**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 the existing cell array.

**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: Simple**

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

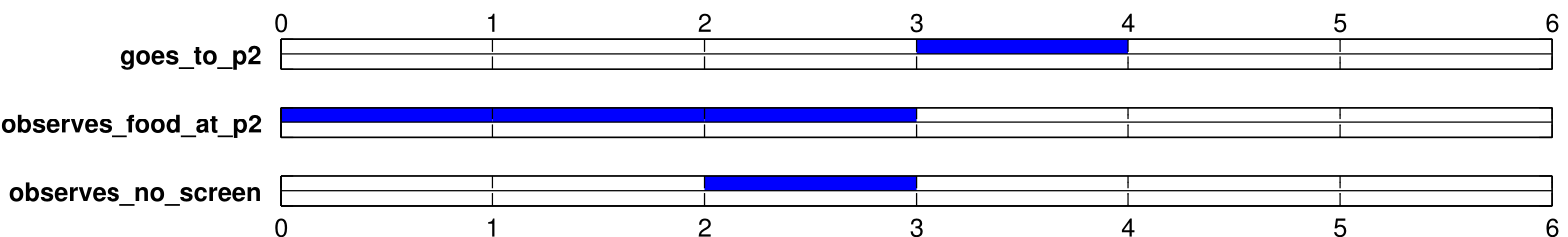
close all

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

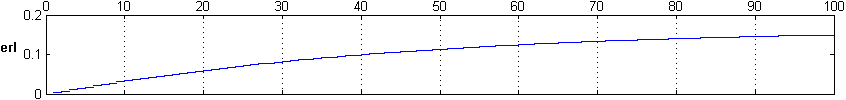
close all

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: ???**

**Example 4: Water**

A phase transition is the transformation of a thermodynamic system from one phase or state of matter to another one by heat transfer. Consider a very simple scenario in which we have a substance at a particular temperature. If this temperature changes, either by actively heating it or by simple adjusting to the room temperature, the state of the substance may change if the temperature reaches its freezing or boiling point.

This model has been implemented as follows.

**sorts.l2**

SWITCH; {off, on}  
STATE; {solid, liquid, gas}

We create a switch, to turn a heater on or off. A second sort describes the state of the substance.

**predicates.l2**

% Heater on/off  
heater; SWITCH  
  
% Substance state, temperature  
substance; {STATE, REAL}

Two predicates are used, one for the heater and another for the substance. Note the usage of comments to describe the meaning of the predicates.

**parameters.l2**

default (  
 room\_temp; 20  
 heater\_temp; 150  
 transfer; 0.1  
 freezing\_point; 0  
 boiling\_point; 100  
)

ethanol (  
 room\_temp; 20  
 heater\_temp; 150  
 transfer; 0.1  
 freezing\_point; -114.6  
 boiling\_point; 78.4  
)

We define two sets of parameters, the default set using the freezing and boiling point of water and a second set for ethanol. Furthermore, a room temperature, heater temperature and heat transfer is defined.

**rules.m**

function [ fncs ] = rules() …

%ADD RULES BELOW

function result = ddr1( trace, params, t )

result = {};

%substance is warming up

if strcmp(trace(t).heater{1}, 'on')

ambientTemp = params.heater\_temp;

else

ambientTemp = params.room\_temp;

end

%get all substances and go through them

for elem = trace(t).substance

%get the contents of the element

substance = elem{1};

%if colder than ambientTemp, warm up

if substance{2} < ambientTemp

if strcmp(substance{1},'solid') &&   
 substance{2} ~= params.freezing\_point

%solids increase to max freezing\_point

new\_temp = substance{2} + params.transfer \*

(ambientTemp - substance{2});

new\_temp = min(new\_temp, params.freezing\_point);

result = [result {'substance', t+1, {substance{1}, new\_temp}}];

elseif strcmp(substance{1}, 'liquid') &&

substance{2} ~= params.boiling\_point

%liquids increase to max boiling\_point

new\_temp = substance{2} + params.transfer \*

(ambientTemp - substance{2});

new\_temp = min(new\_temp, params.boiling\_point);

result = [result {'substance', t+1, {substance{1}, new\_temp}}];

elseif strcmp(substance{1}, 'gas')

%gas increases to max ambientTemp

new\_temp = substance{2} + params.transfer \*

(ambientTemp - substance{2});

result = [result {'substance', t+1, {substance{1}, new\_temp}}];

end

end

end

end…

Here the first rule describing the temperature increase of a substance is shown. Firstly, the ambient temperature is derived based on the state of the heater. Afterwards, for each element that is below this temperature, the temperature is increased until either its freezing or boiling point. To go through each of the substances, a for loop is created that retrieves all the substances at time point t. In each iteration, that substance is returned as a cell *elem*, of which we first need to retrieve the contents *elem{1}*, before we can access the values of that substance. Furthermore, as there are possible more results, each result is added to an cell array created at the start of this function. A similar function exists for the cooling down of substances and is not shown here.

**rules.m**

…  
function result = ddr3( trace, params, t )   
 result = {};

%substance is warming up, state change  
 if strcmp(trace(t).heater{1}, 'on')  
 ambientTemp = params.heater\_temp;  
 else  
 ambientTemp = params.room\_temp;  
 end

%if warmer than freezing, change solids at freezing\_point to liquid

if params.freezing\_point <= ambientTemp

for elem = l2.getVar(trace, t, 'substance', {'solid',

params.freezing\_point})

result = [result {'substance', t+1, {'liquid',

params.freezing\_point}}];

end

end

%if warmer than boiling, change liquids at boiling\_point to gas

if params.boiling\_point <= ambientTemp

for elem = l2.getVar(trace, t, 'substance', {'liquid',

params.boiling\_point})

result = [result {'substance', t+1, {'gas',

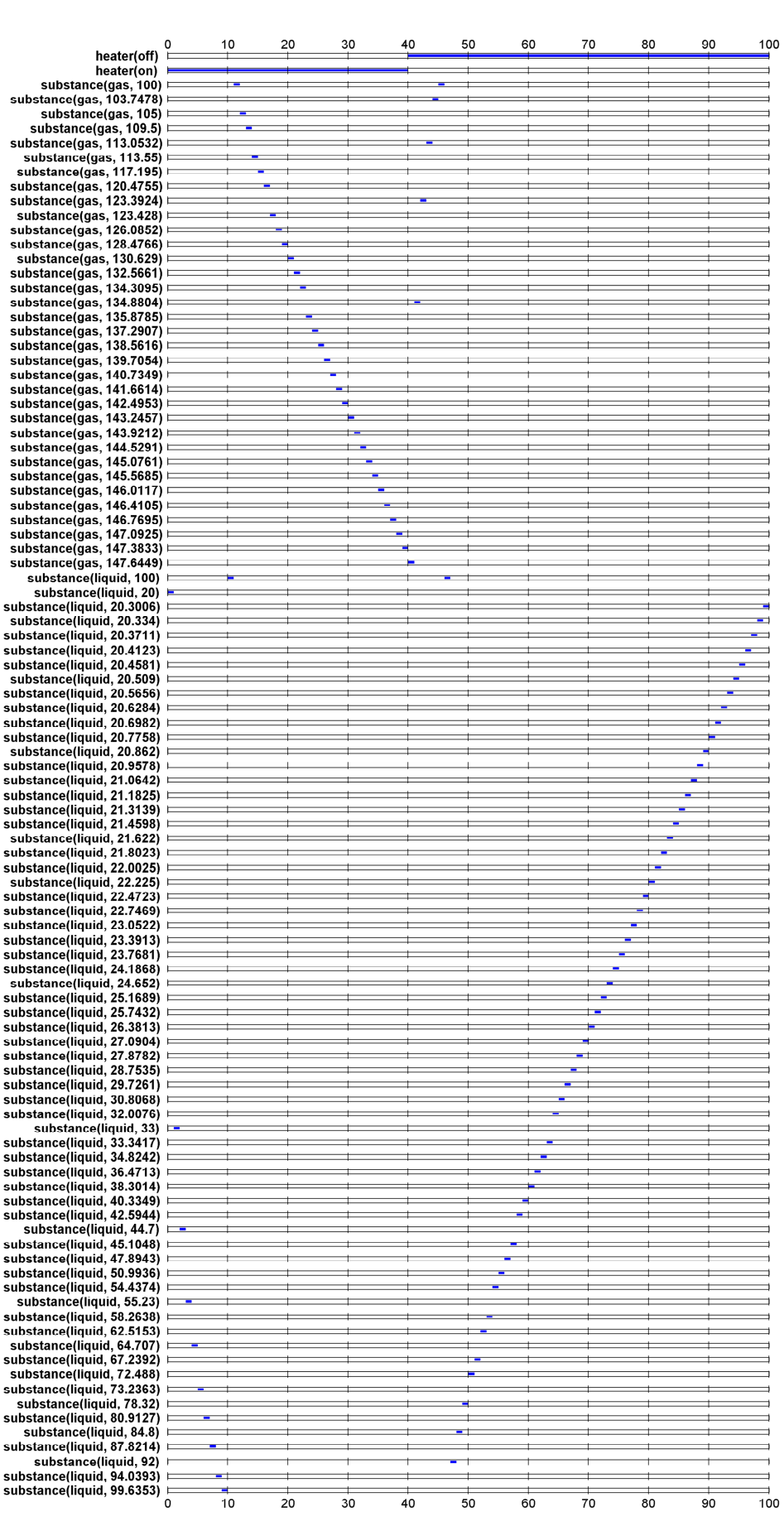
params.boiling\_point}}];

end

end

end  
…

This rules described the state change of a substance if the ambient temperature is above either the freezing or the boiling point. Take note of the l2.getVar() function, which retrieves for a given trace and a given time point those values of a given predicate that match a certain pattern. In this case, all solid substances at freezing point and all liquid substances at boiling point are retrieved. An empty array *[]* can be used as a wildcard in the pattern such that   
 l2.getVar(trace, t, 'substance', {'solid', []})  
would retrieve all solid substances, regardless of their temperature. Again a simular function as ddr3 exist for cooling down, but is not described here.



**Running the example in Matlab**

clear all

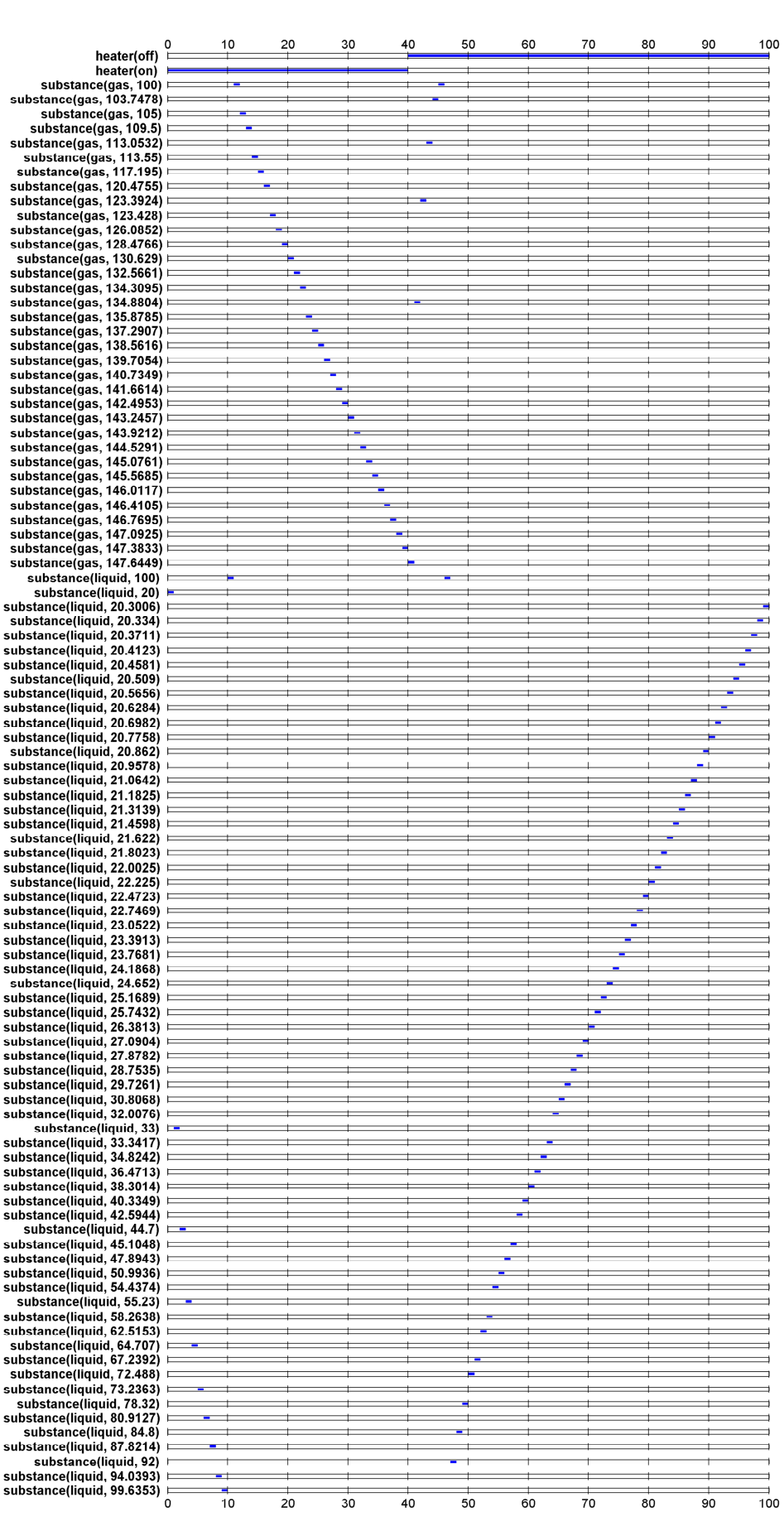
close all

model = l2('thermodynamics')

model.simulate(100)

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

model.plotFile('water')



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)