

There are only 2 new commands used in the models for this unit and they were discussed in the unit text. So there is not really anything new to discuss about the functions used to run the models. Instead, what will be described in this document is a different way of writing experiments for models than you have seen previously.

So far you have seen what is best described as a “trial at a time” or iterative approach to the experiments. The experiments run by executing some setup code, running the model to completion on that trial, recording a result and then repeating that process for the next trial. That style is a commonly used approach, but it has some drawbacks which you may have encountered. For instance, when debugging the model using the stepper it is not possible to stop the experiment. To stop the experiment you have to break out of the execution of the experiment function – the stop button of the stepper only terminates a particular trial. For models with few trials or that get reset on each trial that may not be too big of a problem, but for large experiments or models that need to learn from trial to trial that can make things difficult to work with, particularly if there is a problem with the model on a later trial that forces one to abandon a very long run.

An alternative way to write the experiments is with a more event-driven approach. The scheduling mechanisms that run the model are implemented as a general event dispatching system. Calling pm-run causes the events generated by the model, both internal like production firing and memory retrieval and external like key presses and mouse clicks, to be executed in either a simulated time or real time sequence. We have also seen in the sperling task that it is possible to schedule arbitrary events (like the tone to respond in that task) to occur at particular times. One can use the events generated by the model (key presses, mouse clicks, etc) as well as explicitly scheduled events to have an experiment that runs “with” the model instead of “around” it. Such an experiment only needs to call the pm-run function one time to run the whole experiment instead of once (or more) per trial and that can make some things easier.

With the model running in an event-driven experiment allows for more interactive control of the task as a whole. The stepper’s stop button will now stop the whole experiment. That allows one to see exactly what is happening without having to abort the experiment function. To continue after stopping all one needs to do is then call pm-run again to have the model and the experiment continue from where they left off. It can also make the model writing easier because one doesn’t have to make sure that the model “stops” when it should, but just that it can respond to the events that occur. The model then completes the task as a continuous process instead of many discrete situations.

The demonstration model, the paired associate task, is written using the iterative approach. The assignment model, the Zbrodoff task, is written with an event based experiment.

First the code from the **Misc Window** of the **Paired** model:

```

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

(defvar *pairs* '(("bank" "0") ("card" "1") ("dart" "2") ("face" "3")
                  ("game" "4") ("hand" "5") ("jack" "6") ("king" "7")
                  ("lamb" "8") ("mask" "9") ("neck" "0") ("pipe" "1")
                  ("quip" "2") ("rope" "3") ("sock" "4") ("tent" "5")
                  ("vent" "6") ("wall" "7") ("xray" "8") ("zinc" "9")))

(defconstant *paired-latencies* '(0.0 2.158 1.967 1.762 1.68 1.552 1.467 1.402))
(defconstant *paired-probability* '(0.000 .526 .667 .798 .887 .924 .958 .954))

(defun do-experiment (size trials)
  (if *actr-enabled-p*
      (do-experiment-model size trials)
      (do-experiment-person size trials)))

(defun do-experiment-model (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible t)))

    (reset)
    (pm-install-device window)

    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))

          (clear-exp-window)
          (add-text-to-exp-window :text (car x) :x 150 :y 150)

          (setf *response* nil)
          (setf *response-time* nil)
          (setf start-time (pm-get-time))

          (pm-proc-display)
          (pm-run 5.0 :full-time t)

          (when (equal (second x) *response*)
            (incf score 1.0)
            (incf time (- *response-time* start-time)))

          (clear-exp-window)
          (add-text-to-exp-window :text (second x) :x 150 :y 150)

          (pm-proc-display)
          (pm-run 5.0 :full-time t))

        (push (list (/ score size) (and (> score 0) (/ time (* score 1000.0))))
              result)))

    (reverse result)))

(defun do-experiment-person (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible t)))

    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))

          (clear-exp-window)

```

```

      (add-text-to-exp-window :text (car x) :x 150 :y 150 :width 50)
      (setf *response* nil)
      (setf *response-time* nil)

      (setf start-time (pm-get-time))
      (while (< (- (pm-get-time) start-time) 5000)
        (allow-event-manager window))

      (when (equal (second x) *response*)
        (incf score 1.0)
        (incf time (/ (- *response-time* start-time) 1000.0)))

      (clear-exp-window)
      (add-text-to-exp-window :text (second x) :x 150 :y 150)
      (sleep 5.0))

      (push (list (/ score size) (and (> score 0) (/ time score))) result)))

;; return the list of scores
(reverse result)))

(defun collect-data (n)
  (do ((count 1 (1+ count))
      (results (do-experiment 20 8)
              (mapcar #'(lambda (lis1 lis2)
                          (list (+ (first lis1) (first lis2))
                                (+ (or (second lis1) 0)
                                    (or (second lis2) 0))))
                      results (do-experiment 20 8))))
      ((equal count n)
       (output-data results n))))

(defun output-data (data n)
  (let ((probability (mapcar #'(lambda (x) (/ (first x) n)) data))
        (latency (mapcar #'(lambda (x) (/ (or (second x) 0) n)) data)))
    (print-results latency *paired-latencies* "Latency")
    (print-results probability *paired-probability* "Accuracy")))

(defun print-results (predicted data label)
  (format t "~%~%~A:~%" label)
  (correlation predicted data)
  (mean-deviation predicted data)
  (format t "Trial    1      2      3      4      5      6      7
8~%"
    (format t "      ~{~8,3f~}~%" predicted)))

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

```

and then the code from the **Commands Window**:

```

(sgp :rt -2 :lf 0.35 :ans 0.5 :bll 0.5 :ga 0 :pm nil :esc t :v t)

(setf *actr-enabled-p* t)

(goal-focus goal)

```

The call to `sgp` has a few new parameters.

```

(sgp :rt -2 :lf 0.35 :ans 0.5 :bll 0.5 :ga 0 :pm nil :esc t :v t)

```

:rt is the retrieval threshold which sets minimum activation a chunk can have and still be retrieved. :lf is the latency factor and is the F value in the retrieval time equation. :ans is the s value for the transient activation noise. :bll is the base level learning parameter and it is used to both enable base level learning (when it is non-nil) and to specify the d parameter for the base level learning equation. The recommended value when enabling it is .5. That value has worked in a vast number of models, and unless you have a good reason for using a different value it should not be used as a free parameter when fitting data. The :ga and :pm parameters will be described in the next unit. Here they are set to remove the effects of context on activation so that the model demonstrates only the effects of base-level learning on activation.

The rest of the code there should be very familiar by now.

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

Now we will look at the code from the Misc window. First we define the global variables to hold the key response and time.

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

Then we define a list of the stimuli to be presented in the task.

```
(defvar *pairs* '(("bank" "0") ("card" "1") ("dart" "2") ("face" "3")
                 ("game" "4") ("hand" "5") ("jack" "6") ("king" "7")
                 ("lamb" "8") ("mask" "9") ("neck" "0") ("pipe" "1")
                 ("quip" "2") ("rope" "3") ("sock" "4") ("tent" "5")
                 ("vent" "6") ("wall" "7") ("xray" "8") ("zinc" "9")))
```

The experimental data is defined in global constants.

```
(defconstant *paired-latencies* '(0.0 2.158 1.967 1.762 1.68 1.552 1.467 1.402))
(defconstant *paired-probability* '(0.000 .526 .667 .798 .887 .924 .958 .954))
```

The do-experiment function takes two parameters which are the number of pairs to present in a trial and the number of trials to run. It calls the appropriate function to run the task based on whether it's the model or a person running as a participant.

```
(defun do-experiment (size trials)
  (if *actr-enabled-p*
      (do-experiment-model size trials)
      (do-experiment-person size trials)))
```

The do-experiment-model function takes two parameters which are the number of pairs to present in a trial and the number of trials to run. It runs the experiment for the size and number of trials requested and returns a list of lists. There is one sublist for each of the trials and they are in the order of presentation. Each of the sublists contains the percentage of answers correct and the average response time for the correct answers in the corresponding trial.

```
(defun do-experiment-model (size trials)
```

Start by defining a variable to hold the results and opening a window for the task.

```
(let ((result nil)
      (window (open-exp-window "Paired-Associate Experiment" :visible t)))
```

Reset the model and tell it which window to interact with.

```
(reset)
(pm-install-device window)
```

Repeat for the required number of trials.

```
(dotimes (i trials)
```

Declare some local variables to hold the score and timing information.

```
(let ((score 0.0)
      (time 0.0)
      (start-time))
```

Iterate over a randomized list of the required number of stimuli from the global set.

```
(dolist (x (permute-list (subseq *pairs* (- 20 size))))
```

Clear the window and present the word from the pair.

```
(clear-exp-window)
(add-text-to-exp-window :text (car x) :x 150 :y 150)
```

Clear the response variables and record the trial start time.

```
(setf *response* nil)
(setf *response-time* nil)
(setf start-time (pm-get-time))
```

Have the model process the display and run for 5 seconds.

```
(pm-proc-display)
(pm-run 5.0 :full-time t)
```

If the answer provided was correct increment the score and the cumulative response time.

```
(when (equal (second x) *response*)
      (incf score 1.0)
      (incf time (- *response-time* start-time)))
```

Clear the window and display the correct answer.

```
(clear-exp-window)
(add-text-to-exp-window :text (second x) :x 150 :y 150)
```

Have the model process the display and run for 5 seconds.

```
(pm-proc-display)
(pm-run 5.0 :full-time t))
```

Compute the response results for the trial and save it on the list of results.

```
(push (list (/ score size) (and (> score 0) (/ time (* score 1000.0))))
      result)))
```

Return the list of results in the proper order.

```
(reverse result)))
```

The do-experiment-person function operates just like the do-experiment-model function except that it waits for a response from a person instead of running the model.

```
(defun do-experiment-person (size trials)
  (let ((result nil)
        (window (open-exp-window "Paired-Associate Experiment" :visible t)))
    (dotimes (i trials)
      (let ((score 0.0)
            (time 0.0)
            (start-time))
        (dolist (x (permute-list (subseq *pairs* (- 20 size))))
          (clear-exp-window)
          (add-text-to-exp-window :text (car x) :x 150 :y 150 :width 50)
          (setf *response* nil)
          (setf *response-time* nil)

          (setf start-time (pm-get-time))
          (while (< (- (pm-get-time) start-time) 5000)
            (allow-event-manager window))

          (when (equal (second x) *response*)
            (incf score 1.0)
            (incf time (/ (- *response-time* start-time) 1000.0)))

          (clear-exp-window)
          (add-text-to-exp-window :text (second x) :x 150 :y 150)
          (sleep 5.0))

        (push (list (/ score size) (and (> score 0) (/ time score))) result)))

    ;; return the list of scores
    (reverse result)))
```

The collect-data function takes one parameter which is the number of times to repeat the full experiment. The experiment with 20 pairs and 8 trials is run that many times and the results are averaged and compared to the experimental results.

```
(defun collect-data (n)
  (do ((count 1 (1+ count))
        (results (do-experiment 20 8)
                  (mapcar #'(lambda (lis1 lis2)
                              (list (+ (first lis1) (first lis2))
                                    (+ (or (second lis1) 0)
                                        (or (second lis2) 0))))
                            results (do-experiment 20 8))))
      ((equal count n)
        (output-data results n))))
```

The output-data function takes two parameters. The first is a list of cumulative data from running multiple iterations of the experiment and the second parameter indicates how

many repetitions were added into that cumulative data. It averages that data and then calls print-results to display the comparison and table for both the latency and accuracy data.

```
(defun output-data (data n)
  (let ((probability (mapcar #'(lambda (x) (/ (first x) n)) data))
        (latency (mapcar #'(lambda (x) (/ (or (second x) 0) n)) data)))
    (print-results latency *paired-latencies* "Latency")
    (print-results probability *paired-probability* "Accuracy")))
```

The print-results function takes three parameters. The first is the list of data from running the experiment. The second is the experimental results for that data, and the third is the label to print when displaying that data. The new data is compared to the experimental results and printed along with a table of the new data.

```
(defun print-results (predicted data label)
  (format t "~%~%~A:~%" label)
  (correlation predicted data)
  (mean-deviation predicted data)
  (format t "Trial    1      2      3      4      5      6      7
8~%")
  (format t "      ~{~8,3f~}~%" predicted))
```

The key handler for the window just records the key press and the time of that press in the global variables.

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

The other model for this unit is the **Zbrodoff** task and is written using the event-based approach. Because this is a more complex experiment than most you have seen, there are some slightly more advanced Lisp facilities used to help keep things clearer and easier to use. Here is the code from the **Misc window**:

```
(defvar *trials*)
(defvar *results*)
(defvar *start-time*)
(defvar *block*)
(defvar *show-window*)

(defconstant *zbrodoff-control-data* '(1.84 2.46 2.82 1.21 1.45 1.42 1.14 1.21
1.17))

(defstruct trial block addend1 addend2 sum answer visible)
(defstruct response block addend correct time)

(defun present-trial (trial)
  (let ((window (open-exp-window "Alpha-arithmetic Experiment" :visible (trial-
visible trial))))
    (add-text-to-exp-window :text (trial-addend1 trial) :x 100 :y 150 :width 25)
    (add-text-to-exp-window :text "+" :x 125 :y 150 :width 25)
    (add-text-to-exp-window :text (trial-addend2 trial) :x 150 :y 150 :width 25)
    (add-text-to-exp-window :text "=" :x 175 :y 150 :width 25)
    (add-text-to-exp-window :text (trial-sum trial) :x 200 :y 150 :width 25)
```

```

(pm-install-device window)
(pm-proc-display :clear t)

(setf *start-time* (pm-get-time)
window))

(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  (let ((trial (pop *trials*)))
    (push (make-response :block (trial-block trial)
                        :addend (trial-addend2 trial)
                        :time (/ (- (pm-get-time) *start-time*) 1000.0)
                        :correct (string-equal (trial-answer trial) (string
key)))
          *results*)
    (when *trials*
      (present-trial (first *trials*))))))

(defun collect-responses (trial-count)
  (setf *results* nil)
  (let ((window (present-trial (first *trials*))))
    (if *actr-enabled-p*
        (pm-run (* 10 trial-count))
        (while (< (length *results*) trial-count)
              (allow-event-manager window))))))

(defun do-trial (addend1 addend2 sum answer &optional (visible (not *actr-
enabled-p*)))
  (setf *show-window* visible)
  (setf *block* 1)
  (setf *trials* (list (construct-trial (list addend1 addend2 sum answer))))
  (collect-responses 1)
  (analyze-results))

(defun do-set (&optional (visible (not *actr-enabled-p*)))
  (setf *show-window* visible)
  (setf *block* 1)
  (setf *trials* (create-set))
  (collect-responses 24)
  (analyze-results))

(defun do-block (&optional (visible (not *actr-enabled-p*)))
  (setf *show-window* visible)
  (setf *block* 1)
  (setf *trials* nil)
  (dotimes (i 8)
    (setf *trials* (append *trials* (create-set))))
  (collect-responses 192)
  (analyze-results))

(defun do-experiment (&optional (visible (not *actr-enabled-p*)) (display t))
  (reset)
  (setf *show-window* visible)
  (setf *trials* nil)
  (dotimes (j 3)
    (setf *block* (+ j 1))
    (dotimes (i 8)
      (setf *trials* (append *trials* (create-set))))
    (collect-responses 576)
    (analyze-results display))

(defun collect-data (n)
  (let ((results nil))
    (dotimes (i n)

```

```

(push (do-experiment nil nil) results))

(let ((rts (mapcar #'(lambda (x) (/ x (length results)))
                  (apply #'mapcar #'+ (mapcar #'first results))))
      (counts (mapcar #'(lambda (x) (truncate x (length results)))
                      (apply #'mapcar #'+ (mapcar #'second results)))))

  (correlation rts *zbrodoff-control-data*)
  (mean-deviation rts *zbrodoff-control-data*))

(print-analysis rts counts '(1 2 3) '("2" "3" "4") '(64 64 64))))

(defun analyze-results (&optional (display t))
  (let ((blocks (sort (remove-duplicates (mapcar #'response-block *results*))
                     #'<))
        (addends (sort (remove-duplicates (mapcar #'response-addend *results*))
                       #'<))
        :test #'string-equal) #'string<))
    (counts nil)
    (rts nil)
    (total-counts nil))

  (setf total-counts (mapcar #'(lambda (x)
                                (/ (count x *results*
                                             :key #'response-addend
                                             :test #'string=)
                                    (length blocks)))
                              addends))

  (dolist (x blocks)
    (dolist (y addends)
      (let ((data (mapcar #'response-time
                          (remove-if-not #'(lambda (z)
                                             (and (response-correct z)
                                                  (string= y (response-addend z))
                                                  (= x (response-block z))))
                          *results*))))
        (push (length data) counts)
        (push (/ (apply #'+ data) (max 1 (length data))) rts))))

  (when display
    (print-analysis (reverse rts) (reverse counts) blocks addends total-
                    counts))

  (list (reverse rts) (reverse counts))))

(defun print-analysis (rts counts blocks addends total-counts)
  (format t "~%          ")
  (dotimes (addend (length addends))
    (format t " ~6@a (~2d)" (nth addend addends) (nth addend total-counts)))
  (dotimes (block (length blocks))
    (format t "~%Block ~2d" (nth block blocks))
    (dotimes (addend (length addends))
      (format t " ~6,3f (~2d)" (nth (+ addend (* block (length addends))) rts)
              (nth (+ addend (* block (length addends))) counts))))
  (terpri))

(defun create-set ()
  (permute-list (mapcar #'construct-trial
                        '(("a" "2" "c" "k")("d" "2" "f" "k")
                          ("b" "3" "e" "k")("e" "3" "h" "k")
                          ("c" "4" "g" "k")("f" "4" "j" "k")
                          ("a" "2" "d" "d")("d" "2" "g" "d")
                          ("b" "3" "f" "d")("e" "3" "i" "d"))))

```

```

("c" "4" "h" "d")("f" "4" "k" "d")
("a" "2" "c" "k")("d" "2" "f" "k")
("b" "3" "e" "k")("e" "3" "h" "k")
("c" "4" "g" "k")("f" "4" "j" "k")
("a" "2" "d" "d")("d" "2" "g" "d")
("b" "3" "f" "d")("e" "3" "i" "d")
("c" "4" "h" "d")("f" "4" "k" "d"))))

(defun construct-trial (settings)
  (make-trial :block *block*
             :addend1 (first settings)
             :addend2 (second settings)
             :sum (third settings)
             :answer (fourth settings)
             :visible *show-window*))

```

and here is the code from the **Commands window**:

```

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

(sgp :v t :esc t :ga 0 :lf 0.35 :bll 0.5 :ans 0.5 :rt 0)

(setf *actr-enabled-p* t)
(setallbaselevels 10000000 -1000)
(add-dm-fct '((goal isa problem)))
(goal-focus goal)

```

Much of the code in the **Commands window** is similar to that from all the other units.

The parameters used in the `sgp` call are:

```

(sgp :v t :esc t :ga 0 :lf 0.35 :bll 0.5 :ans 0.5 :rt 0)

```

It sets the parameters described in the unit – latency factor (`:lf`), base-level decay (`:bll`), the activation noise `s` value (`:ans`) and the retrieval threshold (`:rt`). The `:ga` parameter will be described in the next unit. Setting it to 0 essentially turns off the effect of context on activations to simplify the task for this unit.

The new functions used are the following two calls which are discussed in the unit text.

```

(setallbaselevels 10000000 -1000)
(add-dm-fct '((goal isa problem)))

```

Thus we will go on to the description of the code in the **Misc window**.

Start by defining some global variables to hold the trials to present, the results collected, the start time of a trial, a block count, and flag to specify whether or not to show the window. This already shows some difference from the iterative approach because instead of storing the responses in globals which are collected by the “do-trial” type function the results are stored directly in a global variable for later use. Similarly, the trials are also stored in a global variable instead of being iterated over within a function.

```

(defvar *trials*)
(defvar *results*)
(defvar *start-time*)

```

```
(defvar *block*)
(defvar *show-window*)
```

Then define the global constant that holds the experimental results.

```
(defconstant *zbrodoff-control-data* '(1.84 2.46 2.82 1.21 1.45 1.42 1.14 1.21
1.17))
```

Instead of using lists to represent the information in a trial and to describe a response we create some abstract structures to hold that data in a more descriptive format. A trial consists of a block number, the two addends to present, a sum to present, the correct answer, and whether or not to display the trial in a real window. A response contains the block in which it was given, the numeric addend of the trial (2,3, or 4), whether the response was correct and the response time.

```
(defstruct trial block addend1 addend2 sum answer visible)
(defstruct response block addend correct time)
```

The present-trial function takes one parameter which is a trial structure. It does the same thing whether it is a person or a model performing the task.

```
(defun present-trial (trial)
```

It opens a window and presents the equation.

```
(let ((window (open-exp-window "Alpha-arithmetic Experiment" :visible (trial-
visible trial))))

  (add-text-to-exp-window :text (trial-addend1 trial) :x 100 :y 150 :width 25)
  (add-text-to-exp-window :text "+" :x 125 :y 150 :width 25)
  (add-text-to-exp-window :text (trial-addend2 trial) :x 150 :y 150 :width 25)
  (add-text-to-exp-window :text "=" :x 175 :y 150 :width 25)
  (add-text-to-exp-window :text (trial-sum trial) :x 200 :y 150 :width 25)
```

Then it sets that window as the one for the model to look at and processes it (if a person is doing the task that has no effect).

```
(pm-install-device window)
(pm-proc-display :clear t)
```

The time the trial starts is recorded in a global variable.

```
(setf *start-time* (pm-get-time))
```

The window being used is then returned.

```
window))
```

The key handler for the task does more for the experiment in an event-based experiment.

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
```

The current trial is removed from the global list of trials.

```
(let ((trial (pop *trials*)))
```

A structure that represents the response is then added to the global list of responses.

```
(push (make-response :block (trial-block trial)
                    :addend (trial-addend2 trial)
                    :time (/ (- (pm-get-time) *start-time*) 1000.0)
                    :correct (string-equal (trial-answer trial) (string
key)))
      *results*)
```

If there are more trials to present, then the next one is presented. This is the critical difference between the iterative and event-based experiments. In the event-based experiment the participant's response leads directly to the presentation of the next trial.

```
(when *trials*
  (present-trial (first *trials*)))
```

The collect-responses function takes one parameter which is the number of trials that are to be run.

```
(defun collect-responses (trial-count)
```

The global set of results is cleared.

```
(setf *results* nil)
```

The first trial is presented.

```
(let ((window (present-trial (first *trials*))))
```

If the model is performing the task then it is run for up to 10 seconds times the number of trials that need to be collected.

```
(if *actr-enabled-p*
    (pm-run (* 10 trial-count))
```

If a person is performing the task then the system waits for the appropriate number of responses to be recorded.

```
(while (< (length *results*) trial-count)
  (allow-event-manager window))))
```

The do-trial function takes four required parameters and one optional parameter. The four required parameters are the strings that represent the equation to present, for example "A" "2" and "C", and the string indicating the correct response – either "K" for correct or "D" for incorrect. The optional parameter defaults to the negation of whether or not the model is doing the task. Thus, if the model is doing the task visible is nil in which case the window will be kept virtual and if a person is doing the task the window will be shown. Providing a value of t for the fifth parameter will cause the window to be displayed while the model is doing the task.

```
(defun do-trial (addend1 addend2 sum answer &optional (visible (not *actr-
enabled-p*)))
```

Set the global values to indicate whether to show the window and which block is being presented.

```
(setf *show-window* visible)
(setf *block* 1)
```

This is only one trial, so set the list of trials to a list of only one trial.

```
(setf *trials* (list (construct-trial (list addend1 addend2 sum answer))))
```

Call collect-responses to perform the task and then analyze the results.

```
(collect-responses 1)
(analyze-results)
```

The do-set function takes one optional parameter as described above for do-trial. It runs once through a random permutation of the set of equations. A set is two instances of each of the equations with the addends (2, 3, and 4) in each of the true and false conditions which is a total of 24 problems.

```
(defun do-set (&optional (visible (not *actr-enabled-p*)))
  (setf *show-window* visible)
  (setf *block* 1)
  (setf *trials* (create-set))
  (collect-responses 24)
  (analyze-results))
```

The do-set function takes one optional parameter as described above for do-trial. It runs one block of the experiment, which is eight repetitions of the set of equations, or a total of 192 problems.

```
(defun do-block (&optional (visible (not *actr-enabled-p*)))
  (setf *show-window* visible)
  (setf *block* 1)
  (setf *trials* nil)
  (dotimes (i 8)
    (setf *trials* (append *trials* (create-set))))
  (collect-responses 192)
  (analyze-results))
```

The do-experiment function runs one iteration of the experiment, which is three full blocks, or a total of 576 trials. It takes two optional parameters. The first is to control whether the window is shown or not as with the previous functions. The other controls whether the analysis is printed. The default is to have the analysis printed. Note that this function also calls reset to return the model to its initial condition. It is the only function in the experiment to do so the others allow the model to maintain all the information it has gained (which is mostly the history of chunk use).

```
(defun do-experiment (&optional (visible (not *actr-enabled-p*)) (display t))
  (reset)
  (setf *show-window* visible)
  (setf *trials* nil)
  (dotimes (j 3)
    (setf *block* (+ j 1))
    (dotimes (i 8)
      (setf *trials* (append *trials* (create-set))))))
  (collect-responses 576)
  (analyze-results display))
```

The collect-data function takes one parameter, which is the number of times to run the whole experiment. It runs that many times through the experiment collecting the data which is then averaged and compared to the original experiment's results. It assumes that a model will be running the task and forces the experiment window to be virtual.

```
(defun collect-data (n)
  (let ((results nil))
```

Run the experiment n times with a virtual window and without displaying the individual analysis of each run.

```
(dotimes (i n)
  (push (do-experiment nil nil) results))
```

Compute the average of the response times and number of correct answers.

```
(let ((rts (mapcar #'(lambda (x) (/ x (length results)))
                  (apply #'mapcar #'+ (mapcar #'first results))))
      (counts (mapcar #'(lambda (x) (truncate x (length results)))
                     (apply #'mapcar #'+ (mapcar #'second results)))))
```

Display the data fit between the current run and the experimental data and print the table of the results.

```
(correlation rts *zbrodoff-control-data*)
(mean-deviation rts *zbrodoff-control-data*)
(print-analysis rts counts '(1 2 3) '("2" "3" "4") '(64 64 64))))
```

The analyze-results function takes one optional parameter which controls whether or not the data table is displayed. It computes the average response time and number of correct responses in the *results* global variable as a function of the number of blocks presented and the numerical addend.

```
(defun analyze-results (&optional (display t))
  (let ((blocks (sort (remove-duplicates (mapcar #'response-block *results*))
                     #'<))
        (addends (sort (remove-duplicates (mapcar #'response-addend *results*)
                                             :test #'string-equal) #'string<))
        (counts nil)
        (rts nil)
        (total-counts nil))
    (setf total-counts (mapcar #'(lambda (x)
                                  (/ (count x *results*
                                             :key #'response-addend
                                             :test #'string=)
                                      (length blocks)))
                              addends))
    (dolist (x blocks)
      (dolist (y addends)
        (let ((data (mapcar #'response-time
                            (remove-if-not #'(lambda (z)
                                              (and (response-correct z)
                                                    (string= y (response-addend z))
                                                    (= x (response-block z))))
                                      *results*))))
          (push (length data) counts)
          (push (/ (apply #'+ data) (max 1 (length data))) rts))))))
```

```

      (when display
        (print-analysis (reverse rts) (reverse counts) blocks addends total-
counts))

      (list (reverse rts) (reverse counts))))

```

The `print-analysis` function takes five parameters that describe a set of data for the task. The data is a list of response times and a list of corresponding correct answers. Then there are two lists that indicate the blocks and addend conditions represented in the data and finally a list of the total number of correct trials in each addend condition. Those data are then displayed in a table.

```

(defun print-analysis (rts counts blocks addends total-counts)
  (format t "~%
            ")
  (dotimes (addend (length addends))
    (format t " ~6@a (~2d)" (nth addend addends) (nth addend total-counts)))
  (dotimes (block (length blocks))
    (format t "~%Block ~2d" (nth block blocks)
            (dotimes (addend (length addends))
              (format t " ~6,3f (~2d)" (nth (+ addend (* block (length addends))) rts)
                      (nth (+ addend (* block (length addends))) counts))))
    (terpri))

```

The `create-set` function takes no parameters. It returns a randomly ordered list of 24 trial structures that make up one set of the control condition of the experiment.

```

(defun create-set ()
  (permute-list (mapcar #'construct-trial
                        '(("a" "2" "c" "k")("d" "2" "f" "k")
                          ("b" "3" "e" "k")("e" "3" "h" "k")
                          ("c" "4" "g" "k")("f" "4" "j" "k")
                          ("a" "2" "d" "d")("d" "2" "g" "d")
                          ("b" "3" "f" "d")("e" "3" "i" "d")
                          ("c" "4" "h" "d")("f" "4" "k" "d")
                          ("a" "2" "c" "k")("d" "2" "f" "k")
                          ("b" "3" "e" "k")("e" "3" "h" "k")
                          ("c" "4" "g" "k")("f" "4" "j" "k")
                          ("a" "2" "d" "d")("d" "2" "g" "d")
                          ("b" "3" "f" "d")("e" "3" "i" "d")
                          ("c" "4" "h" "d")("f" "4" "k" "d")))))

```

The `construct-trial` function takes one parameter which is a list of 4 strings that represent a trial in the task. The first three strings are the elements of the equation to display and the fourth is the key that should be pressed for a correct response. It returns a trial structure with the appropriate slot values set.

```

(defun construct-trial (settings)
  (make-trial :block *block*
             :addend1 (first settings)
             :addend2 (second settings)
             :sum (third settings)
             :answer (fourth settings)
             :visible *show-window*))

```