Behavior Research Methods, Instruments, & Computers
2003, 35 (4), 565-576

PsyScript: A Macintosh application for scripting experiments

TIMOTHY C. BATES and LAWRENCE D'OLIVEIRO
Macquarie University, Sydney, New South Wales, Australia

PsyScript is a scriptable application allowing users to describe experiments in Apple's compiled high-level object-oriented AppleScript language, while still supporting millisecond or better within-trial event timing (delays can be in milliseconds or refresh based, and PsyScript can wait on external I/O, such as eye movement fixations). Because AppleScript is object oriented and system-wide, PsyScript experiments support complex branching, code reuse, and integration with other applications. Included AppleScript-based libraries support file handling and stimulus randomization and sampling, as well as more specialized tasks, such as adaptive testing. Advanced features include support for the BBox serial port button box, as well as a low-cost USB-based digital I/O card for millisecond timing, recording of any number and types of responses within a trial, novel responses, such as graphics tablet drawing, and use of the Macintosh sound facilities to provide an accurate voice key, saving voice responses to disk, scriptable image creation, support for flicker-free animation, and gaze-dependent masking. The application is open source, allowing researchers to enhance the feature set and verify internal functions. Both the application and the source are available for free download at www.maccs.mq.edu.au/~tim/psyscript/.

PsyScript was designed as an easy-to-learn application enabling researchers who speak a human language, not C, to accomplish complex experimental tasks. Users design their experiments in AppleScript (www.apple.com/applescript), so PsyScript supports code reuse and complex branching and provides an accessible interface to the objects and properties of an experiment (such as memory-resident stimuli and subject response data). Through a series of evolutions over the past 4 years, a partial listing of PsyScript features now includes the following.

1. Any stimulus supported by QuickTime (including movies, graphics, and most sound files) can be displayed. Stimuli can be double buffered (allowing flicker-free animation and gaze-dependent displays when coupled to a compatible eye tracker). Time-based stimuli, including movies, can be played asynchronously and can even be animated along a path.

2. Responses can be from a keyboard, a mouse (or continuous graphics tablet input), a voice key (from any input, including built-in microphones with saving of voice files to disk), or millisecond USB-based input and output, with access to the Eyelink gaze tracking system and to the serial-based BBox.

3. Time can be specified in terms of intervals (across-trial timing), refresh-based timing, and waiting on external inputs (verification of the timing performance is provided in the Appendix).

4. There is the ability to draw shapes (lines, shapes, polygons, or Bezier curves) to the screen or file.

Correspondence concerning this article should be addressed to T. C. Bates, Macquarie Centre for Cognitive Science, Macquarie University, Sydney, NSW 2109, Australia (e-mail: tim@maccs.mq.edu.au).

We next will describe how these features are accessed in experiments, with example scripts included. Finally, the future directions for PsyScript will be described, including its on-line availability, on-line script repository, and user help group.

Using PsyScript

The PsyScript manual is on line at www.maccs.mq.edu.au/~tim/psyscript/. Therefore, in the next sections, we will introduce the elements of PsyScript that are used in common experiments, with an aim to giving an overview of its features and utility, rather than an exhaustive description.

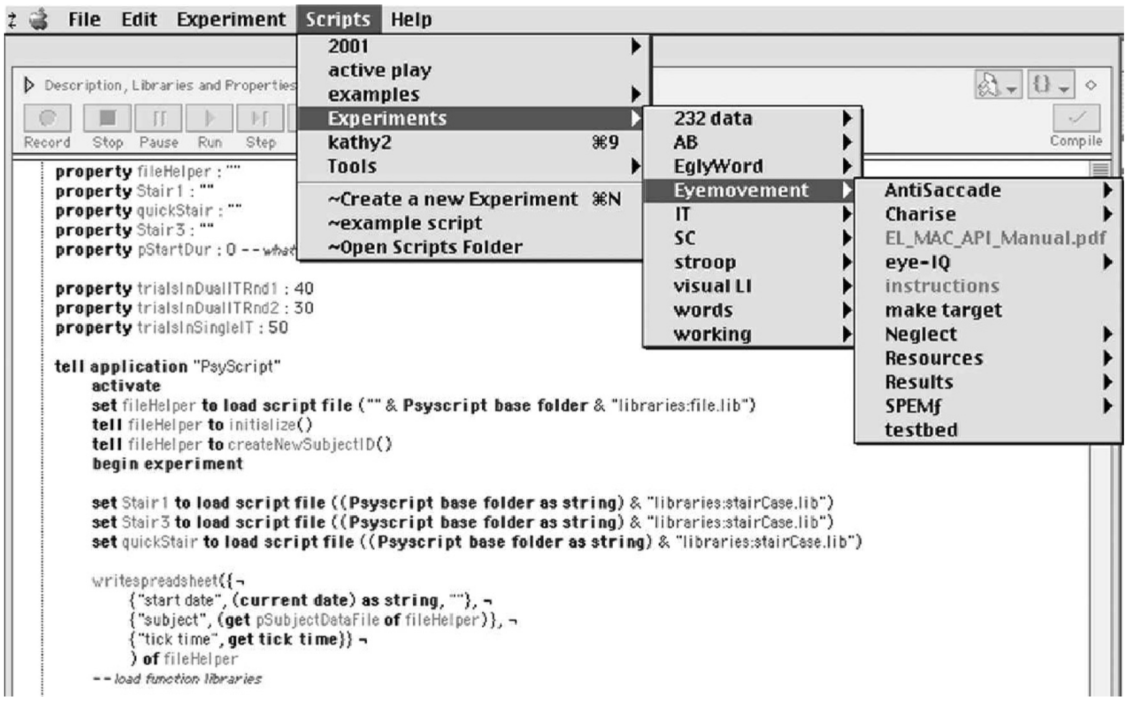
Interface

For the person running an experiment, PsyScript presents a very simple interface (see Figure 1). Only two menus are used in practice: The Experiment menu gives access to a real-time view of sound input and voice key settings thresholds, and most important, the Scripts menu gives access to experiments. User scripts are added to this hierarchical menu by placing them in the "Scripts" folder beside PsyScript. Scripts can be activated by menu selection or by a user-definable keyboard command assigned to each experiment. Compiled scripts saved as applications will execute like any stand-alone Mac application. The power of PsyScript, of course, lies in creating these scripts.

Scripting

Doing an experiment involves writing a script describing the trial events and trial sequence, choosing which subject responses will be noticed, setting word fonts, and so forth. Experiments are written in AppleScript (included on every MacOS since System 7). AppleScript is a full object-

Copyright 2003 Psychonomic Society, Inc.



```
property fileHelper : ""
property Stair1 : ""
property quickStair : ""
property Stair3 : ""
property pStartDur : 0 -- what

property trialsInDualITRnd1 : 40
property trialsInDualITRnd2 : 30
property trialsInSingleIT : 50

tell application "PsyScript"
    activate
    set fileHelper to load script file ("" & Psyscript base folder & "libraries:file.lib")
    tell fileHelper to initialize()
    tell fileHelper to createNewSubjectID()
    begin experiment

    set Stair1 to load script file ((Psyscript base folder as string) & "libraries:stairCase.lib")
    set Stair3 to load script file ((Psyscript base folder as string) & "libraries:stairCase.lib")
    set quickStair to load script file ((Psyscript base folder as string) & "libraries:stairCase.lib")

    writespreadsheet({¬
        {"start date", (current date) as string, ""}, ¬
        {"subject", (get pSubjectDataFile of fileHelper)}, ¬
        {"tick time", get tick time}} ¬
        ) of fileHelper
    -- load function libraries
```

Figure 1. Script Menu: The Scripts menu will display any script or folder placed in the "Script" folder beside PsyScript. The background of this figure shows a script editor window open behind the pulldown menu. This is not related to the current selection in the Scripts menu but does show the syntax styling that AppleScript employs to aid script readability and maintenance.

oriented language designed to be accessible to users and to allow application and OS functions to be scripted. AppleScript is compiled and executes rapidly—well over 100,000 repeat loops per second, with less than a 1-msec delay between calls to PsyScript commands. Within trials, PsyScript executes highly tuned C code, which ensures that the trial execution is precise at a millisecond level. Because AppleScript is a core OS technology for Apple, it is being heavily developed, and a range of new learning materials are emerging for the learning of AppleScript. We will, therefore, focus on PsyScript, introducing just enough AppleScript to make sense of the examples given. Scripts are developed from a free script editor, such as Apple's free Script Editor (search for it on your hard drive) SMILE (http://www.satimage-software.com/) or from a commercial application, such as Script Debugger (http://www.latenightsw.com/).

A Simple Experiment

The concepts of PsyScript will now be introduced in the context of writing a basic experiment. If the reader has downloaded PsyScript, the following scripts and script fragments can be entered into a script editor document and executed, to see at first hand how they function.

Every experiment begins and ends with the following event sequence:

```
tell application "PsyScript"
    activate
    begin experiment
    —experiment events in here
    end experiment
end tell
```

This sequence brings PsyScript to the front (*activate*) and takes control of the screen (*begin experiment*). The experimental trials are then executed, and a call to *end experiment* gives back control of the screen for other applications.[1] As can be seen, the user is aided in reading the script because AppleScript automatically checks syntax and pretty-prints the script into a highly readable format.

Because PsyScript is object oriented, the getting and setting of experiment variables is accomplished by commands to *get*, *set*, and *make*. This is most simply conveyed by extending our example as follows:

```
tell application "PsyScript"
    activate
    begin experiment
    set word size to 28
    set word font to "Times"
    do trial " 'this is a display in 28 point times' "
end tell
```

This script sets the size of subsequent text displays to 28-point Times and the do-trial line displays the phrase "this is a display in 28 point times" on the screen in that font. All other text properties are available to be set in similar fashion.

The Dictionary

One can think of scripting PsyScript as talking to a conscientious assistant who has a reasonable domain-specific vocabulary and a superb memory, but no initiative. Like all AppleScript applications, the commands to which PsyScript can respond and the properties that it can control are revealed in its "dictionary," a collection of "nouns" and "verbs" (objects and the commands available to act on the properties of these objects). This is viewed by opening PsyScript's application dictionary (probably accessed using the File menu of your script editor). There are commands to display instructions, initialize inputs such as the serial port, save voice data to disk, draw lines or more complex objects, or do basic bit-wise arithmetic or base conversion. Also described in the dictionary are classes, such as "key" and "click-area," which allow the user to set mappings for keys or locations on screen and define whether clicks on the screen or particular keys will be noticed. From the application object in the dictionary, the user can see how to access the screen refresh rate (tick time), set the default picture folder in which PsyScript searches for stimulus files, set such properties as the background color, word size, font, color, and so forth, and, importantly, define the type and number of responses that are requested on each trial. Most of the time, properties can be left at their defaults, but when needed, nearly everything can be customized.

Nouns (Objects and Their Properties)

Many of the properties relevant to experiment designers are contained within PsyScript's *application* object (the properties of which are viewed by opening PsyScript's dictionary), which is where properties such as the size, color, and font PsyScript uses to draw text to the screen are instantiated (called, unsurprisingly, *word size*, *word color*, and *word font*).[2] PsyScript's properties and object set are far more extensive than just font properties. Among these are such variables as *response type*, *subject response*, and *reaction time*. This use of plain English, multiple-word descriptors plays a large role in making scripts easy to write, read, and remember.

Verbs

In addition to these application properties, PsyScript understands how to execute a range of commands to preload stimuli into RAM and to access serial or sound port information. Most important, PsyScript understands *do trial* as a request to run a trial as described in the following event string. Shown in use in Figure 2, this is the heart of PsyScript. Its use is most easily understood by example. Imagine that the events of a trial are to show a fixation cross, wait 500 msec, show a picture called "image," wait 100 msec, erase this image, and write a prompt on screen which asks "was that living?" The simplest way to say this in PsyScript is shown in Figure 2.

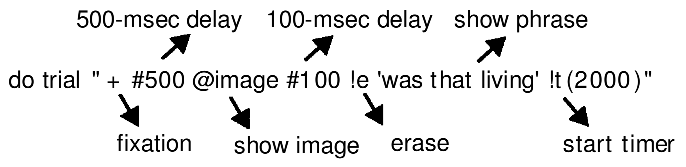
In the example *do trial*, the first event (+) is the code to display a fixation cross. This will be displayed for half a second (#500); then the "@" symbol tells PsyScript that the next string is the name of an image to be loaded and displayed. PsyScript displays this image and then waits 100 msec before erasing the image (!e) and displaying the phrase in single quotes centered on the screen. Finally, the reaction timer is started with a timeout of 2 sec [!t(2000)]. Internally, PsyScript runs an interpreter over the trial string events, turning this into a list of tasks to be executed in sequence. These are then executed in a tightly coded series of C functions. The full list set of mnemonics of this trial language is encapsulated in a dozen or so codes (shown in Table 1). This set of codes is compact, and experience shows that research assistants learn the dozen codes with just 1 or 2 days of experience. Many codes have powerful options for customizing their behavior.

In the fragment below, we introduce two new features: the ability to load stimuli ahead of time for repeated quick RAM-based presentation, and the ability of *do trial* to substitute variables on a given trial (see Figure 3).

The first new line shown in Figure 3 calls *load permanent stimulus* to load the image into RAM. This loads the stimulus into RAM for quick presentation. The command is highly flexible—if the name is omitted, it defaults to the filename, and any file type can be loaded into RAM (movies, sounds, or text). In addition, PsyScript supports extremely powerful masked-image modes for flicker-free animation of a target over a complex background and even sophisticated shading and peek-through modes to support gaze-dependent eye movement research.

The other new features are shown in the *do trial* line. In the portion of the trial string reading "•?1," the • (option-8) symbol tells PsyScript to display the permanent stimulus whose name follows. "?1" is a wild card. PsyScript will replace this with the first item passed in from the *given* parameter of the *do trial* command—in this case, the name of our "rabbit" image.

After a trial is run, the application properties *reaction time* and *subject response* contain the RT and the subject response, respectively. These can then be stored to disk or sent over a network. Where more than one response is



```
do trial " + #500 @image #100 !e 'was that living' !t(2000)"
```

Figure 2. An example trial event sequence. "!t(2000)" signifies that the trial will time out after 2 sec if no responses are detected.

Table 1
List of Event Control Commands

xxx	Load and draw a single word in the center of the screen.
‘xxx'	Draw a string (including multiple words). Add (*x y*) to specify where.
@xxx	Load and display an image named “xxx.” Add (*x y*) to specify where.
~xxx	Load and display QuickTime media named “xxx.” Add (*x y*) to specify where.
*xxx	Display an already loaded stimulus. Add (*x y*) to specify where.
•xxx	Display permanent stimulus “xxx.” Add (*x y*) to specify where.
	Use option-shift-8 (°) to move on without waiting for the movie or sound to complete.
:n<xxx>	Repeat the events inside angle brackets for *n* times.
!l(n)	Force the trial to end *n* msec after this command.
!t(n)	Start the reaction timer. Add (*n*) to set a timeout of *n* msec.
!w(x y)	Word wrap on to draw text like a word-processor. Add (*x y*) to set the position.
#n	Wait *n* msec. Add (name) to load a named stimulus during the pause.
#mn	Wait a multiple of *n* msec since the last call to #m. A single # zeros the interval timer.
!vn	Wait *n* screen refreshes. Use !v by itself to wait one refresh.
#i(l t r b)	Wait until gaze fixation is within the rectangle defined by (left top right bottom).
#b	Wait for a space bar press.
#x	Wait for BBox input.
#a	Wait for an ActiveWire input.
!a	Show/hide the mouse pointer (arrow).
!e	Erase the screen. Use !e(l t r b) to erase a portion of the screen
!h	Hide the screen (paint it with background color).
!u	Unhide the screen.
!s(xxx yyy)	Set the variable xxx to the value yyy [e.g., !s(word_size 12)].
!m	Mark the current time as time 0.
!r(comment)	Report the time since the last mark (!m). Time and comment are stored in *reported time*.
!oa	Output some data from the ActiveWire board
!oic(command)	Output an eyelink command.
!oim(message)	Output an eyelink message.
!is	Read input bytes from the serial port.
!os()	Write output bytes to the serial port.
!ox(n)	Set BBox output line *n* on/off.

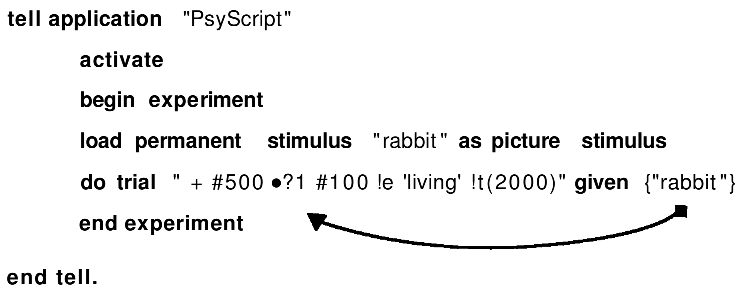
recorded per trial (in some attention paradigms, for instance, or in neglect or memory studies in which multiple mouse clicks or continuous tablet drawing is recorded), these are stored in a variable called, again unsurprisingly, *responses.* This is an object-oriented list, so it can be addressed with such requests as “get every response whose RT is greater than 500” and will return behaviors committed after 0.5 sec or more.

Next we will describe the implementation of some complete experiments, including executing lists of trials, interacting with the subject, and storing data to disk. After this section, we will discuss the range of response modes and advanced stimulus displays supported by PsyScript, as well as additional functionality available from reusable code libraries.

Building a Complete Experiment

Rather than supporting such concepts as *blocks*, *conditions*, or *Latin squares*, PsyScript instead uses such concepts as *lists* and such control structures as *repeat loops* and *conditional execution.* These constructs are very general but still map nicely onto traditional design notions, as will be demonstrated below. Some users wishing, for instance, simply to show a list of words and record the voice RT to each, may consider a simpler application suitable for their needs (Cedrus's SuperLab is often recommended in this context). However, although PsyScript excels at more complex designs, such very simple experiments are correspondingly simple in PsyScript. Assuming that the user has created a tab-delimited text file called “words” in which each line begins with a stimulus word, followed by a tab and any descriptive information (semantic class, etc.), the following 17-line script is a complete experiment in which a list of words is read from a file and each of them is shown in turn (the trial consists of a 500-msec fixation), followed by stimulus onset (the ?1 is replaced with the current word), after which the trial information and voice key RT are written to disk. Note that this script introduces the concept of a *library*: a bundle of useful handlers that can be loaded into any experiment. From a functional point of view, the line beginning “set helper to load script file . . .” makes around 50 useful functions accessible via the object named “helper,” as shown in the following script:

```
tell application "PsyScript"
  activate
  begin experiment
  set helper to load script file ("" & Psyscript base folder ¬
  & "libraries:file.lib")
  tell helper to initialize()
  tell helper to createNewSubjectID()
  set response type to sound response
  set recording of current sound to true
  set stimList to readSpreadsheet("words.txt") of helper
  set stimList to randomize(stimList) of helper
  repeat with aStimulus in stimList
```



```
tell application "PsyScript"
        activate
        begin experiment
        load permanent stimulus "rabbit" as picture stimulus
        do trial " + #500 •?1 #100 !e 'living' !t(2000)" given {"rabbit"}
        end experiment
end tell.
```

Figure 3. Do trial with a wild card. Note that the "?n" marker is replaced with the corresponding item from the list passed in as the given parameter. Named records can also be used. So this trial could be written as follows: *do trial* "+ 500 •?stimulus #100 !e 'living' !t(2000)" *given* {stimulus: "rabbit"}.

```
            do trial "!e + #250 !e !t ?1 " given aStimulus
            set theRT to reaction time
            tell helper to writespreadsheet(aStimulus & theRT)
        end repeat
        end experiment
    end tell
```

Designing in PsyScript

Two key insights to successful PsyScripting are to build from one's trial string outward and to use custom objects and *handlers* (AppleScripts term for functions) to hide complexity. By building from the trial out, we mean starting with a simple skeleton that activates PsyScript and executes one real trial. (An easy way to generate an experiment skeleton is to select "create a new experiment" from PsyScript's Scripts menu. PsyScript will build a file, plus resources and results folders, and will open the script in one's favorite script editor.) Within 1 5 min, a trial line can usually be created that executes the core trial logic in one's experiment. At this point, the experiment is working, and from this point the user can write some more trials, add a line to store the data, and then add any variable elements to the trial by including wild cards in the trial string and passing in values for each wild card in the do-trials *given* option. Finally, and this is a key to simplifying designs, the custom do-trial and data storage events can be combined in a simple handler that hides the complexity of the trial in one line, perhaps called "DoMyTrial()." This custom trial handler can then be embedded in a repeat loop and can be automatically called for every stimulus in a list of stimuli. At this point, a line can be enabled to read the stimuli from a text file, or read all the image names in a certain folder, perhaps adding a line to randomize these stimuli. In the next section, these stages will be explored in a concrete fashion, showing how an experiment can be written out "long hand" and, then, how complexity can be hidden and how structure can be used to to produce a very compact but powerful experiment.

The Design Process

In any PsyScript experiment, two necessary events are calling *do trial*, and writing the response (and any other information needed to be stored, such as stimulus parameters) to disk. Most of these are provided via libraries, which users can extend. The results of a trial are stored to disk by asking the helper to do *writespreadsheet()*:

```
tell helper to writespreadsheet({reaction time, subject¬
response, stimName})
```

Given this basic core of resources, we can write out in "long hand," as it were, an experiment with two conditions, each containing a block of trials:

```
tell application "PsyScript"
  activate
  begin experiment
  set helper to load scriptfile ("" & Psyscript base folder¬
  & "libraries:file.lib")
  tell helper
    initialize()
    createNewSubjectID()—prompts for a unique ID
  end tell
  display instructions "
    Please press a key 1-5 for how pleasant each word is
    Press space to start"
  do trial " + #500 'cat' !t #500 !e "
  set theRT to reaction time
  set theResponse to subject response
  tell helper to writespreadsheet({"rating", "cat", theRT, ¬
  theResponse})
  do trial " + #500 'dog' !t #500 !e "
  set theRT to reaction time
  set theResponse to subject response
  tell helper to writespreadsheet({"rating", "dog", theRT, ¬
  theResponse})
  do trial " + #500 'lard' !t #500 !e "
  set theRT to reaction time
  set theResponse to subject response
  tell helper to writespreadsheet({"rating", "lard", theRT, ¬
  theResponse})
  do trial " + #500 'squib' !t #500 !e "
  set theRT to reaction time
  set theResponse to subject response
    tell helper to writespreadsheet({"rating", "squib", ¬
    theRT, theResponse})
    display instructions "Next, please press n for new or ¬
    o for old
```

```
        Press space after you have had a rest and are ready to ¬
        continue"
    do trial " + #500 'cat' !t #500 !e "
    set theRT to reaction time
    set theResponse to subject response
    tell helper to writespreadsheet({"familiar","cat", theRT, ¬
    theResponse})
    do trial " + #500 'dog' !t #500 !e "
    set theRT to reaction time
    set theResponse to subject response
    tell helper to writespreadsheet({"familiar","rat", theRT, ¬
    theResponse})
    do trial " + #500 'lard' !t #500 !e "
    set theRT to reaction time
    set theResponse to subject response
    tell helper to writespreadsheet({"familiar", "butter", ¬
    theRT, theResponse})
    do trial " + #500 'squib' !t #500 !e "
    set theRT to reaction time
    set theResponse to subject response
    tell helper to writespreadsheet({"familiar", "squib" ¬
    theRT, theResponse})
    end experiment
    end tell
```

This works but is very repetitive. We could take advantage of the repetitive nature of the experiment and write a handler that can take a condition name and word and run the trial:

```
on DoTrial(thecondition,theWord)
    tell application "PsyScript"
        do trial " + #500 '?1' !t #500 !e " given {theWord}
        set theRT to reaction time
        set theResponse to subject response
        tell helper to writespreadsheet({thecondition,the Word,¬
        theRT, theResponse})
    end tell
end DoTrial
```

The core of the experiment would then look like the following:

```
    display instructions "
    Please press a key 1-5 for how pleasant each word is
    Press space to start"
    my DoTrial("rating", "cat")
    my DoTrial("rating", "dog")
    my DoTrial("rating", "lard")
    my DoTrial("rating", "squib")
    display instructions "Next, please press n for new or ¬
    o for old
    Press space after you have had a rest and are ready to ¬
    continue"
    my DoTrial("familiar", "cat")
    my DoTrial("familiar", "rat")
    my DoTrial("familiar", "butter")
    my DoTrial("familiar", "squib")
    end experiment
```

Still, we are using a line for every trial; we cannot easily randomize the stimulus order or read stimuli from disk. One could easily take much better advantage of the structure of the experiment.

Because there is a list of stimuli to present, we can store these as a list and repeat the trials for each item of the stimulus list, under each condition. This can be written as

```
tell application "PsyScript"
    activate
    begin experiment
    set helper to load script file ("" & Psyscript base folder ¬
    & "libraries:file.lib")
    tell helper
    initialize()
        createNewSubjectID()
    end
    set wordList to {"cat", "dog", "lard", "squib"}
    set checkList to {"cat", "rat", "butter", "squib"}
    set condition to "rating"
    display instructions "Please press key 1-5 for how ¬
    pleasant each word is. Space to start"
    repeat with aWord in wordList
        my DoTrial(condition, aWord)
    end repeat
    set condition to "familiar"
    display instructions "Next, please press n for new or o ¬
    for old. Space to continue"
    repeat with aWord in checkList
        my DoTrial(condition, aWord)
    end repeat
    end experiment
end tell
```

This not only is shorter, but also is more easily understood and modified. Essentially, what we have been doing is recovering the structure of the experiment by writing a procedural script. In the final example, we see how, by abstracting the experiment into some properties (at the top of the file), an experiment can be built that is very easily modified to accommodate new designs in a compact and informative fashion. Each condition has an associated condition name and instructions. We can add in a unique word list accessed from a file as well. These can be stored in a record as follows:

```
property rating : {label: "rating", instructions: "Please
press a key 1-5 for how pleasant each word is.",
wordfile:"ratingwords.txt"}
property familiarity : {label: "familiar", instructions:
"Next, please press n for new or o for old", wordfile:
"ratingwords.txt"}
tell application "PsyScript"
    activate
    begin experiment
    set helper to load script file ("" & Psyscript base folder ¬
    & "libraries:file.lib")
    tell helper to initialize()
    tell helper to createNewSubjectID()
—the next 6 lines are the experiment
repeat with aCondition in {ratingCondition,
    familiarCondition}
        display instructions instructions of aCondition
    set wordList to helper's read1DSpreadsheet(wordFile of ¬
    aCondition)
        repeat with aWord in wordList
        my DoTrial(label of aCondition, aWord)
```

```
end repeat
    end experiment
end tell
```

The word stimuli will be read into "wordlist," and then each will be displayed in return and an informative data file will be written containing all the parameters of each trial on one line. At this point, our experiment is very compact and easily modified. An RA could change the word list simply by editing a text file. If the experiment changed to require the contents of "wordFile" to be randomized for each subject, the user can simply add one line:

```
set wordList to helper's randomize(wordList)
```

In PsyScript, a condition can be viewed as a list of elements, such as stimuli, labels, and instructions. These can be labeled for easy access. A block is just a run of trials in a repeat loop. The key conceptual move is to think of these experimental constructs in terms of lists and repeat loops on these lists. It is probably true that the learning threshold for understanding PsyScript is higher than that of some applications. However, we would suggest that the power in PsyScript is incommensurably higher while remaining accessible in a way that C is not. Moreover, where a set of related experiments is required to be modified by inexperienced users, it is very easy to build PsyScript experiments that just read in text files or folders of images for their stimuli.

Having shown how an experiment can be implemented in PsyScript, in the next sections of this paper, we will turn to the more powerful features of PsyScript: libraries, response types, and stimulus presentation and creation.

Libraries

Libraries supplement PsyScript by providing procedures for reading data files, storing responses, and randomizing lists and for the myriad of other common tasks in an experiment. These are delivered as free open-source libraries that can be used as is or extended to suit different styles or purposes, building on the properties of PsyScript. Libraries are included for screen-related functions (translating visual angle into pixels and creating images), implementing an adaptive staircase, supporting a serial-based skin conductance system, and basic graphing functions.

A library is simply a collection of handlers that can be imported into any experiment. When loaded via the *load script* command, these handlers can be accessed via the newly loaded script object. The object-oriented nature of these libraries is soon appreciated as one begins passing messages from one object to another and implementing an intelligent data flow system. Although programmers can contribute directly to the core of PsyScript, libraries extend much of this power to the end user. Whenever the user writes a handler, it can be saved into his or her own library and then loaded like the built-in libraries, making this new functionality available to all subsequent scripts written in that lab. Suggestions for new functions are welcome, and requests from users are usually coded by me or another developer and then rolled into the library as functions available to all, usually within just a day or two. The 50 or so most commonly used handlers are included in "file. lib." Access to these functions is aided by a script (called "authoring aid") that is installed by placing it into the Scripts menu of the script editor and that, when called, displays a pop-up list of the library functions and pastes the selected handlers into the current script or creates a new template experiment and pastes the selected handlers into this.

Subject IDs and Saving Data to File

At the beginning of most experiments, the script should create a data file in which to store responses, probably one based on the subject ID and experiment name. To take a concrete example, an experimenter wants to prepend "AB" for "attentional blink" and postfix the data files with "_2," because this happens to be Experiment 2. This is accomplished in the three new lines below:

```
set helper to load script file ("" & Psyscript base folder & ¬
"libraries:file.lib")
tell helper
   setSubjectFileParameters({prefix:"AB_", suffix: "_2"})
   initialize()
   createNewSubjectID()
end
```

The *createNewSubjectID()* handler is surprisingly powerful. It creates a prompt dialog requesting the subject name and checks that the entered name will lead to a unique data file (if not, it offers options to overwrite, append, or choose again). In this example, when data are written to file, they will be stored in a file called "AB_ID_2" (e.g., "AB_timb_2").

After each trial, it is usual to store the subject response and RT, along with a record of the stimuli used on that trial (assuming the stimulus names are in variables "stim1" and "stim2"). Just tell helper:

```
writespreadsheet({stim1, stim2, subject response, ¬
reaction time})
```

Again, this is a powerful handler. It will accept one-dimensional lists (as above), two-dimensional (2-D) lists, or strings, and it will correctly turn these into tab-delimited data and write them to disk. AppleScript is able to do complex operations, and so one can also compute summary statistics on line and append these to the data file. As will be shown below, responses can be mapped to meaningful labels, such as "left" or "smaller," rather than key codes, such as "l" or "<." Because of this, it is easy to arrange an experiment so as to generate data files ready for direct import into spreadsheets or stats files, complete with column labels, summary statistics, and so forth.

Reading From Disk

It is often convenient to store stimulus names in text files—say, lists of words or image names. PsyScript can handle delimited stimulus files. For example, stimuli for an experiment on naming might be stored in a file called "words" that contains text lines consisting of three tab-delimited identifiers, as below:

```
“cat→       living→       regular↲
dog→        living→       regular↲
quoit→      nonliving→    irregular↲
sew→        action→       irregular”
```

The following script line will read this file into a variable storing the trials as a 2-D array:

```
set theStimList to readSpreadsheet("words")
––> {{"cat", "living", "regular"}, {"dog", "living", ¬
"regular"}, {"quoit", "dead", "irregular"}, {"sew", ¬
"action", "irregular"}}
```

List items are accessed by asking for them with phrases such as *get item* 2 *of* stimList.

PsyScript's library of handlers includes functions to sort, randomize, and generate selections from item lists, either with or without replacement. Included are some quite sophisticated tools allowing, for instance, sampling with replacement, when items must not repeat within a certain minimum number of items. We will give just one example:

```
sparseRepeat given taking: 9, outof: {1, 2, 3, 4, 5, 6, 7, ¬
8, 9}, separatedBy: 3
––>{4,5,1,8,4,6,3,8,5}
```

It can be seen that items are allowed to repeat, but not within three items of their last appearance. So "4" appears twice, but separated by three items. The time saved by using these built-in handlers is considerable. Because there are over 50 handlers in "file.lib," it is recommended that the user just print out the library and keep it handy for reference. Use the authoring aid for quick access to the available handlers.

Response Types

PsyScript can use keyboard response, mouse clicks, BBox (the CMU button box at http://psyscope.psy.cmu.edu/bbox/index.html), voice key (any Macintosh sound input, including built-in sound, and USB devices such as the iMic), and the ActiveWire USB board. Each of these response modes has addressable properties. Multiple input types are available within a single trial. To listen for both keypresses and voice responses, the user would say,

```
set response type to {keyboard response, voice response}
```

When keypresses and mouse clicks are counted, certain responses can be ignored—for instance, we might only wish to notice presses on the "Y" and "N" keys and to receive responses on noticed keys with more informative labels or "mappings." To do this, we simply tell PsyScript to notice certain keys and to map any keypresses to more informative text:

```
set noticed of key "Y" to true
set mapping of key "Y" to "Yes"
tell key "N" to set {noticed, mapping } to {true, "No"}
```

Here, I used two ways to map the noticed responses onto English mnemonics "Yes" and "No," respectively. Now, at the end of trial, if we ask PsyScript for the RT and subject response, we will get something like

```
get {reaction time, subject response}
    → {546, "Yes"}
```

This makes for highly informative results files. Because PsyScript supports a full "object model," we can check which objects are set to noticed with the following line:

```
get mapping of every key whose noticed is true
    → {"No", "Yes"}
```

Click regions are also objects, so we can define rectangles within which mouse clicks will be noticed (to notice all clicks, just define a rectangle as big as the screen), to be mapped to text labels. The following code shows a simple mouse experiment:

```
tell application "PsyScript"
    set response type to mouse response
    tell click area 1
        set bounds to {0, 0, 100, 100}
        set mapping to "top left"
            set noticed to true
    end tell
    activate
    begin experiment
    draw rectangle in bounds of click area 1 with filling
    do trial "'click in here'(50 50)"
    end experiment
    reset all
end tell
```

Voice Key

PsyScript uses built-in sound sources for voice key measurement. Especially in a larger lab, this can save some expense. The PsyScript voice key allows the user to select a sound input from the available devices, to control properties such as pretriggering (an opportunity to store sound from before the trigger point—useful for off-line analysis of missed onsets) and trigger levels (the volume at which PsyScript will detect an RT or a voice onset). These can be chosen interactively in a dialog box. The voice response returns an RT just like a keyboard response, and in addition, the actual sampled voice can be saved to disk. A simple Stroop experiment demonstrates this:

```
tell application "PsyScript"
    reset all
    set theData to {}
    set helper to load script file ("" & Psyscript base folder ¬
    & "libraries:file.lib")
    activate
    begin experiment
    set word size to 32
    set response type to sound response
    set recording of current sound to true
    set theColors to {"red", "green", "blue", "yellow"}
    repeat with n from 1 to 50
        set {theColor, theword} to helper's sample(2, the ¬
        Colors, false)
        —takes 2 items from the color list, without
        replacement
        do trial "+ #500 '?theColor' #150 !e" given ¬
        {theword:theword, theColor:theColor}
        set the end of theData to {theColor, reaction time}
        save sound response data to ("" & (pPathto ¬
        ResultsFolder of helper) & ".aiff")
    end repeat
    end experiment
    tell helper to writespreadsheettoFile(theData, choose ¬
    file name)
end tell
```

Much of this code we have already seen. On each trial, a fixation cross is drawn; then, after a 500-msec delay, a word is presented for 150 msec, and the subject's vocal RT is collected. New features demonstrated here include setting the response type to sound response and starting sound recording (PsyScript listens to the sound port continuously to avoid any latency glitches). In addition, this experiment demonstrates a repeat loop, in which we repeatedly execute a trial with new values (in this case, colors selected at random).

Millisecond Response Boxes

To use the legacy CMU BBox requires a USB serial converter. But once it is plugged in, the user can simply add a line to initialize the BBox and then address it like a keyboard:

```
initialize bbox
    set response type to bbox response
    set {noticed, mapping} of bbox key 1 to {true, "left"}
    set {noticed, mapping} of bbox key 2 to {true, "left"}
```

The BBox can also be addressed within a trial. Output lines can be set on or off (perhaps to turn lights on and off or to trigger a Scanner) by saying "!ox(n t)" or "!ox(n f)" to set line *n* of the BBox to true or false, respectively. PsyScript can also wait for a BBox key or line [using the trial code "#x(n)"].

A new millisecond-accurate response is also supported: the ActiveWire USB card (current price, $59 at www.activewireinc.com; we are investigating using an even cheaper part from www.delcom-eng.com for the OS X version of PsyScript). The Activewire is a 75 × 50 mm card providing 16 bits of user-definable digital I/O. PsyScript can address these I/O bits, both outside trial lines (turning lights on and off, talking to other equipment, etc.) and in three within-trial contexts. Generic digital output purposes, such as triggering an fMRI scan, are implemented using the *!oa(n)* command ["!o" means "output," "a" stands for "ActiveWire," and the brackets hold the digital value to output (specified as a decimal string or a binary number)]. One can also wait on an input [e.g., waiting for an fMRI machine to signal that it has started a scan; this is done with *#a(n)*, where the brackets contain the digital line number to monitor]. Finally, the ActiveWire can be used as a millisecond response box, monitored by setting the response mode to ActiveWire and by setting ActiveWire lines to be noticed or ignored and to be mapped to informative labels, just as with the keyboard. ActiveWire events are polled within 1 2 msec, and this event is registered against the Mac's internal 64-bit microsecond timer that is accessed within a few microseconds to accurately stamp ActiveWire events relative to the !t event in a trial. A script using ActiveWire is shown below:

```
tell application "PsyScript"
    activate
    begin experiment
    initialize activewire
set response type to activewire response
    set noticed of every activewire pin to false
    tell activewire pin 1
        set mapping to "red"
        set noticed to true
    end tell
    tell activewire pin 2
        set mapping to "blue"
        set noticed to true
    end tell
    do trial "!s(word_color red) 'press the red or blue ¬
    button'(50 50)"
    end experiment
    reset all
end tell
```

This will set two buttons to be noticed, mapping their buttonpresses to the labels "red" and "blue." It then executes a trial using the ActiveWire as a millisecond timer. This example also demonstrates setting the word color within a trial, as well as drawing multiple words as a phrase and setting the location for drawing (the phrase is drawn in red centered on the point $x = 50$ $y = 50$).

User Interaction

User interaction is also readily handled by dialogs and other tools. In the following example, we show some instructions, ask the subject his or her age (with a default of 19), do a trial in which we show this, and then pause the experiment, with some on-screen help:

```
showAsInstructions("Next I will ask your age: press space¬
to continue", 20, 200)
set theAge to getInfo("what is your age?", 19)
do trial "' my, you certainly don't look ?1" given {theAge}
takeABreak("you should be pressing "y" when you see the cat")
```

External Devices

PsyScript can currently read data from any USB HID (human input device), such as keyboards, mice, joysticks, graphics pens, and so forth. It can also read and write to the serial port and BBox data lines. Integration with TMS, fMRI, and/or EEG equipment is typically trivial to implement, whereas the serial port is adequate for collecting some data directly (such as a serial skin conductance device; http://www.psylab.com/html/default_sc5sa.html).

Timing

Control over time is quite flexible, as the following trial line demonstrates:

```
do trial "#m(2000) + #100 !e #b #i(200 200 300 300) !v3"
```

The "#m(2000)" tells this trial to first wait until 2 sec have elapsed since the last time a #m call was made. This gives a very simple method of generating millisecond precision across trials, as required in fMR, and fMRS experiments. A fixation is then displayed for 100 msec [+ #100 !e]. For the sake of this demonstration, we wait for a space bar press [#b] and then wait until the subject is fixating within a 100-pixel-square fixation rectangle [#i(200 200 300 300)]. (Note that use of the latter command assumes access to an eyelink tracker.) Finally, the trial synchs to the vertical refresh timer and waits three screen refreshes, exiting during the retrace phase of the third refresh.

Stimuli

PsyScript uses QuickTime to provide access to dozens of files types, including jpg, gif, mov, mpg, snd, aiff, and tiff. PsyScript also has the ability to create lines, polygons (rectangles, arcs, and ellipses, as well as arbitrary open or closed polygons and polylines), and polycurves (Bezier figures) and to draw these directly to the screen or store them in memory to create new RAM-based stimuli or to save to disk. The latter two capabilities are useful for prebuilding several hundred systematic variations of stimuli for later use, where doing this manually would be rather dull.

This example code will draw two colored buttons on screen labeled "Yes" and "No." This can be extended to simulate a graphical response pad on screen:

```
tell application "PsyScript"
    reset all
    activate
    begin experiment
    set word size to 28
    set rect1 to {50, 50, 150, 100}
    set rect2 to {160, 50, 260, 100}
    set word color to {32000, 32000, 32000}
    draw rectangle in rect1 using {pen color: green} with ¬
    filling
    draw rectangle in rect2 using {pen color: red} with ¬
    filling
    do trial "'YES'(?1) 'NO'(?2)" given {{100, 85}, {210,
    85}}
    end experiment
end tell
```

A replication of the experiment in Egly, Rafal, Driver, and Starreveld (1994; available on the Web site) uses this drawing capability to create a set of 32 distinct stimulus images, each with at least nine on-screen elements. An advantage of such programmatic stimulus creation is that these images are easily redesigned if viewing distance or response tasks change. The experiment also maintains a very small storage size, relative to the cost of distributing many dozens of image files with the script.

Levels of Use

It can be seen from the above sampling of PsyScript's stimulus and output control that complex experiments can be generated. Scripting can, however, be approached at several levels. Local users have written several "template" experiments that load a text file of words or a folder of images and execute these, using a preprepared event sequence. In this way, undergraduate students who perhaps could not write a full experiment can nevertheless produce variations on a theme simply by varying the contents of a stimulus list or image folder. Current templates are available for masked priming and lexical decision among other paradigms.

Next in complexity, many experiments can be produced by taking a basic example and modifying the stimuli and the do-trial event sequence. Many complete experiments are only a dozen or so lines of code, and variations, such as changing timing parameters in a *do trial* or changing the *response type* from *keyboard response* to *mouse response* or *voice response*, can be as simple as editing just one line of such a "starter" script. To support this mode, a growing suite of examples of well-known paradigms are available, including Posner's attention network task (Fan, McCandliss, Sommer, Raz, & Posner, 2002), Egly's object-space paradigm (Egly et al., 1994), and attentional blink (Shapiro, 1994), as well as lexical decision tasks used in the study of dyslexia (Castles & Coltheart, 1993) and context processing tasks used in the study of schizophrenia (Cohen, Barch, Carter, & Servan-Schreiber, 1999), to mention a few.

Future

In this brief article, we have not touched on several aspects of PsyScript that are, nevertheless, of value to end users. The ability of AppleScript to access applications other than PsyScript is of increasing value. For instance, we recently created scripts using Apple's standard "Internet Scripting Extension," which allows PsyScript to save data directly over the Web to our center data base (an OS X server running Apache, MySQL, and php). This allowed us to collect data via ftp from a laboratory in another state and to make the class results available over a Web page, using MySQL to allow the students to access average RTs for the conditions of interest in their tutorial. A second recent use was conducting an analysis of data (means and standard deviations) inside the tutorial script for reporting to the user, again for use in a tutorial. This could easily be expanded so that individual data are submitted to a Web form and a current result sheet is returned for display on screen. Such innovations give an idea of the utility of AppleScript in scientific work flows and the flexibility PsyScript gains by leveraging AppleScript.

PsyScript is available for free on the Web, including the full source code, and we facilitate a user group that readers are welcome to join. We anticipate that the open source will help provide solutions for any problems or new uses that are needed. A native version for OS X is being coded (the current version runs under "classic," but a native version will enable us to utilize the reliability of OS X's BSD underpinnings and sophisticated new graphics modes [OpenGL and Quartz]). There is already a diverse community of users around the world. Via AppleScript libraries, even users with no experience in such languages as C can contribute to PsyScript's development. Because PsyScript is open source (C source code is in text files; the project is in CodeWarrior Development Studio, Mac OS X Edition, Version 8 format), laboratories can take the source and either contribute to the main version or create their own derivatives. In this way, the tools of experimental psychology can be more like our research output: open to peer review and more directly linked to selective pressures for change.

Resources

PsyScript home page (manual, application download and source code repository): www.maccs.mq.edu.au/~tim/psyscript/.

PsyScript support and suggestion list mailing list subscription page: http://www.maccs.mq.edu.au/mailman/listinfo/psyscript.

AppleScript: http://www.apple.com/applescript/.

AppleScript Language Guide: http://developer.apple.com/techpubs/macosx/Carbon/interapplicationcomm/AppleScript/AppleScriptLangGuide/index.html.

MacScripter.net: http://www.applescript.net/.

REFERENCES

Castles, A., & Coltheart, M. (1993). Varieties of developmental dyslexia. *Cognition*, **47**, 149-180.

Cohen, J. D., Barch, D. M., Carter, C., & Servan-Schreiber, D. (1999). Context-processing deficits in schizophrenia: Converging evidence from three theoretically motivated cognitive tasks. *Journal of Abnormal Psychology*, **108**, 120-133.

Egly, R., Rafal, R., Driver, J., & Starreveld, Y. (1994). Covert orienting in the split brain reveals hemispheric specialization for object-based attention. *Psychological Science*, **5**, 380-383.

Fan, J., McCandliss, B. D., Sommer, T., Raz, A., & Posner, M. I. (2002). Testing the efficiency and independence of attentional networks. *Journal of Cognitive Neuroscience*, **14**, 340-347.

Shapiro, K. L. (1994). The attentional blink: The brain's "eyeblink." *Current Directions in Psychological Science*, **3**, 86-89.

NOTES

1. When an experiment ends, the *end experiment* command releases the screen. If this is not included in the script, the user will see a blank screen, and input will appear to be ignored. To end an experiment at any time, you can simply press the <Escape> key, and PsyScript will execute an *end experiment* command for you. If your trial string does not pause for user input (and hence, never gets a chance to see the <Escape>), the application switching command (command-tab by default) can be used to get back to your script editor. This single-line script can then be accessed from the "scripts" folder of your editor to return control to the user: "tell app "PsyScript" to end experiment."

2. These properties can be set within a trial as well. For example, to change word color within a trial, one could use *do trial* " !s(word_color red) 'this text is red' !s(word_color blue) 'text is now blue'."

(*Continued on next page*)

APPENDIX
Timing Data

Internally, PsyScript uses 64-bit timestamps derived from the Mac's internal microsecond clock. The accuracy of interevent timing can be verified by the user by bracketing the events of interest in “!m” (start marker) and “!r()” (report time) codes. Timing data so generated are stored in the *reported times* application property. For instance, the following fragment verifies the wait function:

```
tell application "PsyScript"
    activate
    begin experiment
    do trial "!m #100 !r()"
    end experiment
    return reported time
    →{ 101 }
end tell
```

External timing validation has been carried out on several machines locally using a photodiode trigger to time screen updates and a LabVIEW A/D card and oscilloscope to time the ActiveWire card, voice key, and various complex trials. Data from an older midrange G3 (black-and-white 350-MHz single-processor, 256 MB of RAM) running OS 9.2.2, AppleScript 1.8 and QuickTime 6b1, with an Apple 17-in. CRT monitor (75-Hz refresh rate) are as follows.

Screen Refresh

A series of durations of a stimulus (white patch) were generated by the following script and the activity of a photodiode pointed at the white patch recorded on a triggered storage oscilloscope. Over 200 trials, all white/black trigger cycles occurred at the one-refresh interval, with no intervals missed and no measurable interval extension or compression.

```
tell application "PsyScript"
    activate
    begin experiment
    repeat with N from 1 to 200
        do trial "!e #500 !v •white !v?1 •black" given {N}
    end repeat
    end experiment
end tell
```

ActiveWire Timing

ActiveWire timing was verified by setting the response type to ActiveWire and then triggering the ActiveWire by making a loop-back connection between an ActiveWire output line connected back into the input line. The do-trial syntax was “ #100 !t #?1 !oa(1)” with a random delay of 100 1,000 msec being used to prevent any spurious synchronization between the trial events and other processes. In a run of 100 repetitions, the median trigger-to-RT detection latency was 1 msec (range, 1 3).

Voice Key Timing

Voice key timing was verified by setting the response type to voice response and then triggering the sound input line connecting an ActiveWire line into the line-in input of a USB iMic device with sound input set to 16-bit, 48-kHz recording (these values minimize the input buffer size, minimizing latency).

http://www.griffintechnology.com/audio/imic_main.html

Over a run of 150 trials using the same trial string as that used for ActiveWire timing, the latency from trigger-to-RT detection ranged from 2 to 3 msec with VM off. Under OS X, it is hoped to constrain this value further.

(Manuscript received December 11, 2001;
revision accepted for publication March 31, 2003.)