Computing Dynamic Meanings

Adrian Brasoveanu, Jakub Dotlačil[1]

November 29, 2016

[1]ACKNOWLEDGMENTS to be inserted here ... This document has been created with LaTeX and PythonTex (Poore, 2013). The usual disclaimers apply.

Contents

1 Introduction **7**

1.1 Using pyactr – people familiar with Python . . . 7

1.2 Using pyactr – beginners . . . 7

2 Basics of ACT-R **9**

2.1 Introduction . . . 9

2.2 Why do we care about ACT-R, and cognitive architectures and modeling in general . . . 10

2.3 Knowledge in ACT-R . . . 11

2.3.1 Representing declarative knowledge: chunks . . . 11

2.3.2 Representing procedural knowledge: productions . . . 12

2.4 Using `pyactr` . . . 12

2.5 Writing chunks in `pyactr` . . . 12

2.6 Modules and buffers . . . 15

2.7 Writing productions in `pyactr` . . . 16

2.8 More examples on queries . . . 18

2.9 Running a model . . . 19

2.10 Example 2 – a top-down parser . . . 20

2.10.1 First steps in the model . . . 21

2.10.2 Production rules . . . 23

2.10.3 Running the model . . . 27

2.10.4 Stepping through a model . . . 31

2.10.5 Exercises . . . 32

2.11 The environment in ACT-R . . . 33

2.11.1 Introduction . . . 33

2.11.2 A simple lexical decision task . . . 33

2.11.3 Motor module . . . 35

2.11.4 Vision in ACT-R . . . 39

2.11.5 Manual processes in ACT-R . . . 41

2.11.6 Exercises . . . 41

3 Performance **49**

3.1 Introduction . . . 49

3.2 Understanding the (basic) activation equation . . . 49

3.2.1 The base-level learning equation . . . 50

3.2.2 The attentional weighting equation 56
3.2.3 The associative strength equation 56
3.3 Activation, probability of retrieval, and latency of retrieval 56
3.3.1 Probability of retrieval 59
3.3.2 Latency of retrieval 63
3.4 Modelling performance 65
3.4.1 Modelling lexical decision tasks 65
3.5 pyactr model of lexical decision 68
3.6 Exercises 73
3.6.1 Exercise 1 73

List of Figures

1.1 Opening Bash in PythonAnywhere. 8

3.1 Ebbinghaus retention data 52
3.2 Base-level activation as a function of time 55
3.3 Base-level activation, probability of retrieval, and latency of retrieval as a function of time 58
3.4 Base-level activation, probability of retrieval, and odds of retrieval as a function of time 60
3.5 Probability and odds of retrieval as a function of activation 62
3.6 Time of retrieval as a function of activation and as a function of odds of retrieval 64
3.7 Model fitting of Exp. 1, Murray and Forster (2004). The solid line represents the best fit of the f parameter, the dashed line is the best fit of the d parameter, the dotted line is the best fit of log-frequency to latencies. 68

Chapter 1

Introduction

– overview of the book, intended audience, getting started (installation instructions etc.)

1.1 Using pyactr – people familiar with Python

If you are familiar with Python, you can install `pyactr` (the Python package that enables ACT-R) and proceed to Chapter 2. `pyactr` is a Python 3 package and can be installed using pip (for Python 3):

```
$ pip3 install pyactr
```

Alternatively, you can download the package here: https://github.com/jakdot/pyactr and follow the instructions there to install the package.

If you are not familiar with Python, you should consider the steps below.

1.2 Using pyactr – beginners

`pyactr` is a package in Python 3. To get started, you should consider a web-based service for Python 3 like `PythonAnywhere`. In this type of services, computation is hosted on separate servers and you don't have to install anything on your computer (of course, you'll need Internet access). If you find you like working with Python and `pyactr`, you can install them on your computer at a later point together with a good text editor for code – or install an integrated desktop environment (IDE) for Python – a common choice is `anaconda`, which comes with a variety of ways of working interactively with Python (IDE with Spyder as the editor, `ipython` notebooks etc.). But none of this is required to run `pyactr` and the code in this book.

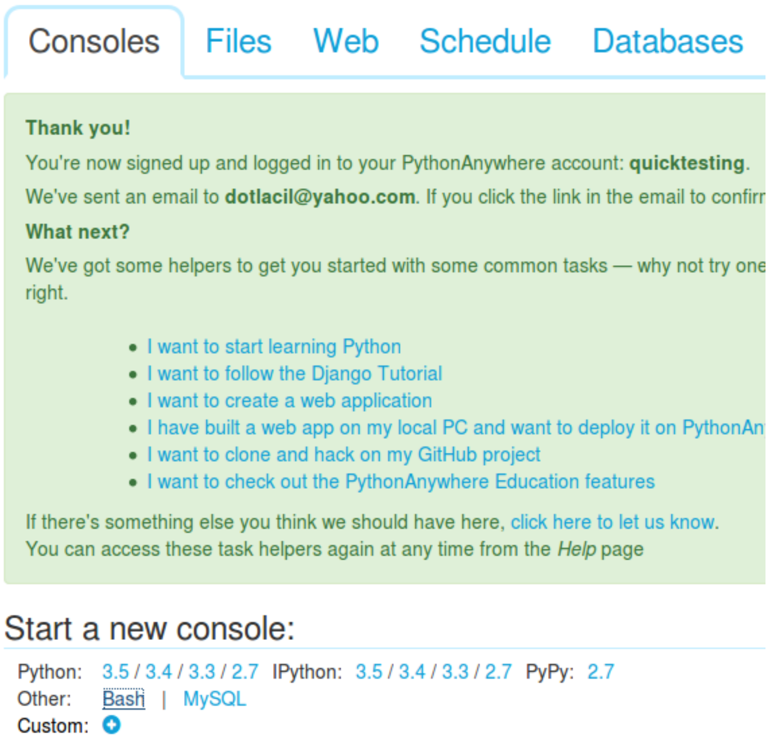
a. Go to www.pythonanywhere.com and sign up there.

b. You'll receive a confirmation e-mail. Confirm your account.

c. Log into your account on www.pythonanywhere.com.

d. You should see a window like the one below. Click on Bash (below "Start a new Console").

Figure 1.1: Opening Bash in PythonAnywhere.



e. In Bash, type:

```
$ pip3 install --user pyactr
```

This will install `pyactr` in your Python account (not on your computer).

f. Go back to Consoles. Start Python by clicking on any version higher than 3.2.

g. A console should open. Type:

```python
import pyactr
```

If no errors appear, you are set and can proceed to Chapter 2.

Throughout the book, we will introduce and discuss various ACT-R models coded in Python. You can either type them in line by line or even better, load them as files in your session on PythonAnywhere. Scripts are uploaded under the tab Files. You should be aware that the free account of PythonAnywhere allows you to run only two consoles, and there is a limit on the amount of CPU you might use per day. The limit should suffice for the tutorials but if you find this too constraining, you should consider installing Python (Python 3) and `pyactr` on your computer and running scripts directly there.

Chapter 2

Basics of ACT-R

2.1 Introduction

ACT-R is a cognitive architecture. It is a theory of the structure of the brain that explains and predicts human cognition. The theory of ACT-R has been implemented in several programming languages, including Java (jACT-R, Java ACT-R), Swift (PRIM), Python2 (ccm). The canonical implementation has been created and is maintained in Lisp. In this book, we will use a novel Python (Python3) implementation (`pyactr`). This implementation is very close to the official implementation in Lisp, so once you learn it, you should be able to transfer your skills very quickly to code models in Lisp ACT-R if you wish to do that. At the same time, since Python is currently much more widespread than Lisp, coding parts that do not directly pertain to the ACT-R model (like data manipulation and data munging, interaction with environment etc.) are much better supported than the same tasks in Lisp. In that way, the programming language stands less in a way of your learning ACT-R than it does in case of Lisp, and you can fully focus on learning nuts and bolts of the cognitive models.

This book and the models we build and discuss are not intended as a reference manual for ACT-R. For learning theories of the model, rather than programming in the model itself, consider (Anderson, 1990; Anderson and Lebiere, 1998; Anderson et al., 2004; Anderson, 2007, a.o.). The main goal of this book is to take a hands-on approach to introducing ACT-R by constructing models that solve (or attempt to solve) linguistic problems. We will mix theoretical notes and `pyactr` code.

In general, we will display python code and its associated output in numbered examples and / or numbered blocks.

For example, when we want to discuss the code, we will display it as:

(1)
```
1 2 + 2 == 4
2 3 + 2 == 6
```

Note the numbers on the left – we can use them to refer to specific lines of code, e.g.: the equality in (1), line `1` is true, while the equality in (1), line `2` is false. We will sometime also include in-line Python code, displayed like this: `2 + 2 == 4`.

When we want to discuss both the code and its output, we will display it in the same way it would appear in your interactive Python interpreter, for example:

[**py1**]
```
>>> 2 + 2 == 4
True
>>> 3 + 2 == 6
False
```

Once again, all lines are numbered (both the Python code and its output) so that we can refer back to it.

2.2 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 cognitive science 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 (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, 25)

In 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, 19)

2.3 Knowledge in ACT-R

There are two types of knowledge in ACT-R: declarative knowledge and procedural knowledge (see also Newell 1990).

The declarative knowledge represents our knowledge of facts. For example, if one knows what the capital of the Netherlands is, this would be represented in one's declarative knowledge.

Procedural knowledge is knowledge that we display in our behavior (cf. Newell 1973). It is often the case that our procedural knowledge is internalized, we are aware that we have it but we would be hard pressed to explicitly and precisely describe it. Driving, swimming, riding a bicycle are examples of procedural knowledge. Almost all people who can drive / swim / ride a bicycle do so in an automatic way. They are able to do it but they might completely fail to describe how exactly they do it when asked. This distinction is closely related to the distinction between explicit ('know that') and implicit ('know how') knowledge in analytical philosophy (see Ryle 1949 and Polanyi 1967; see also Davies 2001 and references therein for more recent discussions).

The two parts of knowledge in ACT-R are represented in two very different ways. The declarative knowledge is instantiated in chunks. The procedural knowledge is instantiated in production rules, or productions for short.

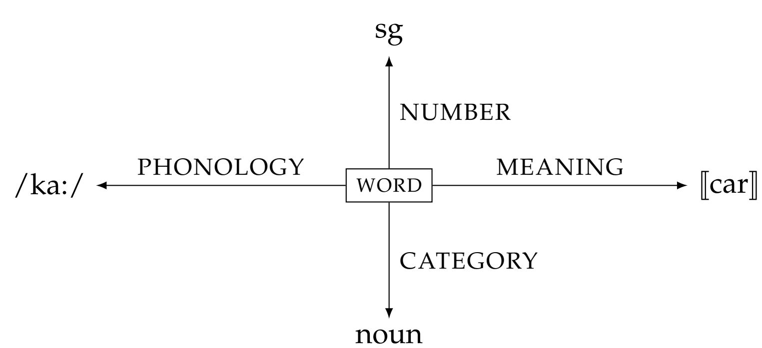
2.3.1 Representing declarative knowledge: chunks

!!! CONTINUE HERE

Chunks are lists of attribute-value pairs, familiar to linguists from phrase structure grammars (e.g., LFG and HPSG). However, in ACT-R, we use the term *slot* instead of *attribute*. For example, we might think of one's knowledge of the word *car* as a chunk of type WORD with the value /ka:/ for the slot *phonology*, the value ⟦car⟧ for the slot *meaning*, the value noun for the slot *category* and the value sg for the slot *number*.

The slot values are the primitive elements /ka:/, ⟦car⟧, noun and sg, 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 that labels the arrow.

(2)



The graph representation will be useful when we introduce activations and more generally, ACT-R subsymbolic components. The same chunk can be represented as an attribute-value matrix (AVM), and we'll overwhelmingly use AVM representations from now on.

(3) $$\left[\begin{array}{ll} \text{PHONOLOGY:} & /\text{ka:}/ \\ \text{MEANING:} & [\![\text{car}]\!] \\ \text{CATEGORY:} & \text{noun} \\ \text{NUMBER:} & \text{sg} \end{array}\right]_{\text{WORD}}$$

2.3.2 Representing procedural knowledge: productions

A production is an if-statement. It describes an action that takes place if the if-part is satisfied. For example, agreement on a verb can be (abstractly) expressed as follows: IF subject number in currently constructed sentence is sg THEN verb number in currently constructed sentence is sg. Of course, this is only half of the story – another rule would state: IF subject number in currently constructed sentence is pl THEN verb number in currently constructed sentence is pl. To repeat the basic intuition about the construction of these rules: productions specify conditions (the if-part of the statement); if these conditions are true, then actions take place (the THEN part of the statement).

Sticking with the example in the previous paragraph, it might look like a roundabout way of specifying agreement. Could we not state that the verb has the same number that the subject has? In fact, we can, if we use variables. Variables are assigned their value when they appear on the left side of a production. The variable keeps its value inside a rule (i.e., a rule is the scope for any variable assignment). Given that (and given the convention that variables are signaled in ACT-R using '='), we could write: IF subject number in currently constructed sentence is =x THEN verb number in currently constructed sentence is =x.

2.4 Using `pyactr`

After this brief introduction, we will continue by combining the theoretical part of ACT-R with discussing how it is implemented in `pyactr`. We will begin with describing details of declarative knowledge in ACT-R and its implementation in `pyactr`. After that we turn to the discussion of modules and buffers, which is needed before we can turn to the second type of knowledge in ACT-R, productions.

But as the very first thing, we have to import the relevant package:

[**py2**]
```
>>> import pyactr as actr
```

We use the `as` keyword, so that every time we use the `pyactr` package, we can write `actr` instead of the longer `pyactr`.

2.5 Writing chunks in `pyactr`

There is one thing we have to do before writing chunks themselves: we should start by specifying a chunk type and all the slots you think it should have. This will help you be clear about your intentions on what should be carried in declarative memory from the start. Let's create a chunk type that will correspond to our knowledge of words, as indicated above.

Needless to say, we don't strive here for the linguistically realistic theory of word representations at this point. It is just a toy example, showing the inner workings of ACT-R. Anyway, here is our chunk type:

[**py3**]
```
>>> actr.chunktype("word", "phonology, meaning, category, number")
```

The function `chunktype` creates a type `word`, which consists of the following slots: `phonology`, `meaning`, `category`, `number`. The type itself is written as the first argument of the function, the slots are written as the second argument and are separated by commas.

After declaring the chunk type, we can create new chunks using this type.

[**py4**]
```
1 >>> car = actr.makechunk(nameofchunk="car",\
2 ...                      typename="word",\
3 ...                      phonology="/ka:/",\
4 ...                      meaning="[[car]]",\
5 ...                      category="noun",\
6 ...                      number="sg")
7 >>> print(car)
8 word(category=noun, meaning=[[car]], number=sg, phonology=/ka:/)
```

The chunk is created using the function `makechunk`. Every `makechunk` has two fixed arguments: `nameofchunk` ([**py4**], line 1), `typename` ([**py4**], line 2). Furthermore, it has slot-value pairs, present in the chunk. Lines 3-6 show how values of slots are specified. You do not have to specify all the slots that a chunk of a particular type should have (in that case, the particular slots are empty). We finally print the chunk (line 7). Notice that the order of slot-value pairs is different than in instantiating the chunk (i.e., we defined `phonology` as first, but it appears as the last in the output). This is because chunks are unordered lists of slot-value pairs. Python assumes some arbitrary (alphabetic) ordering when printing chunks.

Specifying chunk types is optional. In fact, the information about chunk type is relevant for `pyactr`, but it has no theoretical significance (it's just a syntactic sugar). However, it is recommended, as doing so might clarify what kind of attribute-value matrices you will need in your model. Also if you don't specify the chunk type that your chunk uses, Python prints a warning message. This might help you debug your code (e.g., if you accidentally named your chunk "morphreme", you would get a warning message that a new chunk type has been created – probably, not what you wanted; warnings are not displayed in book). (See Python documentation for more on warnings.)

It is also recommended that you only use attributes you defined first (or you used in the first chunk of a particular type). However, you can always add new attributes along the way (it is assumed that other chunks up to now had no value for those attributes in that case). For example, imagine we realize that it's handy to specify what syntactic function a word is part of. We didn't have that in our example of car. So let's create a new chunk, `car2`, which is like `car` but it adds this extra piece of information (and we assume this word has been used as part of subject):

[**py5**]
```
>>> car2 = actr.makechunk(nameofchunk="car2",\
...                       typename="word",\
...                       phonology="/ka:/",\
...                       meaning="[[car]]",\
```

```
5 ...                           category="noun",\
6 ...                           number="sg",\
7 ...                           syncat="subject")
8 >>> print(car2)
9 word(category=noun, meaning=[[car]], number=sg, phonology=/ka:/, syncat=subject)
```

Line 7 in [**py5**] is the new part. We are adding a new slot `syncat`, and assign it the value `subject`. The command goes through successfully (as shown by the fact that we can print `car2`), but a warning message is issued (not displayed above), namely "`UserWarning`: Chunk type word is extended with new attributes."

There is another way of specifying a chunk, which is maybe more intuitive: using `chunkstring`. In that case, you write down the chunk type after the `isa`-attribute, and attribute value pairs are written after each other, separated only by a comma.

[**py6**]

```
1 >>> car2 = actr.makechunk(nameofchunk="car2",\
2 ...                           typename="word",\
3 ...                           phonology="/ka:/",\
4 ...                           meaning="[[car]]",\
5 ...                           category="noun",\
6 ...                           number="sg",\
7 ...                           syncat="subject")
8 >>> print(car2)
9 word(category=noun, meaning=[[car]], number=sg, phonology=/ka:/, syncat=subject)
```

We are using the new function `chunkstring`. It has the same power as `makechunk`. The argument `string` defines what the chunk consists of. The value pairs are written as a plain string. Notice that we use three quote marks, rather than one. These signal to Python that the string can appear on more than one line. The first slot-value pair ([**py6**], line 2) is special – it specifies the type of chunk, and a special slot is used for this, `isa`. Notice that the resulting chunk is identical to the previous one, as shown on [**py6**], line 8.

As we mentioned above, productions work by testing whether a particular condition is satisfied and then acting upon that. In practice, for most parts this means that productions check chunks. Thus, we have to define comparisons across chunks. This is done in an intuitive way: one chunk is identical to another if they have the same attributes and they have the same values for all the attributes. A chunk a is part of a chunk b if a has all the attributes of b and a has the same values as b in those attributes (however, chunk b might have extra attribute-value pairs).

`pyactr` overloads standard comparison operators for these tasks. The code below and its output should be self-explanatory:

[**py7**]

```
>>> car2 == car2
True
>>> car == car2
False
>>> car <= car2
True
>>> car < car2
True
>>> car2 < car
False
```

Note that chunk types are irrelevant for deciding part-of relations. This might be counter-intuitive, but that's just how ACT-R works – chunk types are 'syntactic sugar' useful only for the human modeler. This means that if we define a new chunk type that happens to have the same slots as another chunk type, one might be part of the other:

[**py8**]
```
>>> actr.chunktype("synlabel", "category")
>>> noun = actr.makechunk(nameofchunk="noun",
...                       typename="synlabel",
...                       category="noun")
>>> noun < car2
True
```

2.6 Modules and buffers

Chunks do not live in a vacuum, they are always part of an ACT-R architecture, which consists of modules and buffers. Each module in ACT-R serves a different task. Furthermore, modules cannot be accessed or updated directly in ACT-R; rather, this always happens through the use of a buffer, and each module comes equipped with one such buffer. A buffer, in its turn, is a carrier of exactly one chunk.

In this chapter, we will be concerned with only two modules, the goal module (representing one's goals) and the declarative module (representing one's declarative knowledge). These are the two most common modules in ACT-R. They appear with their buffers, which are called goal and retrieval, respectively.

For the sake of concreteness, let's create the declarative module and the goal and retrieval buffers. And since it does not make sense to think about modules without instantiating a model in which these modules work, let's start by doing just that:

[**py9**]
```
>>> agreement = actr.ACTRModel()
```

The command above instantiated an ACTRModel as the value of the variable `agreement`. We will now be filling in details of this model with information about buffers, models, and productions.

We start by creating relevant modules and buffers inside this model.

[**py10**]
```
>>> dm = agreement.DecMem()
>>> retrieval = agreement.dmBuffer(name="retrieval", declarative_memory=dm)
>>> g = agreement.goal(name="g")
```

- `DecMem` instantiates declarative memory. Notice that `DecMem` is an attribute of the model `agreement`. We just specified that this will be the declarative memory of our model and we bound it to the variable `dm`.

- `dmBuffer` instantiates the buffer of the declarative memory in the model. We fill in two arguments of this attribute. The second argument says to which declarative memory the buffer should be connected (i.e., from which memory it should be retrieving). The first argument says under what name the buffer will be seen in the model. The name of a buffer is needed if we are going to refer to these buffers later on in productions (without that productions would not be able to manipulate buffers). Notice also that the variable we bind this buffer to has the same name as the name used in the model (`retrieval`). This is just convenience.

- `goal` instantiates the goal buffer.

The declarative memory was just instantiated, so it should be empty. Let's check that:

[py11]
```
>>> dm
{}
```

We might want to add the best chunk we created so far – `car2`:

[py12]
```
>>> dm.add(car2)
>>> print(dm)
{word(category=noun, meaning=[[car]], number=sg, phonology=/ka:/, syncat=subject) {0.0}}
```

- Chunks are added by the attribute `add` on the declarative memory. As the argument, we specify a chunk (or chunks) that should be added.

`dm` now shows the chunk we added. It also ties the chunk to the time point at which it was introduced. Since we did not start any model simulation, the time point is 0 right now.

2.7 Writing productions in `pyactr`

In their core, productions are IF-statements.

Productions have two parts: left-hand side rules (tests) precede the double arrow (==>); right-hand side rules (actions) follow the arrow.

Let's now create productions that simulate a verb agreement.[1] We will simplify things a lot. We will only care about 3rd person agreement, present tense. We will do no syntactic parsing, just assume that our memory includes only the subject of the clause and we have the verb of the clause at our disposal. Since our goal is creating verb agreement, we should assume that the verb itself is all the time in the goal. What should agreement do? One production should state that IF goal has a verb and task is to agree THEN the subject should be retrieved. The second production should state that IF subject number in retrieval is `=x` THEN verb number in goal is `=x`. The third rule should say that if the verb is assigned a number the task is done.

Let's write down the second rule first.

[py13]
```
>>> agreement.productionstring(name="agree", string="""
...     =g>
...     isa verbagreement
...     task trigger_agreement
...     category 'verb'
...     =retrieval>
```

[1] The full code for this model is also available as `u1_agreement` on `http://www.jakubdotlacil.com/tutorials` and in the appendix to this chapter.

```
7 ...     isa word
8 ...     category 'noun'
9 ...     syncat 'subject'
10 ...     number =x
11 ...     ==>
12 ...     =g>
13 ...     isa verbagreement
14 ...     task done
15 ...     category 'verb'
16 ...     number =x
17 ...     """)
```

- Productions are created by the command `productionstring` and they have two arguments (later on, we will see that there is a third argument): `name` (the name of the production) and `string` (the string that specifies what the production does).

2.–11. The left hand side of the rule and the right hand side of the rule are separated by ==>. That is, what appears before ==> is tests, what appears after ==> are actions. Second, tests and actions have always the same structure: first, you specify what buffer should be considered: this is done by writing the name of the buffer between = and > (see line 2 and 6). The name of the buffer has to match the name you used when you created these buffers. After choosing the buffer you specify a chunk (lines 3–5 and lines 7–10). In case of tests the chunks specified in a rule must be part of a chunk that is present in the corresponding buffer (i.e., the part-of test, discussed in Sect. writing-chunks-in-pyactr, must be true between the chunk specified in the test and the chunk in the corresponding buffer). Chunks in productions are written in the same way as chunks in the function `chunkstring`: you write slot-value pairs, and each slot and value are separated by one or more spaces. (We also wrote each pair on a separate line, but that is just aesthetics.) The `isa` slot is used to specify chunk types.

12.–17. If all tests are true, then a chunk in a buffer is modified as specified after ==>.

All in all, we can read the rule agree as follows: IF the goal buffer has a chunk with category verb and the task is to trigger agreemnt AND the retrieval buffer has a chunk with the category noun and syncat subject and it has some number, assigned to x, THEN modify the chunk in the goal buffer so that it carries the number that was assigned to x.

The other rule should appear as follows:

[py14]
```
>>> agreement.productionstring(name="retrieve", string="""
...     =g>
...     isa verbagreement
...     task agree
...     category 'verb'
...     ?retrieval>
...     buffer empty
...     ==>
...     =g>
...     isa verbagreement
...     task trigger_agreement
...     category 'verb'
```

```
13 ...     +retrieval>
14 ...     isa word
15 ...     category 'noun'
16 ...     syncat 'subject'
17 ...     """)
```

6. Instead of =retrieval> in the test, we write ?retrieval>. While =retrieval> tests whether the retrieval carries a particular chunk ?retrieval> queries the buffer directly. The query in this case checks whether the buffer is empty (i.e., it carries no chunk). Strictly speaking, this is not necessary (the model would work just as well without this test). But we add it here for instruction purposes.

13. We specify +retrieval> in actions. While =retrieval> would modify a chunk present in the buffer, + states that a new chunk should be created/set. In case of the retrieval buffer chunks are 'created' by being retrieved from their module of declarative memory (in our case, dm).

We will look at some more examples of querying in the next section (i.e., cases in which we use ? instead of = in front of the name of a buffer). Before that, we add the third rule discussed above, which should check that the verb in goal carries a number, and if so, it should consider the task done.

[**py15**]
```
1  >>> agreement.productionstring(name="retrieve", string="""
2  ...     =g>
3  ...     isa verbagreement
4  ...     task agree
5  ...     category 'verb'
6  ...     ?retrieval>
7  ...     buffer empty
8  ...     ==>
9  ...     =g>
10 ...     isa verbagreement
11 ...     task trigger_agreement
12 ...     category 'verb'
13 ...     +retrieval>
14 ...     isa word
15 ...     category 'noun'
16 ...     syncat 'subject'
17 ...     """)
```

8. \textasciitilde{}g\textgreater{} is an action we did not see before. It discards the chunk present in the goal buffer.

2.8 More examples on queries

So far, we mentioned only one way of querying - checking that a buffer is full. Here are some more cases:

[**py16**]
```
>>> '?g> buffer full'
'?g> buffer full'
```

```
>>> '?retrieval> state busy'
'?retrieval> state busy'
>>> '?retrieval> state error'
'?retrieval> state error'
```

- This checks whether a buffer is full (whether it carries a chunk).
- This is true if the retrieval buffer is working on retrieving a chunk.
- This is true if the last retrieval failed (no chunk has been found).

2.9 Running a model

We have almost everything ready to run our first model, we are just missing one piece: having a chunk in the goal buffer in the start of our simulation (without that, there is no goal and without a goal, the model has no reason to change its internal state). So let's add the goal:

[**py17**]
```
>>> actr.chunktype("verbagreement", "task, category")
>>> g.add(actr.chunkstring(string="isa verbagreement task agree category 'verb'"))
>>> g
{verbagreement(category=verb, task=agree)}
```

- The chunk is added to the goal buffer in the same way as to other modules and buffers – by the attribute `add`.

We can now run the model.

[**py18**]
```
>>> simulation = agreement.simulation()
>>> simulation.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: retrieve')
(0.05, 'PROCEDURAL', 'RULE FIRED: retrieve')
(0.05, 'g', 'MODIFIED')
(0.05, 'retrieval', 'START RETRIEVAL')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
(0.1, 'retrieval', 'CLEARED')
(0.1, 'retrieval', 'RETRIEVED: word(category=noun, meaning=[[car]], number=sg, phonology=/ka:/,
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: agree')
(0.15, 'PROCEDURAL', 'RULE FIRED: agree')
(0.15, 'g', 'MODIFIED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
```

- First, we have to instantiate the simulation of the model.
- The simulation is run.

What you see in the output is the trace of a model. Each line specifies three elements: the first element is time (in seconds), the second element is the module that is affected, the third element is a description of what's happening to the module.

The first line states that conflict resolution takes place in the module procedural (i.e., the module responsible for controlling production rules). This happens at time 0. There is one rule that matches the current state of affairs, and that is `retrieve` (`retrieve` requires that the goal buffer has a chunk with the category `verb` and an empty and free retrieval buffer). It can fire (i.e., its left-hand side is satisfied by the state of the model at 0 ms, so we can proceed to the right-hand side of the production rule). In ACT-R, firing takes 50 ms, as we see above in the time specification of the third line. After that, goal is (vacuously) modified (the modification is vacuous given our rules above). Then the retrieval starts, and it takes 50 ms to finish the retrieval. When the retrieval happens (line 9), the retrieval buffer carries the right chunk. Followingly, a new rule can be selected, `agree` (`agree` requires that the retrieval carries a subject chunk, and consequently, it modifies the chunk in goal to match the number between a verb and a boun.

After that, the last rule fires (`done`), which cleares the goal buffer. When the goal buffer is cleared, its information does not disappear. It is assumed in ACT-R that that information is transferred to the declarative memory. This is also the case here (our past goals become our newly acquired memory facts).

We can now check the final state of the declarative memory to see that this is the case:

[**py19**]
```
>>> dm
{word(category=noun, meaning=[[car]], number=sg, phonology=/ka:/, syncat=subject) {0.0}}
```

2.10 Example 2 – a top-down parser

We will now turn to a more realistic case, a parser. There will be more parsers considered throughout the tutorials. Our starting point is one of the simplest parsers – a top-down parser.[2]

Suppose we have a context-free grammar with the following rules:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	ProperN
VP	$\rightarrow$	V NP

Furthermore, there are two nouns and one verb in our language: Mary, Bill, likes. We will analyze one sentence with our parser, Mary likes Bill.

A top-down parser can be understood as a push-down automaton. Push-down automata have a memory, represented as a stack. In the parser, the stack represents categories that have to be parsed. For example, the stack may consist of one symbol, S - this would express that a sentence needs to be parsed (obviously, this is the starting point of a parser). Or the stack could consist of two elements: NP, VP – expressing that the parser needs to parse an NP, followed by parsing a VP.

[2]The full code for this model is also available as u1_topdownparser on: http://www.jakubdotlacil.com/tutorials

The parser proceeds by modifying the contents of its stack based on two pieces of information: the top element on its stack (also written as the leftmost element below) and, possibly, a word that has to be parsed (the leftmost word in the stream of words).

We can sum up the parsing rules into just two general algorithm schemata (see, for example, Hale 2014):

- expand: if the stack shows a symbol X on top, and the grammar contains a rule X -> A B or X -> A, replace the symbol X with the symbol A, B or the symbol A, respectively.
- scan: if the stack shows a terminal and w, the word to be parsed, is of the right category, then remove the terminal from the stack and w from the parsed sentence.

We will now implement these general parsing rules to our grammar, which will be able to parse the sentence Mary likes Bill.

2.10.1 First steps in the model

Let us start with the first standard step, importing `pyactr`.

[**py20**]
```
>>> import pyactr as actr
```

Now, we should specify what chunktypes we need. We will have one chunktype for the parser. This will keep the information about stack contents, what word was parsed but also what the current task of the parser is (for most parts, it will be just that, parsing).

[**py21**]
```
>>> actr.chunktype("parsing", "task stack_top stack_bottom parsed_word ")
```

- The chunk type has four slots: what task we are doing, what the current top element in the stack is, what the bottom element is and what the parsed word is. Note that we have only two positions in our chunktype, stack top and stack bottom. This suffices for the simple case of binary structures we consider here, so we will leave it at this.

The second chunktype will represent the sentence. This might look weird: why should we represent a sentence in a chunk? In most of the cases, the sentence is external to an agent, it's what the agent reads or hears. However, at this point we have no way to represent the surrounding environment, so we have to represent a sentence internally, as a chunk. Later on, we will see a more elegant solution. The chunktype sentence will be assumed to carry at most three words.

[**py22**]
```
>>> actr.chunktype("sentence", "word1 word2 word3")
```

We will now initialize the model and assume it has a declarative memory, retrieval and a goal buffer.

[**py23**]
```
>>> parser = actr.ACTRModel()
>>> dm = parser.DecMem()
>>> retrieval = parser.dmBuffer(name="retrieval", declarative_memory=dm)
>>> g = parser.goal(name="g")
```

- We call our model `parser`.
- The declarative memory is declared in the standard way, using the attribute `DecMem`.
- The retrieval is declared in the standard way. We tie it to the just created declarative memory.
- `g` is our goal buffer.

The goal buffer will carry the information about parsing (that is, it will have the chunk `parsing`, whose type was already created). But we also need to carry the information about the parsed sentence (the chunk `sentence`). It would be nice to leave that information to the environment but we cannot do it yet, so let's create a second buffer, which is identical to goal and which carries the information about a sentence. In fact, that is not such a strange solution. ACT-R commonly assumes two goal buffers, one, which we used so far and which keeps information about one's goals, another one which keeps the internal image of current information. It might not be so far-fetched to use the imaginal buffer for the sentence itself. We will start this new buffer.

[**py24**]
```
>>> g2 = parser.goal(name="g2", set_delay=0.2)
```

- The imaginal buffer, `g2`, is created in almost the same way as the goal buffer. However, one extra argument is specified: `set\_delay`. This parameter specifies the delay required to set a chunk in the buffer. That is, it would take 0.2 s to set a chunk in g2. This is the standard value for the imaginal buffer (the goal buffer requires only 0.05 s to set a chunk).

We can now add chunks into g and g2.

[**py25**]
```
>>> g.add(actr.chunkstring(string="isa parsing  task parse stack_top 'S'"))
>>> g2.add(actr.chunkstring(string="isa sentence word1 'Mary' word2 'likes' word3 'Bill'"))
```

- We assume that the parser's goal is to parse a sentence.
- The sentence to be parses is *Mary likes Bill*.

The toughest part is coming now: how to code the parsing itself?

We will assume that grammar (and parsing rules stemming from grammar) is part of production knowledge. This is in contrast to lexical information, which is commonly treated as part of declarative memory (see Lewis and Vasishth 2005, for arguments for this distinction). So, our first task is to specify lexical knowledge. Let's do that (only syntactic categories will be specified):

[**py26**]
```
1 >>> actr.chunktype("word", "form, cat")
2 >>> dm.add(actr.chunkstring(string="isa word form 'Mary' cat 'ProperN'"))
3 >>> dm.add(actr.chunkstring(string="isa word form 'Bill' cat 'ProperN'"))
4 >>> dm.add(actr.chunkstring(string="isa word form 'likes' cat 'V'"))
```

- We start by creating a new type that will accommodate lexical information.

2.–4. We have three words. Their values should be obvious.

We now have to specify production rules that mimic context-free grammar rules and that encode top-down parsing, represented in the schemata expand and scan.

2.10.2 Production rules

Let's start with the first rule, expanding S into NP and VP. This should be relatively straightforward. We specify it as:

[**py27**]
```
1 >>> parser.productionstring(name="expand: S->NP VP", string="""
2 ...     =g>
3 ...     isa  parsing
4 ...     task parse
5 ...     stack_top 'S'
6 ...     ==>
7 ...     =g>
8 ...     isa parsing
9 ...     stack_top 'NP'
10 ...     stack_bottom 'VP'
11 ... """)
```

2. The rule tests against the goal buffer.

3.–5. It requires that the goal buffer carries a chunk whose task is to parse and whose element on top is S.

7. Its action is to modify the goal buffer.

8.–10. The rule will set the top element as NP and the bottom as VP. That is, this is the rule that expands S into NP and VP according to the abstract schema discussed above (see the general algorithm schema expand).

Notice that this oversimplifies things slightly. If we now have a symbol following S in the stack, it would be overwritten by VP - hardly a behavior we would want to have. This oversimplification is to a large extent caused by the fact that we only work with two-element stack. It will not affect our example or several other examples, so we will leave this simplification in place.

The second rule states that NP is expanded into ProperN:

[**py28**]
```
1 >>> parser.productionstring(name="expand: NP->ProperN", string="""
2 ...     =g>
3 ...     isa parsing
4 ...     task parse
5 ...     stack_top 'NP'
6 ...     ==>
7 ...     =g>
8 ...     isa parsing
9 ...     stack_top 'ProperN'
10 ... """)
```

9. The rule says that the symbol on the top of the stack should be rewritten from NP to N. Notice that unlike the previous rule, nothing is done to the bottom of the stack. Thus, it will be left unmodified.

The third rule in our grammar describes the expansion of VP into V and NP. So let's deal with it in the parallel way as the previous rules:

[py29]
```
1  >>> parser.productionstring(name="expand: VP -> V NP", string="""
2  ...     =g>
3  ...     isa parsing
4  ...     task parse
5  ...     stack_top 'VP'
6  ...     ==>
7  ...     =g>
8  ...     isa parsing
9  ...     stack_top 'V'
10 ...     stack_bottom 'NP'
11 ... """)
```

1.–10. Notice that the rule is almost identical to the first rule. We only changed the symbols, according to the context-free grammar rules.

Now, for the most complext part. Once we have terminals (ProperN, V), we have to check that the terminal matches the category of the word to be parsed. If so, the word is scanned.

We achieve this by splitting the task into two rules. If we have a terminal, say ProperN, the category of the word has to be retrieved from memory (rule `retrieve`). If the category matches the top of stack, the word is scanned.

[py30]
```
1  >>> parser.productionstring(name="retrieve: ProperN", string="""
2  ...     =g>
3  ...     isa parsing
4  ...     task parse
5  ...     stack_top 'ProperN'
6  ...     =g2>
7  ...     isa sentence
8  ...     word1 =w1
9  ...     ==>
10 ...     =g>
11 ...     isa parsing
12 ...     task retrieving
13 ...     +retrieval>
14 ...     isa word
15 ...     form =w1
16 ... """)
```

2.–5. We test that the top of the stack has a terminal, ProperN.

6.–8. The imaginal buffer has the leftmost word; the word is assigned to the variable w1.

10.-12. The goal is switched from parsing to retrieving.

13.–15. The retrieval starts. We are retrieving the chunk with the form of w1. This will retrieve a chunk with the lexical information about the particular word.

[py31]
```
1  >>> parser.productionstring(name="retrieve: V", string="""
2  ...     =g>
3  ...     isa parsing
4  ...     task parse
```

```
5 ...     stack_top 'V'
6 ...     =g2>
7 ...     isa sentence
8 ...     word1 =w1
9 ...     ==>
10 ...     =g>
11 ...     isa parsing
12 ...     task retrieving
13 ...     +retrieval>
14 ...     isa word
15 ...     form =w1
16 ... """)
```

5. We test that the top of the stack has a terminal, V. APart from this one line, the rule is identical to the previous one.

Now, we define the rule that deals with the retrieved information and scans the upcoming word:

[**py32**]
```
1 >>> parser.productionstring(name="scan: string", string="""
2 ...     =g>
3 ...     isa parsing
4 ...     task retrieving
5 ...     stack_top =y
6 ...     stack_bottom =x
7 ...     =retrieval>
8 ...     isa word
9 ...     form =w1
10 ...     cat =y
11 ...     =g2>
12 ...     isa sentence
13 ...     word1 =w1
14 ...     word2 =w2
15 ...     word3 =w3
16 ...     ==>
17 ...     =g>
18 ...     isa parsing
19 ...     task print
20 ...     stack_top =x
21 ...     stack_bottom empty
22 ...     parsed_word =w1
23 ...     =g2>
24 ...     isa sentence
25 ...     word1 =w2
26 ...     word2 =w3
27 ...     word3 empty
28 ... """)
```

2.–6. This checks that the goal buffer has the task `retrieving`. Furthermore, it assigns stack symbols to two variables.

7.–10. The syntactic category of the retrieval must match the symbol on top of the stack.

11.–15. The imaginal buffer carries the sentence. Three words are assigned to three variables.

17.–22. This action achieves that the symbol on the bottom of the stack is moved to the top position. Notice also that the goal buffer has been changed into a new stage, print. This is not necessary, it serves only the purpose of checking that everything went fine. We want to print the word that has been currently parsed. We will do that in a separate production. For the same reason, we keep the information about the currently parsed word in the goal buffer, in the slot parsed_word.

23.–27. Words are moved one level up (the word on the second position is moved to the first position etc.). The last position is left empty.

The printing production that follows scanning the string, is specified below:

[**py33**]
```
1  >>> parser.productionstring(name="print parsed word", string="""
2  ...     =g>
3  ...     isa parsing
4  ...     task print
5  ...     =g2>
6  ...     isa sentence
7  ...     word1 ~empty
8  ...     ==>
9  ...     !g>
10 ...     show parsed_word
11 ...     =g>
12 ...     isa parsing
13 ...     task parse
14 ...     parsed_word None""")
```

2.–4. This tests that the goal buffer has the task `print`.

5.–7. The value of the slot `word1` in the imaginal buffer is not empty (the squiggle is negation).

9.–10. -

11.–12. This part will print the parsed word. !g> says that Python should carry out an action in the goal buffer. After !g>, we have to specify what Python should do: we specify that we want Python to show something (i.e., it should execute the method `show`) and what should be shown, that is, the value of the slot `parsed\_word`.

13.–16. The last action deletes whatever was in `parsed\_word`.

The last production we have to consider is the production at the end of parsing. The parsing ends when `word1` has the value empty and the task is `print` (i.e., no parsing or retrieving is going on in the goal buffer). As a way of summary, we will also print all our rules.

[**py34**]
```
>>> productions = parser.productionstring(name="done", string="""
...     =g>
...     isa parsing
...     task print
...     =g2>
```

```
6 ...     isa sentence
7 ...     word1 empty
8 ...     ==>
9 ...     =g>
10 ...     isa parsing
11 ...     task done
12 ...     !g>
13 ...     show parsed_word
14 ...     ~g2>
15 ...     ~g>""")
16 >>> print(productions)
17 None
```

1. We bind the output to the variable `productions`. The output is all the production rules in the model. We can print them afterwards.

6.–8. We check that there is no leftmost word (the whole sentence was parsed).

14.–15. The imaginal and goal buffers are cleared.

16. We print all production rules.

2.10.3 Running the model

We run the model in the same way as before.

[py35]
```
>>> sim = parser.simulation()
>>> sim.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S->NP VP')
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S->NP VP')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP->ProperN')
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP->ProperN')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'NO RULE FOUND')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'retrieval', 'RETRIEVED: word(cat=ProperN, form=Mary)')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: scan: string')
(0.25, 'PROCEDURAL', 'RULE FIRED: scan: string')
(0.25, 'g2', 'MODIFIED')
(0.25, 'g', 'MODIFIED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: print parsed word')
```

```
(0.3, 'PROCEDURAL', 'RULE FIRED: print parsed word')
Mary
(0.3, 'g', 'EXECUTED')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: expand: VP -> V NP')
(0.35, 'PROCEDURAL', 'RULE FIRED: expand: VP -> V NP')
(0.35, 'g', 'MODIFIED')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')
(0.4, 'PROCEDURAL', 'RULE FIRED: retrieve: V')
(0.4, 'g', 'MODIFIED')
(0.4, 'retrieval', 'START RETRIEVAL')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.45, 'retrieval', 'CLEARED')
(0.45, 'retrieval', 'RETRIEVED: word(cat=V, form=likes)')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'RULE SELECTED: scan: string')
(0.5, 'PROCEDURAL', 'RULE FIRED: scan: string')
(0.5, 'g2', 'MODIFIED')
(0.5, 'g', 'MODIFIED')
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'RULE SELECTED: print parsed word')
(0.55, 'PROCEDURAL', 'RULE FIRED: print parsed word')
likes
(0.55, 'g', 'EXECUTED')
(0.55, 'g', 'MODIFIED')
(0.55, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.55, 'PROCEDURAL', 'RULE SELECTED: expand: NP->ProperN')
(0.6, 'PROCEDURAL', 'RULE FIRED: expand: NP->ProperN')
(0.6, 'g', 'MODIFIED')
(0.6, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.6, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.65, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
(0.65, 'g', 'MODIFIED')
(0.65, 'retrieval', 'START RETRIEVAL')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'NO RULE FOUND')
(0.7, 'retrieval', 'CLEARED')
(0.7, 'retrieval', 'RETRIEVED: word(cat=ProperN, form=Bill)')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'RULE SELECTED: scan: string')
(0.75, 'PROCEDURAL', 'RULE FIRED: scan: string')
(0.75, 'g2', 'MODIFIED')
(0.75, 'g', 'MODIFIED')
(0.75, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.75, 'PROCEDURAL', 'RULE SELECTED: done')
(0.8, 'PROCEDURAL', 'RULE FIRED: done')
Bill
(0.8, 'g', 'EXECUTED')
(0.8, 'g', 'MODIFIED')
```

```
(0.8, 'g2', 'CLEARED')
(0.8, 'g', 'CLEARED')
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8, 'PROCEDURAL', 'NO RULE FOUND')
```

- We instantiate the simulation of the model.
- The simulation is run.

This all looks good. We parsed the three words and we ended up in the stage done. We can also check our declarative memory. Since we cleared g and g2 at the end of done, it should consist of those elements (it should also carry the chunks we put in there before, the lexical knowledge). The chunks from `g` and `g2` should have empty positions in `stack\_top` and `stack\_bottom`, as well as `word1` – `word3`. Let's see.

[py36]
```
>>> dm
{word(cat=ProperN, form=Bill): {0.0}, word(cat=V, form=likes): {0.0, 0.7}, word(cat=ProperN, for
```

This is all good.

As a further check, let's see whether our simple parser correctly fails if we feed it an ungrammatical sentence, say *Bill Mary likes*. It should fail during parsing of the second word, *Mary*, because the noun would not match its expectations.

We add relevant chunks into the goal and the imaginal buffers and start the new simulation.

[py37]
```
>>> g.add(actr.chunkstring(string="isa parsing  task parse stack_top 'S'"))
>>> g2.add(actr.chunkstring(string="isa sentence word1 'Bill' word2 'Mary' word3 'likes'"))
>>> sim = parser.simulation()
>>> sim.run()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: expand: S->NP VP')
(0.05, 'PROCEDURAL', 'RULE FIRED: expand: S->NP VP')
(0.05, 'g', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP->ProperN')
(0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP->ProperN')
(0.1, 'g', 'MODIFIED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
(0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
(0.15, 'g', 'MODIFIED')
(0.15, 'retrieval', 'START RETRIEVAL')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'RULE SELECTED: scan: string')
(0.2, 'retrieval', 'CLEARED')
(0.2, 'PROCEDURAL', 'RULE FIRED: scan: string')
(0.2, 'retrieval', 'RETRIEVED: word(cat=ProperN, form=Bill)')
(0.2, 'g2', 'MODIFIED')
(0.2, 'g', 'MODIFIED')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
```

```
(0.2, 'PROCEDURAL', 'RULE SELECTED: print parsed word')
(0.25, 'PROCEDURAL', 'RULE FIRED: print parsed word')
Bill
(0.25, 'g', 'EXECUTED')
(0.25, 'g', 'MODIFIED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: expand: VP -> V NP')
(0.3, 'PROCEDURAL', 'RULE FIRED: expand: VP -> V NP')
(0.3, 'g', 'MODIFIED')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: retrieve: V')
(0.35, 'PROCEDURAL', 'RULE FIRED: retrieve: V')
(0.35, 'g', 'MODIFIED')
(0.35, 'retrieval', 'START RETRIEVAL')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'NO RULE FOUND')
(0.4, 'retrieval', 'CLEARED')
(0.4, 'retrieval', 'RETRIEVED: word(cat=ProperN, form=Mary)')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
```

- The goal should be to parse a sentence, as before.

- The imaginal buffer should carry the information about the sentence, *Bill Mary likes*.

This is good. The parser correctly parsed the first word, but it failed at the second word. After it was retrieved, the parser could not match its category to the top of the stack (which required V).

But it is not enough that the parser correctly parses grammatical sentences and fails in ungrammatical ones. ACT-R is not a theory of computationally effective parsers, it is a theory of human cognition. ACT-R parsers should then model human processing as realistically as possible. Is that so in this case? One thing we would expect from such a parser is that its time requirements should correspond to human processing. We see that it takes 800 ms to parse the sentence *Mary likes Bill*. This might be roughly correct, but there are things to worry about. For example, the parser requires this much time while abstracting away from what people have to do during parsing (internalizing visual information, projecting sentence meaning, a.o.), so ultimately, 800 ms might be too much given the amount of work this parser does. Another worry is that retrieving lexical information always takes 50 ms (see above). But this is hardly realistic. We know that lexical retrieval is dependent on various factors, and frequency is probably the most relevant one. This is completely ignored here. Finally, top-down parsers works quite well for a right-branching structures like the sentence *Mary likes Bill*, but it would have problems with left branching. In left branching the parser would have to store as many symbols on the stack as there are levels of embedding. Since every expansion of a rule takes 50 ms, we would expect that left branching structures of n-level embeddings should take $50 * n$ ms. This is at odds with human perforamce (cf. Resnik 1992). Thus, there is a lot of room for improvement to get to a more plausible human parser.

2.10.4 Stepping through a model

So far, when we checked a model, we always did that in one step, by running it from start to the end. This is fine, but there are cases when we might want to proceed more carefully. For example, we might want to check each step to see at which point the goal buffer gets its `parsed\_word`. Or our model is running an infinite loop, and we only want to check what's going on in the first few rules. Or we want to check what our declarative memory looks like after the retrieval is cleared for the first time. Etc.

For all these cases, it is handy to step through the simulation, rather than running it as a whole. Let's start our model again and do that.

[**py38**]
```
>>> g = parser.goal(name="g")
>>> g2 = parser.goal(name="g2", set_delay=0.2)
>>> g.add(actr.chunkstring(string="isa parsing  task parse stack_top 'S'"))
>>> g2.add(actr.chunkstring(string="isa sentence word1 'Bill' word2 'likes' word3 'Mary'"))
>>> sim = parser.simulation()
>>> sim.step()
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
```

Ok, what's that? Nothing happened so far. The simulation only proceeded through the first step (setting up the model), and there is no output. Let's add some more steps:

[**py39**]
```
1  >>> for _ in range(10):
2  ...     sim.step()
3  ...
4  (0, 'PROCEDURAL', 'RULE SELECTED: expand: S->NP VP')
5  (0.05, 'PROCEDURAL', 'RULE FIRED: expand: S->NP VP')
6  (0.05, 'g', 'MODIFIED')
7  (0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
8  (0.05, 'PROCEDURAL', 'RULE SELECTED: expand: NP->ProperN')
9  (0.1, 'PROCEDURAL', 'RULE FIRED: expand: NP->ProperN')
10 (0.1, 'g', 'MODIFIED')
11 (0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
12 (0.1, 'PROCEDURAL', 'RULE SELECTED: retrieve: ProperN')
13 (0.15, 'PROCEDURAL', 'RULE FIRED: retrieve: ProperN')
```

1 We add more steps using the for-loop. This line says that the loop will run 10 times.

2 Every time, the simulation steps forward by one step.

Let's now move to the point at which the rule 'scan: string' has just fired.

In order to be able to do that, we have to be able to see into the current event. The current event is an attribute of the simulation. This is how we can check it:

[**py40**]
```
>>> sim.current_event
Event(time=0.15, proc='PROCEDURAL', action='RULE FIRED: retrieve: ProperN')
```

The event has three arguments: time, proc and action. Time is the time at which the event took place. proc is the name of the module that's affected. action represents the action that's taking place. So, let's move to the action of firing of 'scan: string'.

[**py41**]
```
1  >>> while sim.current_event.action != 'RULE FIRED: scan: string':
2  ...     sim.step()
3  ...
4  (0.15, 'g', 'MODIFIED')
5  (0.15, 'retrieval', 'START RETRIEVAL')
6  (0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
7  (0.15, 'PROCEDURAL', 'NO RULE FOUND')
8  (0.2, 'retrieval', 'CLEARED')
9  (0.2, 'retrieval', 'RETRIEVED: word(cat=ProperN, form=Bill)')
10 (0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
11 (0.2, 'PROCEDURAL', 'RULE SELECTED: scan: string')
12 (0.25, 'PROCEDURAL', 'RULE FIRED: scan: string')
```

1 We specify a loop that will run until the action is 'scan: string'.

2 The simulation proceeds forward while the loop is True.

Now, we can check, for example, what our buffers look like:

[**py42**]
```
>>> g
{parsing(parsed_word=None, stack_bottom=VP, stack_top=ProperN, task=retrieving)}
>>> g2
{sentence(word1=likes, word2=Mary, word3=empty)}
```

2.10.5 Exercises

As an exercise, consider expanding the top-down parser. Additionally to what we have now, we should also be able to process the following rules from our grammar:

VP $\rightarrow$ V CP
VP $\rightarrow$ V
CP $\rightarrow$ C S

Furthermore, we will add following lexical items into our memory: that, cat C; believes, cat V; sleeps, cat V; John, cat ProperN.

With these additions, you should be able to parse sentences like 'Mary believes that Bill sleeps' (but see below).

You can probably see right away that the created parser might run into problems. For example, the parser might get stuck if you feed it the sentence 'Mary believes that Bill likes Mary' and it decides to expand the first VP into V and NP or into just V. This is a typical property of top-down parsers: they hypothesize about categories/structures before seeing them. In our model, the parser will have several ways to expand VP, so it should run into troubles when it uses the rule that happens to be incompatible with input.

So, what happens in those cases? What will our ACT-R top down parser do? What do you think?

The problem with top-down parsing can be avoided if we switch our strategy: rather than postulating the structure before having evidence, we might want to defer creating the structure until all the relevant evidence is available. This different strategy has a name - it is a bottom-up parser. We will consider how it can be built in ACT-R in the next chapter, as well as some other models relevant for language.

2.11 The environment in ACT-R

2.11.1 Introduction

We introduced a few modules of ACT-R in the previous sections:

- the declarative memory module and its retrieval buffer
- the goal buffer
- the imaginal buffer
- procedural knowledge

These are core modules of ACT-R. But using just those modules yields a model that is completely internal. It does not interact in any way with the environment. We are going to change that in this section.

2.11.2 A simple lexical decision task

We will consider a very simple lexical decision task, one in which the ACT-R model searches a (virtual) screen, finds a word, and if the word matches its (extremely impoverished) lexicon, it will press the J key, otherwise it will press the F key.

We start by import `pyactr`.

[**py1**]
```
>>> import pyactr as actr
```

Creating an environment

ACT-R models can interact with an environment, which, currently, is just a (simulated) computer screen, and a very primitive one at that (currently, only plain text is supported). We start it as follows:

[**py43**]
```
>>> environment = actr.Environment(focus_position=(0,0))
```

The class `Environment` has various arguments when initialized. Here we only specified `focus\_position` (this indicates at which position the eye focus is when the simulation starts). Two other most relevant arguments are `simulated\_screen\_size` and `viewing\_distance`. The first argument specifies the size of the screen we are trying to simulate in our model. If you are running a simulation of an experiment, you would encode the physical size of the screen (in cm) of the monitor you used. The second important argument specifies the distance from which the monitor is seen. Some reasonable defaults are assumed here: the screen is of size 50x28 cm, the distance is 50cm.

After the environment is initialized, we can initialize our ACT-R model.

[**py44**]
```
>>> model = actr.ACTRModel(environment=environment, automatic_visual_search=False)
```

The initialization is similar as previously, the only difference is that this time, we specify arguments when creating the model. First of all, we state what environment the model is interacting with. This is the environment we just created. Second, we specify a parameter in the model. The parameter is `automatic\_visual\_search` and we set it to `False`. This will ensure that ACT-R does not search the environment unless we specifically tell it to do so (more on this later).

There are many other parameters in the model, and a big part of these tutorials is to discuss their role in cognitive modelling. You can glance at all possible parameters available by checking the attribute `MODEL\_PARAMETERS`:

[py45]
```
>>> model.MODEL_PARAMETERS
{’retrieval_threshold’: 0, ’latency_factor’: 0.1, ’eye_mvt_angle_parameter’: 1, ’utility_alpha’:
```

Adding standard modules

After this, we add modules that should be familiar from previous parts. Since we are simulating a primitive lexical decision task, we will need some words in the declarative memory that ACT-R can access and check against the stimuli in the simulated experiment. So, we add a few words into the declarative memory.

[py46]
```
1 >>> actr.chunktype("goal", "state")
2 >>> actr.chunktype("word", "form")
3 >>> dm = model.DecMem()
4 >>> for i in {"elephant", "dog", "crocodile"}:
5 ...     dm.add(actr.makechunk(typename="word", form=i))
6 ...
7 >>> retrieval = model.dmBuffer("retrieval", dm)
8 >>> g = model.goal("g")
```

4 – 6 Notice that we add three words into our declarative memory using a `for` loop, rather than tediously writing the code for each word. This way of adding chunks can save a lot of time if you want to add a lot of elements (e.g., the whole lexicon).

Finally, we add a chunk into the goal buffer that will be present there when our simulation starts.

[py47]
```
>>> g.add(actr.makechunk(nameofchunk=’start’, typename="goal", state=’start’))
```

Visual module

The visual module will allow the ACT-R model to ‘see’ the environment. This is achieved through the interaction of two buffers: `visual\_location` searches the environment for elements matching its search criterie; `visual` stores the element that was found using `visual\_location`. The two buffers are sometimes called the visual Where and What buffers.

The visual Where buffer checks the environment (the screen) and outputs the location of an element on the screen that matches search criteria. Three slots are possible for a search: `color`, `screen\_x` (the horizontal position on the screen) and `screen\_y` (the vertical position on the screen). The x and y positions can be specified in precise terms (e.g., find an element in the location `screen\_x 100 screen\_y 100` where numbers represent pixels) or only roughly. Stating `screen\_x <100` would search the part of the screen that has 100 or fewer pixels on the x-axis and `screen\_x >100` would search the other part of the screen. Three other keywords are supported in search. `screen\_x lowest` expresses that the element with the lowest position on the horizontal axis should be found. `screen\_x highest` searches for the element with the highest position on the same axis. `screen\_x closest` searches for the closest element (the axis is ignored in this case). The same keywords and search terms can be used for the y axis.

The visual What buffer stuffs the element whose location was found in the Where buffer. Thus, usually, this buffer follows the workings of the previous buffer, as we'll see in production rules.

The vision module as a whole is an implementation of an EMMA (Eye Movements and Movement of Attention) computational model (Salvucci, 2001), which, in turn, is a generalization (and simplification) of E-Z Reader model (Reichle et al., 1998). While the latter model is used for reading, the EMMA model attempts to simulate any visual task, not just reading. Empirical support and modelling claims for E-Z Reader and EMMA can be found in (Reichle et al., 1998; Salvucci, 2001; Staub, 2011). Since these issues go beyond the scope of the book, we will not discuss them here.

2.11.3 Motor module

The motor module models pressing a key on a keyboard (or typing, if more key strokes are chained). The ACT-R typing model is based on EPIC's Manual Motor Processor (Meyer and Kieras, 1997). It has one buffer that accepts requests to execute motor commands. The ACT-R model implemented in `pyactr` is more limited, it currently supports only one command can be: that of `press\_key`). But this should suffice for simulations of many experimental designs, since it is common to allow only for keyboard interaction in experiments.

The hands of the model are assumed to be positioned on the standard keyboard at the home row position (index fingers are at F and J). The model assumes a competent but not an expert typist.

Adding productions

In the production, we want to model a lexical decision task. Five rules will suffice for this.

The first rule will ensure that the visual Where buffer looks for a word in the (virtual) screen.

[**py48**]
```
>>> model.productionstring(name="find_word", string="""
...     =g>
...     isa     goal
...     state   'start'
...     ?visual_location>
...     buffer  empty
...     ==>
...     =g>
...     isa     goal
...     state   'attend'
...     +visual_location>
```

```
12 ...     isa _visuallocation
13 ...     screen_x closest""")
```

The rule requires that the `start` chunk is in the goal buffer (lines 2 – 4) and there is no chunk in the visual location buffer (lines 5 – 6). If this is satisfied, the goal will be updated (lines 8 – 10) and the visual location will be updated (lines 11 – 13). The visual location is updated with the position of the closest element (line 13).

After this rule fired, the ACT-R model would know a position of an element but it would not know what that element is. For that, the visual What buffer has to be employed. This is done in the second rule, `attend\_probe`.

[**py49**]
```
1  >>> model.productionstring(name="attend_probe", string="""
2  ...     =g>
3  ...     isa     goal
4  ...     state   'attend'
5  ...     =visual_location>
6  ...     isa    _visuallocation
7  ...     ?visual>
8  ...     state   free
9  ...     ==>
10 ...     =g>
11 ...     isa     goal
12 ...     state   'retrieving'
13 ...     +visual>
14 ...     isa     _visual
15 ...     cmd     move_attention
16 ...     screen_pos =visual_location
17 ...     ~visual_location>""")
```

This rule checks that there is an element in the visual Where buffer (lines 5 and 6). If so and if the visual What buffer is free (i.e., it does not carry out any action, a new chunk is added to it (lines 13 – 16). The last line states that the visual location is cleared.

Notice how the visual buffer finds an element. This is achieved by specifying the command in the chunk as `move\_attention`. This requires that the attention is moved to a new position. The new position is specified in the slot `screen\_pos`, and it is the position that the visual location has uncovered.

The interaction between the two buffers in vision simulates a two-step process: (i) noticing an object (through the visual location buffer), (ii) finding what that object is, i.e., attending to the object (through the visual buffer).

The next rule start the retrieval process. The retrieval process should be simple: match the value in the chunk carried in the visual module to the retrieval module that will search the declarative memory for the same element. In the rule that follows, the crucial bits are line 7 and line 14, which ensure that the value of the seen chunk forms the basis for the retrieval of a word.

[**py50**]
```
1  >>> model.productionstring(name="recalling", string="""
2  ...     =g>
3  ...     isa     goal
4  ...     state   'retrieving'
```

```
...     =visual>
...     isa     _visual
...     value   =val
...     ==>
...     =g>
...     isa     goal
...     state   'retrieval_done'
...     +retrieval>
...     isa     word
...     form    =val""")
```

The last two rules are specified below. They consider two possibilities: (i) a chunk was retrieved (the first rule), (ii) no chunk was found (the second rule). The rules should look familiar, the only new bits are lines 12 – 15 and 45 – 48. These lines set the motor module in action. The motor module is there only to carry out a task, which is done using the special chunk `\_manual`. The chunk has two slots: `cmd` (what command should be carried out) and `key` (what key should be pressed).

[**py51**]
```
1  >>> model.productionstring(name="can_recall", string="""
2  ...     =g>
3  ...     isa     goal
4  ...     state   'retrieval_done'
5  ...     ?retrieval>
6  ...     buffer  full
7  ...     state   free
8  ...     ==>
9  ...     =g>
10 ...     isa     goal
11 ...     state   'done'
12 ...     +manual>
13 ...     isa     _manual
14 ...     cmd     press_key
15 ...     key     'J'""")
16
17 >>> model.productionstring(name="cannot_recall", string="""
18 ...     =g>
19 ...     isa     goal
20 ...     state   'retrieval_done'
21 ...     ?retrieval>
22 ...     buffer  empty
23 ...     state   error
24 ...     ==>
25 ...     =g>
26 ...     isa     goal
27 ...     state   'done'
28 ...     +manual>
29 ...     isa     _manual
30 ...     cmd     press_key
31 ...     key     'F'""")
```

Before we run the simulation of the model, we have to specify one last bit: what should appear on the screen. We use a dictionary data structure for that. The dictionary data structure is represented using {} brackets and it consists of key-value pairs. Values can themselves be dictionaries.

[**py52**]
```
>>> word = {1: {'text': 'elephant', 'position': (320, 180)}}
```

We specify here that our first (and only) stimulus is printing the word *elephant* in the position 320x180 pixels.

Initializing the simulation is done below.

[**py53**]
```
>>> sim = model.simulation(realtime=True, gui=False, environment_process=environment.environment
```

Unlike in previous models, we now call the simulation with various arguments. The first argument (`realtime`) states that the simulation should proceed in real time. `gui` specifies whether a graphical user interface (a separate window) should be started to represent the environment (this option is switched off here, but by all means, switch it on on your computer by setting the argument to `True`). The third argument states what environment process should appear in our environment. You could in principle create your own but there is one predefined in the class `Environment`. This will print stimuli and after a specific time elapses or the right trigger is pressed, it will remove the stimulus and print a new one (or end). The following three arguments (`stimuli`, `triggers` and `times`) specify values in the environment process. The first one states what stimuli should be printed. In our case, this will be the word *elephant*, as specified in the variable `word`. The second one states what triggers the process should respond to (we assume that it should not respond to anything. The last argument states how long the stimulus should be printed (1 s).

We run the simulation using the method `run`.

[**py54**]
```
>>> sim.run(2)
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: find_word')
****Environment: {1: {'position': (320, 180), 'text': 'elephant'}}
(0.05, 'PROCEDURAL', 'RULE FIRED: find_word')
(0.05, 'g', 'MODIFIED')
(0.05, 'visual_location', 'CLEARED')
(0.05, 'visual_location', "ENCODED LOCATION:'_visuallocation(color=None, screen_x=320, screen_y=
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: attend_probe')
(0.1, 'PROCEDURAL', 'RULE FIRED: attend_probe')
(0.1, 'g', 'MODIFIED')
(0.1, 'visual_location', 'CLEARED')
(0.1, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION STARTED')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'NO RULE FOUND')
(0.123, 'visual', 'CLEARED')
(0.123, 'visual', "ENCODED VIS OBJECT:'_visual(cmd=move_attention, color=None, screen_pos=_visua
(0.123, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.123, 'PROCEDURAL', 'RULE SELECTED: recalling')
(0.173, 'PROCEDURAL', 'RULE FIRED: recalling')
(0.173, 'g', 'MODIFIED')
(0.173, 'retrieval', 'START RETRIEVAL')
```

```
(0.173, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.173, 'PROCEDURAL', 'NO RULE FOUND')
(0.2212, 'visual', 'PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED')
(0.2212, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2212, 'PROCEDURAL', 'NO RULE FOUND')
(0.223, 'retrieval', 'CLEARED')
(0.223, 'retrieval', 'RETRIEVED: word(form=elephant)')
(0.223, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.223, 'PROCEDURAL', 'RULE SELECTED: can_recall')
(0.273, 'PROCEDURAL', 'RULE FIRED: can_recall')
(0.273, 'g', 'MODIFIED')
(0.273, 'manual', 'COMMAND: press_key')
(0.273, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.273, 'PROCEDURAL', 'NO RULE FOUND')
(0.3321, 'visual', 'SHIFT COMPLETE TO POSITION: [320, 182]')
(0.3321, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3321, 'PROCEDURAL', 'NO RULE FOUND')
(0.423, 'manual', 'PREPARATION COMPLETE')
(0.423, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.423, 'PROCEDURAL', 'NO RULE FOUND')
(0.473, 'manual', 'INITIATION COMPLETE')
(0.473, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.473, 'PROCEDURAL', 'NO RULE FOUND')
(0.483, 'manual', 'KEY PRESSED: J')
(0.483, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.483, 'PROCEDURAL', 'NO RULE FOUND')
(0.573, 'manual', 'MOVEMENT FINISHED')
(0.573, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.573, 'PROCEDURAL', 'NO RULE FOUND')
(1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(1, 'PROCEDURAL', 'NO RULE FOUND')
```

Let us first consider the general picture this trace paints. In this model, we see that it should take roughly 450 ms to find a stimulus, decide whether it is a word and to press the right key (check the event 'KEY PRESSED'). This is slightly faster than 500-600 ms usually found in lexical decision tasks (Forster, 1990; Murray and Forster, 2004). But notice that while we model eye movement and finger movement in quite some detail, we completely abstract away from memory retrieval. The retrieval always takes 50 ms regardless of any parameter of the word. This is definitely not correct. We will improve the state of affairs in the next chapter.

Before going there, we notice that there are a few new things in the trace of the model. They represent the visual model and the motor model. The events of the first model are signalled by the name 'visual' and they simulate attention to a visual object. The events of the second model appear under the name of 'motor'. What do they mean? We will explain that in the next two sections.

2.11.4 Vision in ACT-R

Traditionally, it has been assumed that attention corresponds to the focus position of eyes (see, e.g., Just and Carpenter 1980; Just et al. 1982), so to understand what one attends it suffices to look at one's eye positions. But this is too simplistic. In reading, it is known that some words (especially high-frequent ones) are processed without ever receiving eye focus (Schilling et al., 1998; Rayner, 1998, a.o.). The EMMA model captures this by disassociating eye focus and attention: the two processes are related but not identical.

A shift of attention to a visual object (the command `move\_attention`) triggers an immediate attempt to encode the object as an internal representation. At the same time, it also triggers eye movement. However, the two processes proceed independently of each other.

The time needed to encode an object, t_{enc} is modeled using a gamma distribution as follows:

(4) $t_{enc} \approx Gamma(\text{shape} = T_{enc}, \text{scale} = T_{enc}/9)$

That is, it is the gamma distribution with mean T_{enc} and the standard deviation $\frac{T_{enc}}{3}$. The parameter T_{enc} is found using the following formula:

(5) $T_{enc} = Ke^{kd}$

Where:

- d is a distance between the current focal point of the eyes and the object to be encoded, measured in degrees of visual angle
- k is a free parameter, scaling the effect of distance (it is set at 1 by default)
- K is a free parameter, scaling the encoding time itself (set at 0.01 by default)

The time needed to shift eyes to the new object is split into two sub-processes: preparation and execution. The preparation is modeled as a gamma distribution with mean 135 ms and standard deviation 45 ms. The execution, which follows the preparation, is modeled as a gamma distribution with mean 70 ms + 2 ms for every degree of visual angle between the current eye position and the targeted visual object, and standard deviation one third of the mean. It is only at the end of the execution that eyes focus the new position. Thus, the whole process of eye movement takes around 200 ms, which corresponds to average saccade latencies reported in previous studies (see, e.g., Fuchs 1971).

In the trace of the model, [**py54**], the time point of encoding a visual object is signalled by the event 'ENCODED VIS OBJECT'. The end of the preparation phase is signalled by 'PREPARATION TO SHIFT VISUAL ATTENTION COMPLETED'. The end of the execution shift is signalled by 'SHIFT COMPLETE TO POSITION'. It is only at the last event that eyes end up at the new location, but the internal representation of the object has been encoded for a long time at this point as you can check, so cognitive processes had time to proceed while eyes were moving to a new position.

How do visual encoding and eye movements interact? Three options could take place. First, encoding could be done before the end of the preparation phase. (This is the case here.) If the following cognitive processes are rapid enough to cancel eye movement or reques a new position before the end of the preparation phase, eye shift to is interrupted. (This is not the case here, hence eye shift is carried out.) Second, encoding could be finished during the execution phase. At that point, eye movement cannot be stopped any more. Finally, it could happen that visual encoding is still not done after eyes shift to a new position. In that case encoding is re-started. Given the original time needed to encode, t_{enc}, and the time completed in the original encoding, t_c, and the new encoding time, t'_{enc}, the new time to encode is calculated as:

$$(6) \quad t = (1 - (t_c/t_{enc}) * t'_{enc}$$

Since the eye position is now closer to the object, the new process should proceed faster and it is furthermore decreased by the amount of encoding that was already achieved.

2.11.5 Manual processes in ACT-R

Similarly to the vision module, the motor module is split in several sub-phases when carrying out a command: the preparation phase, the initiation phase, the actual key press and finishing the movement (returning to the original position). As in the case of the visual module, cognitive processes can interrupt a movement, but only during the preparation phase. The time needed to carry out every phase is dependent on several variables:

[**py1**] Is this the first movement or not? If something was pressed before, was it pressed with the same hand or not? Answers to these questions influence the amount of time the preparation phase takes.

[**py2**] Is the key to be pressed on the home row or not? The answer to this question influences the amount of time the actual movement requires, as well as the preparation phase.

TODO - bottom up parser; the code for that has to be cleaned up because of changes to pyactr since the last time

2.11.6 Exercises

Exercise 1

In our model, visual object encoding was faster than the preparation of the eye shift. Try to get the encoding follow the preparation phase and the execution phase. You could do that in two ways: (i) by changing the position of the object and/or the original focus position; (ii) by changing the parameters related to visual encoding (eye_mvt_angle_parameter and/or eye_mvt_scaling_parameter); these parameters are specified when initializing an ACT-R model, e.g., by stating:

[**py55**]
```
>>> actr.ACTRModel(environment=environment,\
...     automatic_visual_search=False, eye_mvt_angle_parameter=10)
<pyactr.model.ACTRModel object at 0x7fb347921320>
```

Exercise 2

In the model, only one stimulus was used. But in experiments, it is standard that many stimuli follow each other. Recode the model so it could simulate lexical decision on two (or more) stimuli following each other (e.g., find the word, recall the word, press the key, wait for the next stimulus etc.). In order to test the model, you'll also need to change the stimuli you use in your enviornment. They should look as follows:

[**py56**]
```
>>> word = [{1: {'text': 'elephant', 'position': (320, 180)}},\
...     {1: {'text': 'wug', 'position': (220, 140)}}]
```

To break this down, multiple stimuli are written as a list (enclosed in the [] brackets) and each element in the list is one stimulus, appearing on a screen for the amount of time given when starting simulation.

Appendix: The agreement model

File **ch2_agreement.py**:

```
"""
An example of a very simple model that simulates subject-verb agreement. We abstract away from syntac
"""

import pyactr as actr

car = actr.makechunk(nameofchunk="car",\
                      typename="word", phonology="/ka:/", meaning="[[car]]", category="noun", number=

agreement = actr.ACTRModel()

dm = agreement.DecMem()
dm.add(car)

retrieval = agreement.dmBuffer(name="retrieval", declarative_memory=dm)

g = agreement.goal(name="g")
g.add(actr.chunkstring(string="isa word task agree category 'verb'"))

agreement.productionstring(name="agree", string="""
    =g>
    isa  word
    task trigger_agreement
    category 'verb'
    =retrieval>
    isa  word
    category 'noun'
    syncat 'subject'
    number =x
    ==>
    =g>
    isa  word
    task done
    category 'verb'
    number =x
    """)

agreement.productionstring(name="retrieve", string="""
    =g>
    isa  word
    task agree
    category 'verb'
    ?retrieval>
    buffer empty
    ==>
    =g>
    isa  word
    task trigger_agreement
```

```
    category 'verb'
    +retrieval>
    isa  word
    category 'noun'
    syncat 'subject'
    """)

agreement.productionstring(name="done", string="""
    =g>
    isa  word
    task done
    category 'verb'
    number =x
    ==>
    ~g>""")

if __name__ == "__main__":
    x = agreement.simulation()
    x.run()
```

Appendix: The top-down parser

File **ch2_topdown_parser.py**:

```
"""
A simple top-down parser.
"""

import pyactr as actr

actr.chunktype("parsing", "task stack_top stack_bottom parsed_word ")
actr.chunktype("sentence", "word1 word2 word3")

parser = actr.ACTRModel()

dm = parser.DecMem()
dm.add(actr.chunkstring(string="isa word form 'Mary' cat 'ProperN'"))
dm.add(actr.chunkstring(string="isa word form 'Bill' cat 'ProperN'"))
dm.add(actr.chunkstring(string="isa word form 'likes' cat 'V'"))

retrieval = parser.dmBuffer(name="retrieval", declarative_memory=dm)

g = parser.goal(name="g")
g2 = parser.goal(name="g2", set_delay=0.2)
g.add(actr.chunkstring(string="isa parsing  task parse stack_top 'S'"))
g2.add(actr.chunkstring(string="isa sentence word1 'Mary' word2 'likes' word3 'Bill'"))

parser.productionstring(name="expand: S->NP VP", string="""
        =g>
        isa         parsing
        task        parse
```

```
        stack_top   ’S’
        ==>
        =g>
        isa         parsing
        stack_top   ’NP’
        stack_bottom ’VP’
    """)

parser.productionstring(name="expand: NP->ProperN", string="""
        =g>
        isa         parsing
        task        parse
        stack_top   ’NP’
        ==>
        =g>
        isa         parsing
        stack_top   ’ProperN’
    """)

parser.productionstring(name="retrieve: ProperN", string="""
        =g>
        isa         parsing
        task        parse
        stack_top   ’ProperN’
        =g2>
        isa         sentence
        word1       =w1
        ==>
        =g>
        isa         parsing
        task        retrieving
        =g2>
        isa         sentence
        +retrieval>
        isa         word
        form        =w1
    """)

parser.productionstring(name="retrieve: V", string="""
        =g>
        isa         parsing
        task        parse
        stack_top   ’V’
        =g2>
        isa         sentence
        word1       =w1
        ==>
        =g>
        isa         parsing
        task        retrieving
        =g2>
        isa         sentence
```

```
        +retrieval>
        isa         word
        form        =w1
    """)

parser.productionstring(name="scan: string", string="""
        =g>
        isa         parsing
        task        retrieving
        stack_top   =y
        stack_bottom =x
        =retrieval>
        isa         word
        form        =w1
        cat         =y
        =g2>
        isa         sentence
        word1       =w1
        word2       =w2
        word3       =w3
        ==>
        =g>
        isa         parsing
        task        print
        stack_top   =x
        stack_bottom empty
        parsed_word =w1
        =g2>
        isa         sentence
        word1       =w2
        word2       =w3
        word3       empty
    """)

parser.productionstring(name="expand: VP -> V NP", string="""
        =g>
        isa         parsing
        task        parse
        stack_top   'VP'
        ==>
        =g>
        isa         parsing
        stack_top   'V'
        stack_bottom 'NP'
    """)

parser.productionstring(name="print parsed word", string="""
        =g>
        isa         parsing
        task        print
        =g2>
        isa         sentence
```

```
        word1      ~empty
        ==>
        =g2>
        isa         sentence
        !g>
        show        parsed_word
        =g>
        isa         parsing
        task        parse
        parsed_word None""")

parser.productionstring(name="done", string="""
        =g>
        isa         parsing
        task        print
        =g2>
        isa         sentence
        word1       empty
        ==>
        !g>
        show        parsed_word
        ~g2>
        ~g>""")

if __name__ == "__main__":
    x = parser.simulation()
    x.run()
    print(dm)
```

Appendix: The lexical decision model

File **ch2_lexical_decision_1.py**:
?? PythonTeX ??

Chapter 3

Performance

3.1 Introduction

The goal of ACT-R is to provide accurate cognitive models of learning and performance, as well as neural mapping of cognitive activities. So far, our models were lacking in all these respects. We will start closing the gap by considering several cases of how ACT-R is mapped to performance.

When studying performance, we are usually interested in two measures: (i) what response people choose given some stimulus, (ii) how much time it takes them to react. In linguistics, the first measure often appears as the "Accept–Reject" response when people judge the grammatical or interpretational status of a sentence or a discourse. But other responses can fit here, as well, for example, answers in forced-choice tasks, responses in lexical decision tasks etc. The second measure often encodes how much time it took one to choose a particular response, but other options also exist, e.g., how much time it took to shift eye gaze, to move a mouse etc. We will now go into the part of ACT-R models that can make predictions on both counts.

3.2 Understanding the (basic) activation equation

(7) 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 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: $S_{ji} \approx \log\left(\frac{prob(i|j)}{prob(i)}\right)$

3.2.1 The base-level learning equation

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

(9) Anderson and Schooler (1991, 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.

(10) **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.

```
> ebbinghaus_data = read.csv("ebbinghaus_retention_data.csv", header=T)
> ebbinghaus_data

  delay_in_hours percent_savings
1           0.33            58.2
2           1.00            44.2
3           8.80            35.8
4          24.00            33.7
5          48.00            27.8
6         144.00            25.4
7         744.00            21.1

> summary(ebbinghaus_data)
```

```
delay_in_hours  percent_savings
Min.   :  0.3   Min.   :21.1
1st Qu.:  4.9   1st Qu.:26.6
Median : 24.0   Median :33.7
Mean   :138.6   Mean   :35.2
3rd Qu.: 96.0   3rd Qu.:40.0
Max.   :744.0   Max.   :58.2
```

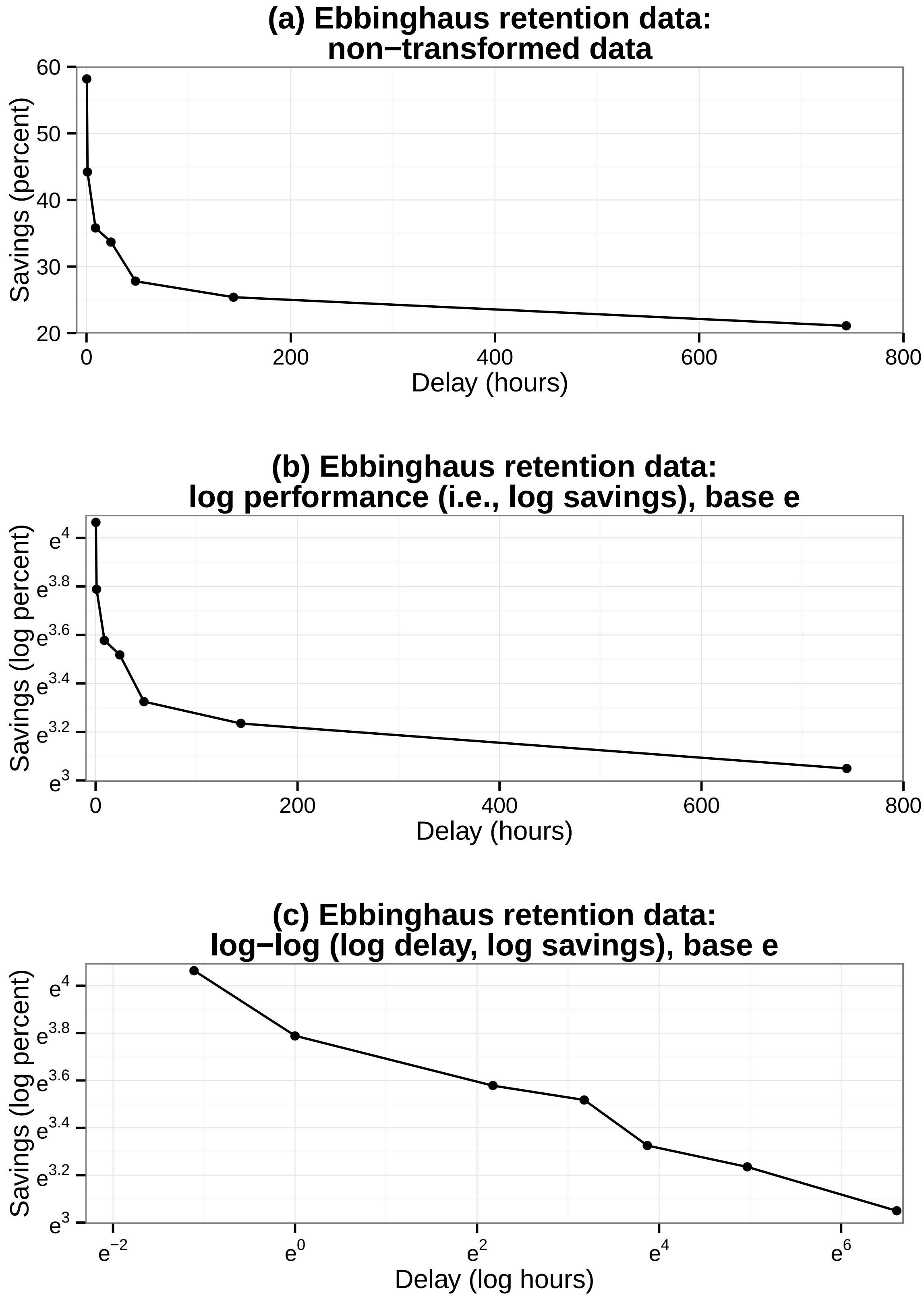


Figure 3.1: Ebbinghaus retention data

The forgetting curve plotted in panel (a) of Figure 3.1 is sometimes taken to reflect an underlying negative exponential forgetting function of the form:

(11) $P = Ae^{-bT}$, 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 3.1 shows that is not the case:

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

Instead, we see a power function, as panel (c) of Figure 3.1 shows. That is, performance is a linear function of time only if you log-transform both of them:

(13) $\log(P) = \log(A) - b\log(T)$, 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):

(14) 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, 400)

In sum:

(15) The base-level activation equation encodes that (see Anderson and Schooler 1991, 407, and Anderson et al. 2004, 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.

(16) 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 that labels the arrow.

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

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.

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

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.

We define a `base_activation` function: its inputs are the presentation times for the chunk, and also the moments of time at which to obtain activation. The output is the base-level activation values at the corresponding moments of time.

```
> base_activation <- function(pres_times, moments) {
+     base_act = numeric(length=length(moments))
+     for (i in 1:length(moments)) {
+         base_act[i] = sum(1/sqrt(moments[i] - pres_times[pres_times<moments[i]]))
+     }
+     base_act[which(base_act!=0)] = log(base_act[which(base_act!=0)])
+     return(base_act)
+ }
>
> pres_times = seq(0, 1200, length.out=5)
> moments = 0:3500
> base_act = base_activation(pres_times, moments)
```

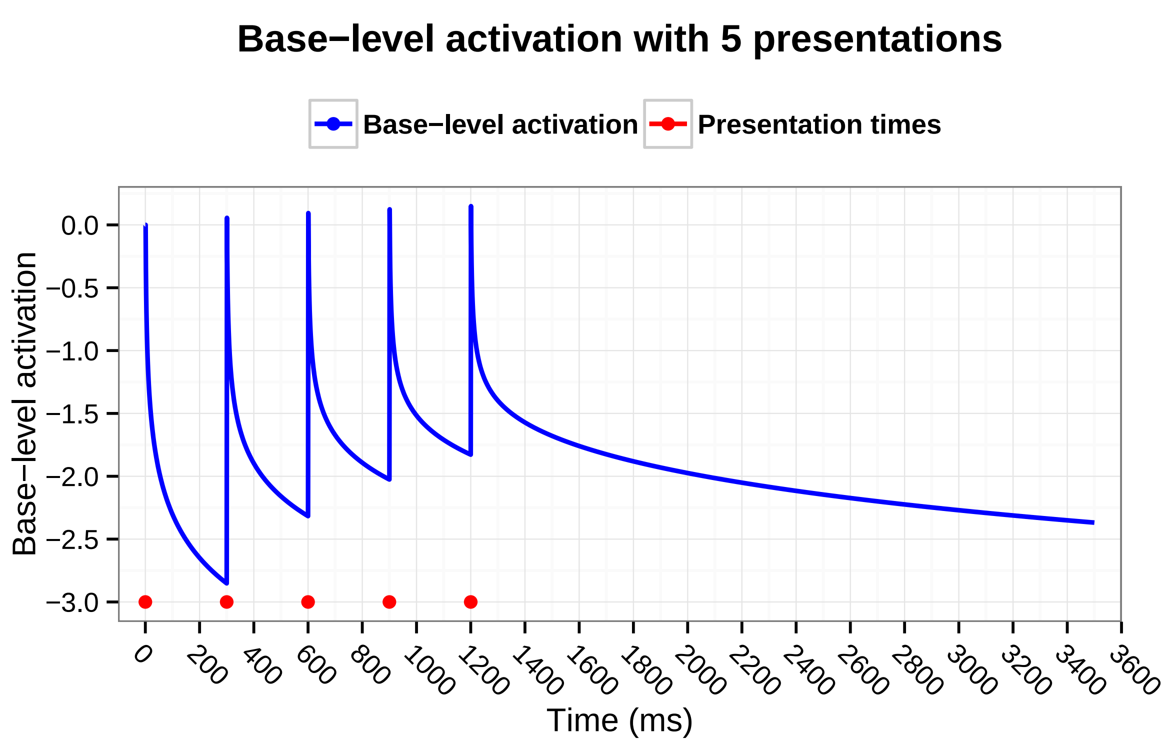


Figure 3.2: Base-level activation as a function of time

3.2.2 The attentional weighting equation

(17) 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.

3.2.3 The associative strength equation

(18) 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.

3.3 Activation, probability of retrieval, and latency of retrieval

(19) 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.

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

(21) 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 the base-level activation. We assume:

- noise $s = 0.4$
- threshold $\tau = -2$
- latency factor $F = 50$ (ms)

Note that according to the above equation, $F \approx 0.35e^{-2} \approx 0.35 \times 0.1353 \approx 0.04736$ (s), so our value of 50 ms is very close to this. Also note that this value is different from $F = 0.46$ in Vasishth et al. (2008, 692)), or $F = 0.14$ in Lewis and Vasishth (2005, 382).

```
> pres_times = seq(0, 1200, length.out=5)
> moments = 0:3500
> base_act = base_activation(pres_times, moments)
>
> s = 0.4
> tau = -2
> F = 50 # in ms
>
> prob_retrieval = 1/(1 + exp(-(base_act - tau)/s))
> latency_retrieval = F * exp(-base_act)
```

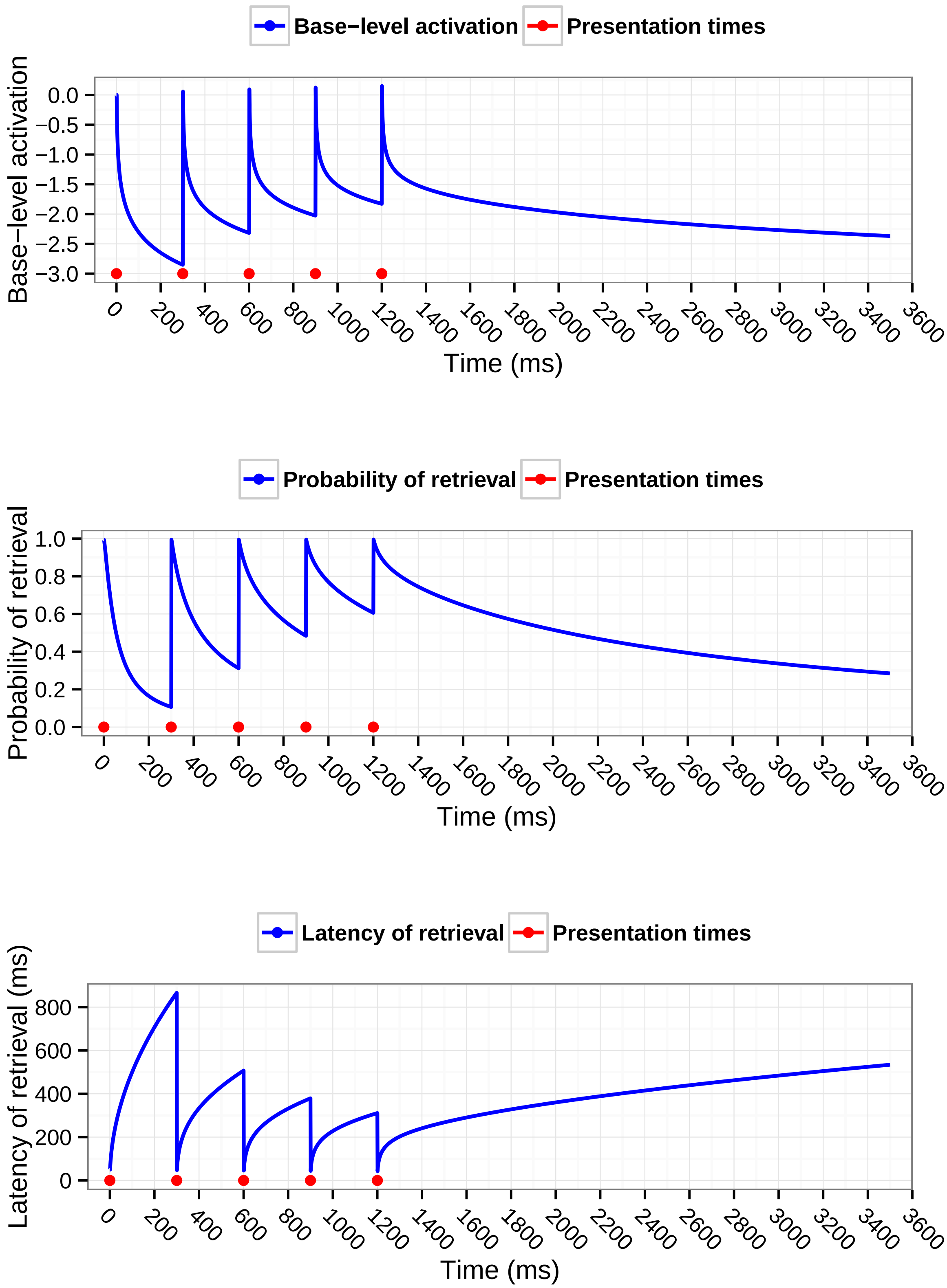


Figure 3.3: Base-level activation, probability of retrieval, and latency of retrieval as a function of time

3.3.1 Probability of retrieval

Let's take a closer look at probability of retrieval. We plot the odds of retrieval in addition to probability of retrieval, and also plot odds against activation.

```
> pres_times = seq(0, 1200, length.out=5)
> moments = 0:3500
> base_act = base_activation(pres_times, moments)
>
> s = 0.4
> tau = -2
>
> prob_retrieval = 1/(1 + exp(-(base_act - tau)/s))
> odds_retrieval = exp((base_act - tau)/s)
```

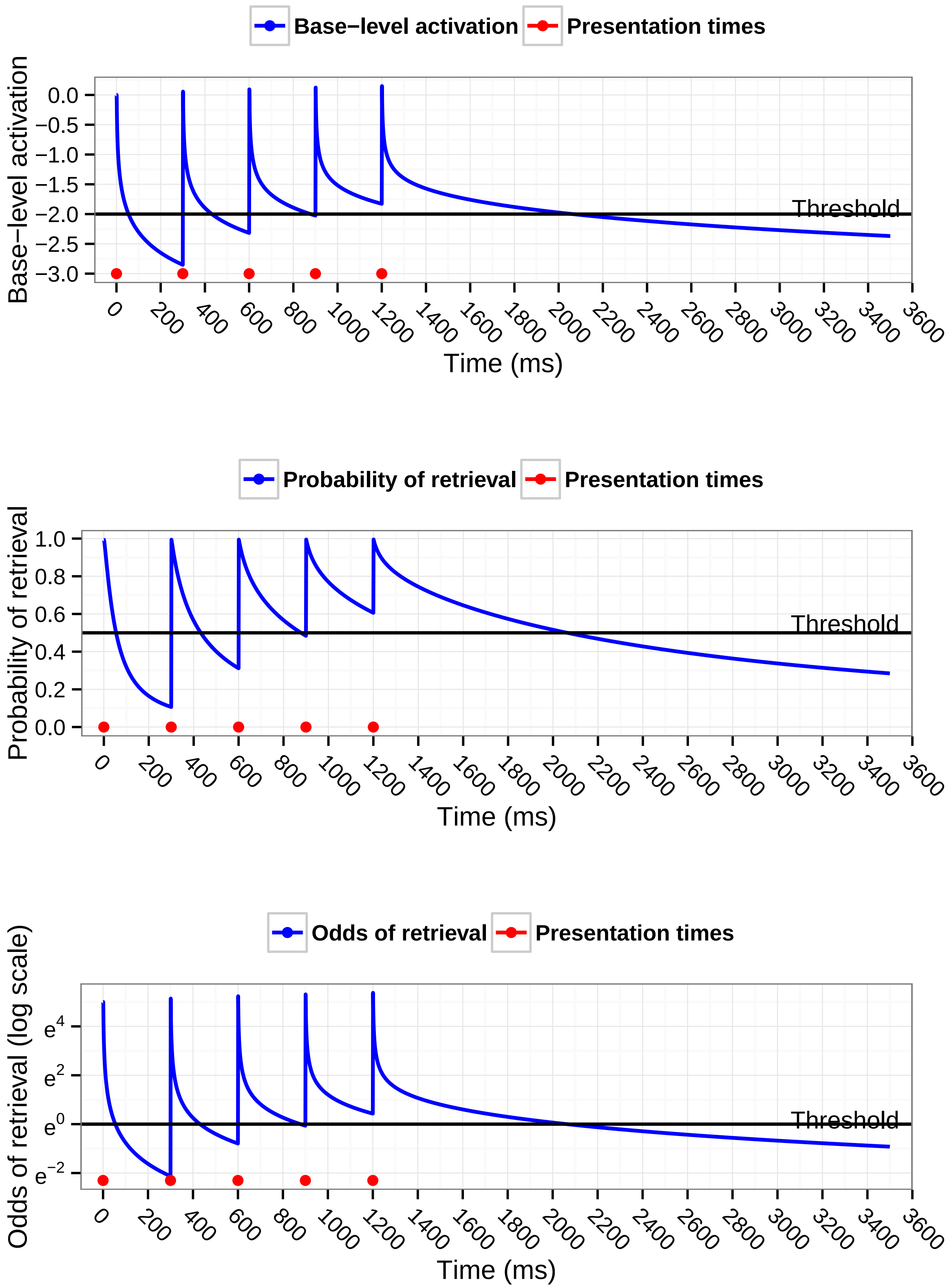


Figure 3.4: Base-level activation, probability of retrieval, and odds of retrieval as a function of time

Let's plot probability and odds of retrieval against activation. Note the *linear* relationship between activation and odds of retrieval on the log scale, i.e., log-odds, i.e., logits.

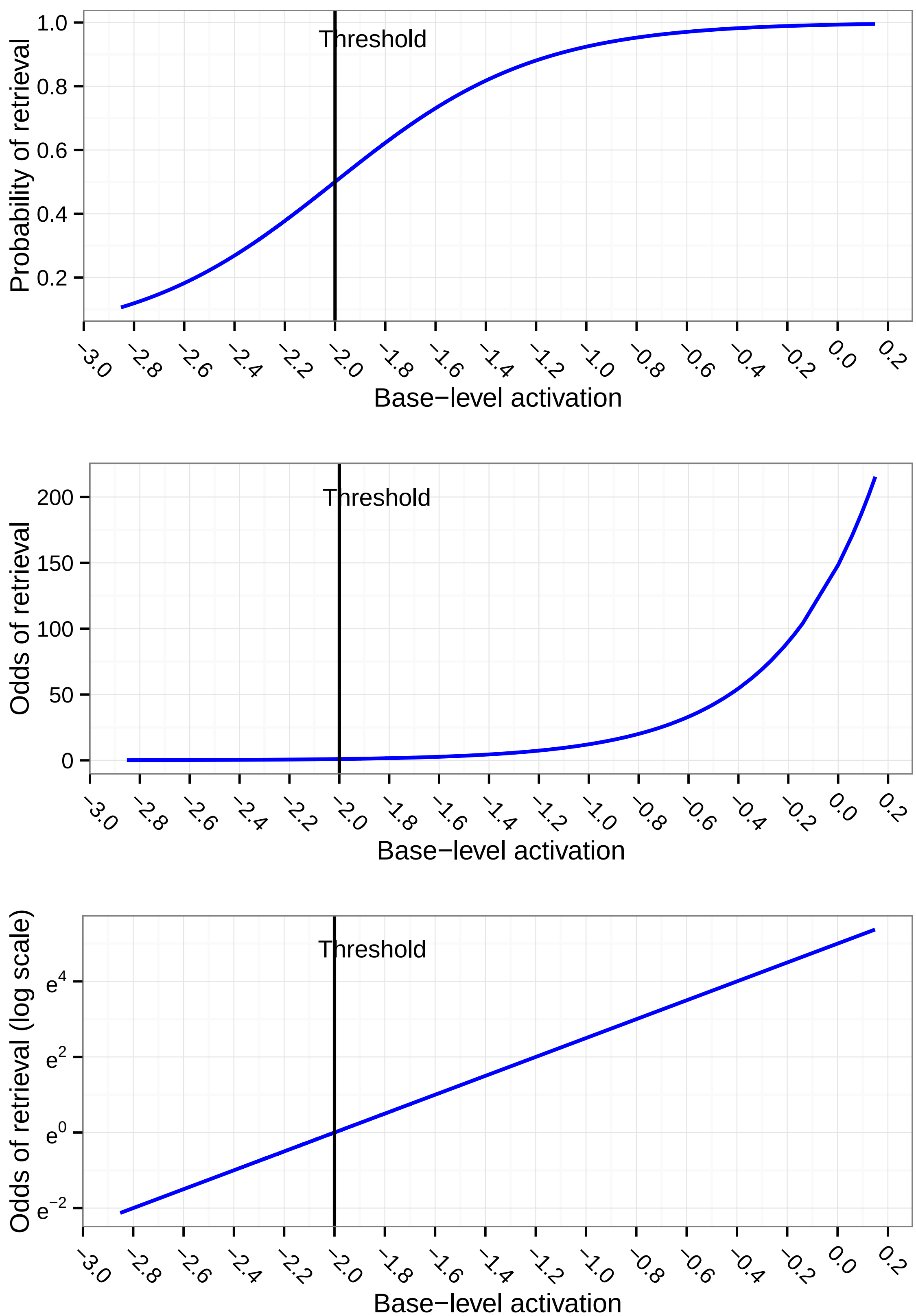


Figure 3.5: Probability and odds of retrieval as a function of activation

3.3.2 Latency of retrieval

Let's plot time of retrieval and log time of retrieval against activation – and also against log odds of retrieval. Note the *linear* relationship between activation and time of retrieval (or odds of retrieval) on the log scale.

You can get an intuitive interpretation for the latency scale parameter F by looking at how much time it takes to retrieve a chunk that has a threshold (τ) activation.

```
> pres_times = seq(0, 1200, length.out=5)
> moments = 0:3500
> base_act = base_activation(pres_times, moments)
>
> s = 0.4
> tau = -2
>
> prob_retrieval = 1/(1 + exp(-(base_act - tau)/s))
> odds_retrieval = exp((base_act - tau)/s)
>
> F = 50 # in ms
> latency_retrieval = F * exp(-base_act)
```

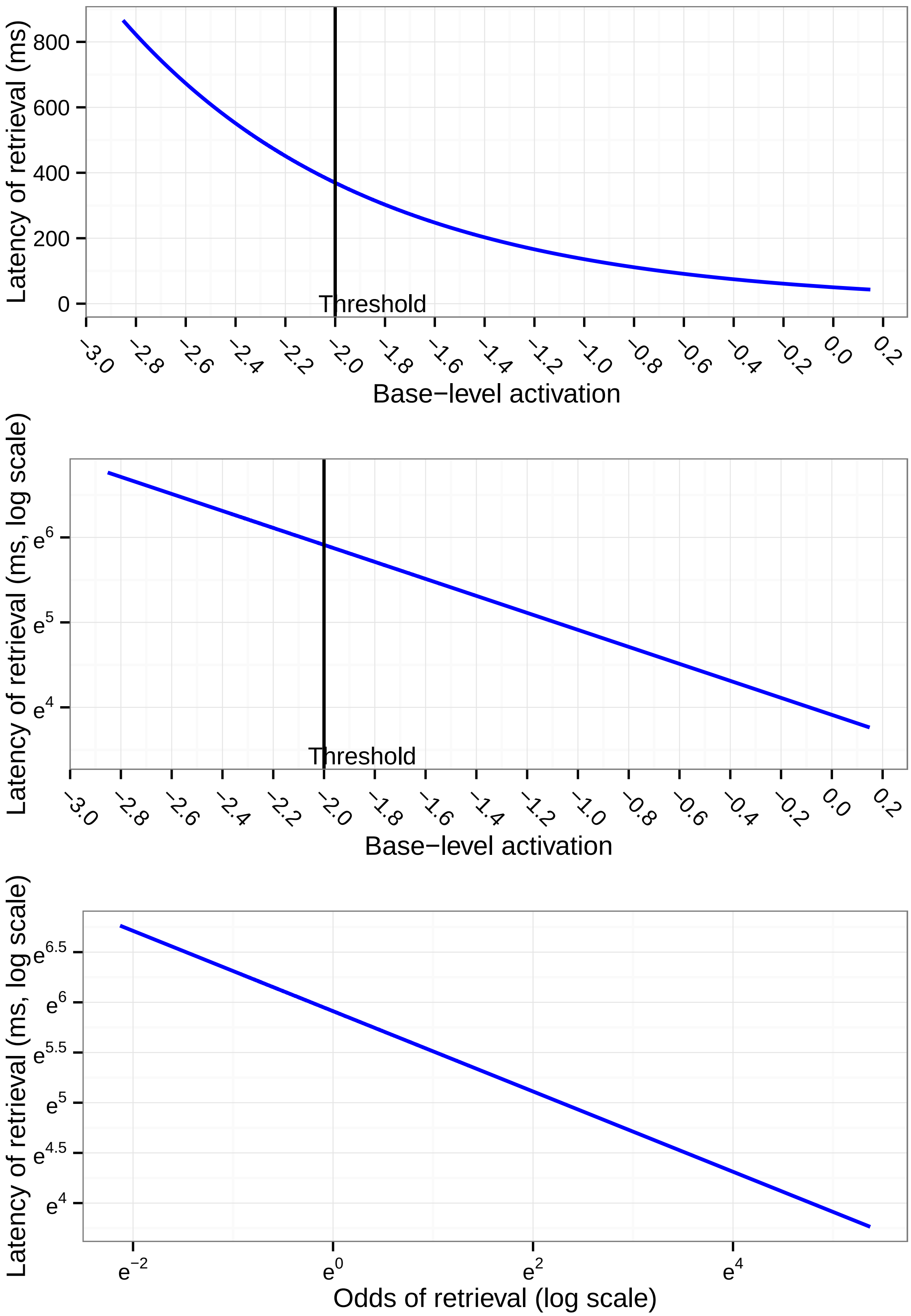


Figure 3.6: Time of retrieval as a function of activation and as a function of odds of retrieval

Sub-symbolic architecture of ACT-R
Adrian – fill in?
Retrieval – based on activation for a chunk i – A_i

$$A_i = B_i + \sum_k \sum_j W_{kj} * S_{ji}$$

Base-level activation B_i

$$B_i = ln(\sum_{j=1}^{n} (t_j^{-d})) + \epsilon$$

Latency of retrieval:

$$T = Fe^{-fA_i}$$

Failure of retrieval: whenever A_i below a particular threshold τ

3.4 Modelling performance

We will now consider several studies to justify the sub-symbolic part we just introduced.

3.4.1 Modelling lexical decision tasks

In the previous chapter, we introduced a simple lexical decision task. We noted there that while the model might be sufficient (for our needs) in the way it simulates the interaction with environment, it is too simplistic in its assumption about memory since memory retrievals are not dependent on any parameters of the retrieved word.

One very robust parameter affecting latencies and accuracies of lexical decision tasks is frequency (Whaley, 1978). In fact, frequency effects have been found not just in lexical decision tasks, but in many, if not all, tasks that involve some kind of lexical processing (Forster, 1990; Monsell, 1991). Such frequency effects are not arbitrary, they impose a specific form. It has been known since Howes and Solomon (1951) that lexical access can be well approximated as the log-function of frequency.

While the approximation between log-frequency and lexical access is good, it is not perfect. Murray and Forster (2004), who studied the role of frequency in extreme detail, pointed out limits of log-frequency. They collected responses and response times in the lexical decision task using words from 16 frequency bands, as summarized in Table 3.1, and showed that log-frequency gets middle values right, but it tends to underestimate the amount of time needed to access the words at extreme ends of the frequency scale.

Murray and Forster (2004) take this as an argument for a specific information retrieval mechanism, the so-called Rank Hypothesis (see Forster 1976, 1992). But as they note, this is not the only way to model retrieval mechanism that fits their data. One popular method is to treat frequency effects as skill learning, which is standardly represented as a power function (Newell and Rosenbloom, 1981; Anderson, 1982; Logan, 1990). Since skill learning is implemented in ACT-R, we will look at this approach in more detail.

Group	Frequency range	Mean frequency
1	315–197	242.0
2	100–85	92.8
3	60–55	57.7
4	42–39	40.5
5	32–30	30.6
6	24–23	23.4
7	19	19.0
8	16	16.0
9	14-13	13.4
10	12–11	11.5
11	10	10.0
12	9	9.0
13	7	7.0
14	5	5.0
15	3	3.0
16	1	1.0

Table 3.1: Frequency bands of words used in Murray and Forster (2004) (Exp. 1). Frequency is reported in number of tokens per 1 million words

The power function could be seen as relating latencies to the number of practice trials. We represent it in the following form:

(22) $$t = t_0 + a * x^{-b}$$

t is the latency to be estimated. t_0 is the asymptote, a is the multiplier, b the exponent (free parameters), and x represents (some form of) practice, e.g., word frequency. Given what the function describes, it also goes under the name of the power law of practice or the power law of learning. For the lexical decision task, we could say the following: the function treats t, the latency needed to decide that the stimulus is an existing word, as a function of frequency to the power of $-b$, scaled and shifted from 0.

The power function has also been used to describe forgetting (Wickelgren, 1972; Anderson and Schooler, 1991; Schooler and Anderson, 1997). In that case, the independent variable x in (22) represents the time elapsed between learning and testing. For the lexical decision task, we could say that in the law of forgetting, the latency to recognize a word is the function of x, the time elapsed since learning the word, raised by the exponent of $-b$, scaled and shifted from 0.

The two functions are closely related (see, e.g., Anderson and Lebiere 1998; Anderson et al. 1999) and when there is only a single instance of learning they colapse. But they are not identical, as we'll see in a second.

Instances of both laws can be fitted in ACT-R. The second one is fit by modulating the d

parameter in base-level learning:

$$B_i = ln(\sum_{j=1}^{n}(t_j^{-d})) + \epsilon \tag{3.1}$$

The first one is fit by modulating the f parameter in the formula that relates latency to activation:

$$T = Fe^{-fA_i} \tag{3.2}$$

Which of the two is correct for lexical decision? Before being able to answer that, we have to decide one issue: how is frequency related to the time elapsed between learning and the experiment (t_j in the formula above)? Let's assume we have a 15-year old speaker. How would we estimate time points at which a word was used?

We know how much time elapsed in his life. If we know how many words the speaker was exposed to in total, we can easily calculate how many times a particular word was used on average since we know their frequency. Keeping the simulation as general as possible we can then let each word occurrence appear randomly during the life span.[1]

What remains to be solved is the amount of words a speaker is exposed to per year. A good approximation of that, based on recordings of 42 families, can be found in Hart and Risley (1995). They estimate that children comprehend between 10 million to 35 million words a year, depending to a large extent on the social class of the family, and this amount increases linearly with age. According to the study, a 15-year old would be exposed to between 50 and 175 million words in total. We'll consider the middle value, 112.5 million words, as the total amount of words a speaker is exposed to. This is a very conservative measure (since we ignore production, as well as the role of mass media) but that's ok. It is important to note that absolute numbers are of little consequence here, since we are not interested in the *absolute* effect of frequency, but in its *relative* effect (i.e., at this point we do not want to predict how much time a word from one frequency band requires, but how much a word requires compared to a word from another frequency band).

With this background, we can fit three models to the data from the lexical decision task:

- The basic log-frequency model. This model is not related to ACT-R but it is a common baseline in lexical decision tasks; it estimates the intercept + the scale parameter of log-frequency.

- The ACT-R model representing the law of forgetting. The model estimates the intercept, the d parameter and the parameter F. The last parameter scales activation to match latencies more closely.

- The ACT-R model representing the law of practice. The model estimates the intercept, the f parameter and the parameter F.

[1] Alternatively, we could simplify this and assume that word occurrences of every word are evenly spaced during the lifetime. This makes computation easier and closely approximates random sampling in mid-frequency and high-frequency words (but it underestimates activation of low-frequency words compared to the random sampling method).

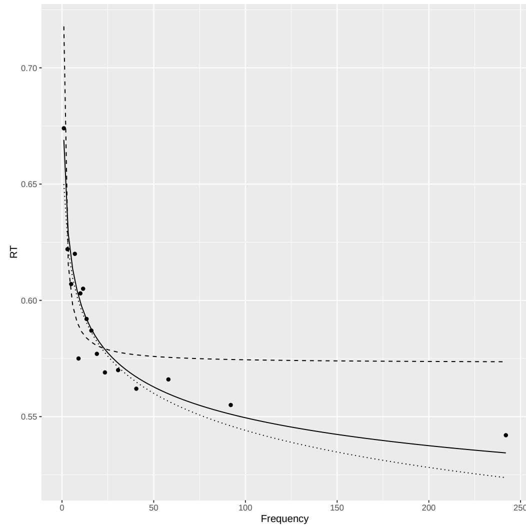


Figure 3.7: Model fitting of Exp. 1, Murray and Forster (2004). The solid line represents the best fit of the f parameter, the dashed line is the best fit of the d parameter, the dotted line is the best fit of log-frequency to latencies.

Note that the second and third models are just specific instances of the power function shown in (22), in which F is the multiplier, the intercept corresponds to t_0 and f and d represent exponents.

Fig. 3.7 compares the two instances of the law in ACT-R and the baseline log-frequency model. In the figures we present the best fit of the f and the d parameters, as well as the other free parameters of the power law: t_0 and the scaling parameter a (which corresponds to the latency factor F in ACT-R).

The log-model (the dotted line) represents a relatively good fit, even though, as we noted above, it tends to underestimate latencies at extreme ends.

The second model, fitting the d parameter, is shown using the dashed line. This model is worse than the baseline.

The best model is the third model, fitting the f parameter (the solid line). It is very close to the log-model in mid-values, but unlike the log-model, it is also very close to the data in the extreme ends of the scale.

All this shows that a lexical decision task can be modelled as a case of the law of practice. More importantly, we also see that this law can be captured in ACT-R.

3.5 pyactr model of lexical decision

We will now show how the model that we considered in somewhat abstract terms can be inputed in `pyactr`.

The best fit of the model had the following values: t_0 should be 420 ms, f should be 0.14 and F should be 0.13. t_0 represents the time it requires to do the task when disregarding the retrieval of words.

We already created a model for lexical decision in the previous chapter. We will carry that model over to this chapter, but we will assume one modification.

In the previous chapter, we assumed that the eye focus is away from the stimulus and participants have to first look for the word on the screen (that is, using visual location, they have to find it, and then they have to shift attention towards the word). That was fine when we wanted to explore the properties of the visual module, but it's hardly a good representation of how lexical decision tasks proceed. Normally it is signalled to participants where a word would appear, precisely to avoid the delay caused by focus shifts. So let's start the simulation with the focus position of the middle of the screen (the same position where an item will appear).

[**py1**]
```
>>> import pyactr as actr

>>> environment = actr.Environment(focus_position=(320, 180))
```

When calling the model, we will now have to specify two values in the parameters. The f value is set up by `latency_exponent`, the F value by `latency_factor`. (For the full list of ACT-R parameters and their names in the model, see Appendix.)

[**py57**]
```
>>> model = actr.ACTRModel(environment=environment, subsymbolic=True, automatic_visual_search=Tr
```

A careful readed probably noticed that we modified four other values: `subsymoblic`. `automatic_visual_search`, `emma_noise` and `retrieval_threshold`. The first parameter simply states that we are going to use the sub-symbolic system of ACT-R. The second parameter states that visual search is automatic: if a word appears on a screen, it is automatically buferred in the visual location, the visual module will not wait for a specific command from production rules. The third parameter, `emma_noise`, controls whether the visual module should provide deterministic values, or whether the values should be drawn from probability distributions, as discussed in Chapter **??**. Since we do are not interested in simulating variance in visual encoding, we will simplify the matter here and set this parameter to `False` (the visual module is fully deterministic).

The last parameter specifies the level of the retrieval threshold. The threshold controls whether a chunk present in the memory can be retrieved or not: chunks whose activation falls below the threshold cannot be retrieved. Since we at this point want all words to be retrieved (we will modify that position later) and since the lowest activation is around -4.5 given our calculation of activation, we have two options: either increase activation or decrease the retrieval threshold. We noted in the previous section that the way we calculate activation is very conservative and for this reason the first option might be preferred. For example, we have a very conservative estimate of the average of words spoken per year. We also count all the time in one's life towards the chunk decay, while in reality some moments (e.g., sleeping) most likely have less effect on decay of word knowledge than others. Taking these considerations into account would definitely yield higher activation estimates. But the second option (decreasing the retrieval threshold) is very simple and because of its simplicity, we will use it here.

The rest of the model is identical to the first version of the lexical decision task, discussed in the previous chapter:

[**py58**]
```
>>> actr.chunktype("goal", "state")
>>> actr.chunktype("word", "form")
```

```
>>> model.productionstring(name="attend_probe", string="""
...     =g>
...     isa     goal
...     state   'start'
...     =visual_location>
...     isa    _visuallocation
...     ==>
...     =g>
...     isa     goal
...     state   'recall'
...     =visual_location>
...     isa     _visuallocation
...     +visual>
...     isa     _visual
...     cmd     move_attention
...     screen_pos =visual_location""")

>>> model.productionstring(name="prepare_retrieving", string="""
...     =g>
...     isa     goal
...     state   'recall'
...     =visual>
...     isa     _visual
...     value   =val
...     ==>
...     =g>
...     isa     goal
...     state   'retrieving'
...     word    =val""")

>>> model.productionstring(name="retrieving", string="""
...     =g>
...     isa     goal
...     state   'retrieving'
...     word    =val
...     ==>
...     =g>
...     isa     goal
...     state   'retrieval_done'
...     +retrieval>
...     isa     word
...     form    =val""")

>>> model.productionstring(name="can_recall", string="""
...     =g>
...     isa     goal
...     state   'retrieval_done'
...     ?retrieval>
...     buffer  full
...     state   free
...     ==>
```

```
...     ~g>
...     +manual>
...     isa     _manual
...     cmd     press_key
...     key     'J'""")

>>> model.productionstring(name="cannot_recall", string="""
...     =g>
...     isa     goal
...     state   'retrieval_done'
...     ?retrieval>
...     buffer  empty
...     state   error
...     ==>
...     ~g>
...     +manual>
...     isa     _manual
...     cmd     press_key
...     key     'F'""")
```

What is new, though, is the way we want to run this model. We are not simply interested in inputing one word in the model and running the simulation with that. Rather, we want to check what happens for any word representing the mean value of each band in the experiment of Murray and Forster (2004). We will do this by creating the dictionary of words that store their corresponding frequencies, as well as the experimentally observed reaction times:

[**py59**]

```
>>> FREQ = {}
>>> FREQ['nothing'] = (242, 0.542)
>>> FREQ['section'] = (92, 0.555)
>>> FREQ['crowd'] = (58, 0.566)
>>> FREQ['bridge'] = (40.5, 0.562)
>>> FREQ['knife'] = (30.6, 0.57)
>>> FREQ['bunch'] = (23.4, 0.569)
>>> FREQ['medium'] = (19, 0.577)
>>> FREQ['subtle'] = (16, 0.587)
>>> FREQ['punish'] = (13.4, 0.592)
>>> FREQ['patent'] = (11.5, 0.605)
>>> FREQ['denial'] = (10, 0.603)
>>> FREQ['attain'] = (9, 0.575)
>>> FREQ['drain'] = (7, 0.62)
>>> FREQ['assault'] = (5, 0.607)
>>> FREQ['disdain'] = (3, 0.622)
>>> FREQ['amber'] = (1, 0.674)
```

We will now create a loop that

- picks one word from this dictionary
- creates a past experience for that word, based on the frequency of that word, by choosing as many random moments in the past as the frequency of the word would allow
- runs the simulation with that past experience, using the word as a stimulus

- prints the time from the start of the simulation until pressing the key (i.e., the whole procedure of lexical decision task)
- goes to Step 1

Since there is some randomization required, we will also have to import a new package, `random`.

[**py60**]
```
1  >>> import random
2
3  >>> SEC_IN_YEAR = 365*24*3600
4  >>> SEC_IN_TIME = 15*SEC_IN_YEAR
5
6  >>> for lemma in FREQ:
7  ...             dm = model.DecMem()
8  ...             for _ in range(int(FREQ[lemma][0]*112.5)):
9  ...                 dm.add(actr.makechunk(typename="word", form=lemma), time=random.randint(-SEC
10 ...             word = {1: {'text': lemma, 'position': (320, 180)}}
11 ...             retrieval = model.dmBuffer("retrieval", dm)
12 ...             g = model.goal("g", default_harvest=dm)
13 ...             g.add(actr.makechunk(nameofchunk='start', typename="goal", state='start'))
14 ...             environment.current_focus = [320,180]
15 ...             sim = model.simulation(realtime=False, gui=True, trace=False, environment_proces
16 ...             while True:
17 ...                 sim.step()
18 ...                 if sim.current_event.action == "KEY PRESSED: J":
19 ...                     print(lemma, FREQ[lemma], sim.show_time())
20 ...                     break
21 ...
22 attain (9, 0.575) 0.5988
23 assault (5, 0.607) 0.6169
24 disdain (3, 0.622) 0.6351
25 amber (1, 0.674) 0.6643
26 drain (7, 0.62) 0.607
27 punish (13.4, 0.592) 0.5916
28 denial (10, 0.603) 0.5976
29 crowd (58, 0.566) 0.5601
30 nothing (242, 0.542) 0.5345
31 medium (19, 0.577) 0.5833
32 patent (11.5, 0.605) 0.5964
33 knife (30.6, 0.57) 0.5734
34 section (92, 0.555) 0.5507
35 subtle (16, 0.587) 0.585
36 bridge (40.5, 0.562) 0.5673
37 bunch (23.4, 0.569) 0.5788
```

The crucial bit in lines 22–37 (output) is the comparison of the last column, predicted RTs, to the last but one column, observed RTs. The fit is very close. When you run the model, predicted RTs will probably differ very slightly from the ones presented here since the results depend on random sampling in the past experience. To get more robust findings, it would be good to repeat this simulation several times and average observed RTs. But from our own experience, that result will not drastically alter the goodness of the fit in this case.

3.6 Exercises

3.6.1 Exercise 1

Appendix: The lexical decision model

File **ch3_lexical_decision_2.py**:

```
"""
A simple model of lexical decision.
"""

import random

import pyactr as actr

environment = actr.Environment(focus_position=(320, 180))
model = actr.ACTRModel(environment=environment, subsymbolic=True, automatic_visual_search=True, activ

actr.chunktype("goal", "state")
actr.chunktype("word", "form")

SEC_IN_YEAR = 365*24*3600
SEC_IN_TIME = 15*SEC_IN_YEAR

FREQ = {}
FREQ['nothing'] = 242*112.5
FREQ['section'] = 92*112.5
FREQ['crowd'] = 58*112.5
FREQ['bridge'] = 40.5*112.5
FREQ['knife'] = 30.6*112.5
FREQ['bunch'] = 23.4*112.5
FREQ['medium'] = 19*112.5
FREQ['subtle'] = 16*112.5
FREQ['punish'] = 13.4*112.5
FREQ['patent'] = 11.5*112.5
FREQ['denial'] = 10*112.5
FREQ['attain'] = 9*112.5
FREQ['drain'] = 7*112.5
FREQ['assault'] = 5*112.5
FREQ['disdain'] = 3*112.5
FREQ['amber'] = 1*112.5

model.productionstring(name="attend_probe", string="""
    =g>
    isa     goal
    state   'start'
    =visual_location>
    isa    _visuallocation
    ==>
    =g>
    isa     goal
    state   'recall'
    =visual_location>
    isa     _visuallocation
    +visual>
```

```
    isa     _visual
    cmd     move_attention
    screen_pos =visual_location""")

model.productionstring(name="prepare_retrieving", string="""
    =g>
    isa     goal
    state   'recall'
    =visual>
    isa     _visual
    value   =val
    ==>
    =g>
    isa     goal
    state   'retrieving'
    word    =val""")

model.productionstring(name="retrieving", string="""
    =g>
    isa     goal
    state   'retrieving'
    word    =val
    ==>
    =g>
    isa     goal
    state   'retrieval_done'
    +retrieval>
    isa     word
    form    =val""")

model.productionstring(name="can_recall", string="""
    =g>
    isa     goal
    state   'retrieval_done'
    ?retrieval>
    buffer  full
    state   free
    ==>
    ~g>
    +manual>
    isa     _manual
    cmd     press_key
    key     'J'""")

model.productionstring(name="cannot_recall", string="""
    =g>
    isa     goal
    state   'retrieval_done'
    ?retrieval>
    buffer  empty
    state   error
    ==>
```

```
    ~g>
    +manual>
    isa     _manual
    cmd     press_key
    key     'F'""")

if __name__ == "__main__":
    for lemma in FREQ:
        for _ in range(10):
            dm = model.DecMem()
            for _ in range(int(FREQ[lemma])):
                dm.add(actr.makechunk(typename="word", form=lemma), time=random.randint(-SEC_IN_TIME,
            word = {1: {'text': lemma, 'position': (320, 180)}}
            retrieval = model.dmBuffer("retrieval", dm)
            g = model.goal("g", default_harvest=dm)
            g.add(actr.makechunk(nameofchunk='start', typename="goal", state='start'))
            environment.current_focus = [320,180]
            sim = model.simulation(realtime=False, gui=True, trace=False, environment_process=environ
            while True:
                sim.step()
                if sim.current_event.action == "KEY PRESSED: J":
                    print(lemma, FREQ[lemma], sim.show_time())
                    break
```

ACT-R – subsymbolic parameters

Base-level learning

Switched on by subsymbolic=True.
The equation describing learning of base-level activation for a chunk i is:

$$B_i = ln(\sum_{j=1}^{n}(t_j^{-d})) + \eta$$

- n: The number of presentations for chunk i
- t_j : The time since the jth presentation
- d: The decay parameter (set by decay)
- η: the instantaneous noise

The (instantaneous) noise:

$$\sigma^2 = s^2 * \pi^2/3$$

- s: The noise parameter (set by instantaneous_noise)

Retrieval latency:

$$T = Fe^{-A}$$

- A: Activation of the chunk retrieved
- F: The latency parameter (set by latency_parameter)

Retrieval latency when retrieval fails:

$$T = Fe^{-\tau}$$

- τ: The retrieval threshold (set by retrieval_threshold)
- F: The latency parameter (set by latency_parameter)

For an example see u4_paired in **tutorials**.

Source and activation

Switched on by subsymbolic=True and specifying buffer_spreading_activation (see below).

$$A_i = B_i + \sum_k \sum_j W_{kj} * S_{ji}$$

- A_i: activation of the chunk i
- B_i: base-level activation, see above

- W_{kj}: the amount of activation from source j in buffer k
- S_{ji}: the strength of association from source j to chunk i

W_{kj} is set by buffer_spreading_activation. The value of this parameter is a dictionary in which keys specify what buffers should be used for spreading activations, values specify the amount of activation in these buffers.

$$S_{ji} = S - ln(fan_j)$$

- S: the maximum associative strength (set by strength_of_association)
- fan_j: the number of chunks in declarative memory in which j is the value of a slot plus one for chunk j being associated with itself

For an example see u5_fan in **tutorials**.

Adding partial matching

Switched on by subsymbolic=True and partial_matching=True.

$$A_i = B_i + \sum_k \sum_j W_{kj} * S_{ji} + \sum_l M_{li}$$

- M_{li}: The similarity between the value l in the retrieval specification and the value in the corresponding slot of chunk i

The similarity currently only uses default values - a maximum similarity (0) and a maximum different (-1). To be added: let the modeler set these values. For an example see u5_grouped in **tutorials**.

Utility in production rules

Switched on by partial_matching=True. The (utility) noise:

$$\sigma^2 = s^2 * \pi^2 / 3$$

- s: The noise parameter (set by utility_noise)

Each rule can specify its own utility (by having the parameter utility=n, where n is a number). Each rule can also specify reward it creates for utility learning (by having the parameter reward=n, where n is a number). Utility learning is set by utility_learning=True. The learning rate for utility learning is set by utility_alpha. For an example see u6_simple in **tutorials**.

Bibliography

Anderson, John R. 1982. Acquisition of cognitive skill. *Psychological review* 89:369.

Anderson, John R. 1990. *The adaptive character of thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Anderson, John R. 2007. *How can the human mind occur in the physical universe?*. Oxford University Press.

Anderson, John R., Daniel Bothell, and Michael D. Byrne. 2004. An integrated theory of the mind. *Psychological Review* 111:1036–1060.

Anderson, John R, Jon M Fincham, and Scott Douglass. 1999. Practice and retention: a unifying analysis. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 25:1120–1136.

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. *Psychological Science* 2:396–408.

Davies, Martin. 2001. Knowledge (explicit and implicit): Philosophical aspects. In *International encyclopedia of the social and behavioral sciences*, ed. N. J. Smelser and B. Baltes, 8126–8132. Elsevier.

Ebbinghaus, Hermann. 1913. *Memory: A contribution to experimental psychology*. New York: Teachers College, Columbia University. URL `http://psychclassics.yorku.ca/Ebbinghaus/index.htm`.

Forster, Kenneth. 1992. Memory-addressing mechanisms and lexical access. In *Orthography, phonology, morphology, and meaning*, ed. R. Frost and L. Katz, volume 94, 413–434. Amsterdam: North-Holland.

Forster, Kenneth I. 1976. Accessing the mental lexicon. In *New approaches to language mechanisms*, ed. R. J. Wales and E. Walker, volume 30, 257–287. Amsterdam: North-Holland.

Forster, Kenneth I. 1990. Lexical processing. In *Language: An invitation to cognitive science*, ed. Daniel Osherson and Howard Lasnik, 95–131. Cambridge, MA: MIT Press.

Fuchs, Albert. 1971. The saccadic system. *The control of eye movements* 343–362.

Hale, John T. 2014. *Automaton theories of human sentence comprehension*. Stanford: CSLI Publications.

Hart, Betty, and Todd R Risley. 1995. *Meaningful differences in the everyday experience of young american children.*. Baltimore: Paul H Brookes Publishing.

Howes, Davis H, and Richard L Solomon. 1951. Visual duration threshold as a function of word-probability. *Journal of experimental psychology* 41:401.

Just, Marcel A., and Patricia A. Carpenter. 1980. A theory of reading: From eye fixations to comprehension. *Psychological Review* 87:329–354.

Just, Marcel A., Patricia A. Carpenter, and Jacqueline D. Woolley. 1982. Paradigms and processes in reading comprehension. *Journal of Experimental Psychology: General* 111:228–238.

Lewandowsky, S., and S. Farrell. 2010. *Computational modeling in cognition: Principles and practice*. Thousand Oaks, CA, USA: SAGE Publications.

Lewis, Richard, and Shravan Vasishth. 2005. An activation-based model of sentence processing as skilled memory retrieval. *Cognitive Science* 29:1–45.

Logan, Gordon D. 1990. Repetition priming and automaticity: Common underlying mechanisms? *Cognitive Psychology* 22:1–35.

Meyer, David E, and David E Kieras. 1997. A computational theory of executive cognitive processes and multiple-task performance: Part i. basic mechanisms. *Psychological review* 104:3.

Monsell, Stephen. 1991. The nature and locus of word frequency effects in reading. In *Basic processes in reading: Visual word recognition*, ed. D. Besner and G. W. Humphreys, 148–197. Hillsdale, NJ: Erlbaum.

Murray, Wayne S, and Kenneth I Forster. 2004. Serial mechanisms in lexical access: the rank hypothesis. *Psychological Review* 111:721.

Newell, A. 1990. *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Newell, Alan. 1973. Production systems: Models of control structures. In *Visual information processing*, ed. W.G. Chase et al., 463–526. New York: Academic Press.

Newell, Allen, and Paul S Rosenbloom. 1981. Mechanisms of skill acquisition and the law of practice. In *Cognitive skills and their acquisition*, ed. John R. Anderson, 1–55. Hillsdale, NJ: Erlbaum.

Polanyi, Michael. 1967. *The tacit dimension*. London: Routledge and Kegan Paul.

Poore, Geoffrey M. 2013. Reproducible documents with pythontex. In *Proceedings of the 12th Python in Science Conference*, ed. Stéfan van der Walt, Jarrod Millman, and Katy Huff, 78–84.

Rayner, Keith. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin* 124:372–422.

Reichle, Erik D, Alexander Pollatsek, Donald L Fisher, and Keith Rayner. 1998. Toward a model of eye movement control in reading. *Psychological review* 105:125.

Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the Fourteenth International Conference on Computational Linguistics*. Nantes, France.

Ryle, Gilbert. 1949. *The concept of mind*. London: Hutchinson's University Library.

Salvucci, Dario D. 2001. An integrated model of eye movements and visual encoding. *Cognitive Systems Research* 1:201–220.

Schilling, Hildur EH, Keith Rayner, and James I Chumbley. 1998. Comparing naming, lexical decision, and eye fixation times: Word frequency effects and individual differences. *Memory & Cognition* 26:1270–1281.

Schooler, Lael J., and John R. Anderson. 1997. The role of process in the rational analysis of memory. *Cognitive Psychology* 32:219–250.

Staub, Adrian. 2011. Word recognition and syntactic attachment in reading: Evidence for a staged architecture. *Journal of Experimental Psychology: General* 140:407–433.

Vasishth, Shravan, Sven Bruüssow, Richar L. Lewis, and Heiner Drenhaus. 2008. Processing polarity: How the ungrammatical intrudes on the grammatical. *Cognitive Science* 32:685–712.

Whaley, Charles P. 1978. Word£nonword classification time. *Journal of Verbal Learning and Verbal Behavior* 17:143–154.

Wickelgren, Wayne A. 1972. Trace resistance and the decay of long-term memory. *Journal of Mathematical Psychology* 9:418–455.