Intro to Python ACT-R – LaLoCo Lab, Fall 2014

Adrian Brasoveanu[*]

November 17, 2014

Contents

1 Why do we care about ACT-R, and cognitive architectures and modeling in general

Linguistics is part of the larger field of cognitive science. So the answer to this question is one that applies to cog sci in general. Here's one recent version of the argument, taken from chapter 1 of Lewandowsky and Farrell 2010. The argument is an argument for *process* models as the proper scientific target to aim for

[*]This intro to Python ACT-R is partly based on Kam Kwok's *Python ACT-R User Guide version 1.0* (12/21/2009) (`https://sites.google.com/site/kamkwokcogsci/cogsci-papers-and-projects/pACTRUserGuideReleasdv1.pdf`), on Robert West's tutorials available at `https://sites.google.com/site/pythonactr/`, and on chapters 11 and 12 of Stewart 2007. Python ACT-R has been created and is maintained by Terrence Stewart (see Stewart 2007, Stewart and West 2007). The github repo is available here `https://github.com/tcstewar/ccmsuite`. This document has been created with PythonTex (Poore, 2013).

(roughly, models of human language performance), rather than *characterization* models (roughly, models of human language competence).

Both of them are better than simply *descriptive* models, "whose sole purpose is to replace the intricacies of a full data set with a simpler representation in terms of the model's parameters. Although those models themselves have no psychological content, they may well have compelling psychological implications. [Both characterization and process models] seek to illuminate the workings of the mind, rather than data, but do so to a greatly varying extent. Models that characterize processes identify and measure cognitive stages, but they are neutral with respect to the exact mechanics of those stages. [Process] models, by contrast, describe all cognitive processes in great detail and leave nothing within their scope unspecified. Other distinctions between models are possible and have been proposed [. . .], and we make no claim that our classification is better than other accounts. Unlike other accounts, however, **our three classes of models map into three distinct tasks that confront cognitive scientists. Do we want to describe data? Do we want to identify and characterize broad stages of processing? Do we want to explain how exactly a set of postulated cognitive processes interact to produce the behavior of interest?**" (Lewandowsky and Farrell, 2010, p. 25)

In a bit more detail: "Like characterization models, [the power of process models] rests on hypothetical cognitive constructs, but by providing a detailed explanation of those constructs, they are no longer neutral. [. . .] At first glance, one might wonder why not every model belongs to this class. After all, if one can specify a process, why not do that rather than just identify and characterize it? The answer is twofold. First, it is not always possible to specify a presumed process at the level of detail required for [a process] model [. . .] Second, there are cases in which a coarse characterization may be preferable to a detailed specification. For example, it is vastly more important for a weatherman to know whether it is raining or snowing, rather than being confronted with the exact details of the water molecules' Brownian motion. Likewise, in psychology [and linguistics!], modeling at this level has allowed theorists to identify common principles across seemingly disparate areas. That said, we believe that in most instances, cognitive scientists would ultimately prefer an explanatory process model over mere characterization." (Lewandowsky and Farrell, 2010, p. 19)

2 The very basics: Defining an agent, a model / environment, and running the model

If you want to create an agent for your model, the first step is to define its class. The following example defines a user agent class called `MyAgent` (a 'class' is a template for creating an object).

The lines below are `import` statements for including the `ccm` library, which provides all the necessary Python ACT-R classes for modeling.

(1) File **example_1.py**:

```
import ccm # import ccm module library for Python ACT-R classes
from ccm.lib.actr import *
```

We then declare a user defined agent class called `MyAgent`, which is inherited from the CCMSuite `ACTR` class. A class is a type. To get a token, i.e., an object of that class, the class must be instantiated. An agent is created with a production module by default. The production module is a part of the ACTR class we're inheriting from. So the user can define production rules as a part of the agent without any additional setup.

(2) File **example_1.py** (ctd.):

```
class MyAgent(ACTR):
    focus=Buffer() # Creating the goal buffer for the agent
    def init(): # this rule fires when the agent is instantiated.
        focus.set("sandwich bread") # set focus buffer to direct program flow
    def bread_bottom(focus="sandwich bread"): # if focus="sandwich bread" , fire rule
        print "I have a piece of bread"
        focus.set("stop") # set focus buffer to direct program flow
    def stop_production(focus="stop"):
        self.stop() # stop the agent
```

Let's examine our class in detail. The line `focus=Buffer()` creates the focus buffer for the agent. By convention, the *focus buffer* is Python ACT-R name for the goal buffer. Buffers are created by calling the `Buffer()` class. These are simple ACT-R buffers which are able to store a single chunk. They are the main system for communicating between modules. You can create as many as you like and give them any names you like, e.g., `goal = Buffer()`, `retrieval = Buffer()`, `imaginal = Buffer()`. Traditionally, ACT-R models have included a goal buffer, a retrieval buffer, and a visual buffer. More recently this has been expanded to include an imaginal buffer as well. (Apparently, there's a tradition for ACT-R buffers to end in "-al".)

The initialization production rule `init()` is the first production rule that an agent will execute when the agent object is instantiated (created). In this example, the action of the rule is to set the focus buffer to `"sandwich bread"` using the buffer's `set` method. In general, a buffer can be set by its `set` method. A method is a function that is associated with / part of an object.

The main purpose of the focus buffer is to represent the model's central executive attention function and it is used to guide or direct the flow of the cognitive task. Typically, inside a production rule, the focus buffer is set to the value of the buffer condition for the next step in the cognitive task.

For example, the `init` rule sets the focus buffer to `"sandwich bread"`, which effectively directs the agent to the `bread_bottom` production rule below that has its (pre)condition for firing that the focus buffer is equal to `"sandwich bread"`. The actions of this rule are to print the string `"I have a piece of bread"` to the console, and then to set the focus buffer to the condition of the next rule that should be applied in the cognitive task. In this very simple case, the task is basically over: the `stop` production rule can fire now, which stops the agent.

Every ACT-R agent automatically has a procedural system / module. This is what causes productions which match the current situation to 'fire', i.e., to perform actions by passing messages to modules or changing buffer values. The default procedural system is a very simple one that does not perform any learning to adjust the utility of each production, so if more than one production can fire at a given time, one will be chosen at random. By default, productions require 0.05 seconds (50 ms) to fire. This can be adjusted by setting the `production_time` value, as shown in the example below. We can also set `production_time_sd` to adjust the standard deviation of the production times (the default is 0). Finally, we can also set the `production_threshold`, which indicates that a production must have a utility above this threshold before it will fire. There are a variety of ways to change how the production system works; we'll talk more about this, and about production utilities and utility learning in a later section.

(3)
```
class SimpleModel(ACTR):
    production_time = 0.05
    production_sd = 0.01
    production_threshold = -20
```

Let's see how this works. We instantiate the class, i.e., we create a specific agent `tim`:

[py1]
```
>>> from example_1 import * # we import our agent and environment classes
>>> tim = MyAgent()
```

Python ACT-R always requires an environment / model in which the agents needs to be placed, but in this case we will not be using anything in the environment so we 'pass' on putting things in there.

(4) File **example_1.py** (ctd.):

```
class MyEnvironment(ccm.Model):
    pass
```

We instantiate the environment and put the agent `tim` in it:

```
[py2] >>> empty_environment = MyEnvironment()
      >>> empty_environment.agent = tim
```

We log / print out everything that happens in the environment:

```
[py3] >>> ccm.log_everything(empty_environment)
         0.000 agent.production_threshold None
         0.000 agent.production_time_sd None
         0.000 agent.production_match_delay 0
         0.000 agent.production_time 0.05
         0.000 agent.focus.chunk None
```

We now run the environment / model:

```
[py4] >>> empty_environment.run()
         0.000 agent.focus.chunk sandwich bread
         0.000 agent.production bread_bottom
         0.050 agent.production None
      I have a piece of bread
         0.050 agent.focus.chunk stop
         0.050 agent.production stop_production
         0.100 agent.production None
```

Note how modules – in this example, the procedural module, which is the only module built into any Python ACT-R agent by default – are activated instantaneously when conditions satisfy their use; instantaneously in simulated time (this obviously requires some machine time to compute).

Once activated, modules calculate the total cost in time for the action they are scheduled to do. This is the reason for the third line in the above output `0.050 agent.production None`, which is emitted at time 0.000 s, but lists the projected action for the immediately following time period when the procedural module could schedule another action. Given the current context (i.e., the current contents of all the buffers the agent has) and the production rules the agent possesses, nothing can actually be scheduled for the next time period.

Once the 50 ms required for the `bread_bottom` rule to fire elapse, the actions associated with this rule are performed instantaneously (in simulated time), i.e., we print `"I have a piece of bread"` and update the chunk in the focus / goal buffer to `"stop"`. The production rule `stop_production` can now be activated – and it is, also instantaneously, i.e., at the same time step 0.050 s. Since a new production is activated, the procedural module calculates the necessary time to perform it (50 ms) and lists what production rule can be performed once this time has elapsed: `0.100 agent.production None`. Just as before, no production can be triggered given the current context.

The `stop_production` rule instructs the agent to stop, so the current model run terminates at time step 0.100 s (not explicitly listed; it is implicit given the schedule for the next time step `0.100 agent.production None`
provided by the procedural module).

So now we can stop the whole environment / model:

```
[py5] >>> ccm.finished()
```

One note about the way chunks – i.e., sets of slot-value pairs, or in non-ACT-R but more standard terminology, attribute-value matrices (AVMs) – are represented in Python ACT-R. The full representation for the chunk `"sandwich bread"` is in fact something like `"goal:sandwich object:bread"`, where `"goal:\dots object:\dots"` are the slots / features / values, and `"\dots:sandwich \dots:bread"` are the values. When the chunk is represented simply in terms of the sequence of values, i.e., `"sandwich bread"`, the attributes are implicitly introduced as 'attribute 1', 'attribute 2', etc., i.e., the chunk is treated as a partial variable assignment with the variables distinguished by their position in the sequence. Chunks / AVMs are really dictionaries (in the Python sense, i.e., hash tables), which are really just souped up partial variable assignments (mathematically).

In ACT-R, chunks have a limited number of slots ($7 + / - 2$), so they are rather small partial variable assignments. The values of their slots can be other chunks though, and this is how complex structures (e.g., syntactic trees) can be represented – very much like in GPSG and HPSG.

3 Central Executive Control

Let's understand a bit better the output of the model.

The central executive control of an agent is one of the core components of the ACT-R theory. The main function of the central executive control is to represent 'consciousness' (in a somewhat technical term of monitoring the contents of all buffers) and intentionality in the agent.

When an agent is created, the central executive control function starts to operate as the agent's awareness. It continuously examines the buffers and the module states to determine if a particular production rule is to be activated.

A production rule is "fired" or activated when the conditions of the rule are completely matched, i.e., when all the buffer values are matched with the conditions stated in the condition-part of a production rule. For example, the production rule `bread_bottom` in the above example has as its only condition `focus="sandwich bread"`, and its actions are printing a message and resetting the focus / goal buffer.

Production rules are activated one rule at a time and each firing consumes 50 ms (0.05 s). For more complex models, multiple rules could be activated in parallel in different agents or modules.

If no productions match, then the production system does nothing. However, as soon as the content of a buffer or a module state changes (see below for more discussion of module states wrt the Declarative Memory module) such that a production rule could fire, the production system notices this and instantaneously (in simulated time) readies that production to fire 50 ms later. Thus, the production system performs its search for matching patterns every time there is a state in context (i.e., buffers) or in module states, and it is not currently waiting to fire a previously scheduled production.

The production system also checks that the contents of the relevant buffers and module states that were preconditions for the currently scheduled production rule are still a valid match after the 50 ms delay. That is, even if the contents of the buffers have changed since the beginning of the 50 ms delay, the changes should not affect the relevant condition pattern matches or the production could be inappropriate. If this change occurs in Python ACT-R, the production selection process is restarted after the 50 ms without firing the previously scheduled rule (it's unclear when the process is restarted in Lisp ACT-R – immediately upon a buffer change or after the 50 ms have elapsed).

The central executive control and the production rules together are responsible for the algorithm of the program/agent. In Example 1, the rules are fired or executed according to the sequence set by the focus buffer.

4 Declarative Memory

Declarative memory refers to memories that can be consciously recalled and discussed. The two types of declarative memories are *semantic memory* and *episodic memory*.

Semantic memory is knowledge independent of time and place – a piece of information. For example, *A robin is a bird* can be stored as a semantic memory chunk.

Episodic memory is factual knowledge of personal experience at a specific time and in a specific place.

Declarative memory is subject to forgetting: memories have an activation level that decays over time. The probability and latency of retrieving / recalling a declarative memory depends on its activation value.

In ACT-R, declarative memories are represented / formalized as "chunks", which are attribute-value matrices (AVMs, or feature-value matrices) of the kinds used in GPSG and HPSG. Chunks are used to communicate / transfer information between modules through buffers, which are the interfaces of the various modules that can store only one chunk at any given time.

4.1 A very simple example

Declarative memories are the data store for the agent. In Python ACT-R, the agent can remember or learn new information by creating new chunks. Our Example 2 shows how an agent can store and retrieve information from declarative memory.

(5) File **example_2.py**:

```
import ccm
from ccm.lib.actr import *


class MyAgent(ACTR):

    focus = Buffer()
    # create a buffer for the declarative memory
    DMbuffer = Buffer()
    # create a declarative memory object called DM
    DM = Memory(DMbuffer)

    def init():
        # put a chunk into DM; has a slot/feature/attribute "condiment"
        DM.add("condiment:mustard")
        # set focus buffer to trigger recalling rule
        focus.set("get_condiment")

    def recalling(focus="get_condiment"):
        print "Retrieving the condiment chunk from DM into the DM buffer"
        # the value "?" means that the slot can match any content
        DM.request("condiment:?")
        # set focus buffer to trigger condiment rule
        focus.set("sandwich condiment")

    # Note how DMbuffer is used as a condition for the rule below
    def condiment(focus="sandwich condiment", DMbuffer="condiment:?condiment"):
        print "My recall of the condiment from memory is...."
        # the condiment variable we print is associated with the ?condiment variable
        print condiment
        # set focus buffer to trigger stop_production rule
```

```
            focus.set("stop")

        def stop_production(focus="stop"):
            self.stop()


    class MyEnvironment(ccm.Model):
        pass
```

Example 2 is an extension of Example 1. It includes a declarative memory module `DM` and the associated buffer `DMbuffer`. We use the `Buffer()` class to create `DMbuffer`, a new buffer for the memory object. We then use the `Memory()` class and `DMbuffer` to create a declarative memory object called `DM` for the agent, which will serve as the agent's declarative memory module.

We then add a new chunk to the declarative memory by using the `add` method of the memory object `DM`. Note that a single quoted string is used as a parameter for the method. The quoted string represents a chunk, i.e., a list of slot-value pairs. Just as before, the ":" inside the string is a separator which is used to separate the slot name from its value – `"condiment:mustard"` denotes a chunk with a single slot called `condiment`, the value of which is `mustard`.

In Python ACT-R, to retrieve a value from declarative memory usually takes two rules:

- the `DM.request` rule makes a request to the memory object `DM` using a cue (slot name) – `condiment` in our case; the "?" notation in the memory request indicates that the value of the slot can be anything as long as the chunk has a `condiment` slot;
- the second rule – the production rule
  `condiment(focus="sandwich condiment", DMbuffer="condiment:?condiment")`
  in our case – processes the resulting retrieved value, which is stored in the memory buffer; the meaning of "?" changes here: when "?" precedes a name, it defines a variable for the value of the slot, namely $\boxed{?}$`condiment`

Variables can then be used by the agent for further processing. In our example, the variable $\boxed{?}$`condiment` is assigned the value `mustard` by the chunk that has been retrieved from the declarative memory module and placed in the `DMbuffer`. We use the $\boxed{?}$`condiment` variable later on – but without the initial "?" – to print this value.

Once a variable is assigned a value, that value remains fixed for the life of the variable.

Any variable from the left-hand / condition side of a production rule can be used on the right-hand / action side, but the variable-value pair is discarded after the production is fired. This means that the expressive power of variables is limited in time, and also in space – variables created within the production system / procedural module cannot be used outside of it.

(6) **Convention**: the initial "?" that indicates something is an ACT-R buffer variable is dropped before these variables are used with regular Python commands like `print`.

In our case, the value `mustard` will be printed to the console as a result of the `print` statement.

(7) **Convention**: we name variables over values by using the corresponding slot name, i.e., `"<slot name>:?<slot name>"`.

This why our slot `condiment` has as its value the variable $\boxed{?}$`condiment`.

The rest is very similar to Example 1 before.

We instantiate the environment and put the agent `tim` in it:

```
[py6] >>> from example_2 import *
      >>> tim = MyAgent()
      >>> empty_environment = MyEnvironment()
      >>> empty_environment.agent = tim
```

We now run the model:

```
[py7] >>> ccm.log_everything(empty_environment)
         0.000 agent.production_threshold None
         0.000 agent.production_time_sd None
         0.000 agent.production_match_delay 0
         0.000 agent.production_time 0.05
         0.000 agent.DM.record_all_chunks False
         0.000 agent.DM.threshold 0
         0.000 agent.DM.latency 0.05
         0.000 agent.DM.busy False
         0.000 agent.DM.maximum_time 10.0
         0.000 agent.DM.error False
         0.000 agent.DMbuffer.chunk None
         0.000 agent.focus.chunk None
      >>> empty_environment.run()
         0.000 agent.focus.chunk get_condiment
         0.000 agent.production recalling
         0.050 agent.production None
      Retrieving the condiment chunk from DM into the DM buffer
         0.050 agent.DM.busy True
         0.050 agent.focus.chunk sandwich condiment
         0.100 agent.DMbuffer.chunk condiment:mustard
         0.100 agent.DM.busy False
         0.100 agent.production condiment
         0.150 agent.production None
      My recall of the condiment from memory is....
      mustard
         0.150 agent.focus.chunk stop
         0.150 agent.production stop_production
         0.200 agent.production None
      >>> ccm.finished()
```

The DM module has various parameters, listed in the initialization part of the model (the output of the `ccm.log_everything(empty_environment)` call). We will discuss them soon. For now, notice only that during the model run (the output of the `empty_environment.run()` call), we also keep track of whether the DM module is busy with a retrieval request or not (`0.050 agent.DM.busy True` and `0.100 agent.DM.busy False`).

This is important, because production rules can have such module-state specifications as part of their firing conditions patterns. That is, production rule conditions in ACT-R can include 2 kinds of pattern matches: (*i*) to the context (the state of any of the buffers), and (*ii*) to module states. To accomplish this, a module can include a separate buffer that holds information about its state, with slots and values specific to each module but that include the information about whether the module is currently performing an action or not. In Python ACT-R, we don't need to explicitly add such buffers, but we can pattern match on them.

In Lisp ACT-R, requests are sent to modules by placing the requests in a buffer associated with the module (for DM, for example, this is different from the retrieval buffer). In Python ACT-R, the requests are sent directly to the module. Using the buffer in Lisp ACT-R does not impose any time delay, so the only constraint it imposes is that the request should be a chunk (hence, of limited size). This constraint is the same for Python ACT-R.

4.2 A slightly more complex example

This is a slightly more complicated model with several more productions. Note how the rules are fired according to the sequence set by the focus buffer.

The `condiment(focus="get_condiment")` production first requests the declarative memory module to retrieve the condiment that the customer ordered, which is stored in declarative memory. The `order(focus="sandwich condiment", DMbuffer="condiment:?condiment")` production fires when this has happened.

(8) File **example_3.py**:

```
import ccm
from ccm.lib.actr import *


class MyAgent(ACTR):

    focus = Buffer()
    DMbuffer = Buffer()
    DM = Memory(DMbuffer)

    def init():
        DM.add("condiment:mustard") # put a chunk into DM
        focus.set("sandwich bread")

    def bread_bottom(focus="sandwich bread"):
        print "I have a piece of bread"
        focus.set("sandwich cheese")

    def cheese(focus="sandwich cheese"):
        print "I have put cheese on the bread"
        focus.set("sandwich ham")

    def ham(focus="sandwich ham"):
        print "I have put ham on the cheese"
        focus.set("get_condiment")

    def condiment(focus="get_condiment"):
        print "Recalling the order"
        # retrieve a chunk from DM into the DM buffer
        # ? means that slot can match any content
        DM.request("condiment:?")
        focus.set("sandwich condiment")

    # this production rule matches to DMbuffer as well
```

```
        # and assigns the retrieved value of the condiment slot
        # to the variable ?condiment
        def order(focus="sandwich condiment", DMbuffer="condiment:?condiment"):
            print "I recall they wanted......."
            print condiment
            print "I have put the condiment on the sandwich"
            focus.set("sandwich bread_top")

        def bread_top(focus="sandwich bread_top"):
            print "I have put bread on the ham"
            print "I have made a ham and cheese sandwich"
            focus.set("stop")

        def stop_production(focus="stop"):
            self.stop()


    class MyEnvironment(ccm.Model):
        pass
```

[py8]
```
>>> from example_3 import *
>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 0
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> empty_environment.run()
   0.000 agent.focus.chunk sandwich bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
   0.050 agent.focus.chunk sandwich cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk sandwich ham
   0.100 agent.production ham
   0.150 agent.production None
I have put ham on the cheese
```

```
   0.150 agent.focus.chunk get_condiment
   0.150 agent.production condiment
   0.200 agent.production None
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk sandwich condiment
   0.250 agent.DMbuffer.chunk condiment:mustard
   0.250 agent.DM.busy False
   0.250 agent.production order
   0.300 agent.production None
I recall they wanted.......
mustard
I have put the condiment on the sandwich
   0.300 agent.focus.chunk sandwich bread_top
   0.300 agent.production bread_top
   0.350 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.350 agent.focus.chunk stop
   0.350 agent.production stop_production
   0.400 agent.production None
>>> ccm.finished()
```

5 Memory Retrieval: Sub-symbolic parameters

Memories fade and some become inaccessible. The retrieval of a particular chunk is dependent on the activation level of the chunk. When there are several chunks that match the requested pattern, the chunk with the highest activation is retrieved.

For more complex models, there are a variety of sub-modules which can adjust the activation of the chunks in memory, giving them different probabilities of being retrieved. If no sub-modules are used, then the activation of all chunks is always zero, and the default behavior of the declarative memory system is to randomly choose one of the chunks that match the request.

The main components of the sub-symbolic memory activation system:

- **the base level learning equation**: the activation of a chunk is increased when it is used, and this activation decays over time.
- **partial matching**: a formula for adjusting the activation of chunks that almost match the memory request pattern
- **spreading activation**: a formula for increasing the activation of chunks that are similar to chunks already in buffers
- **noise**: a random amount of noise that leads to variability in behavior

The first step is always to create a retrieval buffer and a basic declarative memory system to go with it, like we did in Example 2 with:

(9) Initializing the declarative memory system (repeated from Example 2):

```
DMbuffer = Buffer()
DM = Memory(DMbuffer)
```

The memory system in Python ACT-R has a variety of parameters:

- `latency`: controls the relationship between activation and how long it takes to recall the chunk
- `threshold`: chunks must have activation greater than or equal to this to be recalled (can be set to `None`)
- `maximum_time`: no retrieval will take longer than this amount of time
- `finst_size`: the FINST system allows you to recall items that have not been recently recalled; this controls how many recent items are remembered
- `finst_time`: controls how recent recalls must be for them to be included in the FINST system.

What are finsts? To avoid the problem of repeatedly retrieving the most active chunk in situations where this is not appropriate, ACT-R has a system for maintaining *fingers of instantiation* (FINSTs). This allows the memory request to indicate that it should not recall a recently retrieved item. See Anderson and Lebiere (1998) for details.

(10) From the ACT-R 6.0 manual (by Dan Bothell, http://act-r.psy.cmu.edu/actr6/reference-manual.pdf):

> The declarative module maintains a record of the chunks which have been retrieved and provides a mechanism which allows one to explicitly retrieve or not retrieve one which is so marked. This is done through the use of a set of finsts (fingers of instantiation) which mark those chunks. The finsts are limited in the number that are available and how long they persist. The number and duration are both controlled by parameters. If more finsts are required than are available, then the oldest one (the one marking the chunk retrieved at the earliest time) is removed and used to mark the most recently retrieved chunk. (p. 226)

Here's an example of initializing the declarative memory system (module + buffer) in which we explicitly use these parameters:

(11) Initializing the declarative memory system and explicitly setting its parameters:

```
DMbuffer = Buffer()
DM = Memory(DMbuffer,\
            latency=0.05,\
            threshold=0,\
            maximum_time=10.0,\
            finst_size=0,\
            finst_time=3.0)
```

The following systems adjust the activation in various ways. They are shown along with the default values for their parameters.

(12) Adjusting activation:

```
# standard random noise
dm_n = DMNoise(DM, noise=0.3, baseNoise=0.0)

# the standard base level learning system
# if limit is set to a number, then the hybrid optimized equation
# from Petrov (2006) is used.
dm_bl = DMBaseLevel(DM, decay=0.5, limit=None)

# the spacing effect system from (Pavlik & Anderson 2005)
dm_space = DMSpacing(DM, decayScale=0.0, decayIntercept=0.5)

# the standard fan-effect spreading activation system
```

```
dm_spread = DMSpreading(DM, focus) # specify the buffer(s) to spread from

# other parameters are configured like this:
dm_spread.strength = 1
dm_spread.weight[focus] = 0.3
```

5.1 Understanding the (basic) activation equation

(13) Activation equation: $A_i = B_i + \sum\limits_{j \in C} W_j S_{ji}$, for a chunk i and elements j that are part of the current goal / focus chunk.
This equation has three major components:

a. Base-level learning equation: $B_i = \log\left(\sum\limits_{k=1}^{n} t_k^{-d}\right) = \log\left(\sum\limits_{k=1}^{n} \frac{1}{\sqrt{t_k}}\right)$ (since usually $d = 0.5$), where t_k is the time since the k-th practice / access of chunk i.
b. Attentional weighting equation: $W_j = \frac{W}{n}$
c. Associative strength equation: $S_{ji} \approx \log\left(\frac{prob(i|j)}{prob(i)}\right)$

5.1.1 The base-level learning equation

(14) Base-level learning equation: $B_i = \log\left(\sum\limits_{k=1}^{n} t_k^{-d}\right) = \log\left(\sum\limits_{k=1}^{n} \frac{1}{\sqrt{t_k}}\right)$ (since usually $d = 0.5$), where t_k is the time since the k-th practice / access of chunk i.

(15) Anderson and Schooler (1991, p. 396):

> In this paper we explore the issue of whether human memory is behaving optimally with respect to the pattern of past information presentation. Each item in memory has had some history of past use. For instance, our memory for one person's name may not have been used in the past month but might have been used five times in the month previous to that. What is the probability that the memory will be needed (used) during the conceived current day? Memory would be behaving optimally if it made this memory less available than memories that were more likely to be used but made it more available than less likely memories.
> In this paper we examine a number of environmental sources to determine how probability of a memory being needed varies with pattern of past use.

Let's first examine the Ebbinghaus (1913) retention data presented in his chapter 7.

(16) **Ebbinghaus (1913, ch. 7) retention data**

a. Stimulus materials: nonsense CVC syllables, about 2300 in number; mixed together, randomly selected to construct series of different lengths.
b. Method: learning to criterion; the subject repeats the material as many times as necessary to reach a prespecified level of accuracy (e.g., one perfect reproduction).
c. Retention measure: 'savings', i.e., subtracting the number of repetitions required to relearn material to a criterion from the number originally required to learn the material to the same criterion.

[py9]
```
>>> import pandas as pd
>>> ebbinghaus_data = pd.DataFrame(pd.read_csv("ebbinghaus_retention_data.csv",
...                                                    header=0))
>>> print ebbinghaus_data
   delay_in_hours  percent_savings
0            0.33             58.2
1            1.00             44.2
2            8.80             35.8
3           24.00             33.7
4           48.00             27.8
5          144.00             25.4
6          744.00             21.1
>>> ebbinghaus_data.mean().round(2)
delay_in_hours     138.59
percent_savings     35.17
dtype: float64
>>> ebbinghaus_data.std().round(2)
delay_in_hours     271.66
percent_savings     12.66
dtype: float64
```

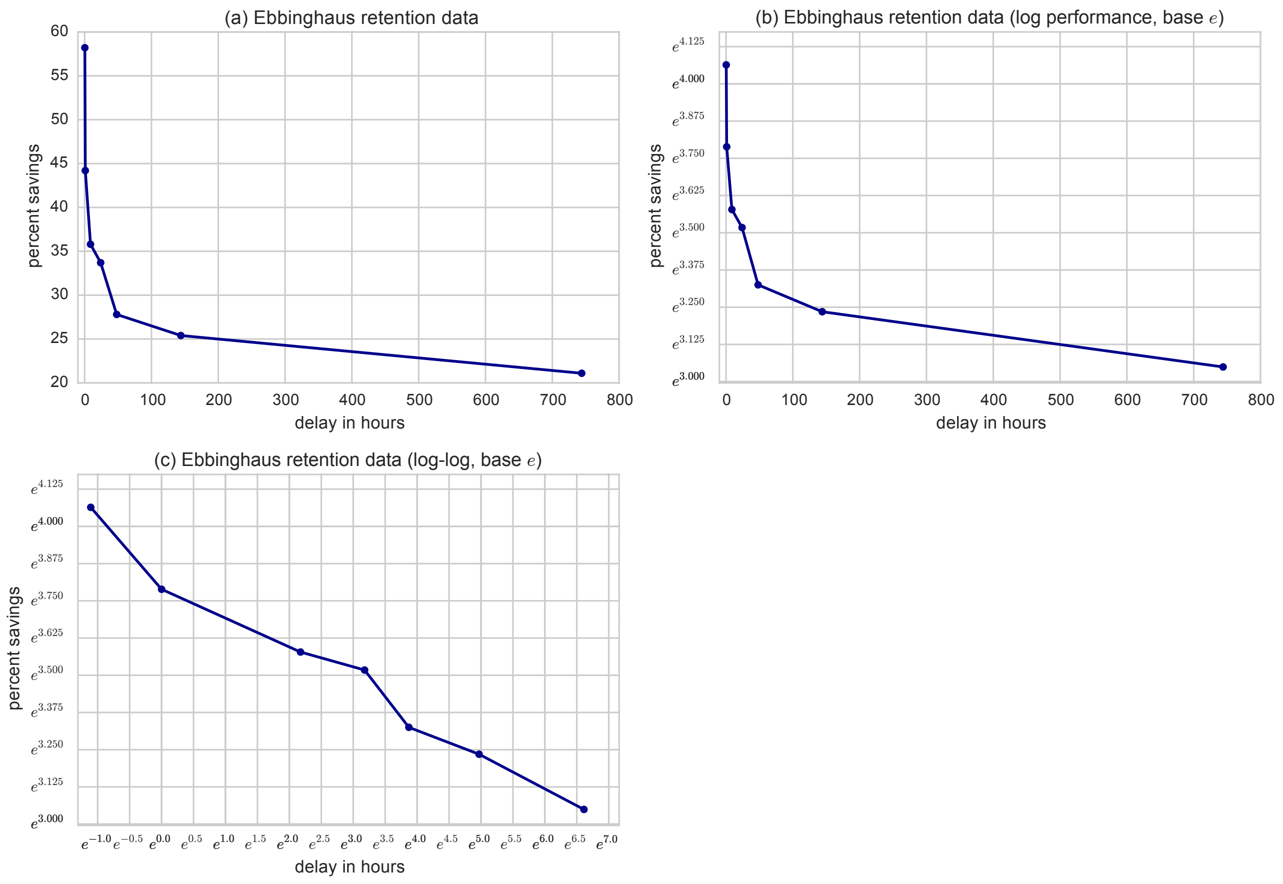

Figure 1: Ebbinghaus retention data.

The forgetting curve plotted in panel (a) of Figure 1 is sometimes taken to reflect an underlying negative exponential forgetting function of the form:

(17) $P = Ae^{-bT}$, where P is the performance measure (percent savings in the Ebbinghaus data), T is the delay in time, and A, b are the parameters of the model.

But this predicts that performance should be a linear function of time if we log-transform P, and panel (b) of Figure 1 shows that is not the case:

(18) $\log(P) = \log(A) - bT$

Instead, we see a power function, as panel (c) of Figure 1 shows. That is, performance is a linear function of time only if you log-transform both of them:

(19) $\log(P) = \log(A) - b\log(T)$, i.e., $\boxed{P = AT^{-b}}$

The base-level learning equation $B_i = \log\left(\sum_{k=1}^{n} t_k^{-d}\right)$ reflects exactly this: the base-level activation B_i is basically a log-performance value.

The basic idea of the account in Anderson and Schooler (1991):

(20) The basic idea is that at any point in time, memories vary in how likely they are to be needed and the memory system tries to make available those memories that are most likely to be useful. The memory system can use the past history of use of a memory to estimate whether the memory is likely to be needed now. This view sees human memory in some sense as making a statistical inference. However, it does not imply that memory is explicitly engaged in statistical computations. Rather, the claim is that whatever memory is doing parallels a correct statistical inference.

What memory is inferring is something we call the need probability, which is the probability that we will need a particular memory trace now. The basic assumption developed in Anderson (1990) is that memories are considered in order of their need probabilities until the need probability is so low that it no longer is worth considering any more. If we let p be the need probability, C be the cost of considering a memory, and G be the gain associated with a successful retrieval, one should stop when $C > pG$.

Despite the description of this process in terms that evoke images of memories being considered one at a time, there are equivalent parallel processes. We prefer a parallel model in which different memories are allocated different resources according to their need probability.

[...]

This analysis does allow predictions to be derived about the relationship between need probability and the dependent measures of recall latency and recall accuracy. With respect to recall latency, the critical assumption is that there is a distribution of memories in terms of their estimated need probabilities. The reasonable assumption is that there will be a mass of need probabilities near zero with a tail of a few higher probability memories; that is, to say the distribution of memories will be J-shaped or highly skewed. It is more convenient to think about the shape of such a distribution in terms of need odds. If p is need probability, then $q = p/(1-p)$ will be need odds. An odds measure has the advantage of varying from zero to infinity. Thus, the expectation is that most memories will have near-zero odds and a rapidly diminishing few will have higher odds. (Anderson and Schooler, 1991, p. 400)

In sum:

(21) The base-level activation equation encodes that (see Anderson and Schooler 1991, p. 407, and Anderson, Bothell, et al. 2004, p. 1042):

a. the strength of a memory trace provides an encoding of its need odds memory performance (base-level activation tracks log odds);

b. the strengths from individual presentations sum to produce a total strength (each presentation has an impact on odds, and the impacts of different presentations add up);

c. strengths of individual presentations decay as a power function of the time (the fact that the impact on odds of an individual presentation decays as a power function produces the power law of forgetting).

Let's work through some examples. Assume we have a fact – it can be an addition fact like the one below, or the lexical representation of a word etc.

(22) a. A chunk of type ADDITION-FACT with slots ADDEND_1, ADDEND_2 and SUM which models the fact 5 + 2 = 7. The slot values are the primitive elements 5, 2 and 7, respectively. Chunks are boxed, whereas primitive elements are simple text. A simple arrow ($\longrightarrow$) signifies that the chunk at the start of the arrow has the value at the end of the arrow in the slot with the name

$$\begin{tikzcd}
5 & \boxed{\text{ADDITION-FACT}} \arrow[l, "\text{ADDEND}_1"'] \arrow[r, "\text{ADDEND}_2"] \arrow[d, "\text{SUM}"] & 2 \\
 & 7 &
\end{tikzcd}$$

that labels the arrow.

b. The same chunk represented as an attribute-value matrix (AVM). We'll use only AVM representations from now on. The various components of the activation equation have been added.

$$\begin{bmatrix} \text{ADDEND}_1\left(S_{ji}\right): & 5\left(W_j\right) \\ \text{ADDEND}_2\left(S_{ji}\right): & 2\left(W_j\right) \\ \text{SUM}: & 7 \end{bmatrix}_{\text{ADDITION-FACT}\left(B_i\right)}$$

Assume this chunk is presented 5 times, once every 300 ms, starting at time 0 ms. We want to plot its base-level activation for the first 3500 ms.

[**py10**]
```
>>> import numpy as np

>>> def base_activation(pres_times, moments):
...     """
...     Input: pres_times and moments of time at which to obtain activation (numpy
...     arrays).
...     Output: base activation values at the moments (numpy array of the same
...     length as moments).
...     """
...     import numpy as np
...     base_act = np.zeros(len(moments))
...     for i in range(len(moments)):
...         base_act[i] = sum(1/np.sqrt(moments[i] - pres_times[pres_times<moments[i]]))
...     base_act[np.where(base_act!=0)] = np.log(base_act[np.where(base_act!=0)])
...     return base_act
...
>>> pres_times = np.linspace(0, 1200, num=5)
>>> moments = np.arange(3500)
>>> base_act = base_activation(pres_times, moments)
```

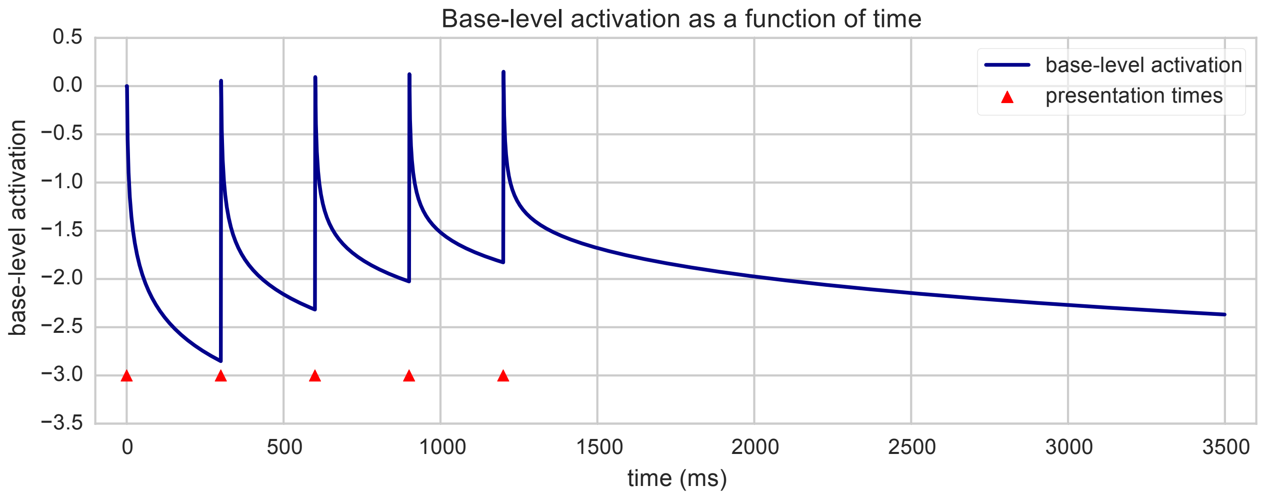

Figure 2: Base-level activation as a function of time: $B_i = \log\left(\sum_{k=1}^{n} t_k^{-d}\right) = \log\left(\sum_{k=1}^{n} \frac{1}{\sqrt{t_k}}\right)$.

5.1.2 The attentional weighting equation

(23) Attentional weighting equation: $W_j = \frac{W}{n}$

W is usually set to 1, so the attention weights are usually $\frac{1}{n}$, where n is the number of sources of activation / terms.

5.1.3 The associative strength equation

(24) Associative strength equation: $S_{ji} \approx \log\left(\frac{prob(i|j)}{prob(i)}\right)$

S_{ji} is usually set to $S - \log(fan_j)$, where fan_j is the number of facts associated with term j. S is usually set to 2.

5.2 Activation, probability of retrieval, and latency of retrieval

(25) Probability of retrieval equation: $P_i = \frac{1}{1+e^{-\frac{A_i-\tau}{s}}}$, where s is the noise parameter and is typically set at about 0.4, and τ the retrieval threshold.

(26) Latency of retrieval equation: $T_i = Fe^{-A_i}$, where F is the latency factor.

(27) The threshold τ and the latency factor F vary from model to model, but there is a general relationship between them:
$F \approx 0.35e^{\tau}$
i.e., the retrieval latency at threshold (when $A_i = \tau$) is approximately 0.35 seconds.

Let's plot the probability and latency of retrieval for the same hypothetical case as above, assuming the activation of the items is just its base-level activation. We assume: noise $s = 0.4$, threshold $\tau = -2$, latency factor $F = 0.46$ (the values in Vasishth et al. 2005, p. 692).

[py11]
```
>>> import numpy as np

>>> pres_times = np.linspace(0, 1200, num=5)
>>> moments = np.arange(3500)
>>> base_act = base_activation(pres_times, moments)

>>> s = 0.4
>>> tau = -2
>>> F = 0.46
>>> prob_retrieval = 1/(1 + np.exp(-(base_act - tau)/s))
>>> latency_retrieval = F * np.exp(-base_act)
```

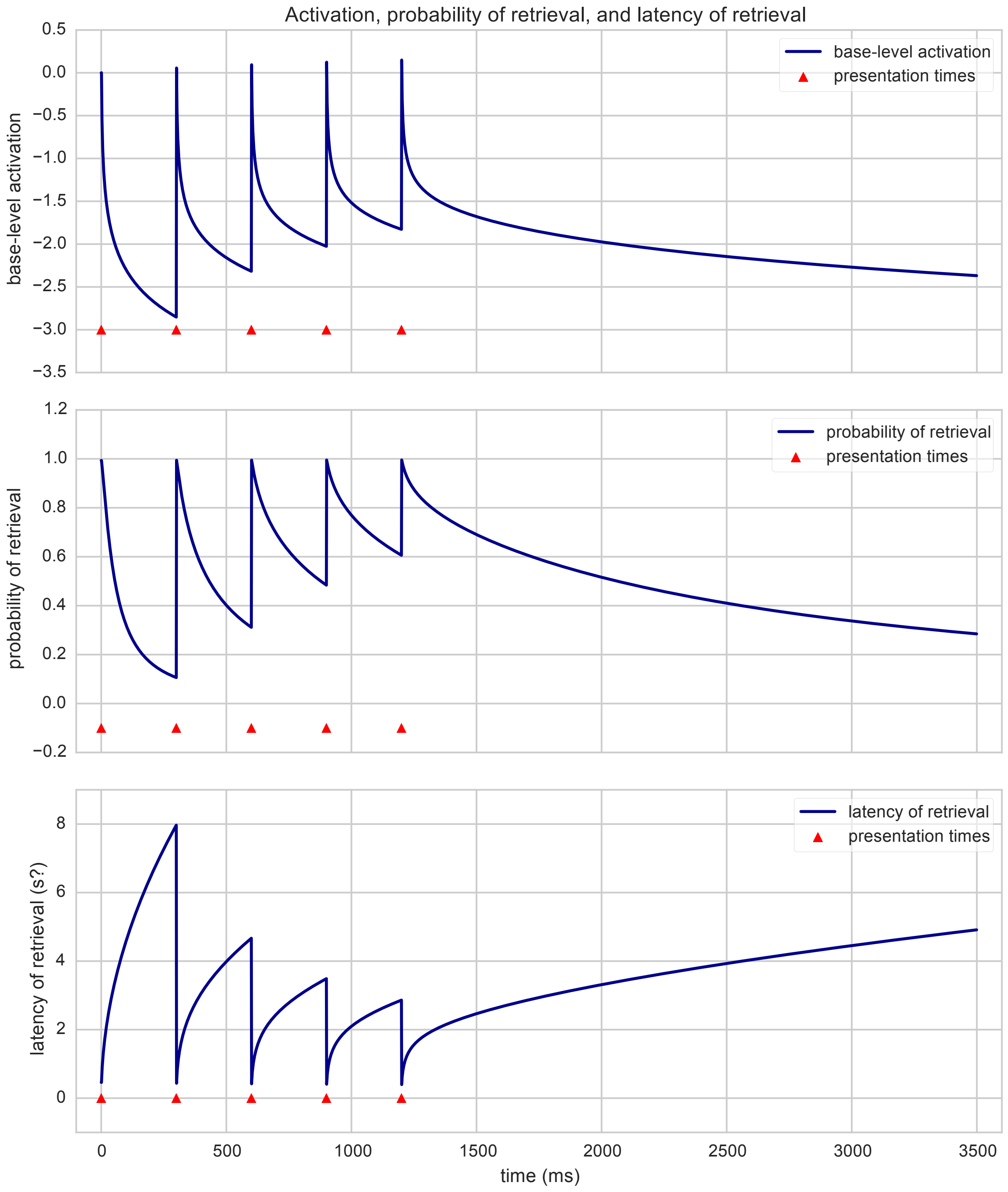

Figure 3: Base-level activation, probability of retrieval, and latency of retrieval as a function of time.

5.3 Example: Forgetting

We will now turn on the sub-symbolic processing for DM, which causes forgetting.

In particular, we will turn on the base-level activation functions for DM. This means the contents of DM decay over time. If the threshold parameter is raised, it will cause the model to forget. So we need an extra production to handle an unsuccessful memory request.

(28) File **example_4_simple_forget.py**:

```
import ccm
from ccm.lib.actr import *


class MyEnvironment(ccm.Model):
    pass


class MyAgent(ACTR):

    focus = Buffer()

    DMbuffer = Buffer()
    # latency controls the relationship between activation and recall
    # activation must be above threshold - can be set to none
    DM = Memory(DMbuffer, latency=0.05, threshold=0)
    # turn on for DM subsymbolic processing
    dm_n = DMNoise(DM, noise=0.0, baseNoise=0.0)
    # turn on for DM subsymbolic processing
    dm_bl = DMBaseLevel(DM, decay=0.5, limit=None)

    def init():
        DM.add("cue:customer condiment:mustard")
        focus.set("sandwich bread")

    def bread_bottom(focus="sandwich bread"):
        print "I have a piece of bread"
        focus.set("sandwich cheese")

    def cheese(focus="sandwich cheese"):
        print "I have put cheese on the bread"
        focus.set("sandwich ham")

    def ham(focus="sandwich ham"):
        print "I have put ham on the cheese"
        focus.set("customer condiment")

    def condiment(focus="customer condiment"):
        print "Recalling the order"
        DM.request("cue:customer condiment:?condiment")
        focus.set("sandwich condiment")
    def order(focus="sandwich condiment", DMbuffer="cue:customer condiment:?condiment"):
        print "I recall they wanted......."
        print condiment
        print "I have put the condiment on the sandwich"
        focus.set("sandwich bread_top")

    # DMbuffer = none means the buffer is empty
    # DM = "error:True" means the search was unsucessful
    def forgot(focus="sandwich condiment", DMbuffer=None, DM="error:True"):
        print "I recall they wanted......."
        print "I forgot"
        focus.set("stop")

    def bread_top(focus="sandwich bread_top"):
        print "I have put bread on the ham"
        print "I have made a ham and cheese sandwich"
        focus.set("stop")

    def stop_production(focus="stop"):
        self.stop()
```

[**py12**]

```
>>> from example_4_simple_forget import *
>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 0
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> empty_environment.run()
   0.000 agent.focus.chunk sandwich bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
   0.050 agent.focus.chunk sandwich cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk sandwich ham
   0.100 agent.production ham
   0.150 agent.production None
```

```
I have put ham on the cheese
   0.150 agent.focus.chunk customer condiment
   0.150 agent.production condiment
   0.200 agent.production None
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk sandwich condiment
   0.222 agent.DMbuffer.chunk condiment:mustard cue:customer
   0.222 agent.DM.busy False
   0.222 agent.production order
   0.272 agent.production None
I recall they wanted.......
mustard
I have put the condiment on the sandwich
   0.272 agent.focus.chunk sandwich bread_top
   0.272 agent.production bread_top
   0.322 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.322 agent.focus.chunk stop
   0.322 agent.production stop_production
   0.372 agent.production None
>>> ccm.finished()
```

5.4 Example: Activation

We continue our exploration of subsymbolic processing for DM. In this model, a chunk is retrieved over and over. Each time a chunk is successfully retrieved in ACT-R, it is referred to as a 'harvest'. In ACT-R theory, when a chunk is harvested it receives a boost in activation. In Python ACT-R, this is not automatic (as it is in LISP ACT-R), so you need to add a `.add` function (method, in Python terms).

With this in place, the activation of the chunk gets stronger and stronger and the time to retrieve it decreases, along the lines of what we saw in the plots above.

(29) File **example_5_simple_activation.py**:

```
import ccm
from ccm.lib.actr import *


class MyEnvironment(ccm.Model):
    pass


class MyAgent(ACTR):

    focus = Buffer()

    DMbuffer = Buffer()
    # latency controls the relationship between activation and recall
    # activation must be above threshold; can be set to None
    DM = Memory(DMbuffer, latency=1.0, threshold=1)
    dm_n = DMNoise(DM, noise=0.0, baseNoise=0.0)
    dm_bl = DMBaseLevel(DM, decay=0.5, limit=None)

    def init():
        DM.add("customer:customer1 condiment:mustard")
        focus.set("rehearse")

    def request_chunk(focus="rehearse"):
        print "recalling the order"
        DM.request("customer:customer1 condiment:?condiment")
        focus.set("recall")

    def recall_chunk(focus="recall", DMbuffer="customer:customer1 condiment:?condiment"):
        print "Customer 1 wants......."
        print condiment
        # each time we put something in memory (DM.add), it increases activation
        DM.add("customer:customer1 ?condiment")
        DMbuffer.clear()
        focus.set("rehearse")

    def forgot(focus="recall", DMbuffer=None, DM="error:True"):
        print "I recall they wanted......."
        print "I forgot"
        focus.set("stop")
```

We run the model just as before, except this time we run it for 2 seconds only, and we also turn on html logging (well, it's commented out in the code listing below because it's not compatible with some PythonTex internals, but you should uncomment it when you run this code yourself).

[**py13**]
```
>>> from example_5_simple_activation import *
>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 1
   0.000 agent.DM.latency 1.0
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> # log = ccm.log(html=True) # we turn on html logging
>>> empty_environment.run(2)
   0.000 agent.focus.chunk rehearse
   0.000 agent.production request_chunk
   0.050 agent.production None
```

```
recalling the order
   0.050 agent.DM.busy True
   0.050 agent.focus.chunk recall
   0.274 agent.DMbuffer.chunk condiment:mustard customer:customer1
   0.274 agent.DM.busy False
   0.274 agent.production recall_chunk
   0.324 agent.production None
Customer 1 wants.......
mustard
   0.324 agent.DMbuffer.chunk None
   0.324 agent.focus.chunk rehearse
   0.324 agent.production request_chunk
   0.374 agent.production None
recalling the order
   0.374 agent.DM.busy True
   0.374 agent.focus.chunk recall
   0.741 agent.DM.error True
   0.741 agent.DMbuffer.chunk None
   0.741 agent.DM.busy False
   0.741 agent.production forgot
   0.791 agent.production None
I recall they wanted.......
I forgot
   0.791 agent.focus.chunk stop
>>> ccm.finished()
```

We can look at the html log – it is generated in a subfolder with the same name as our python script (`example_5_simple_activation`) – and the name of the file includes the date and time when the model was run. So we can run the same model multiple times and compare, aggregate, etc. the simulation outputs.

The code below is based on code from McKinney 2012, pp. 167-169 (the author is the creator of the very useful pandas library).

[**py14**]
```
>>> import os
>>> from urllib2 import urlopen
>>> from lxml.html import parse
>>> import pandas as pd
>>> from pandas.io.parsers import TextParser

>>> dir_path ="." + "/example_5_simple_activation"
>>> html_log_filename = os.listdir(dir_path)[0]
>>> parsed = parse(urlopen("file://" + os.path.abspath(dir_path) + "/" + html_log_filename))
>>> doc = parsed.getroot()
>>> tables = doc.findall(".//table")
>>> row_sets = [table.findall(".//tr") for table in tables]

>>> # we define a function that extracts data from an html table
>>> def unpack(row, kind="td"):
...     """
...     Extracts data from an html table.
...     kind="td" is for data, kind="th" is for headers.
...     """
...     elts = row.findall(".//%s" % kind)
...     return [val.text_content() for val in elts]
...
>>> # we print the results
>>> for row_set in row_sets:
...     for line in [unpack(row, kind="th") for row in row_set]:
...         print line
...     print "\nEnd of header\n"
...     for line in [unpack(row, kind="td") for row in row_set]:
...         print line
...     print "\nEnd of data\n"
...     print "End of table\n"
...
['time', 'agent']
['', 'DM', 'DMbuffer', 'focus', 'production']
['', 'busy', 'chunk', 'chunk', '']
[]
[]
[]
[]
[]
[]
[]

End of header

[]
[]
[]
['0.000', '', '', 'rehearse', 'request_chunk']
['0.050', 'True', 'recall', '']
['0.274', 'False', 'condiment:mustard customer:customer1', 'recall_chunk']
['0.324', '', 'rehearse', 'request_chunk']
['0.374', 'True', 'recall', '']
['0.741', 'False', '', 'forgot']
['0.791', 'stop', '']

End of data

End of table

[]

End of header

['agent.DM.error', 'True']

End of data
End of table

>>> # we see that the first table stores the main simulation data
>>> table = tables[0]

>>> # we define a function to extract data
>>> # from an html table into a pandas dataframe
>>> def parse_data(table):
...     rows = table.findall(".//tr")
...     header = [unpack(row, kind="th") for row in rows]
...     header = [a + "_" + b + "_" + c for (a, b, c) in zip(header[0] + [""] * 3, header[1],
...     data = [unpack(row, kind="td") for row in rows]
...     data = data[3:]
...     data[1].insert(2, "")
...     data[2].insert(3, "")
...     data[3].insert(2, "")
...     data[4].insert(2, "")
...     data[5].insert(2, "")
...     data[6].insert(1, "")
...     data[6].insert(2, "")
...     return TextParser(data, names=header).get_chunk()
...
>>> simulation_data = parse_data(table)
>>> simulation_data
   time__ agent_DM_busy                       _DMbuffer_chunk _focus_chunk  \
0   0.000           NaN                                   NaN     rehearse
1   0.050          True                                   NaN       recall
2   0.274         False  condiment:mustard customer:customer1          NaN
3   0.324           NaN                                   NaN     rehearse
4   0.374          True                                   NaN       recall
5   0.741         False                                   NaN          NaN
6   0.791           NaN                                   NaN         stop

    _production_
0  request_chunk
1            NaN
2   recall_chunk
3  request_chunk
4            NaN
5         forgot
6            NaN
>>> simulation_data.describe(include=["number"])
         time__
count  7.000000
mean   0.364857
std    0.306960
min    0.000000
25%    0.162000
50%    0.324000
```

```
75%    0.557500
max    0.791000
>>> simulation_data.describe(include=["object"])
       agent_DM_busy                      _DMbuffer_chunk _focus_chunk  \
count              4                                    1            5
unique             2                                    1            3
top             True  condiment:mustard customer:customer1     rehearse
freq               2                                    1            2

         _production_
count               4
unique              3
top     request_chunk
freq                2
```

As you can see, it is actually pretty painful to parse these html logs into a dataframe. It would be even harder without the excellent pandas library. If any of you find out an easier way to aggregate data from simulations for subsequent plotting, data analysis, etc., *please* let me know.

5.5 Example: Spreading Activation

This model turns on spreading activation. This means that DM chunks that have slot contents that match what's in the buffers will receive a boost in activation. Note that the threshold is turned up to 1, which would cause forgetting in the previous example.

(30) File **example_6_simple_forget.py**:

```
import ccm
from ccm.lib.actr import *


class MyEnvironment(ccm.Model):
    pass


class MyAgent(ACTR):

    focus = Buffer()

    DMbuffer = Buffer()
    DM = Memory(DMbuffer, latency=0.05, threshold=1)
    dm_n = DMNoise(DM, noise=0.0, baseNoise=0.0)
    dm_bl = DMBaseLevel(DM, decay=0.5, limit=None)
    # turn on spreading activation for DM from focus
    dm_spread = DMSpreading(DM, focus)
    # set strength of activation for buffers
    dm_spread.strength = 2
    # set weight to adjust for how many slots in the buffer
    # usually this is strength divided by number of slots
    dm_spread.weight[focus] = .5
```

```
        def init():
            DM.add("customer:customer condiment:mustard")
            focus.set("sandwich bread")

        def bread_bottom(focus="sandwich bread"):
            print "I have a piece of bread"
            focus.set("sandwich cheese")

        def cheese(focus="sandwich cheese"):
            print "I have put cheese on the bread"
            focus.set("sandwich ham")

        def ham(focus="sandwich ham"):
            print "I have put ham on the cheese"
            focus.set("customer condiment")

        # customer will spread activation to "customer mustard"
        def condiment(focus="customer condiment"):
            print "recalling the order"
            # request gets boost from spreading activation
            DM.request("customer:customer condiment:?condiment")
            focus.set("sandwich condiment")

        def order(focus="sandwich condiment", DMbuffer="customer:customer condiment:?condiment")
            print "I recall they wanted......."
            print condiment
            print "I have put the condiment on the sandwich"
            focus.set("sandwich bread_top")

        def forgot(focus="sandwich condiment", DMbuffer=None, DM="error:True"):
            print "I recall they wanted......."
            print "I forgot"
            focus.set("stop")

        def bread_top(focus="sandwich bread_top"):
            print "I have put bread on the ham"
            print "I have made a ham and cheese sandwich"
            focus.set("stop")

        def stop_production(focus="stop"):
            self.stop()
```

[**py15**]
```
>>> from example_6_simple_spread import *
>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 1
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> # log = ccm.log(html=True)
>>> empty_environment.run()
   0.000 agent.focus.chunk sandwich bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
   0.050 agent.focus.chunk sandwich cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk sandwich ham
   0.100 agent.production ham
   0.150 agent.production None
I have put ham on the cheese
   0.150 agent.focus.chunk customer condiment
   0.150 agent.production condiment
   0.200 agent.production None
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk sandwich condiment
   0.212 agent.DMbuffer.chunk condiment:mustard customer:customer
   0.212 agent.DM.busy False
   0.212 agent.production order
   0.262 agent.production None
I recall they wanted.......
mustard
I have put the condiment on the sandwich
   0.262 agent.focus.chunk sandwich bread_top
   0.262 agent.production bread_top
   0.312 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.312 agent.focus.chunk stop
   0.312 agent.production stop_production
   0.362 agent.production None
>>> ccm.finished()
```

5.6 Example: Partial Matching

In this model partial matching is turned on and the similarity between chunks is set. Therefore, when we add some noise it is possible for the model to recall the wrong chunk, especially if it has a high similarity level. The noise also makes it possible that the model will fail to make a retrieval at all. Run the model several times to see all of these effects.

(31) File **example_7_simple_partial.py**:

```
import ccm
from ccm.lib.actr import *


class MyEnvironment(ccm.Model):
    pass


class MyAgent(ACTR):
    focus = Buffer()

    DMbuffer = Buffer()
    DM = Memory(DMbuffer, latency=0.05, threshold=1)
    # turn on some noise to allow errors
    dm_n = DMNoise(DM, noise=0.6, baseNoise=0.0)
    dm_bl = DMBaseLevel(DM, decay=0.5, limit=None)
    dm_spread = DMSpreading(DM, focus)
    dm_spread.strength = 2
    dm_spread.weight[focus] = .5
    # turn on partial matching
    partial = Partial(DM, strength=1.0, limit=-1.0)
    # set the similarity between customer1 and customer2 - they are very similar
    partial.similarity("customer1", "customer2", -0.1)
    # set the similarity between customer1 and customer3 - not so similar
    partial.similarity("customer1", "customer3", -0.9)

    def init():
        DM.add("customer:customer1 condiment:mustard")     # customer1's order
        DM.add("customer:customer2 condiment:ketchup")     # customer2's order
        DM.add("customer:customer3 condiment:mayonnaise")  # customer3's order
        focus.set("sandwich bread")

    def bread_bottom(focus="sandwich bread"):
        print "I have a piece of bread"
        focus.set("sandwich cheese")

    def cheese(focus="sandwich cheese"):
        print "I have put cheese on the bread"
        focus.set("sandwich ham")

    def ham(focus="sandwich ham"):
        print "I have put ham on the cheese"
        focus.set("customer1 condiment")

    # customer1 will spread activation to "customer1 mustard"
    # but also some to "customer2 ketchup" and less to "customer3 mayonaise"
    def condiment(focus="customer1 condiment"):
        print "Recalling the order"
        DM.request("customer:customer1 condiment:?condiment")
        focus.set("sandwich condiment")

    def order(focus="sandwich condiment", DMbuffer="customer:? condiment:?condiment"):
        print "I recall they wanted......."
        print condiment
        print "I have put the condiment on the sandwich"
        focus.set("sandwich bread_top")

    def forgot(focus="sandwich condiment", DMbuffer=None, DM="error:True"):
        print "I recall they wanted......."
        print "I forgot"
        focus.set("stop")

    def bread_top(focus="sandwich bread_top"):
        print "I have put bread on the ham"
        print "I have made a ham and cheese sandwich"
        focus.set("stop")

    def stop_production(focus="stop"):
        self.stop()
```

[**py16**]
```
>>> from example_7_simple_partial import *

>>> # run 1

>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 1
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> # log = ccm.log(html=True)
>>> empty_environment.run()
   0.000 agent.focus.chunk sandwich bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
```

```
   0.050 agent.focus.chunk sandwich cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk sandwich ham
   0.100 agent.production ham
   0.150 agent.production None
I have put ham on the cheese
   0.150 agent.focus.chunk customer1 condiment
   0.150 agent.production condiment
   0.200 agent.production None
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk sandwich condiment
   0.218 agent.DM.error True
   0.218 agent.DMbuffer.chunk None
   0.218 agent.DM.busy False
   0.218 agent.production forgot
   0.268 agent.production None
I recall they wanted.......
I forgot
   0.268 agent.focus.chunk stop
   0.268 agent.production stop_production
   0.318 agent.production None
>>> ccm.finished()

>>> # run 2

>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 1
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> # log = ccm.log(html=True)
>>> empty_environment.run()
   0.000 agent.focus.chunk sandwich bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
```

```
   0.050 agent.focus.chunk sandwich cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk sandwich ham
   0.100 agent.production ham
   0.150 agent.production None
I have put ham on the cheese
   0.150 agent.focus.chunk customer1 condiment
   0.150 agent.production condiment
   0.200 agent.production None
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk sandwich condiment
   0.214 agent.DMbuffer.chunk condiment:mustard customer:customer1
   0.214 agent.DM.busy False
   0.214 agent.production order
   0.264 agent.production None
I recall they wanted.......
mustard
I have put the condiment on the sandwich
   0.264 agent.focus.chunk sandwich bread_top
   0.264 agent.production bread_top
   0.314 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.314 agent.focus.chunk stop
   0.314 agent.production stop_production
   0.364 agent.production None
>>> ccm.finished()

>>> # run 3

>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold 1
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 10.0
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> # log = ccm.log(html=True)
>>> empty_environment.run()
   0.000 agent.focus.chunk sandwich bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
   0.050 agent.focus.chunk sandwich cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk sandwich ham
   0.100 agent.production ham
   0.150 agent.production None
I have put ham on the cheese
   0.150 agent.focus.chunk customer1 condiment
   0.150 agent.production condiment
   0.200 agent.production None
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk sandwich condiment
   0.218 agent.DMbuffer.chunk condiment:ketchup customer:customer2
   0.218 agent.DM.busy False
   0.218 agent.production order
   0.268 agent.production None
I recall they wanted.......
ketchup
I have put the condiment on the sandwich
   0.268 agent.focus.chunk sandwich bread_top
   0.268 agent.production bread_top
   0.318 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.318 agent.focus.chunk stop
   0.318 agent.production stop_production
   0.368 agent.production None
>>> ccm.finished()
```

5.7 Example: Refraction

In ACT-R, a retrieval makes a chunk stronger so it is more likely to be recalled the next time. However, in some cases this is problematic. For example, if you are recalling different types of animals and the tiger has the highest activation it will be recalled first, and also second, and third, and so on. So there is a way to tell a production to retrieve a chunk that has not recently been recalled. In this model a sandwich maker has several different orders and makes them in the order he recalls them.

(32) File **example_8_simple_refraction.py**:

```
import ccm
from ccm.lib.actr import *


class MyEnvironment(ccm.Model):
    pass


class MyAgent(ACTR):

    focus = Buffer()

    DMbuffer = Buffer()
    # turn down threshold
    # maximum time - how long it will wait for a memory retrieval
    # finst_size - how many chunks can be kept track of
    # finst_time - how long a chunk can be kept track of
    DM = Memory(DMbuffer, latency=0.05, threshold=-25, maximum_time=20, finst_size=10, finst_
    dm_n = DMNoise(DM, noise=0.0, baseNoise=0.0)
    dm_bl = DMBaseLevel(DM, decay=0.5, limit=None)
    dm_spread = DMSpreading(DM, focus)
    dm_spread.strength = 2
    dm_spread.weight[focus] = .5
    partial = Partial(DM, strength=1.0, limit=-1.0)
    partial.similarity("customer1", "customer2", -0.1)
    partial.similarity("customer1", "customer3", -0.9)

    # note that this model uses slot names - slotname:slotcontent
    def init():
        # customer1's order
        DM.add("isa:order customer:customer1 type:ham_cheese condiment:mustard")
        # customer2's order
        DM.add("isa:order customer:customer2 type:ham_cheese condiment:ketchup")
        # customer3's order
        DM.add("isa:order customer:customer3 type:ham_cheese condiment:mayonnaise")
        # customer4's order
        DM.add("isa:order customer:customer4 type:ham_cheese condiment:hot_sauce")
        focus.set("isa:ingredient type:bread")

    def bread_bottom(focus="isa:ingredient type:bread"):
        print "I have a piece of bread"
        focus.set("isa:ingredient type:cheese")

    def cheese(focus="isa:ingredient type:cheese"):
        print "I have put cheese on the bread"
        focus.set("isa:ingredient type:ham")

    def ham(focus="isa:ingredient type:ham"):
        print "I have put ham on the cheese"
        focus.set("isa:order customer:customer1 type:ham_cheese condiment:unknown")

    def condiment(focus="isa:order customer:customer1 type:ham_cheese condiment:unknown"):
        print "recalling the order"
        # retrieve something that has not recently been retrieved
        DM.request("isa:order type:ham_cheese", require_new=True)
```

```
            focus.set("retrieve_condiment")

        def order(focus="retrieve_condiment",
                  DMbuffer="isa:order type:ham_cheese condiment:?condiment_order"):
            print "I recall they wanted......."
            print condiment_order
            print "I have put the condiment on the sandwich"
            focus.set("isa:ingredient type:bread_top")

        def bread_top(focus="isa:ingredient type:bread_top"):
            print "I have put bread on the ham"
            print "I have made a ham and cheese sandwich"
            focus.set("isa:ingredient type:bread")
            # clear the buffer for the next cycle
            DMbuffer.clear()
```

[**py17**]
```
>>> from example_8_simple_refraction import *
>>> tim = MyAgent()
>>> empty_environment = MyEnvironment()
>>> empty_environment.agent = tim
>>> ccm.log_everything(empty_environment)
   0.000 agent.production_threshold None
   0.000 agent.production_time_sd None
   0.000 agent.production_match_delay 0
   0.000 agent.production_time 0.05
   0.000 agent.DM.record_all_chunks False
   0.000 agent.DM.threshold -25
   0.000 agent.DM.latency 0.05
   0.000 agent.DM.busy False
   0.000 agent.DM.maximum_time 20
   0.000 agent.DM.error False
   0.000 agent.DMbuffer.chunk None
   0.000 agent.focus.chunk None
>>> # log = ccm.log(html=True)
>>> empty_environment.run(3)
   0.000 agent.focus.chunk isa:ingredient type:bread
   0.000 agent.production bread_bottom
   0.050 agent.production None
I have a piece of bread
   0.050 agent.focus.chunk isa:ingredient type:cheese
   0.050 agent.production cheese
   0.100 agent.production None
I have put cheese on the bread
   0.100 agent.focus.chunk isa:ingredient type:ham
   0.100 agent.production ham
   0.150 agent.production None
I have put ham on the cheese
   0.150 agent.focus.chunk condiment:unknown customer:customer1 isa:order type:ham_cheese
   0.150 agent.production condiment
   0.200 agent.production None
```

```
Recalling the order
   0.200 agent.DM.busy True
   0.200 agent.focus.chunk retrieve_condiment
   0.208 agent.DMbuffer.chunk condiment:mustard customer:customer1 isa:order type:ham_cheese
   0.208 agent.DM.busy False
   0.208 agent.production order
   0.258 agent.production None
I recall they wanted.......
mustard
I have put the condiment on the sandwich
   0.258 agent.focus.chunk isa:ingredient type:bread_top
   0.258 agent.production bread_top
   0.308 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.308 agent.focus.chunk isa:ingredient type:bread
   0.308 agent.DMbuffer.chunk None
   0.308 agent.production bread_bottom
   0.358 agent.production None
I have a piece of bread
   0.358 agent.focus.chunk isa:ingredient type:cheese
   0.358 agent.production cheese
   0.408 agent.production None
I have put cheese on the bread
   0.408 agent.focus.chunk isa:ingredient type:ham
   0.408 agent.production ham
   0.458 agent.production None
I have put ham on the cheese
   0.458 agent.focus.chunk condiment:unknown customer:customer1 isa:order type:ham_cheese
   0.458 agent.production condiment
   0.508 agent.production None
Recalling the order
   0.508 agent.DM.busy True
   0.508 agent.focus.chunk retrieve_condiment
   0.532 agent.DMbuffer.chunk condiment:hot_sauce customer:customer4 isa:order type:ham_cheese
   0.532 agent.DM.busy False
   0.532 agent.production order
   0.582 agent.production None
I recall they wanted.......
hot_sauce
I have put the condiment on the sandwich
   0.582 agent.focus.chunk isa:ingredient type:bread_top
   0.582 agent.production bread_top
   0.632 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.632 agent.focus.chunk isa:ingredient type:bread
   0.632 agent.DMbuffer.chunk None
   0.632 agent.production bread_bottom
   0.682 agent.production None
```

```
I have a piece of bread
   0.682 agent.focus.chunk isa:ingredient type:cheese
   0.682 agent.production cheese
   0.732 agent.production None
I have put cheese on the bread
   0.732 agent.focus.chunk isa:ingredient type:ham
   0.732 agent.production ham
   0.782 agent.production None
I have put ham on the cheese
   0.782 agent.focus.chunk condiment:unknown customer:customer1 isa:order type:ham_cheese
   0.782 agent.production condiment
   0.832 agent.production None
Recalling the order
   0.832 agent.DM.busy True
   0.832 agent.focus.chunk retrieve_condiment
   0.863 agent.DMbuffer.chunk condiment:mayonnaise customer:customer3 isa:order type:ham_chees
   0.863 agent.DM.busy False
   0.863 agent.production order
   0.913 agent.production None
I recall they wanted.......
mayonnaise
I have put the condiment on the sandwich
   0.913 agent.focus.chunk isa:ingredient type:bread_top
   0.913 agent.production bread_top
   0.963 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   0.963 agent.focus.chunk isa:ingredient type:bread
   0.963 agent.DMbuffer.chunk None
   0.963 agent.production bread_bottom
   1.013 agent.production None
I have a piece of bread
   1.013 agent.focus.chunk isa:ingredient type:cheese
   1.013 agent.production cheese
   1.063 agent.production None
I have put cheese on the bread
   1.063 agent.focus.chunk isa:ingredient type:ham
   1.063 agent.production ham
   1.113 agent.production None
I have put ham on the cheese
   1.113 agent.focus.chunk condiment:unknown customer:customer1 isa:order type:ham_cheese
   1.113 agent.production condiment
   1.163 agent.production None
Recalling the order
   1.163 agent.DM.busy True
   1.163 agent.focus.chunk retrieve_condiment
   1.199 agent.DMbuffer.chunk condiment:ketchup customer:customer2 isa:order type:ham_cheese
   1.199 agent.DM.busy False
   1.199 agent.production order
   1.249 agent.production None
```

```
I recall they wanted.......
ketchup
I have put the condiment on the sandwich
   1.249 agent.focus.chunk isa:ingredient type:bread_top
   1.249 agent.production bread_top
   1.299 agent.production None
I have put bread on the ham
I have made a ham and cheese sandwich
   1.299 agent.focus.chunk isa:ingredient type:bread
   1.299 agent.DMbuffer.chunk None
   1.299 agent.production bread_bottom
   1.349 agent.production None
I have a piece of bread
   1.349 agent.focus.chunk isa:ingredient type:cheese
   1.349 agent.production cheese
   1.399 agent.production None
I have put cheese on the bread
   1.399 agent.focus.chunk isa:ingredient type:ham
   1.399 agent.production ham
   1.449 agent.production None
I have put ham on the cheese
   1.449 agent.focus.chunk condiment:unknown customer:customer1 isa:order type:ham_cheese
   1.449 agent.production condiment
   1.499 agent.production None
Recalling the order
   1.499 agent.DM.busy True
   1.499 agent.focus.chunk retrieve_condiment
>>> ccm.finished()
```

References

Anderson, John R. (1990). *The Adaptive Character of Thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, John R., Daniel Bothell, et al. (2004). "An Integrated Theory of the Mind". In: *Psychological Review* 111.4, pp. 1036–1060.

Anderson, John R. and Christian Lebiere (1998). *The Atomic Components of Thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, John R. and Lael J. Schooler (1991). "Reflections of the Environment in Memory". In: *Psychological Science* 2.6, pp. 396–408.

Ebbinghaus, Hermann (1913). *Memory: A Contribution to Experimental Psychology*. Trans. by Henry A. Ruger and Clara E. Bussenius. New York: Teachers College, Columbia University. URL: http://psychclassics.yorku.ca/Ebbinghaus/index.htm.

Lewandowsky, S. and S. Farrell (2010). *Computational Modeling in Cognition: Principles and Practice*. Thousand Oaks, CA, USA: SAGE Publications.

McKinney, W. (2012). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media.

Poore, Geoffrey M. (2013). "Reproducible Documents with PythonTeX". In: *Proceedings of the 12th Python in Science Conference*. Ed. by Stéfan van der Walt et al., pp. 78–84.

Stewart, Terrence C. (2007). "A Methodology for Computational Cognitive Modelling". PhD thesis. Ottawa, Ontario: Carleton University.

Stewart, Terrence C. and Robert L. West (2007). "Deconstructing and reconstructing ACT-R: Exploring the architectural space". In: *Cognitive Systems Research* 8, pp. 227–236.

Vasishth, Shravan et al. (2005). "Processing Polarity: How the Ungrammatical Intrudes on the Grammatical". In: *Cognitive Science* 32, pp. 685–712.