Netlogo 4.0.3 Useful Code

ARCHITECTURE OF PROGRAM

• **clearing world** (clear-all, clear-turtles, clear-patches, clear-output)

Typically called at the beginning of a program. Clear-all resets all global variables to zero; also clears out turtles, patches, and output. Other commands clear subsets of the model.

```
to setup
        clear-all ;; clears  entire world, usually called at beginning of setup command
end
```

• **creating agents** (create, hatch, sprout)

All create new agents, though from different contexts. The “create” command all called by the observer, the “hatch” command is called by the turtles, and the “sprout” command is called by patches. Turtles “hatched” have features identical to their predecessor. Note that, when using breeds, one can modify these commands to create a specific breed.

examples:

```
ask n-of 10 patches [ sprout 1 ] ;; ask 10 random patches to sprout 1 turtule

ask turtle 0 [ hatch 1 ] ;; ask turtle 0 to “hatch” a new turtle

create-turtles 10 ;; creates 10 turtles
create-wolverines ;; creates 10 turtles of breed wolverine
```

note that you can immediately assign turtle characteristics and issue commands:

```
breed [wolverines wolverine] ;; define a breed called “wolverines”

to setup
        clear-all ;; clears world
        create-wolverines 50 [ set color blue ] ;; creates 50 wolverines
        ask n-of 25 wolverines [ set shape “person” ] ;; ask 1/2 of wolverines to
                                                      ;; take on the “person” shape
        ask wolverine 4 [ hatch 1 ]  ;; asks wolverine with ID=4 to hatch 1 new
                                     ;; wolverine
end
```

• **killing agents** (die)

Self explanatory. Removes one or more agents.

```
ask turtle 4 [ die ]
```

• **stopping model** (stop)

The agent immediately exists from the current procedure or “ask” command. Stop can also end a procedure controlled by a forever button.

```
if not any? turtles with [ happy = true ] [ stop ] ;; exits if there are no happy turtles
```

• **loops & conditional statements** (if/else, if, while, foreach, repeat)

Control whether and how many times something happens.

IF

if [some condition is met] [do something]

```
ask wolverines [  ;; get agent set of all wolverines
        if color = blue ;; if own color = blue
        [ set pcolor [color] of self ]  ;; set patch currently on to be blue too
]
```

IF/ELSE

ifelse [some condition is met]
[if true, do something]
[if not true, do something else]

```
 ask wolverines [  ;; get agent set of all wolverines
        ifelse shape = “person” ;; if wolverine is shaped like person
        [ set pcolor [color] of self ]  ;; IF condition met, set patch to be blue
        [ set pcolor yellow ]  ;; ELSE set patch color to yellow
]
```

WHILE

while [some condition is true] [keep doing whatever is in these brackets]

```
while [ ticks < 100 ] ;; while tick count is less than 100
        [ go ] ;; run command “go”
```

FOREACH

foreach [item in some list] [do something with that item]

```
let my-turtle-list (list  (turtle 1) (turtle 2) (turtle 3) )
let my-patch-list (list (patch 4 5) (patch 5 5) (patch 6 5))

foreach my-turtle-list [ ask ? [ fd 2 ] ] ;; ask each turtle in list to move forward 2 steps

( foreach my-turtle-list my-patch-list [ ask ?1 [ move-to ?2 ] ]) ;; moves turtle to item in same
                                                                    ;; position on patch list
```

• **setting variable values** (let, set)

The “let” command creates a new local value with a given value. If you want to later change the value of any variable, you use the “set” command.

```
let finished? true
type finished? ;; report value of finished? to output screen
set finished? false
```

• **printing output to screen** (print, type, show)

The “print” command prints some value to the screen, followed by a carriage return. The “type” command also prints a value to the screen, but does not include a carriage return. The “show” command also prints a value (followed by a carriage return), but includes the identifying features of the calling agent.

```
ask turtle 5 [ print color ]
        > 105
ask turtle 5 [ print color type " and i am a " print breed ]
        > 105
        > and i am a wolverines
ask turtle 5 [ show color ]
        > (wolverine 5): 105
```

CODE FOR TURTLES AND PATCHES

• **movement** (back, forward, setxy, move-to, towards, towardsxy, jump, uphill)

There are different ways of moving turtles around the lattice. Here are a few examples. Note that turtles can move anywhere on a patch. If you want them in the center of their patch, you can add *move-to patch-here*, which will center them on their current patch.

```
ask turtles [ fd 10 rt 30 fd 10 ] ;; ask all agents to move forward 10 steps, turn right 30
                                    ;; degrees, and then forward another 10 steps
```

```
ask turtle 0 [ setxy random-xcor random-ycor ]  ;; move turtle 0 to random x,y coordiante

ask turtles with [ hat = “pink” ] [ move-to one-of patches ] ;; move turtles with pink hats to
                                                    ;; center of random patch

ask turtles [ if  any? turtles-here [ jump 5 ] ]  ;; ask turtles who currently share a patch with
                                                    ;; another turtle to jump forward
```

• **neighbors** (neighbors, neighbors4, in-cone, radius)

There are different ways of specifying neighborhoods around turtles or patches. “Neighbors” returns an agentset containing the 8 surrounding patches. “Neighbors4” returns the four north/south/east/west facing neighbors.

```
show sum [count turtles-here] of neighbors  ;; prints the total number of turtles on
                                                    ;; the eight patches around the calling
                                                    ;; turtle or patch
show count turtles-on neighbors  ;; a shorter way to say the same thing
ask neighbors4 [ set pcolor red ]  ;; turns the four neighboring patches red
```

The “in-cone” neighborhood returns a neighborhood representing some “cone of vision” for a given agent. The cone is defined by the two inputs, the vision distance (radius) and the viewing angle. The viewing angle may range from 0 to 360 and is centered around the turtle's current heading. (If the angle is 360, then in-cone is equivalent to in-radius.)

```
ask turtles [
    set heading 90 ;; direction the turtle is facing
    ask patches in-cone-nowrap 3 60 [ set pcolor grey]
      ;; gives turtles a "cone of vision" in front of themselves
      ;; if the angle is 360, then in-cone is equivalent to in-radius
 ]
```

The “radius” neighborhood returns all patches within some radius of the calling agent. The “nowrap” condition ensures that agents at the edges of the world don’t “wrap around” to collect neighbors on the opposite edge.

```
ask turtles [
    ask patches in-radius-nowrap 1 [ set pcolor grey]
    ]
```

• **self- or other agent reference** (myself, nobody, turtles-at, turtles-on, turtles-here, any?, all?, n-of, max-n-of)

There are a number of different ways of referring to specific turtles or patches, and groups of turtles or patches. Some examples follow below.

SELF

“Self” refers to the agent who is calling the command. It’s easy to confuse “self” and “myself.” Here’s one way of using self.

```
ask turtles
[
  let ME self
;; since ME is local, is can be used directly by any code inside
;; this set of brackets (the brackets for "ask turtles")
 ask patches with [ distance ME > 10 ] [ set pcolor “red” ]
]
```

MYSELF

Refers to whatever turtle or patch is issuing some command. Often used with “of” to determine some relationship between the asking agent and other agents or the environment.

```
ask turtles
 [ ask patches with [distance myself > 10 ] [ set pcolor red ] ;; does same as code for “self”
]

 [ set pcolor [color] of myself ] ] ;; each turtle makes a colored "splotch" around itself]
```

NOBODY

Value that indicates if no agent was found.

```
set other one-of other turtles-here ;; set variable “other” to be one of turtles on caller agent’s
                                                                    ;; patch
if other != nobody ;; if “other” is actually a turtle
 [ ask other [ set color red ] ] ;; turn it red
```

ANY? or ALL? <agentset>

“Any” returns the value “true” if some condition is met for at least one agent in the given agentset. “All” returns the value “true” if some condition is met for all of the agents in a given agentset.

```
if all? turtles [color = red]
  [ type "every turtle is red!" ]

if any? turtles [ color = red]
[ type “at least one turtle is red!” ]
```

TURTLES-AT <dx dy> or TURTLES-ON <agent/agentset> or TURTLES-HERE <agent/agentset>

“Turtles-at <dx dy>” reports all turtles within some distance determined by dx dy.

```
create-turtles 5 [ setxy 2 3 ] ;; create 5 turtles, and set their x,y coords to 2,3
show count [turtles-at 1 1] of patch 1 2 ;; show how many turtles are within 1
                                          ;; x-coordinate and 1 y coordinate of patch 1 2
```

“Turtles-on” reports all turtles that are on a given patch or set of patches, or on the same patches as a given turtle or set of turtles.

```
ask turtles [
  if not any? turtles-on patch-ahead 1 ;; if no turtles on the patch that is 1 patch
                                       ;; ahead
  [ fd 1 ] ;; move forward to that patch
]

ask turtles [
  if not any? turtles-on neighbors [ ;; if not any turtles on the 8 contiguous neighbors
    set isolated? true ;; set variable isolated to be true
  ]
]
```

“Turtles-here” reports all turtles on the same patch as the calling turtle.

```
crt 10 ;; create 10 turtles -- all start on the 0 0 patch
ask turtle 0 [ show count turtles-here ] ;; ask turtle 0 to count how many turtles on its patch
```

> 10

N-OF <number> & MAX-N-OF <number> <agentset> [reporter]

The command “n-of” takes as its argument an agentset, and reports an agentset of size size randomly chosen from the input set, with no repeats.

```
ask n-of 25 turtles [ set shape “butterfly” ] ;; ask 25 turtles to set their shape to butterfly
```

The command max-n-of returns an agentset containing number agents from agentset with the highest values of reporter. The agentset is built by finding all the agents with the highest value of reporter, if there are not number agents with that value then agents with the second highest value are found, and so on. At the end, if there is a tie that would make the resulting agentset too large, the tie is broken randomly.

```
;; assume the world is 11 x 11
show max-n-of 5 patches [pxcor]
;; shows 5 patches with pxcor = max-pxcor
```

```
show max-n-of 5 patches with [pycor = 0] [pxcor]
;; shows an agentset containing:
;; (patch 1 0) (patch 2 0) (patch 3 0) (patch 4 0) (patch 5 0)
```

• **spread** (diffuse)

When studying the “spread” of some process or resource, it may be useful to use the “diffuse” command. This tells each patch to give equal shares of (*number* * 100) percent of the value of patch-variable to its eight neighboring patches. *number* should be between 0 and 1. Regardless of topology the sum of patch-variable will be conserved across the world. (If a patch has fewer than eight neighbors, each neighbor still gets an eighth share; the patch keeps any leftover shares.) Note that this is an observer command.

```
diffuse pcolor 0.5
;; each patch diffuses 50% of its variable
;; color to its neighboring 8 patches. Thus,
;; each patch gets 1/8 of 50% of the color
;; from each neighboring patch.)
```

• **coordinates** (max-pxcor, max-pycor, random-pxcor, random-pycor)

Patches are identified by their variables pxcor and pycor. However, there are also commands to pull up specific (or random) coordinates. The commands “max-pxcor” and “max-pycor” return the maximum x and y coordinates, respectively. The commands random-pxcor and random-pycor return random x,y coordinates.

• **distances** (distance <agent>, distancexy)

Determines the distance between the calling agent and some target. The command “distance” reports the distance from the calling agent to the given turtle or patch. The command “distancexy” reports the distance from this agent to the point (xcor, ycor).

```
ask turtles [ show max-one-of turtles [distance myself] ]
;; each turtle prints the turtle farthest from itself

ask turtle 0 [ show distancexy 0 0 ] ;; turtle 0 reports distance from origin patch
```

FILES, PLOTTING, MOVIES

• **files** (file-read, file-open, file-write, file-delete, file-close)

It is often useful to be able to read and write data from files. Netlogo has a number of primitives for managing files. An example follows below.

```
to test-files
        set-current-directory user-directory ;; allow user to select directory for files
        file-open "location.txt"  ;; Opening file for writing
        ask turtles
        [ file-write xcor file-write ycor ] ;; ask turtles to write their coordinates out to file
         file-close

        file-open "location.txt"  ;; Opening file for reading
        ask turtles
         [ setxy file-read file-read ] ;; ask turtles to read their coordinates from file
         file-close

        show file-exists? "location.txt" ;; reports true if file exists
        file-delete "location.txt"
        show file-exists? "location.txt" ;; should now report false
end
```

• **making movies** (movie-start, movie-grab-view, movie-close)

To make a quicktime movie, the bare minimum is that you must name it, tell Netlogo how many “frames” of the program to grab, and close it at the end.

Note: these Quicktime movies are not compressed, so they can quickly become huge!

```
;; export a 10 frame movie of the view
setup
        movie-start "abm.mov" ;; setup movie and name it
        movie-grab-view ;; show the initial state
        repeat 10 ;; repeat what's in brackets 10 times
        [ go ;; run the go procedure
         movie-grab-view ]
        movie-close
```

• **plotting results** (set-current-plot, set-current-plot-pen, plot)

If you have only 1 plot, you can start plotting on it using the “plot” command. However if you have multiple plots, you need to first tell Netlogo which plot to plot to using the “set-current-plot” command. Each line on a plot is drawn by a “pen.” With one line, you have only 1 pen. But if you want to plot multiple lines, you need to specify which output corresponds to which pen. See examples below.

```
to setup-plot
        set-current-plot "Turtle 0 Position" ;; identify which plot to use
end
```

```
to update-plot
        set-current-plot-pen "Xcor" ;; use the pen labeled “Xcor”
        plot [xcor] of turtle 0          ; plot xcor of turtle 0
        set-current-plot-pen "Ycor" ;; use the pen labeled “Ycor”
        plot [ycor] of turtle 0          ; plot ycor of turtle 0
end
```

MATHEMATICAL OPERATORS AND RANDOM NUMBERS

• **random numbers** (random, random-normal, random-poisson, new-seed, random-seed, random-float, random-exponential)

Random numbers in Netlogo are actually “psuedo-random.” This means that they appear random, but are actually created via a deterministic process. The sequence of random numbers created by the Netlogo random number generator are determined by a random “seed.” Once the generator has been seeded with the random seed, it always generates the same sequence of numbers. For example, the lines of code below will always generate the numbers 95, 7, and 54 in that order.

```
random-seed 137
show random 100
show random 100
show random 100
```

If you don’t seed the random number generator yourself, Netlogo will create a seed using the current date and time. The functions “random,” “random-normal,” “random-poisson,” “random-float,” etc. all return random numbers of the appropriate distribution. See examples below:

```
random 10 ;;  returns integer greater than or equal to zero, but less than 10
random-float 1 ;; returns floating-point number greater than or equal to zero, but less than 1
random-mean 0 5 ;; returns normally distributed value with mean 0 and SD 5
```

functions (ceiling, floor, int, min, max, mean, standard-deviation, round)

There are a number of functions already built into Netlogo as primitives. A few examples follow below:

```
show ceiling 4.5 ;; reports smallest integer greater than or equal to a given number
        >  5
show floor 4.5 ;; reports largest integer less than or equal to a given number
        > 4
round 3.7555592 ;; rounds a given number to the nearest integer
        > 4
show mean [ 2 6 12 9 10 12 16 ]
```

```
        > 9.571428571428571
show mean [ who ] of turtles ;; reports mean ID number of turtles
```

arithmetic operators (+, -, ...)

You can use a wide range of arithmetic operators in Netlogo. Just make sure that you put spaces in between operators and numbers, as shown below.

```
show (5 + 5) /  2
      > 2
```