


```

(add-button-to-exp-window :x 10 :y 50 :height 20 :width 20 :text "B"
:action #'button-b-pressed)
(add-button-to-exp-window :x 10 :y 75 :height 20 :width 20 :text "C"
:action #'button-c-pressed)
(add-button-to-exp-window :x 10 :y 125 :height 20 :width 38 :text "Reset"
:action #'reset-display)

(add-line-to-exp-window (list 50 35) (list (+ a 50) 35) 'black)
(add-line-to-exp-window (list 50 60) (list (+ b 50) 60) 'black)
(add-line-to-exp-window (list 50 85) (list (+ c 50) 85) 'black)
(add-line-to-exp-window (list 50 110) (list (+ target 50) 110) 'green)

(allow-event-manager *experiment-window*)

(defun button-a-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'under))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-a*))
        (setf *current-stick* (+ *current-stick* *stick-a*)))
    (update-current-line)))

(defun button-b-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'over))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-b*))
        (setf *current-stick* (+ *current-stick* *stick-b*)))
    (update-current-line)))

(defun button-c-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'under))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-c*))
        (setf *current-stick* (+ *current-stick* *stick-c*)))
    (update-current-line)))

(defun reset-display (button)
  (declare (ignore button))

  (unless *done*
    (setf *current-stick* 0)
    (update-current-line)))

(defun update-current-line ()

  (when *current-line*
    (remove-items-from-exp-window *current-line*))

  (if (= *current-stick* *target*)
      (progn
        (setf *done* t)
        (setf *current-line* (add-line-to-exp-window (list 50 135) (list (+
*target* 50) 135) 'blue))
        (add-text-to-exp-window :x 180 :y 200 :width 50 :text "Done"))
      (if (zerop *current-stick*)

```

```

      (setf *current-line* nil)
      (setf *current-line* (add-line-to-exp-window (list 50 135) (list (+
*current-stick* 50) 135) 'blue))))

      (allow-event-manager *experiment-window*)

      (when *actr-enabled-p* (pm-proc-display)))

(defun do-experiment (sticks)

  (apply #'build-display sticks)

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

(defun do-experiment-model ()
  (pm-install-device *experiment-window*)
  (pm-proc-display :clear t)
  (pm-run 60.0))

(defun do-experiment-person ()
  (while (not *done*)
    (sleep .25))
  (sleep 1))

(defun do-set ()
  (let ((result nil))
    (when *actr-enabled-p*
      (reset))
    (dolist (stim *bst-stimuli*)
      (do-experiment stim)
      (push *choice* result))
    (reverse result)))

(defun collect-data (n)
  (let ((result (make-list (length *bst-stimuli*) :initial-element 0))
        (p-values (list '(decide-over 0) '(decide-under 0) '(force-over 0)
'force-under 0))))
    (dotimes (i n result)
      (setf result (mapcar #'(lambda (x) (if (equal x 'over)
1 0)) (do-set))))
      (setf p-values (mapcar #'(lambda (x) (list (car x) (+ (second x)
production-p-value (car x))))
p-values)))

    (setf result (mapcar #'(lambda (x) (* 100.0 (/ x n))) result))

    (when (= (length result) (length *bst-exp-data*))
      (correlation result *bst-exp-data*)
      (mean-deviation result *bst-exp-data*))

    (format t "~%Trial ")

    (dotimes (i (length result))
      (format t "~8s" (1+ i)))

    (format t "~% ~{~8,2f~}~%~%" result)

    (dolist (x p-values)
      (format t "~12s: ~6,4f~%" (car x) (/ (second x) n))))))

(defun production-p-value (prod)
  (caar (no-output (spp-fct (list prod :p)))))

```

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

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

(sgp :v nil :esc t :pl t :egs 3 :ut -100)

(setf *actr-enabled-p* t)

(pm-start-hand-at-mouse)

(goal-focus goal)

(spp :efforts 500 :successes 100)
(spp decide-over :failures 7 :successes 13 :efforts 100)
(spp decide-under :failures 7 :successes 13 :efforts 100)
(spp force-over :failures 10 :successes 10 :efforts 100)
(spp force-under :failures 10 :successes 10 :efforts 100)

(spp read-done :success t)
(spp pick-another-strategy :failure t)
```

We will start the discussion with the code from the **Commands window**. The first call enables the fixation ring display for the model's visual attention and has been used before.

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

The next line shows a couple of new parameters being passed to sgp.

```
(sgp :v nil :esc t :pl t :egs 3 :ut -100)
```

The `:pl`, parameter learning, parameter specifies whether the production parameter learning mechanism is enabled. The `:egs`, expected gain `s`, parameter specifies the `s` parameter for the noise distribution of the utility (or expected gain) noise, and the `:ut`, utility threshold, parameter specifies the minimum utility a production must have to allow it to fire (the setting to `-100` essentially negates the threshold so that any production will be able to fire regardless of a low utility or a large amount of noise).

Then the model is set as the participant.

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

Because the model will be using the mouse to press buttons the following call ensures that the model's hand is on the virtual mouse at the start of a run.

```
(pm-start-hand-at-mouse)
```

The next line sets the initial contents of the goal buffer.

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



```
(15 250 49 137)(10 179 32 105)
(20 213 42 104)(14 237 51 116)
(12 149 30 72)
(14 237 51 121)(22 200 32 114)
(14 200 37 112)(15 250 55 125))
```

The build-display function takes four parameters which are the pixel lengths of the sticks. It opens a window, draws the initial display and adds the buttons which the user will use to perform the task.

```
(defun build-display (a b c target)
```

Open a window and save it in the global variable.

```
(setf *experiment-window* (open-exp-window "Building Sticks Task"
                                          :visible nil
                                          :width 600
                                          :height 400))
```

Initialize all of the global variables.

```
(setf *stick-a* a)
(setf *stick-b* b)
(setf *stick-c* c)
(setf *target* target)
(setf *current-stick* 0)
(setf *done* nil)
(setf *choice* nil)
(setf *current-line* nil)
```

Add the buttons that the user will use to the task. This is a new function and all of the parameters will be explained in the detailed description below. The most important one is the action parameter which specifies a function to be executed when the button is pressed.

```
(add-button-to-exp-window :x 10 :y 25 :height 20 :width 20 :text "A"
:action #'button-a-pressed)
(add-button-to-exp-window :x 10 :y 50 :height 20 :width 20 :text "B"
:action #'button-b-pressed)
(add-button-to-exp-window :x 10 :y 75 :height 20 :width 20 :text "C"
:action #'button-c-pressed)
(add-button-to-exp-window :x 10 :y 125 :height 20 :width 38 :text "Reset"
:action #'reset-display)
```

Draw the initial set of lines on the display. This is also a new function and will be described in detail below.

```
(add-line-to-exp-window (list 50 35) (list (+ a 50) 35) 'black)
(add-line-to-exp-window (list 50 60) (list (+ b 50) 60) 'black)
(add-line-to-exp-window (list 50 85) (list (+ c 50) 85) 'black)
(add-line-to-exp-window (list 50 110) (list (+ target 50) 110) 'green)
```

Call the system dependent event manager to give the window a chance to display.

```
(allow-event-manager *experiment-window*))
```

The next three functions, button-a-pressed, button-b-pressed, and button-c-pressed are called automatically when the corresponding buttons are pressed because they were

specified in the add-button-to-exp-window calls. They all get passed the button itself as a parameter, but do not need it. All three of them do basically the same thing. If this is the user's first choice it saves whether it is the overshoot or undershoot strategy. Then it computes the new length of the current stick after the application (either addition or subtraction) of the chosen stick, and then calls update-current-line to display the new configuration.

```
(defun button-a-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'under))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-a*))
        (setf *current-stick* (+ *current-stick* *stick-a*)))
    (update-current-line)))

(defun button-b-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'over))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-b*))
        (setf *current-stick* (+ *current-stick* *stick-b*)))
    (update-current-line)))

(defun button-c-pressed (button)
  (declare (ignore button))

  (unless *choice* (setf *choice* 'under))

  (unless *done*
    (if (> *current-stick* *target*)
        (setf *current-stick* (- *current-stick* *stick-c*))
        (setf *current-stick* (+ *current-stick* *stick-c*)))
    (update-current-line)))
```

The reset-display function is called when the reset button is pressed. It takes one parameter which will be the button object, but it does not need it. It sets the length of the current stick to 0 and calls update-current-line to redisplay the state.

```
(defun reset-display (button)
  (declare (ignore button))

  (unless *done*
    (setf *current-stick* 0)
    (update-current-line)))
```

The update-current-line function takes no parameters. It redraws the current line on the screen and sets the done flag if the current stick is the same length as the target.

```
(defun update-current-line ())
```

First remove the old line if there is one.

```
(when *current-line*
  (remove-items-from-exp-window *current-line*))
```

```
(if (= *current-stick* *target*)
```

If the target has been reached then set the done flag, draw the stick and display the done message.

```
(progn
  (setf *done* t)
  (setf *current-line* (add-line-to-exp-window (list 50 135) (list (+
*target* 50) 135) 'blue))
  (add-text-to-exp-window :x 180 :y 200 :width 50 :text "Done"))
```

If the target has not yet been reached then check to see if it has been reset. If it has been reset then just set the current line to nil, otherwise display a new line of the correct length.

```
(if (zerop *current-stick*)
  (setf *current-line* nil)
  (setf *current-line* (add-line-to-exp-window (list 50 135) (list (+
*current-stick* 50) 135) 'blue))))
```

Call the system event manager so that the display gets a chance to update.

```
(allow-event-manager *experiment-window*)
```

If the model is performing the task have it reevaluate the display.

```
(when *actr-enabled-p* (pm-proc-display)))
```

The do-experiment function takes one parameter which should be a list of stick lengths. It calls build-display to open a window and display those sticks and then calls the appropriate function to run one trial depending on whether a person or the model is doing the task.

```
(defun do-experiment (sticks)
  (apply #'build-display sticks)
  (if *actr-enabled-p*
      (do-experiment-model)
      (do-experiment-person)))
```

The do-experiment-model function takes no parameters. It sets which window to interact with and has the model process that display as a new screen, and then runs the model for up to 60 seconds to perform the trial.

```
(defun do-experiment-model ()
  (pm-install-device *experiment-window*)
  (pm-proc-display :clear t)
  (pm-run 60.0))
```

The do-experiment-person function takes no parameters. It just loops waiting for the done flag to be set. Then it waits one second before returning.

```
(defun do-experiment-person ()
  (while (not *done*)
    (sleep .25))
  (sleep 1))
```

The do-set function takes no parameters. It iterates over the list of stimuli in the global list *bst-stimuli* and runs one trial of each recording which strategy was chosen first on each one. If the model is doing the task it is reset before the first trial. It returns the list of strategy choices.

```
(defun do-set ()
  (let ((result nil))
    (when *actr-enabled-p*
      (reset))
    (dolist (stim *bst-stimuli*)
      (do-experiment stim)
      (push *choice* result))
    (reverse result)))
```

The collect-data function takes one parameter which is the number of times to run the full experiment. It runs the experiment the requested number of trials and computes the percentage of times overshoot was chosen first on each trial. It compares that to the experimental data and displays a table of the results. It also records the learned value of the p parameter for the productions that are responsible for the model's choices and reports their average at the end as well.

```
(defun collect-data (n)
  (let ((result (make-list (length *bst-stimuli*) :initial-element 0))
        (p-values (list '(decide-over 0) '(decide-under 0) '(force-over 0)
                          '(force-under 0))))
    (dotimes (i n result)
      (setf result (mapcar #'(lambda (x) (if (equal x 'over)
                                             1 0)) (do-set))))
      (setf p-values (mapcar #'(lambda (x) (list (car x) (+ (second x)
                                                             (production-p-value (car x)))))
                            p-values)))

      (setf result (mapcar #'(lambda (x) (* 100.0 (/ x n))) result))

      (when (= (length result) (length *bst-exp-data*))
        (correlation result *bst-exp-data*)
        (mean-deviation result *bst-exp-data*))

      (format t "~%Trial ")

      (dotimes (i (length result))
        (format t "~8s" (1+ i)))

      (format t "~%  ~{~8,2f~}~%~%" result)

      (dolist (x p-values)
        (format t "~12s: ~6,4f~%" (car x) (/ (second x) n))))))
```

The production-p-value function takes one function which should be a production name. It returns the p value of that production without printing the information to the listener.

```
(defun production-p-value (prod)
  (caar (no-output (spp-fct (list prod :p)))))
```

The new functions used in this model are:

add-button-to-exp-window – this function is similar to the `add-text-to-exp-window` function that you have seen many times before. It places a button in the window that was opened using **open-exp-window**. It takes a few keyword parameters. `:text` specifies the text to display on the button. `:x` and `:y` specify the pixel coordinate of the upper-left corner of the button, `:height` and `:width` specify the size of the button in pixels, and the `:action` parameter specifies a function to be called when this button is pressed. That function will be called with the button object itself as the only parameter.

add-line-to-exp-window – this is similar to the other `add-*-to-exp-window` functions. It draws a line in the window that was opened with **open-exp-window**. It takes two required parameters and one optional parameter. The required parameters specify the pixel coordinates of the end points of the line and each should be a two element list of the x and y coordinate. The optional parameter can be used to specify the color that the line is to be drawn in. The default is black if it is not specified. It must be a symbol (be careful especially when using ACL because many of these names are global variables that evaluate to an rgb object) and can be any of black, blue, red, green, white, pink, yellow, gray, light-blue, dark-green, purple, brown, light-gray, or dark-gray.

remove-items-from-exp-window – this function takes any number of parameters. Each one must be an object that was added to the window opened with **open-exp-window**. Those objects are then removed from that window.