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 the model more directly through code. First we will describe the code for implementing the tasks in this unit. Then we will describe the new commands, and that will include some discussion of some special issues with regard to providing parameter values for ACT-R commands. Finally, there is a section at the end of the document 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.

Fan Experiment

There are two versions of the fan experiment code and model 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 most 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 goal chunk before running the model and reads the response from a slot of the goal chunk 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.

The code which implements the experiment window version of the task is very similar to many of the previous experiments and will not be described here. Instead, we will look at the code in the fan-no-pm.lisp and fan_no_pm.py files which perform the task without using an experiment window, and also look at some of the differences in the corresponding fan-no-pm-model.lisp file.

Lisp

First load the corresponding model file for this version of the task.

```
(load-act-r-model "ACT-R:tutorial;unit5;fan-no-pm-model.lisp")
```

Define a global variable that holds the experimental data to which the model will be compared.

```
(defvar *person-location-data* '(1.11 1.17 1.22
                                 1.17 1.20 1.22
                                 1.15 1.23 1.36
                                 1.20 1.22 1.26
                                 1.25 1.36 1.29
                                 1.26 1.47 1.47))
```

The fan-sentence function takes 4 parameters which are very similar to the ones that were passed to the fan-sentence function described in the main unit text, and for now we will ignore the difference but it will be described later. The first, person, is the string of the person named in the sentence. The second, location, is the string of the location named in the sentence. The third, target, is either t or nil to indicate whether this is a target or a foil trial respectively, and the fourth, term, specifies which of the productions to favor in retrieving the study item and should be either the symbol person or location. The function returns a list of two items. The first is the time the model spends running in seconds and the second item is t if the response was correct or nil if there was no response or the response was incorrect.

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

Start by resetting the model.

```
(reset)
```

Depending on which item should be used one of the retrieving productions is disabled using the new pdisable command.

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

Instead of presenting the items visibly just modify the chunk which will be placed into the goal buffer when the model runs.

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

Run the model for up to 30 seconds recording how much time passed and get the model's response from the state slot of the chunk in the goal buffer. Using the value returned by run as a response time is not generally a useful approach because there are often actions which are executed at the end of a run which did not affect the time of the model's actual response – things like the motor module returning to the free state after an action, an unneeded retrieval request completing or failing, etc. This model was written so that it "stops" at the appropriate time for determining the response since it does not actually perform any actions, but that can be difficult to do in general and it is usually better to record the actual time of responses as has been used in most of the experiments in the tutorial.

```
(let ((response-time (run 30.0))
    (response (chunk-slot-value-fct (buffer-read 'goal) 'state)))
```

Return the list of the time and whether the correct answer was given.

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

The do-person-location function takes one required parameter, term, which specifies which production to favor for the model as specified for fan-sentence. It iterates over all of the test sentences for the task collecting the data as returned by the fan-sentence function. It presents and returns a list of responses in the same order as the results in the global data list.

```
(defun do-person-location (term)
  (let ((test-set '(("lawyer" "store" t)("captain" "cave" t)("hippie" "church" t)
                    ("debutante" "bank" t)("earl" "castle" t)("hippie" "bank" t)
                    ("fireman" "park" t)("captain" "park" t)("hippie" "park" t)
                    ("fireman" "store" nil)("captain" "store" nil)
                    ("giant" "store" nil)("fireman" "bank" nil)
                    ("captain" "bank" nil)("giant" "bank" nil)
                    ("lawyer" "park" nil)("earl" "park" nil)
                    ("giant" "park" nil)))
        (results nil))

    (dolist (sentence test-set)
      (push (apply 'fan-sentence (append sentence (list term))) results))

    (reverse results)))
```

The fan-experiment function takes no parameters and will run the model through the experiment. It calls the do-person-location function once to favor the person and once to favor the location and prints out a table of the average response times and whether or not both responses were correct.

```
(defun fan-experiment ()
  (output-person-location (mapcar (lambda (x y)
                                    (list (/ (+ (car x) (car y)) 2.0)
                                          (and (cadr x) (cadr y))))
                            (do-person-location 'person)
                            (do-person-location 'location))))
```

The output-person-location function takes one parameter which is a list of responses that are in the same order as the global experimental data. It prints the comparison of that data to the experimental results and prints a table of the response times for targets and foils.

```
(defun output-person-location (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *person-location-data*)
    (mean-deviation rts *person-location-data*)
    (format t "~%TARGETS:~%                         Person fan~%")
    (format t  "  Location      1             2             3~%")
    (format t "    fan")

    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* i 3)) data))))

    (format t "~%~%FOILS:")
    (dotimes (i 3)
      (format t "~%     ~d    " (1+ i))
      (dotimes (j 3)
        (format t "~{~8,3F (~3s)~}" (nth (+ j (* (+ i 3) 3)) data))))))
```

Python

The first thing the code does is import the actr module to provide the ACT-R interface.

```
import actr
```

It loads the corresponding model file for this version of the task.

```
actr.load_act_r_model("ACT-R:tutorial;unit5;fan-no-pm-model.lisp")
```

Define a global variable that holds the experimental data to which the model will be compared.

```
person_location_data = [1.11, 1.17, 1.22,
                        1.17, 1.20, 1.22,
                        1.15, 1.23, 1.36,
                        1.20, 1.22, 1.26,
                        1.25, 1.36, 1.29,
                        1.26, 1.47, 1.47]
```

The sentence function takes 4 parameters which are very similar to the ones that were passed to the sentence function described in the main unit text, and for now we will ignore the difference but it will be described later. The first, person, is the string of the person named in the sentence. The second, location, is the string of the location named in the sentence. The third, target, is to be either True or False to indicate whether this is a target or a foil trial respectively, and the fourth, term, specifies which of the productions to favor in retrieving the study item and should be either the string person or location. The function returns a tuple of two items. The first is the time that the model spent running in seconds and the second item is True if the response was correct or False if there was no response or the response was incorrect.

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

Start by resetting the model.

```
    actr.reset()
```

Depending on which item should be used one of the retrieving productions is disabled using the new pdisable command.

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

Instead of presenting the items visibly just modify the chunk which will be placed into the goal buffer when the model runs.

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

Run the model for up to 30 seconds recording how much time passed (the first value returned by run) and get the model's response from the state slot of the chunk in the goal buffer. Using the value returned by run as a response time is not generally a useful approach because there are often actions which are executed at the end of a run which did not affect the time of the model's actual response – things like the motor module returning to the free state after an action, an unneeded retrieval request completing or failing, etc. This model was written so that it "stops" at the appropriate time for determining the response since it does not actually perform any actions, but that can be difficult to do in general and it is usually better to record the actual time of responses as has been used in most of the experiments in the tutorial.

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

Return the list of the time and whether the correct answer was given.

```
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 do_person_location function takes one required parameter, term, which specifies which production to favor for the model as specified for the sentence function. It iterates over all of the test sentences for the task collecting the data as returned by the sentence function. It presents and returns a list of responses in the same order as the results in the global data list. You may notice something odd about the way the words are specified below and how the response was tested above. The reason for that will be described below with the new commands.

```
def do_person_location(term):

    data = []

    for person,location,target in [("'lawyer'", "'store'", True),
                                   ("'captain'", "'cave'", True),
                                   ("'hippie'", "'church'", True),
                                   ("'debutante'", "'bank'", True),
                                   ("'earl'", "'castle'", True),
                                   ("'hippie'", "'bank'", True),
                                   ("'fireman'", "'park'", True),
                                   ("'captain'", "'park'", True),
                                   ("'hippie'", "'park'", True),
                                   ("'fireman'", "'store'", False),
                                   ("'captain'", "'store'", False),
                                   ("'giant'", "'store'", False),
                                   ("'fireman'", "'bank'", False),
                                   ("'captain'", "'bank'", False),
                                   ("'giant'", "'bank'", False),
                                   ("'lawyer'", "'park'", False),
                                   ("'earl'", "'park'", False),
                                   ("'giant'", "'park'", False)]:

        data.append(sentence(person,location,target,term))

    return data
```

The experiment function takes no parameters and will run the model through the experiment. It calls the do_person_location function once to favor the person and once to favor the location and prints out a table of the average response times and whether or not both responses were correct.

```
def experiment():

         output_person_location(list(map(lambda  x,y:((x[0]+y[0])/2,(x[1]  and
y[1])),
                                    do_person_location('person'),
                                    do_person_location('location'))))
```

The output_person_location function takes one parameter which is a list of responses that are in the same order as the global experimental data. It prints the comparison of that data to the experimental results and prints a table of the response times for targets and foils.

```
def output_person_location(data):

    rts = list(map(lambda x: x[0],data))

    actr.correlation(rts,person_location_data)
    actr.mean_deviation(rts,person_location_data)

    print("\nTARGETS:\n                         Person fan")
    print("  Location      1             2             3")
    print("    fan")

    for i in range(3):
        print("     %d      " % (i+1),end="")
        for j in range(3):
            print("%6.3f (%-5s)" % (data[j + (i * 3)]),end="")
        print()

    print()
    print("FOILS:")
    for i in range(3):
        print("     %d      " % (i+1),end="")
        for j in range(3):
            print("%6.3f (%-5s)" % (data[j + ((i + 3) * 3)]),end="")
        print()
```

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 tradeoffs when approaching a new task with respect to the objectives of the modeling effort.

Another difference is that this model uses the goal buffer instead of the imaginal module to hold the items. Therefore it must set the :ga parameter to have any activation spreading from those items since by default only the imaginal buffer is a source of activation. It is set to 1 in this model to match the other version since that is the default value for activation spreading from the imaginal buffer.

Grouped

The grouped model is really just a demonstration of partial matching. The experiment code is only there to collect and display 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 capability which will be described after the experiment code.

Lisp

It loads the corresponding model, and creates a global variable to hold the responses that the model makes:

```
(load-act-r-model "ACT-R:tutorial;unit5;grouped-model.lisp")

(defvar *response* nil)
```

The grouped-recall function is fairly simple. It adds a new command for the record-response function defined later, clears the global response variable, resets the model, runs it for up to 20 seconds, removes the new command, and then returns the response list after reversing it.

```
(defun grouped-recall ()
  (add-act-r-command "grouped-response" 'record-response
                     "Response recording function for the tutorial grouped model.")
  (setf *response* nil)
  (reset)
  (run 20)
  (remove-act-r-command "grouped-response")
  (reverse *response*))
```

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 pushes the values provided onto the response list. How that function actually gets called by the model will be described below.

```
(defun record-response (value)
  (push value *response*))
```

Python

It imports the actr module, loads the corresponding model, and creates a global variable to hold the responses that the model makes:

```
import actr

actr.load_act_r_model("ACT-R:tutorial;unit5;grouped-model.lisp")

response = []
```

The recall function is fairly simple. It adds a new command for the record_response function defined later, clears the global response variable, resets the model, runs it for up to 20 seconds, removes the new command, and then returns the response list.

```
def recall ():

    actr.add_command("grouped-response",record_response,
                     "Response recording function for the tutorial grouped model.")
    global response
    response = []
    actr.reset()
    actr.run(20)
    actr.remove_command("grouped-response")
    return response
```

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 appends the values provided onto the response list. How that function actually gets called by the model will be described below.

```
def record_response (item):

    global response
    response.append(item)
```

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 using the string of its name.]

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 binding of that variable in the instantiation is what will be used in the expression.

On the LHS of a production a !eval! specifies a condition that must be met before the production can be selected just like all 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 technically 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, so it should be easy to follow.

Lisp

It starts by loading the corresponding model.

```
(load-act-r-model "ACT-R:tutorial;unit5;siegler-model.lisp")
```

It defines global variables to hold the response, record whether or not it is already monitoring the speech output command, and to hold the experimental data to which the model will be compared.

```
(defvar *response*)
(defvar *monitor-installed* nil)

(defvar *siegler-data* '((0   .05 .86  0  .02  0  .02  0   0  .06)
                         (0   .04 .07 .75 .04  0  .02  0   0  .09)
                         (0   .02  0  .10 .75 .05 .01 .03  0  .06)
                         (.02  0  .04 .05 .80 .04  0  .05  0   0)
                         (0    0  .07 .09 .25 .45 .08 .01 .01 .06)
                         (.04  0   0  .05 .21 .09 .48  0  .02 .11)))
```

The record-model-speech function will monitor the output-speech command to record the model’s vocal response.

```
(defun record-model-speech (model string)
  (declare (ignore model))

  (setf *response* string))
```

Because there are multiple functions which can be used to perform different subsets of the task we create some functions to add and remove the monitoring function and keep track of whether or not it has already been created so we do not have to keep adding and removing it for efficiency when running the whole task repeatedly.

```
(defun add-speech-monitor ()
  (unless *monitor-installed*
    (add-act-r-command "siegler-response" 'record-model-speech "Siegler task model response")
    (monitor-act-r-command "output-speech" "siegler-response")
    (setf *monitor-installed* t)))

(defun remove-speech-monitor ()
  (remove-act-r-command-monitor "output-speech" "siegler-response")
  (remove-act-r-command "siegler-response")
  (setf *monitor-installed* nil))
```

The siegler-trial function takes two parameters which are the numbers to present.

```
(defun siegler-trial (arg1 arg2)
```

It resets the model and installs the microphone device to record the speech. In previous tasks that wasn’t necessary because it is installed automatically with an experiment window, but because there is no window this time it must be installed explicitly.

```
  (reset)


  (install-device (list "speech" "microphone"))
```

Check and record whether the speech monitor has already been added.

```
  (let ((need-to-remove (add-speech-monitor)))
```

It schedules the presentation of the digits to the model aurally with the new-digit-sound command which is very similar to the new-tone-sound command seen previously in the tutorial.

```
    (new-digit-sound arg1)
    (new-digit-sound arg2 .75)
```

It clears the response variable, runs the model for up to 30 seconds, removes the speech monitor if it was installed with this trial, and then returns the response.

```
    (setf *response* nil)
    (run 30)
    (when need-to-remove
      (remove-speech-monitor))

  *response*))
```

The siegler-set function takes no parameters. It runs one trial of each of the 6 stimuli and returns the list of the responses. Like the single trial function it also checks the monitoring state and removes the monitor if it had to add it.

```
(defun siegler-set ()

  (let ((need-to-remove (add-speech-monitor))
        (data (list (siegler-trial 1 1)
                    (siegler-trial 1 2)
                    (siegler-trial 1 3)
                    (siegler-trial 2 2)
                    (siegler-trial 2 3)
                    (siegler-trial 3 3))))
    (when need-to-remove
      (remove-speech-monitor))
    data))
```

The siegler-experiment function takes one parameter which is the number of times to repeat the set of 6 test stimuli. It installs the speech output monitor, runs that many times over the set of stimuli, removes the monitor, and then outputs the results.

```
(defun siegler-experiment (n)
  (add-speech-monitor)
  (let ((data nil))
    (dotimes (i n)
      (push (siegler-set) data))
    (remove-speech-monitor)
    (analyze data)))
```

The analyze function takes one parameter which is a list of lists where each sublist contains the responses to the 6 test stimuli. It calculates the percentage of each of the answers from zero to eight or other from the given trial data and calls display-results to print out that information.

```
(defun analyze (responses)
  (display-results
   (mapcar (lambda (x)
             (mapcar (lambda (y)
                       (/ y (length responses))) x))
     (apply 'mapcar
            (lambda (&rest z)
              (let ((res nil))
                (dolist (i '("zero" "one" "two" "three" "four"
                             "five" "six" "seven" "eight"))
                  (push (count i z :test 'string-equal) res)
                  (setf z (remove i z :test 'string-equal)))
                (push (length z) res)
                (reverse res)))
            responses))))
```

The display-results function takes one parameter which is a list of six lists where each sublist is a list of ten items which represent the percentages of the answers 0-8 or other for each of the 6 test stimuli. It prints out the comparison to the experimental data and then displays the table of the results.

```
(defun display-results (results)
  (let ((questions '("1+1" "1+2" "1+3" "2+2" "2+3" "3+3")))
    (correlation results *siegler-data*)
    (mean-deviation results *siegler-data*)
    (format t "       0     1     2     3     4     5     6     7     8   Other~%")
    (dotimes (i 6)
      (format t "~a~{~6,2f~}~%" (nth i questions) (nth i results)))))
```

Python

It starts by importing the actr module and loading the corresponding model.

```
import actr

actr.load_act_r_model("ACT-R:tutorial;unit5;siegler-model.lisp")
```

It defines global variables to hold the response, record whether or not it is already monitoring the speech output command, and to hold the experimental data to which the model will be compared.

```
response = False
monitor_installed = False

siegler_data = [[0, .05, .86,  0,  .02,  0, .02, 0, 0, .06],
                [0, .04, .07, .75, .04,  0, .02, 0, 0, .09],
                [0, .02, 0, .10, .75, .05, .01, .03, 0, .06],
                [.02, 0, .04, .05, .80, .04, 0, .05, 0, 0],
                [0, 0, .07, .09, .25, .45, .08, .01, .01, .06],
                [.04, 0, 0, .05, .21, .09, .48, 0, .02, .11]]
```

The record_model_speech function will monitor the output-speech command to record the model’s vocal response.

```
def record_model_speech (model,string):
    global response
    response = string.lower()
```

Because there are multiple functions which can be used to perform different subsets of the task we create some functions to add and remove the monitoring function and keep track of whether or not it has already been created so we do not have to keep adding and removing it for efficiency when running the whole task repeatedly.

```
def add_speech_monitor():
    global monitor_installed

    if monitor_installed == False:
        actr.add_command("siegler-response",record_model_speech,
                         "Siegler task model response")
        actr.monitor_command("output-speech","siegler-response")
        monitor_installed = True
        return True
    else:
        return False

def remove_speech_monitor():

    actr.remove_command_monitor("output-speech","siegler-response")
    actr.remove_command("siegler-response")

    global monitor_installed
    monitor_installed = False
```

The trial function takes two parameters which are the numbers to present.

```
def trial(arg1,arg2):
```

It resets the model and installs the microphone device to record the speech. In previous tasks that wasn't necessary because it is installed automatically with an experiment window, but because there is no window this time it must be installed explicitly.

```
    actr.reset()
    actr.install_device(["speech","microphone"])
```

Check and record whether the speech monitor has already been added.

```
    need_to_remove = add_speech_monitor()
```

It schedules the presentation of those digits to the model aurally with the new_digit_sound function which is very similar to the new_tone_sound function seen previously in the tutorial.

```
    actr.new_digit_sound(arg1)
    actr.new_digit_sound(arg2,.75)
```

It clears the response variable, runs the model for up to 30 seconds, removes the speech monitor if it was installed with this trial, and then returns the response.

```
global response
response = False
actr.run(30)
if need_to_remove:
    remove_speech_monitor()

return response
```

The set function takes no parameters. It runs one trial of each of the 6 stimuli and returns the list of the responses. Like the single trial function it also checks the monitoring state and removes the monitor if it had to add it.

```
def set ():

    need_to_remove = add_speech_monitor()

    data = [trial(1,1),trial(1,2),trial(1,3),
            trial(2,2),trial(2,3),trial(3,3)]

    if need_to_remove:
        remove_speech_monitor()

    return data
```

The experiment function takes one parameter which is the number of times to repeat the set of 6 test stimuli. It installs the speech output monitor, runs that many times over the set of stimuli, removes the monitor, and then outputs the results.

```
def experiment(n):

    add_speech_monitor()

    data = []

    for i in range(n):
        data.append(set())

    remove_speech_monitor()
    analyze(data)
```

The analyze function takes one parameter which is a list of lists where each sublist contains the responses to the 6 test stimuli. It calculates the percentage of each of the answers from zero to eight or other from the given trial data and calls display_results to print out that information.

```
def analyze(responses):

    results = [[0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0]]
    positions = {'zero':0,'one':1,'two':2,'three':3,'four':4,'five':5,
                 'six':6,'seven':7,'eight':8}


    for r in responses:
        for i in range(6):
            if r[i] in positions:
                results[i][positions[r[i]]] += 1
            else:
                results[i][9] += 1

    n = len(responses)

    for i in range(6):
        for j in range(10):
            results[i][j] /= n

    display_results(results)
```

The display_results function takes one parameter which is a list of six lists where each sublist is a list of ten items which represent the percentages of the answers 0-8 or other for each of the 6 test stimuli. It prints out the comparison to the experimental data and then displays the table of the results.

```
def display_results(results):

    questions = ["1+1","1+2","1+3","2+2","2+3","3+3"]

    actr.correlation(results,siegler_data)
    actr.mean_deviation(results,siegler_data)


    print("       0     1     2     3     4     5     6     7     8   Other")
    for i in range(6):
        print(questions[i],end="")

        for j in range(10):
            print("%6.2f" % results[i][j],end="")
        print()
```

1-hit Blackjack

Next, we’ll look at the 1-hit blackjack task. It 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. How exactly to use that will be discussed in a separate section below, and for now we will just describe the code.

Lisp

Unlike the other tasks this one starts by defining a function and adding a new command before loading the corresponding model. The function computes the similarities between numbers for the model and it must be added as a command before loading the model because the model’s :sim-hook parameter setting specifies the “1hit-bj-number-sims” command so it needs to be available when it is loaded.

```
(defun 1hit-bj-number-sims (a b)
  (when (and (numberp a) (numberp b))
    (- (/ (abs (- a b)) (max a b)))))

(add-act-r-command "1hit-bj-number-sims" '1hit-bj-number-sims
                   "Similarity between numbers for 1-hit blackjack task.")

(load-act-r-model "ACT-R:tutorial;unit5;1hit-blackjack-model.lisp")
```

It defines a lot of global variables which are used to hold the game information, player responses, and to control the default opponent for the model.

```
(defvar *deck1*)
(defvar *deck2*)
(defvar *opponent-rule*)
(defvar *opponent-feedback*)
(defvar *model-action*)
(defvar *human-action*)
(defvar *opponent-threshold*)
(defvar *key-monitor-installed* nil)
```

The respond-to-keypress function will be monitoring the output-key command and records the response for either a model or person playing the game and like the siegler code it has functions for adding and removing the monitor for efficiency.

```
(defmethod respond-to-keypress (model key)
  (if model
      (setf *model-action* key)
    (setf *human-action* key)))

(defun add-key-monitor ()
  (unless *key-monitor-installed*
    (add-act-r-command "1hit-bj-key-press" 'respond-to-keypress
                       "1-hit blackjack task key output monitor")
    (monitor-act-r-command "output-key" "1hit-bj-key-press")
    (setf *key-monitor-installed* t)))

(defun remove-key-monitor ()
  (remove-act-r-command-monitor "output-key" "1hit-bj-key-press")
  (remove-act-r-command "1hit-bj-key-press")
  (setf *key-monitor-installed* nil))
```

The onehit-hands function takes one required parameter which is the number of hands to play and an optional parameter which indicates whether or not to print the details of each hand played.

```
(defun onehit-hands (hands &optional (print-game nil))
```

Initialize a list of results and add the key press monitor if needed and record whether it was added.

```
  (let ((scores (list 0 0 0 0))
    (need-to-remove (add-key-monitor)))
```

For each hand deal the cards for the players from the appropriate decks, show the model its first two cards and the opponents first card and record its response, and then do the same for the opponent.

```
(dotimes (i hands)
  (let* ((mcards (deal *deck1*))
         (ocards (deal *deck2*))
         (mchoice (show-model-cards (butlast mcards) (first ocards)))
         (ochoice (show-opponent-cards (butlast ocards) (first mcards))))
```

If a player hit set their hand to all three cards, otherwise only the first two.

```
    (unless (string-equal "h" mchoice)
      (setf mcards (butlast mcards)))
    (unless (string-equal "h" ochoice)
      (setf ocards (butlast ocards)))
```

Determine the appropriate values for the players hands, determine their final outcomes, and then show those outcomes to the model and the opponent.

```
    (let* ((mtot (score-cards mcards))
           (otot (score-cards ocards))
           (mres (compute-outcome mcards ocards))
           (ores (compute-outcome ocards mcards)))

      (show-model-results mcards ocards mres ores)
      (show-opponent-results ocards mcards ores mres)
```

If the details are requested output those to *standard-output*.

```
      (when print-game
        (format t "Model: ~{~2d ~} -> ~2d (~4s)   Opponent: ~{~2d ~}-> ~2d (~4s)~%"
          mcards mtot mres ocards otot ores))
```

Update the scores based on the outcomes.

```
      (setf scores (mapcar '+ scores
                     (list (if (eq mres 'win) 1 0)
                           (if (eq ores 'win) 1 0)
                           (if (and (eq mres 'bust) (eq ores 'bust)) 1 0)
                           (if (and (= mtot otot)
                                    (not (eq mres 'bust))
                                    (not (eq ores 'bust))) 1 0)))))))
```

When done check whether the monitor needs to be removed and return the total scores.

```
(when need-to-remove
  (remove-key-monitor))

scores))
```

The onehit-blocks function takes two parameters: a number of blocks to run and how many hands to play in each block. It uses onehit-hands to play each block and returns the list of block scores.

```
(defun onehit-blocks (blocks block-size)
  (let (res
        (need-to-remove (add-key-monitor)))
    (dotimes (i blocks)
      (push (onehit-hands block-size) res))
    (when need-to-remove
      (remove-key-monitor))
    (reverse res)))
```

The game0 function sets all of the global variables to the values which describe the default game. *deck1* and *deck2* are set to the functions that implement the regular card deck. The *opponent-rule* is set to the fixed-threshold function, and the threshold used is set to 15. The default opponent does not change its play and does not need to see the feedback.

```
(defun game0 ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-rule* 'fixed-threshold)
  (setf *opponent-threshold* 15)
  (setf *opponent-feedback* nil))
```

The onehit-learning function takes one required parameter which is the number of 100 hand games to play and average. It takes two optional parameters. The first indicates whether or not a graph of the model's win proportions should be drawn and the second is the function to call to initialize the game control information. The defaults are to draw the graph and play the game set by the game0 function. It plays the indicated number of games resetting the model before each and collecting the data. If requested the graph is drawn, and then the lists of the win proportions are returned grouped into blocks of size 25 and 5.

```
(defun onehit-learning (n &optional (graph t) (game 'game0))
  (let ((data nil)
        (need-to-remove (add-key-monitor)))
    (dotimes (i n)
      (reset)
      (funcall game)
      (if (null data)
          (setf data (onehit-blocks 20 5))
        (setf data (mapcar (lambda (x y)
                             (mapcar '+ x y))
                     data (onehit-blocks 20 5)))))

    (when need-to-remove
      (remove-key-monitor))

    (let ((percentages (mapcar (lambda (x) (/ (car x) n 5.0)) data)))
      (when graph
        (draw-graph percentages))

      (list (list (/ (apply '+ (subseq percentages 0 5)) 5)
                  (/ (apply '+ (subseq percentages 5 10)) 5)
```

```
                    (/ (apply '+ (subseq percentages 10 15)) 5)
                    (/ (apply '+ (subseq percentages 15 20)) 5))
                    percentages))))
```

The draw-graph function takes one parameter which is a list of win percentages. It opens an experiment window and then draws a graph of those results in that window.

```
(defun draw-graph (points)
  (let ((w (open-exp-window "Data" :visible t :width 550 :height 460)))
    (add-line-to-exp-window w '(50 0) '(50 420) 'white)
    (dotimes (i 11)
      (add-text-to-exp-window w (format nil "~3,1f" (- 1 (* i .1)))
                              :x 5 :y (+ 5 (* i 40)) :width 35)
      (add-line-to-exp-window w (list 45 (+ 10 (* i 40)))
                              (list 550 (+ 10 (* i 40))) 'white))

    (let ((x 50))
      (mapcar (lambda (a b)
                (add-line-to-exp-window w (list x (floor (- 410 (* a 400))))
                                        (list (incf x 25) (floor (- 410 (* b 400))))
                                        'blue))
        (butlast points) (cdr points)))))
```

The deal function takes one parameter which is a function that implements a deck of cards for a player. That function is called three times to get the next three cards for a player and returns them in a list.

```
(defun deal (deck)
  (list (funcall deck)
        (funcall deck)
        (funcall deck)))
```

The score-cards function takes one required parameter which is a list of cards and an optional parameter which indicates the value over which a player busts. It returns the total value for the cards provided counting 1s as 11 as long as it does not bust.

```
(defun score-cards (cards &optional (bust 21))
  (let ((total (apply '+ cards)))
    (dotimes (i (count 1 cards))
      (when (<= (+ total 10) bust)
        (incf total 10)))
    total))
```

The compute-outcome function takes two required parameters which are the lists of cards for the two players and an optional parameter which is the limit before busting. It computes the score for each player and returns one of the symbols bust, win, or lose to indicate the result for the player holding the first set of cards provided.

```
(defun compute-outcome (p1cards p2cards &optional (bust 21))
  (let ((p1tot (score-cards p1cards bust))
        (p2tot (score-cards p2cards bust)))
    (if (> p1tot bust)
        'bust
      (if (or (> p2tot bust) (> p1tot p2tot))
          'win
        'lose))))
```

The show-model-cards function takes two parameters. The first is a list of the model's starting cards, and the second is the visible card of the opponent. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for the start of a hand, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds and the response it makes returned.

```
(defun show-model-cards (mcards ocard)
  (if (buffer-read 'goal)
      (mod-focus-fct `(mc1 ,(first mcards) mc2 ,(second mcards) mc3 nil
                      mtot nil mstart ,(score-cards mcards) mresult nil
                      oc1 ,ocard oc2 nil oc3 nil otot nil
                      ostart ,(score-cards (list ocard)) oresult nil
                      state start))
    (goal-focus-fct (car (define-chunks-fct
                           `((isa game-state mc1 ,(first mcards)
                                  mc2 ,(second mcards)
                                  mstart ,(score-cards mcards)
                                  oc1 ,ocard ostart ,(score-cards (list ocard))
                                  state start))))))
  (setf *model-action* nil)
  (run-full-time 10)
  *model-action*)
```

The show-model-results function takes four parameters. The first is a list of the model's cards, and the second is the list of the opponent's cards. The next two are the symbols representing the outcome for the model and the opponent respectively. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for providing the model the results, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds.

```
(defun show-model-results (mcards ocards mres ores)
  (if (buffer-read 'goal)
      (mod-focus-fct `(mc1 ,(first mcards)  mc2 ,(second mcards)
                  mc3 ,(third mcards) mtot ,(score-cards mcards)
                  mstart ,(score-cards (subseq mcards 0 2))
                  mresult ,mres oc1 ,(first ocards)
                  oc2 ,(second ocards) oc3 ,(third ocards)
                  otot ,(score-cards ocards)
                  ostart ,(score-cards (list (first ocards)))
                  oresult ,ores state results))
    (goal-focus-fct (car (define-chunks-fct
                           `((isa game-state mc1 ,(first mcards)
                              mc2 ,(second mcards) mc3 ,(third mcards)
                              mtot ,(score-cards mcards)
                              mstart ,(score-cards (subseq mcards 0 2))
                              mresult ,mres oc1 ,(first ocards)
                              oc2 ,(second ocards) oc3 ,(third ocards)
                              otot ,(score-cards ocards)
                              ostart ,(score-cards (list (first ocards)))
                              oresult ,ores state results))))))
  (run-full-time 10))
```

The play-human function is similar to the show-model-cards function but for a human player. It is takes two parameters. The first is a list of the player's starting cards, and the second is the visible card of the opponent. It opens an experiment window and displays the available information and then waits 10 seconds. It returns any keypress made during that time, or "s" indicating stay if no response is made.

```
(defun play-human (cards oc1)
  (let ((win (open-exp-window "Human")))
    (add-text-to-exp-window win "You" :x 50 :y 20)
    (add-text-to-exp-window win "Model" :x 200 :y 20)
    (dotimes (i 2)
      (dotimes (j 3)
        (add-text-to-exp-window win (format nil "C~d" (1+ j))
                                :x (+ 25 (* j 30) (* i 150)) :y 40 :width 20)
        (cond ((and (zerop i) (< j 2))
               (add-text-to-exp-window win (princ-to-string (nth j cards))
                                       :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20))
              ((and (= i 1) (zerop j))
               (add-text-to-exp-window win (princ-to-string oc1)
                                       :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20)))))
    (setf *human-action* nil)

    (let ((start-time (get-time nil)))
      (while (< (- (get-time nil) start-time) 10000)
        (process-events)))

    (if *human-action*
        *human-action*
      "s")))
```

The show-human-results function is similar to the show-model-results function but for a human player. It takes four parameters. The first is a list of the player's cards, and the second is the list of the opponent's cards. The next two are the symbols representing the outcome for the player and the opponent respectively. It displays the information in an experiment window and then waits for 10 seconds to pass.

```
(defun show-human-results (own-cards others-cards own-result others-result)
  (let ((win (open-exp-window "Human")))
    (add-text-to-exp-window win "You" :x 50 :y 20)
    (add-text-to-exp-window win "Model" :x 200 :y 20)
    (dotimes (i 2)
      (dotimes (j 3)
        (add-text-to-exp-window win (format nil "C~d" (1+ j))
                                :x (+ 25 (* j 30) (* i 150)) :y 40 :width 20)
        (if (zerop i)
            (when (nth j own-cards)
              (add-text-to-exp-window win (princ-to-string (nth j own-cards))
                                      :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20))
            (when (nth j others-cards)
              (add-text-to-exp-window win (princ-to-string (nth j others-cards))
                                      :x (+ 25 (* j 30) (* i 150)) :y 60 :width 20)))))
    (add-text-to-exp-window win (princ-to-string own-result) :x 50 :y 85)
    (add-text-to-exp-window win (princ-to-string others-result) :x 200 :y 85)
    (let ((start-time (get-time nil)))
      (while (< (- (get-time nil) start-time) 10000)
        (process-events)))))
```

The play-against-model function can be used to play as a person against the current model. It takes one required parameter which is the number of hands to play, and if the optional parameter is provided as non-nil then it will print out the details for each hand. If a visible window can be displayed, then it sets the interface variables to those necessary for a person to play, plays the requested number of hands, and then restores the interface variables to their previous values.

```
(defun play-against-model (count &optional (print-game nil))
  (if (visible-virtuals-available?)
      (let* ((old-rule *opponent-rule*)
             (old-feedback *opponent-feedback*)
             (*opponent-rule* 'play-human)
             (*opponent-feedback* 'show-human-results))
        (unwind-protect
            (onehit-hands count print-game)
          (progn
            (setf *opponent-rule* old-rule)
            (setf *opponent-feedback* old-feedback))))
        (print-warning "Cannot play against the model without a visible
window.")))
```

The show-opponent-cards function takes two parameters which are the list of cards and the other player’s face up card. It calls the function which has been set to determine the model’s opponent’s action to take using that information.

```
(defun show-opponent-cards (cards mc1)
  (funcall *opponent-rule* cards mc1))
```

The show-opponent-results function takes four parameters. The first is a list of the player’s cards, and the second is the list of the model’s cards. The next two are the symbols representing the outcome for the player and the model respectively. It calls the function which has been set to provide feedback to the model’s opponent if there is such a function.

```
(defun show-opponent-results (ocards mcards ores mres)
  (when *opponent-feedback*
    (funcall *opponent-feedback* ocards mcards ores mres)))
```

The regular-deck function returns card values as if they are dealt from an infinitely large deck of cards in the regular card deck proportions – one card for each of 1-9 and four 10s.

```
(defun regular-deck ()
  (min 10 (1+ (act-r-random 13))))
```

The fixed-threshold function implements an opponent for the model that will hit if the total value of their starting cards is less than the value of *opponent-threshold*.

```
(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*)
      "h"
    "s"))
```

Create a global variable for holding the list of cards in a deck that is stacked in a particular fashion prior to the deal.

```
(defvar *card-list* nil)
```

The always-hit function implements an opponent for the model that will hit regardless of the cards it is dealt.

```
(defun always-hit (cards mc1)
  "h")
```

The load-stacked-deck function creates a list of six cards to deal which are stacked as described for game 1 in the unit – the face up card for the opponent is a perfect indicator for the action the model must take to win.

```
(defun load-stacked-deck ()
  (let* ((c1 (+ 5 (act-r-random 6)))
         (c2 (+ 7 (act-r-random 4)))
         (c4 (if (> (act-r-random 1.0) .5) 2 8))
         (c3 (if (= c4 2) 10 (- 21 (+ c1 c2))))
         (c5 10)
         (c6 (if (= c4 2) 10 2)))
    (list c1 c2 c3 c4 c5 c6)))
```

The stacked-deck function deals the cards from the list created by load-stacked-deck, and it generates a new list of cards when needed.

```
(defun stacked-deck ()
  (cond (*card-list* (pop *card-list*))
        (t (setf *card-list* (load-stacked-deck))
           (pop *card-list*))))
```

The game1 function takes no parameters and sets the variables which control the game to those necessary for playing game 1 as described in the unit text, and it can be passed to onehit-learning as the third parameter.

```
(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit)
  (setf *opponent-feedback* nil))
```

Call the game0 function to set the default values for the control variables to those needed for playing game 0.

```
(game0)
```

Python

Start by importing the actr module and some other modules for the floor function and the Number class.

```
import actr
import math
import numbers
```

Unlike the other tasks this one starts by defining a function and adding a new command before loading the corresponding model. The function computes the similarities between numbers for the model and it must be added as a command before loading the model because the model's :sim-hook parameter setting specifies the "1hit-bj-number-sims" command so it needs to be available when it is loaded.

```
def onehit_bj_number_sims(a,b):

    if isinstance(b,numbers.Number) and isinstance(a,numbers.Number):
        return (- ( abs(a - b) / max(a,b)))
    else:
        return False

actr.add_command("1hit-bj-number-sims",onehit_bj_number_sims,
                 "Similarity between numbers for 1-hit blackjack task.")

actr.load_act_r_model("ACT-R:tutorial;unit5;1hit-blackjack-model.lisp")
```

It defines a lot of global variables which are used to hold the game information, player responses, and to control the default opponent for the model.

```
deck1 = None
deck2 = None
opponent_rule = None
opponent_feedback = None
model_action = None
human_action = None
opponent_threshold = None
key_monitor_installed = False
```

The respond_to_keypress function will be monitoring the output-key command and records the response for either a model or person playing the game and like the siegler code it has functions for adding and removing the monitor for efficiency.

```
def respond_to_keypress(model,key):
    global model_action,human_action

    if model:
        model_action = key
    else:
        human_action = key



def add_key_monitor():
    global key_monitor_installed

    if key_monitor_installed == False:
        actr.add_command("1hit-bj-key-press",respond_to_keypress,
                         "1-hit blackjack task key output monitor")
        actr.monitor_command("output-key","1hit-bj-key-press")
        key_monitor_installed = True
        return True
    else:
        return False

def remove_key_monitor():

    actr.remove_command_monitor("output-key","1hit-bj-key-press")
    actr.remove_command("1hit-bj-key-press")

    global key_monitor_installed
    key_monitor_installed = False
```

The hands function takes one required parameter which is the number of hands to play and an optional parameter which indicates whether or not to print the details of each hand played.

```
def hands(hands,print_game=False):
```

Initialize a list of results and add the key press monitor if needed and record whether it was added.

```
    scores = [0,0,0,0]
    need_to_remove = add_key_monitor()
```

For each hand deal the cards for the players from the appropriate decks, show the model its first two cards and the opponents first card and record its response, and then do the same for the opponent.

```
    for i in range(hands):
        mcards = deal(deck1)
        ocards = deal(deck2)
        mchoice = show_model_cards(mcards[0:2],ocards[0])
        ochoice = show_opponent_cards(ocards[0:2],mcards[0])
```

If a player hit set their hand to all three cards, otherwise only the first two.

```
        if not(mchoice.lower() == 'h'):
            mcards = mcards[0:2]

        if not(ochoice.lower() == 'h'):
            ocards = ocards[0:2]
```

Determine the appropriate values for the players hands, determine their final outcomes, and then show those outcomes to the model and the opponent.

```
        mtot = score_cards(mcards)
        otot = score_cards(ocards)
        mres = compute_outcome(mcards,ocards)
        ores = compute_outcome(ocards,mcards)

        show_model_results(mcards,ocards,mres,ores)
        show_opponent_results(ocards,mcards,ores,mres)
```

If the details are requested print them out.

```
        if print_game:
            print("Model: ",end="")
            for c in mcards:
                print("%2d "%c,end="")
            print(" -> %2d (%-4s)   Opponent: "%(mtot,mres),end="")
            for c in ocards:
                print("%2d "%c,end="")
            print("-> %2d (%-4s)"%(otot,ores))
```

Update the scores based on the outcomes.

```
        if mres == 'win':
            scores[0] += 1
        if ores == 'win':
            scores[1] += 1
        if mres == 'bust' and ores == 'bust':
            scores[2] += 1
        if mtot == otot and not(mres == 'bust') and not(ores == 'bust'):
            scores[3] += 1
```

When done check whether the monitor needs to be removed and return the total scores.

```
    if need_to_remove:
        remove_key_monitor()

    return scores
```

The blocks function takes two parameters: a number of blocks to run and how many hands to play in each block. It uses hands to play each block and returns the list of block scores.

```
def blocks(blocks,block_size):

    res = []
    need_to_remove = add_key_monitor()

    for i in range(blocks):
        res.append(hands(block_size))

    if need_to_remove:
        remove_key_monitor()
    return res
```

The game0 function sets all of the global variables to the values which describe the default game. deck1 and deck2 are set to the functions that implement the regular card deck. The opponent_rule is set to the fixed_threshold function, and the threshold used is set to 15. The default opponent does not change its play and does not need to see the feedback.

```
def game0():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback

    deck1 = regular_deck
    deck2 = regular_deck
    opponent_rule = fixed_threshold
    opponent_threshold = 15
    opponent_feedback = None
```

The sum_lists function returns a list which is the sum of the values from the two lists provided, and is added to make the learning function a little easier to read.

```
def sum_lists(x,y):
    return list(map(lambda v,w: v + w,x,y))
```

The learning function takes one required parameter which is the number of 100 hand games to play and average. It takes two optional parameters. The first indicates whether or not a graph of the model's win proportions should be drawn and the second is the function to call to initialize the game control information. The defaults are to draw the graph and play the game set by the game0 function. It plays the indicated number of games resetting the model before each and collecting the data. If requested the graph is drawn, and then the lists of the win proportions are returned grouped into blocks of size 25 and 5.

```
def learning(n,graph=True,game=game0):

    data = [[0,0,0,0]]*20
    need_to_remove = add_key_monitor()

    for i in range(n):
        actr.reset()
        game()
        data = list(map(lambda x,y: sum_lists(x,y),data,blocks(20,5)))

    if need_to_remove:
        remove_key_monitor()

    percentages = list(map(lambda x: x[0] / n / 5,data))

    if graph:
        draw_graph(percentages)

    return [[sum(percentages[0:5])/5,
             sum(percentages[5:10])/5,
             sum(percentages[10:15])/5,
             sum(percentages[15:20])/5],
                percentages]
```

The draw_graph function takes one parameter which is a list of win percentages. It opens an experiment window and then draws a graph of those results in that window.

```
def draw_graph(points):

    w = actr.open_exp_window('Data', visible=True, width=550, height=460)
    actr.add_line_to_exp_window(w,[50,0],[50,420],'white')

    for i in range(11):
        actr.add_text_to_exp_window(w,"%3.1f"%(1.0 - (i * .1)),
                                    x=5, y=(5 + (i * 40)), width=35)
        actr.add_line_to_exp_window(w,[45,10 + (i * 40)],
                                    [550,10 + (i * 40)],'white')

    x = 50

    for (a,b) in zip(points[0:-1],points[1:]):
        actr.add_line_to_exp_window(w,[x,math.floor(410 - (a * 400))],
                                    [x+25,math.floor(410 - (b * 400))],
                                    'blue')
        x += 25
```

The deal function takes one parameter which is a function that implements a deck of cards for a player. That function is called three times to get the next three cards for a player and returns them in a list.

```
def deal(deck):
    return [deck(),deck(),deck()]
```

The score_cards function takes one required parameter which is a list of cards and an optional parameter which indicates the value over which a player busts. It returns the total value for the cards provided counting 1s as 11 as long as it does not bust.

```
def score_cards(cards,bust=21):

    total = sum(cards)

    for i in range(cards.count(1)):
        if (total + 10) <= bust:
            total += 10

    return total
```

The compute_outcome function takes two required parameters which are the lists of cards for the two players and an optional parameter which is the limit before busting. It computes the score for each player and returns one of the strings bust, win, or lose to indicate the result for the player holding the first set of cards provided.

```
def compute_outcome(p1cards,p2cards,bust=21):
    p1tot = score_cards(p1cards,bust)
    p2tot = score_cards(p2cards,bust)
    if p1tot > bust:
        return 'bust'
    elif p2tot > bust:
        return 'win'
    elif p1tot > p2tot:
        return 'win'
    else:
        return 'lose'
```

The show_model_cards function takes two parameters. The first is a list of the model’s starting cards, and the second is the visible card of the opponent. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for the start of a hand, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds and the response it makes returned (the default action is to stay unless the model hits a key).

```
def show_model_cards(mcards,ocard):

    if actr.buffer_read('goal'):
        actr.mod_focus('mc1',mcards[0],'mc2', mcards[1],'mc3',None,'mtot',None,
                       'mstart',score_cards(mcards),'mresult',None,'oc1',ocard,
                       'oc2',None,'oc3',None,'otot',None,'ostart',score_cards([ocard]),
                       'oresult',None,'state','start')
    else:
        actr.goal_focus(actr.define_chunks(['isa','game-state','mc1',mcards[0],
                                            'mc2', mcards[1],'mstart',score_cards(mcards),
                                            'oc1', ocard,'ostart',score_cards([ocard]),
                                            'state','start'])[0])

    global model_action
    model_action = 's'
    actr.run_full_time(10)
    return model_action
```

The show_model_results function takes four parameters. The first is a list of the model’s cards, and the second is the list of the opponent’s cards. The next two are the strings representing the outcome for the model and the opponent respectively. If there is a chunk in the goal buffer then it is modified to contain the appropriate information for providing the model the results, and if there is not a chunk in the goal buffer a new chunk is created with that information and then placed into the goal buffer. Then the model is run for exactly 10 seconds.

```
def show_model_results(mcards,ocards,mres,ores):

    if len(mcards) ==3:
        mhit = mcards[2]
    else:
        mhit = 'nil'

    if len(ocards) ==3:
        ohit = ocards[2]
    else:
        ohit = 'nil'

    if actr.buffer_read('goal'):
        actr.mod_focus('mc1',mcards[0],'mc2', mcards[1],'mc3',mhit,
                       'mtot',score_cards(mcards),'mstart',score_cards(mcards[0:2]),
                       'mresult',mres,'oc1',ocards[0],'oc2',ocards[1],'oc3',ohit,
                       'otot',score_cards(ocards),
                       'ostart',score_cards(ocards[0:1]),'oresult',ores,'state','results')
    else:
                                              actr.goal_focus(actr.define_chunks(['isa','game-
state','mc1',mcards[0],'mc2',mcards[1],
                                            'mc3',mhit,'mtot',score_cards(mcards),
                                            'mstart',score_cards(mcards[0:2]),'mresult',mres,
                                            'oc1',ocards[0],'oc2',ocards[1],'oc3',ohit,
                                            'otot',score_cards(ocards),
                                            'ostart',score_cards(ocards[0:1]),'oresult',ores,
                                            'state','results'])[0])

    actr.run_full_time(10)
```

The play_human function is similar to the show_model_cards function but for a human player. It is takes two parameters. The first is a list of the player's starting cards, and the second is the visible card of the opponent. It opens an experiment window and displays the available information and then waits 10 seconds. It returns any keypress made during that time, or "s" indicating stay if no response is made.

```
def play_human(cards,oc1):

    win = actr.open_exp_window('Human')
    actr.add_text_to_exp_window(win,'You',50,20)
    actr.add_text_to_exp_window(win,'Model',200,20)

    for i in range(2):
        for j in range(3):
            actr.add_text_to_exp_window(win,"C%d"%(j+1),x=(25 + (j * 30) + (i * 150)),
                                        y=40, width=20)
            if i == 0 and j < 2:
                actr.add_text_to_exp_window(win,str(cards[j]),
                                            x=(25 + (j * 30) + (i * 150)), y=60, width=20)
            if i == 1 and j == 0:
                actr.add_text_to_exp_window(win,str(oc1),x=(25 + (j * 30) + (i * 150)),
                                            y=60, width=20)

    global human_action
    human_action = None

    start_time = actr.get_time(False)

    while (actr.get_time(False) - start_time) < 10000:
        actr.process_events()

    if human_action:
        return human_action
    else:
        return 's'
```

The show_human_results function is similar to the show_model_results function but for a human player. It takes four parameters. The first is a list of the player's cards, and the second is the list of the opponent's cards. The next two are the strings representing the outcome for the player and the opponent respectively. It displays the information in an experiment window and then waits for 10 seconds to pass.

```
def show_human_results(own_cards,others_cards,own_result,others_result):

    win = actr.open_exp_window('Human')
    actr.add_text_to_exp_window(win,'You',50,20)
    actr.add_text_to_exp_window(win,'Model',200,20)

    for i in range(2):
        for j in range(3):
            actr.add_text_to_exp_window(win,"C%d"%(j+1),x=(25 + (j * 30) + (i * 150)),
                                        y=40, width=20)
            if i == 0:
                if j < len(own_cards):
                    actr.add_text_to_exp_window(win,str(own_cards[j]),
                                                x=(25 + (j * 30) + (i * 150)),
                                                y=60, width=20)
            else:
                if j < len(others_cards):
                    actr.add_text_to_exp_window(win,str(others_cards[j]),
                                                x=(25 + (j * 30) + (i * 150)),
                                                y=60, width=20)

    actr.add_text_to_exp_window(win,own_result,50,85)
    actr.add_text_to_exp_window(win,others_result,200,85)

    start_time = actr.get_time(False)

    while (actr.get_time(False) - start_time) < 10000:
        actr.process_events()
```

The play_against_model function can be used to play as a person against the current model. It takes one required parameter which is the number of hands to play, and if the optional parameter is provided as True then it will print out the details for each hand. If a visible window can be displayed, then it sets the interface variables to those necessary for a person to play, plays the requested number of hands, and then restores the interface variables to their previous values.

```
def play_against_model(count,print_games=False):
    global opponent_rule,opponent_feedback

    if actr.visible_virtuals_available():
        old_rule = opponent_rule
        old_feedback = opponent_feedback

        opponent_rule = play_human
        opponent_feedback = show_human_results

        result = ()
        try:
            result = hands(count,print_games)
        finally:
            opponent_rule = old_rule
            opponent_feedback = old_feedback
        return(result)
    else:
        actr.print_warning("Cannot play against the model without a visible
                            window.")
```

The show_opponent_cards function takes two parameters which are the list of cards and the other player’s face up card. It calls the function which has been set to determine the model’s opponent’s action to take using that information.

```
def show_opponent_cards(ocards,mc1):
    return opponent_rule(ocards,mc1)
```

The show_opponent_results function takes four parameters. The first is a list of the player’s cards, and the second is the list of the model’s cards. The next two are the strings representing the outcome for the player and the model respectively. It calls the function which has been set to provide feedback to the model’s opponent if there is such a function.

```
def show_opponent_results(ocards,mcards,ores,mres):
    if opponent_feedback:
        opponent_feedback(ocards,mcards,ores,mres)
```

The regular_deck function returns card values as if they are dealt from an infinitely large deck of cards in the regular card deck proportions – one card for each of 1-9 and four 10s.

```
def regular_deck():
    return min(10,actr.random(13)+1)
```

The fixed_threshold function implements an opponent for the model that will hit if the total value of their starting cards is less than the value of opponent_threshold.

```
def fixed_threshold(cards,mc1):

    if score_cards(cards) < opponent_threshold:
        return 'h'
    else:
        return 's'
```

Create a global variable for holding the list of cards in a deck that is stacked in a particular fashion prior to the deal.

```
card_list = []
```

The always_hit function implements an opponent for the model that will hit regardless of the cards it is dealt.

```
def always_hit(cards,mc1):
    return 'h'
```

The load_stacked_deck function creates a list of six cards to deal which are stacked as described for game 1 in the unit – the face up card for the opponent is a perfect indicator for the action the model must take to win.

```
def load_stacked_deck():
    c1 = 5 + actr.random(6)
    c2 = 7 + actr.random(4)
    if actr.random(1.0) > .5:
        c4 = 2
    else:
        c4 = 8
    if c4 == 2:
        c3 = 10
        c6 = 10
    else:
        c3 = 21 - (c1 + c2)
        c6 = 2
    c5 = 10

    return [c1,c2,c3,c4,c5,c6]
```

The stacked_deck function deals the cards from the list created by load_stacked_deck, and it generates a new list of cards when needed.

```
def stacked_deck():
    global card_list

    if len(card_list) == 0:
        card_list = load_stacked_deck()

    c = card_list[0]
    card_list = card_list[1:]

    return c
```

The game1 function takes no parameters and sets the variables which control the game to those necessary for playing game 1 as described in the unit text, and it can be passed to the learning function as the third parameter.

```
def game1():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback,card_list

    card_list = []
    deck1 = stacked_deck
    deck2 = stacked_deck
    opponent_rule = always_hit
    opponent_feedback = None
```

Call the game0 function to set the default values for the control variables to those needed for playing game 0.

```
game0()
```

1hit-blackjack-model

In addition to the new parameters set in the model definition, which were described in the main unit text, there are three new commands used in this model. 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 informs the procedural module that a buffer will be set with chunk slot names other than those which are set 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.

This command is used to avoid 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 the goal 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 have no slots we can simply specify their 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 along with the window for the vision module, but since this task does not have a visual interface for the model we are not creating an experiment window and need to install the keyboard explicitly so that we can collect the model's actions from key presses. It is also used in the siegler task in this unit to install the microphone device for the speech module to record the model's vocal responses.

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)
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1

(SECOND-GOAL-0)

>>> actr.buffer_chunk('goal')
```

```
GOAL: SECOND-GOAL-0
SECOND-GOAL-0
   ARG1  5
   ARG2  2
   SUM  6
   COUNT  1
['SECOND-GOAL-0']
```

Strings in slots

Generally, using strings to name things should 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. Similarly, when returning a value which should be considered as a string it will contain single quotes around the item. That can be seen in the code for the fan task above where it is testing the value which the model placed into the state slot since that was a string:

```
    response = actr.chunk_slot_value(actr.buffer_read("goal"),"state")

    if target:
        if response.lower() == "'k'".lower():
     ...
```

and also in the specification of the values which are being placed into the slots of the goal buffer by the code:

```
    for person,location,target in [("'lawyer'", "'store'", True),
                                   ("'captain'", "'cave'", True),
                                   ("'hippie'", "'church'", True),
    ...
```

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, say 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. The following is not going to work:

```
? (defvar *number* 3)
*NUMBER*
? (chunk-type test slot)
TEST
? (add-dm (foo isa test slot *number*))
(FOO)
```

The add-dm macro doesn't evaluate the variable *number* to get its value. Instead that will create a chunk that literally contains *number* as shown here using the dm command shown in unit 1:

```
? (dm foo)
FOO
   SLOT  *NUMBER*
```

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 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 'isa 'test 'slot *number*)))
(FOO)
? (dm foo)
FOO
   SLOT  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 would be that as we saw in this unit 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. 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, and 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).

chunk-slot-value – This command returns the value in a particular slot of a chunk. The parameters for all versions, Lisp macro (**chunk-slot-value**), Python function (**chunk_slot_value**), and Lisp function (**chunk-slot-value-fct**), are the name of the chunk and the name of the slot. It returns the value of that slot from that chunk or nil (Lisp) or None (Python) if the chunk does not have the specified slot.

buffer-read - This command is used to get the name of the chunk in a buffer. The Lisp function (**buffer-read**) and the Python function (**buffer_read**) both take one parameter which must be the name of a buffer. If that buffer contains a chunk then its name is returned otherwise nil or None is returned to indicate the buffer is empty.

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

The last section for this unit will be to 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 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.

For the default game described in the unit text both the model and opponent are dealt cards from these functions in Lisp or Python respectively:

```
(defun regular-deck ()
  (min 10 (1+ (act-r-random 13))))

def regular_deck():
    return min(10,actr.random(13)+1)
```

and the decks for both players are set to that value in the game0 functions like this:

```
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
```

or

```
    deck1 = regular_deck
    deck2 = regular_deck
```

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 (that is used if you use the function to play the game yourself against the model).

For the default game from the unit the opponent has a simple rule of always hitting when it has a total of less than 15 for its starting cards. These are the functions which implement that:

```
(defun fixed-threshold (cards mc1)
  (if (< (score-cards cards) *opponent-threshold*) "h" "s"))
```

```
def fixed_threshold(cards,mc1):

    if score_cards(cards) < opponent_threshold:
        return 'h'
    else:
        return 's'
```

The score-cards and score_cards functions compute the score for a list of card values taking into account the rule of a 1 being counted as 11 when possible. Also note that we have introduced another variable for manipulating the game in these functions. The value at which the opponent will stay is set with the variable *opponent-threshold* or opponent_threshold. All of these variables are set to create the default game scenario in the game0 functions:

```
(defun game0 ()
  (setf *deck1* 'regular-deck)
  (setf *deck2* 'regular-deck)
  (setf *opponent-rule* 'fixed-threshold)
  (setf *opponent-threshold* 15)
  (setf *opponent-feedback* nil))
```

```
def game0():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback

    deck1 = regular_deck
    deck2 = regular_deck
    opponent_rule = fixed_threshold
    opponent_threshold = 15
    opponent_feedback = None
```

That game is the default value for the third parameter to onehit-learning and learning if no game function is specified.

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 this 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 the setup function looks like this:

```
(defun game1 ()
  (setf *card-list* nil)
  (setf *deck1* 'stacked-deck)
  (setf *deck2* 'stacked-deck)
  (setf *opponent-rule* 'always-hit)
  (setf *opponent-feedback* nil))
```

```
def game1():
    global deck1,deck2,opponent_threshold,opponent_rule,opponent_feedback,card_list

    card_list = []
    deck1 = stacked_deck
    deck2 = stacked_deck
    opponent_rule = always_hit
    opponent_feedback = None
```

The details of the functions used there can be found in the description of the code above.

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 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.