Behavior Research Methods, Instruments, & Computers
1994, 26 (2), 94-106

2. INVITED SPEAKER

Chaired by Paula Goolkasian, *University of North Carolina*

Using computers in empirical and theoretical work in cognitive psychology

ROGER RATCLIFF
Northwestern University, Evanston, Illinois

This article presents some of the issues I have faced in setting up my own laboratory system for research in cognitive psychology. First, for real-time subject testing systems, criteria for choosing hardware and software are presented along with a brief discussion of benchmarks. Stimulus generation and data analysis choices are also discussed. Second, choices of software for data analysis and data exploration are presented, including graphical presentation, statistics, and modeling tools. Third, some consideration is given to benchmarking workstations to be used in modeling (neural modeling in particular), and benchmarks of some of the fastest single processor systems available are presented. Fourth, two examples of the use of the various theoretical tools are given—one for testing compound cue models of priming, and the other for testing global memory models.

In this article, I discuss a number of issues in the development and use of a computer laboratory for research in cognitive psychology. The problems I discuss have been major ones for me in my research, and here I present them, together with my solutions, as examples of the kinds of problems that are often faced. The first domain concerns real-time experimental testing systems and relevant software, hardware, and benchmarking problems. The second involves techniques for handling data, including data analysis, graphical presentation, and modeling tools and languages. The third set of issues concerns the speed at which computers can run modeling programs, and problems that can arise in benchmarking workstations. Following a presentation of various tools in each of these domains, I give two examples of substantive issues that can be resolved relatively quickly by using them. In one example, I show how the space of parameter values for a model can be searched to find the parameters that best fit a set of data that shows priming effects in lexical decision. In the second example, a simple binomial model is used to predict the slope of the z-ROC function for recognition memory data.

This research was supported by NIMH Grants HD MH44640 and MH00871 to R.R., and NIDCD Grant R01-DC01240 and NSF Grant SBR-9221940 to Gail McKoon. I thank Gail McKoon for extensive comments on this article. Address correspondence to R. Ratcliff, Psychology Department, Northwestern University, Evanston, IL 60208.

DESIGN ISSUES FOR A REAL-TIME SUBJECT TESTING SYSTEM

In cognitive psychology, we have come a long way from the primitive empirical methods of 25 years ago. At that time, PDP-11 computers were just making their appearance; much reaction time work was still done using Hunter timers, for which the poor experimenters (cf. Keenan & Kintsch, cited in Kintsch, 1974) had to write down the response times by hand. The early computers were a help, but they often had to be programmed in assembly language, sometimes with paper tapes. For those lucky enough to have a programmer to write a real-time system, there was always the potential for blackmail. In general, performing a reaction time experiment was a daunting task and required a considerable investment. Even 10 years ago, a hardware platform presented a big problem—any computer was very expensive, and Apple II computers were several thousand dollars. Now, with personal computer (PC) prices so low, a real-time subject station costs as little as $600 (standard PC hardware).

The hardware choice for our laboratory is the PC; it is here to stay, the upgrade paths are clear, and there are millions of them out in the field. Given this choice, a number of subsidiary issues arise. One is the issue of hardware support. In our laboratory, when a PC breaks, it is usually repaired if it is new, but it is replaced with a new and faster computer if it is old.

Another major consideration is real-time software. Our goal has been stable software that rarely needs modifica-

tion and, therefore, minimizes our reliance on programming support personnel. When modifications are needed, because the system is written in standard Borland C, any competent PC programmer can work with it quite easily. An important piece of advice to graduate students is to avoid becoming the laboratory expert and support person for software or hardware. It can result in much unproductive effort in supporting other research projects (but every student should know enough to facilitate his or her own research).

The ease with which real-time software can be used is extremely important for overall laboratory performance. We have attempted to make our system easy and foolproof for our assistants and students to use, without compromising generality. Each of our subject station PCs is connected to our main host workstation via serial lines to a multiport device on the workstation (this was cheaper than an ethernet solution, and if the network goes down the experiments still run). At the beginning of an experimental session, the list of stimuli together with real-time commands is transferred from the workstation to the PC's memory. Stimuli for the whole experimental session are stored in memory to avoid any potential delays due to transfer from disk to memory. In most text-based experiments, there is enough memory to do this. For example, 5 letters per word at 200 msec per word = 25 letters per second = 1,500 letters per minute = 75,000 letters per 50-min session, or 75K of data, which is less than the available memory. During the session, no data are stored on disks to be transferred (or lost) at a later time; instead, data files are transferred from the PC disk to the workstation (using a scripting language for communication) immediately after the experiment ends. To run an experimental session, the experimenter simply logs in at the PC screen to the workstation, changes to the appropriate directory, hits the F10 key, enters the name of a file containing the stimuli, enters a name for the file in which data will be written (which must not already exist), gets a chance to change these in case of error, and then the experiment is executed. At the end of the experiment, the data are on the workstation hard disk, and the system returns to a UNIX prompt, ready for the next session. On the UNIX workstation, a "restricted shell" is used, which allows only the commands cd, ls, and xmodem (the transfer protocol). This stops ignorant experimenters (and malicious subjects) from being able to execute the following commands: cd ~;/bin/rm -rf *.

ISSUES OF DISPLAY AND RESPONSE RECORDING

Screen Characteristics

For short presentation times, a monitor with a rapid decay phosphor is required so that a stimulus will not remain visible after the screen is cleared. Information about a monitor's type of phosphor and phosphor decay rate is virtually impossible to get; very few monitor manufacturers will specify this information. In addition, especially for the cheaper manufacturers, monitors that are ostensibly identical and that are from the same manufacturer can have different characteristics (because the manufacturer buys tubes from different tube manufacturers for the same model display). In the laboratory, screen characteristics can be measured with a fast photodiode and a fast oscilloscope with a repeating stimulus. A more informal approach is to look for afterimages on the screen, or scroll something by very quickly. For many experiments, either screen characteristics are not important, or a mask can solve the problem.

Raster Scan

A major problem when short stimulus exposures are required is the raster scan of the screen (a screen is written line by line at, typically, a rate of 16.67 msec per screen). Interstimulus intervals usually must be restricted to 16.67-msec steps by the raster scan rate (this can be avoided with a very fast vector scope that allows 1-msec steps). A further problem is that there is no way of knowing exactly where the electron beam is when the programmer thinks a stimulus is being written to the screen. It may be just above or just below the position of the letter, and so there may be either 1 msec until the actual write, or as much as 16 msec. There are simple low-technology solutions, such as tape a piece of cardboard over all except the top line, print the stimulus just below the top line of the screen, scroll the stimulus up, and, after the number of raster scans required for the proper stimulus duration, scroll it off the screen. There are also technological solutions, which involve detecting in software when the beam is at the top of the screen and using this to unblank a screen onto which the stimulus was previously written. But these issues only apply when stimulus duration must be very short; once presentation time is above 50–100 msec, the sources of variability from raster scanning are small compared with variability from the subjects.

Keyboard Accuracy

The possible variability introduced through the use of keyboards for response collection is a frequently mentioned, but ill-understood problem. In a recent review of a manuscript of ours, we were criticized for reporting accuracy in milliseconds, even though the scanning rate of keyboards is "slow." There are two parts to the problem—scanning rate and standard deviations. It is the size of typical standard deviations that allows keyboard accuracy to be relatively poor. In fact, if a scanning rate was so slow that a fast typist could not work, reaction time data could still be obtained without detectable loss of accuracy compared with an accurate keyboard. A typical (fast) RT (reaction time) standard deviation is 200 msec. If the keyboard standard deviation (the standard deviation in the delay between keypress and recording the time) is 10 msec, the overall standard deviation is $\sqrt{(10^2+200^2)} = 200.24$. Or, if the keyboard standard deviation is 50 msec, the overall standard deviation is 206.1 msec. Thus, a delay of 0–100 msec in accessing the keyboard (SD ~ 50 msec) leads to only a 3% increase in RT standard deviation. In our lab system, when a key

is pressed, a hardware interrupt is generated, the program that is in a timing loop detects this, and values for the key pressed and response times are stored in the keyboard buffer. A response recording routine continually scans the buffer, and when it detects an entry it reads the 1-msec clock to produce a reaction time.

Real-Time Software

We have previously reported on a language for controlling real-time experiments (Ratcliff & Layton, 1981; Ratcliff, Pino, & Burns, 1986). The language was first implemented on proprietary hardware, then on an inexpensive Radio Shack Color Computer, and now on standard PC hardware. Our experience with this language has been excellent. The language is designed to accept files that contain textual stimuli to be printed on the screen, interleaved with real-time commands. So, for example, to present the word *cat* for 500 msec, clear the screen, present the word *dog*, and collect a keypress and reaction time, the sequence cat#W500@Cdog#R is used. #W500 waits 500 msec, @C clears the screen, and #R collects a keypress and reaction time. We have not found any text-based experiment that we wanted to run that could not be run with this system. Recently, we decided to implement a button box response recording routine. Because we have control over the software and have all the routines and compilers, this implementation took about an hour for a C programmer. One of the important advantages in having an interpretive real-time language is that "what you see is what you get." If there is an error, it is only necessary to follow a listing of the stimulus file to the point of the error, and the error becomes obvious. This means that little time is spent on debugging the real-time aspects of programming.

Picture Presentation

For displaying pictures, we have implemented some simple commands that store pictures in extended memory as screen images (e.g., we can store 105 pictures in 4 MB of high memory) and block-move them into video memory for display (e.g., #B500 presents a picture for 500 msec). For some video cards, painting a whole picture on the screen takes much longer than one raster scan (16.67 msec). To improve presentation speed, we implemented an algorithm by which each picture is broken into eight strips, and only the strips that change from one picture to the next are loaded into memory. This also increased the number of pictures that could be stored in upper memory. In other words, if all the pictures in a set of stimuli fill only half the screen, only that half of the screen is modified when each following picture is displayed, cutting the time required to paint a picture on the screen in half, and allowing twice as many pictures to be stored. This eliminates the need to use a buffering scheme, in which successive pictures are written to alternate pages of video memory.

Auditory Stimuli

To present auditory stimuli, we prepare the stimuli on a NeXT computer (and yes, service facilities will be provided for the next 4 years) with one stereo channel to play the stimuli and the other for signals that cause a pulse to be sent into the game port of a PC. The pulse is used to initiate visual events (e.g., for cross-modal priming experiments) and to initiate response recording.

Benchmarking Real-Time Software

In any computer system, an important issue is benchmarking. For a real-time system, for example, clearing the screen means writing all blank characters to it, and this can take a significant number of milliseconds.

We benchmarked our 386SX (16 MHz) PC with a stopwatch, with the following results:

1. Time for 10,000 clear-the-screen commands: 14.5 sec—that is, 14.5 msec to clear the screen.
2. Time for a wait of 120 sec in 2-sec intervals: 120.0 sec.
3. Time for 10,000 IF statements: 15 sec—that is, 1.5 msec per IF.
4. Time for 10,000 "wait 1 msec": 12.5 sec—that is, 0.3 msec overhead per wait command.
5. Minimum time between 50 pairs of keypresses (in which the keys are hit as simultaneously as possible): 2 msec.

Benchmark 5 shows that if two keys are pressed almost simultaneously, then differences as small as 2 msec can be detected. The interrupt system guarantees that the first detection is rapid, and the 2-msec difference guarantees that a second keypress can be recorded with as little as a 2-msec delay. Even if random delays are added to these figures, the computations above show that this variability is of no real concern.

The examples show the kinds of benchmarks that are needed to check the accuracy of timing commands; run a lot of them, time them with a stopwatch, and divide.

LIST GENERATION

One of the factors that is time intensive in experimental work is generating stimulus lists that have the right counterbalancing and randomizations. Steve Greene and I developed a language that would speed the required programming. The language was designed to be as flexible as possible so that it would not restrict experimental designs (we did not consider a template model). The language we (mainly Steve) developed (Greene, Ratcliff, & McKoon, 1988) has shorthand ways of placing items in various list positions, randomizing items within those constraints, sampling with and without replacement, and so on. The big practical wins of this system over standard programming languages are (1) reading in items and manipulating them is transparent whether they are single letters, words, or paragraphs, and (2) randomizations and constraints on randomizations within the test list are transparent and handled automatically by the program. So, for example, if initial constraints place list items in certain positions that are inconsistent with later constraints, the system automatically recognizes this and tries again. In regular programming, handling failures such as this can

take a much larger amount of code. Third, transparency comes about because the language is essentially a shorthand for FORTRAN. What would be multiple lines of code in a FORTRAN program (e.g., up to 20 lines) becomes a single line, and the single line corresponds to a single kind of stimulus. So the stimuli and their presentation requirements are laid out in as few lines as possible—that is, as many lines as there are kinds of stimuli. This means that the program all fits (usually) on one workstation text window. Consequently, it is easy to look across the whole design at once, making writing and debugging the program easier and less prone to mistakes. Further discussion of the design and implementation of this list generation system can be found in Greene et al. (1988).

DATA ANALYSIS

Once data is received back on our workstation, we then have to analyze it, graph it, and run various statistical tests. For many years, our initial set of analyses (for means, trimming of outlier reaction times, etc.) used FORTRAN programs, which often turned out to be minor modifications of earlier programs. But we kept wanting other analyses such as medians, standard deviations on particular conditions, materials analyses, and so on, and these required new programming for each experiment. I decided that we (i.e., our current programmer) could produce a shorthand language to perform our typical kinds of data analyses. The requirements were that it be workstation based so that all the data could be read into memory, even for a large experiment. This ensures that any new analyses can be performed in memory (e.g., different reaction time cutoffs, standard deviations), making them much faster than if the data were read off disk. For the data analysis program, we decided that data must come in a standard format for all experiments for each response: keypress, RT, return, multidigit code. A simple example for the analysis of perceptual identification data (Ratcliff & McKoon, in press) is shown in Table 1. The first line is a file that contains a list of the names of the data files for each subject. The second line tells how each code is to be split up: i1 defines variable *i* as the first single digit, *j*1 defines *j* as the second single digit, and *k*2 defines *k* as the last, two-digit number. The next line defines the key types: / is one response category, and *z* or *Z* is another response category. The fourth line specifies a loop over subjects, and the fifth line specifies the conditions that will be analyzed: If *i* is 2, print analyses for conditions $j = 1, 2, 3$, and 4. Table 1 also shows the data analysis output that this example produces. On each line, the conditions for loop *j* are shown in parentheses, followed by the number of observations for that condition with a / response, the mean RT for those responses, and the probability of those responses (the number of responses divided by the total number of observations for response key categories). The next three numbers on the line are the same for the other response category (*Zz*).

Table 1
Data Analysis Script for an Object Decision Experiment

Input Script

```
ip4all
i1 j1 k2
/ Zz
FOR SUBJ
IF (i==2) j=1-4+1
END
%%
```

Data Analysis Output

```
stat
Command file (CR to end): ip4inp
MAIN_INDEX #1
  (1)      426  880.6  0.667      213  901.6  0.333
  (2)      366  922.4  0.574      272  925.0  0.426
  (3)      264  955.0  0.416      370  924.9  0.584
  (4)      258  931.0  0.405      379  842.3  0.595
MAIN_INDEX #1, CMD >
```

After this initial analysis, the program offers several useful options. For example, entering "detail" produces a subject-by-subject display of the data. This allows very rapid examination for slow or inaccurate subjects, and these can be excluded from further analyses. The program also allows many of the analyses found in a recent paper (Ratcliff, 1993) that examined the power of reaction time analyses under a number of different ways of handling reaction time outliers. Interactively, with one line of command each, cutoffs can be altered, medians computed, transformations performed (e.g., log, or 1/RT), cutoffs based on standard deviations produced, and so on. In addition, cutoffs based on individual subject data can easily be determined, and these different cutoffs can be carried across to materials analyses. The mean reaction times, number of observations, or response probabilities can be dumped to a file that, with some formatting information, can then be input to a simple analysis of variance (ANOVA) program (Hacker & Angiolillo-Bent, 1981). Perhaps the main advantage of the data analysis program is that it increases the ease of programming analyses by a factor of 10 or more. A typical analysis program of ours takes 8-15 lines of code, much of which is ritualized from experiment to experiment. It takes a few seconds to read all the data for an experiment from disk (e.g., 20-60 subjects), and recalculation varies from almost immediate to just a few seconds. Multiple analyses can be carried out quickly and easily in real time, and a feeling for the data can be obtained very quickly. For example, fast and slow responses can be examined by simply entering GUB 700, where GUB specifies a cutoff for long RTs (greatest upper bound) and 700 msec is the mean RT. This produces an analysis of data for all RTs less than the mean. Entering GLB 700 (greatest lower bound 700) produces the data analysis for RTs greater than the mean.

If the data require more than simple hypothesis testing with ANOVA, they can be dumped into the data analysis/graphical presentation package, S. This package offers many different analysis methods and graphical output op-

tions, and it contains a full programming language so there are few constraints on what it can do. This package is also useful for the various kinds of modeling described below. In addition, operations in S are very compact.

Graphical output from packages like S is sometimes problematic in that it is fixed and not amenable to annotation or modification. For S, there is a public domain output filter that allows output to be placed in a file in Framemaker format, which means that all the text manipulation (size, font) and drawing tools in that system are available.

TOOLS FOR MODELING

For any serious computer modeling, workstations running UNIX are much preferable to PCs and Macs. Workstations are much faster, they come with big screens, big disks, and lots of memory, and they include lots of tools. The main drawback is that to keep these things working, you need a UNIX system manager who knows how to configure mail for ethernet, add new peripherals, set up printers, add new users, configure host addresses, cross-mount disk file systems, and so on. These are not trivial, and, from personal experience, they can take huge amounts of time for the untrained. One good half-time programmer/system manager can maintain a network of 20 workstations of different types in steady state (occasional replacements, software upgrades, addition of disks and memory, etc.). A positive feature of UNIX is that you can run several things at once without crashing the machine. Routinely, our workstations are up for over 100 days, running simulations almost continuously.

For the kind of modeling I do, I have found FORTRAN to be the most useful language. Workstations are tuned for maximum speed under FORTRAN and optimizing compilers for FORTRAN are very good (compilers are just about as good for C for numerical work). The other packages we have needed and used are Mathematica, S, and Framemaker. Framemaker provides a document processing capability, table builder, math equation builder, simple drawing program (good enough for most nondata figures), and complete formatting system. We typically prepare our papers in an ASCII file with some macros (a .mml file); then, when we want to produce a final version, we read it into Framemaker with a prepared file of settings to make it APA style or two-column style (like a journal article).

For data analysis and graphing, S is hard to beat. We typically dump the output of our data analysis program into a file and then read the results into S. We can draw histograms—hist(x,nclass = 10)—plot x,y graphs—plot(x,y)—label axes, headers, and so on. There are also many tools built into the system.

Mathematica is a well-known tool for all kinds of mathematical work. It does symbolic mathematics (integrals, derivatives, solving equations, etc.) as well as numerical mathematics. It has hundreds of built-in functions and extensive graphic output options. The latter is one of its great strengths. The main drawback is the speed of the numerical operations, so there are some interesting tradeoffs for simple modeling.

The issue of speed for these high-level systems (such as Mathematica and S) is critical. I have found that it is possible to quickly program fairly complicated models in Mathematica and S. If the models do not require too much computation, or require only a single solution, then using these systems is a great win in terms of my time. But if the problem requires many runs, function fitting with numerous parameter values, and so on, Mathematica and S are too slow and I have found it better to recode the problem into FORTRAN.

One advantage of these languages is the availability of high-level functions (matrix eigenvalues, fitting routines). For programming languages, there are resources for such high-level functions in various archives containing well-tested routines. Two useful sources are Statlib and Netlib. Statlib contains a large number of statistical routines, and Netlib contains several standard matrix manipulation packages as well as all kinds of numerical analyses, linear algebra, numerical optimization, differential equation routines, and many others. To obtain information about libraries, archive servers are available, and a simple e-mail message is enough to get going.

Statlib: to begin just send email containing the one line message
send index
to statlib@lib.stat.cmu.edu"

Netlib: If you have a numerical problem, it is worth looking through the index of this library for something that might help. To get started send a one line message:
send index
to netlib@ornl.gov

Of course there are many other archives, including one for Mathematica code. It is useful to have knowledge about what is available and how to search for it on Internet; with tools such as the Internet "surfing" package MOSAIC (or gopher), accessing these archives is becoming very easy indeed.

Probably one of the biggest problems in computational modeling is how to check that a model is correctly implemented. The most worrisome aspect of this is that there might be a bug in the computer code that is responsible for the behavior of the model (some may say that this might then create a new and better model). There are a number of ways to address such concerns. First, if a special case or possibly a few special cases can be found, then errors that are common to all cases can be ruled out. If there are no special cases, it might be possible to modify the model to predict a special case by, for example, reducing the number of alternatives from many to two, turning off noise in the model, or keeping some of the varying parameters constant. Another method is to write the program in a high-level language first, to examine some aspects of behavior (e.g., Mathematica), and then reprogram it into a regular programming language (FORTRAN or C). Then only the algorithm needs to be checked if the codes give the same answers. If all else fails, two independent people can program the model and compare results (e.g., the

investigator and a programmer, graduate student, or postdoctoral fellow). This might seem obvious, but it is very tempting, if the program seems to give the right results, to simply forget testing it and move on.

THREE EXAMPLES

The following three examples were chosen to show the use of the S language and Mathematica, and in particular to show how compact the code can be, even for procedures that are quite involved.

Plotting Z-ROC Functions

The code shown in Table 2 reads in data from the files "hit" and "fa" and converts the data to z scores using the qnorm function. The data consist of four conditions each, with five hit rates and five false-alarm rates derived from confidence judgments. The data are read into a 4×5 matrix for hits and a 4×5 matrix for false alarms, and matplot plots the z-ROC functions for the four conditions, as shown in Figure 1. Adding the axis labels and headings takes no more than an extra 5 min and results in a publishable plot.

Table 2
S Example

```
Read in hit and false-alarm rates for:
  5 different confidence levels
  4 different conditions
Convert to z scores using "qnorm"
Put them each in a matrix
Plot them

  mf < -matrix (qnorm(read('fa'),0,1),ncol=4)
  mh < -matrix (qnorm(read('hit'),0,1),ncol=4)

  matplot (mf,mh,type='b',xlab='Zfa',ylab='Zhit')
```

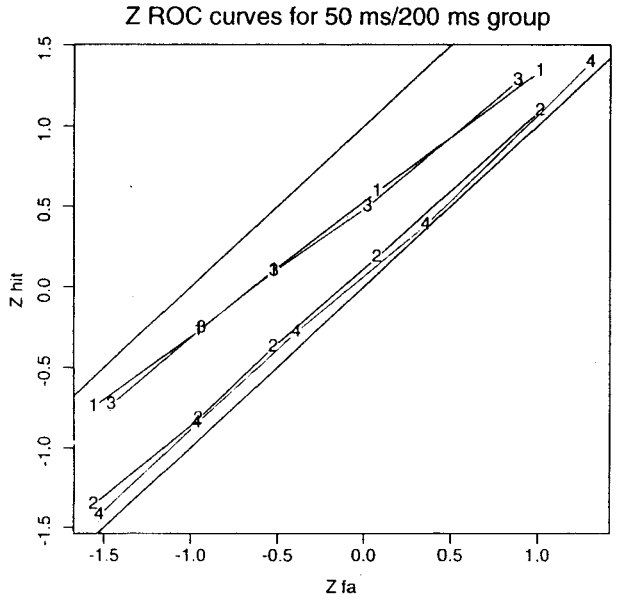


Figure 1. Sample z-ROC functions plotted using the S language.

Table 3
Mathematica Code for a Random Walk

```
Define the random walk (FoldList produces a running sum of random numbers that are 1/2 or -1/2):

  Randwalk[n_Integer] :=
  FoldList[Plus,0,Table[Random[Integer]-1/2,{n}]]

ListPlot plots the random walk:

  ListPlot[Randwalk[1000]]

The Position statement will print out when the walk first reaches position 4, in this example:

  Position[Randwalk[1000],4,{1}][[1]]
```

Random Walk Simulation

This example (Table 3) is taken from the Mathematica book and illustrates how simple it is to simulate the random walk and find first passage times (number of steps to first cross a boundary). The first line defines the random walk, where FoldList applies the first function (addition or plus) starting at the second argument (zero) to the list in the third argument position, and creates a running sum. The result is the running sum of random numbers with values of +1/2 or −1/2. ListPlot plots the random walk, as shown in Figure 2, and the Position function prints the position at which the walk first reaches size 4 (for example).

Spreading Activation

Spreading activation models assume that concepts or words are stored in separate nodes in memory, and that when an item is presented to the network, activation is sent from the item's node out to nodes to which it is connected. These send activation to nodes to which they are connected, and so on, and the amount of activation decreases as distance from the stimulus node increases. The most detailed spreading activation model as applied to psychological data is ACT* (J. R. Anderson, 1983). Because this model has been used in our research (e.g., McKoon & Ratcliff, 1992), I implemented the asymptotic activation version (when activation reverberation has settled down) in Mathematica to give predictions for nonmediated and mediated priming conditions (see McKoon & Ratcliff, 1992; McNamara, 1992). For the ACT* model, asymptotic conditions exist when the net input to each node (multiplied by a transmission constant—e.g., $p = .8$) is the same as the activation at that node. For activation at node i, a_i, and net input $n_i = c_i/p + \Sigma_j r_{ji} a_j$, where c_i is the external activation to node i, $0 = pn_i - a_i$. Rewriting these equations in matrix and vector form, $\mathbf{A} = \mathbf{C} + pR\mathbf{A}$, where $\mathbf{A}$ and $\mathbf{C}$ are vectors corresponding to a and c, respectively, and R is the matrix of interconnections. Solving for $\mathbf{A}$ gives the vector of activation values, given an input vector of activation values $\mathbf{C}$: $\mathbf{A} = (I - pR)^{-1}\mathbf{C}$, where I is the identity matrix (diagonal values 1, and off-diagonal values 0). Mathematica code for this equation is shown in Table 4 and begins by defining the matrix of interconnections between nodes, as shown both schematically and as a matrix of numbers in Figure 3. The next line defines the number of nodes (nn) and forms the iden-

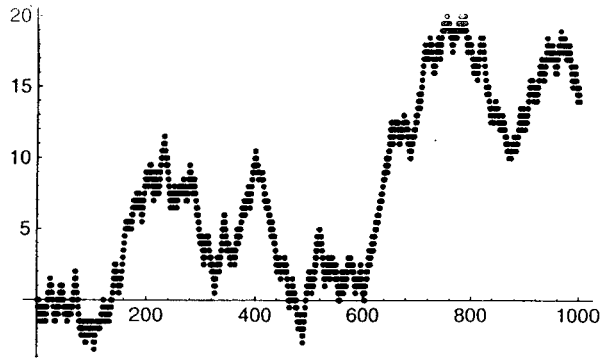


Figure 2. A simple random walk implemented and plotted in Mathematica.

Table 4
Mathematica Code for the Spreading Activation Model (J. R. Anderson, 1983)

First, set up the matrix of connections strengths:

```
m={{0,.33,0,.17,.17,.17,.17,0,0,0,0,0,0,0},
{.38,0,.38,0,0,0,0,.08,.08,.08,0,0,0,0},
{0,.5,0,0,0,0,0,0,0,0,.125,.125,.125,.125},
etc.
nn=14;im=IdentityMatrix[nn];
inv=Inverse[im-m*.8];
```

The vector/matrix product po.inv provides as output the activation values in the nodes:

```
po={0, 1, 1, 0, 0, 0, 0, 0, etc};
po.inv
```

Output:

```
{1.60, 3.01, 2.81, 0.21, 0.21, 0.21, etc}
po={1, 0, 1, 0, 0, 0, 0, 0, etc};
po.inv
```

Output:

```
{2.81, 1.99, 2.36, 0.37, 0.37, 0.37, etc}
```

tity matrix. The next line takes the inverse of the matrix, and "po" is defined as the input vector (C) of activation values (a 1 denotes activation input to that node). Finally, multiplying the input vector by the matrix "inv" produces an output vector of activation values. Thus, the model is implemented economically in six commands, and also runs fairly quickly (in seconds).

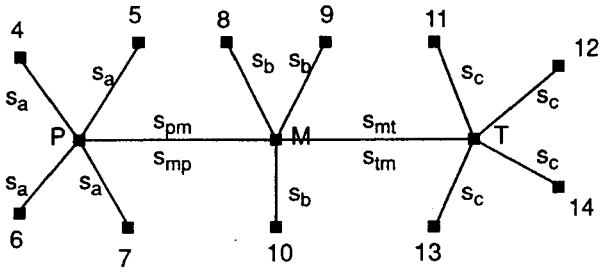
CAVEAT

This presentation describes a set of tools we have found useful and discusses some of the reasons why we do things the way we do. The caveat is that there are many other approaches—just read *Behavior Research Methods, Instruments, & Computers*. For example, for real-time work, MEL is supported by technical people whom you can call. There are many statistics packages around. There are alternatives to Mathematica—for example, MAPLE V—matrix manipulation packages (e.g., Gauss), and there are a variety of word-processing packages, graphing tools, and so on. But it is the aim of this article to mention a range of tools that might suggest things that might have been missed. My hidden agenda for this discussion is that I may get some "Have you seen X" comments, and these will point me to new tools and methods.

BENCHMARKING WORKSTATIONS

One of the most important issues for people doing serious computational modeling is processing speed. In particular, connectionist and neural models can take hours or days to run, even on the fastest workstations (and even models that run quickly can be slowed by increasing the number of units in the model to make it more "plausible"). So a major concern is which machine is fastest and, relatedly, whether programs can be run fast enough to have several shots at the problem in 1 day, or whether they will take several days per run. This leads to the number 1 rule of benchmarking: Always run your own production code on the machines you are considering for purchase. If you cannot get access to the machines, find available benchmarks that correlate with your code on the machines and base your judgment on them. For this article, I was able to run my own programs as well as some designed by Jay McClelland; standard benchmarks are also reported.

Standard benchmarks for scientific computation are LINPACK, SPECint92, and SPECfp92. Vendors tune their machines for these benchmarks, but in general the



From Node

To Node	P	M	T	4	5	6	7	8	9	10	11	12	13	14
P	0	s_{mp}	0	1	1	1	1	0	0	0	0	0	0	0
M	s_{pm}	0	s_{tm}	0	0	0	0	1	1	1	0	0	0	0
T	0	s_{mt}	0	0	0	0	0	0	0	0	1	1	1	1
4	s_a	0	0	0	0	0	0	0	0	0	0	0	0	0
5	s_a	0	0	0	0	0	0	0	0	0	0	0	0	0
6	s_a	0	0	0	0	0	0	0	0	0	0	0	0	0
7	s_a	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	s_b	0	0	0	0	0	0	0	0	0	0	0	0
9	0	s_b	0	0	0	0	0	0	0	0	0	0	0	0
10	0	s_b	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	s_c	0	0	0	0	0	0	0	0	0	0	0
12	0	0	s_c	0	0	0	0	0	0	0	0	0	0	0
13	0	0	s_c	0	0	0	0	0	0	0	0	0	0	0
14	0	0	s_c	0	0	0	0	0	0	0	0	0	0	0

Figure 3. A sample spreading activation network and matrix of interconnections.

codes used in neural modeling follow SPECfp92 and LINPACK. The LINPACK benchmark is a standard matrix manipulation program, and the SPEC programs are combinations of real-life programs (e.g., fluid mechanics, chemistry, etc.). There are a number of design features that make benchmarking require more effort than simply following these standard benchmarks. One is the use of very fast cache memory as a buffer between the processor and main memory. If a program and data are small enough to fit in the cache, then performance will be many times faster than if they do not. For floating point computation, speed of transfer from main memory to cache memory is important (and this is a strong point of IBM RS6000 machines). Also, if the work load on the computer is high enough so that the program and/or data are swapped out of main memory and onto disk paging space, this can slow programs down a lot (as occurred with the large ptest program for the IBM RS6000-320 and SGI Indy, both of which had limited memory). This can be remedied by buying more main memory for the system. So, in benchmarking, it is necessary to examine a range of program and data sizes to see if there is a sudden drop in performance. Also, it is necessary to determine whether the program is paged out onto disk at any time (running the program using "time program" will give the number of page faults for the IBM machines). There are also a number of issues of program structure that can make huge differences in speed. For example, on the IBM RS6000, keeping arrays sequential is important. When arrays are used element by element without skipping around, they reside in cache and run much faster. I found that by rearranging code for backpropagation according to suggestions from the IBM tuning guide, I obtained a factor of 2 speedup.

In Table 5, I present the standard LINPACK and SPEC benchmarks along with three of my own programs and two of Jay McClelland's. My three programs are (1) an implementation of the brain-state-in-a-box model of J. A. Anderson (1991); (2) an implementation of the GRAIN model of McClelland (1993), which uses the mean field algorithm (Peterson & Hartman, 1989) and introduces variability into activation updates; and (3) a simulation of the diffusion process (Ratcliff, 1978), using an approximation from the simple random walk with small step sizes. These three programs are typical of the kinds of programs that take the longest to run in my research. The benchmarks devised by McClelland investigate the effect of array size on performance (cache buster) for neural-modeling-like computations. These enable performance to be assessed as the program size begins to exceed available cache memory. The atest routine accesses elements of an array sequentially (e.g., synchronous updating), and the ptest routine accesses memory randomly (asynchronous updates).

In running the benchmarks shown in Table 5, I had some problems using HP FORTRAN; the resulting reso-

Table 5
Sample Benchmarks for Fast Workstations

	Larger Numbers Better (atest, ptest, LINPACK in MFlops)							Smaller Numbers Better (Execution Times)		
Computer	Large atest	Large ptest	Small atest	Small ptest	LINPACK	SPEC92fp	SPEC92int	BSB	GRAIN	Rand
IBM RS 6000-320 (16 MHz)	8.0	0.06*	6.9	5.0	9.2	30.0	14.2	35.5	201.1	134.8
DEC Alpha 4000/710 (190 MHz)	19.0	3.6	31.7	19.4	39.3	185.1	122.6	9.4	55.6	15.9
HP 735 99 MHz	14.3	4.6	15.4	17.4	41.0	149.8	80.6	10.3	94.6	29.9
SGI Indy 50/100 MHz	7.8	*	12.1	12.1	12.0	60.7	59.1	33.5	158.5	32.6
SS 10/30 36 MHz	4.9	1.3	5.2	4.9	9.3	54.0	45.2	58.9	269.5	48.9
IBM RS 6000-590 (66 MHz)	36.4	13.3	26.7	21.0	130.4	242.4	117.0	6.0	44.0	31.0

Note—All compilers were run with the -O option. The atest, ptest, and LINPACK benchmarks are in units of MFlops, or millions of floating point instructions per second. Higher numbers indicate better performance. SPEC benchmarks are for sets of programs that are mainly dependent on integer performance (SPEC92int) and floating point performance (SPEC92fp). Higher numbers indicate better performance. The atest and ptest benchmarks mimic the computations needed for connectionist models with synchronous and asynchronous updates, respectively. The last three benchmarks are my own programs in FORTRAN. They represent the brain-state-in-a-box model (J. A. Anderson, 1991), the GRAIN model (McClelland, 1992), and a simulation version of the diffusion model (Ratcliff, 1978). The reported numbers show execution time, and the smaller numbers indicate better performance. Top of the line SGI Indy 2 with a 75/150-MHz processor (R4400) might run about 1.5 times faster than the SGI Indy above, and a top of the line SUN SPARCStation 10 might run 1.5 times faster than the SS 10 above. *Denotes a large number of page faults—that is, the program was shipped off to disk, which reduces performance significantly, relative to a machine with more memory. Performance would be much better with more system memory.

lution provides an example of the kinds of problems that can arise in benchmarking. The random walk simulation ran with no optimization, but the results were slightly different from those found with the other systems. When the program was optimized, it terminated with an error. It turned out that this was a problem with an assumption built into the random-number generator (ran3 from Numerical Recipes; Press, Flannery, Teukolsky, & Vetterling, 1986). The routine assumes that the variables are left untouched when the routine is entered a second time, and the optimizer interfered with that assumption. Compiling the program with a flag that set the variables "static" resulted in a run time of 38 sec (faster than the unoptimized 51 sec). Declaring the variables that were to be the same on subsequent calls in a common statement resulted in a run time of 30 sec—an improvement of 20%. The Rand program also failed when compiled with a POWER 2 code generation option on the RS6000-590.

The GRAIN simulation has a similar story; the original program ran much more slowly on the HP than it did on the DEC machine or even on the IBM RS6000-320. The problem on the HP machine was with calls to the built-in random-number generator, rand(). Using an old version for the generator produced times of 237 sec. The benchmarking contact person at HP extracted the random-number generator from a new unreleased version of the FORTRAN compiler, and the run time was 114 sec, a factor of 2 decrease in run time. Using a new unreleased version of the FORTRAN compiler produced a run time of 95 sec—another 20% improvement.

The conclusion from this is that, for programs that take a long time to run, it is important to do a profile analysis to find the routines that take a lot of time and to try to optimize these. For benchmarking, it is important to run your own code. When a choice of a few target machines has been reached, if it is possible to find an expert in the organization or in the company building the workstations, you might find ways to speed up your own applications beyond your wildest dreams (i.e., by over a factor of 2).

What is the bottom line for all of this? Well, you need to run your own code to be reasonably sure about speed. You also need to run the largest problem you are likely to deal with, because if small problems fit in the cache and a large problem exceeds cache, the speed can drop by as much as a factor of 10.

Another factor to be considered in assessing workstations is the cost of third-party software. For example, packages tend to cost three times as much for workstation software than for PC software. But this may change as emulators for PCs and Macs become available for workstations (these will not be blindingly fast, but will do the job). The prior discussion of processing speed ignores other performance improvements—faster disk speeds (SCSI 2), faster graphics, faster bus speeds, and so on—which should weigh in any choice of machine. A specific example of other improvements is compilation speed; the DEC ALPHA compiles 100- to 200-line programs in 1–2 sec, whereas other machines can take 10 sec or more. This difference can be important in developing and debugging programs that implement models. Besides speed, one of the most important issues is the quality of the software. Evaluation is important because faulty system software can eat up weeks of time. Problems can often be uncovered by personal contact and by reading appropriate newsgroups for a few weeks before a final decision. The number and type of complaints, as well as the questions being asked, can give clues to problems. Finally, local expertise is a critical component of any decision. If the department has a lot of one brand of machine and the system managers and programmers know and understand these machines, it is worth weighing this heavily into the cost/speed equation.

The results in Table 5 have to be cost factored into any decision about purchasing a machine. First, the DEC ALPHA is a high-cost machine, but there are versions that are only 8% slower and cost about $15,000. The HP machine is in the $35,000 range (list price), but there will be new versions out in the next few months that should be at least 50% faster and probably cheaper (this is probably true of all vendors). The SGI Indy is a low-end machine; top of the line machines in the $20,000 range should run 50% faster than the figures in Table 5 for the SGI Indy. The Indy can be obtained in the $10,000 range. I use the IBM RS6000-320, but it is aging in performance. The IBM RS6000-590 is a $64,000 machine that is based on the new POWER 2 architecture, though a workstation class machine is rumored. There are also some machines available that have multiple CPUs, which means that jobs that can be broken up into parts (such as running a model with different simulated subjects) can take advantage of this and run N times faster, where N is the number of processors. Currently these are expensive (SGI and SUN have such machines), but there are plans from other companies for multiprocessor machines.

For the integer application ("Rand"), the DEC machine is very fast. Two machines had memory that was insufficient to run the ptest benchmark without page faulting. Adding memory would bring the results up to roughly the same ratio as that for the other tests. The SPARCStation is remarkably slow, and even a 50-MHz machine (Sun's top of the line) would be hard pressed to beat the IBM RS6000-320 on floating point applications. The new IBM POWER 2 machine does produce, roughly, a fivefold increase in performance over my current machine. New machines from HP and DEC should push close to that target. The exceedingly high LINPACK performance figure for the RS6000-590 is probably cache related; the program probably remains in cache and therefore runs much faster. However, single-precision LINPACK is half as fast as double-precision LINPACK (double-precision values are reported in Table 5), so the double-precision values should be viewed as best case and possibly not typical of general floating point performance.

If running Mathematica is an important consideration, there is little information easily available. I posted a request for information to the Mathematica newsgroup, but there were no replies. For most computations in Mathematica, speed varies as a function of integer performance

(except matrix and other purely floating point operations within single commands), so integer performance would probably be the best predictor.

TWO CASE STUDIES IN MODELING

To illustrate the use of tools in modeling, I present two simple examples. The first case study is to fit data to the compound cue model for priming phenomena. The second is calculating the slope of the *z*-ROC function for a simple binomial model of memory.

Compound Cue Model of Priming

There is a current debate between Tim McNamara, and Gail McKoon and me about the relative merits of compound cue versus spreading activation models for priming phenomena. One of the foci of the debate concerns sequential effects in priming, and one specific issue concerns the weighting scheme in Gillund and Shiffrin's (1984) SAM model account of the experimental results. To model the processes of responding to sequential stimuli (words and nonwords), a joint cue consisting of a prime, the item that comes before the prime (the preprime), and a target is used to probe memory (e.g., Ratcliff & McKoon, 1988). The output of the model in response to the probe is a value of familiarity of the probe to the system. To account for sequential effects from a variety of different kinds of sequences, the target must receive most of the weight in the calculation of total familiarity for the probe, and the prior items must receive less weight. McNamara (1992) used a scheme in which the prime and preprime are weighted 0.3 and 0.2, respectively, which means that the target receives only half the weight, with the consequence that if the prime and preprime were both nonwords and the target was a word, responses would often be incorrect (in contradiction of the data). McNamara (in press) has argued that his scheme provides the best fits to the data (better than other weighting schemes used by McKoon & Ratcliff, 1992): "Clearly the best-fitting weights are 0.2, 0.3, and 0.5."

In Gillund and Shiffrin's (1984; see also Ratcliff & McKoon, 1988) implementation of the compound cue model, memory is assumed to be composed of strengths between each item in the compound cue and each item in memory. At test, the strength of each cue in the compound to each item in memory is determined. This is weighted (raised to the power of the weight), and these values are multiplied together to provide the degree of match between the compound cue and the item in memory. Individual match values are calculated for each item in memory and summed over all the items in memory to give familiarity, which is assumed to drive a decision process (e.g., diffusion process, Ratcliff, 1978) to produce reaction time and accuracy predictions. For small changes in familiarity, a linear relationship between familiarity and reaction time is assumed. So,

$$F = \Sigma_i s_{pp,i}^{c} s_{p,i}^{b} s_{t,i}^{a},$$

where i is the index for items; a, b, and c are the weights for the target, prime, and preprime, respectively; $s_{t,i}$ is the strength from the target cue to item i; $s_{p,i}$ is the strength from the prime cue to item i; and $s_{pp,i}$ is the strength from the preprime to item i. The matrix of cue to target strengths used in the following examples is shown in Table 6, along with the equation for familiarity.

Table 6
Cue to Target Strengths in the SAM Model

	Target									
Cue	1	2	3	4	5	6	7	8	9	10
1	1	1	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
2	1	1	1	0.2	0.2	0.2	0.2	0.2	0.2	0.2
3	0.2	1	1	1	0.2	0.2	0.2	0.2	0.2	0.2
4	0.2	0.2	1	1	1	0.2	0.2	0.2	0.2	0.2
5	0.2	0.2	0.2	1	1	1	0.2	0.2	0.2	0.2
6	0.2	0.2	0.2	0.2	1	1	1	0.2	0.2	0.2
7	0.2	0.2	0.2	0.2	0.2	1	1	1	0.2	0.2
8	0.2	0.2	0.2	0.2	0.2	0.2	1	1	1	0.2
9	0.2	0.2	0.2	0.2	0.2	0.2	0.2	1	1	1
10	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	1	1
11	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
12	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Note—Cues 11 and 12 are assumed to be nonwords with strengths of 0.1, the residual strengths from word cues to other words are assumed to be 0.2, and the strengths of words connected to each other are assumed to be 1. Familiarity is computed from

$$F(\text{cue } i, \text{ cue } j, \text{ cue } k) = \Sigma\, S_{il}^{w1}\, S_{jl}^{w2}\, S_{kl}^{w3},$$

where S_{il}^{w1} is the strength of cue i to target l with weight $w1$.

To examine the way the model works, a short program was written in S, as shown in Table 7. This is nonstochastic—that is, it does not have noise introduced as it would in a full version of the model, but it is useful for deriving asymptotic predictions. The S version contains the structure of the problem; it is compact and easy to understand. To see whether the model is capable of fitting the data without high weights on the prime and preprime, it is necessary to use a minimization routine. I have worked with the SIMPLEX minimization routine (from Ben Murdock's laboratory) since 1974, and I still find it extremely efficient for solving minimization problems. Table 8 contains a short FORTRAN subroutine to compute the difference between model predictions and data to serve the SIMPLEX routine. Writing and testing the program took about half an hour.

Table 7
S Code for a Nondeterministic SAM Compound Cue Model

```
x<-matrix(0.2,20,10);for(i in 11:20)x[i,]<-.1

for(i in 1:10){for(j in 1:10)
{if(abs(i-j)<2)x[i,j]<-1)}

a<-.7;b<-.2;cc<-.1

sum(x[2,]^a*x[5,]^b*x[8,]^cc)
sum(x[2,]^a*x[7,]^b*x[8,]^cc)
sum(x[2,]^a*x[8,]^b*x[3,]^cc)
sum(x[2,]^a*x[3,]^b*x[8,]^cc)
```

Table 8
Code for a SIMPLEX Solution to the SAM Compound Cue Model

```
      FUNCTION FOF(NV,X)
      REAL X(9),xx(10,20)
      FOF=0.0
      a=X(1)
      b=X(2)
      c=1.-a-b
      r=x(3)
      w=x(4)
      v=x(5)
      sc=x(6)
C r = residual word, w = connection strength
C related, v=resid nonword
C sc is fam to RT scaling factor
      do 1 j=1,10
      do 1 i=1,10
      xx(i,j)=r
      xx(i,j+10)=v
      if(abs(i-j).lt.2)xx(i,j)=w
    1 continue
      uu=0.
      x1=0.
      x2=0.
      x3=0.
      x4=0.
      x5=0.
      x6=0.
      do 2 i=1,10
      uu=uu+xx(i,2)**a*xx(i,5)**b*xx(i,8)**c
      x1=x1+xx(i,2)**a*xx(i,3)**b*xx(i,8)**c
      x2=x2+xx(i,2)**a*xx(i,8)**b*xx(i,3)**c
      x3=x3+xx(i,2)**a*xx(i,7)**b*xx(i,8)**c
      x4=x4+xx(i,2)**a*xx(i,3)**b*xx(i,15)**c
      x5=x5+xx(i,2)**a*xx(i,5)**b*xx(i,15)**c
    2 x6=x6+xx(i,2)**a*xx(i,15)**b*xx(i,8)**c
      x1=(x1-uu)*sc-30
      x2=(x2-uu)*sc-21
      x3=(x3-uu)*sc
      x4=(x4-uu)*sc
      x5=(x5-uu)*sc+30
      x6=(x6-uu)*sc+40
      x6=x6*2.
      FOF=x1*x1+x2*x2+x3*x3+x4*x4+x5*x5+x6*x6
      fof=fof*1000.
      return
      end
```

The substantive result is that the sequential data can be accommodated with low weights on the prime and preprime, contrary to McNamara's claim (see Table 10). Note that there is debate over some of the experimental findings—nonword inhibition in particular (see Ratcliff & McKoon, 1994)—and we use values from our work in Table 10. The parameters are shown in Table 9. The connection strength parameters are in the range of those found in Gillund and Shiffrin (1984; residual strengths are 0.2 and 0.1 for words and nonwords, respectively, and self- and interitem strengths are 1.6. It should also be noted that the model is constrained and could not fit all patterns of results. For example, all conditions with related prime and target have to have higher familiarity than those with unrelated prime and target, conditions with a nonword in them have to have lower familiarity than those with an unrelated word in them, and so on. Violations of these orderings would result in inadequate fits of the model.

From a computational point of view, the S program provides a rapid way of exploring the model. S is slow, but it only takes a few seconds to provide the results for the program shown in Table 7. However, in a fitting routine, where the function would have to be called hundreds of times or more, I preferred the FORTRAN program because it is extremely fast.

Z-ROC Functions for the Attention Likelihood Theory of Glanzer et al. (1993)

Glanzer, Adams, Iverson, and Kim (1993) have proposed an attention likelihood ratio model for recognition memory. Their model is basically a feature model designed to explain ROC functions for recognition. The model assumes that items to be remembered are composed of features. A certain number of features are "marked" prior to studying a list of words. As a result of study, more of these features become marked. When an item is tested, a sample of the features is selected and the proportion that are marked is used to decide whether the item was studied or not. Rather than using a fixed criterion based on the number of marked features observed in the sample examined at test, a likelihood ratio is used to determine whether the item came from the old-item distribution or the new-item distribution. Essentially, for a particular score, the probability of it being an old item is divided by the probability of it being a new item, and the result is compared with a criterion (larger than criterion, respond "old," smaller, respond "new").

The model is specifically designed to explain the mirror effect in recognition memory. The mirror effect is the finding that when a materials difference produces better performance in recognition (e.g., low-frequency words are better recognized than high-frequency words), the hit rate is higher and the false-alarm rate is lower for the ma-

Table 9
Parameter Values for SAM Compound Cue Model

Parameter of Model	Value
Target weight	0.715
Prime weight	0.166
Preprime weight	0.118
Strength for connected words	1.680
Residual strength for words	0.219
Residual strength for nonwords	0.109
Strength to RT multiplier	53.5*

*In milliseconds.

Table 10
Fits of SAM Compound Cue Model to Rounded Data

Condition Relative to UUU Baseline Preprime, Prime, Target	Fit*	Data*
URR	30.6	30
RUR	20.9	21
RRU	2.6	0
XRR	−0.8	0
XUU	−28.9	−30
UXU	−40.0	−40
XXU	−65.7	????

*In milliseconds. U denotes an unrelated word; R, a word related to the other R word; and X, a nonword.

terials with better performance. To explain the mirror effect for word frequency, the attention likelihood model assumes that low-frequency words are more attention grabbing than high-frequency words, so the number of sampled features at test is larger and the probability of marking features at study is larger.

The likelihood transformation has no effect on the ROC functions. For a criterion setting that produces a particular hit rate and false-alarm rate on a likelihood scale, an equivalent criterion setting can be found on the feature count scale. Glanzer, Adams, and Iverson (1991) assumed that the normal approximation to the binomial distribution held, so that the slope of the z-ROC function would be the ratio of the standard deviations for the new- and old-item distributions. However, for the parameter values found in Glanzer et al. (1991), the approximation does not hold.

In the experimental data, Glanzer et al. (1991; see also Ratcliff et al., in press, for replication) found that the slope of the z-ROC function was typically about 0.7–0.8 for high-frequency words and 0.6–0.7 for low-frequency words. These experiments were performed using a confidence judgment procedure that essentially produces hit and false-alarm rates for various values of criterion setting. When the ROC functions are computed directly from the probability density functions (the ROC functions are the same whether the distributions are transformed by the likelihood transformation or not), the slope is considerably larger—in the 0.9 range rather than 0.6–0.8. To find ratios near 0.6 in the model, extreme distributions are required (e.g., highly nonsymmetric; see Ratcliff et al., in press).

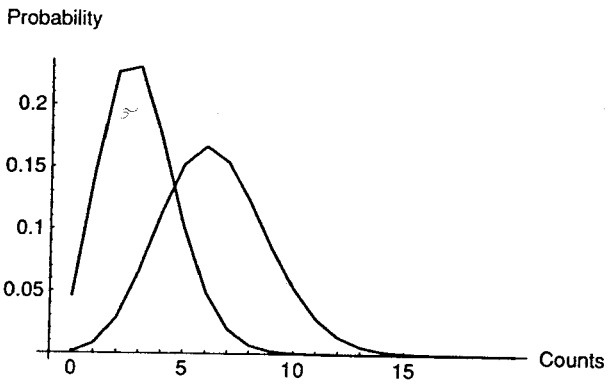


Figure 4. Binomial density functions from Glanzer et al.'s (1993) feature marking model. The right-hand distribution has $p = .107$, the left-hand distribution has $p = .05$, and the number of features is 60.

Table 11
Mathematica Code for Computing the Slope of *Z*-ROC Functions for Binomial Distributions

```
<<Statistics`DiscreteDistributions`
<<Statistics`ContinuousDistributions`
<<Statistics`InverseStatisticalFunctions`

n=60;pn=.05;po=.107;

ndn=BinomialDistribution[n,pn];
ndo=BinomialDistribution[n,po];
ndl=NormalDistribution[0,1.];

w={0.,0.,0.,0.,0.,0.,0.,0.,0., etc.}
x={0.,0.,0.,0.,0.,0.,0.,0.,0., etc.}
y={0.,0.,0.,0.,0.,0.,0.,0.,0., etc.}
z={0.,0.,0.,0.,0.,0.,0.,0.,0., etc.}

Do[y[[i+1]]=1.0-CDF[ndn,i],{i,0,20}];
Do[z[[i+1]]=1.0-CDF[ndo,i],{i,0,20}];

Do[w[[i]]=Quantile[ndl,y[[i]]],{i,1,21}]
Do[x[[i]]=Quantile[ndl,z[[i]]],{i,1,21}]

ww={w,x}; p=Transpose[ww];
a2=ListPlot[p];
pp=Fit[p,{1,xx},xx]
Result is 1.60 + 0.906 xx

a1=Plot[pp,{xx,-7,2}]
Show[{a2,a1},DefaultFont->{"Helvetica",16},
AxesLabel->{"Zfa","Zhit"}]

Ratio of standard deviations for likelihood
  transformed distributions

varn=n*pn*(1-pn)*Log[po*(1-pn)/(pn*(1-po))]^2
varo=n*po*(1-po)*Log[po*(1-pn)/(pn*(1-po))]^2

Sqrt[varn/varo]
Result is 0.705
```

To perform these z-ROC analyses, I used the Mathematica code that is shown in Table 11. The parameters of the model for this example (typical of those used by Glanzer et al., 1991) are shown in the fourth line of Table 11—number of features, $n = 60$, probability marked when the item is new = .05, and probability marked when the item is old = .107. These values give rise to the distributions shown in Figure 4.

The Mathematica code is easily described. First, packages are loaded. Then the old-item and new-item binomial distributions are defined, the standard normal distribution (to provide z scores) is defined, and some vectors are initialized. The first two "Do" statements set up one minus the cumulative density function, and the next two compute z scores of those values to provide z_{hit} and z_{fa}. All that is left is to plot the z scores and fit a straight line in the Fit routine. As can be seen, the code is quite compact. The next three lines computing "varo" and "varn" provide the ratio of standard deviations for the likelihood ratio model. The standard deviation ratio is 0.7, whereas the slope of the z-ROC is 0.9—a substantial difference.

The substantive point here is that likelihood ratio theory has problems predicting the slope of the z-ROC functions with the parameter values promoted by Glanzer et al. (1991). This is a problem for the particular values used in Glanzer et al.'s articles, but it may be that more extreme values of p_{new} and p_{old} could give a better fit to the data. However, it is clear that very small values of p_{new} are needed, which produce highly skewed binomial distributions.

Computational tools such as Mathematica—with built-in distributions (normals and binomials), cumulative distribution functions, quantiles, fitting tools, and plotting tools—allow rapid examination of simple theoretical models. To do this in a high-level language would require these functions. I also tried this using the S language; it handles densities, distributions, and so on very well, but my version was compiled as single precision, and values in the tails of the distribution provided numerical inaccuracy and numerical overflow and underflow.

CONCLUSIONS

I have reviewed a number of tools used in real-time experimental work and modeling and it is fair to conclude that, with a lot of work, the routine tasks involved in these research methods can be made considerably simpler. For experimental work, the real-time system, list generation tools, and data analysis tools have speeded experimental work by a factor of 4 or 5 from straight programming, while leaving the generality untouched. For computational modeling, I have found that FORTRAN (other researchers, read C) is fast, and it is optimized for speed on most workstations because of the investment in routines by people in physics, engineering, chemistry, and other such disciplines. However, over the last 5 years I have found that high-level languages such as Mathematica and S are extremely useful. The languages trade speed for both compactness and variety of built-in functions plus extremely powerful graphing functions. I have been very pleased with the payout for the investment I have put into learning these languages and tools, and I think that a similar investment would be beneficial to most researchers.

REFERENCES

ANDERSON, J. A. (1991). Why, having so many neurons, do we have so few thoughts? In W. E. Hockley & S. Lewandowsky (Eds.), *Relating theory and data: Essays on human memory in honor of Bennet B. Murdock* (pp. 477-507). Hillsdale, NJ: Erlbaum.

ANDERSON, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

GILLUND, G., & SHIFFRIN, R. M. (1984). A retrieval model for both recognition and recall. *Psychological Review*, **91**, 1-67.

GLANZER, M., ADAMS, J. K., & IVERSON, G. (1991). Forgetting and the mirror effect in recognition memory: Concentering of underlying distributions. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, **17**, 81-93.

GLANZER, M., ADAMS, J. K., IVERSON, G. J., & KIM, K. (1993). The regularities of recognition memory. *Psychological Review*, **100**, 546-567.

GREENE, S., RATCLIFF, R., & MCKOON, G. (1988). A flexible programming language for generating stimulus lists for cognitive psychology experiments. *Behavior Research Methods, Instruments, & Computers*, **20**, 119-128.

HACKER, M. J., & ANGIOLILLO-BENT, J. S. (1981). A BASIC package for N-way ANOVA with repeated measures, trend analysis, and user-defined contrasts. *Behavior Research Methods & Instrumentation*, **13**, 688.

KINTSCH, W. (1974). *The representation of meaning in memory*. Hillsdale, NJ: Erlbaum.

MCCLELLAND, J. L. (1993). Toward a theory of information processing in graded, random, interactive networks. In D. E. Meyer & S. Kornblum (Eds.), *Attention and performance XIV: Synergies in experimental psychology, artificial intelligence and cognitive neuroscience* (pp. 655-688). Cambridge, MA: MIT Press.

MCKOON, G., & RATCLIFF, R. (1992). Spreading activation versus compound cue accounts of priming: Mediated priming revisited. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, **18**, 1155-1172.

MCNAMARA, T. P. (1992). Priming and constraints it places on theories of memory and retrieval. *Psychological Review*, **99**, 650-662.

MCNAMARA, T. P. (in press). Theories of priming: II. Types of primes. *Journal of Experimental Psychology: Learning, Memory, & Cognition*.

PETERSON, C., & HARTMAN, E. (1989). Explorations of the mean field theory learning algorithm. *Neural Networks*, **2**, 475-494.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., & VETTERLING, W. T. (1986). *Numerical recipes*. Cambridge: Cambridge University Press.

RATCLIFF, R. (1978). A theory of memory retrieval. *Psychological Review*, **85**, 59-108.

RATCLIFF, R. (1993). Methods for dealing with reaction time outliers. *Psychological Bulletin*, **114**, 510-532.

RATCLIFF, R., & LAYTON, W. M. (1981). A microcomputer interface for control of real-time experiments in cognitive psychology. *Behavior Research Methods & Instrumentation*, **13**, 216-220.

RATCLIFF, R., & MCKOON, G. (1988). A retrieval theory of priming in memory. *Psychological Review*, **95**, 385-408.

RATCLIFF, R., & MCKOON, G. (1994). *Sequential effects in lexical decision: Tests of compound cue theory*. Manuscript submitted for publication.

RATCLIFF, R., & MCKOON, G. (in press). Bias and explicit memory in priming of object decisions. *Journal of Experimental Psychology: Learning, Memory, & Cognition*.

RATCLIFF, R., MCKOON, G., & TINDALL, M. H. (in press). The effects of rapid presentation, list length, word frequency, and category membership on recognition memory ROC functions. *Journal of Experimental Psychology: Learning, Memory, & Cognition*.

RATCLIFF, R., PINO, C., & BURNS, W. T. (1986). An inexpensive real-time microcomputer-based cognitive laboratory system. *Behavior Research Methods, Instruments, & Computers*, **18**, 214-221.