CHICAGO_ABM.R INSTRUCTION MANUAL

THOMAS LAETSCH

Contents

Part 1. Introduction and Conventions

Throughout these notes, we will often refer to *dynamic variables* and *dynamic function variables*, which can be considered the "tuning parameters" of the ABM. The point of these dynamic objects is that if they obey the constraints discussed in the remainder, then the ABM should still run, respecting whatever changes were made. For example, we have many dynamic function variables,

which accept certain inputs and must produce a certain type of output; however, how we use the input to then produce the output is the "tunable" feature over which we have control.

In the simplest case, a dynamic variable is a simple atomic value, such as a positive integer, where if we vary amongst the same constrained atomic type, we can witness how this affects the ABM. The more intricate considerations are the dynamic function variables. It will seem that these functions act as wrappers around the actual out-of-the-box function we want to use, but this is not the correct interpretation. Instead, we should consider the out-of-the-box function, which we will refer to as an *auxiliary function*, as one of infinitely many possible realizations of the the dynamic function variable, and is chosen for some principled reason. To illustrate this via a trivial example, suppose that we have the dynamic function variable `INT_ADJUSTer` which accepts a single integer value and must output an integer value. Of course, there are infinitely many ways to do this, and our approach will be to define some auxiliary function which will be one of these possible ways, but which may give us some extra dials to adjust to make tuning a bit easier. So, e.g., let's define `Add` as

```
Add <- function( x, y ){
     return( x + y )
}
```

Then, we could realize `INT_ADJUSTer` by

```
INT_ADJUSTer <- function( x ){
     Add(x = x, y = 2)
}
```

if for some reason we believe that addition is a reasonable way for `INT_ADJUSTer` to work. Note that as the constraints require, this definition of `INT_ADJUSTer` inputs a single integer value and outputs an integer value (which, if `INT_ADJUSTer` were truly a dynamic function variable for the ABM, means that somewhere in the background workings of the ABM, this function will be called, being passed and integer value with an expected return of an integer value). However, via the auxiliary function `Add`, we've been able to introduce another parameter `y` to easily tune the addition. In this `INT_ADJUSTer` example, there really isn't a great benefit to defining the auxiliary function `Add`, since it is so simple and we'd easily be able to just define `INT_ADJUSTer` directly, but many of the dynamic function variables used by the ABM are far more complex.

Finally, to introduce the last of the vocabulary which will be sewn throughout these notes, we will often refer to functions which run the ABM via calls to the dynamic objects as *static functions*.

The emphasis here is that they are not the tunable pieces of the ABM, but rather compile the tunable pieces to run the ABM.

Part 2. Hostile Agent Movement

On initialization, each hostile agent is assigned a "base node" in the playground, corresponding to a randomly selected node within their assigned neighborhood. During each timestep, a hostile agent will first choose a neighborhood within which to move, done via the dynamic function variable `HOST_NBHD_JUMPer`, and from there, choose a node within that neighborhood to move to, done via the dynamic function variable `HOST_NODE_JUMPer`. The static function which updates the hostile motion via these operations is `UpdateHostilesDF`. The code for `UpdateHostilesDF` is the following:

```
UpdateHostilesDF <- function( timestep ){
     # make sure HOSTILES_DF is correctly defined
     if( !HostDFExistChecker() ){
          warning('HOSTILES_DF does not exist or is empty. No movement updated.')
          return( F )
     }
     # create an alias to work on
     agents <- HOSTILES_DF
     # partition by family since attacks are family-level events
     for( fam in unique( agents$family) ){
          atk <- ATTACK( fam )
          atk_bool  <- !is.null(atk)
          # if there is an attack, update hostility records
          if( atk_bool ) UpdateAttackHostilities( fam, atk, timestep )
          in_fam <- agents$family == fam
          ag_in_fam <- agents[in_fam, ]
          # partition by nbhd; JUMPers are nbhd specific
          for( base_nbhd in unique( ag_in_fam$base_nbhd ) ){
               in_nbhd <- ag_in_fam$base_nbhd == base_nbhd
               ag_in_nbhd <- ag_in_fam[in_nbhd, ]
               # choose which nbhd to move to
               ag_in_nbhd <- HOST_NBHD_JUMPer( ag_in_nbhd,
                                               attacking = atk_bool,
```

```
                                                timestep )
                # choose which node to move to
                ag_in_nbhd <- HOST_NODE_JUMPer( ag_in_nbhd,
                                                attacking = atk_bool,
                                                timestep = timestep )
                ag_in_fam[in_nbhd,] <- ag_in_nbhd
            }
            agents[in_fam, ] <- ag_in_fam
        }
        # update HOSTILES_DF
        assign('HOSTILES_DF', agents, .GlobalEnv)
        return( T )
}
```

We continue from here, describing the out-of-the-box functionality of `HOST_NBHD_JUMPer` and `HOST_NODE_JUMPer`. However, the general scheme can be quickly described. When the hostile agent chooses to move within their assigned base-neighborhood, the node selection is simply a return to their base node. If the hostile agent chooses to move to a different neighborhood, then node selection becomes much more intricate, where the selection can be based on distance from the base-neighborhood, distance from enemy neighborhoods, encounters with authority, and previously experienced hostilities. The first step, neighborhood selection, is also a thoughtful procedure where the likeliness of jumping between neighborhoods can be adjusted based on distance, size of the base-neighborhood, and tension levels.

1. Neighborhood Selection

The ABM requires the dynamic function variable `HOST_NBHD_JUMPer`, which will input a data.frame `agents` of hostile agents, a logical variable `attacking` indicating whether the passed agents are in the attack mode, and the integer variable `timestep` indicating the current timestep during the call. The output of `HOST_NBHD_JUMPer` is and updated data.frame of the input agents, with the column `agents$current_nbhd` updated to indicate the updated neighborhood selection for each agent.

1.1. Built-in Functionality.

In the default setup, this neighborhood selection proceeds by producing a hostile-neighborhood specific PMF indexed by neighborhood names, giving likelihoods that agents move to those neighborhoods; from here, that PMF is randomly sampled for each

agent in the specific neighbhorhood and moved accordingly. The intricacies of this procedure fall within producing a meaningful PMF.

1.1.1. *PMF Creation.* The function `PMFer_StdHostileNbhdJumper` produces this PMF based on the input values:

- `base_nbhd` The character string indicating what the base-neighborhood is for which the PMF is in reference to.
- `timestep` An integer value indicating the timestep at which this PMF is being generated with respect to.
- `home_wt_by_quantile` A numeric value or vector indicating relative weight(s), relative to `outside_wt`, with which the agents from the base-neighborhood will choose to move into the base-neighborhood. If a vector is is given, then the weight chosen will be based on the size of the base-neighborhood based by quantile.
- `outside_wt` A numeric value giving relative weight, relative to the corresponding value of `home_wt_by_quantile`, with which the agents from the base-neighborhood will choose to move outside of the base-neighborhood.
- `ether_wt` A numeric value giving relative weight, relative to `friendly_wt` and `enemy_wt`, the agents will move into the ether *given that they move outside of the base-neighborhood.*
- `friendly_wt` A numeric value giving relative weight, relative to `ether_wt` and `enemy_wt`, the agents will move into a friendly designated neighborhood *given that they move outside of the base-neighborhood.*
- `enemy_wt` A numeric value giving relative weight, relative to `friendly_wt` and `ether_wt`, the agents will move into an enemy designated neighborhood *given that they move outside of the base-neighborhood.*
- `dfun` A function which inputs the distance from the base-neighborhood to a alternate neighborhoods to give distance-based weightings within an alternate designation (ether, friendly, enemy). If this is set to a constant value, then the neighborhoods within a given designation are chosen uniformly. *The inverse of these distance weightings are what will be used to create probabilities*, and hence a larger weight corresponds to a higher "cost" of moving to that neighborhood.

Example 1.1. Suppose we execute

```
PMFer_StdHostileNbhdJumper( base_nbhd = nbhd,
                            timestep = 2,
```

```
                    home_wt_by_quantile = c(60,70,80),
                    outside_wt = 10,
                    ether_wt = 100,
                    friendly_wt = 15,
                    enemy_wt = 2,
                    dfun  = function(x){ x^3 } )
```

The base-neighborhood we are creating this PMF for is `nbhd`; the timestep at which we are creating the PMF is 2. Since `home_wt_by_quantile` is a vector of length 3, the hostile neighborhoods will be partitioned into 3 size-based groups, where if the size of `nbhd` is in the bottom third, it will be assigned a home weight of 60, if it's in the middle third, it will be assigned a weight of 70, and in the top third, a weight of 80. The point of this option is that it seems intuitive that the necessity of an agent to move outside of their base-neighborhood might depend on neighborhood size. For this example, suppose that `nbhd` happens to be a large neighborhood, and hence given the home weight of 80. With `outside_wt` set at 10 (and the home weight set at 80), an agent from `nbhd` will have a $\frac{80}{80+10}$ chance of staying within `nbhd` and a $\frac{10}{80+10}$ chance of moving into a different neighborhood.

The final three weight inputs `ether_wt`, `friendly_wt`, and `enemy_wt` give the relative weights for moving either into the ether, into a friendly-designated neighborhood, or into an enemy-designated area. In this example, that means that if the agent chooses to move outside of the base-neighborhood, then that move will be to the ether with probability $\frac{100}{100+15+2}$, to a friendly neighborhood with probability $\frac{15}{100+15+2}$, and to an enemy neighborhood with a probability $\frac{2}{100+15+2}$.

Finally, if the agent has chosen to move outside its base-neighborhood `nbhd` and has chosen which of the three designations of outside neighborhood (ether, friendly, enemy) it will move into, it remains to choose which specific neighborhood within that designation will be chosen. For this, we use distance via `dfun` to give weights to these neighborhoods. For clarity, suppose that an agent has chosen to go to a neighborhood designated as friendly and there are two neighborhoods `f1` at a distance 3 from `nbhd`, and `f2` at a distance 5 from `nbhd`. Then, the selection of which of these two neighborhoods to move into will be weighted as `f1` $\to$ `1/dfun(3)` $= 1/27$ and `f2` $\to$ `1/dfun(5)` $= 1/125$; hence the move into `f1` happens with probabilitly $\frac{1/27}{1/27+1/125} = \frac{125}{125+27}$, and the probability into `f2` happens with probability $\frac{1/125}{1/27+1/125} = \frac{27}{125+27}$.

Of course, we can compile this multi-stage intuition into a concise PMF by correctly calculating what these weightings translate into for each neighborhood in the playground. So, if we also assume there are two enemy neighborhoods `e1` at a distance of 4 from `nbhd` and `e2` at a distance of 6 from

nbhd, the return PMF will be

nbhd	$\frac{80}{80+10}$
ether	$\left(\frac{10}{80+10}\right)\left(\frac{100}{100+15+2}\right)$
f1	$\left(\frac{10}{80+10}\right)\left(\frac{15}{100+15+2}\right)\left(\frac{125}{125+27}\right)$
f2	$\left(\frac{10}{80+10}\right)\left(\frac{15}{100+15+2}\right)\left(\frac{27}{125+27}\right)$
e1	$\left(\frac{10}{80+10}\right)\left(\frac{2}{100+15+2}\right)\left(\frac{216}{216+64}\right)$
e2	$\left(\frac{10}{80+10}\right)\left(\frac{2}{100+15+2}\right)\left(\frac{64}{216+64}\right)$

where the triple line represents the division from home weight and outside weight (manifesting as the first factor in each product); the double lines within the "outside home" neighborhoods represent the division between designations (affecting the second factor in each product); and the single lines represent the division between individual neighborhoods within designations (creating the distance-based third factor in each product). △

1.1.2. *Neighborhood Selection.* Housing the PMF creation is the function StdHostileNbhdJumper, which will pass the appropriate user-defined (and potentially adjusted; to be described shortly) inputs into PMFer_StdHostileNbhdJumper, and sample from the returned PMF, resulting in updated neighborhoods for the passed agents. The inputs for StdHostileNbhdJumper are

- agents A data.frame of agents with which to update the current_nbhd column.
- timestep The timestep of the call.
- home_wt_by_quantile As used in PMFer_StdHostileNbhdJumper.
- home_wt_by_quantile As used in PMFer_StdHostileNbhdJumper.
- outside_wt As used in PMFer_StdHostileNbhdJumper.
- ether_wt As used in PMFer_StdHostileNbhdJumper.
- friendly_wt As used in PMFer_StdHostileNbhdJumper.
- enemy_wt As used in PMFer_StdHostileNbhdJumper.
- dfun As used in PMFer_StdHostileNbhdJumper.
- adjust_for_tension A logical value indicating whether or not to adjust home_wt_by_quantile based on current tension between neighborhoods (using fear_fac and anger_fac values below).
- fear_fac A numeric value acting as a coefficient factor, used if adjusting for tension.
- anger_fac A numeric value acting as a coefficient factor, used if adjusting for tension.

As mentioned above, `PMFer_StdHostileNbhdJumper` returns a neighborhood specific PMF, so sensibly, the rough pipeline of `StdHostileNbhdJumper` is to group `agents` by neighborhood, then retrieve the corresponding PMF for each neighborhood, updating `current_nbhd` within each group by sampling from this PMF.

The added dynamism that `StdHostileNbhdJumper` gives is the ability to adjust the `home_wt_by_quantile` values to account for tension. Intuitively, this functionality will make hostile agents more likely to stay within their base-neighborhood when enemy neighborhoods feel anger towards them (adjustments based on `fear_fac` value passed), or possible be less likely to stay within their base-neighborhoods when they feel anger towards enemy neighborhoods (adjustments based on `anger_fac` value passed). Explicitly, if the neighborhood we are considering to create a PMF for is `nbhd`, let M_{anger} be the maximum value of `TENSION_MATRIX` along the *row* corresponding to `nbhd`, and let M_{fear} be the maximum value of `TENSION_MATRIX` along the *column* corresponding to `nbhd`. Then, with `adjust_for_tension` set to `TRUE`, we make the adjustment

$$\texttt{home\_wt\_by\_quantile} \to \texttt{(1.0 + fac) * home\_wt\_by\_quantile}$$

where

$$\texttt{fac} = \left[\texttt{fear\_fac} \times \frac{M_{\text{fear}} - \texttt{MIN\_TENSION}}{\texttt{MAX\_TENSION} - \texttt{MIN\_TENSION}}\right] - \left[\texttt{anger\_fac} \times \frac{M_{\text{anger}} - \texttt{MIN\_TENSION}}{\texttt{MAX\_TENSION} - \texttt{MIN\_TENSION}}\right]$$

thus potentially increasing the "stay home" weights due to fear, and decreasing the "stay home" weights in times of anger.

1.1.3. *Putting It Together:* `HOST_NBHD_JUMPer`. The dynamic function variable `HOST_NBHD_JUMPer` is what the static function calls expect when updating agent motion. One might interpret `HOST_NBHD_JUMPer` as a wrapper around `StdHostileNbhdJumper`, but instead, it's better to think of `StdHostileNbhdJumper` as one of infinitely many possible realizations of how the hostile agents might choose to jump between neighborhoods, and that by default `HOST_NBHD_JUMPer` uses `StdHostileNbhdJumper`. Out of the box, we have the following definition

```
HOST_NBHD_JUMPer <- function( agents, attacking, timestep, ... ){
     if( attacking ){
          updated_agents <- StdHostileNbhdJumper( agents,
                                                  timestep,
                                                  home_wt_by_quantile =
                                                         c(60, 70, 80, 90, 100),
```

```
                                                outside_wt = 2,
                                                ether_wt = 100,
                                                friendly_wt = 15,
                                                enemy_wt = 0,
                                                dfun  = function(x){
                                                  dist3( dist = x,
                                                         coef = 1,
                                                         min_val = 1)
                                                },
                                                adjust_for_tension = T,
                                                fear_fac = .15,
                                                anger_fac = 0.0 )

    } else {
         updated_agents <- StdHostileNbhdJumper( agents,
                                                timestep,
                                                home_wt_by_quantile =
                                                       c(60, 70, 80, 90, 100),
                                                outside_wt = 5,
                                                ether_wt = 100,
                                                friendly_wt = 15,
                                                enemy_wt = 2,
                                                dfun = function(x){ x^3 },
                                                adjust_for_tension = T,
                                                fear_fac = 0.1,
                                                anger_fac = 0.0 )
    }
    return( updated_agents )
}
```

To note, the ABM is setup so that attacks happen at the family level, so multiple neighborhoods all belonging to one single family will simultaneously be in the same attack mode (attacking another family or not). This means that, while we could use `HOST_NBHD_JUMPer` one agent at a time, correctly adjusting the `T/F` value of **`attacking`** for each agent, we can and do speed up the process by

aggregating hostile agents by family, and passing the entire family of agents into `HOST_NBHD_JUMPer` with the family-wide attacking value.

2. Node Selection

The ABM requires the dynamic function variable `HOST_NODE_JUMPer`, which will input a data.frame `agents` of hostile agents, a logical variable `attacking` indicating whether the passed agents are in the attack mode, and the integer variable `timestep` indicating the current timestep during the call. These are the same inputs as `HOST_NBHD_JUMPer` takes (discussed above); however, since node-jumping is the subsequent call after jumping neighborhoods, it is assumed that the column `agents$current_nbhd` has been updated to the neighborhood where the agents will be moving to, and hence used to decide which nodes are candidates for relocation for each agent.

2.1. **Built-in Functionality.** Similar to neighborhood selection, the rough outline of how a node is selected once a neighborhood is determined is by sampling from a base-neighborhood specific PMF, which is calculated via a weighting scheme on the nodes. Unlike the neighborhood selection, some sources of node weights will act to deter agents from landing on nodes, while other sources will act to attract agents to nodes, so combining these multiple sources of node weights is not as simple as before. That said, the input `dfun` in the neighborhood-jumping machinery above acted as a special case of the weighting we will discuss here, and is consistent with our following scheme.

These multiple sources of weights are combined to give a cumulative weight to each node. We will consider that cumulation as a detracting weight and hence assign the inverse of this cumulative weight when calculating the PMF (again, reference to the `dfun` weighting in the neighborhood-jumping section to confirm that this is following the same paradigm). It is helpful to think of the cumulative weight as a "cost" in this context so that the larger the weight, the more it "costs" an agent to go there.

The sources of weights for the nodes are broken into two types: 1) distance-based weights, 2) encounter-based weights. The distance-based weights are setup so that hostile agents are more prone to move nearby their base-neighborhood, but prefer staying further away from enemy neighborhoods. The encounter-based weights deal with the "memory" of the agents, which may choose to avoid nodes where they have had an undesirable encounter (such as encountering authority agents, or experiencing a hostility). Currently, these weights only apply when an agent is moving outside their own neighborhood, since as mentioned above, our out-of-the-box setup has it so that if an agent jumps to their base-neighborhood, then they simply return to their base-node, determined on initialization.

Before delving into more specifics of the function setup, let us first make concrete the discussion above on how we combine these different weight sources. For this, let *positive* weights refer to those which increase the "cost" and *negative* weights as those which decrease the "cost" (i.e., positive weights will deter agents, whereas negative weights will attract agents). If we let $a_1, ..., a_k$ be the positive weights and $b_1, ..., b_m$ be negative (these are node dependent values) , then the *cumulative weight* at node x is given by

$$\text{wt}(x) = w + \sum_{i=1}^{k} a_i(x) - \sum_{j=1}^{m} b_j(x) - \min_y \left(\sum_{i=1}^{k} a_i(y) - \sum_{j=1}^{m} b_j(y) \right) \tag{1}$$

Here the value w is called the *weight offset* and attributes a constant weight across all nodes. The minimum value on the right hand side is taken over all nodes y in the neighborhood the agent would be moving to and has the effect of shifting the non-offset part to 0 at the least weighted nodes (although we do refer to "negative weights" we don't want the adjusted weights to ever be negative at any node, otherwise that will force us to try to assign negative probability). Notice also that the shift ensure that if subtracting the negative weights results in negative values, the more negative values will be shifted up nearer zero, keeping consistent with our desire to give smaller weights (less cost) more likelihood. From here, the assignment of a PMF is simply done via normalization of the inverse of the cumulative weight

$$\text{pmf}(x) = \frac{\big(1/\text{wt}(x)\big)}{\sum_y \big(1/\text{wt}(y)\big)}. \tag{2}$$

where the sum in the denominator is over all nodes y in the selected neighborhood the agent would be moving to. Note that asymptotically as $w \to 0$, the only weights on a node are given by the weight sources and this very possibly means that certain nodes will strongly dominate attraction (those which are shifted nearest to 0). Conversely, if $w \to \infty$, then the offset dominates the other weights and the PMF tends towards a uniform distribution over the nodes, giving very little preference to one node over another.

2.1.1. *Distance-Based Weight Generation.* There are two built-in distance-based weight creation functions. The first we discuss is `WTer_StdHostileNodeWtsByDist` with inputs:

- `base_nbhd` The character string name of the base-neighborhood of the agent(s) for which these weights will apply.
- `dfun` A function which inputs the distance of a node to the base-neighborhood and outputs a distance-based weight. For natural motion (staying nearer the base-neighborhood),

this function should be defined so increasingly larger distances result in increasingly larger output values.

This function is implemented whenever a hostile agent moves away from their base-neighborhood and is a *positive* weight, since larger values imply larger cost.

The second distance-based weighting function is `WTer_AvoidEnemyNodesByDist`, with similar inputs:

- `base_nbhd` The character string name of the base-neighborhood of the agent(s) for which these weights will apply.
- `dfun` A function which inputs the distance *from a node to the nearest enemy neighborhood* and outputs a distance-based weight. For natural motion (tending away from enemy neighborhoods), this function should be defined so increasingly larger distances result in increasingly larger output values.

This function is also implemented whenever a hostile agent moves away from their base-neighborhood, but is a *negative* weight, since larger values imply lesser cost (larger values here – when defined for natural motion – will mean further distance from enemy nodes). In this case, since it is possible a hostile agent moves into an enemy neighborhood, the function `dfun` can have an input of 0 (if the ABM metric is setup in any sensible way, this should be an if and only if statement), hence how `dfun` is defined at 0 will determine the weights from this function while an agent is in an enemy neighborhood – an intuitively sensible definition is `dfun(0) = 0`, based on what these weights represent.

Example 2.1. Suppose that a hostile agent from neighborhood `nbhd` moves has decided to move into the ether neighborhood. If x_1 and x_2 are nodes in the ether, with x_1 a distance 7 from `nbhd` and a distance 5 from the nearest enemy neighborhood, and x_2 a distance 9 from `nbhd` and a distance 10 from the nearest enemy neighborhood, then given the following calls

```
WTer_StdHostileNodeWtsByDist( base_nbhd = nbhd, dfun  = function(x){ 2*x^3 } )
WTer_AvoidEnemyNodesByDist( base_nbhd = nbhd, dfun  = function(x){ x^2 } )
```

the resulting distance-based relative weights for x_1 and x_2 are

$$x_1 \to 2 \times 7^3 - 5^2 = 661$$

$$x_2 \to 2 \times 9^3 - 10^2 = 1358$$

△

It's important to notice in this previous example that since the difference of these two `dfun` outputs will be compared via subtraction, we could make sensible definitions for each individually, but when combined cause strange behavior (such as having agents move to far away nodes because the preference is given too strongly towards avoiding nodes near enemies rather than staying near the base-neighborhood). The moral is, *consider all weightings together when adjusting their individual parameters.*

2.1.2. *Encounter-Based Weight Generation.* These weights are based on "memory" as discussed in §2.2.2. Out of the box, there are three types of encounters which can be considered for hostile agent motion: encountering authority agents, encountering enemy agents, and encountering a hostility. All three of these types of encounters are considered positive weights, since more weight will be given where the encounters take place and those weights increase the "cost" to the hostile agent, making them more likely to avoid nodes where they have encountered authority, enemies, or experienced a hostility. The function creating weights for authority encounters is `WTer_AvoidAuthNodes`, for creating weights in enemy encounter is `WTer_AvoidEnemyNodes`, and for creating weights based on hostilities is `WTer_AvoidHostilityNodes`. Each of these functions share the inputs:

- `base_nbhd` The character string indicating what the base-neighborhood is for which the weights are in reference to.
- `timestep` An integer value indicating the timestep at which this PMF is being generated with respect to.
- `mem_coef` A non-negative numeric value indicating the coefficient for the memory of an encounter. (See §2.2.2)
- `mem_persist` A positive numeric value indicating the persistence for the memory of an encounter. (See §2.2.2)

In addition to the previous, `WTer_AvoidHostilities` also has the input:

- `only_victim` A logical value, where if set to `T`, will only weight nodes of hostilities when an agent from the base-neighborhood was the victim of the hostility, otherwise giving weights if either they were the victim or perpetrator of the hostilities.

The function creating weights for enemy encounters is `WTer_AvoidEnemyNodes`, with inputs:

2.1.3. *Putting It Together:* `HOST_NODE_JUMPer`. Now that we have described how the weights are calculated via the `WTer` functions, and how they are cumulated and converted to a PMF via (1) and (2), the node-jumping motion of hostile weights is almost fully explained. To finish off, we note that the function to cumulate and convert the weights into a PMF is `PMFer_StdHostileNodeJumper_new`

and is called within `StdHostileNodeJumper`, the function which takes this PMF and samples from it to place each hostile agent into a node. As before, we consider `StdHostileNodeJumper` one of infinitely many possible choices for hostile agent node-jumping, which can be used to define the dynamic global function `HOST_NODE_JUMPer`. Out of the box, we have the following definition

```
HOST_NODE_JUMPer <- function( agents, attacking, timestep, ... ){
     # currently not using the attacking information for this stage
     updated_agents <- StdHostileNodeJumper( agents,
                                             timestep,
                                             wt_offset = 1,
                                             adjust_by_travel_dist = T,
                                             adjust_by_enemy_dist = T,
                                             adjust_by_auth_enc = F,
                                             adjust_by_enemy_enc = F,
                                             adjust_by_hostility_enc = F,
                                             travel_dfun = function(x){
                                                  distGraded( dist = x,
                                                              close_coef = 2,
                                                              close_cutoff = 10,
                                                              mid_cutoff = 20 )
                                             },
                                             enemy_dfun = function(x){
                                                  dist1( dist = x,
                                                         coef = 1 )
                                             },
                                             auth_mem_coef = 2,
                                             auth_mem_persist = 1.5,
                                             enemy_mem_coef = 2,
                                             enemy_mem_persist = 1.5,
                                             hostility_mem_coef = 2,
                                             hostility_mem_persist = 1.5,
                                             only_victim = T  )
     return( updated_agents )
}
```

2.2. Distances and Memory in Node Weights: In Depth.

2.2.1. *Distance in Node Weights.* There are several distance-based weighting functions built-in from which to choose. The first three – `dist1`, `dist2`, `dist3` – are quite simple to interpret

```
dist1 <- function( dist, coef, min_val = 0 ){
     return( coef * max( dist, min_val ) )
}



dist2 <- function( dist, coef, min_val = 0 ){
     return( coef * max( dist, min_val )^2 )
}



dist3 <- function( dist, coef, min_val = 0 ){
     return( coef * max( dist, min_val )^3 )
}
```

where `distN` will return the distance raised to the `N`th power for $N = 1, 2, 3$. In these, the argument `min_val` is to prevent the return from hitting 0 if the user is nervous about dividing by the distance. (In `HOST_NBHD_JUMPer`, the distance weights are calculated via `dist1` with `min_val=1`, since we divide by these distances to get weights and use this as an extra measure of security; in `HOST_NBHD_JUMPer` we do not need to worry about setting `min_val` to a non-zero value since the cumulative weights we divide by include a positive weight-offset).

The fourth weight-from-distance function – `distGraded` – is more involved and warrants a bit more discussion. The code is

```
distGraded <- function( dist,
                        close_coef,
                        close_cutoff,
                        mid_cutoff,
                        min_val = 1 ){
     # coef chosen to create smooth cutoff boundaries
     mid_coef <- close_coef / close_cutoff
     far_coef <- mid_coef / mid_cutoff
```

```
## Error in initFields(scales = scales):  could not find function "initRefFields"
```

FIGURE 1. `distGraded` with `close_cutoff` = 10 and `mid_cutoff` = 20

```
    if( dist <= close_cutoff ){
        return( close_coef * dist )
    } else if( close_cutoff< dist & dist <= mid_cutoff ) {
        return( mid_coef * dist^2 )
    } else if( dist > mid_cutoff ){
        return( far_coef * dist^3 )
    }
    # should have already returned, but just in case
    return( 1 )
}
```

`distGraded` is intended to work for weighting nodes based on distance from a hostile agent's base-neighborhood, progressively making it more costly as the agent move farther away from their base, in such a way that at far distances, it will potentially dominate distance-based weights. There are infinitely many ways to go about assigning such weights, so we tend on the side of a reasonably simple and intuitively justifiable form. The distance weighting we use depends on three parameters: the *distance coefficient* $\beta \in [0, \infty)$; and three cut-offs $0 \leq \alpha_{\text{near}} < \alpha_{\text{far}} < \infty$. The *distance weight function* $f(x, \omega)$ takes in a node x, and a base neighborhood assignment ω (distances from neighborhoods certainly depend on which neighborhood we're referring to) is defined as

$$f(x) = \begin{cases} \beta_{\text{near}} \times \text{dist}_\omega(x) & 0 \leq \text{dist}_\omega(x) \leq \alpha_{\text{near}} \\ \beta_{\text{mid}} \times \text{dist}_\omega(x)^2 & \alpha_{\text{near}} \leq \text{dist}_\omega(x) \leq \alpha_{\text{far}} \\ \beta_{\text{far}} \times \text{dist}_\omega(x)^3 & \alpha_{\text{far}} \leq \text{dist}_\omega(x) \end{cases} \tag{3}$$

where $\text{dist}_\omega(x)$ is the distance from neighborhood ω to node x. The $\beta_{\text{near/mid/far}}$ values are all derivable from the parameters so that $\beta_{\text{near}} = \beta$ and the other two values chosen such that values agree at the boundaries: $\text{dist}_\omega(x) = \alpha_{\text{near}}$ or $\text{dist}_\omega(x) = \alpha_{\text{far}}$. Explicitly, $\beta_{\text{mid}} = \frac{\beta_{\text{near}}}{\alpha_{\text{near}}}$ and $\beta_{\text{far}} = \frac{\beta_{\text{mid}}}{\alpha_{\text{far}}}$.

To give some sense as to why this (mostly arbitrary) weighting is chosen, consider being placed at a node in the center of a large rectangular lattice. You want to decide where to move based on distance weights, so you decide to calculate weights which grow with distance with the end goal of

```
## Error in initFields(scales = scales):  could not find function "initRefFields"
```

FIGURE 2. Red line indicates the difference of weights `distGraded` (blue) and `dist1` (green) at same distances, illustrating the dominance in weights by `distGraded at large distances`.

inverting these weights to give you relative probabilities of jumping around between nodes. Drawing concentric circles (nodes of equal-distance) from you, consider how you may move between these circles. Roughly, the number of nodes on the circumference of each concentric circle is proportional to the distance from your node to that circle. So if we were to only weight by a linear term of distance, the total weight assigned to each concentric circle would be roughly constant, and be thusly translated into equal probability of you moving to into any of the concentric circles, regardless of the distance! While this might be standard for nearby concentric circles, it doesn't seem reasonable for distant ones, hence the cutoff α_{near}. After α_{near}, by scaling with the distance-squared, you are now in a realm so that the weights of each concentric circle are decreasing roughly proportional to $1/\operatorname{dist}$. Instead of considering movement between one concentric circle to the next, consider moving into two large regions bounded between two distant, but equally spaced concentric circles. Since the number of nodes within each of these interiors will roughly grow as the distance to the boundaries squared, if we were to remain at a distance-squared scaling, the probability of going into either of these regions again remains roughly constant (or worse, if the outer bounding circle moves much further away, the probability of jumping into the outer region will grow unboundedly as summing up the individual node contributions will result in a harmonic-like series). To mitigate this, we introduce the far cut-off α_{far} at which the distance-cubed scaling will not allow this other undesired behavior.

2.2.2. *Memory in Node Weights.* We define an *observation family* be a time-step indexed collection of sets $\mathcal{O} := \{\mathcal{O}_t\}_{t=0}^{\infty}$, with the index t representing the time-step with the observations took place. As a convention, we take $\mathcal{O}_0 = \varnothing$, since we have no records at or before time-step 0. Each observation family is intended to represent one type of observation made by agents whom it might affect; for example, one observation family could keep track of node locations where hostile agents have interacted with authority agents in such a way that $\mathcal{O}_t$ is a meaningful collection of these observations up to time-step t. Each *observation* $\mathbf{o} \in \mathcal{O}_t$ is a tuple $\mathbf{o} = (x, \omega)$, where x is a node on the playground lattice representing the observing agent's location during the observation (in fact, we take "location" to not only mean the specific node of the witnessing agent, but also the neighboring nodes as well, to relate the perception of a surrounding area of the observation), and ω

is some agent characteristic (e.g.: base neighborhood, family, authority-agent) such that all agents with a matching characteristic share the memory of these observations.

For clarity of exposition, we will return frequently to the example of hostile agents witnessing authority figures, with characteristic ω being base-neighborhood of the hostile witness. This means that if at time-step t, a hostile agent from neighborhood ω is at or neighboring node x and witnesses an authority agent, then $(x, \omega) \in \mathcal{O}_t$. Effectively, this also means that all hostile agents from ω share the memory of this observation.

At time-step $t+1$, the observations $\mathcal{O}_t$ are used to create "memory weights" on each node of the lattice, different for all affected agents following the intuitive pattern of memory: the weight of the memory is largest for the most recent observations and fades with time. Of course, there is an unlimited supply of functions to choose which could in someway mimic this intuitive behavior, and in reality, it is doubtful that any would deterministically grab the true, complex behavior of memory of such events. In light of this, we choose a mapping from observations to weightings which is reasonably simple, but flexible enough to easily maneuver between large ranges of memory behaviors, allowing us to consider sensitivity and robustness to such memory behaviors in the subsequent emergent behaviors within the ABM.

The *memory weighting function* $f(t, x, \omega) \in [0, \infty)$ (specific to one observation family $\mathcal{O}$) inputs a time-step t, node x, and characteristic ω, and outputs the weighting at node x for agents of characteristic ω during time-step t. This memory function is specified by two parameters: the *memory coefficient* $\beta \in [0, \infty)$ and *memory persistence* $\delta \in [1, \infty)$, and defined as

$$f(t+1, x, \omega) := \beta \sum_{s=1}^{t} \frac{1_{\mathcal{O}_s}(x, \omega)}{\delta^{t-s}} \tag{4}$$

where $1_{\mathcal{O}_t}$ is the indicator function

$$1_{\mathcal{O}_s}(x, \omega) = \begin{cases} 1 & (x, \omega) \in \mathcal{O}_s \\ 0 & \text{otherwise} \end{cases}$$

Note that these weights are all 0 at $t = 1$, since no observations have occurred yet by the initializing 0th time-step. Asymptotic trends to notice here are that if $\delta \to 1$, the weights do not fade in time whereas if $\delta \to \infty$, then the only significant weights are the most recent; hence δ behaves as an observation-family specific persistence of the memory of the observation. On the other hand, β acts as the observation-family specific weight per observation where if $\beta \to 0$, then observations of this type won't contribute much to the overall weight of a given node, whereas if $\beta \to \infty$, then these types of observations heavily weight a given node.

Example 2.2. Suppose that during time-steps 1 and 3, agents from base neighborhood ω witnessed authority agents in such a way that node x was the location of the observer or a neighboring node of the observer during each of the two observations. Then, $(x,\omega) \in \mathcal{O}_1$ and $(x,\omega) \in \mathcal{O}_3$. Also suppose that no authority observations involving node x were made by agents from ω occurred during time-steps 2 or 4; hence, $(x,\omega) \notin \mathcal{O}_2$ and $(x,\omega) \notin \mathcal{O}_4$. Taking $\beta = 3$ and $\delta = 2$, the weightings of node x from these observations for hostile agents from neighborhood ω calculated for the first few time-steps are

$$f(1,x,\omega) = 0$$

$$f(2,x,\omega) = 3 \times \sum_{s=1}^{1} \frac{1_{\mathcal{O}_s}(x,\omega)}{2^{1-s}} = 3 \times \Big(\underbrace{\frac{1}{1}}_{(1,x,\omega)\in\mathcal{O}_1} \Big) = 3$$

$$f(3,x,\omega) = 3 \times \sum_{s=0}^{2} \frac{1_{\mathcal{O}_s}(x,\omega)}{2^{2-s}} = 3 \times \Big(\underbrace{\frac{1}{2}}_{(x,\omega)\in\mathcal{O}_1} + \underbrace{\frac{0}{1}}_{(x,\omega)\notin\mathcal{O}_2} \Big) = \frac{3}{2}$$

$$f(4,x,\omega) = 3 \times \sum_{s=0}^{3} \frac{1_{\mathcal{O}_s}(x,\omega)}{2^{3-s}} = 3 \times \Big(\underbrace{\frac{1}{4}}_{(x,\omega)\in\mathcal{O}_1} + \underbrace{\frac{0}{2}}_{(x,\omega)\notin\mathcal{O}_2} + \underbrace{\frac{1}{1}}_{(x,\omega)\in\mathcal{O}_3} \Big) = 3 + \frac{3}{4}$$

△

Remark 2.1. While varying the parameters within our memory weight function allows for a variety of memory behavior (persistent, quick but short-lived, etc), the exponential decay has a computational benefit as well. Indeed, notice that

$$f(t+1,x,\omega) = \beta \times 1_{\mathcal{O}_t}(x,\omega) + \frac{1}{\delta} \times f(t,x,\omega) \tag{5}$$

giving a time-step recursive formula which allows us to use the previously calculated weights along with only a few new calculations to update the current weights, rather than recalculating the entire sum each time-step. △

As mentioned above, for the case of hostile agents observing authority agents, the observers location and neighboring nodes are all considered relevant for the observation. Symbolically, this means that if x is the node of an observer from ω at time-step t, then $(x,\omega) \in \mathcal{O}_t$ and $(y,\omega) \in \mathcal{O}_t$ for each node y which is a neighbor of x. Let us agree to call these relevant nodes *hot nodes* for this observation type. Along with witnessing authority agents, currently built into the ABM is

the functionality for hostile agents to also give weights to nodes based on observing enemy hostile agents in the analogous way. As for authority agents, they also have a weighted lattice where the weightings in this case are determined by the victim of a hostile action between hostile groups. In this case, the hot nodes will be determined by the node of the victim of the act and the neighboring nodes.

As is reasonable, authority agents will want to move towards areas of hostilities to try to prevent further such actions. On the other hand, hostile agents will eschew nodes where they have encountered authority agents (and potentially where they have encountered enemy hostile agent nodes). Hence the heavier weights from hostilities will attract authorities, whereas heavier weights from hostile agents experiencing negative encounters will detract them from these nodes. This is taken care of when combining the weights to create a PMF for movement as generally described above.

Part 3. Dominance and Attacks

Within the context of this ABM, *dominance* is a family-level pecking order, with inter-family relative dominance depending on hostilities between agents of those respective families. We will consider an *attack* as family-level, tension-increasing, exchange of dominance. A bit more explicitly, if one family attacks another during a timestep, the outcome of the attack is that the tension between all neighborhoods of these two family increases depending on the severity of the attack, while the attacking family will gain dominance relative to the attacked family, where the gain of dominance also depends on the severity of the attack. Whether or not an attack happens depends on the dominance gap (the difference in dominance) between the two families entering into the timestep, where one family, the *attacker*, is more likely to attack another, the *potential attackee*, when the potential attackee family has a larger dominance over the attacker.

Computationally, this comes down how to assign and tally dominance during each timestep depending on the interactions between hostile agents, done via the dynamic function variable `DOM_MATRIX_ENTRY` to be stored in the matrix `DOMINANCE_MATRIX`; how to use this tally to decide whether or not to attack, done via the dynamic function variable `ATTACK`; and from there, if an attack occurs, how to choose the severity of the attack, done via the dynamic function variable `ATTACK_SEVERITY`.

3. Tallying Dominance: `DOM_MATRIX_ENTRY`

The rows and columns of the square matrix `DOMINANCE_MATRIX` are indexed by the families within the ABM, where the row i column j entry represents the dominance felt by i towards j. A positive number n in the ijth entry represents a dominance gap of n in i's favor, whereas a negative entry

$-n$ represents a dominance gap of n in j's favor. Hence a negative value in the ijth entry should results in a desire for i to attack j to try either decrease the dominance gap, or potentially create a gap in i's favor.

The dynamic function variable `DOM_MATRIX_ENTRY` is called at each timestep and is used as the calculator to assign a dominance value. It is used to update the entries in `DOMINANCE_MATRIX` and called one entry at a time.

- `outward` A non-negative integer value. If during the call for the ijth entry in `DOMINANCE_MATRIX`, this value represents the number of hostilities i has imparted on j.
- `inward` A non-negative integer value. If during the call for the ijth entry in `DOMINANCE_MATRIX`, this value represents the number of hostilities j has imparted on i.
- `old_dom` A numeric value. If during the call for the ijth entry in `DOMINANCE_MATRIX`, this is the previous value of the ijth entry.

3.1. **Built-in Functionality.** Out of the box, we use the function `DomEntryCalculator` as our selection of infinitely many possible realizations of `DOM_MATRIX_ENTRY`, with inputs

- `outward` Same as for `DOM_MATRIX_ENTRY`.
- `inward` Same as for `DOM_MATRIX_ENTRY`.
- `old_dom` Same as for `DOM_MATRIX_ENTRY`.
- `max_dom` The maximal value we are willing to assign to a dominance gap. Default set to `Inf`.
- `min_dom` The minimal value we are willing to assign to a dominance gap. Default set to `-Inf`.

This simple function updates `old_dom` via net difference between `outward` and `inward`, making sure to respect the maximal and minimal values if assigned.

```
DomEntryCalculator <- function( outward,
                                inward,
                                old_dom,
                                max_dom = Inf,
                                min_dom = -Inf ){
    dom <- min( max_dom, max( min_dom, outward - inward + old_dom ) )
    return( dom )
}
```

4. Who to Attack: ATTACK

The dynamic function variable `ATTACK` inputs a single character string `attacker_fam`, indicating the family name under consideration as the attacker, and will output either a `NULL` value, indicating that `attacker_fam` will attack no other families during the timestep of the call, or a character vector of family names indicating which families will be attacked by `attacker_fam` during this timstep.

Example 4.1. Suppose we call `ATTACK( attacker_fam = fam )` and are returned the character vector `c('fam2', 'fam3')`. This indicates that family `fam` will attack both `fam2` and `fam3` during this timestep. Alternatively, if the returned output is `NULL`, then `fam` will not attack any other family during this timestep. △

4.1. Built-in Functionality.

The auxiliary function `AttackDecision` has inputs

- `hfam` A character string with the name of the family who will be considered the attacker.
- `will_attack_at` A positive numeric value, representing the max dominance gap before forcing an attack.
- `allow_multiple_attacks` A logical value. If set to `T`, the return vector may include more than one other family for `hfam` to attack; otherwise, the return value will have at most one family name to attack.

The way this function decides which, if any, families `hfam` will attack is by first considering which families `hfam` feels has dominance over them; effectively, those potential attackee families with dominance over `hfam` are the column indices for which there is a negative entry in the row of `DOMINANCE_MATRIX` corresponding to `hfam`. If the dominance gap of these dominations is at least `will_attack_at`, then the potential attackee family will become a candidate for attack. If the dominance gap of a potential attackee is less than `will_attack_at`, say it is some value k, then the probability that the potential attackee will become a candidate for attack is $p = \frac{k}{\texttt{will\_attack\_at}}$. Finally, if there are no candidates for attack, a value `NULL` is returned (indicating that `hfam` will not attack any other family); if `allow_multiple_attacks` is `T`, then a character vector with all candidates will be returned (meaning `hfam` will attack all candidates); otherwise, if `allow_multiple_attacks` is `F`, only one of the candidates will be selected uniformly at random to be returned.

Based on this discussion, a trend to note is that as `will_attack_at` $\to \infty$, attacks will become increasingly rare. Alternatively, as `will_attack_at` $\to 0$, attacks will become increasingly prevalent.

Putting this together, the realization of `ATTACK` we use out-of-the-box is

```
ATTACK <- function( attacker_fam, ... ){ AttackDecision( attacker_fam,
                                                         will_attack_at = 10,
                                                         allow_multiple_attacks = T ) }
```

5. Severity of Attack: ATTACK_SEVERITY

While interactions between agents are not part of an attack within this ABM, the outcome of an attack is sensibly tallied as a type of hostile act from the attacker family towards the attackee family. Keeping this in mind, they dynamic function variable `ATTACK_SEVERITY` inputs a integer value `dom`, representing the current domination perceived by the attacker family towards the attackee family, and will output a non-negative integer representing how many standard hostilities the attack imparted from the attacker family towards the attackee family is worth; this value will be called the *severity* of the attack. The severity of an attack is family-specific, but not neighborhood-specific, meaning that it will affect the tension between all neighborhoods belonging to the involved families.

Example 5.1. Suppose during `timestep` it happens that `fam1` attacks `fam2` with the output of `ATTACK_SEVERITY` being 3 for this attack. Suppose also that `nbhd1` and `nbhd2` are the neighborhoods belonging to `fam1` and `nbhd3` is the only neighborhood belonging to `fam2`. Then during `timestep`, as hostilities are tallied during hostile interaction updates, there will be 3 additional hostilities directed from `fam1` towards `fam2`; importantly this includes the effect this will have on their relative dominance gaps, which was the supposed impetus behind attack in the first place. As the tension is updated during `timestep`, the tension between `nbhd1` and `nbhd3` will be updated as if `nbhd1` had 3 more hostilities towards `nbhd3` during `timestep`, and similiarly for `nbhd2` towards `nbhd3`. △

5.1. **Built-in Functionality.** The out-of-the-box auxiliary function `AttackSeverityCalculator` for `ATTACK_SEVERITY` has inputs

- `min_severity` A non-negative integer value representing the minimal severity imparted from the attacker family towards the attackee family.
- `max_severity` A non-negative integer value representing the maximal severity imparted from the attacker family towards the attackee family.
- `old_dom_gap` A non-negative integer value representing the domination gap between the attacker and attackee at the time just before the attack.

- `escal_quant` An integer value representing the severity escalation value during attack, where a positive value represents an attempted escalation (gaining dominance over) the attackee.

The output of `AttackSeverityCalculator` is an integer value between `min_severity` and `max_severity`, easily understood from the code.

```
AttackSeverityCalculator <- function( min_severity,
                                      max_severity,
                                      old_dom_gap,
                                      escal_quant = 1 ){

     val <- max( min_severity, min( old_dom_gap + escal_quant, max_severity ) )
     return( val )

}
```

Example 5.2. Suppose that `fam1` attacks `fam2` and the value of `DOMINANCE_MATRIX[fam1,fam2]` is -3 when the decision to attack is made. We then have `old_dom_gap` is equal to $|-3| = 3$ for this attack. If further we set `min_severity = 2`, `max_severity = 6`, and `escal_quant = 1`, then the output of `AttackSeverityCalculator` for this attack will be `old_dom_gap` + `escal_quant` $= 4$. Hence, *pending there are no other hostilities between* `fam1` *and* `fam2` *during this timestep*, the updated value of `DOMINANCE_MATRIX[fam1,fam2]` will be $-3 + 4 = 1$, now favoring `fam1` (family `fam1` has "escalated" the situation by switching the direction of the domination by the quantity `escal_quant`). On the other hand, if at the time the decision was made, the value of `DOMINANCE_MATRIX[fam1,fam2]` is -7, say, then (with `min_severity`, `max_severity`, and `escal_quant` as before) the output of `AttackSeverityCalculator` would be 6 rather than 8 as the max value allowed is set by `max_severity`. △

Putting this together using out-of-the-box defaults,

```
ATTACK_SEVERITY <- function( dom ){
     AttackSeverityCalculator( min_severity = 2,
                               max_severity = 6,
                               old_dom_gap = abs(dom),
                               escal_quant = 1 )
```

```
}
```

Part 4. Authority Agent Movement

Part 5. Agent Interaction