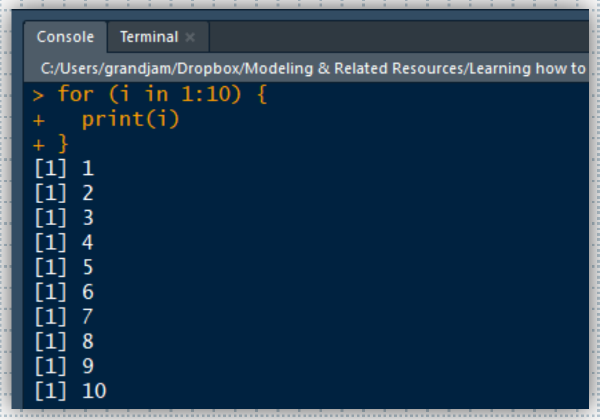
- Control flow statements
  - Branching – execute code conditionally
    - » If-else: Execute different code based on conditional statement
      1. Conditional checks – can be anything that returns a T or F ouput
         A. == or != → check if variable equals/does not equal value
         B. > or < → check if variable is greater/less than value
         C. >= or <= → check if variable is greater/less than or equal to value
      2. Else statement not required – nothing will happen if check fails

```r
if (x == 1) {
  ## Do this if pass the check
} else {
  # Do this if fail the check
}
```

Programming Fundamentals in R

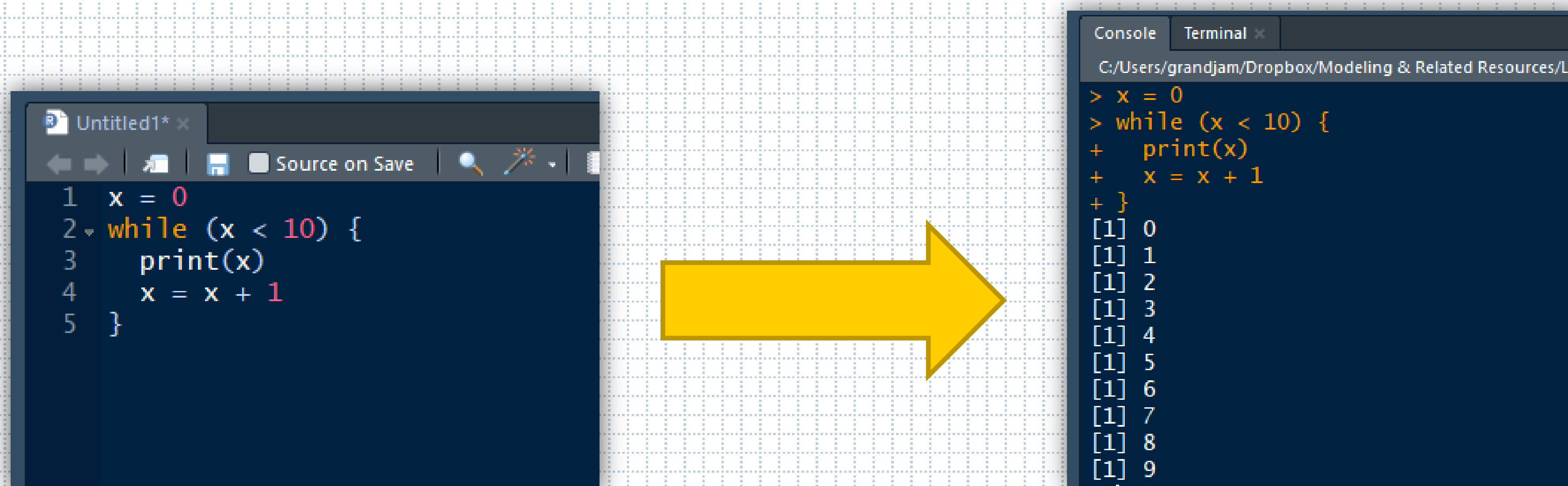
- Control flow statements
  - Repetition – execute same code multiple times
    - <u>For loops</u>: Execute code a specific number of times
      1. Set an "iterator" that will take on an a set of a priori determined values
      2. Iterator assigned a value
      3. Perform operations
      4. Once operations are complete, iterator assigned new value
      5. Repeat Steps 3-4 until iterator runs through all values





PROGRAMMING FUNDAMENTALS IN R

- Control flow statements
  - Repetition – execute same code multiple times
    - <u>While loops</u>: perform code until condition is satisfied
      1. Set a condition to check
      2. Perform operations until that condition is met
      3. Be careful not to get stuck in endless loop! (i.e., condition never satisfied)



Programming Fundamentals in R

- Control flow statements
  - Typical/exemplar uses in computational modeling

Control Statement	How is it Useful?
if else	• Run different code for different simulation conditions • Control when and which agents can act
for	• Run simulation for a fixed period of time • Control order of agent actions (who speaks first, etc.)
while	• Run simulation until all tasks are complete • Run simulation until all agents "die" (evolutionary models)

PROGRAMMING FUNDAMENTALS IN R

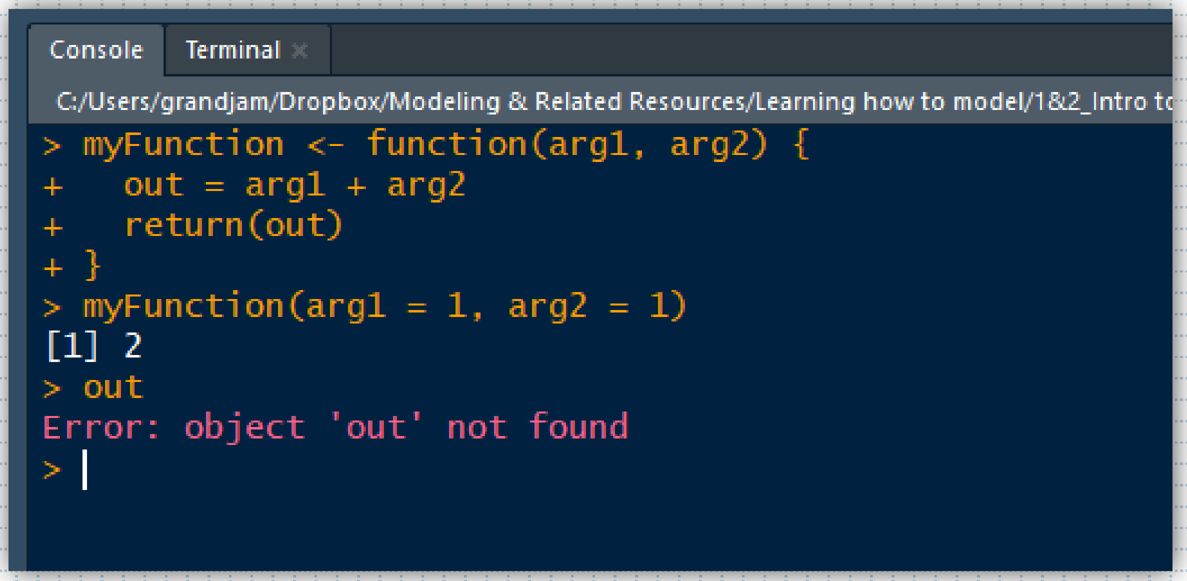
- Functions
  - Base R contains built-in functions; packages add functions
  - Can also write custom functions:
    - Assign function to an object (Don't forget this step!)
    - Specify arguments to function
    - Use arguments in function
    - Specify what to return from function (Don't forget this step either!)

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

```
> myFunction <- function(arg1, arg2) {
+   out = arg1 + arg2
+   return(out)
+ }
> myFunction(arg1 = 1, arg2 = 1)
[1] 2
>
```

Programming Fundamentals in R

- Functions
  - Important notes about functions
    - » Everything inside a function is local (i.e., only exist inside function)
      1. e.g., “out” can only be called/used inside “myFunction”



PROGRAMMING FUNDAMENTALS IN R

- Functions
  - Important notes about functions
    - If a variable is not defined in the arguments given to a function, R will try to find it in the global environment
    - If variable does not exist, function will not run

```
Console   Terminal ×
C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning how to model/18
> arg3 = 3
> myFunction <- function(arg1, arg2) {
+   out = arg1 + arg2 + arg3
+   return(out)
+ }
> myFunction(arg1 = 1, arg2 = 2)
[1] 6
>
```

```
Console   Terminal ×
C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning how to model/1&2_Intro to R Programming/
> myFunction <- function(arg1, arg2) {
+   out = arg1 + arg2 + arg3
+   return(out)
+ }
> myFunction(arg1 = 1, arg2 = 2)
Error in myFunction(arg1 = 1, arg2 = 2) : object 'arg3' not found
>
```

Programming Fundamentals in R

- Functions
  - Important notes about functions
    - » Make sure to **return** desired output from function – otherwise nothing will happen!

```
Untitled1*
Source on Save
1  myFunction <- function(arg1, arg2) {
2    out = arg1 + arg2 + arg3
3  }
```

No return statement specified

```
Console   Terminal
C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning how to mod
> myFunction <- function(arg1, arg2) {
+   out = arg1 + arg2
+ }
> myFunction(arg1 = 1, arg2 = 2)
>
```

```
Console   Terminal
C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning how to mode
> myFunction <- function(arg1, arg2) {
+   out = arg1 + arg2
+ }
> myFunction(arg1 = 1, arg2 = 2)
> out
Error: object 'out' not found
```

Programming Fundamentals in R

- Functions
  - Important notes about functions
    - Can only return **one** object from a function
    - To return multiple things, combine them together in the function

```
myFunction <- function(arg1, arg2) {
  out1 = arg1 + arg2
  out2 = arg1 * arg2
  return(out1, out2)
}
```

Trying to return multiple objects

```
> myFunction <- function(arg1, arg2) {
+   out1 = arg1 + arg2
+   out2 = arg1 * arg2
+   return(out1, out2)
+ }
> myFunction(arg1 = 1, arg2 = 2)
Error in return(out1, out2) : multi-argument returns are not permitted
>
```

```
myFunction <- function(arg1, arg2) {
  out1 = arg1 + arg2
  out2 = arg1 * arg2
  return(list(out1, out2))
}
```

Return multiple objects in a list

```
> myFunction <- function(arg1, arg2) {
+   out1 = arg1 + arg2
+   out2 = arg1 * arg2
+   return(list(out1, out2))
+ }
> myFunction(arg1 = 1, arg2 = 2)
[[1]]
[1] 3

[[2]]
[1] 2
```

PROGRAMMING FUNDAMENTALS IN R

- Functions
  - Typical/exemplar uses in computational modeling
    - » Any time you need to use the same code more than once – put it in a function!

Functions	How is it Useful?
function	• Initializing agents and environments • Specifying (complex) actions or computations of agents • Performing computations for output • Creating simulation that allows parameter manipulation

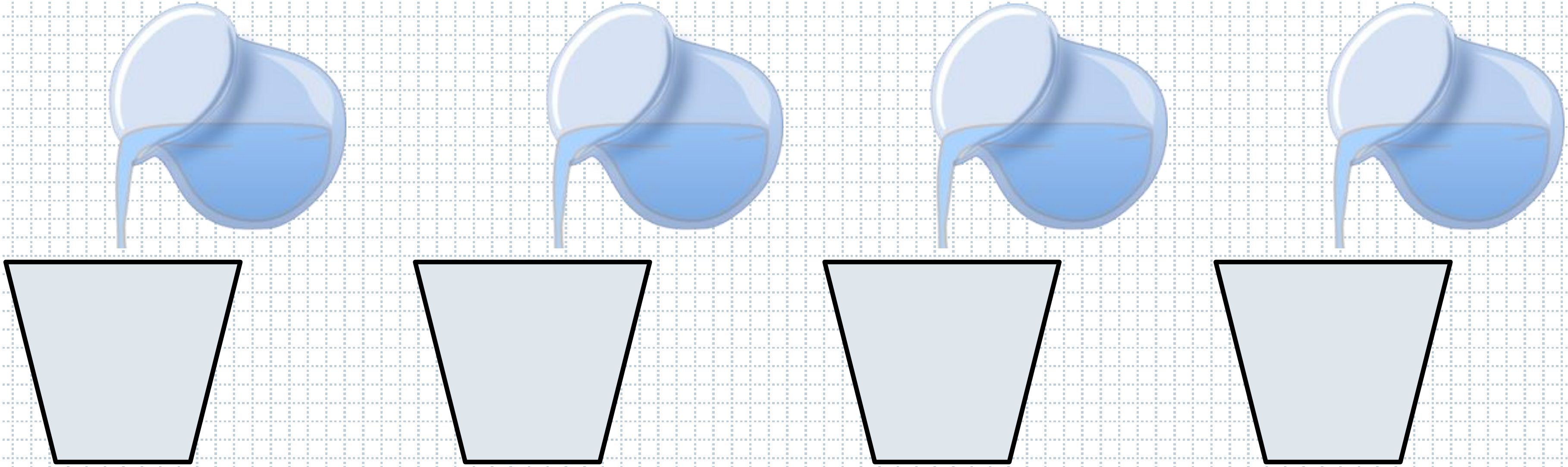
Programming Fundamentals in R

- Vectorization in R
  - R is a vectorized language
    - Operations can be conducted on entire vector rather than single item at a time
  - Means many operations can be performed *without* control statements
  - Code written in vectorized formats tends to run faster…
  - …but can be more difficult to read (until you get used to it)

  - Two types of helpful vectorization:
    - Apply statements
    - Conditional selections

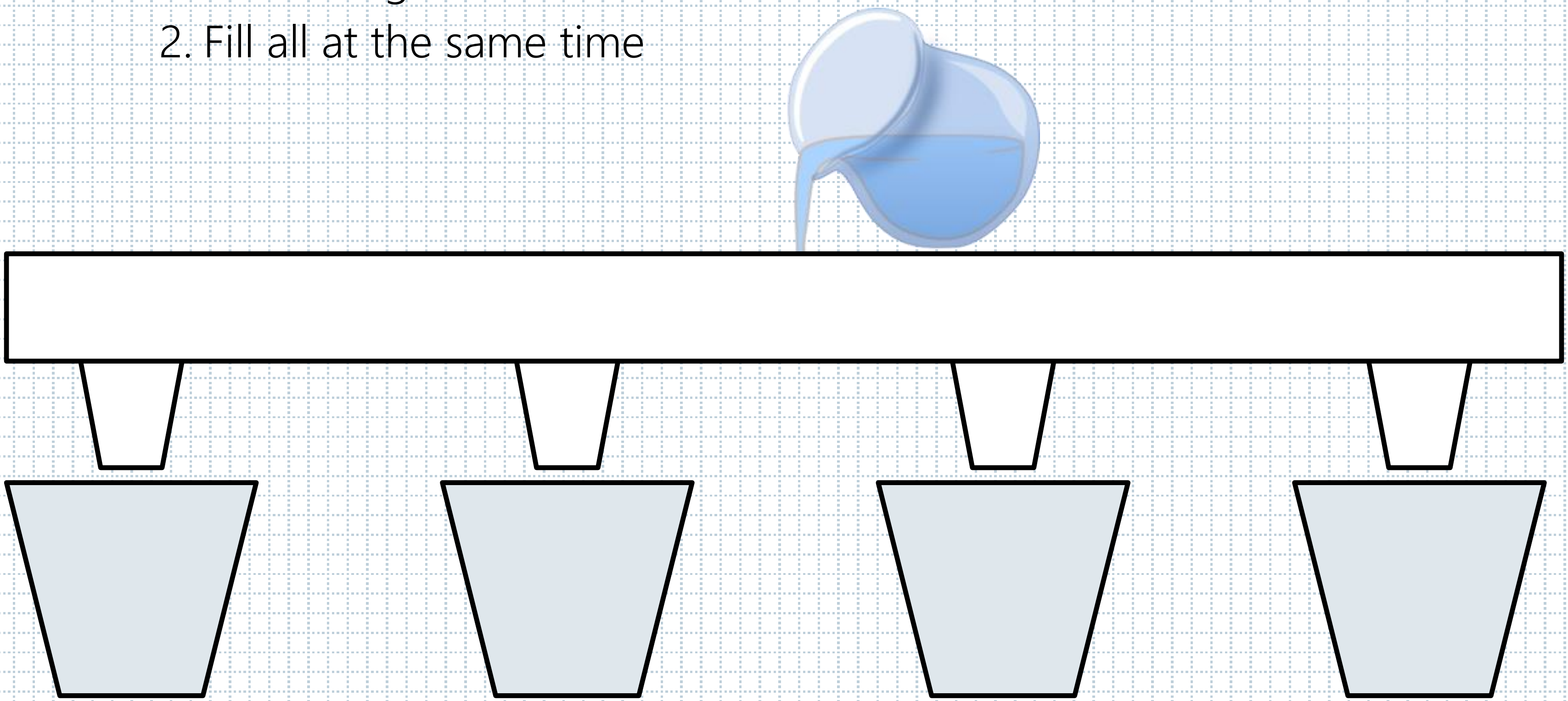
PROGRAMMING FUNDAMENTALS IN R

- Vectorization in R
  - Example: filling glasses of water
    - » For loop
      1. Select one glass
      2. Fill
      3. Move to next & repeat until all glasses filled



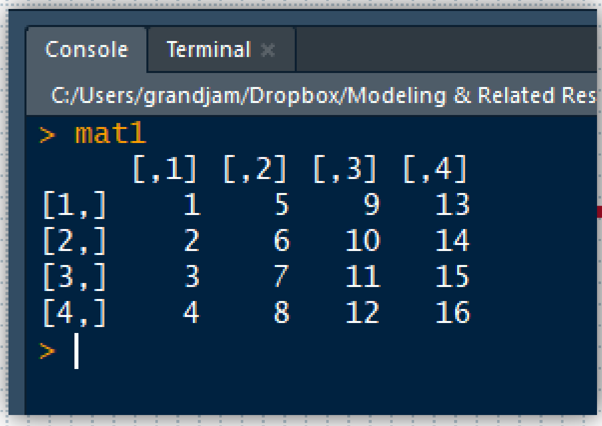
PROGRAMMING FUNDAMENTALS IN R

- Vectorization in R
  - Example: filling glasses of water
    - Vectorized
      1. Select all glasses
      2. Fill all at the same time

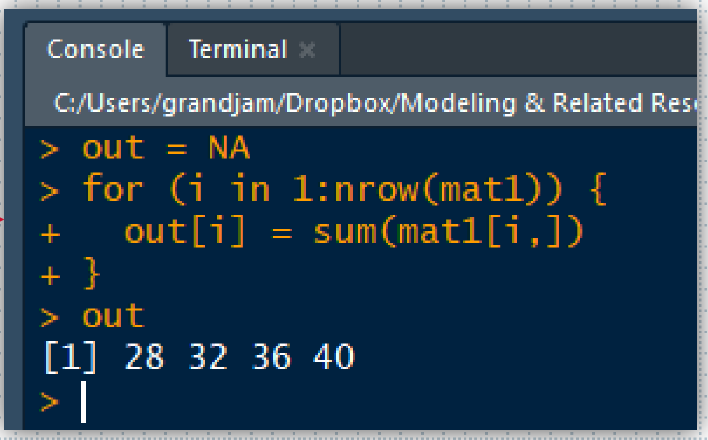


PROGRAMMING FUNDAMENTALS IN R

- Vectorization in R
  - Apply statements – vectorized version of for loops



Sum each row with for loop



Programming Fundamentals in R

- Vectorization in R
  - Apply statements – vectorized version of for loops
    - » apply() → apply a function over rows (1), columns (2), or cells (c(1,2))of an array

Console Terminal

C:/Users/grandjam/Dropbox/Modeling & Related Res

```
> mat1
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
>
```

Sum each row

Console Terminal

C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning hov

```
> apply(mat1, 1, function(x) {sum(x)})
[1] 28 32 36 40
>
```

Sum each column

Console Terminal

C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning how

```
> apply(mat1, 2, function(x) {sum(x)})
[1] 10 26 42 58
>
```

Multiply each cell by 10

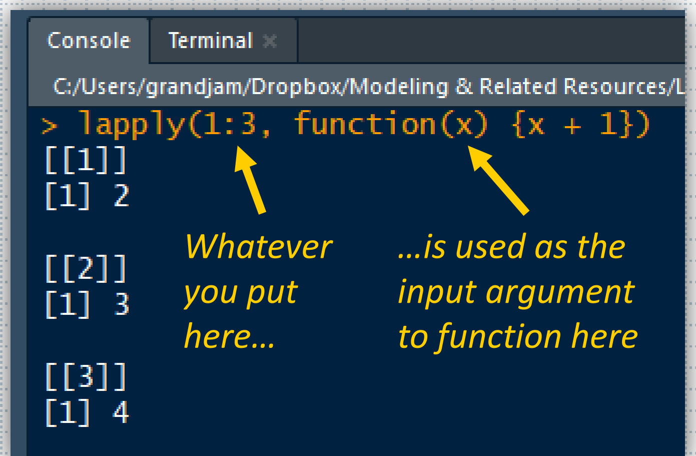
Console Terminal

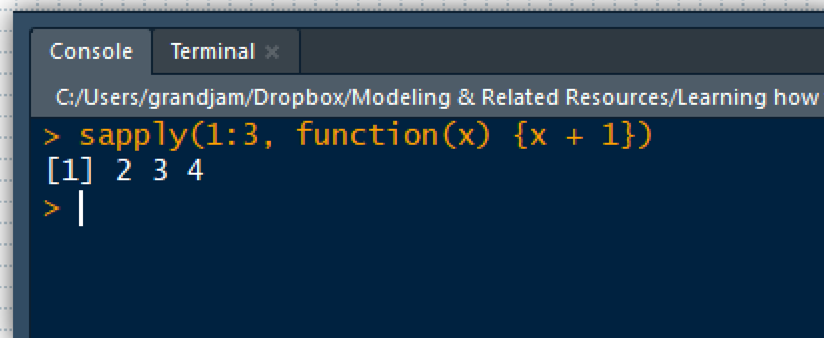
C:/Users/grandjam/Dropbox/Modeling & Related Resources/Learning how

```
> apply(mat1, c(1,2), function(x) {x*10})
     [,1] [,2] [,3] [,4]
[1,]   10   50   90  130
[2,]   20   60  100  140
[3,]   30   70  110  150
[4,]   40   80  120  160
>
```

Programming Fundamentals in R

- Vectorization in R
  - Apply statements – vectorized version of for loops
    - » lapply() → apply function over a vector and return a list
    - » sapply() → apply function over a vector and return as non-list (if possible)

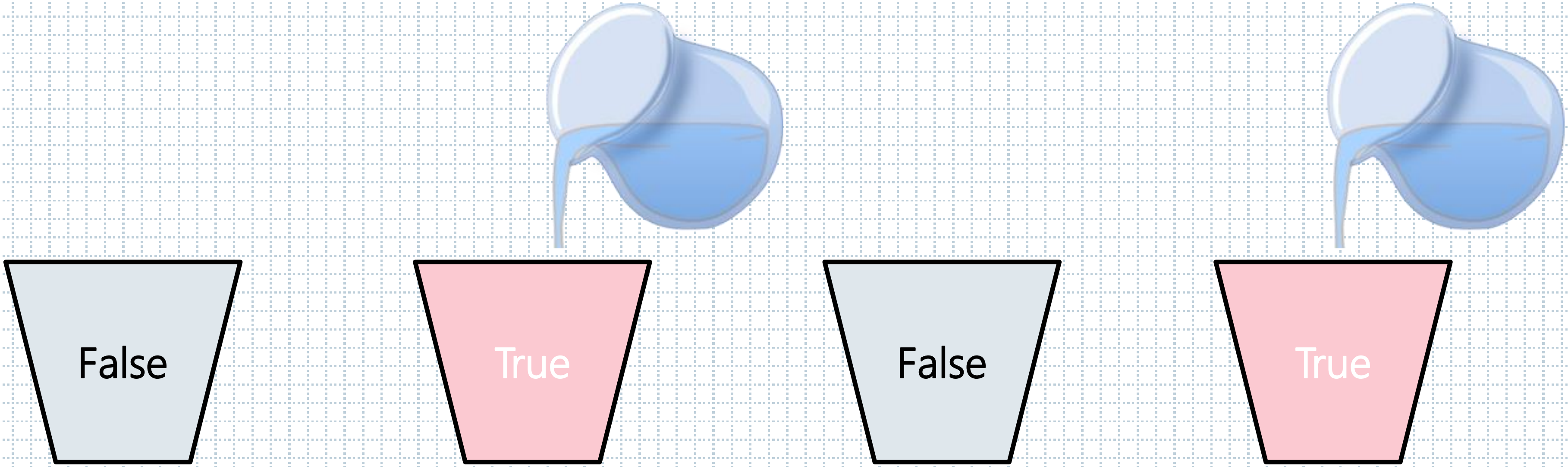




  - Function can be built-in or custom inside apply statement

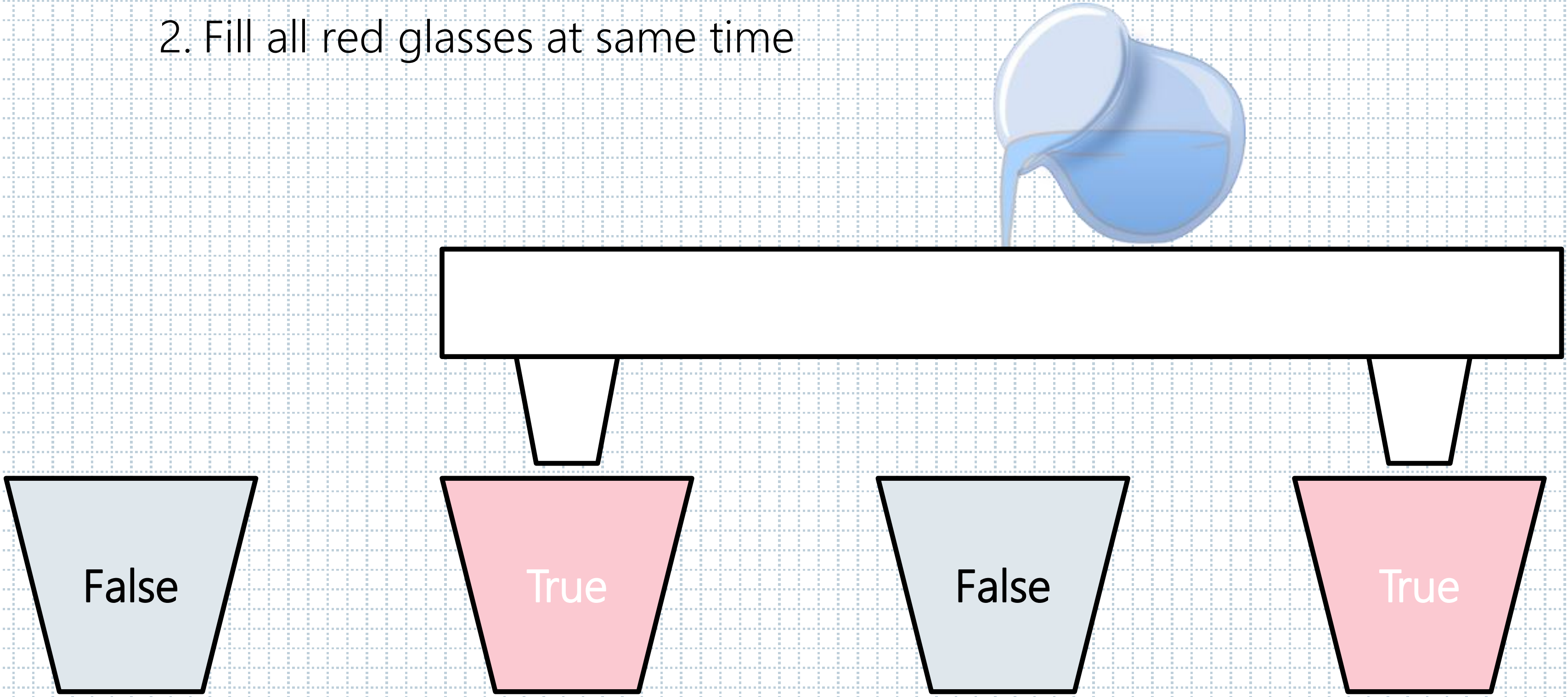
Programming Fundamentals in R

- Vectorization in R
  - Example: filling specific glasses of water
    - » If statement – fill red glasses
      1. Select first glass
      2. Check if red → if true fill, if false do nothing
      3. Move to next and repeat until all glasses checked



PROGRAMMING FUNDAMENTALS IN R

- Vectorization in R
  - Example: filling a glass of water
    - » Vectorized – fill red glasses
      1. Pick all red glasses
      2. Fill all red glasses at same time



Programming Fundamentals in R

- Vectorization in R
  - Conditional selections
    - » Select and work with only elements that meet a condition

Console Terminal

C:/Users/grandjam/Dropbox/Modeling & Related Res

```
> mat1
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> 
```

Replace values > 6 using conditional Selection

Console Terminal

C:/Users/grandjam/Dropbox/Modeling & Related Resou

```
> mat1[mat1 > 6] <- 99
> mat1
     [,1] [,2] [,3] [,4]
[1,]    1    5   99   99
[2,]    2    6   99   99
[3,]    3   99   99   99
[4,]    4   99   99   99
> 
```

Replace values > 6 using if statements & for loop

Console Terminal

C:/Users/grandjam/Dropbox/Modeling & Related Resou

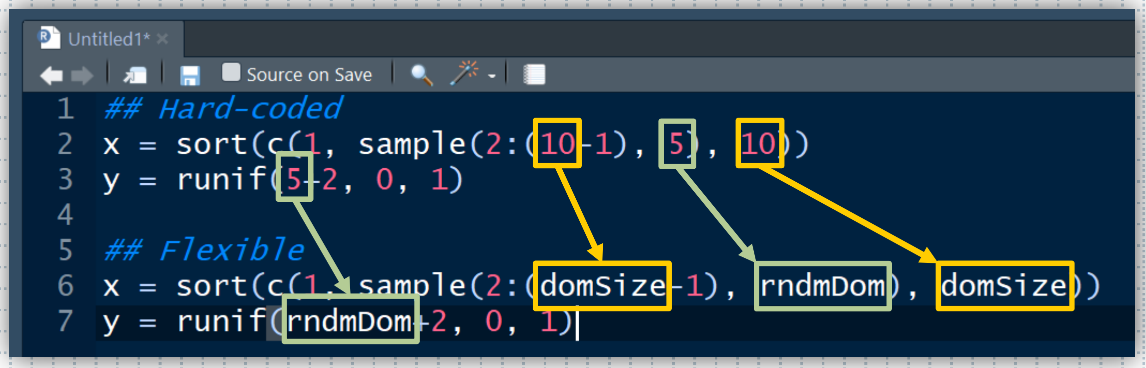
```
> for (i in 1:nrow(mat1)) {
+   for (j in 1:ncol(mat1)) {
+     if (mat1[i,j] > 6) {
+       mat1[i,j] <- 99
+     }
+   }
+ }
> mat1
     [,1] [,2] [,3] [,4]
[1,]    1    5   99   99
[2,]    2    6   99   99
[3,]    3   99   99   99
[4,]    4   99   99   99
> 
```

Programming Fundamentals in R

- Tips & tricks to make you a better coder
  - Break the problem down
    - » What needs to be accomplished? What should the end result look like?
    - » What data, variables, etc. do I need to accomplish the task?
  - Fail fast by testing often
    - » Run new code frequently!
      1. Confirm that what you think should happen does happen
    - » Can use fake inputs to avoid running all code…but test everything too
    - » Don't ignore error or warning messages in R!
      1. Error messages → computations don't run, processes terminate
      2. Warning messages → computations run, processes <u>don't</u> terminate
  - Comment <u>EVERYTHING</u>
    - » Use # at beginning of line to add comment
    - » Code doesn't run slower, you will know what is happening, others will know what is happening

PROGRAMMING FUNDAMENTALS IN R

- Tips & tricks to make you a better coder
  - Make it work first, worry about efficiency/flexibility later
    - Writing efficient/flexible code in R takes practice…don't let perfect be the enemy of good
    - Inefficient code takes longer to run—but it works!
  - Try to do as little "hard coding" as possible
    - Hard coding = using numeric values rather than variables in code
    - Assign to variable if:
      1. You will end up using same variable in many different places in code
      2. You think you may want to change value of variable



Programming Fundamentals in R

Any questions at this point?

Ready to start modeling!?

STRUCTURE OF COMPUTATIONAL MODEL CODE

Untitled1*

Source on Save

```
###################
# MODEL PARAMETERS #
###################


##################
# MODEL FUNCTIONS #
##################


###################################
# INITIALIZATION (INPUTS & OUTPUTS) #
###################################


#########
# MODEL #
#########


##############
# DATA OUTPUT #
##############
```

VARIABLES THAT WILL BE MANIPULATED OR REMAIN CONSTANT

CUSTOM FUNCTIONS USED TO PERFORM CALCULATIONS, PROCEDURES, ETC.

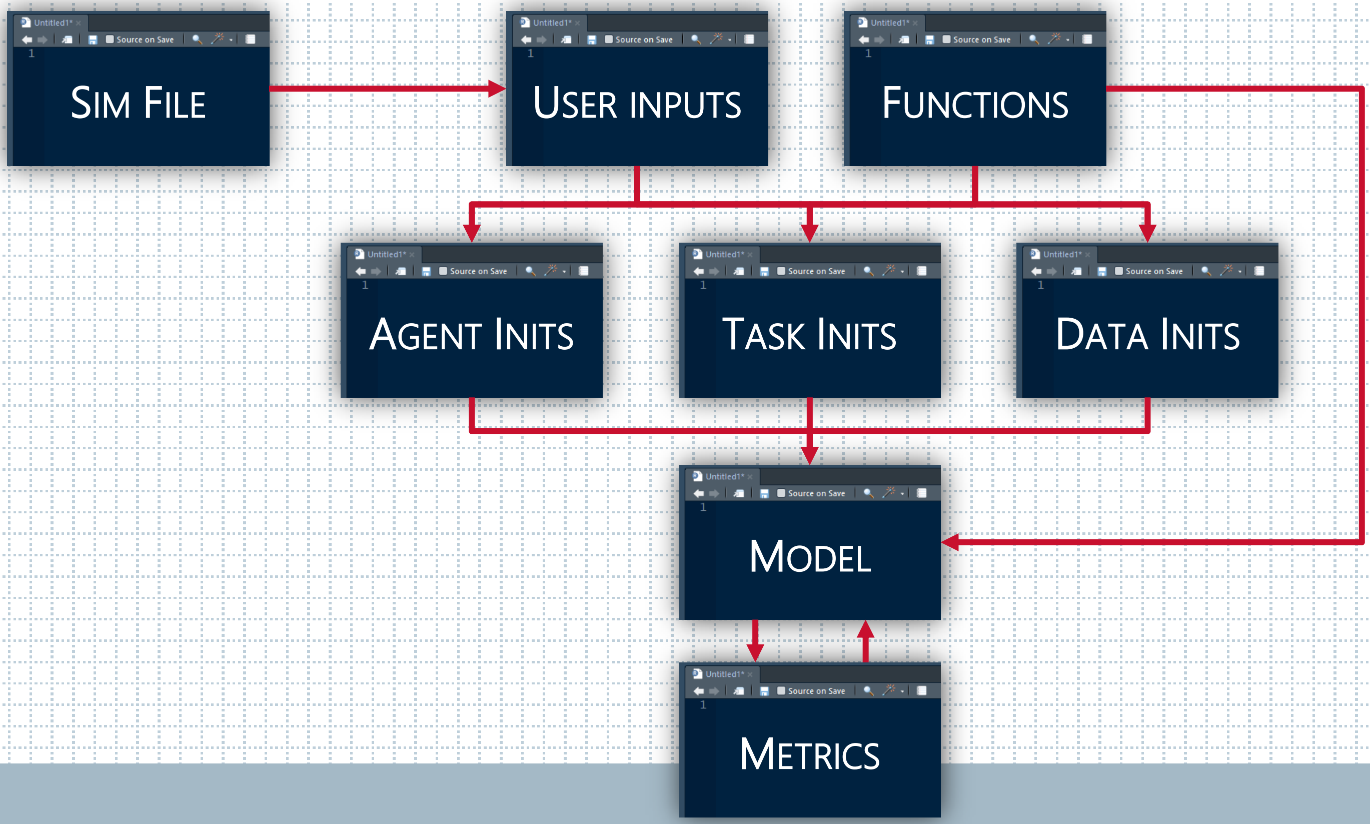
INITIALIZE AGENTS, ENVIRONMENT, AND OBJECTS FOR STORING OUTPUT

STEPS, PROCEDURES, AND ACTIONS CARRIED OUT IN THE MODEL

DATA AGGREGATION, TRANSFORMATION, ETC. AND SAVING OUTPUT

Structure of Computational Model Code

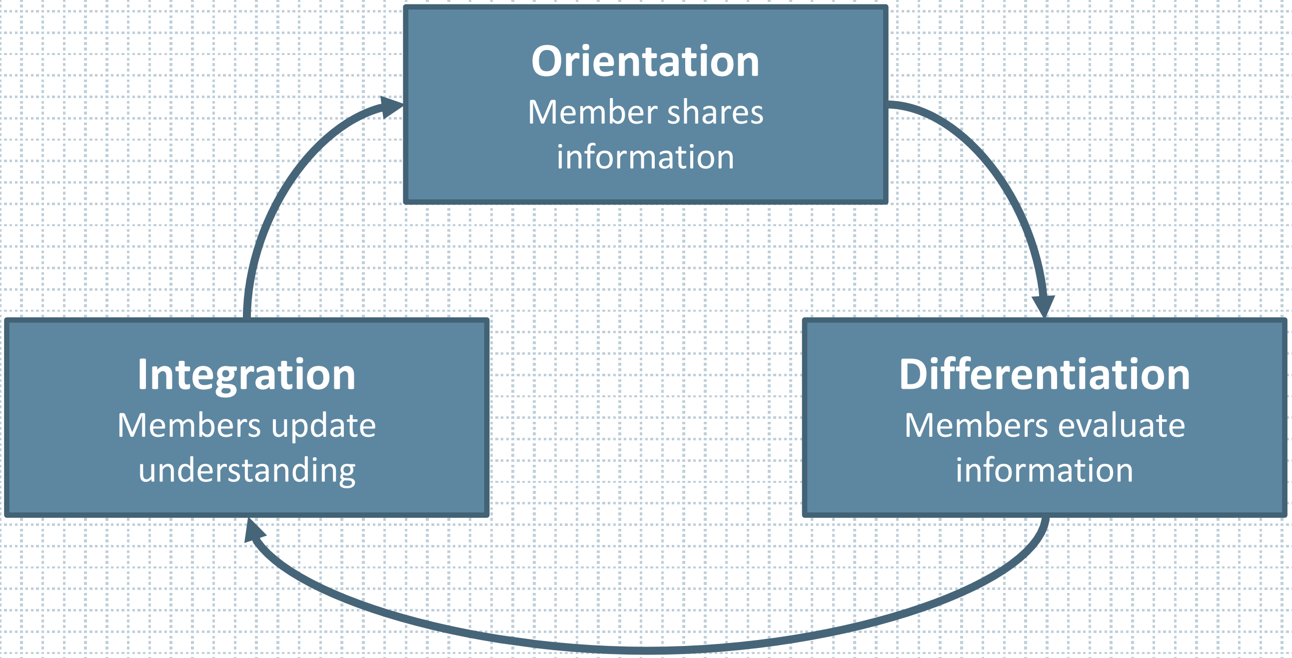
- For more complex models, can be convenient to make code more "modular"
  - Example: Team effectiveness model (ARI)
  - More on this tomorrow afternoon…



BUILDING A MODEL

- 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:



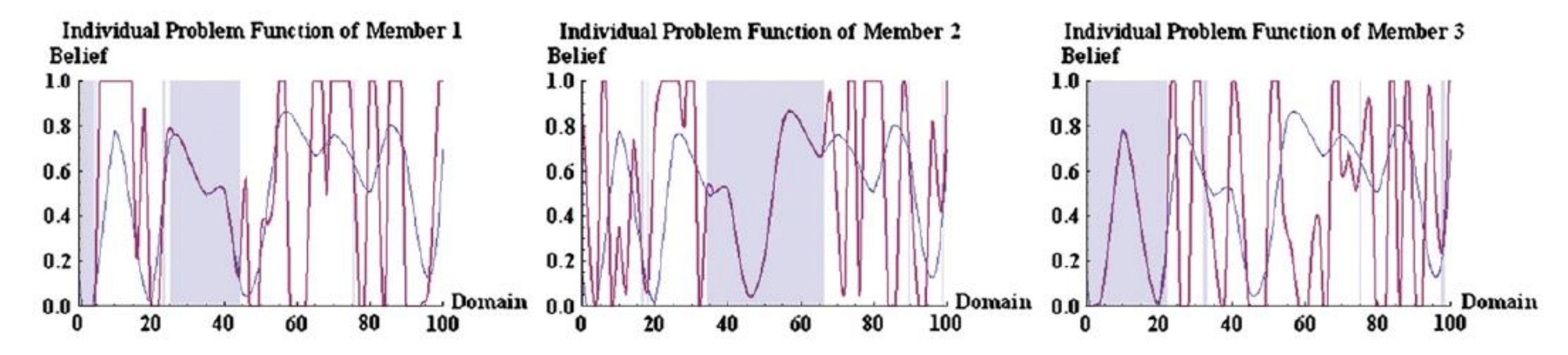
Building a Model

- Will NOT code entire model
  - Point of reference – took me ~25hrs to understand and code model…and it still doesn't perfectly replicate Dionne et al.'s results!
  - We will work on:
    - Some initialization steps
    - Some process steps
  - My full model code is available on Github repository

- *Goals for this exercise:*
  - Step through thought process → How to go from words/ideas to code?
  - Build some familiarity with programming in R

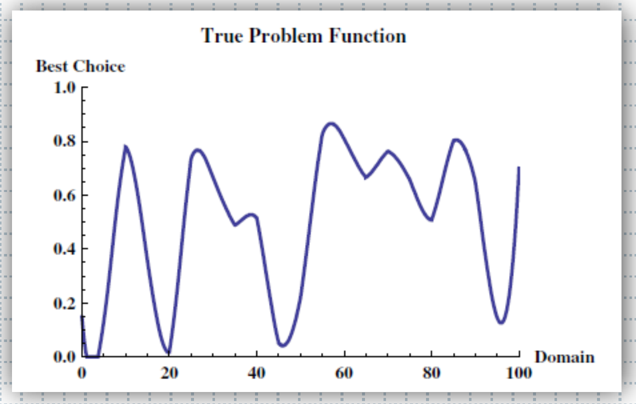
BUILDING A MODEL

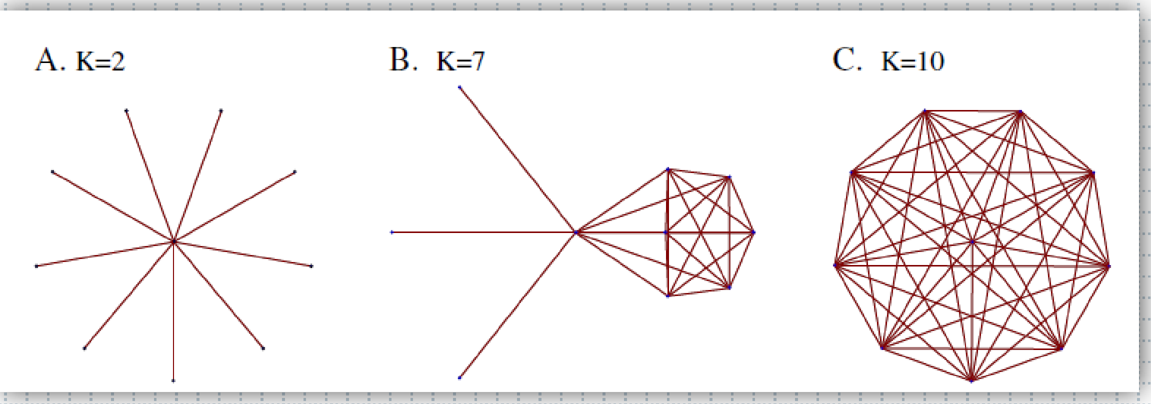
- Agent characteristics?
  - Variables
    - » Individual problem functions (time-varying)
    - » Confidence perceptions (time-varying)
    - » Domain of expertise (static)
    - » Level of mutual interest (M)* – *manipulated*
  - States
    - » Formal leader/follower (static)
    - » Speaking/not speaking (time-varying)



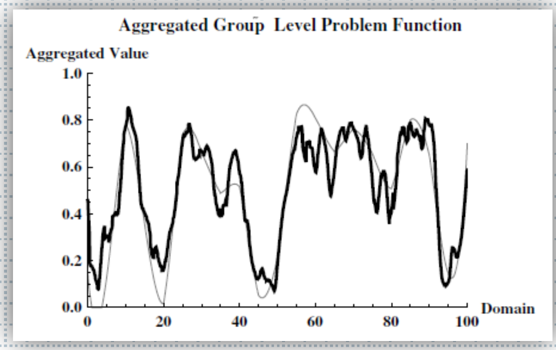
BUILDING A MODEL

- Environment characteristics?
  - True problem function
  - Size of team
  - Duration of teamwork
  - Group heterogeneity in expertise domains (H)* – *manipulated*
  - Number of members in in-group network (K) – *manipulated*
  - Team network

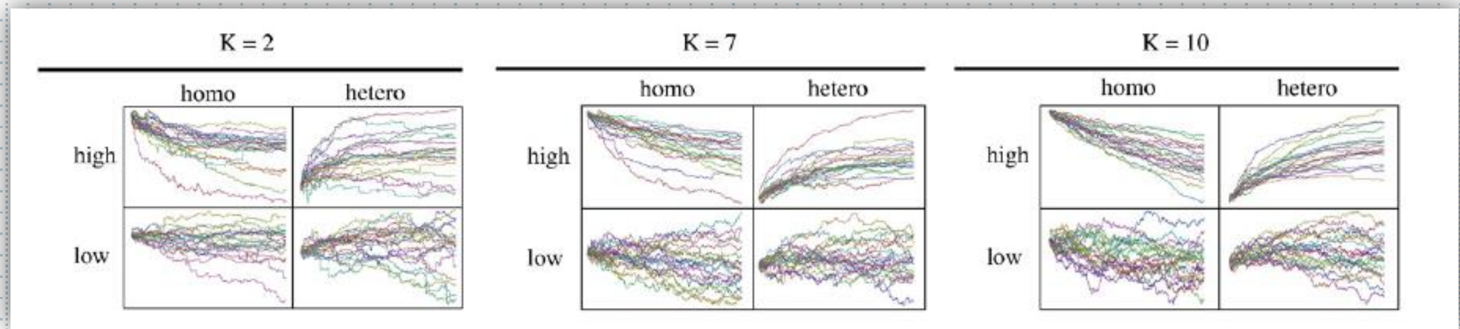


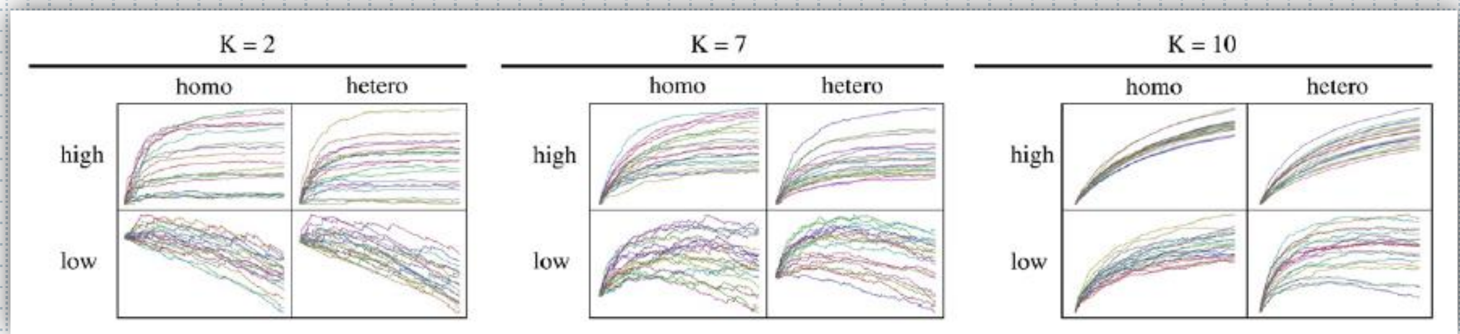


BUILDING A MODEL



- Output variables?
  - Group problem function (time-varying)
  - Mental model accuracy (time-varying)
  - Mental model convergence (time-varying)



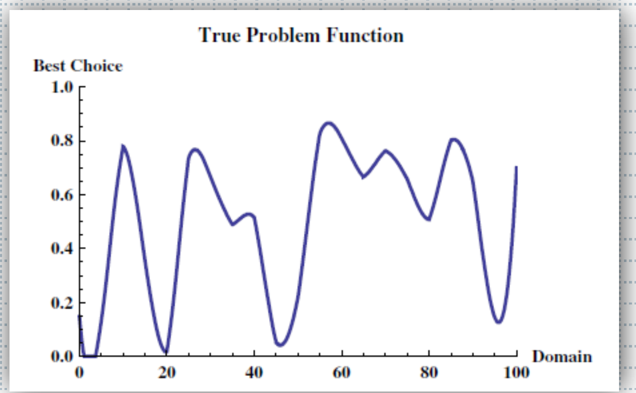


Building a Model

- Pseudocode?
  - Initialization
    - » Create true problem function
    - » Create individual problem functions
    - » Create confidence perceptions
    - » Create team network
  - Process
    - » Step 1: Select speaker as a function of self-confidence
    - » Step 2: Speaker selects topic to speak about as a function of self-confidence and shares with members
    - » Step 3: Connected members hear and form evaluation of opinion
    - » Step 4: Connected members update confidence perceptions and individual problem functions
    - » Repeat Steps 1-4 until time limit is reached

Building a Model

- Initializing true problem function (TPF)
  - What do we need to create?
    - » Values for x, values for y
      1. x = area of problem domain
      2. y = optimal choice for problem domain @ x
  - How could we do this?
    - » Simplest solution?
      1. Randomly sample y values for each x
    - » What did Dionne et al. do?
      1. Interpolation procedure (p. 1039)



Our model represents a development process of a team working on a problem representation task in a one-dimensional continuous problem domain between 0 and 100 (arbitrarily chosen). A true problem function (TPF) is constructed by assigning random numbers between 0 and 1 to several sample points in the problem domain and then interpolating between those random sample values (example shown in Fig. 1, top). At each point in the problem domain, the value of the TPF represents the best choice for that particular aspect of the problem. The objective of a team is to collectively estimate the shape of the TPF as accurately as possible.

Building a Model

- Initializing true problem function (TPF)
  - Interpolation procedure seems innocuous…but carries hidden assumption!
    - y-values implicitly correlated
      1. Using my interpolation procedure → Lag-1 r = .99, Lag-10 r = .62
      2. Knowledge of y @ x = 1 implies knowledge about y @ x = 2
  - Learning point!
    - Everything you code will represent an assumption about the "world"
    - Sometimes obvious/purposeful…other times "hidden"/unintended
      1. Sensitivity analyses = testing robustness of model to assumptions
    - Model-building = exercise in precision!
  - Exercise: Work with partner to write pseudocode/steps for initializing TPF

Building a Model

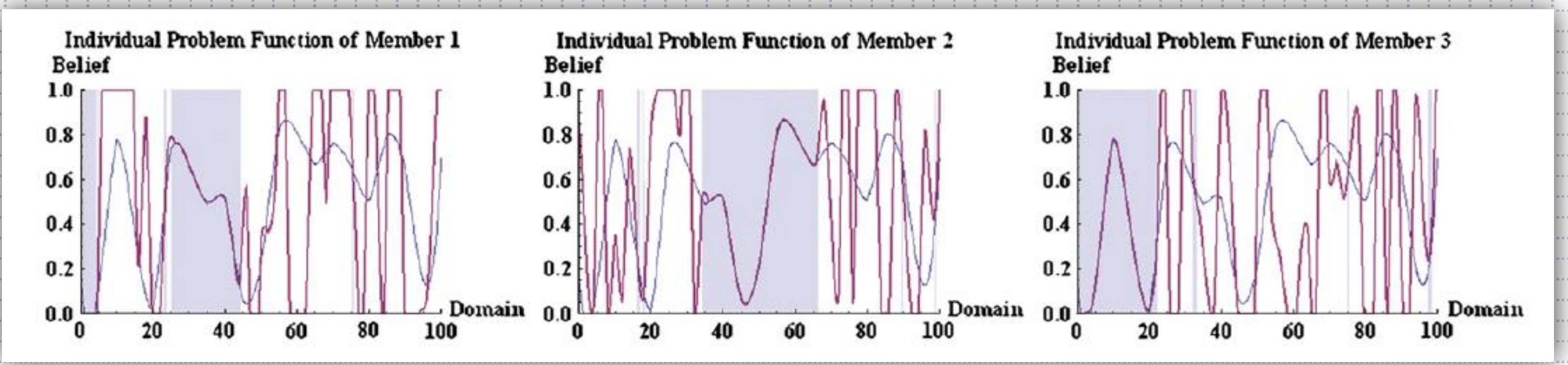
- Initializing true problem function (TPF)
  - Pseudocode
    - Select random # of x-values from problem domain
    - Randomly sample same # of y-values between 0-1
    - Interpolate between sampled x-y coordinates
    - Ensure all y values fall between 0 and 1

Building a Model

- Initializing true problem function (TPF)
  - Exercise: Work with partner to code TPF initialization
    - Create x-y values using Dionne et al. interpolation procedure
      1. Hint: always need y value for x = 1 and x = 100…
      2. Hint: try spline() for interpolation (see ?spline for help)
    - Store x-y values in data frame with two columns labeled "x" and "y"
    - Advanced: Turn into custom function that allows you to:
      1. Create TPF for different sized problem-domains (e.g., x > or < 100)
      2. Change number of x-values to select for interpolation
         A. Hint: size of problem domain & # of x-values to select should be inputs/arguments to function

Building a Model

- Initializing individual problem function (IPF)
  - What do we need to create?
    - x-y coordinates for *each* team member
    - Individual expertise (H) – more accurate at certain places in problem domain
      1. When expertise present → determine area of expertise (x-values), what should IPF (y-values) look like inside & outside these areas
      2. When expertise absent → decide how "far off" individuals should be from TPF



Building a Model

- Initializing individual problem function (IPF)
  - How could we do this?
    - » Dionne et al. (p. 1040) not very detailed…so we'll do our best
  - <u>Exercise</u>:
    - » Work with partner to write out pseudocode/programming steps for:
      1. Creating *<u>single agent</u>* with expertise (H = 1)
      2. Creating *<u>single agent</u>* without expertise (H = 0)
    - » Learning point!/Hint!
      1. In cases where you need to do the same thing multiple times:
         A. Figure out how to do procedure once
         B. Figure out easiest method for replicating (usually combination of writing custom function and using an apply statement)

Building a Model

- Initializing individual problem function (IPF)
  - Pseudocode for expertise present ($H = 1$)
    - » Identify area of expertise <u>size</u> for agent (width ≤ 50)
    - » Identify area of expertise <u>range</u> for agent (continuous range)
    - » Create IPF inside area of expertise
      1. x for IPF = TPF, y for IPF = TPF
    - » Create IPF outside area of expertise
      1. x for IPF = TPF, y for IPF ≠ TPF
    - » Store x- & y-values in two-column data frame

  - Pseudocode for expertise absent ($H = 0$)
    - » Create IPF that is "equally inaccurate" across total problem domain
      1. x for IPF = TPF, y for IPF ≠ TPF
    - » Store x- & y-values in two-column data frame

Building a Model

- Initializing individual problem function (IPF)
  - <u>Exercise</u>: Work with partner to code expertise *present* for *<u>single agent</u>*
    - » Identify expertise <u>size</u> for agent (width ≤ 50)
      1. Hint: store as variable in R
    - » Identify expertise <u>range</u> for agent (continuous)
      1. Hint: want to make this random…play around with sample() function (?sample)
      2. Hint: range can't "loop" around end of problem domain…how to ensure that range won't be cut off?
    - » Create IPF inside area of expertise
    - » Create IPF outside area of expertise
      1. Hint: Try to reuse the TPF function from before…
         A. Learning point! Reuse code whenever possible
      2. Hint: Need to identify where TPF and IPF overlap…play around with findInterval() function (?findInterval)
         A. Learning point! Oftentimes easier to *overwrite* part of a vector/matrix than to build it piece-by-piece
    - » Store x- & y-values in two-column data frame

BUILDING A MODEL

- Initializing individual problem function (IPF)
  - **Exercise**: Work with partner to code expertise *absent* for *single agent*
    - » Create IPF that is "equally inaccurate" across total problem domain
      1. Hint: how could you add random noise to existing TPF?
         A. Take a look at rnorm() and runif() for random sampling…
    - » Store x- & y-values in two-column data frame

Building a Model

- Initializing individual problem function (IPF)
  - Exercise: Work with partner to combine both these procedures into a custom function that can be used to generate the IPF for a *single agent*
    - » What do we *have* to know (i.e., what are inputs to function)?
      1. Does person have area of expertise (H)?
         A. Hint: Use H in an if-else statement to select which IPF construction procedure to run
      2. What is TPF?
    - » Optional things we might like to add:
      1. Inputs for specifying size & location of area of expertise
      2. If we reuse TPF function *inside* IPF function, want to pass inputs for that function as well

Building a Model

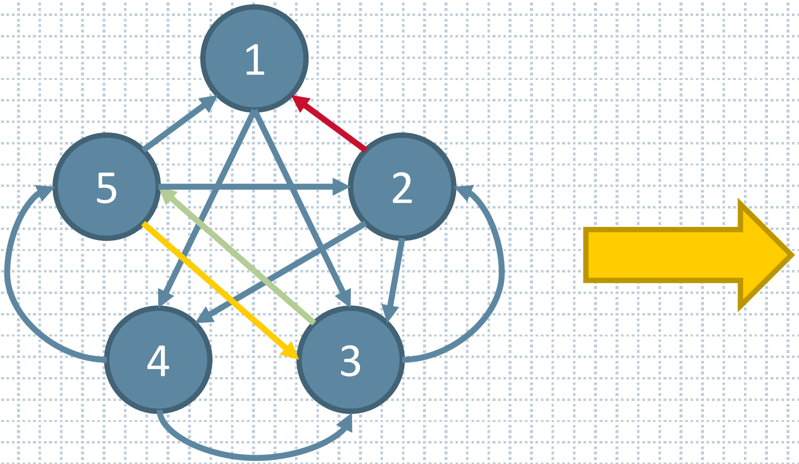
- Initializing individual problem function (IPF)
  - How do we repeat this procedure for *each* agent?
    - » Two considerations:
      1. Method:
         A. for() loop…
         B. replicate() – repeat same function n times
            a. Use when inputs to function don't need to change
         C. apply statements
         D. replicate() would work for us here…but we'll use an apply statement
      2. Data storage:
         A. IPF is a data frame…so we have many options
         B. "Stacked" data frames, 3-dimensional array, list of data frames
         C. Lists tend to be more efficient, so we'll use those
    - » Let's use lapply() to run our IPF function for each agent
      1. How many agents do we need to initialize?
      2. Doesn't matter for our function…but team size is a parameter so let's use that

Building a Model

- Initializing individual problem function (IPF)
  - Extending IPF function
    - H parameter is a *team-level* parameter in Dionne et al.
      1. Team is either *all* specialists or *all* generalists
    - How could code be changed to allow for teams of *mixed* specialists and generalists?
      1. Hint: How is H parameter being used in lapply statement?

Building a Model

- Initializing team network
  - Lots of ways to handle networks in R…
    - » Easiest/most flexible is to represent in adjacency matrix
      1. n x n matrix, where n = # of objects (e.g., people, concepts) involved in relationship
      2. Might have input and output networks
         A. e.g., Input = relationship, communication, interdependence
         B. e.g., Output = cohesion, influence, cognition
    - » IMPORTANT – designate and distinguish "source" & "target"
      1. Source = rows, Target = columns (i.e., mat[1,2] = relationship from 1 to 2)
      2. Simple adjacency network → 1 = relationship present, 0 = no relationship present



Target

Source

	1	2	3	4	5
1	1	0	1	1	0
2	1	1	1	1	0
3	0	1	1	0	1
4	0	0	1	1	1
5	1	1	1	0	1

Building a Model

- Initializing team network
  - What do we need to create?
    - n x n matrix with 0's in cells where agents are not connected and 1's in cells where agents are connected
    - Linkages determined by leader in-group designation…need to specify:
      1. Which agent is leader
      2. Size of in-group (k)
      3. Which agents are part of in-group
  - How could we do this?
    - Description on p. 1042 of Dionne et al. (2010)

> In our model the interaction and convergence described above takes place only through social ties between members in a network that is pre-formed by the leader (i.e., the leader's ego network). To represent different leadership styles in social network structure, a final experimental parameter "Number of In-group Members," K, is introduced, whose values range from 2 to N. A social network is created using this parameter in the following way: first, a fully connected network of K members (including the leader) is generated to represent an in-group cluster. Then the remaining $N-K$ out-group members are connected to the leader with a single tie. With this algorithm, $K=2$ reflects a star-shaped network more likely typified by LMX leadership, while $K=N$ represents a fully connected network more likely typified by participative leadership (see Fig. 3).

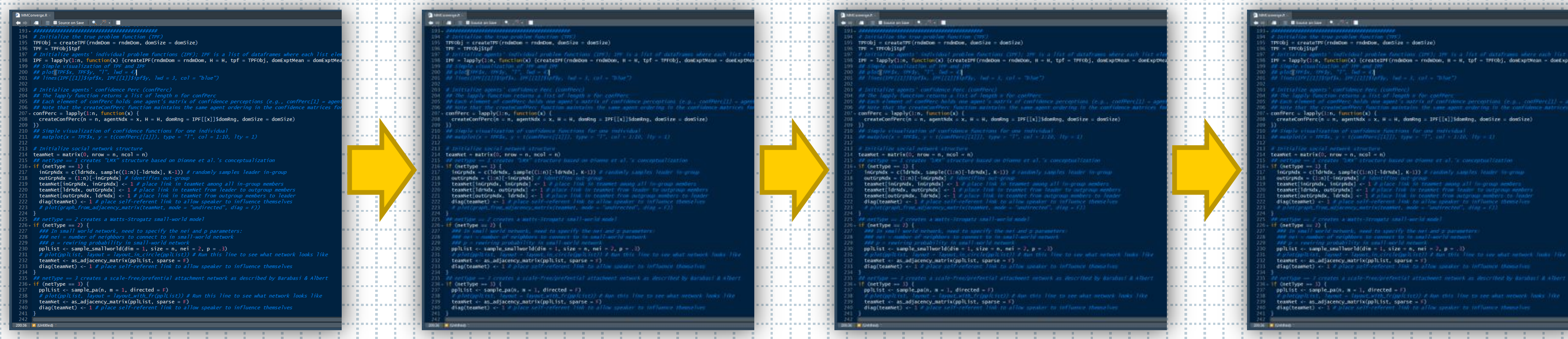
BUILDING A MODEL

- Initializing team network
  - <u>Exercise</u>: Work with partner to write pseudocode & R Code for team network
    - Make leader designation random
    - Make in-group/out-group selection random
  - Pseudocode
    - Randomly sample single leader from among n agents
    - Create n x n team relationship matrix of all 0's
    - Identify in-group by randomly sampling K agents from remaining agents
      1. Out-group = non-in-group agents
    - Link all agents in the in-group together in team relationship matrix
    - Link leader to all out-group members
    - Link out-group members to leader
    - Link agents to self

Building a Model

- Model process code
  - Learning point!
    - » All computational models involve actions unfolding over time
      1. Processes are repeated/carried out, variables/outputs updated, etc.

Time 1 Time 2 Time 3 Time 4



    - » Will <u>always</u> need at least one loop in R code to iterate process
    - » Use for loop or while loop to advance time?
      1. For loop → number of time points to simulate is known
      2. While loop → stopping criteria is known

Building a Model

- Model process code
  - Basic structure/components:

Untitled1*

Source on Save

```r
#########
# MODEL #
#########
for (i in 1:nIter) {
  # Update input/outputs at beginning of time i

  # Actions to carry out during time i

  # Update input/outputs at end of time i
}
```

Time loop

Code to update variables

Code to perform actions

  - Learning point!
    - » Order of events matter – make sure updating occurs when it's supposed to!

BUILDING A MODEL

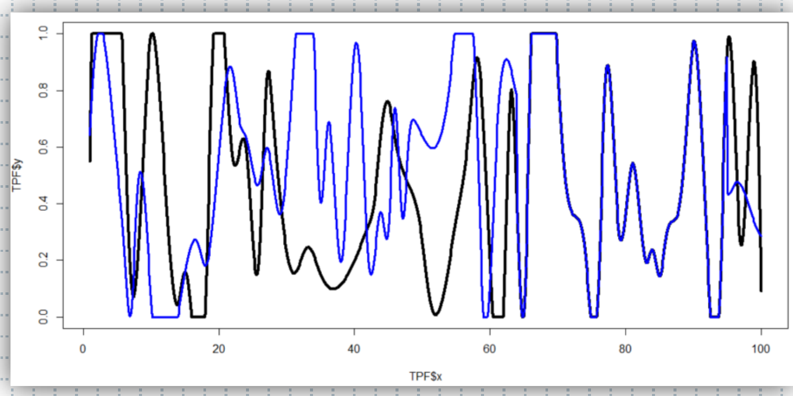
- Reminder of pseudocode
  - Initialization
    - Create true problem function
    - Create individual problem functions
    - Create confidence perceptions
    - Create team network
  - Process
    - Step 1: Select speaker as a function of self-confidence
    - Step 2: Speaker selects topic to speak about as a function of self-confidence and shares with members
    - Step 3: Connected members hear and form evaluation of opinion
    - Step 4: Connected members update confidence perceptions and individual problem functions
    - Repeat Steps 1-4 until time limit is reached

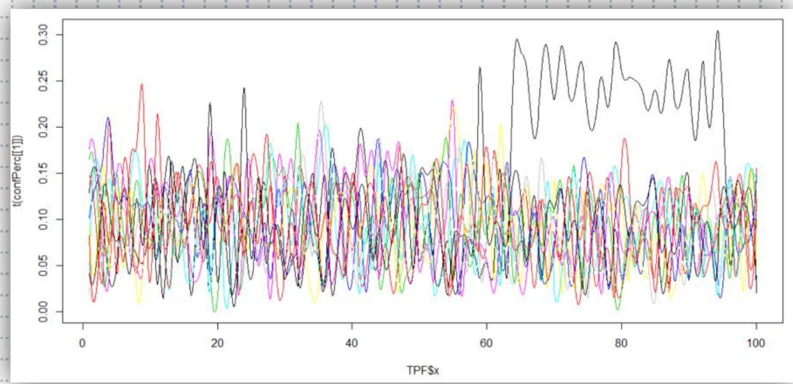
BUILDING A MODEL

- Open file: MMConverge_InitNoProcess.R
  - Contains code for complete model initialization, but no process code
    - » Parameters
      1. Optional exercise: I've added a few additional parameters to allow some additional control over initialization…try to figure out what these do later!
    - » Model functions
      1. createTPF
      2. createIPF
      3. createConfPerc → create confidence perceptions for agent
         A. Higher (self-)confidence where agent has expertise
      4. computeGPF → compute aggregate IPF weighted by average confidence
      5. evalBelief → equation 1 in Dionne et al. (2010)
      6. confUpdate → update confidence perceptions following evaluation
      7. ipfUpdate → update IPFs following evaluation
      8. confDist → compute convergence of all agent's confidence perceptions

BUILDING A MODEL

- Open file: MMConverge_InitNoProcess.R
  - Contains code for complete model initialization, but no process code
    - » Initialization
      1. TPF
      2. IPF → list of each agent's IPF
      3. confPerc → list of each agent's confidence perceptions
         A. confPerc[[x]] = n x 1000 matrix containing agent x's confidence perceptions
         B. confPerc[[x]][y,] = agent x's confidence perceptions of agent y

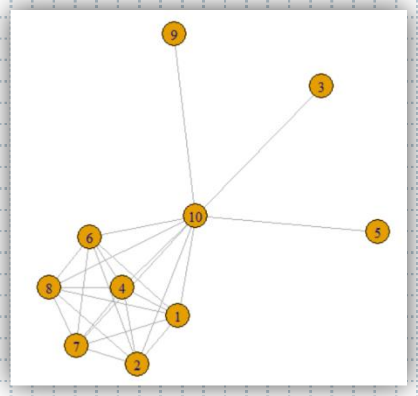


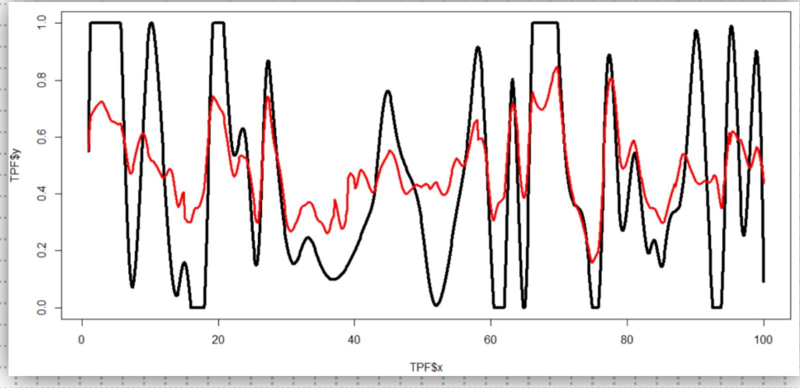


Building a Model

- Open file: MMConverge_InitNoProcess.R
  - Contains code for complete model initialization, but no process code
    - » Initialization

      4. teamNet → n x n adjacency matrix
      5. GPF
      6. speaker → vector of which agent speaks @ each time point
      7. mmStats → data frame for storing output
         A. Column 1 = time point ("time")
         B. Column 2 = MM accuracy ("acc")
         C. Column 3 = MM convergence ("confDist")





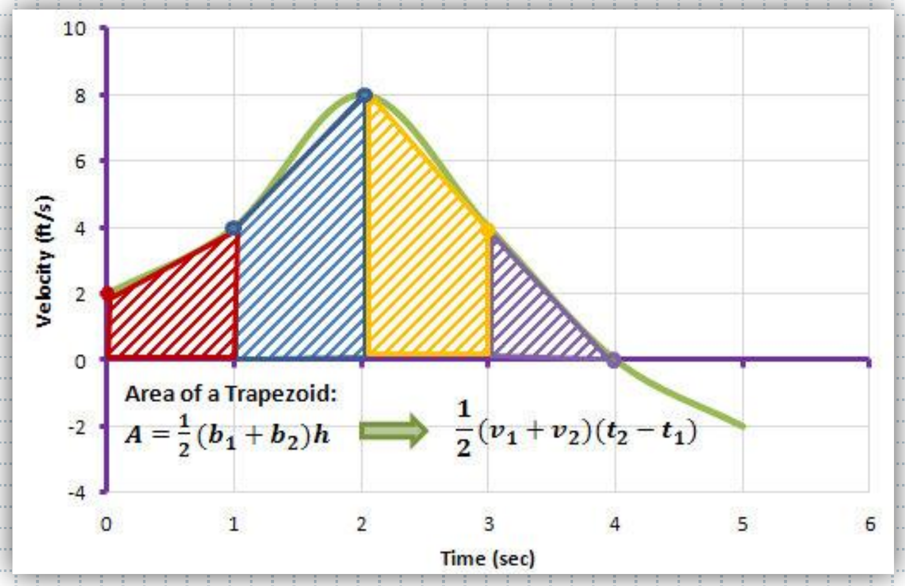
Building a Model

- Running Step 1: Select speaker
  - What happens in this action?
    - » Speaker is chosen to share information
    - » Selection occurs probabilistically based on overall self-confidence
  - What needs to occur for this to happen?
    - » Compute integral for each agents' self-confidence curve
    - » Transform integrals into probabilities
    - » Sample single speaker given probabilities

1. A speaker is selected out of N agents, using their overall self-confidence as the probabilities of selection. An overall self-confidence of an agent is calculated by integrating the agent's individual self-confidence function over the entire problem domain. This assumption models the notion that the more self-confident a member is, the more often he/she speaks.

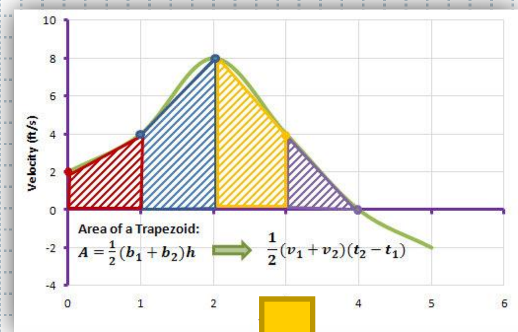
Building a Model

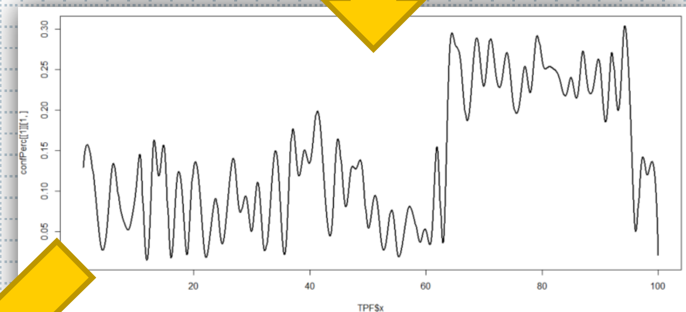
- Running Step 1: Select speaker
  - Compute integral for each agents' self-confidence curve
    - » Integral = area under curve (AUC)
    - » AUC can be estimated using *trapezoid rule*
      1. Draw trapezoids that connect points on curve and add together areas
    - » Equations:
      1. $A_{trapezoid} = (x_2 - x_1)*(y_1 + y_2)*0.5$
      2. $AUC = \sum(A_{trapezoid})$



Building a Model

- Running Step 1: Select speaker
  - What do we need to know to compute AUC using trapezoid rule?
    - Base of trapezoid (e.g., x_2 - x_1)
    - Height at both ends of trapezoid (e.g,. y @ x_1, y @ x_2)





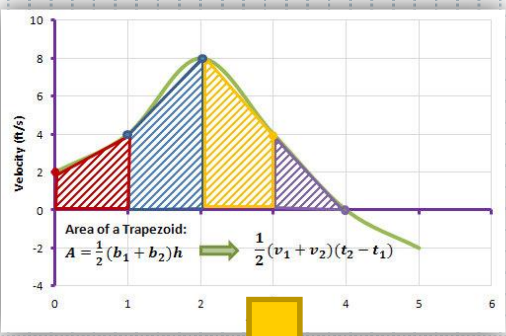
(x-values look odd because of interpolation)

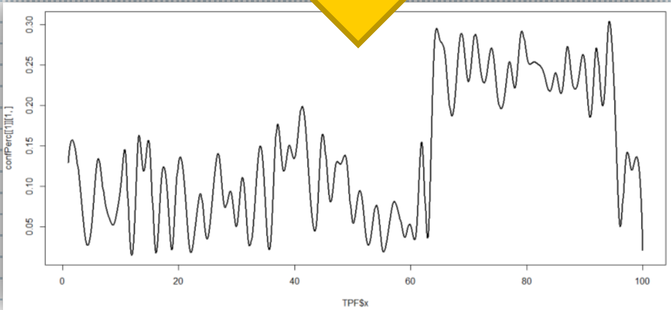
Sum ≈ AUC

Trapezoid #	X_1	X_2	$X_2 - X_1$	Y_1	Y_2	$A_{trapezoid}$
1	1	1.099	.099	.129	.138	.013
2	1.099	1.198	.099	.138	.144	.014
3	1.198	1.297	.099	.144	.149	.015
4	1.297	1.396	.099	.149	.152	.015
5	1.396	1.495	.099	.155	.157	.015
…	…	…	…			…
999	99.900	100	.099	.047	.020	.003

BUILDING A MODEL

- Running Step 1: Select speaker
  - Where do we get these values from?
    - » x → save from createTPF function (TPF$x)
    - » y → confPerc[[x]][x,] (self-confidence)
  - Let's look at this object in R:
    - » TPF$x gives actual x-values…
    - » …but we need <u>difference between</u> values
      1. Use diff() function in R

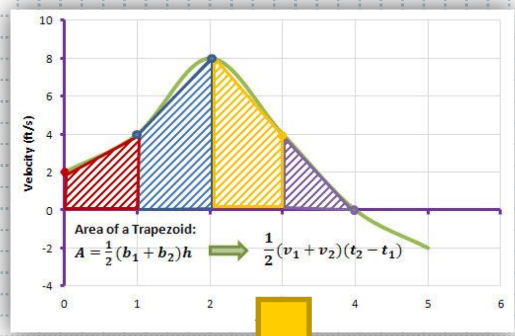


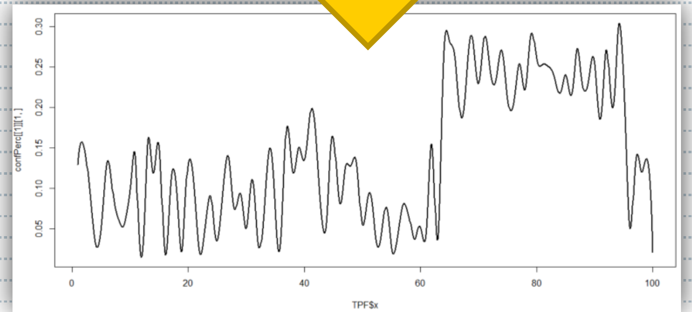


TPF$x	1	1.099	1.198	1.297	1.396

diff(TPF$x)	1.099-1	1.198-1.099	1.297-1.198	1.396-1.297
output	.099	.099	.099	.099

BUILDING A MODEL





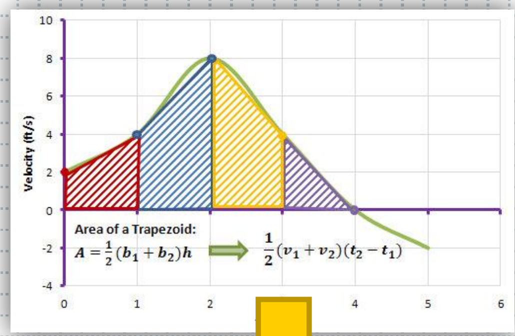
- Running Step 1: Select speaker
  - Where do we get these values from?
    - » x → save from createTPF function (TPF$x)
    - » y → confPerc[[x]][x,] (self-confidence)
  - Let's look at this object in R:
    - » confPerc[[x]][x,] gives actual y-values…
    - » …but we need <u>sum between</u> values
      1. No base function in R to do this…so need to be creative!

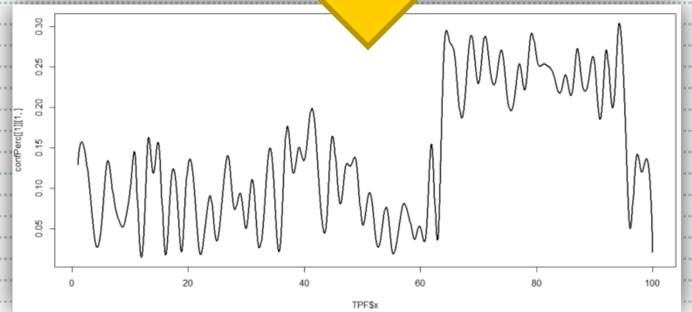
confPerc	.129	.138	.144	.149	.155

sumConfPerc	.129+.138	.138+.144	.144+.149	.149+.155
output	.267	.282	.293	.304

Building a Model

- Running Step 1: Select speaker
  - Where do we get these values from?
    - x → save from createTPF function (TPF$x)
    - y → confPerc[[x]][x,] (self-confidence)
  - Let's look at this object in R:
    - confPerc[[x]][x,] gives actual y-values…
    - …but we need <u>sum between</u> values
      1. No base function in R to do this…so need to be creative!
         A. head(x, -1) → give all values in x EXCEPT last one
         B. tail(x, -1) → give all values in x EXCEPT first one

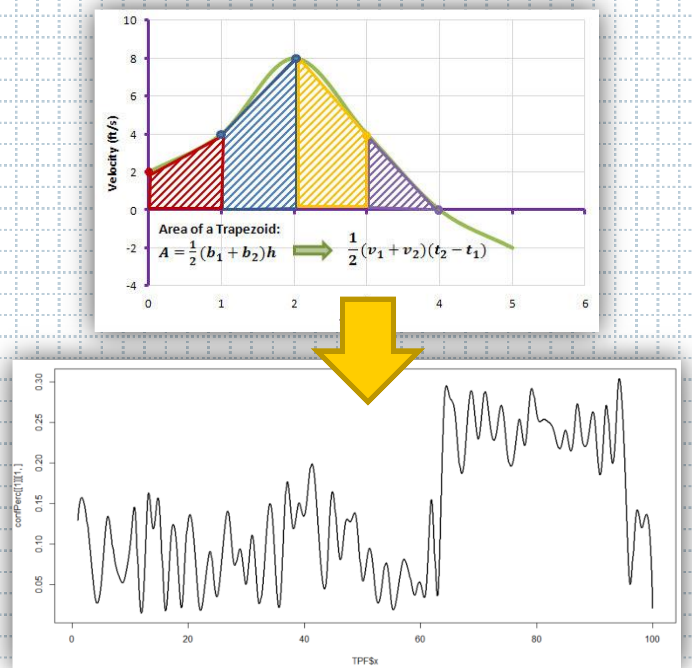




head(confPerc, -1)	.129	.138	.144	.149	.155
	+	+	+	+	
tail(confPerc, -1)	.138	.144	.149	.155	…
output	.267	.282	.293	.304	…

Building a Model

- Running Step 1: Select speaker
  - <u>Exercise</u>: Work with a partner to write R code for computing & saving AUC for EACH agent to a single vector
    - » Equations:
      1. $A_{trapezoid} = (x_2 - x_1)*(y_1 + y_1)*0.5$
      2. $AUC = \sum(A_{trapezoid})$
    - » <u>Hints</u>:
      1. Figure out how to compute AUC for just one agent first
         - A. Grab x values, compute diff()
         - B. Grab y values for agent, compute sums
      2. For multiple agents, you'll need to loop or use sapply()
      3. Final output should be a vector of length n



Building a Model

- Running Step 1: Select speaker
  - What needs to occur for this to happen?
    - » ~~Compute integral for each agents' self-confidence curve~~
    - » Transform integrals into probabilities
      1. Need to convert AUC to be on scale from 0 – 1
      2. pSpeak = AUC / sum(AUC)
    - » Sample single speaker given probabilities
      1. Use sample() function to select one speaker using pSpeak

1. A speaker is selected out of N agents, using their overall self-confidence as the probabilities of selection. An overall self-confidence of an agent is calculated by integrating the agent's individual self-confidence function over the entire problem domain. This assumption models the notion that the more self-confident a member is, the more often he/she speaks.

Building a Model

- Running Step 2: Speaker selects topic
  - What happens in this action?
    - » Speaker selects part of problem domain to share probabilistically based on self-confidence
  - What needs to occur for this to happen?
    - » Transform speaker's self-confidence to probabilities (between 0-1)
    - » Sample topic to share by speaker using probabilities

2. *Orientation*. A topic (i.e., a specific location within the problem domain) is selected from the entire problem domain, using the speaker's self-confidence function as a probability distribution function. This assumption models the notion that people tend to speak about topics in which they have more confidence. Then, the speaker expresses his/her opinion on the selected topic (i.e., the value of his/her IPF at the selected location in the problem domain).

Building a Model

- Running Step 2: Speaker selects topic
  - <u>Exercise</u>: Work with partner to write R code for sampling topic to share by speaker
    - Hints:
      1. Only need to work with the confidence perceptions of the speaker!

Building a Model

- Takeaways from afternoon session
  - Stick at it!
    - » Learning to program is like learning any new software…takes time, patience, and practice
  - "Thinking" like a modeler will help you get better at programming
    - » Developing computational models = thinking about psychological phenomena at the "pseudocode" level
    - » Most of us weren't trained this way…but can be developed!
  - The internet is your friend
    - » Major advantage of using R is huge user base and many forums where people <u>have</u> answered <u>your</u> question
    - » stackoverflow will save you countless hours…

Wrap-up

- Plan for tomorrow:
  - Morning – Verification, validation, and reviewing models
  - Afternoon – Creating & running simulations
    - Will use full model code, so make sure to download
    - Work through model code we didn't complete and try to understand
      1. Bring questions about remaining code