Unit 5 Code Description

The experiments for this unit are a little more complex than the previous ones. They use a few new ACT-R commands for creating perceptual information as well as several new commands for interacting with a model more directly through code. First we will describe a detail you may have noticed about chunks being copied in buffers. Then we will discuss the code and models for the tasks in this unit, describe the new commands used in the tasks, and some discussion of some technical issues about passing parameters to ACT-R commands. After that is a section which describes how to add new game configurations to the 1-hit blackjack task along with some suggested alternate games for testing the assignment model. Finally, there's a section at the end which walks through how the parameters were adjusted to fit the data in the siegler task that may be helpful when adjusting parameters in your own models.

A note on chunks in buffers and the :dcnn parameter

When a chunk is set in a buffer the buffer makes a copy of it – it does not hold the original chunk. Typically, that detail does not affect how you work with or inspect a model as it runs, but there is one situation where that may be noticeable. When the name of the chunk in a buffer is used to set the value of a slot in a buffer's chunk that value will not match the name of the original chunk since the buffer's chunk is a copy which requires a unique name. That situation shows up in the fan model from this unit. The slots of the chunk in the **imaginal** buffer are set using chunks from the **retrieval** buffer. If the model retrieves the chunk lawyer, then the chunk in the **imaginal** buffer will look like this after the encode-person production fires and the modification to the **imaginal** buffer occurs:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   ARG1  LAWYER-0
```

The copy of the lawyer chunk, named lawyer-0, which was in the **retrieval** buffer was placed into the arg1 slot.

The lawyer-0 chunk is not modified by the model while it is in the **retrieval** buffer, and the **retrieval** buffer is also cleared by that production. Thus, that lawyer-0 chunk is still a perfect match to the original lawyer chunk in declarative memory when it is cleared from the buffer, and because of that it is merged with the lawyer chunk and both names now refer to the same chunk. As discussed in the previous unit, the :ncnar parameter can be used to have the system normalize such references to make it easier to debug the system. If :ncnar is enabled (not **nil**) then the setting of the :dcnn (dynamic chunk name normalizing) parameter determines when those names are corrected. If :dcnn is set to **t**, which is the default value, then those changes are made immediately while the model runs. Since the fan model leaves the :ncnar and :dcnn parameters at their default values of **t**, the chunk in the **imaginal** buffer will be changed to look like this after the lawyer-0 and lawyer chunks are merged:

```
IMAGINAL: IMAGINAL-CHUNK0
IMAGINAL-CHUNK0
   ARG1  LAWYER
```

If :dcnn were set to **nil**, then the arg1 slot would continue to hold the value lawyer-0 until the model stopped running at which time the chunk would be updated if the :ncnar parameter were not **nil**. The important thing to note is that regardless of which name is shown in the slot, once those chunks have been merged the same chunk is being referenced regardless of which name is shown in the slot.

Often having :dcnn and :ncnar set to **t** makes it easier to debug a model, but sometimes it may be useful to disable the dynamic updating so that one can more directly track a reference to a chunk from a buffer, even if it is eventually merged. For this unit the demonstration models leave :dcnn and :ncnar set to their default values of **t**, thus the slot values are immediately adjusted as the model runs.

Fan Experiment

There are two versions of the fan experiment code and corresponding models included with this unit. One pair, fan and fan-model which are used in the main unit text, uses an experiment window to present and perform the task as you have seen for many of the tasks in the tutorial. The other one runs the model through the task without using the visual interface for input or the motor module for output and produces exactly the same timing results. Instead it puts the input into the slots of the chunk placed into the **goal** buffer before running the model and reads the response from a slot of the chunk in the **goal** buffer upon completion. This is done through the use of commands to access the model's buffers and chunks. Mechanisms like these can be used when the details of the visual/motor systems are not of interest in the task being modeled and an abstraction of the experiment is acceptable for the objectives of the modeling work, but doing so usually involves some additional work in the model as well.

The code which implements the experiment window version of the task is very similar to many of the previous experiments and doesn't need any description here. Instead, we will just look at the fan-sentence (Lisp) and sentence (Python) functions from the code that runs the model in the fan-no-pm.lisp and fan_no_pm.py files, and then describe what was done differently with the model.

Functions to run a trial

```
(defun fan-sentence (person location target term)
  (reset)

  ; disable the production that isn't being used for retrieval

  (case term
    (person (pdisable retrieve-from-location))
    (location (pdisable retrieve-from-person)))

  ; modify the chunk named goal (which will be set in the goal buffer
  ; when the model runs) to set the arg1 and arg2 slots to the probe
  ; items and state slot to test
```

```
  (mod-chunk-fct 'goal (list 'arg1 person 'arg2 location 'state 'test))

  ; run the model recording the time spent running
  ; and get the value from the state slot of the goal buffer representing the
  ; model's response to the task

  (let ((response-time (run 30.0))
        (response (buffer-slot-value 'goal 'state)))

    (list response-time
          (or (and target (string-equal response "k"))
              (and (null target) (string-equal response "d"))))))
```

```
def sentence (person, location, target, term):

    actr.reset()

    # disable the production that isn't being used for retrieval

    if term == 'person':
        actr.pdisable("retrieve-from-location")
    else:
        actr.pdisable("retrieve-from-person")

    # modify the chunk named goal (which will be placed into the goal buffer
    # when the model runs) to set the arg1 and arg2 slots to the probe
    # items and state slot to test

    actr.mod_chunk("goal","arg1",person,"arg2",location,"state","test")

    # run the model recording the time spent running
    # and get the value from the state slot of the goal buffer representing the
    # model's response to the task

    response_time = actr.run(30)[0]
    response = actr.buffer_slot_value("goal","state")

    if target:
        if response.lower() == "'k'".lower():
            return (response_time ,True)
        else:
            return (response_time ,False)
    else:
        if response.lower() == "'d'".lower():
            return (response_time ,True)
        else:
            return (response_time ,False)
```

The parts of those functions which show new mechanisms being used are briefly described with the comments before them, but we will describe that in a little more detail here.

The first new thing we see is the use of the pdisable command in ACT-R. This command can be used to disable productions in the model (more details below). In both this version and the other version of the fan experiment it is used to disable one of the response productions, which was done to simplify the data collection as described in the main unit text.

The next new command used is mod-chunk/mod_chunk. This can be used to modify the contents of a chunk. It requires the name of the chunk, and then any number of pairs of slot names and values. The indicated chunk has all of the slots specified set to the provided values. That is done here to avoid having to include productions in the model for processing the visual scene to get the items – they are provided directly to the model.

After that, we see something that hasn't been done previously in the tutorial with respect to the run command. Here we are actually recording the first return value from calling it which is the amount of time that the model spent running. That is being used to determine the time it takes the model to respond to the task instead of recording the time that a key is pressed. That can be simpler, but does require that the model stops completely at the appropriate time – if there were any additional actions to execute after that point the time would not be correct. That is sometimes difficult to actually achieve in the model because of ongoing actions like a retrieval failure or motor action completion.

The final new operation in those functions is the use of buffer-slot-value/buffer_slot_value to get the model's response. That command takes the name of a buffer and the name of a slot and returns the value of that slot from the chunk in the indicated buffer. That is how this model and task avoid having to use the motor module to press a key to record a response.

Fan-no-pm Model

The model for the task which does not use the perceptual and motor modules is very similar to the one described in the unit. It goes through the same steps of encoding the person and location and then retrieving the item, but it does not have to read them from the display nor perform a key press to respond. Because those perceptual and motor actions take time, a model that does not perform them will be faster at the task than one that does. To still fit the human performance on the task this model adjusts the time it takes to fire the productions from the default time of 50ms to account for that difference. That is done using the spp command which was used in unit 3 to adjust the starting utility of the productions. These settings at the end of the model file adjust the :at value (action time) of the productions which controls how long they take to fire:

```
(spp mismatch-location-no :at .21)
(spp mismatch-person-no :at .21)
(spp respond-yes :at .21)
(spp start :at .250)
(spp harvest-person :at .285)
```

Those settings are free parameters in the model, and they are not unique – there are many ways to set them to get the same results and it could have all been attributed to a single production. In fitting this data one also has to adjusted the latency factor and maximum associative strength values which control the timing and activation of the underlying chunks. Using the perceptual and motor modules to perform the task gives the model a reasonable starting point for human performance and saves having to estimate the additional action time along with the declarative memory performance, but it may involve more work to setup the task and the model. Something else to note is that we kept the two versions of this model similar for comparison purposes, but the one that does not use the perceptual and motor components could have been made even simpler by skipping the encoding steps and just performing the retrieval after placing the appropriate values into the slots and then testing the result in the code as well instead of with

productions. There is no single best approach for creating the model and task, and you will have to consider the options and their trade-offs with respect to the objectives of the modeling effort.

Grouped Task

The grouped model is really just a demonstration of partial matching. The experiment code is only there to collect and display the key presses of the model. It is not an interactive experiment which a person can perform nor does it have a direct comparison to any existing data. The experiment code is very simple, and does not use any new commands. However, the model which interacts with that code does use a new production action. Unlike previous tasks where the data collection was done through monitoring the model's output actions, here we create a simple function for collecting the data directly which just records the values provided onto a response list. That function is then called directly from the actions of a production.

Calling ACT-R commands from productions

It is possible to call a command which is available in ACT-R from within a production, and that is how this model provides its responses – it directly calls the grouped-response command that is added by the experiment code. That is done in the harvest-first-item, harvest-second-item, and harvest-third-item productions. Here is the harvest-first-item production.

```
(p harvest-first-item
   =goal>
      isa      recall-list
      element  first
      group    =group
   =retrieval>
      isa      item
      name     =name
  ==>
   =goal>
      element  second
   +retrieval>
      isa      item
      group    =group
      position second
      :recently-retrieved nil
   !eval! ("grouped-response" =name))
```

The new operation shown in that production is !eval!. [The '!' is called bang in Lisp, so that's pronounced bang-eval-bang.] It can be placed on either the LHS or RHS of a production and must be followed by a call to an ACT-R command using Lisp style syntax specifying the string which names the command followed by any parameters to pass to it inside of parentheses. [Note: a valid Lisp expression can also be provided for a !eval! operation instead of specifying an ACT-R command name string.]

On the RHS of a production all the !eval! operation does is evaluate the expression provided – when the production fires, the !eval! is just another action that occurs. If that expression contains variables from the production the current bindings of those variables in the instantiation of the production are the values which will be passed to the command.

On the LHS of a production a !eval! specifies a condition that must be met before the production can be selected just like all of the other items on the LHS. The value returned by the evaluation of a LHS !eval! must be true for the production to be selected (where true is anything that is not nil in Lisp and if the command is implemented in some other language the values must not be the equivalent of nil e.g. False and None in Python are equivalent to nil in Lisp). Because it will be called during the selection process, a LHS !eval! is likely to be evaluated very often and may be called even when the production that it is in is not the one that will be eventually selected and fired.

Using !eval! can be a powerful tool, but it can easily be abused. Using it to call commands as an abstraction for an aspect of the model which is not necessary to model in detail for a particular task is the recommended use. In general, the predictions of the model should not depend heavily on the use of !eval!, otherwise there is not really any point to using ACT-R to create a model – you might as well just generate some functions to produce the data you want.

In this model !eval! is used to collect the model's responses without needing the overhead of creating an experiment with which to interact. Because this task is not presenting information to the model or concerned with the response time there is not really a need for an interactive experiment and !eval! provides an easy alternative. For the assignment tasks in the tutorial however you should **not** be using !eval! in **any** of the productions which you write.

Siegler

The **Siegler** task is only available for a model to perform because there is no display to see and it records the model's speech as a response. There is only one new function used in the code, new-digit-sound, which will be described later with the new commands. It also uses install-device to install a microphone device for the speech module instead of installing an experiment window (which would automatically install a microphone device as was used in the subitizing task in unit3).

1-hit Blackjack

The 1-hit blackjack task has a lot of code to go with it to play the game, control the opponent, analyze the results, and allow a person to play against the model, and this code is flexible in that it allows for the game and opponent to be changed without having to change the existing code. It runs the model through the task by modifying the **goal** buffer (like the fan experiment described above) and uses a couple of new commands to do so, but it does use an experiment window to display the data in a graph and when it plays against a person (so the person can see the cards). There is also a slight difference with respect to when the model is loaded relative to the code in the task.

Similarity hook function

Unlike the previous tasks in the tutorial, this one starts by defining a function and adding a new command before loading the corresponding model. That function computes the similarities between numbers for the model. It must be added as a command before loading the model because the model's :sim-hook parameter setting specifies that "1hit-bj-number-sims" command as the value, and that is not valid if the command does not exist. That command will be called whenever a similarity is required between two items (unless the :cache-sim-hook-results

parameter is set to t in which case it is only called once for a given pair of items), and if it returns a number that will be used as the similarity between them. For this task the function associated with "1hit-bj-number-sims" will test whether the values it is passed are numbers and if they both are it will return a similarity based on the equation:

$$Similarity(a,b) = -\frac{abs(a-b)}{\max(a,b)}$$

Drawing the graph

The graph of the results is shown in an experiment window. It is drawn using the add-text-to-exp-window/add_text_to_exp_window commands which have been used in previous tasks and add-line-to-exp-window/add_line_to_exp_window which will be described below. This shows that the experiment windows are not only usable for presenting information to the model in a task, but can also be used for displaying information to the modeler as well.

Providing the model information

To provide the information to the model the chunk in the **goal** buffer is modified with the current game data if there is a chunk in the buffer. If there isn't a chunk in the **goal** buffer then it must create a new chunk and set that chunk in the buffer. To test whether there is a chunk in the buffer the buffer-read/buffer_read function is used which returns the name of the chunk in a buffer if there is one. If there is, then the mod-focus/mod_focus command can be used to modify it (this is a command specific to the **goal** buffer). If there isn't then a new chunk needs to be created, and as a general principle, we do not want that chunk to be in declarative memory yet since the model hasn't seen it before. So, we use the define-chunks command to create it which creates chunks without placing them into declarative memory (instead of add-dm which creates chunks and places them into declarative memory). Once we have the new chunk we can use the goal-focus/goal_focus command to put it into the goal buffer (a similar process that is not specific to the **goal** buffer will be shown in a future unit). Details on those new commands will be provided later.

1hit-blackjack-model

In addition to the new parameters set in the model definition, which were described in the main unit text, there are some new commands used in the model definition. The first is this one:

```
(declare-buffer-usage goal game-state :all)
```

That command is there to avoid the style warnings from the procedural module because the **goal** buffer is being tested for chunks which are not generated by the model itself. The declare-buffer-usage command is a way for the modeler to tell the procedural module that a buffer will be using slots other than those which are set directly in the model. It takes two required parameters which are the name of a buffer and the name of a chunk-type defined in the model. It takes an arbitrary

number of additional parameters which indicate the slots from that chunk-type which will be set from outside of the model, or as is used here, the symbol :all to indicate that all of the slots from that chunk-type may be set externally.

Doing so avoids the style warnings like these which would normally be printed when slots that are not set in the model are used in productions:

```
#|Warning: Productions test the MRESULT slot in the GOAL buffer which is not
requested or modified in any productions. |#
#|Warning: Productions test the MC1 slot in the GOAL buffer which is not requested
or modified in any productions. |#
```

Alternatively, instead of using declare-buffer-usage to specify the details of slots used in a buffer from outside of the model itself we could just turn off the warnings, but that is not recommended because with the warnings off one may miss other serious problems.

The next new command used is this one:

```
(define-chunks win lose bust retrieving start results)
```

That command is also used in the code which implements the game and will be described in detail below. The reason this exists in the model is to create the chunks for the items that are used in the productions to indicate the game information and the model's internal state. That avoids the warnings for undefined chunks as was shown in the semantic model of unit 1, and because those chunks do not need to have any slots we can simply specify names to create them.

The last new command is actually one that we have used in many of the previous experiments, but this time we have included it in the model in a slightly different form:

```
(install-device '("motor" "keyboard"))
```

Previously we have seen install-device used to have the model interact with a window in a task. It is more general than that however, and can be used to install any ACT-R device which has been created for a model to use. In this case, we are installing the keyboard device for the motor module to use. That has not been done previously in the experiments because the experiment windows automatically install a keyboard and mouse for the motor module, the window for the vision module, and a microphone for the speech module, but since this task does not have a visual interface for the model we are not installing an experiment window for the model to use. A similar usage occurs in the siegler task of this unit to install the microphone device for the speech module to record the model's vocal responses since it too does not use an experiment window.

Command Information

There were several new commands used in the tasks and models of this unit. However, before describing those commands we will first discuss some details about working with ACT-R from code.

Names

The code which implements ACT-R is written in Lisp, and in Lisp one of the fundamental data types is called a symbol. Very roughly speaking any sequence of alpha-numeric characters which is not purely numeric represents a symbol. They can be used for a variety of purposes, and a convenient use of symbols is to name things. Most of what you see in the model files are Lisp symbols e.g. everything in the count.lisp file for unit 1 except for the parentheses and numbers are symbols. The important issue is that internally all of the names of things in ACT-R are symbols – chunks, productions, parameters, slots, buffers, modules, etc. A Lisp symbol is different from a Lisp string which is specified in double quotes e.g. "goal" (like strings in most other languages) and that distinction can be an important factor when creating the chunks for a model (as we will discuss later with respect to the fan model from this unit). However, most other languages do not provide a construct like a symbol and the communication protocol used to connect to ACT-R also does not provide such a default type. Therefore, to accommodate interacting with other systems the ACT-R commands evaluated through the dispatcher accept strings as the names of items and convert strings to symbols to get the names, and similarly, names are converted to strings when returning them through the dispatcher. If you have been looking at the Python versions of the tasks and interface functions from other units you will have seen strings being passed to the functions and strings being returned, with the buffer-chunk function shown in unit2 as a good example where the Lisp version used symbols and the Python version used strings:

```
? (buffer-chunk goal visual)
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   ARG1  FIVE
   ARG2  TWO
   SUM  SEVEN

VISUAL: NIL
(GOAL-CHUNK0 NIL)
```

```
>>> actr.buffer_chunk('goal','visual')
GOAL: GOAL-CHUNK0
GOAL-CHUNK0
   ARG1  FIVE
   ARG2  TWO
   SUM  SEVEN

VISUAL: NIL
['GOAL-CHUNK0', None]
```

Strings in slots

Generally, using strings to name things does not cause an issue, but there is one place where a little extra effort is required when using commands through the dispatcher. That situation is when one wants to put a string value into a chunk's slot or get the value of a slot which contains a string, as is done in the version of the fan experiment that does not use the perceptual and motor modules. Since the assumption is that strings from the external interface should be converted to symbols to specify names, to specify that something should be a string requires additional effort. To indicate a value is a string instead of a name one needs to add a set of single quotes around the item in the string e.g. "'value'". Similarly, when returning a value which should be considered as a string it will contain single quotes around the item. Having to work with strings in the slots of

chunks using commands through the dispatcher is probably not the sort of thing one will need to do very often, but when needed, some extra care will be required.

Additional Lisp command information

For many of the ACT-R commands the Lisp version provides both a macro and a function for accessing the command. The biggest distinction from the user's perspective is that when using a function in Lisp the parameters are evaluated first whereas a macro does not evaluate its parameters. Many of the ACT-R commands you've seen are actually macros e.g. chunk-type, add-dm, and p. They are macros so that you don't have to worry about quoting symbols or lists and other issues with Lisp syntax, but because they don't evaluate their parameters there are some things that you can't do with the macros (at least not without using an explicit call to eval and/or backquoting but those are Lisp programming techniques which will not be described here). For example, if you have a variable called *number* and you'd like to create a chunk that has the value of *number* in one of its slots. Using the add-dm macro will not work as shown here using the dm command discussed in unit1 for printing a chunk from declarative memory:

```
? (defvar *number* 3)
*NUMBER*
? (add-dm (foo size *number*))
(FOO)
? (dm foo)
FOO
   SIZE  *NUMBER*
```

The add-dm macro doesn't evaluate the variable *number* to get its value. Instead it creates a chunk that literally contains *number* in its slot.

To allow modelers to do things like that, most of the ACT-R macros have a corresponding function that does the same thing and the naming convention in the ACT-R interface is to add "-fct" to the end of the name for the functional form of a command. When using the functional form of an ACT-R command you have to pay more attention to Lisp syntax and make sure that symbols are quoted and lists are constructed appropriately, and often the required parameters are a little different – things that do not need to be in a list for the macro version need to be in a list for the function. For example, add-dm-fct requires a list of lists as its only parameter instead of an arbitrary number of lists. Here is the code that would generate the chunk as desired above:

```
? (add-dm-fct (list (list 'foo 'size *number*)))
(FOO)
? (dm foo)
FOO
   SIZE  3
```

Not all of the ACT-R commands have both a macro and function available from Lisp, but for those that do we will provide the details from this point on when describing the new commands.

New Commands

Below we will describe the new commands used in this unit.

pdisable – This command can be used to disable productions. The Lisp macro and the Python function take any number of production names as parameters. The Lisp function **pdisable-fct** requires a list of production names as its only parameter. The named productions will be disabled in the current model. A disabled production cannot be selected during the conflict resolution process. Disabled productions will be enabled again if the model is reset or if explicitly enabled using the corresponding **penable** command. It returns a list with the names of all the productions which have been disabled in the current model.

define-chunks – Define-chunks is similar to add-dm which has been used in all of the previous models to create chunks and add them to the model's declarative memory. The difference between define-chunks and add-dm is that define-chunks creates the chunks specified but does not add them to the model's declarative memory. Keeping unnecessary information out of the model's declarative memory can be useful in multiple situations. One reason is because the spreading activation depends on the fan of items and that fan is based on the contents of the chunks in declarative memory. So, putting chunks which do not represent the actual knowledge of the model into declarative memory could affect the activation of the important chunks by affecting the fan values. It is also important when creating chunks for a buffer which might later need to be retrieved after the model has manipulated them e.g. if the 1hit-blackjack model were retrieving the game-state chunks to make its decision we would not want the starting goal chunk placed into declarative memory prior to the model actually playing that game. Finally, it can also make things easier on the modeler when inspecting declarative memory while working with the model because it can be easier to find the relevant information if there are not a lot of irrelevant chunks there as well.

The Lisp macro (**define-chunks**) and the Python function (**define_chunks**) take any number of lists of chunk descriptions or names for chunks which will have no slots whereas the Lisp function (**define-chunks-fct**) requires a list of chunk description lists or names for chunks with no slots. They return a list of the names of the chunks that were created.

goal-focus – We have seen goal-focus used in models throughout the tutorial to schedule an action to place a chunk into the goal buffer. Here we are calling it from code to do the same thing. All versions, the Lisp macro, Python function (**goal_focus**), and the Lisp function (**goal-focus-fct**) take one optional parameter which names a chunk to place into the goal buffer. If the parameter is not specified then the command will print out the chunk in the goal buffer. It returns the name of the chunk which will be placed into the goal buffer or the chunk that is already there if no parameter is provided and no previous goal-focus action remains unexecuted.

mod-chunk - This command is used to modify a chunk. The Lisp macro version and the Python function (**mod_chunk**) require a chunk name and then an even number of additional parameters which indicate slots and values whereas the Lisp function (**mod-chunk-fct**) requires a chunk name and then a list of slots and values. Each of the slots specified for the chunk is given the corresponding value. It returns the name of the chunk which was modified.

One important thing to note is that once a chunk enters declarative memory it cannot be modified. That is another reason why one may want to use define-chunks instead of add-dm.

mod-focus – This command is very similar to **mod-chunk** except that it does not require the name of a chunk, only the slots and values. It schedules those changes to be made at the current

time to the chunk which is in the goal buffer, and it returns the name of the chunk which is in the goal buffer (or the chunk which is scheduled to enter the goal buffer if a goal-focus command has scheduled one to be put there at the current time as well).

buffer-slot-value – This command returns the value from a slot of a chunk in a buffer. The parameters for both the Lisp function and Python function (**buffer_slot_value**) are the name of the buffer and the name of the slot. It returns the value of that slot from the chunk in the indicated buffer or nil (Lisp) or None (Python) if the chunk does not have the specified slot.

new-digit-sound – This command creates a new sound stimulus for the model to provide numeric information and is similar to the new-tone-sound command which was used in unit 3 to present a tone. Both the Lisp function (**new-digit-sound**) and the Python function (**new_digit_sound**) require one parameter which is the number to present and also take two optional parameters. The first optional parameter can be provided to specify the time at which the digit should be presented (the current time will be used if a specific onset time is not given). The second optional parameter can be specified as a true value to indicate that the time is specified in milliseconds instead of the default units of seconds.

add-line-to-exp-window – this is similar to the Lisp function **add-text-to-exp-window** and the Python function **add_text_to_exp_window** which have been used in previous units. This function draws a line in an experiment window. It takes three required parameters and one optional parameter. The first required parameter is the window in which to draw the line. The other two are each a list of two integers which indicate the pixel coordinates of the end points of the line to be drawn (given as x and then y). The optional parameter can be provided to indicate the color of the line, and the default is black if it is not provided.

Modifying 1-hit blackjack

This section will discuss how to change the decks and/or the opponent for the 1-hit blackjack game as well as provide some suggestions for some other opponents to test a model against.

To make the game flexible the code relies on functions being specified to handle the three changeable components of the game: the model's deck of cards, the opponent's deck of cards, and the code to determine whether or not the opponent will hit or stay. The functions are stored in global variables which are then called when needed. Thus, writing new functions for those parts of the game and setting the corresponding variables to those functions will change the way the game plays. To help make that more manageable, a function can be written to set all of the appropriate values for a particular game scenario and that function can be passed to the onehit-learning or learning function as the (optional) third parameter. That setup function will be called at the start of each of the rounds that is played. Thus, to play 5 rounds of a game specified by a function named game1 and display the graph of the model's results these would be how that is called for the Lisp and Python versions (assuming the game1 function was also defined in the onehit.py file for the Python version):

```
? (onehit-learning 5 t 'game1)
```

```
>>> onehit.learning(5,True,onehit.game1)
```

The functions for the decks are held in the variables *deck1* and *deck2* for Lisp and deck1 and deck2 for Python. Deck1 holds the deck for the model's cards and deck2 the opponent's cards. A deck function should return a number from 1-10 representing the value of the card being dealt. On every hand each of the deck functions will be called three times. The first call will be for the face up card's value, the second call will be for the player's hidden card and the third call will be for the card that the player will receive on a hit. All three cards are dealt at the start of the hand, but only shown to the players when appropriate. The model's cards are dealt before the opponent's cards. Thus, if the same deck function is used for both players, as it is for the example games and the suggested alternatives, then the function will be called 6 times per hand with the first three calls returning the model's cards and the second set of three being for the opponent's cards.

The function for the model's opponent is called to determine if the opponent will hit or stay. The opponent function is stored in the variable *opponent-rule* for Lisp and opponent_rule in Python. It will be passed two parameters. The first parameter is a list of the opponent's two starting cards. The second parameter is the number of the model's face up card. That function should return either the string "h" if the opponent will hit the hand or the string "s" if the opponent will stay for that hand. Because we are only using a fixed opponent for the model there is no function called to provide feedback on the outcome of the hand to the opponent i.e. it has no way to learn about the game or the model, but the code does provide a variable for that (*opponent-feedback* or opponent_feedback) if one wanted to create a learning opponent to play against.

There is a given function called fixed-threshold or fixed_threshold which can be used for creating an opponent with a fixed value which indicates the hand value at which it will always stay. The

value at which an opponent with that function will stay is set with the variable *opponent-threshold* or opponent_threshold.

All of these variables are set to create the default game scenario as can be seen in the game0 functions in the code.

If you want to create an opponent that needs to know the value of a hand, the score-cards and score_cards functions can be used to compute the value for a list of card values taking into account the rule of a 1 being counted as 11 when possible.

By writing a different game function to set the control variables one can easily modify the game that is played by the model. The simplest change would be to just adjust the threshold at which the default opponent stays. That could be done by adding a new game function to set the variables appropriately and then passing that function to the onehit-learning or learning function. A function like these would change the opponent to stay when it has a score of 12 or more instead:

```
(defun newgame ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-rule* 'fixed-threshold)
  (setf *opponent-threshold* 12)
  (setf *opponent-feedback* nil))
```

```
def newgame():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback

    deck1 = regular_deck
    deck2 = regular_deck
    opponent_rule = fixed_threshold
    opponent_threshold = 12
    opponent_feedback = None
```

Then to run that game we would pass that newgame function to the onehit-learning or learning function like this:

```
? (onehit-learning 5 t 'newgame)
```

```
>>> onehit.learning(5,True,onehit.newgame)
```

There is a second game already programmed and available in the given code. It is called game1 and its setup function serves as another example for reference.

There are some other game scenarios which we have designed to test the model's ability to learn, but which have not been included in the code. These can be implemented as an additional exercise if you would like, and of course, you are also free to create game scenarios of your own design for testing. The testing scenarios we outline here all assume the same deck function will be used for both the model and the opponent to keep things simpler, but that is not necessary since the two variables can be specified separately.

Game 2: In this game the deck consists of only cards numbered 7 and the opponent will always stay. In this game the model will always win if it hits and it should be able to learn that fairly quickly.

Game 3: The deck consists of an essentially infinite number of cards with only the values 8, 9, and 10 in equal proportions and again the opponent will always stay. The model should also learn to always stay because it will always lose if it hits. Staying on every hand will result in winning about 38.9% of the games.

Game 4: The deck consists of the cards 2, 4, 6, 8, and 10 being cycled in that order repeatedly. Thus there are only 5 possible hand combinations which will be cycled through in order:

Model's cards	Opponent's cards
2 4 6	8 10 2
4 6 8	10 2 4
6 8 10	2 4 6
8 10 2	4 6 8
10 2 4	6 8 10

The opponent for this game is one which always hits. In this game the model should be able to learn the correct move to win the 4 situations which can be won out of the 5 possible hands (80% wins by the end).

Game 5: The deck consists of only the following possible triples each equally likely in any deal to either player: (2 10 10) (4 9 9) (6 8 8) (8 8 5) (9 9 3) (10 10 1). These hands are designed so that if the player's initial score is small (12, 13 or 14) then a hit will always bust and if the initial score is large (16, 18, or 20) then a hit will always score 21 total. The opponent for this game will randomly hit or stay with equal probability. The optimal strategy for this game will result in winning about 54% of the hands.

Fitting the data for the siegler task

This section is going to walk through how the parameter values were chosen to fit the data in the siegler task.

Parameters to be adjusted for siegler task

To adjust the fit to the data we will be using partial matching and adjusting the base-level activations of the plus-facts for the model. The specific parameters that we will need to adjust for the model are those related to activation in general (the retrieval threshold, :rt, and the activation noise, :ans), those related to partial matching (the similarities among the number chunks used as the addends of the plus-facts and the match scale value, :mp), and the base-level activation values of the chunks.

That is potentially a lot of free parameters in the model. Treating them that way one could likely produce an extremely strong fit to the reported data. However, doing that is not very practical nor does it result in a model that is of much use for demonstrating anything other than the ability to fit 60 data points using more than 60 parameters.

In the following sections we will describe the effects that the particular parameters have on the model's performance and outline an approach which can be taken to arrive at the parameter settings in a model.

Initial siegler model

The first thing to do for the model is make sure that it can do the task. In this case that is hear the numbers, attempt to retrieve an addition fact, and then speak the result. To do that, we will start without enabling the subsymbolic components of the system. Making sure the model works right with basic symbolic information is a good start for modeling complex tasks because once the subsymbolic components are enabled and more sources of randomness or indeterminate behavior are introduced it can be very difficult to find potential errors in the productions or basic logic of the model.

The assumptions for the model are that the children know the numbers from zero through nine and that they have encountered the addition facts for problems with addends from zero to five. Thus these will be the declarative memory elements with which the model will start. Along with that, we are assuming that the children are not going to use any complex problem solving to try to remember the answer. So, if there is a failure to remember a fact after one try the model will just give-up and answer that it does not know. For a task of this nature where we are modeling the aggregate data, using a single idealized strategy for the model is often a reasonable approach, and has been how all of the other models seen so far in the tutorial operate. In other circumstances, particularly when individual participant data is being modeled, the specific strategy used to perform the task may be important, and in those cases it may be necessary to include different strategies into the model to account for the data.

With the model working correctly in a purely symbolic fashion we should see it answering correctly on every trial and here are the results of the model in that case providing the starting point for the adjustments to be made:

```
CORRELATION:  0.943
MEAN DEVIATION:  0.127
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
1+2  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00  0.00
1+3  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00
2+2  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00  0.00
2+3  0.00  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00  0.00
3+3  0.00  0.00  0.00  0.00  0.00  0.00  1.00  0.00  0.00  0.00
```

Making errors

Now that we have a model which performs perfectly we need to consider how we want it to model the errors. For this task we have chosen to use partial matching to do that. Specifically, we want the model to retrieve an incorrect addition fact as it does the task and also to sometimes fail to retrieve an addition fact at all (an important source of the "other" results for the model). What we do not want it to do is retrieve an incorrect number chunk or fail to retrieve one while encoding the audio input or producing the vocal output. The reason for that is because we are assuming that the children know their numbers and thus do not produce errors because they are failing to understand what they hear or failing to say an answer correctly. That is important because we are not just looking to have the model fit the data but to actually have it do so in a manner which seems plausible for the task.

To make those errors through partial matching requires that the model occasionally retrieve the wrong chunk for the critical request of a plus-fact chunk like this one based on the addend1 and addend2 slots:

```
(f13 ISA plus-fact addend1 one addend2 three sum four)
```

Thus, the items which will need to be similar to result in mismatches are the number chunks which are used as addends, and that is where we will start in setting the parameters.

Setting similarities

The similarity settings between the number chunks will affect the distribution of incorrect retrievals. While this looks like a lot of free parameters to be fit, in practice that is just not reasonable. For a situation like this, where the chunks represent numbers, it is better to set the similarity between two numbers based on the numerical difference between them using a single formula to specify all of the similarities. There is a lot of research into how people rate the similarity of numbers and there are many equations which have been proposed to describe it. For this task, we are going to use a linear function of the difference between the numbers.

Also, to keep things simple we will use the default range of similarity values for the model, which are from 0.0 for most similar to -1.0 for most dissimilar. Since we are working with numbers from 0-9 an obvious choice for setting them seems to be:

$$Similarity(a,b) = -(0.1 * |a-b|)$$

To set those similarities, we need to use the set-similarities command. Because the similarities are symmetric we only need to set each pair of numbers once and we do not need to set the similarity between a chunk and itself because that defaults to the most similar value. We also note that since the model only has chunks for encoding the facts with addends from 0-5 we only need to set the similarities for the chunks which are relevant to the task. Thus, here are the initial similarity values set in the model:

```
(Set-similarities
  (zero one -0.1) (one two -0.1) (two three -0.1) (three four -0.1) (four five -0.1)
  (zero two -0.2)(one three -0.2)(two four -0.2)(three five -0.2)
  (zero three -0.3)(one four -0.3) (two five -0.3)
  (zero four -0.4)(one five -0.4)
  (zero five -0.5))
```

In addition to the similarities, we will also need to set the match scale parameter for the model. Adjusting the match scale will determine how much the similarity values affect the activation of the chunks since it is used to multiply the similarity values. Because we have chosen a linear scale for our similarity values we will actually be able to just use the match scale parameter to handle all of our adjustments instead of needing to adjust the available range or the parameter we chose in our similarity equation.

The similarity value and the match scale are going to determine how close the activations between the correct and incorrect chunks are. How large that needs to be to create the effect we want is going to depend on other settings in the model. Thus, there is not really a good guideline for determining where it should be initially, but from experience we know that it is often easier to adjust the parameters later if we start with values that allow us to see the effect each has on the results. Therefore we want to make sure that we pick a value here which ensures that the similarity will make a difference in the activation values. Since the default base-level activation of chunks is 0.0 when the learning is off we are going to choose a large initial :mp value, like 5, to make sure that the activations will differ noticeably.

With just these settings however, the model will still not make any errors because the correct chunk will always have the highest activation and be the one retrieved. To actually get some errors we will need to also add some noise to the activation values.

Activation noise

In the previous unit, we saw how the activation noise affects the probability that a chunk will be above the retrieval threshold. Now, since there are multiple chunks which could all be above the threshold, it is also going to affect the frequency of retrieving the correct chunk among the incorrect alternatives. The more noise there is the less likely it is that the correct chunk will have the highest activation.

As with the :mp value, choosing the initial value for the noise is not obvious because its effect is determined by other settings in the model. For this parameter however, we do have some general guidelines to work with based on past experience. For many models that have been created in the past an activation noise value in the range of 0.0-1.0 has been a good setting and for most of those the value tends to fall somewhere between 0.2 and 0.5. So, based on that, we will start this

model off with a value of .5, as was used for the models of the previous unit, and then adjust things from there if needed later.

Now, given these settings, :ans .5 and :mp 5 with the similarities set as shown above, we can run the model and see what happens. Here is what we see if we just run it to collect the data:

```
CORRELATION: -0.034
MEAN DEVIATION:  0.339
       0     1     2     3     4     5     6     7     8   Other
1+1  0.01  0.00  0.04  0.04  0.03  0.00  0.00  0.00  0.00  0.88
1+2  0.00  0.01  0.03  0.03  0.02  0.02  0.00  0.01  0.00  0.88
1+3  0.00  0.01  0.02  0.04  0.04  0.01  0.02  0.00  0.00  0.86
2+2  0.00  0.00  0.01  0.03  0.02  0.02  0.00  0.00  0.00  0.92
2+3  0.00  0.01  0.00  0.00  0.02  0.02  0.01  0.02  0.00  0.92
3+3  0.00  0.00  0.00  0.00  0.03  0.01  0.05  0.02  0.02  0.87
```

The model is almost never correct and most of the errors are in the other category which means that it probably did not respond. The important thing to do next is to understand why that is happening. One should not just start adjusting the parameters to try to improve the fit without understanding why the model is performing in that way.

Retrieval threshold and base-levels

Running the model on a few single trials and stepping through its operations shows that the problem is happening because the model is failing to retrieve chunks during all of the retrieval requests, including the initial encoding of the numbers. We want the model to sometimes fail on the retrieval of the addition fact chunks, but we do not want it to be failing during the encoding steps.

To fix that, there are two changes which we will make at this point. The first is to adjust the retrieval threshold so that we eliminate most, if not all, of the retrieval failures. This will allow us to work on setting the other parameters to match the data with the model answering the questions. Then we can come back to the retrieval threshold later and increase it to introduce more of the non-answer responses into the model. Thus, for now we will set the retrieval threshold to a value of -10.0 to make it very unlikely that any chunk will have an activation below the threshold.

The other thing we will do at this time is consider how to keep the number chunks from failing once we bring the retrieval threshold back up to a reasonable value. The easiest way to handle that is to increase the base-level activation of the number chunks so that the noise will be unlikely to ever take them below the retrieval threshold. The justification for doing so in the model is that it is assumed the children have a strong knowledge of the numbers and do not confuse or forget them and thus we need to provide the model with a comparable ability.

To do that we will use the set-base-levels command which works similar to the set-all-base-levels command that was used in the last unit. The difference is that for set-base-levels we can specify specific chunks instead of applying the change to all of them. Again, this seems like it is a lot of free parameters, but since we are not measuring the response time in this model all that matters is that the chunks have a value large enough to not fail to be retrieved – differences among them will not affect the error rate results as long as they are all being retrieved. We will start by assigning

them a value of 10 which is significantly larger than the retrieval threshold we have now of -10 which should result in no failures for retrieving number chunks. When we increase the retrieval threshold later we may need to adjust this value, but for now we will add these settings to the model:

```
(set-base-levels
  (zero 10) (one 10) (two 10) (three 10) (four 10) (five 10)
  (six 10) (seven 10) (eight 10) (nine 10))
```

Unlike the similarities where we only needed to set the values for the numbers from 0-5 based on the task, here we need to set all of the numbers from 0-9 since any of those values is a potential sum of an addition fact in the model's declarative memory which may need to be retrieved.

After making those additions to the model, running it produces this output:

```
CORRELATION:  0.542
MEAN DEVIATION:  0.174
       0     1     2     3     4     5     6     7     8   Other
1+1  0.04  0.14  0.28  0.26  0.15  0.07  0.04  0.01  0.00  0.01
1+2  0.00  0.09  0.23  0.20  0.17  0.17  0.09  0.05  0.00  0.00
1+3  0.01  0.05  0.11  0.17  0.25  0.21  0.15  0.04  0.01  0.00
2+2  0.00  0.01  0.05  0.28  0.22  0.24  0.13  0.06  0.01  0.00
2+3  0.01  0.02  0.04  0.16  0.22  0.22  0.14  0.14  0.04  0.01
3+3  0.00  0.01  0.01  0.03  0.15  0.16  0.26  0.25  0.09  0.04
```

That shows a better fit to the data than the last one, though still not as good as we want, or in fact as good as it was when perfect. Looking at the trace of a few individual runs seems to indicate that the model is working as we would expect – the errors are only due to retrieving the wrong addition fact because of partial matching.

Adjusting the parameters

The next step to take depends on what the objectives of the modeling task are – what are you trying to accomplish with fitting the model to the data and what do you consider as a sufficient fit to the data. If that fit to the data is good enough, then as a next step you would then want to start bringing the retrieval threshold up to introduce more of the "other" responses (failure to respond) and hopefully improve things a little more. In this case however we are not going to consider that sufficient and will first investigate other settings for the :ans and :mp parameters before moving on to adjusting the retrieval threshold.

To do that we are going to search across those parameters for values which improve the model's fit to the data. When searching for parameters in a model there are a lot of approaches which can be taken. In this case, we are going to keep it simple and try manually adjusting the parameters and running the model to see if we can find some better values. When the number of parameters to search is small, the model runs fairly quickly, and one is not looking to precisely model every point this method can work reasonably well. For other tasks, which require longer runs or which have many more parameters to adjust other means may be required. That can involve writing some code to adjust the parameters and perform a more thorough search or going as far as creating an abstraction of the model based on the underlying equations and using a tool like MATLAB, Mathematica, or a spreadsheet solver to then find the best values.

The approach that we use when searching by hand is to search on only one parameter at a time. Pick one parameter and then adjust that to get a better fit. Then, fix that value and pick another parameter to adjust. Do that until each of the parameters has been adjusted. Often, one pass through each of the parameters will result in a much better fit to the data, but sometimes it may require multiple passes to arrive at the performance level you desire (assuming of course that the model is capable of producing such a fit through manipulating the parameters).

Sometimes it is also helpful to work with only a subset of the parameters if you have an idea of the effects which they will have on the data. For example, in this task we know that the retrieval threshold will primarily determine the frequency with which the model gives up. Thus we are going to hold back on trying to fit that parameter until we have adjusted the others to better fit the majority of the data for the trials where it produces an answer.

Since we are starting with a noise value that was based on other tasks and our match scale value was chosen somewhat arbitrarily we will start searching across the match scale parameter. Keeping the noise value at .5 we found that a value of 16 for the match scale parameter seems to be our best fit:

```
CORRELATION:  0.914
MEAN DEVIATION:  0.086
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.09  0.68  0.21  0.02  0.00  0.00  0.00  0.00  0.00
1+2  0.00  0.00  0.20  0.67  0.11  0.02  0.00  0.00  0.00  0.00
1+3  0.00  0.00  0.00  0.13  0.68  0.17  0.02  0.00  0.00  0.00
2+2  0.00  0.00  0.01  0.16  0.73  0.10  0.00  0.00  0.00  0.00
2+3  0.00  0.00  0.00  0.01  0.11  0.76  0.11  0.01  0.00  0.00
3+3  0.00  0.00  0.00  0.00  0.01  0.09  0.72  0.17  0.01  0.00
```

Then, fixing the match scale parameter at 16 and adjusting the noise value we do not seem to find a value which does any better than the starting value of .5. So, we will adjust the retrieval threshold to introduce more of the other responses and hopefully improve the fit some more. Searching finds that a value of -.7 improves the fit slightly to this:

```
CORRELATION:  0.937
MEAN DEVIATION:  0.073
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.12  0.69  0.07  0.00  0.00  0.00  0.00  0.00  0.12
1+2  0.00  0.00  0.07  0.68  0.09  0.00  0.00  0.00  0.00  0.16
1+3  0.00  0.00  0.00  0.06  0.73  0.07  0.01  0.00  0.00  0.13
2+2  0.00  0.00  0.01  0.04  0.69  0.08  0.02  0.01  0.00  0.15
2+3  0.00  0.00  0.00  0.01  0.09  0.69  0.10  0.00  0.00  0.11
3+3  0.00  0.00  0.00  0.00  0.00  0.09  0.72  0.10  0.01  0.08
```

One important thing to do at this point is to make sure that the model is still doing the task as we expect – that changing the parameters has not introduced some problems, like failing to retrieve the number chunks. For the current model, looking at a couple of single trial runs in detail shows that things are still working as expected. So, at this point we could go back and perform another pass through all the parameters trying to find a better fit, but instead we are going to stop and look at where our model seems to be deviating from the experimental data before trying to just find better parameters.

Adjusting the model

It seems that one trend in the data which we are missing is that the children seem to respond correctly more often to the smaller problems and that when they respond incorrectly the answers are more often smaller than the correct answer. There seems to be a bias for the smaller answers. This agrees with other research which finds that addition facts with smaller addends are encountered more frequently in the world.

Accounting for that component of the data is going to require making some adjustment to the model other than just modifying the parameters which we have. The research which finds that the smaller problems occur more frequently suggests a possible approach to take. The base-level activation of a chunk represents its history of use, and thus by increasing the base-level activation of the smaller plus-fact chunks we can simulate that increase in frequency and increase the probability that the model will retrieve them. This should help improve the data fit in a plausible manner.

Like the similarities, this is another instance where it looks like there are a lot of free parameters that could be used to fit the data, but again a principled approach is advised. In this case we are going to increase the base-level activation of all the small plus facts (which we have chosen to be those with a sum less than or equal to four), and we are going to give all of those chunks the same increase to their base-level activation. The default base-level activation for the plus-facts is 0. So we are going to set those chunks to have a value above that by using the set-base-levels command as we have done with the number chunks like this:

```
(set-base-levels
 (f00 .1)(f01 .1)(f02 .1)(f03 .1)(f04 .1)
 (f10 .1)(f11 .1)(f12 .1)(f13 .1)
 (f20 .1)(f21 .1)(f22 .1)
 (f30 .1)(f31 .1)
 (f40 .1))
```

Using the values for the other parameters found previously we will search for a base-level value which improves the data fit and what we find is that a value of .5 seems to improve things to this:

```
CORRELATION:  0.961
MEAN DEVIATION:  0.060
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.15  0.75  0.08  0.00  0.00  0.00  0.00  0.00  0.02
1+2  0.00  0.00  0.12  0.75  0.11  0.00  0.00  0.00  0.00  0.02
1+3  0.00  0.00  0.00  0.10  0.81  0.04  0.01  0.00  0.00  0.04
2+2  0.00  0.00  0.01  0.11  0.76  0.04  0.01  0.01  0.00  0.06
2+3  0.00  0.00  0.00  0.02  0.23  0.62  0.08  0.00  0.00  0.05
3+3  0.00  0.00  0.00  0.00  0.01  0.09  0.72  0.10  0.01  0.07
```

Given that, we will make one more pass over all the parameters (noise, match scale, retrieval threshold, and small plus-fact base-level offset) to find the final set of parameter values which are set in the given model and produce this fit to the data:

```
CORRELATION:  0.947
MEAN DEVIATION:  0.066
       0     1     2     3     4     5     6     7     8   Other
1+1  0.00  0.09  0.67  0.20  0.02  0.00  0.00  0.00  0.00  0.02
```

1+2	0.00	0.00	0.18	0.64	0.09	0.01	0.00	0.00	0.00	0.08
1+3	0.00	0.00	0.00	0.14	0.78	0.05	0.00	0.00	0.00	0.03
2+2	0.00	0.00	0.01	0.17	0.78	0.03	0.00	0.00	0.00	0.01
2+3	0.00	0.00	0.00	0.00	0.25	0.50	0.05	0.00	0.00	0.20
3+3	0.00	0.00	0.00	0.00	0.01	0.02	0.62	0.14	0.01	0.20

We could continue to search over the parameters or attempt other changes, like modifying the similarities used to something other than linear, but these results are sufficient for this demonstration. You are free to explore other changes to the parameters or to the model if you are interested.