**Working with Matlab**

To be able to work with L2, it is important to be familiar with the Matlab environment. If you have never worked with Matlab before, you should take some time to learn the basics of Matlab before continuing with this document. Bundled with L2 is the Matlab Primer for Matlab 2014. As most of the basic functionality has not changed recently, this guide can also be used for older versions of Matlab. Below is an overview of the relevant chapters to be able to work with L2.

**Chapter 1: Quick Start** *(p. 11-40)*

* + *Desktop Basics*
  + *Matrices and Arrays*  
    ***not obligatory:*** *Matrix and Array Operations, Complex Numbers*
  + *Character Strings*
  + *Calling Functions*
  + ***not obligatory:*** *2-D and 3-D Plots*
  + *Programming and Scripts*
  + *Help and Documentation*

**Chapter 2: Language Fundamentals** *(only p. 60-78)*

* + *Indexing*
  + *Types of Arrays*

Chapter 3: Mathematics *(****not obligatory****)*

Chapter 4: Graphics *(****not obligatory****)*

**Chapter 5: Programming** *(only p. 199-206)*

* + *Control Flow*

*Note, the distinction between arrays, cell arrays and structures is very important in L2. If you are not comfortable with these different data structures, please re-read Chapter 2, Types of Arrays and practice creating, accessing and manipulating them.*

**Installing L2**

Unzip L2.zip to some folder on your computer. In Matlab, navigate to this folder.[[1]](#footnote-1) Any models should be added to separate folders in the same directory.

**Folder structure**

working\_dir

* + @l2
  + model
    - *sorts.l2*
    - *predicates.l2*
    - *scenarios.l2*
    - *paramaters.l2*
    - *rules.m*
  + model
    - *…*
  + …

working\_dir Your Matlab working directory

@l2 Folder containing the l2 class files

model Name of your model; a folder containing files described below.

**Defining models in L2**

**sorts.l2**

sort; {value, <predicate>}sort; {value, value, value}  
…

sort Name of the sort  
value Possible value for the sort

<predicate> A predicate that can be nested as a value for sort

*Note, the sort* REAL *is defined by default containing all real values, as well as* BOOLEAN *(true, false).*

**predicates.l2**

predicate; sortpredicate; {sort, sort}  
…

predicate Name of the predicate  
sort Sort used in the predicate, as defined in sorts.l2

**scenarios.l2**

scenario (  
 predicate; time; value  
 predicate; time; {value, <predicate{…}>}  
 …  
)

scenario (  
 …  
)  
…

scenario Name of the scenario  
predicate Name of the predicate, as defined in predicates.l2  
time Timepoints the following value will be true

* x a single time point x
* [x:y] from timepoint x until y
* [x y ..] for timepoints x, y, etcetera

value The value of predicate, adhering to the required sorts and their values as defined in predicates.l2 and sorts.l2.

<predicate{…}> A nested predicate, with its values between {…}.

*Note, if only one scenario is defined, its name and brackets can be left out and will be set to ‘default’.*

**parameters.l2**

set (  
 parameter; value  
 parameter; {value, value}  
 …  
)

set (  
 …  
)  
…

set Name of the set of parameters

parameter Name of the parameter,

value The value for the parameter

*Note, if only one set of parameters is defined, its name and brackets can be left out and will be set to ‘default’. Each parameter needs to be defined in each set, the values given can differ.*

**rules.m**

function [ fncs ] = rules()

%DO NOT EDIT

fncs = l2.getRules();

for i=1:length(fncs)

fncs{i} = str2func(fncs{i});

end

end

%ADD RULES BELOW

function result = rule( trace, params, t )

%some calculation of new value

result = {'predicate', time, value};

end

…

function [ fncs ] = rules() … end

Always add this function exactly as written here to your rules.m file.

function result = rule( trace, parameters, t )

%some calculation of new <value>

result = {'predicate', time, value};

end

rule The name of your rule.

predicate The predicate as defined in predicates.l2 you are setting a new value for.

time The time point this value is true for, usually t+1.

value The value.

*Note, you can return multiple results in one function by creating a cell array of results.*

**Defining rules**

In each rule you have access to the current time step *t,* the trace *trace* up to and including the current time step *t*, and the parameters in *parameters.*

Trace is a named structure array with fields corresponding to the names of the predicates as defined in predicates.l2. Each row is a particular time point in the trace.

* Predicates containing a single real value will be stored as an array of reals for each time point.
* Predicates containing a single boolean value will be stored as an array of logicals for each time point.
* Predicates containing a single other sort will be stored as an cell array of strings[[2]](#footnote-2).
* Predicates containing multiple sorts will be stored as an cell array of cells, with each cell containing the values for that predicate.

Thus, retrieving data depends on the type of data that is stored for a particular predicate. Some examples of retrieving data:[[3]](#footnote-3)

>> trace(1)

ans =

‘predicate’: {value1\_t1}

‘predicate2’: {value1\_t1 value2\_t1}

A structure array with fields corresponding to the predicates and their values for time point 1.

>> trace(1).predicate

ans =

{value1\_t1}

The values of predicate for time point 1 as an array of reals/logicals or a cell array.

>> trace.predicate

ans =

{value1\_t1}

ans =

{value1\_t2}

One by one, for each time point the value for predicate. Thus, if your trace is 5 time points long, this query gives you 5 answers.

>> {trace.predicate}

ans =

{value1\_t1 value1\_t2}

As above, but each time point is put in a cell array, such that the first cell is time point 1, the second cell time point 2 etcetera.

>> trace(1).predicate(1)

ans =

{value1\_t1}

Retrieve the first value for predicate at time point 1, which is a real, logical or a cell.

>> trace(1).predicate{1}

ans =

value1\_t1

In the case that predicate is stored as a cell array, this gives the contents of the first cell at the first time point.

>> trace(1).predicate{:}

ans =

value1\_t1

ans =

value1\_t2

One by one, the contents of each cell for predicate at time point 1.

Parameters is also a named structure array, with each field being a parameter name. There are however no rows for different time points, as parameters are assumed to be constant. Accessing parameters is similar to accessing values in the trace.

Using these values, one or more new values can be calculated in each rule. A single result can be returned as a cell array with the following values; predicate, time point, value. Multiple results can be returned by putting each of those cells in a single cell array.

>> result = {‘predicate’, t+1, value};

Adding value for predicate at the next time point.

>> result = {‘predicate’, [t+1:t+3], value};

Adding value for predicate for the next three time points.

>> result = {{‘predicate’, t+1, value} {‘predicate2’, t+1, value2}};

Adding value for predicate and value2 for predicate2 at the next time point.

>> result = {};

>> for …

>> …

>> result = {result{:} {‘predicate’, t+1, value}};

>> end

Creating a cell array of results in a for loop (or any other type of control flow) by concatenating each new result with all previous results. Similarly, you can use the less intuitive but faster line to concatenate the cell arrays. Notice the extra set of curly brackets around the new result!

>> result = [result { {‘predicate’, t+1, value} }];

**Running a model**

Before we can use the model, first the l2-object needs to be created in Matlab.

>> model = l2(‘folder’);  
Create a l2-object *model* as defined by the files in folder.

>> model.reset();  
Reload the l2 files. Necessary after having made any changes in any of those files.

>> model.setModel(‘folder’);  
Change the ‘folder’ and load the l2 files in that folder.

We can now run the simulation of that model as follows.

>> model.simulate(100);  
Run the simulation for 100 time points, using the default scenario and default parameters.

>> model.simulate(100, ‘scenario’);  
Run the simulation for 100 time points, using the scenario ‘scenario’ and default parameters.

>> model.simulate(100, ‘scenario’, ‘parameters’);  
Run the simulation for 100 time points, using the scenario ‘scenario’ and parameters ‘parameters’.

Afther having ran the simulation, we can plot the trace or plot and save the trace as pdf file.

>> model.plot();  
Plot the entire trace file.

>> model.plot({‘predicate’ ‘predicate2’});  
Plot only ‘predicate’ and ‘predicate2’ of trace file.

>> model.plotFile(‘file’), model.plotPDF(‘file’, {…});  
Similar as plot, but saves a .png of the plot named ‘file’ in a folder plots in the model-folder.

**Example 1: Leadsto tutorial**

Consider an experimental setting, involving two positions (say, p1 and p2), an animal, a piece of food, and a transparent screen (e.g., a window). Suppose the animal is placed at position p1, the food is placed at p2, and the screen is placed in between, separating the animal from the food. Multiple trials are performed in which, at some variable moment, the screen is raised, and the animal is free to go to any position. After a number of trials, it turns out that a regularity can be observed in the behaviour of the animal. This regularity can be expressed informally by the following property:

*Every time that the agent observes that there is food at p2,  
 and no screen is present, it will go to p2.*

In semi-formal form, this property can be written as follows:

**LP1 (Local Property 1)**at any point in time,

if the agent observes that food is present at position p2   
and it observes that no screen is present,   
then it will go to position p2

To implement this model, we create the following files.

**sorts.l2**

The only sort we will use is BOOLEAN, which is defined by default. Therefore, there is no need for a sorts.l2 file.

**predicates.l2**

goes\_to\_p2; BOOLEAN  
observes\_food\_at\_p2; BOOLEAN  
observes\_no\_screen; BOOLEAN

These are the three predicates used in LP1, each of them using a boolean value.

**scenarios.l2**

default (  
 observes\_no\_screen; 3; true  
 observes\_food\_at\_p2; [1:3]; true  
)

no\_food (  
 observes\_no\_screen; 3; true  
)

Here we define our scenarios, in this case two. In the first scenario the animal observes food at time points 1, 2 and 3 and no screen at time point 3. In the second scenario, the animal observes no screen on time point 3, but does not observe any food.

**parameters.l2**

This example does not make use of any parameters.

**rules.m**

function [ fncs ] = rules()

%DO NOT EDIT

fncs = l2.getRules();

for i=1:length(fncs)

fncs{i} = str2func(fncs{i});

end

end

%ADD RULES BELOW

function result = lp1( trace, parameters, t )

move = trace(t).observes\_food\_at\_p2 & trace(t).observes\_no\_screen;

result = {'goes\_to\_p2', t+1, move};

end

Our rules.m file only contains one rule, namely lp1. Here, we check whether both observes\_food\_at\_p2 and observes\_no\_screen are true at time point t in the current trace. The comparison tells us whether the animal goes to p2 at the next time step, thus this is our result for goes\_to\_p2 at time point t+1.

**Running the example in Matlab**

clear all

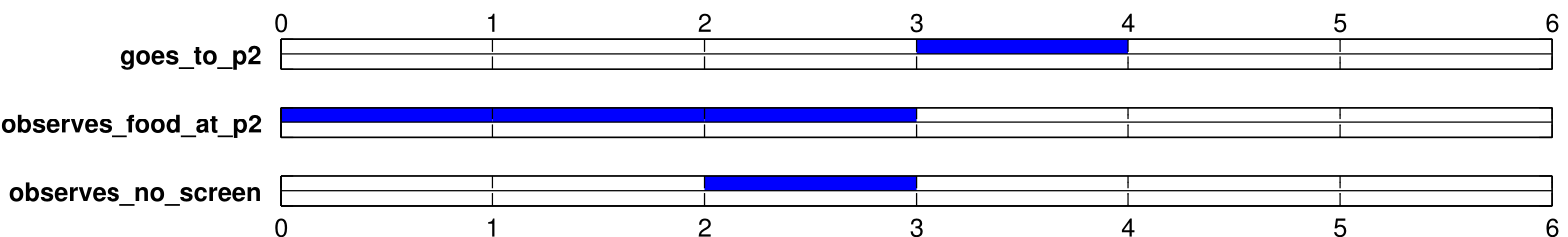
close all

model = l2('leadsto\_tutorial')

model.simulate(5)

model.plot()

This code clears all variables and closes all windows, after which it loads, runs and plots the model. The resulting plot looks as follows.



**Example 2: Emotion regulation**

Emotion regulation refers to ‘all of the conscious and nonconscious strategies we use to increase, maintain, or decrease one or more components of an emotional response’ (Gross, 2001). This ability to regulate our own emotional state is one of the properties that distinguish humans from other higher animals, providing us a high behavioral flexibility, which may result in a higher sense of well-being and mental health. A specific strategy for emotion regulation that is widely studied is reappraisal. Gross (2001) defines reappraisal as a process where ‘the individual reappraises or cognitively re-evaluates a potentially emotion-eliciting situation in terms that decrease its emotional impact’.

A simple simulation model of reappraisal, which is based on the model presented in (Bosse, Pontier, and Treur, 2010) consists of the following two difference equations:

1) new\_ERL = ERL + (1-) \*(((1-w)\* v1 + w\* v2) - ERL) t

2) new\_v2 = v2 - (2 \* d / dmax)t (where d = ERL - ERLnorm)

To implement this model, we create the following files.

**sorts.l2**

The only sort we will use is REAL, which is defined by default. Therefore, there is no need for a sorts.l2 file.

**predicates.l2**

erl; REAL   
d; REAL  
v2; REAL

These are the three values calculated in the description above. Note, we could also calculate *d* ‘on the fly’ in the rule for *v2*, thereby eliminating the need for this predicate.

**scenarios.l2**

erl; 1; 0  
d; 1; 0  
v2; 1; 0

We only define one scenario, with all the values at time point 1 being 0.

**parameters.l2**

default(  
 beta; 0  
 w; 0  
 v1; 0  
 a2; 0  
 delta\_t; 0  
 d\_max; 1  
 erl\_norm; 0  
)

reappraisal(  
 beta; 0,5257  
 w; 0,5773  
 v1; 0,4170  
 a2; 0,0426  
 delta\_t; 0,0439  
 d\_max; 1  
 erl\_norm; 0  
)

This example uses multiple parameters. We define two sets, one default set containing meaningless values and one set tuned to resemble a reappraisal pattern.

**rules.m**

function [ fncs ] = rules()…

%ADD RULES BELOW

function result = ddr1( trace, params, t )

erl\_new = trace(t).erl + (1-params.beta) \* (( (1-params.w) \*  
 params.v1 + params.w \* trace(t).v2 ) - trace(t).erl) \*   
 params.delta\_t;

result = {'erl', t+1, erl\_new};

end

function result = ddr2( trace, params, t )

v2\_new = trace(t).v2 - ( params.a2 \* trace(t).d / params.d\_max ) \*   
 params.delta\_t;

result = {'v2', t+1, v2\_new};

end

function result = ddr3( trace, params, t )

d\_new = trace(t).erl - params.erl\_norm;

result = {'d', t+1, d\_new};

end

Our rules.m file only contains the two formulas given in the text, as well as a separate function for d.

**Running the example in Matlab**

clear all

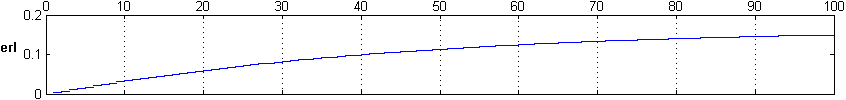
close all

model = l2('emotion\_regulation')

model.simulate(100, 'default', 'reappraisal')

model.plot({‘erl’})

This code clears all variables and closes all windows, after which it loads, runs and plots the model. For this simulation run, the default (and only) scenario is used, but the parameter set ‘reappraisal’ is used instead of the set of default values. For the plot, only ‘erl’ is of our interest.



**Example 3: IAPS pictures**

‘The International Affective Picture System (IAPS) is [..] a set of normative emotional stimuli for experimental investigations of emotion and attention.’ (Lang et al., 1999) Each of these pictures has a rating for both valence and arousal. Consider an agent viewing a number of these pictures. The valance and arousal experienced by that agent will adept to those of the pictures viewed. The speed of this process depends on the speed parameters w1 and w2 respectively.

**DDR1 (Domain Dynamic Relation 1)**at any point in time, for each agent X

if agent X has a valence V and arousal A

and an agent X observes a picture P from the IAPS picture set

and picture P has a valance rating V1 and arousal rating A1

then agent X will get a valence of V+w1(V1-V) and arousal of A+w2(A1-A)

For the implementation of this model, the following files are created.

**sorts.l2**

AGENT; {arnie, bernie, charlie}  
PICTURE; {p1, p2, p3, p4}

We consider here both agent and pictures as sort, with 3 agents and 4 pictures possible.

**predicates.l2**

% picture(PICTURE, VALENCE, AROUSAL}  
picture; {PICTURE, REAL, REAL}

% emotion(AGENT, VALENCE, AROUSAL)  
emotion; {AGENT, REAL, REAL}

observes; {AGENT, PICTURE}

For this model, three predicates are necessary. The first, picture, describes the valence and arousal of each picture. The second, emotion, describes the current valance and arousal value of an agent. The last predicate, observes, is used to denote if an agent views a particular picture.

**parameters.l2**

w1; 0.1  
w2; 0.1

To keep it simple, we only consider one possible set for the parameters given in the model description.

**scenarios.l2**

% IAPS picture valence/arousal  
picture; [1:50]; {p1, 1, 9}  
picture; [1:50]; {p2, 9, 9}

% Starting emotion of the agents  
emotion; 1; {arnie, 4.5, 4.5}  
emotion; 1; {bernie, 4.5, 4.5}

% Observing of pictures  
observes; [1:25]; {arnie, p1}   
observes; [1:25]; {bernie, p1}   
observes; [26:50]; {arnie, p2}

One default scenario is defined, using two pictures and two agents. The first lines describe the static valence and arousal values for the two pictures; one very arousing sad picture (p1) and another very arousing happy picture (p2). The next lines describe the starting emotion values of arnie and bernie. The last lines describe that both agents view the sad picture after which only arnie looks at the happy picture.

**rules.m**

%ADD RULES BELOW

function result = ddr1( trace, params, t )

result = {};

%go through each agent's emotion

for elem = trace(t).emotion

%get the contents of the element

emotion = elem{1};

x = emotion{1}; %agent name

v = emotion{2}; %valence

a = emotion{3}; %arousal

% go through each picture that agent observes

for elem = l2.getVar(trace, t, 'observes', {x, []})

%get the contents of the element

observes = elem{1};

p = observes{2}; %picture name

%get the valence and arousal of that picture

elem = l2.getVar(trace, t, 'picture', {p, [], []});

picture = elem{1};

v1 = picture{2}; %picture valence

a1 = picture{3}; %picture arousal

%adjust v and a accordingly

v = v + params.w1 \* (v1 - v);

a = a + params.w1 \* (a1 - a);

end

%add the new emotion level to the results

result = { result{:} {'emotion', t+1, {x, v, a}} };

end

end

One, relatively large rule is needed to implement this model. The majority of the rule is part of a for loop going through the emotion of each agent. For that agent, each picture it observes is retrieved. For those pictures, the valence and arousal value is looked at and using those values the emotion level of the agent is changed. When each picture an agent looked at is processed, the new emotion level is added to the results.

**Running the example in Matlab**

clear all

close all

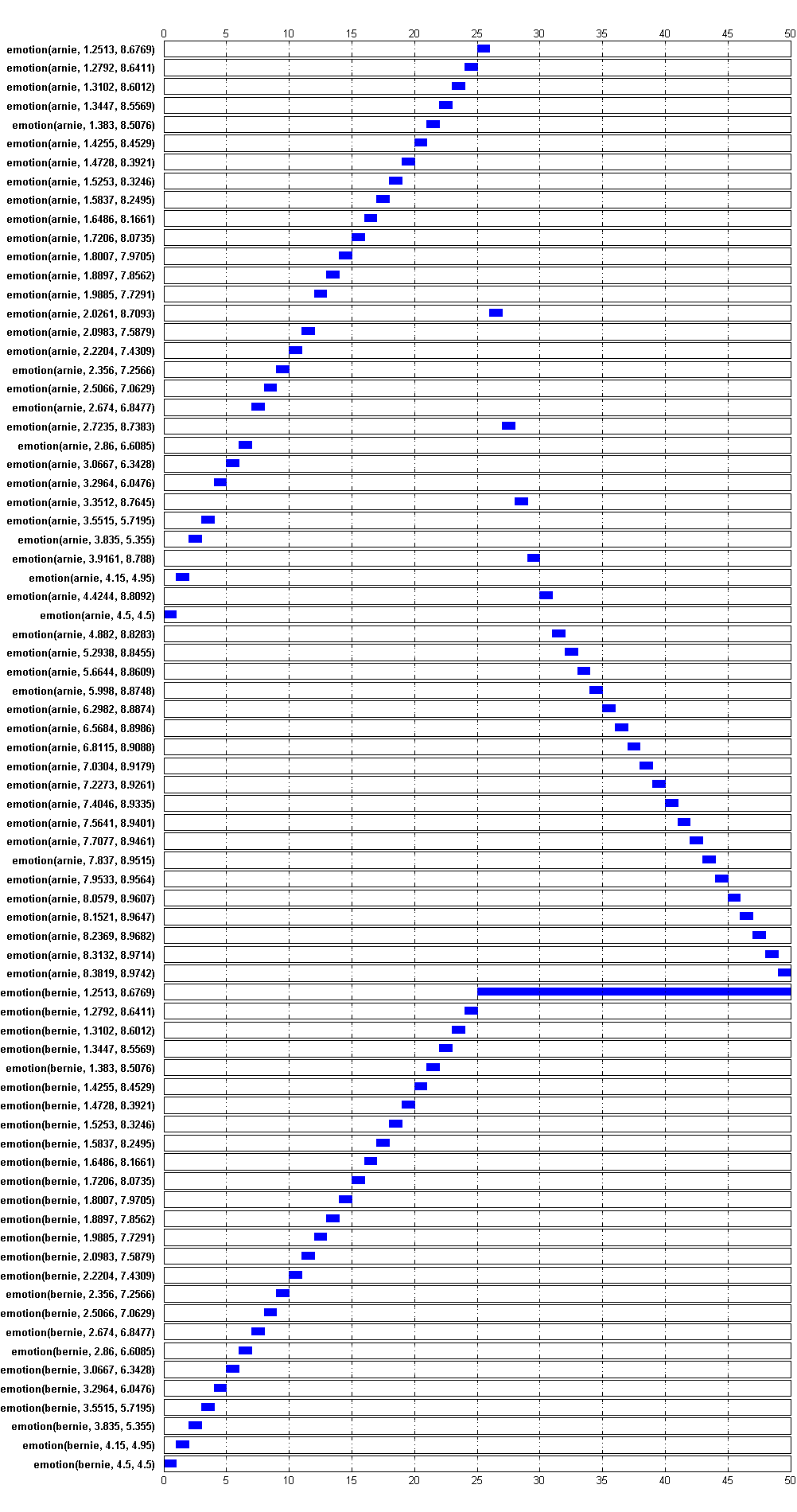
model = l2(IAPS\_pictures)

model.simulate(50)

%Plot is too large for screen, thus print as PDF

model.plotFile(‘emotion’, {‘emotion’})

Part of the large plot is shown below where you can see the emotion level of arnie changes throughout the entire simulation, while the emotion of bernie is constant for the last part of the simulation. For better plotting results, it might be beneficial to write your own plotting function in this case

. ­­

**­­**

**­­Example 4: Theory of mind**

One of the most important milestones in theory of mind development is gaining the ability to attribute *false belief*: that is, to recognize that others can have beliefs about the world that are diverging. In the ‘appearance-reality’, or ‘Smarties’ task, experimenters ask children what they believe to be the contents of a box that looks as though it holds a candy called smarties. After the child guesses (usually) smarties, it is shown that the box in fact contained pencils. The experimenter then re-closes the box and asks the child what she thinks another person, who has not been shown the true contents of the box, will think is inside. The child passes the task if he/she responds that another person will think that there are smarties in the box, but fails the task if she responds that another person will think that the box contains pencils. Gopnik & Astington (1988) found that children pass this test at age four or five years.

This experiment has been implemented as follows.

**sorts.l2**

AGENT; {arnie, bernie, charlie}  
INFO; {smarties, pencils, <belief>}

As can be seen, we have three different agents, and some INFO element which holds information about the contents of the box (smarties or pencils). However, it is also stated that the INFO element can be <belief>, meaning that instead of either smarties or pencils a predicate ‘belief’ is also a valid INFO element. This predicate, as well as two others are defined below.

**predicates.l2**

age; {NAME, REAL}  
belief; {NAME, INFO}  
observe; {NAME, INFO}

Three predicates are used, one describing the age of the agent, one for the beliefs of the agent and a third predicate for the observations of an agent. Keep in mind, that the belief can be either smarties, pencils or a belief about some other agent having a particular belief.[[4]](#footnote-4)

**parameters.l2**

age\_tom; 4

One parameter is used for this model, setting the age from which an agent can apply theory of mind, thereby understanding that a different person, not having seen the contents of the box, will falsely belief that there are smarties instead of pencils in the box.

**scenarios.l2**

default (  
 age; [1:5]; {arnie, 3}  
 belief; [1]; {arnie, smarties}  
 belief; [1]; {arnie, <belief{bernie, smarties}>}  
 observe; [2]; {arnie, pencils}  
)

older (  
 age; [1:5]; {arnie, 5}  
 belief; [1]; {arnie, smarties}  
 belief; [1]; {arnie, <belief{bernie, smarties}>}  
 observe; [2]; {arnie, pencils}  
)

We define two scenarios, one where Arnie is too young to apply theory of mind, and another where he is old enough. Furthermore, we initially set both the belief of arnie that there are smarties in the box and his belief that he beliefs that bernie thinks there are smarties in the box. Lastly, at timepoint 2 arnie observes there are pencils in the box.

On the next page, the rule to implement this model is shown. It cycles through all beliefs and for each beliefs considers all observations made by the agent that has that belief. For each observation, we first check whether the belief was a belief of the agent itself or a nested belief about some other agent’s belief. This is done by checking if the observation is a structure, which it would be in the case of a nested belief. If it isn’t a structure, we update the belief with the current observation. Otherwise, we need to check whether or not the agent is able to apply theory of mind. If not, the nested belief also gets updated to match the current observation.

Although this method works, it relies on the assumption that any nested belief does not contain another nested belief. For example, consider the following belief where arnie believes that bernie believes that charlie believes the box contains smarties.

belief{arnie, <belief{bernie, <belief{charlie, smarties}>}>}

In order to cope with any number of nested beliefs, the rule needs to be implemented differently using a form of recursive programming. This method will be explained next and is included in the example code as comments. For the given scenarios, results for both methods are similar.

**rules.m**

function result = ddr1( trace, params, t )

result = {};

%go through each belief

for elem = trace(t).belief

%get the contents of the element

contents = elem{1};

agent = contents{1}; %agent name

belief = contents{2}; %the belief or a nested belief

% go through each picture that agent observes

for elem2 = l2.getVar(trace, t, 'observe', {agent, []})

%get the contents of the element

contents2 = elem2{1};

observation = contents2{2}; %the info element

%belief is an info element, not a nested belief

if ~isstruct(belief)

belief = observation;

%belief is a (nested) belief of someone else

else

%get age to check if agent can apply theory of mind

elem3 = l2.getVar(trace, t, 'age', {agent, []});

contents3 = elem3{1};

tom = contents3{2} > params.age\_tom;

%if no theory of mind, simply update nested belief

if ~tom

belief = struct('belief',

{belief(1).belief observation} );

end

end

end

result = { result{:} {'belief', t+1, {agent, belief}} };

end

end

On the following page, the rule to implement this rule recursively is shown. The first part of this rule is similar as different from before. It cycles through all the beliefs and updates them for the relevant observations. However, beliefs are now updated by a nested function updateBelief, such that it is possible to cope with any amount of nested beliefs.

Consider the function updateBelief and firstly notice that it is part of the function ddr1. It takes as input the current belief, the observation and whether or not theory of mind is applicable. If the belief is not a structure (~isstruct(belief)), the belief becomes equal to the observation. Otherwise, if the belief is a nested belief, that belief is only updated if no theory of mind is applied. The new belief is a new structure, named belief (the predicate name). The value for the agent is equal to the agent name in the previous belief. To update the belief in this nested belief, the updateBelief function is called again with the belief value from the nested belief, the current observation and whether or not theory of mind is applied. Using this recursive structure, it does not matter how many nested beliefs there are, they always get updated as expected.

**rules.m**

%ADD RULES BELOW

function result = ddr1\_recursive( trace, params, t )

result = {};

%go through each belief

for elem = trace(t).belief

%get the contents of the element

contents = elem{1};

agent = contents{1}; %agent name

belief = contents{2}; %the belief or a nested belief

% go through each picture that agent observes

for elem2 = l2.getVar(trace, t, 'observe', {agent, []})

%get the contents of the element

contents2 = elem2{1};

observation = contents2{2}; %the info element

%get age to check if agent can apply theory of mind

elem3 = l2.getVar(trace, t, 'age', {agent, []});

contents3 = elem3{1};

tom = contents3{2} > params.age\_tom;

belief = updateBelief(belief, observation, tom);

end

result = { result{:} {'belief', t+1, {agent, belief}} };

end

function belief = updateBelief(belief, observation, tom)

%belief is a info element, not a nested belief

if ~isstruct(belief)

belief = observation;

%belief is a (nested) belief of someone else

else

%if no theory of mind, simply update nested belief

if ~tom

belief = struct('belief', {belief(1).belief

updateBelief(belief(2).belief, observation, tom)} );

end

%otherwise, do noting with the belief

end

end

end

**Running the example in Matlab**

clear all

close all

model = l2('theory\_of\_mind')

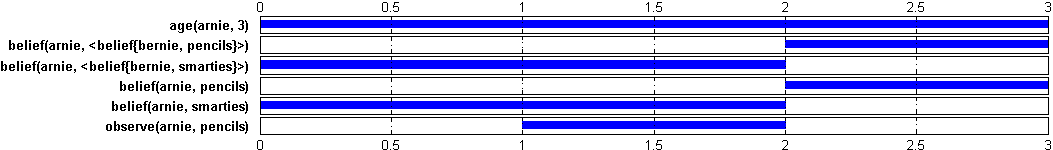
model.simulate(3)

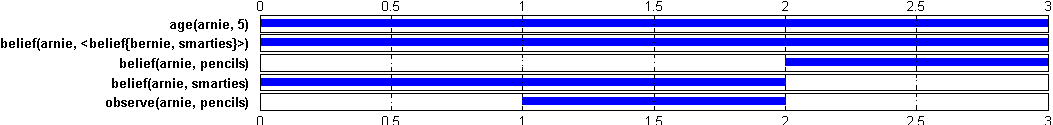
model.plot()

model.simulate(3, 'older')

model.plot()

To run this example, the model is simulated for both scenarios. The resulting graphs are shown below as well. Note the difference between the age of arnie and whether or not the nested belief of bernie gets updated based on the observation arnie makes at timepoint 2.





1. Alternatively, you can add the folder to you Matlab path, but this will not be explained here. [↑](#footnote-ref-1)
2. Or more precisely of character arrays, as Matlab saves strings as character arrays. [↑](#footnote-ref-2)
3. Matlab does not show curly brackets for 1x1 cell arrays, but for the distinction between cell arrays and regular (char) arrays these are shown here. [↑](#footnote-ref-3)
4. Using this formalization, an agent can also have an observation about some particular belief. This however is not used or required for modeling this experiment. [↑](#footnote-ref-4)