Unit 3 Code Description

The experiments in this unit are more complicated than those in the previous unit, and they also involve collecting data and comparing it to existing experimental results. Many of the ACT-R commands that were used in the last unit's tasks will be seen again and there will be a few more introduced in this unit.

Sperling experiment

First we will describe the code which implements the sperling task and the new ACT-R commands will be highlighted in red.

Lisp version

The first thing that the code does is load the corresponding model which can perform the task.

```
(load-act-r-model "ACT-R:tutorial;unit3;sperling-model.lisp")
```

Three global variables are defined. *responses* will hold the list of keys pressed by the participant and *show-responses* is used as a flag to indicate whether or not to print out the participant's responses every trial. *sperling-exp-data* holds the results of the original experiment so that the model's performance can be compared to it.

```
(defvar *responses* nil)
(defparameter *show-responses* t)

(defvar *sperling-exp-data* '(3.03 2.40 2.03 1.50))
```

The sperling-trial function is now defined which will run one trial of the task. It takes one parameter which is the delay time at which to present the auditory cue in seconds. It presents the array of letters, waits for the responses, and returns the number of letters correctly recalled in the target row.

```
(defun sperling-trial (onset-time)
```

It starts by resetting the system to return the model to its starting state.

```
(reset)
```

Some local variables are created. Letters holds a randomly ordered list of the possible letters to display. Answers is set to nil, but will hold the list of letters which are the correct responses. Row is set to a random number between 0 and 2 and will determine which row is the correct one and which tone to present. Window is set to a window which is opened to display the task. Freq is created without a value, and will be used to hold the frequency of the tone presented.

```
(let* ((letters (permute-list '("B" "C" "D" "F" "G" "H" "J"
                                "K" "L" "M" "N" "P" "Q" "R"
                                "S" "T" "V" "W" "X" "Y" "Z")))
       (answers nil)
       (row (act-r-random 3))
       (window (open-exp-window "Sperling Experiment" :visible t))
       freq)
```

It displays the first 12 letters from the letters list in 3 rows of 4 and records the letters from the target row in a list stored in the answers variable.

```
(dotimes (i 3)
  (dotimes (j 4)
    (let ((txt (nth (+ j (* i 4)) letters)))
      (when (= i tone)
        (push txt answers))
      (add-text-to-exp-window window txt :x (+ 75 (* j 50))
                                         :y (+ 100 (* i 50))))))
```

The model is told which window to interact with using the install-device function.

```
(install-device window)
```

The freq variable is set to the appropriate value based on which row is the target.

```
(case row
  (0
   (setf freq 2000))
  (1
   (setf freq 1000))
  (2
   (setf freq 500)))
```

A tone is scheduled to occur for the model at the onset time specified, and which lasts for .5 seconds and has the appropriate frequency.

```
(new-tone-sound freq .5 onset-time)
```

To simulate the persistent visual memory we will clear the screen after a randomly chosen time between .9 and 1.1 seconds have passed. This is done by scheduling an action to happen for the model at that time, and the action we want to perform is the ACT-R clear-exp-window command.

```
(schedule-event-relative (+ 900 (act-r-random 200)) "clear-exp-window"
                              :params (list window) :time-in-ms t)
```

The variable to hold the model's responses is cleared.

```
(setf *responses* nil)
```

A command is added for the respond-to-key-press function (defined below) and then that function is set to monitor the output-key command so that we can record the keys which are pressed.

```
(add-act-r-command "sperling-response" 'respond-to-key-press
                   "Sperling task key press response monitor")
(monitor-act-r-command "output-key" "sperling-response")
```

The model is run for up to 30 seconds in real time mode.

```
(run 30 t)
```

We stop monitoring the output-key command and remove our new command.

```
(remove-act-r-command-monitor "output-key" "sperling-response")
(remove-act-r-command "sperling-response")
```

If the *show-responses* variable is true then we print out the correct answers for the trial and the responses which the model made.

```
(when *show-responses*
  (format t "~%~%answers: ~S~%responses: ~S~%" answers *responses*))
```

Finally we call the compute-score function (defined below) and return its result.

```
(compute-score answers)))
```

The compute-score function is defined. It requires one parameter which is the list of correct answers and it returns the number of correct responses that were made as recorded in the *responses* variable (ignoring duplicates).

```
(defun compute-score (answers)
  (let ((score 0))
    (dolist (x answers score)
      (when (member x *responses* :test 'string-equal)
        (incf score)))))
```

The respond-to-key-press function is defined. It is monitoring the output-key command and thus will be passed two parameters. The first will be the name of a model which makes a key press and the second will be the name of that key. If the key pressed is not the space bar then it is recorded in the *responses* variable.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))

  (unless (string-equal key "space")
    (push key *responses*)))
```

The report-data function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be compared to the original experimental data in the *sperling-exp-data* variable and the print-results function will be called to display the results.

```
(defun report-data (data)

  (correlation data *sperling-exp-data*)
  (mean-deviation data *sperling-exp-data*)
  (print-results data))
```

The print-results function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be printed along with the data in the *sperling-exp-data* variable.

```
(defun print-results (data)

  (format t "~%Condition    Current Participant   Original Experiment~%")
  (do ((condition '(0.00 0.15 0.30 1.00) (cdr condition))
       (temp1 data (cdr temp1))
       (temp2 *sperling-exp-data* (cdr temp2)))
      ((null temp1))
    (format t " ~4,2F sec.          ~6,2F                ~6,2F~%"
              (car condition) (car temp1) (car temp2))))
```

The one-block function is defined which takes no parameters. It runs the model through one trial of each condition in the experiment in a random order and returns the list of results ordered by condition.

```
(defun one-block ()
  (let ((result nil))

    (dolist (x (permute-list '(0.0 .15 .30 1.0)))
      (push (cons x (sperling-trial x)) result))
    (mapcar 'cdr (sort result '< :key 'car))))
```

The sperling-experiment function is defined which takes one parameter. It runs the model through that specified number of blocks of the experiment averaging the results returned by one-block and then passes that average data to the report-data function defined above to compare it to the original task and print them.

```
(defun sperling-experiment (n)
  (let ((results (list 0 0 0 0)))
    (dotimes (i n)
      (setf results (mapcar '+ results (one-block))))
    (report-data (mapcar (lambda (x) (/ x n)) results))))
```

Python

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

```
import actr
```

Then it loads the corresponding model which can perform the task.

```
actr.load_act_r_model("ACT-R:tutorial;unit3;sperling-model.lisp")
```

Three global variables are defined. responses will hold the list of keys pressed by the participant and show-responses is used as a flag to indicate whether or not to print out the participant's responses every trial. exp-data holds the results of the original experiment so that the model's performance can be compared to it.

```
responses = []
show_responses = True

exp_data = [3.03,2.4,2.03,1.5]
```

The trial function is defined which will run one trial of the task. It takes one parameter which is the delay time at which to present the auditory cue in seconds. It presents the array of letters, waits for the responses, and returns the number of letters correctly recalled in the target row.

```
def trial(onset_time):
```

It starts by resetting the system to return the model to its starting state.

```
    actr.reset()
```

Some local variables are created. letters holds a randomly ordered list of the possible letters to display. answers is set to an empty list, but will hold the list of letters which are the correct responses. row is set to a random number between 0 and 2 and will determine which row is the correct one and which tone to present. window is set to a window which is opened to display the task.

```
    letters = actr.permute_list(["B","C","D","F","G","H","J",
                                 "K","L","M","N","P","Q","R",
                                 "S","T","V","W","X","Y","Z"])
    answers = []
    row = actr.random(3)
```

```
window = actr.open_exp_window("Sperling Experiment", visible=True)
```

It displays the first 12 letters from the letters list in 3 rows of 4 and records the letters from the target row in the list stored in the answers variable.

```
for i in range(3):
    for j in range(4):
        txt = letters[j + (i * 4)]
        if i == row:
            answers.append(txt)
        actr.add_text_to_exp_window(window,txt, x=(75 + (j * 50)),
                                    y=(100 + (i * 50)))
```

The model is told which window to interact with using the install-device function.

```
actr.install_device(window)
```

The freq variable is set to the appropriate value based on which row is the target.

```
if row == 0:
    freq = 2000
elif row == 1:
    freq = 1000
else:
    freq = 500
```

A tone is scheduled to occur for the model at the onset time specified, and which lasts for .5 seconds and has the appropriate frequency.

```
actr.new_tone_sound(freq,.5,onset_time)
```

To simulate the persistent visual memory we will clear the screen after a randomly chosen time between .9 and 1.1 seconds have passed. This is done by scheduling an action to happen for the model at that time, and the action we want to perform is the ACT-R clear-exp-window command.

```
 actr.schedule_event_relative(900 + actr.random(200),
                                   "clear-exp-window",
                                   params=[window],time_in_ms=True)
```

The global variable to hold the model's responses is cleared.

```
global responses
responses = []
```

A command is added for the respond_to_key_press function (defined below) and then that function is set to monitor the output-key command so that we can record the keys which are pressed.

```
actr.add_command("sperling-response",respond_to_key_press,
                     "Sperling task key press response monitor")
    actr.monitor_command("output-key","sperling-response")
```

The model is run for up to 30 seconds in real time mode.

```
    actr.run(30,True)
```

We stop monitoring the output-key command and remove our new command.

```
    actr.remove_command_monitor("output-key","sperling-response")
    actr.remove_command("sperling-response")
```

If the show-responses variable is true then we print out the correct answers for the trial and the responses which the model made.

```
    if show_responses:
        print("answers: %s"%answers)
        print("responses: %s"%responses)
```

Finally we call the compute-score function (defined below) and return its result.

```
    return(compute_score(answers))
```

The compute_score function is defined. It requires one parameter which is the list of correct answers and it returns the number of correct responses that were made as recorded in the responses variable (ignoring duplicates).

```
def compute_score(answers):

    score = 0

    for s in responses:
        if s.upper() in answers:
            score += 1

    return(score)
```

The respond_to_key_press function is defined. It is monitoring the output-key command and thus will be passed two parameters. The first will be the name of a model which makes a key press and the second will be the name of that key. If the key pressed is not the space bar then it is recorded in the responses variable.

```
def respond_to_key_press (model,key):
    global responses

    if not(key.lower() == "space"):
        responses.append(key)
```

The report_data function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be compared to the original experimental data in the exp_data variable and the print_results function will be called to display the results.

```
def report_data(data):

    actr.correlation(data,exp_data)
    actr.mean_deviation(data,exp_data)
    print_results (data)
```

The print_results function is defined which takes one parameter that should be a list of the average number of items reported ordered by condition. That data will be printed along with the data in the exp-data variable.

```
def print_results(data):

    print("Condition    Current Participant   Original Experiment")
    for (c,d,o) in zip([0.0,0.15,0.3,1.0],data,exp_data):
        print(" %4.2f sec.          %6.2f                %6.2f"%(c,d,o))
```

The one_block function is defined which takes no parameters. It runs the model through one trial of each condition in the experiment in a random order and returns the list of results ordered by condition.

```
def one_block():

    result = []

    for t in actr.permute_list([0.0,.15,.3,1.0]):
        result.append((t,trial(t)))

    result.sort()
    return (list(map(lambda x: x[1],result)))
```

The experiment function is defined which takes one parameter. It runs the model through that specified number of blocks of the experiment averaging the results returned by one_block and then passes that average data to the report_data function defined above to compare it to the original task and print them.

```
def experiment(n):
    results=[0,0,0,0]

    for i in range(n):
        results=list(map(lambda x,y: x + y,results,one_block()))

    report_data(list(map(lambda x: x/n,results)))
```

New ACT-R commands

open-exp-window and **open_exp_window**. This was introduced in the last unit. Here we see it getting passed a keyword parameter which was not done in the last unit. That parameter is a flag as to whether the window should be visible or virtual. If the visible parameter has a true value (as appropriate for the language) then a real window will be displayed, and that is the default if not provided (which is how the previous unit used it). If it is not true (again as appropriate) then a virtual window will be used and that will be demonstrated below in the section on running the experiments faster.

new-tone-sound and **new_tone_sound**. This function takes 2 required parameters and a third optional parameter. The first parameter is the frequency of a tone to be presented to the model which should be a number. The second is the duration of that tone measured in seconds. If the third parameter is specified then it indicates at what time the tone is to be presented (measured in seconds), and if it is omitted then the tone is to be presented immediately. At that requested time a tone sound will be made available to the model's auditory module with the requested frequency and duration.

schedule-event-relative and **schedule_event_relative**. This function takes 2 required parameters and several keyword parameters (only two of which will be described here). It is used to schedule ACT-R commands to be called during the running of the ACT-R system. The first parameter specifies an offset from the current ACT-R time at which the function should be called and is measured in seconds (by default). The second parameter is the name of the ACT-R command to call specified in a string. The parameters to pass to that command are provided as a list using the keyword parameter params. If no parameters are provided then no parameters are passed to that command. If one wants to set the time using milliseconds instead of seconds then the keyword parameter time-in-ms in Lisp or time_in_ms in Python needs to be specified with a true value. By scheduling commands to be called during the running of the model it is possible to have actions occur without having to stop the running model to do so which often makes writing experiments much easier, and can also make debugging a broken model/task easier because the ACT-R stepper will pause on those scheduled actions in the same way it will for other model actions.

In this experiment we are passing the current task window to the clear-exp-window command to have it cleared at that time. If you set the :trace-detail level of the model to high (not low which is how it is initially set) then you will actually see this command being executed in the trace on a line like this:

```
 0.941   NONE  clear-exp-window (vision exp-window Sperling Experiment)
```

Since this was not generated by one of the ACT-R modules it specifies "NONE" where the module name is normally shown and then it shows the command which was evaluated and the parameter it was passed, which in this case is the representation of the experiment window.

correlation. This function takes 2 required parameters which must be equal length lists of numbers. This function computes the correlation between the two lists of numbers. That correlation value is returned. There is an optional third parameter which indicates whether or not to also print the correlation value. If the optional parameter is true or not specified then it is output, and if it is not true then it does not.

mean-deviation and **mean_deviation**. This function operates just like correlation, except that the calculation performed is the root mean square deviation between the data lists.

Subitize experiment

Now we will look at the subitizing experiment code again highlighting the new commands in red.

Lisp

The first thing that the code does is load the starting model for this task from the unit3 directory of the tutorial.

```
(load-act-r-model "ACT-R:tutorial;unit3;subitize-model.lisp")
```

It defines some global variables to hold the response the participant makes, the time at which that response occurs, and the data from the original experiment for comparison.

```
(defvar *response* nil)
(defvar *response-time* nil)

(defvar *subitize-exp-data* '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58))
```

Two functions are defined to convert a number into a string which are useful for comparing a participant’s response to the correct answer. Number-to-word converts the number to the corresponding word e.g. 1 into “one”, and number-to-string converts the number to the single digit string of that number e.g. 1 into “1” (with 10 being converted to the single digit “0” since that is the key a person is to press for that answer).

```
(defun number-to-word (n)
  (format nil "~r" n))

(defun number-to-string (n)
  (if (= n 10)
      "0"
    (princ-to-string n)))
```

The subitize-trial function is defined and it takes one required parameter which is the number of items to present and an optional parameter which indicates whether a person or model is doing the task. It presents one trial to either the model or a person as indicated and returns a list of two items. The first item is the response time in seconds if the response was correct or 30 if it was not and the second item indicates whether or not the response was correct (t) or incorrect (nil).

```
(defun subitize-trial (n &optional human)
```

First it resets the system to its initial state.

```
  (reset)
```

It creates some local variables. points is set to the result of generate-points (defined below) which will be a list of x,y coordinates for the items in the window. window is set to an experiment window opened to present the trial. start is set to the current time (determined appropriately for a person or the model as described later) and answer is created but not set.

```
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"))
        (start (get-time (if human nil t)))
        answer)
```

It loops through the points placing an x in the display for each.

```
    (dolist (point points)
      (add-text-to-exp-window window "x" :x (first point) :y (second point)))
```

The global variables which will hold the response and time are cleared.

```
    (setf *response* nil)
    (setf *response-time* nil)

    (if human
```

If a person is doing the task.

```
        (progn
```

Create a command to record the person’s response using the respond-to-key-press function and then set that to monitor the output-key command.

```
          (add-act-r-command "subitize-response" 'respond-to-key-press
                             "Subitize task human response")
          (monitor-act-r-command "output-key" "subitize-response")
```

Convert the number of items in the trial to the corresponding key press string and record that in the answer variable.

```
(setf answer (number-to-string n))
```

Wait for the person to respond making sure to call process-events.

```
(while (null *response*)
  (process-events))
```

Stop monitoring output-key and remove the command we created.

```
(remove-act-r-command-monitor "output-key" "subitize-response")
(remove-act-r-command "subitize-response"))
```

If the model is performing the task.

```
(progn
```

Create a command to record the model’s response using the record-model-speech function and then set that to monitor the output-speech command.

```
(add-act-r-command "subitize-response" 'record-model-speech
                   "Subitize task model response")
(monitor-act-r-command "output-speech" "subitize-response")
```

Convert the number of items in the trial to the corresponding word string and record that in the answer variable.

```
(setf answer (number-to-word n))
```

Tell the model which window to interact with.

```
(install-device window)
```

Run the model for up to 30 seconds in real time mode.

```
(run 30 t)
```

Stop monitoring output-speech and remove the command we created.

```
(remove-act-r-command-monitor "output-speech" "subitize-response")
(remove-act-r-command "subitize-response")))
```

Compare the correct answer to the response that the person or model made and if it is correct then return a list of the difference between the *response-time* and the start time converted to seconds and t, and if it is not correct return a list of 30 and nil.

```
(if (string-equal answer *response*)
    (list (/ (- *response-time* start) 1000.0) t)
  (list 30 nil))))
```

The subitize-experiment function takes one optional parameter which indicates whether a person or model will be performing the task. It presents each of the 10 possible conditions once in a random order collecting the data, reorders the data based on the condition, and passes that data to report-data.

```
(defun subitize-experiment (&optional human)
  (let (results)
    (dolist (items (permute-list '(10 9 8 7 6 5 4 3 2 1)))
      (push (list items (subitize-trial items human)) results))

    (setf results (sort results '< :key 'car))
    (report-data (mapcar 'second results))))
```

The report-data function takes one parameter which is a list of response lists as are returned by the subitize-trial function. It prints the comparison of the response times to the experimental data and then passes the data to the print-results function to output a table of the response times and correctness.

```
(defun report-data (data)
  (let ((rts (mapcar 'first data)))
    (correlation rts *subitize-exp-data*)
    (mean-deviation rts *subitize-exp-data*)
    (print-results data)))
```

The print-results function takes one parameter which is a list of response lists as are returned by the subitize-trial function. It prints a table of the response times and correctness along with the original data.

```
(defun print-results (data)
  (format t "Items    Current Participant   Original Experiment~%")
  (dotimes (i (length data))
    (format t "~3d        ~5,2f  (~3s)              ~5,2f~%"
      (1+ i) (car (nth i data)) (second (nth i data))
      (nth i *subitize-exp-data*))))
```

The generate-points function takes 1 parameter which specifies how many points to generate. It returns a list of n randomly generated points (lists of x and y coordinates) to use for displaying the items. The points are generated such that they are not too close to each other and within the default experiment window size (300x300 pixels).

```
 (defun generate-points (n)
   (let ((points nil))
    (dotimes (i n points)
      (push (new-distinct-point points) points))))
```

The new-distinct-point function takes one parameter which is a list of points. It returns a new point that is randomly generated within the default experiment window boundary and which is not too close to any of the points on the list provided.

```
(defun new-distinct-point (points)
  (do ((new-point (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))
                  (list (+ (act-r-random 240) 20) (+ (act-r-random 240) 20))))
      ((not (too-close new-point points)) new-point)))
```

The too-close function takes two parameters. The first is a point and the second is a list of points. It returns true if the first point is within 40 pixels in either the x or y direction of any of the points on the list, otherwise it returns nil.

```
(defun too-close (new-point points)
  (some (lambda (a) (and (< (abs (- (car new-point) (car a))) 40)
                         (< (abs (- (cadr new-point) (cadr a))) 40)))
        points))
```

The respond-to-key-press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press or nil if the key was pressed by a person interacting with the window and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get-time function.

```
(defun respond-to-key-press (model key)
  (declare (ignore model))

  (setf *response-time* (get-time nil))
  (setf *response* key))
```

The record-model-speech function is set to monitor the output-speech command. That command is called whenever a model performs a speak action and is passed the name of the model and the string which the model spoke. Here we are recording the word which is spoken and the time at which that happens as determined by the get-time function.

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

  (setf *response-time* (get-time t))
  (setf *response* string))
```

Python

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

```
import actr
```

Then it loads the starting model for this task from the unit3 directory of the tutorial.

```
actr.load_act_r_model("ACT-R:tutorial;unit3;subitize-model.lisp")
```

It defines some global variables to hold the response the participant makes, the time at which that response occurs, and the data from the original experiment for comparison.

```
response = False
response_time = False

exp_data = [.6,.65,.7,.86, 1.12,1.5,1.79,2.13,2.15,2.58]
```

Two functions are defined to convert a number into a string which are useful for comparing a participant's response to the correct answer. number_to_word converts the number to the corresponding word e.g. 1 into 'one', and number_to_string converts the number to the single digit string of that number e.g. 1 into "1" (with 10 being converted to the single digit "0" since that is the key a person is to press for that answer).

```
def number_to_word (n):
    map = ['','one','two','three','four','five','six','seven','eight','nine','ten']
    return map[n]

def number_to_string (n):
    if n == 10:
        return "0"
    else:
        return str(n)
```

The trial function is defined and it takes one required parameter which is the number of items to present and an optional parameter which indicates whether a person or model is doing the task. It presents one trial to either the model or a person as indicated and returns a list of two items. The first item is the response time in seconds if the response was correct or 30 if it was not and the second item indicates whether or not the response was correct (True) or incorrect (False).

```
def trial (n,human=False):
```

First it resets the system to its initial state.

```
    actr.reset()
```

It creates some local variables. points is set to the result of generate_points (defined below) which will be a list of x,y coordinates for the items in the window. window is set to an experiment window opened to present the trial, and start is set to the current time (determined appropriately for a person or the model as described later).

```
    points = generate_points(n)
    window = actr.open_exp_window("Subitizing Experiment")
    start = actr.get_time(not(human))
```

It loops through the points placing an x in the display for each.

```
for point in points:
    actr.add_text_to_exp_window(window, "x", x=point[0], y=point[1])
```

The global variables which will hold the response and time are cleared.

```
global response,response_time

response = ''
response_time = False

if human:
```

If a person is doing the task.

Create a command to record the person’s response using the respond_to_key_press function and then set that to monitor the output-key command.

```
    actr.add_command("subitize-response",respond_to_key_press,
                     "Subitize task human response")
    actr.monitor_command("output-key","subitize-response")
```

Convert the number of items in the trial to the corresponding key press string and record that in the answer variable.

```
    answer = number_to_string(n)
```

Wait for the person to respond making sure to call process-events.

```
    while response == '':
        actr.process_events()
```

Stop monitoring output-key and remove the command we created.

```
    actr.remove_command_monitor("output-key","subitize-response")
    actr.remove_command("subitize-response")

else:
```

If the model is performing the task.

Create a command to record the model’s response using the record_model_speech function and then set that to monitor the output-speech command.

```
    actr.add_command("subitize-response",record_model_speech,
                             "Subitize task model response")
        actr.monitor_command("output-speech","subitize-response")
```

Convert the number of items in the trial to the corresponding word string and record that in the answer variable.

```
        answer = number_to_word(n)
```

Tell the model which window to interact with.

```
        actr.install_device(window)
```

Run the model for up to 30 seconds in real time mode.

```
        actr.run(30,True)
```

Stop monitoring output-speech and remove the command we created.

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

Compare the correct answer to the response that the person or model made and if it is correct then return a list of the difference between the response_time and the start time converted to seconds and True, and if it is not correct return a list of 30 and False.

```
    if response.lower() == answer.lower():
        return [(response_time - start) / 1000.0, True]
    else:
        return [30, False]
```

The experiment function takes one optional parameter which indicates whether a person or model will be performing the task. It presents each of the 10 possible conditions once in a random order collecting the data, reorders the data based on the condition, and passes that data to report_data.

```
def experiment(human=False):

    results = []
    for items in actr.permute_list([10,9,8,7,6,5,4,3,2,1]):
      results.append((items,trial(items,human)))

    results.sort()
    report_data(list(map(lambda x: x[1],results)))
```

The report_data function takes one parameter which is a list of response lists as are returned by the subitize_trial function. It prints the comparison of the response times to the experimental data and then passes the data to the print_results function to output a table of the response times and correctness.

```
def report_data(data):

    rts = list(map(lambda x: x[0],data))
    actr.correlation(rts,exp_data)
    actr.mean_deviation(rts,exp_data)
    print_results(data)
```

The print_results function takes one parameter which is a list of response lists as are returned by the subitize_trial function. It prints a table of the response times and correctness along with the original data.

```
def print_results(data):

    print("Items    Current Participant   Original Experiment")
    for count, d, original in zip(range(len(data)),data,exp_data):
        print("%3d        %5.2f  (%-5s)            %5.2f" %
              (count+1,d[0],d[1],original))
```

The generate_points function takes 1 parameter which specifies how many points to generate. It returns a list of n randomly generated points (lists of x and y coordinates) to use for displaying the items. The points are generated such that they are not too close to each other and within the default experiment window size (300x300 pixels).

```
def generate_points(n):
    p=[]
    for i in range(n):
        p.append(new_distinct_point(p))
    return p
```

The new_distinct_point function takes one parameter which is a list of points. It returns a new point that is randomly generated within the default experiment window boundary and which is not too close to any of the points on the list provided.

```
def new_distinct_point(points):

    while True:
        x = actr.random(240)+20
        y = actr.random(240)+20
        if not(any(too_close(x,y,p) for p in points)):
            break;
    return [x,y]
```

The too_close function takes three parameters. The first and second are the x and y positions for a proposed point and the second is a list of points. It returns True if the proposed point would be within 40 pixels in either the x or y direction of any of the points on the list, otherwise it returns False.

```
def too_close (x,y,p):
    if (abs(x-p[0]) < 40) and (abs(y-p[1]) < 40):
        return True
    else:
        return False
```

The respond_to_key_press function is set to monitor the output-key command and therefore will be called whenever a key is pressed in the experiment window. It will be passed the name of the model which made the key press or None if the key was pressed by a person interacting with the window and the string which names the key. Here we are recording the key which is pressed and the time at which that happens as determined by the get_time function.

```
def respond_to_key_press (model,key):
    global response,response_time

    response_time = actr.get_time(False)
    response = key
```

The record_model_speech function is set to monitor the output-speech command. That command is called whenever a model performs a speak action and is passed the name of the model and the string which the model spoke. Here we are recording the word which is spoken and the time at which that happens as determined by the get_time function.

```
def record_model_speech (model,string):
    global response,response_time

    response_time = actr.get_time(True)
    response = string
```

New commands

get-time and **get_time**. This function takes an optional parameter and it returns the current time in milliseconds. If the optional parameter is not specified or specified as a true value, then the current ACT-R simulated time is returned. If the optional parameter is specified as a non-true value then the time is taken from a real time clock.

output-speech. The output-speech command in ACT-R is very similar to the output-key command which we used in the previous unit. It is called automatically whenever a model performs a speak action using the **vocal** buffer by a microphone device which is installed automatically when the AGI window device is installed. The command is executed at the time when the sound of that speech starts to occur i.e. it is essentially the same time that would be recorded if we were using a microphone to detect a person’s speech output. It is passed two values which are the name of the model which is speaking and the string containing the text being spoken.

Response recording note

As was mentioned in the last unit the monitoring functions are called in a separate thread from the main task execution. As will be the case throughout the tutorial we are not using any special protection for accessing the globally defined variables in the different threads to keep the example code simple, but this time there is actually the possibility for a problem since we are setting two different variables in the monitoring function and both are used in the main thread. Without protecting them it is possible for the code in the main thread to try and use them both after only one has been set which could result in an error. To avoid that here we set the response-time variable first since the main thread is waiting for the response variable to change before it tries to use the response-time value. That is sufficient to avoid problems in this task (we could have also waited for both to be set before continuing as an alternative), but the best solution would really be to use appropriate thread protection tools.

Buffer stuffing

The buffer stuffing mechanism was introduced in this unit, and with regard to the **visual-location** buffer it mentioned that one can change the default conditions that are checked to determine which item (if any) will be stuffed into the buffer. The ACT-R command which can be used to change that is called **set-visloc-default.** The specification that you pass to it is the same as you would specify in a request to the **visual-location** buffer in a production. Here are a few examples:

```
(set-visloc-default :attended new screen-x lowest)

(set-visloc-default screen-x current > screen-y 100)

(set-visloc-default kind text color red width highest)
```

set-visloc-default – This command sets the conditions that will be used to select the visual feature that gets stuffed into the **visual-location** buffer. When the visual scene changes for the model (denoted by the proc-display event which is only shown when the :trace-detail parameter is set to high) if the **visual-location** buffer is empty a visual-location that matches the conditions specified by this command will be placed into the **visual-location** buffer. Effectively, what happens is that when the proc-display event occurs, if the **visual-location** buffer is empty, a **visual-location** request is automatically executed using the specification indicated with set-visloc-default.

Speeding up the experiments

Because it is often necessary to run a model multiple times and average the results for data comparison, running the model quickly can be important. One thing that can greatly improve the time it takes to run the model (i.e. the real time that passes not the simulated time which the model experiences) is to have it interact with a virtual window instead of displaying a real window on the computer. The virtual windows are an abstraction of a window interface that is internal to ACT-R, and from the model's perspective there is no difference between a virtual window and a real window which is generated by the AGI commands. Another significant factor with respect to how long it takes is whether or not the model is being run in real time mode. In real time mode the model's actions are synchronized with the actual passing of time, but when it is not in real time mode it uses its own simulated clock which can run much faster than real time. Typically, when the model is running with a real window it is also running in real time so that one can actually watch it performing the task.

Since it is usually very helpful to use a real window when creating a model for a task and debugging any problems which occur when running it all of the AGI tools for building and manipulating windows work exactly the same for real windows and virtual windows. All that is necessary to switch between them is one parameter when the window is opened. Thus, you can build the experiment and debug the model using a real window that you can see, and then with one change make the window virtual and run the model more quickly for data collection.

The experiment code as provided for both of the tasks in this unit uses a real window and the model is run in real time mode so that you can watch it interact with that window.

To change the windows in those tasks to virtual windows requires changing the call to open the window to specify the visible parameter as not true (**nil** in Lisp and **False** in Python).

Here is the initial code from the sperling tasks:

Lisp

```
(let* (...
       (window (open-exp-window "Sperling Experiment" :visible t))
```

Python

```
    window = actr.open_exp_window("Sperling Experiment", visible=True)
```

Here is what would need to be changed to make those virtual windows instead:

Lisp

```
(let* (...
       (window (open-exp-window "Sperling Experiment" :visible nil))
```

Python

```
    window = actr.open_exp_window("Sperling Experiment", visible=False)
```

A similar change could be made in the subitize experiment.

To change the code to run the models in simulated time requires not specifying the second parameter as true in the call to run.

Here is the call that currently exists in both experiments:

Lisp

```
(run 30 t)
```

Python

```
actr.run(30,True)
```

Removing that second parameter will run the models in simulated time:

Lisp

```
(run 30)
```

Python

```
actr.run(30)
```

Something else which can improve the time which it takes to run a task is to turn off any output which it generates. As was mentioned in the unit you can turn the model trace off by setting the :v parameter to **nil**. If there is any other output in the experiment code as the task is running you will also want to disable that.

For the sperling experiment the given code prints out the correct answers and model responses for each trial, and turning that off will help if you want to run a lot of trials to see the average performance. In that code we have already added a variable which is used as a flag to control that output. The global variables *show-responses* and show_responses control whether that information is printed. In the given code they are set to true values:

Lisp

```
(defparameter *show-responses* t)
```

Python

```
show_responses = True
```

Change those to non-true values will eliminate that output:

Lisp

```
(defparameter *show-responses* nil)
```

Python

```
show_responses = False
```

Adding a variable to control whether any output from the experiment is printed can be a handy thing to add when creating tasks for models that you want to run many times.

One last thing to note is that if you change the code in the experiment file then you will need to load that file again to have the changes take effect. For the Lisp versions of the task you can simply load the file again to have the changes take effect, but for Python importing the module again in the same session will not work. One way to deal with that is to also import the importlib module. That provides a reload function which can be used to force the module to be reimported to reflect the changes and here is an example of using that to load the sperling module again after changing it:

```
>>> import importlib
>>> importlib.reload(sperling)
```