

The experiments in this unit are more involved than those of the last unit. The experiments are more complicated, and they involve collecting data and comparing it to existing experimental results. Many of the functions that were introduced in the last unit will be seen again (some of them always have to be there, like `pm-run`) and there will be a few more introduced.

Because it is necessary to run the model multiple times and average the results for data comparison speed is important. One thing that can greatly improve the speed of the model is to let it use a virtual window instead of interacting with a real window on the computer. Virtual windows are an abstraction of a window interface that is internal to ACT-R/PM. From the model's perspective there is no difference between a virtual window and a real window, but there is much less overhead involved with virtual windows because they exist entirely within Lisp. The only problem is that you cannot see a virtual window, so you cannot see what the model is looking at which can make debugging a model difficult. To help with that, the tools that were introduced in the last unit 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 with a real window and debug the model using a window that you can see. Then with one small change make the window virtual and run the model quickly over many runs.

Here is the experiment code from the Sperling model.

Misc window:

```
(defvar *responses* nil)
(defvar *show-responses* nil)
(defvar *done* nil)

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

(defun do-trial (onset-time)
  (if *actr-enabled-p*
      (do-trial-model onset-time)
      (do-trial-person onset-time)))

(defun do-trial-model (onset-time)
  (let* ((lis (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)
        (tone (random 3))
        (window (open-exp-window "Sperling Experiment"
                                :visible t
                                :width 300
                                :height 300)))
    (dotimes (i 3)
      (dotimes (j 4)
        (let ((txt (nth (+ j (* i 4)) lis)))
          (when (= i tone)
            (push txt answers))
          (add-text-to-exp-window :text txt
                                :width 40)))))
```

```

                                :x (+ 75 (* j 50))
                                :y (+ 101 (* i 50))))))

(reset)

(pm-install-device window)

(new-tone-sound (case tone (0 2000) (1 1000) (2 500)) .5 onset-time)

(pm-timed-event (+ .9 (random .2)) #'clear-screen)

(pm-proc-display)

(setf *responses* nil)

(pm-run 30)

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

(compute-score answers)))

(defun do-trial-person (onset-time)
  (let* ((lis (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)
    (tone (random 3))
    (window (open-exp-window "Sperling Experiment"
                             :visible t
                             :width 300
                             :height 300)))

  (dotimes (i 3)
    (dotimes (j 4)
      (let ((txt (nth (+ j (* i 4)) lis)))
        (when (= i tone)
          (push txt answers))
          (add-text-to-exp-window :text txt
                                  :width 40
                                  :x (+ 75 (* j 50))
                                  :y (+ 101 (* i 50)))))))

  (sleep onset-time)

  (if (fboundp 'beep)
      (dotimes (i (1+ tone)) (beep))
      (format t "Cannot generate sounds.~%Recall row ~S~%" (1+ tone)))

  (sleep (- 1.0 onset-time))

  (setf *done* nil)
  (setf *responses* nil)
  (clear-exp-window)
  (sgp :v nil)

  (while (null *done*)
    (allow-event-manager window))

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

```

```

(compute-score answers)))

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

(defun clear-screen ()
  (clear-exp-window)
  (pm-proc-display))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (if (string= key " ")
      (setf *done* t)
      (push (string key) *responses*)))

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

(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)))))

(defun run-block ()
  (let ((times (permute-list '(0.0 .15 .30 1.0)))
        (result nil))
    (dolist (x times)
      (push (cons x (do-trial x)) result))
    (sort result #'< :key #'car)))

(defun repeat-experiment (n)
  (let ((results (list 0 0 0 0)))
    (dotimes (i n)
      (setf results (mapcar #'+ results (mapcar #'cdr (run-block)))))
    (report-data (mapcar #'(lambda (x) (/ x n)) results))))

```

Commands window:

```

(sgp :v t :esc t)

(pm-set-params :show-focus t :real-time t)

(setf *actr-enabled-p* t)

(setf *show-responses* t)

(goal-focus goal)

(spp start-report :c 2)

```

```
(spp detected-sound :c 0)
(spp sound-respond-low :c 0)
(spp sound-respond-medium :c 0)
(spp sound-respond-high :c 0)
```

First, we will look at the code in the commands window.

The verbose flag is turned on so that the trace is displayed and enable subsymbolic computations is turned on so that the production parameters set later have an impact.

```
(sgp :v t :esc t)
```

The system is set to show the focus ring and to run in real time.

```
(pm-set-params :show-focus t :real-time t)
```

The model is set to run the task.

```
(setf *actr-enabled-p* t)
```

This next line sets a global variable that is used in the experiment code. If the variable is set to t then the correct answers and the participants responses are printed after every trial.

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

The chunk named goal is placed into the goal buffer.

```
(goal-focus goal)
```

The next several commands are setting the cost value of some of the productions to impose a priority on them. The default cost for a production is .05 seconds, and a production with a lower cost is selected over one with a higher cost. Thus, start-report is given a lower priority than the default and the sound handling productions are given a higher priority.

```
(spp start-report :c 2)
(spp detected-sound :c 0)
(spp sound-respond-low :c 0)
(spp sound-respond-medium :c 0)
(spp sound-respond-high :c 0)
```

Now we will look at the misc window. The new interface functions are displayed in red.

Three global variables are defined. *responses* will hold the list of keys pressed by the participant, *show-response* is used as a flag to indicate whether or not to print out the participant's responses every trial, and *done* is used to signal when a person has finished a trial.

```
(defvar *responses* nil)
(defvar *show-responses* nil)
(defvar *done* nil)
```

Next we define a global constant that holds the results of the original experiment so that the model's performance can be compared to it.

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

The `do-trial` function takes one parameter which is the delay time at which to present the auditory cue in seconds. It then calls the appropriate function depending on whether a model or a person is doing the task. It returns the number of letters correctly recalled.

```
(defun do-trial (onset-time)
  (if *actr-enabled-p*
      (do-trial-model onset-time)
      (do-trial-person onset-time)))
```

The `do-trial-model` function presents one trial of the sperling task to a model with the auditory cue occurring at the time passed as the parameter. It returns the number of items correctly recalled from the target row.

```
(defun do-trial-model (onset-time)
```

First it randomizes a list of letters

```
(let* ((lis (permute-list '("B" "C" "D" "F" "G" "H" "J"
                           "K" "L" "M" "N" "P" "Q" "R"
                           "S" "T" "V" "W" "X" "Y" "Z"))))
```

creates a variable to hold the list target letters

```
(answers nil)
```

randomly chooses which row to be the target

```
(tone (random 3))
```

and opens a window to do the task. The keyword parameters used are new in this unit and will be described below in detail.

```
(window (open-exp-window "Sperling Experiment"
                          :visible t
                          :width 300
                          :height 300)))
```

Then it displays 3 rows of 4 letters recording the letters that are in the target row in the `answers` variable.

```
(dotimes (i 3)
  (dotimes (j 4)
    (let ((txt (nth (+ j (* i 4)) lis)))
      (when (= i tone)
        (push txt answers))
      (add-text-to-exp-window :text txt
                              :width 40
                              :x (+ 75 (* j 50))
                              :y (+ 101 (* i 50))))))
```

The model is reset and it is told which window to interact with

```
(reset)

(pm-install-device window)
```

This call schedules a tone of a particular frequency to be played for the model at the desired onset time. The frequency is determined by which row is to be recalled.

```
(new-tone-sound (case tone (0 2000) (1 1000) (2 500)) .5 onset-time)
```

At a random time between .9 and 1.1 seconds the clear-screen function will be called (it is defined below).

```
(pm-timed-event (+ .9 (random .2)) #'clear-screen)
```

The model is told to process the display

```
(pm-proc-display)
```

The variable to hold the model's responses is cleared

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

The model is run for up to 30 seconds

```
(pm-run 30)
```

If the `*show-responses*` variable is set it prints out the correct answers and the model's responses

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

Then it calls the compute-score function to return the number of correct responses (it is defined below).

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

The `do-trial-person` function is similar to the `do-trial-model` one, so only the differences will be described.

```
(defun do-trial-person (onset-time)
  (let* ((lis (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)
    (tone (random 3))
    (window (open-exp-window "Sperling Experiment"
                             :visible t
                             :width 300
                             :height 300)))

  (dotimes (i 3)
    (dotimes (j 4)
      (let ((txt (nth (+ j (* i 4)) lis)))
        (when (= i tone)
          (push txt answers))
```

```
(add-text-to-exp-window :text txt
                       :width 40
                       :x (+ 75 (* j 50))
                       :y (+ 101 (* i 50))))
```

After displaying the letters wait for the onset time to pass

```
(sleep onset-time)
```

Then play the required number of beeps to indicate which row to recall if the beep function is defined in the Lisp being used, otherwise just print it out

```
(if (fboundp 'beep)
    (dotimes (i (1+ tone)) (beep))
    (format t "Cannot generate sounds.~%Recall row ~S~%" (1+ tone)))
```

Wait until a total of 1 second has elapsed since the display appeared

```
(sleep (- 1.0 onset-time))
```

Clear the done flag and the responses list

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

erase the contents of the window

```
(clear-exp-window)
```

turn off the tracing so that the key press information is not shown for a person

```
(sgp :v nil)
```

wait for the person to finish entering keys (indicated by pressing space)

```
(while (null *done*)
  (allow-event-manager window))

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

(compute-score answers)))
```

The compute-score function takes one parameter which is a list of the letters in the target row. It returns the number of those items that were given by the participant.

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

The clear-screen function is called when the model is running at the appropriate time. It clears the screen and makes the model reprocess it.

```
(defun clear-screen ())
```

```
(clear-exp-window)
(pm-proc-display))
```

The `rpm-window-key-event-handler` gets called automatically when a key is pressed and is passed the window in which the press occurred and the key that was pressed. For this task it places the keys onto the `*responses*` list and sets the `*done*` flag when the space bar is pressed.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (if (string= key " ")
      (setf *done* t)
      (push (string key) *responses*)))
```

The `report-data` function takes one parameter which is a list that should represent average data of participants in the task. Those data are compared to the original experimental data and the correlation, mean deviation, and a table of the results are printed.

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

The `print-results` function takes one parameter which should be a list containing average data of participants in the task. Those data are printed in a table with the onset times and the original data.

```
(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 `run-block` function takes no parameters. It runs one trial of the task at each of the 4 onset times randomly ordered. It returns a list of cons ordered by onset time, with the car of each cons being the onset time and the cdr being the number of correct responses.

```
(defun run-block ()
  (let ((times (permute-list '(0.0 .15 .30 1.0)))
        (result nil))
    (dolist (x times)
      (push (cons x (do-trial x)) result))
    (sort result #'< :key #'car)))
```

The `repeat-experiment` function takes one parameter which is the number of blocks of the experiment to run. A block is one trial at each of the 4 onset times. The results of those blocks are averaged together and then the comparison of that average data to the original experimental data is printed.

```
(defun repeat-experiment (n)
  (let ((results (list 0 0 0 0)))
    (dotimes (i n)
      (setf results (mapcar #' + results (mapcar #'cdr (run-block))))))
  (report-data (mapcar #'(lambda (x) (/ x n)) results)))
```

The new functions that are used in this model are:

Open-exp-window – this was introduced in the last unit. Here we see it getting passed keyword parameters that were not used previously. `:height` and `:width` specify the size of the window in pixels. The `:visible` parameter is the flag that determines whether a real or a virtual window is used. If `:visible` is `t` (the default value if it is not specified) then a real window is used. If `:visible` is `nil`, then a virtual window is used. There are two other parameters, `:x` and `:y` which can be used to specify the position of the upper left corner of the window on the main display.

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. The second is the duration of that tone in seconds. If the third parameter is specified then it indicates at what time the tone is to be presented, and if it is omitted then the tone is to be presented immediately. At the requested time a tone sound will be made available to the model's auditory module of the requested frequency and duration.

Pm-timed-event – This function takes 2 required parameters and any number of additional parameters. It is used to schedule functions to be called during the running of the model. The first parameter specifies the time at which the function should be called. The second parameter is the function to call. Any additional parameters specified are passed to that function when it is called. By scheduling functions to be called during the running of the model it is possible to have the experiment change without stopping the model to do so. The alternative is to use the `:full-time` parameter of `pm-run` to run the model for the desired amount of time, when it stops execute the function call, and then run the model again.

Sleep – `sleep` is actually a function defined in ANSI Common Lisp, but because it is being used for experiment generation it seemed appropriate to discuss it. The Lisp specification for `sleep` says it takes one parameter, *seconds*, which is a non-negative real, and it causes execution to cease and become dormant for approximately the seconds of real time indicated by *seconds*, whereupon execution is resumed. This is only useful for a person doing a task because it has no impact upon a model.

Correlation – this function takes 2 required parameters which must be equal length 'collections' of numbers. The numbers can be in arrays or lists and they do not both need to be in the same format (one could be an array and the other a list). The only requirement is that they have the same number of numbers in them. This function then extracts those numbers and computes the correlation between the two sets of numbers. That correlation value is returned. There is a keyword parameter `:output` which defaults to `t`. When `:output` is `t` the correlation is printed to `*standard-output*`. If `:output` is a string, stream, or pathname then it is used to open a stream to which the results are written.

Mean-deviation – this function operates just like correlation, except that the calculation performed is the mean deviation between the data sets.

Now we will look at the subitizing experiment code.

Misc window

```
(defvar *response* nil)
(defvar *response-time* nil)
(defconstant *subitizing-exp-data*
  '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58))

(defun do-trial (n)
  (if *actr-enabled-p*
      (do-trial-model n)
      (do-trial-person n)))

(defun do-trial-model (n)
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"
                                :visible t
                                :width 300
                                :height 300
                                :x 300
                                :y 300)))

    (dolist (point points)
      (add-text-to-exp-window :text "x"
                             :width 10
                             :x (first point)
                             :y (second point)))

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

    (reset)
    (pm-install-device window)
    (pm-proc-display)
    (pm-run 30.0)

    (let ((response (if *response* (read-from-string *response*) -1)))
      (list (if (null *response-time*) 30.0 (/ *response-time* 1000.0))
            (or (= response n)
                (and (= n 10) (= response 0)))))))

(defun do-trial-person (n)
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"
                                :visible t
                                :width 300
                                :height 300
                                :x 300
                                :y 300)))

    (dolist (point points)
      (add-text-to-exp-window :text "x"
                             :width 10
                             :x (first point)
                             :y (second point))))
```

```

(setf *response* nil)
(setf *response-time* nil)
(sgp :v nil)

(let ((start-time (pm-get-time)))
  (while (null *response*)
    (allow-event-manager window)
    (setf *response-time* (- *response-time* start-time))))

(let ((response (if *response* (read-from-string *response*) -1)))
  (list (if (null *response-time*) 30.0 (/ *response-time* 1000.0))
        (or (= response n)
              (and (= n 10) (= response 0))))))

(defun experiment ()
  (let (result)
    (dolist (items (permute-list '(10 9 8 7 6 5 4 3 2 1)))
      (push (list items (do-trial items)) result))
    (report-data (mapcar 'second (sort result '< :key 'car)))))

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

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

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

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

(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))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response-time* (pm-get-time))
  (setf *response* (string key)))

(defun record-vocal-response (time text)
  (setf *response-time* (pm-get-time))
  (setf *response* text))

```

Commands window

```
(sgp :v t)
```

```
(pm-set-params :real-time t :show-focus t
              :visual-num-finsts 10 :visual-finst-span 10
              :speech-hook-fct #'record-vocal-response)

(setf *actr-enabled-p* nil)

(goal-focus goal)
```

Starting with the commands window, there is only one call that is any different than what has been seen previously.

```
(pm-set-params :real-time t :show-focus t
              :visual-num-finsts 10 :visual-finst-span 10
              :speech-hook-fct #'record-vocal-response)
```

The ACT-R/PM parameters for the number of finsts and the duration of finsts are set “large enough” so that the model only needs to use the attended flag to exhaustively search the window. The last parameter set there, `:speech-hook-fct`, specifies a function to call when the model “speaks” – remember the model will be responding verbally.

Now we will look at the functions in the misc window. There is only one new interface function used here.

First we define some global variables to hold the response and the time of that response.

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

Then we create a global constant that holds the data from the original experiment so that the model’s performance can be compared to it.

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

The `do-trial` function takes one parameter which is the number of items to present. It then calls the appropriate function depending on whether a person or the model is doing the task. The return value is a list of two items. The first item is the response time and the second item indicates whether or not the response was correct (t) or incorrect (nil).

```
(defun do-trial (n)
  (if *actr-enabled-p*
      (do-trial-model n)
      (do-trial-person n)))
```

The `do-trial-model` function takes one parameter which is the number of items to display and presents them for the model and returns a list of two items. The first item is the response time and the second item indicates whether or not the response was correct (t) or incorrect (nil).

```
(defun do-trial-model (n)
```

First it builds a list of n points and creates a window

```
(let ((points (generate-points n))
      (window (open-exp-window "Subitizing Experiment"
                               :visible t
                               :width 300
                               :height 300
                               :x 300
                               :y 300)))
```

Then it draws an x at each of those random points

```
(dolist (point points)
  (add-text-to-exp-window :text "x"
                          :width 10
                          :x (first point)
                          :y (second point)))
```

It clears the response variables

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

Resets the model, tells it which window to look at, makes it process the display, and then run for up to 30 seconds

```
(reset)
(pm-install-device window)
(pm-proc-display)
(pm-run 30.0)
```

Here the return list is generated. The response is checked for correctness and the time is converted to seconds, or set to 30 seconds if there was no response.

```
(let ((response (if *response* (read-from-string *response*) -1)))
  (list (if (null *response-time*) 30.0 (/ *response-time* 1000.0))
        (or (= response n)
              (and (= n 10) (= response 0))))))
```

Do-trial-person operates just like do-trial-model, except it waits for a key press from a person.

```
(defun do-trial-person (n)
  (let ((points (generate-points n))
        (window (open-exp-window "Subitizing Experiment"
                                   :visible t
                                   :width 300
                                   :height 300
                                   :x 300
                                   :y 300)))

    (dolist (point points)
      (add-text-to-exp-window :text "x"
                              :width 10
                              :x (first point)
```

```

                                :y (second point)))
(setf *response* nil)
(setf *response-time* nil)
(sgp :v nil)

(let ((start-time (pm-get-time)))
  (while (null *response*)
    (allow-event-manager window)
    (setf *response-time* (- *response-time* start-time))))

(let ((response (if *response* (read-from-string *response*) -1)))
  (list (if (null *response-time*) 30.0 (/ *response-time* 1000.0))
        (or (= response n)
              (and (= n 10) (= response 0))))))

```

The experiment function takes no parameters. It presents each of the 10 possible conditions once in random order collecting the data. It then prints out the data and the comparison to the experimental results.

```

(defun experiment ()
  (let (result)
    (dolist (items (permute-list '(10 9 8 7 6 5 4 3 2 1)))
      (push (list items (do-trial items)) result))
    (report-data (mapcar 'second (sort result '< :key 'car)))))

```

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

```

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

```

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

```

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

```

(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 that is not too close to any of the points on the list provided.

```
(defun new-distinct-point (points)
  (do ((new-point (list (+ (random 240) 20) (+ (random 240) 20)))
      (list (+ (random 240) 20) (+ (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 `t` 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 `rpm-window-key-event-handler` is called automatically when a key is pressed. It is recording the time of the key press and the key that was pressed. In this task only a person is pressing keys, the model is “speaking”. There is one important thing to note about this function. The `do-trial-person` function is looping until the `*response*` variable is set. Because this function gets called by the system asynchronously and not atomically it is important to set the variable that is being used as a flag last. Otherwise the `do-trial-person` function may attempt to use the `*response-time*` variable before it gets set.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (setf *response-time* (pm-get-time))
  (setf *response* (string key)))
```

The `record-vocal-response` function takes two parameters. Those are the current model time and the text the model is speaking. This function was set as the one to be called by ACT-R/PM when the model speaks. It sets the global variables to record the response and when it occurred. It could just use the time that is passed as a parameter, but `pm-get-time` is used to show that it can be used when either a person or model is doing the task – this function looks almost exactly like the previous one.

```
(defun record-vocal-response (time text)
  (setf *response-time* (pm-get-time))
  (setf *response* text))
```

pm-get-time – This function takes no parameters. It returns the current time in milliseconds. If the model is doing the task (when `*actr-enabled-p*` is `t`) the time is taken from the simulation time and if a person is doing the task (when `*actr-enabled-p*` is `nil`) the time is taken from the internal timer.

Buffer stuffing

Although it was not used in the models, the buffer stuffing mechanism was introduced in this unit text. It mentioned that one can change the default conditions for which item can be stuffed into the buffer. The function which does that is called **pm-set-visloc-default**. It takes keywords that specify the slots to test and the value to test for as would be used in a production. Here are a couple of examples:

```
(pm-set-visloc-default :attended new :screen-x lowest)
(pm-set-visloc-default :screen-x current :screen-y (within 100 230))
(pm-set-visloc-default :kind text :color red)
```

Pm-set-visloc-default – This command sets the conditions that will be used to select the visual location that gets buffer stuffed. When the screen is processed by the model (pm-proc-display is called) 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. Essentially, what happens is when pm-proc-display gets called if the visual-location buffer is empty a +visual-location request is automatically executed using the slot tests set with pm-set-visloc-default.