

Bayesian inference with Stan: A tutorial on adding custom distributions

Jeffrey Annis¹ · Brent J. Miller¹ · Thomas J. Palmeri¹

Published online: 10 June 2016
© Psychonomic Society, Inc. 2016

Abstract When evaluating cognitive models based on fits to observed data (or, really, any model that has free parameters), parameter estimation is critically important. Traditional techniques like hill climbing by minimizing or maximizing a fit statistic often result in point estimates. Bayesian approaches instead estimate parameters as posterior probability distributions, and thus naturally account for the uncertainty associated with parameter estimation; Bayesian approaches also offer powerful and principled methods for model comparison. Although software applications such as WinBUGS (Lunn, Thomas, Best, & Spiegelhalter, *Statistics and Computing*, 10, 325–337, 2000) and JAGS (Plummer, 2003) provide “turnkey”-style packages for Bayesian inference, they can be inefficient when dealing with models whose parameters are correlated, which is often the case for cognitive models, and they can impose significant technical barriers to adding custom distributions, which is often necessary when implementing cognitive models within a Bayesian framework. A recently developed software package called *Stan* (Stan Development Team, 2015) can solve both problems, as well as provide a turnkey solution to Bayesian inference. We present a tutorial on how to use *Stan* and how to add custom distributions to it, with an example using the linear ballistic accumulator model (Brown & Heathcote, *Cognitive Psychology*, 57, 153–178. doi:10.1016/j.cogpsych.2007.12.002, 2008).

Electronic supplementary material The online version of this article (doi:10.3758/s13428-016-0746-9) contains supplementary material, which is available to authorized users.

✉ Jeffrey Annis
jeff.annis@vanderbilt.edu

¹ Vanderbilt University, 111 21st Ave S., 301 Wilson Hall, Nashville, TN 37240, USA

Keywords Bayesian inference · Stan · Linear ballistic accumulator · Probabilistic programming

The development and application of formal cognitive models in psychology has played a crucial role in theory development. Consider, for example, the near ubiquitous applications of accumulator models of decision making, such as the diffusion model (see Ratcliff & McKoon, 2008, for a review) and the linear ballistic accumulator model (LBA; Brown & Heathcote, 2008). These models have provided theoretical tools for understanding such constructs as aging and intelligence (e.g., Ratcliff, Thapar, & McKoon, 2010) and have been used to understand and interpret data from functional magnetic resonance imaging (Turner, Van Maanen, Forstmann, 2015; Van Maanen et al., 2011), electroencephalography (Ratcliff, Philiastides, & Sajda, 2009), and neurophysiology (Palmeri, Schall, & Logan, 2015; Purcell, Schall, Logan, & Palmeri, 2012). Nearly all cognitive models have free parameters. In the case of accumulator models, these include the rate of evidence accumulation, the threshold level of evidence required to make a response, and the time for mental processes not involved in making the decision. Unlike general statistical models of observed data, the parameters of cognitive models usually have well-defined psychological interpretations. This makes it particularly important that the parameters be estimated properly, including not just their most likely value, but also the uncertainty in their estimation.

Traditional methods of parameter estimation minimize or maximize a fit statistic (e.g., SSE , χ^2 , $\ln L$) using various hill-climbing methods (e.g., *simplex* or *Hooke and Jeeves*). The result is usually point estimates of parameter values, and possibly later applying such techniques as parametric or nonparametric bootstrapping to obtain indices of the uncertainty of those estimates (Lewandowsky & Farrell, 2011). By contrast,

Bayesian approaches to parameter estimation naturally treat model parameters as full probability distributions (Gelman, Carlin, Stern, Dunson, Vehtari, & Rubin, 2013; Kruschke, 2011; Lee & Wagenmakers, 2014). By so doing, the uncertainty over the range of potential parameter values is also estimated, rather than a single point estimate.

Whereas a traditional parameter estimation method might find some vector of parameters, θ , that maximizes the likelihood of the data, D , given those parameters [$P(D | \theta)$], a Bayesian method will find the entire posterior probability distribution of the parameters given the data, $P(\theta | D)$, by a conceptually straightforward application of Bayes's rule: $P(\theta | D) = P(D | \theta) P(\theta) / P(D)$. A virtue—though some argue it is a curse—of Bayesian methods is that they allow the researcher to express their a priori beliefs (or lack thereof) about the parameter values, as a *prior distribution* $P(\theta)$. If a researcher thinks all values are equally likely, they might choose a uniform or otherwise flat distribution to represent that belief; alternatively, if a researcher has reason to believe that some parameters might be more likely than others, that knowledge can be embodied in the prior as well. Bayes provides the mechanism to combine the prior on parameter values, $P(\theta)$, with the *likelihood* of the data given certain parameter values, $P(D | \theta)$, resulting in the *posterior distribution* of the parameters given the data, $P(\theta | D)$.

Bayes is completely generic. It could be used with a model having one parameter or one having dozens or hundreds of parameters. It could rely on a likelihood based on a well-known probability distribution, like a normal or a Gamma distribution, or it could rely on a likelihood of response times predicted by a cognitive model like the LBA.

Although the application of Bayes is conceptually straightforward, its application to real data and real models is anything but. For one thing, the calculation of the probability of the data term in the denominator, $P(D)$, involves a multivariate integral, which can be next to impossible to solve using traditional techniques for all but the simplest models. For small models with only one or two parameters, the posterior distribution can sometimes be calculated directly using calculus or can be reasonably estimated using numerical methods. However, as the number of parameters in the model grows, direct mathematical solutions using calculus become scarce, and traditional numerical methods quickly become intractable. For more sophisticated models, a technique called Markov chain Monte Carlo (MCMC) was developed (Brooks, Gelman, Jones, & Meng, 2011; Gelman et al., 2013; Gilks, Richardson, & Spiegelhalter, 1996; Robert & Casella, 2004). MCMC is a class of algorithms that utilize Markov chains to allow one to approximate the posterior distribution. In short, a given MCMC algorithm can take the prior distribution and likelihood as input and generate random samples from the posterior distribution without having to have a closed-form solution or numeric estimate for the desired posterior distribution.

The first MCMC algorithm was the *Metropolis–Hastings algorithm* (Metropolis, Rosenbluth, Rosenbluth, Teller, & Teller, 1953; Hastings, 1970), and it is still popular today as a default MCMC method. On each step of the algorithm a proposal sample is generated. If the proposal sample has a higher probability than the current sample, then the proposal is accepted as the next sample; otherwise, the acceptance rate is dependent upon the ratio of the posterior probabilities of the proposal sample and the current sample. The “magic” of MCMC algorithms like Metropolis–Hastings is that they do not require calculating the nasty $P(D)$ term in Bayes's rule. Instead, by relying on ratios of the posterior probabilities, the $P(D)$ term cancels out, so the decision to accept or reject a new sample is based solely on the prior and the likelihood, which are given. The proposal step is generated via a random process that must be “tuned” so that the algorithm efficiently samples from the posterior. If the proposals are wildly different from or too similar to the current sample, the sampling process can become very inefficient. Poorly tuned MCMC algorithms can lead to samples that fail to meet minimum standards for approximating the posterior distribution or to Markov chain lengths that become computationally intractable on even the most powerful computer workstations.

A different type of MCMC algorithm that largely does away with the difficulty of sampler tuning is *Gibbs sampling*. Several software applications have been built around this algorithm (WinBUGS: Lunn, Thomas, Best, & Spiegelhalter, 2000; JAGS: Plummer, 2003; OpenBUGS: Thomas, O'Hara, Ligges, & Sturtz, 2006). These applications allow the user to easily define their model in a specification language and then generate the posterior distributions for the respective model parameters. The fact that the Gibbs sampler does not require tuning makes these applications effectively “turnkey” methods for Bayesian inference. These applications can be used for a wide variety of problems and include a number of built-in distributions; one must only specify, for example, that the prior on a certain parameter is distributed uniformly or that the likelihood of the data given the parameter is normally distributed. Although these programs provide dozens of built-in distributions, researchers inevitably will discover that some particular distribution they are interested in is not built into the application. This will often be the case for specialized cognitive models whose distributions are not part of the standard suites of built-in distributions that come with these applications. Thus, it is necessary for the researcher who wishes to use one of these Bayesian inference applications to add a custom distribution to the application's distribution library. This process, however, can be technically challenging using most of the applications listed above (see Wabersich & Vandekerckhove, 2014, for a recent tutorial with JAGS).

In addition to the technical challenges of adding custom distributions, both the Gibbs and Metropolis–Hastings algorithms often do not sample efficiently from posterior distributions with correlated parameters (Hoffman & Gelman, 2014; Turner,

Sederberg, Brown, & Steyvers, 2013). Some MCMC algorithms (e.g., MCMC-DE; Turner et al., 2013) are designed to solve this problem, but these algorithms often require careful tuning of the MCMC algorithm parameters to ensure efficient sampling of the posterior. In addition, implementing models that use such algorithms can be more difficult than implementing models in turnkey applications, because the user must work at the implementation level of the MCMC algorithm.

Recently, a new type of MCMC application has emerged, called *Stan* (Hoffman & Gelman, 2014; Stan Development Team, 2015). Stan uses the No-U-Turn Sampler (NUTS; Hoffman & Gelman, 2014) which extends a type of MCMC algorithm known as *Hamiltonian Monte Carlo* (HMC; Duane, Kennedy, Pendleton, & Roweth, 1987; Neal, 2011) NUTS requires no tuning parameters and can efficiently sample from posterior distributions with correlated parameters. It is therefore an example of a turnkey Bayesian inference application that allows the user to work at the level of the model without having to worry about the implementation of the MCMC algorithm itself. In this article, we provide a brief tutorial on how to use the Stan modeling language to implement Bayesian models in Stan using both built-in and user-defined distributions; we do assume that readers have some prior programming experience and some knowledge of probability theory.

Our first example is the exponential distribution. The exponential is built into the Stan application. We will first define the model statistically, and then outline how to implement a Bayesian model based on the exponential distribution using Stan. Following this implementation, we will show how to run the model and collect samples from the posterior. As we will see, one way that this can be done is by interfacing with Stan via another programming language, such as R. The command to run the Stan model is sent from R, and then the samples are sent back to the R workspace for further analysis.

Our second example will again consider the exponential distribution, but this time instead of using the built-in exponential distribution, we will explicitly define the likelihood function of the exponential distribution using the tools and techniques that allow the user to define distributions in Stan.

Our third example will illustrate how to implement a more complicated user-defined function in Stan—the LBA model (Brown & Heathcote, 2008). We will then show how to extend this model to situations with multiple participants and multiple conditions.

Throughout the tutorial, we will benchmark the results from Stan against a conventional Metropolis–Hastings algorithm. As we will see, Stan performs equally as well as Metropolis–Hastings for the simple exponential model, but much better for more complex models with correlated dimensions, such as the LBA. We are quite certain that suitably tuned versions of MCMC-DE (Turner et al., 2013) and other more sophisticated methods would perform at least as well as Stan. The goal here was not to make fine distinctions between alternative successful

applications, but to illustrate how to use Stan as an application that perhaps may be adopted more easily by some researchers.

Built-in distributions in Stan

In Stan, a Bayesian model is implemented by defining its likelihood and priors. This is accomplished in a Stan program with a set of variable declarations and program statements that are displayed in this article using *Courier* font. Stan supports a range of standard variable types, including integers, real numbers, vectors, and matrices. Stan statements are processed sequentially and allow for standard control flow elements, such as `for` and `while` loops, and conditionals such as `if-then` and `if-then-else`.

Variable definitions and program statements are placed within what are referred to in Stan as *code blocks*. Each code block has a particular function within a Stan program. For example, there is a code block for user-defined functions, and others for data, parameters, model definitions, and generated quantities. Our tutorial will introduce each of these code blocks in turn.

To make the most out of this tutorial, it will be necessary to install both Stan (<http://mc-stan.org/>) and R (<https://cran.r-project.org/>), as well as the RStan package (<http://mc-stan.org/interfaces/rstan.html>) so that R can interface with Stan. Step-by-step instructions for how to do all of this can be found online (<http://mc-stan.org/interfaces/rstan.html>).

An example with the exponential distribution

In this section, we provide a simple example of how to use Stan to implement a Bayesian model using built-in distributions. For simplicity, we will use a one-parameter distribution: the exponential. To begin, suppose that we have some data (y) that appear to be exponentially distributed. We can write this in more formal terms with the following definition:

$$y \sim \text{Exponential}(\lambda), \quad (1)$$

This asserts that the data points (y) are assumed to come from an exponential distribution, which has a single parameter called the rate parameter, λ . Using traditional parameter-fitting methods, we might find the value of λ that maximized the likelihood of observed data that we thought followed an exponential. Because here we are using a Bayesian approach, we can conceive of our parameters as probability distributions.

What distribution should we choose as the prior on λ ? The rate parameter of the exponential distribution is bounded between zero and infinity, so we should choose a distribution with the same bounds. One distribution that fits this criterion is the Gamma distribution. Formally, then, we can write

$$\lambda \sim \text{Gamma}(\alpha, \beta). \quad (2)$$

The Gamma distribution has two parameters, referred to as the *shape* (α) and *rate* (β) parameters; to represent our prior beliefs about what parameter values are more likely than others, we chose weakly informative shape and rate parameters of 1. So, Eq. 1 specifies our likelihood, and Eq. 2 specifies our prior. This completes the specification of the Bayesian model in mathematical terms. The next section shows how to easily implement the exponential model in the Stan modeling language.

Stan code To implement this model in Stan, we first open a new text file in any basic text editor. Note that the line numbers in the code-text boxes in this article are for reference and are *not* part of the actual Stan code. The body of every code block is delimited using curly braces `{ }`; Stan programming statements are placed within these curly braces. All statements must be followed by a semicolon.

In the text file, we first declare a `data` block as is shown in Box 1. The `data` code block stores all of the to-be-modeled variables containing the user’s data. In this example, we can see that the data we will pass to the Stan program are contained in a vector of size `LENGTH`. It is important to note that the data are not explicitly defined in the Stan program itself. Rather, the Stan program is interfaced via the command line or an alternative method (like RStan), and the data are passed to the Stan program in that way. We will describe this procedure in a later section.

Box 1 Stan code for the exponential model (in all Boxes, line numbers are included for illustration only)

```

1  data{
2    int LENGTH;
3    vector[LENGTH] Y;
4  }
5  parameters{
6    real<lower=0> lambda;
7  }
8  model{
9    real alpha;
10   real beta;
11   alpha <- 1.0;
12   beta <- 1.0;
13   lambda ~ gamma(alpha,beta);
14   Y ~ exponential(lambda);
15 }
16 generated quantities{
17   real pred;
18   pred <- exponential_rng(lambda);
19 }

```

The next block of code is the `parameters` block, in which all parameters contained in the Stan model are declared. Our exponential model has only a single parameter λ . Here, that parameter `lambda` is a real number of type `real` and is bounded on the interval $[0, \infty)$, so we must constrain our

variable within that range in Stan. We do this by adding the `<lower=0>` constraint as part of its definition.

The third block of code is the `model` block, in which the Bayesian model is defined. The model described by Eqs. 1 and 2 is easily implemented, as is shown in Box 1. First, the variables of the Gamma prior, `alpha` and `beta`, are defined as real numbers of type `real`, and both are assigned our chosen values of 1.0. Note that unlike the variables in the `data` and `parameters` blocks, variables defined in the `model` block are *local* variables. This means that their scope does not extend beyond the block in which they are defined; in less technical terms, other blocks do not “know” about variables initialized in the `model` block.

After having defined these local variables, the next part of the `model` block defines a *sampling statement*. The sampling statement `lambda ~ gamma(alpha,beta)` indicates that the prior on `lambda` is sampled from a Gamma distribution with rate and shape parameters `alpha` and `beta`, respectively. Note that sampling statements contain a tilde character (`~`), distinct from the assignment character (`<-`) in Stan. The next statement, `Y ~ exponential(lambda)`, is also a sampling statement and indicates that the data `Y` are exponentially distributed with rate parameter `lambda`.

The final block of code in the Stan file is the `generated quantities` block. This block can be used, for example, to perform what is referred to as a *posterior predictive check*. The purpose of the check is to determine whether the model accurately fits the data in question; in other words, this lets us compare model predictions with the observed data. Box 1 shows how this is accomplished in Stan. First, a real-number variable named `pred` is created. This variable will contain the predictions of the model. Next, the exponential random number generator (RNG) function `exponential_rng` takes as input the posterior samples of `lambda` and outputs the posterior prediction. The posterior prediction is returned from Stan and can be used outside of Stan—for example, to compare the predictions to the actual data in order to assess visually how well the model fits the data.

This completes the Stan model. When all the code has been entered into the text file, we save the file as `exponential.stan`. Stan requires that the extension `.stan` be used for all Stan model files.

R code Stan can be interfaced from the command line, R, Python, MATLAB, or Julia.¹ In this article, we will describe how to use the R interface for Stan, called RStan (Stan Development Team, 2015). Links to online instructions for how to install R and RStan were given earlier.

¹ Julia is a relatively new high-level programming language for scientific and technical computing, similar in nature to R, MATLAB, and Python (<http://julialang.org/>).

In this section, we will be first simulating data and then fitting the model to those simulated data. This is in contrast to most real-world applications, in which models are fit to the actual observed data from an experiment. It is good practice before fitting a model to real data to fit the model to simulated data with known parameter values and try to recover those values. If the model cannot recover the known parameter values of the model that generated the simulated data, then it will never be able to be fitted with any confidence to real observed data. This type of exercise is usually referred to as *parameter recovery*. Here, this also serves us well in a tutorial capacity.

Box 2 shows the R code that will run the parameter recovery example. The first three lines of the R code clear the workspace (line 1), set the working directory (line 2), and load the RStan library (line 3). Then we generate some simulated data, drawing 500 exponentially distributed samples (line 5) assuming a rate parameter, λ , equal to 1. These simulated data, `dat`, will then be fed into Stan to obtain parameter estimates for λ . If the Stan implementation is working correctly, we should obtain a posterior distribution of λ that is centered over 1. So far, all of this is just standard R code.

Box 2 R code for running the exponential model in Stan and for retrieving and analyzing the returned posterior samples

```

1 rm(list=ls())
2 setwd("~/Exponential/")
3 library(rstan)
4 lambda = 1
5 dat = rexp(500,lambda) #simulated data
6 len = length(dat)
7 #run the Stan model
8 fit <- stan(file = "exponential.stan",
              data = list(Y=dat, LENGTH=len),
              warmup = 750,
              iter = 1500,
              chains = 3)
9 #model summary
10 print(fit)
11 #posterior predictions
12 mcmc_chain = as.matrix(fit)
13 #autocorrelation plot
14 acf(mcmc_chain[, 'lambda'])
15 #samples vs. iteration plot
16 traceplot(fit)
17 #posterior predictions
18 pred = mcmc_chain[, 'pred']
19 hist(dat, probability=T)
20 lines(density(pred))
21 #posterior of lambda
22 plot(density(mcmc_chain[, 'lambda']))

```

The Stan model described earlier (`exponential.stan`) is run via the `stan` function. This is the way that R “talks” to Stan, tells it what to run, and gets back the results of the Stan run. The first argument of the function, `file`, is a character string that defines the location and name of the Stan model file. This is simply the Stan file from Box 1. The data argument is a list containing the data to be passed to the Stan program. Stan will be expecting a variable named `Y` to be holding the data (see line 3 of the Stan code in Box 1). We assign the variable `dat` (in this case, our simulated draws

from an exponential) the name `Y` in line 8 so that Stan knows that these are the data. Stan also expects a variable named `LENGTH` to be holding the length of the data vector `Y`. We assign the variable `len` (the length of `dat` computed in line 6) the name `LENGTH`. This is the way that R feeds data into Stan. Next, the `warmup` argument defines the number of steps used to automatically tune the sampler in which Stan optimizes the HMC algorithm. These samples can be discarded afterward and are referred to as `warmup` samples. The `iter` argument defines the total number of iterations the algorithm

will run. Choosing the number of iterations and warmup steps usually proceeds by starting with relatively small numbers and then doubling them, each time checking for convergence (discussed below). It is recommended that `warmup` be half of `iter`. The `chains` argument defines the number of independent chains that will be consecutively run. Usually, at least three chains are run. After running the model, the samples are returned and assigned to the `fit` object.

A summary of the parameter distributions can be obtained by using `print(fit)`² (line 10), which provides posterior estimates for each of the parameters in the model. Before any inferences can be made, however, it is critically important to determine whether the sampling process has converged to the posterior distribution. Convergence can be diagnosed in several different ways. One way is to look at convergence statistics such as the *potential scale reduction factor*, \hat{R} (Gelman & Rubin, 1992), and the *effective number of samples*, N_{eff} (Gelman et al., 2013), both of which are output in the summary statistics with `print(fit)`. A rule of thumb is that when \hat{R} is less than 1.1, convergence has been achieved; otherwise, the chains need to be run longer. The N_{eff} statistic gives the number of independent samples represented in the chain. For example, a chain may contain 1,000 samples, but this may be equivalent to having drawn very few independent samples from the posterior. The larger the effective sample size, the greater the precision of the MCMC estimate. To give an estimate of an acceptable effective sample size, Gelman et al. (2013) recommended an N_{eff} of 100 for most applications. Of course, the target N_{eff} can be set higher if greater precision is desired.

Both the \hat{R} and N_{eff} statistics are influenced by what is referred to as *autocorrelation*. To give an example, adjacent samples usually have some amount of correlation, due to the way that MCMC algorithms work. However, as the samples become more distant from each other in the chain, this correlation should decrease quickly. The distance between successive samples is usually referred to as the *lag*. The *autocorrelation function* (ACF) relates correlation and lag. The values of the ACF should quickly decrease with increasing lag; ACFs that do not decrease quickly with lag often indicate that the sampler is not exploring the posterior distribution efficiently and result in increased \hat{R} values and decreased N_{eff} values.

The ACF can easily be plotted in R on lines 12 and 14. The separate chains are first collapsed into a single chain with `as.matrix(fit)` (the `as.matrix` function is part of the `base` package in R), and the ACF of `lambda` is plotted with `acf(mcmc_chain[, 'lambda'])`, shown in Fig. 1. The `acf` function is part of the `stats` package, a base

package in R that is loaded automatically when R is opened. The left panel of Fig. 1 shows that the autocorrelation drops to values close to zero at around lags of six for the samples returned by Stan. The Metropolis–Hastings algorithm has slightly higher autocorrelation but is still reasonable in this example.

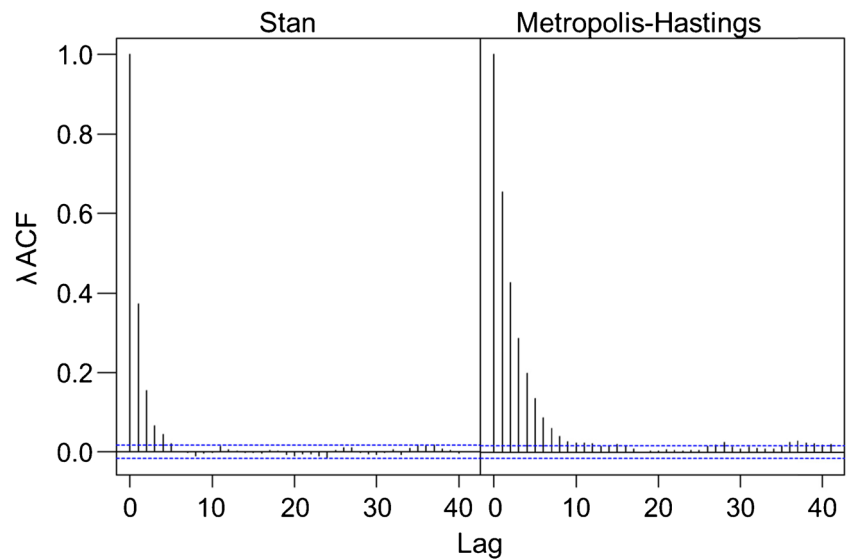
High autocorrelation indicates that the sampler is not efficiently exploring the posterior distribution. This can be overcome by simply running longer chains. By running longer chains, the sampler is given the chance to explore more of the distribution. The technique of running longer chains, however, is sometimes limited by memory and data storage constraints. One way to run very long chains and reduce memory overhead is to use a technique called *thinning*, which is done by saving every n th posterior sample from the chain and discarding the rest. Increasing n reduces autocorrelation as well as the resulting size of the chain. Although thinning can reduce autocorrelation and chain length, it leads to a linear increase in computational cost with increases in n . For example, if one could only save 1,000 samples, but needed to run a chain of 10,000 samples to effectively explore the posterior, one could thin by ten steps, and those 1,000 samples would have lower autocorrelation than if 1,000 samples were generated without thinning. If memory constraints are not an issue, however, it is advised to save the entire chain (Link & Eaton, 2011).

Another diagnostic test that should always be performed is to plot the chains themselves (i.e., the posterior sample at each iteration of the MCMC). This can be used to determine whether the sampling process has converged to the posterior distribution; it is easily performed in R using the `traceplot` function (part of the RStan package) on line 16. The left panel of Fig. 2 shows the samples from Stan, and the right panel shows the samples from Metropolis–Hastings. The researcher can use a few criteria to diagnose convergence. First, as an initial visual diagnostic, one can determine whether the chains look “like a fuzzy caterpillar” (Lee & Wagenmakers, 2014)—do they have a strong central tendency with evenly distributed aberrations? This indicates that the samples are not highly correlated with one another and that the sampling algorithm is exploring the posterior distribution efficiently. Second, the chains should also not drift up or down, but should remain around a constant value. Lastly, it should be difficult to distinguish between individual chains. Both panels of Fig. 2 clearly demonstrate all of these criteria, suggesting convergence to the posterior distribution.

Once it is determined that the sampling process has converged to the posterior, we can then move on to analyzing the parameter estimates themselves and determining whether the

² The `print` function behaves differently given different classes of objects in R. For the Stan `fit` object, it prints a summary table. The RStan library must be loaded for this behavior to occur (the library defines the `fit` object in R).

Fig. 1 Autocorrelation function of the rate parameter



model can fit the observed (in this case, simulated) data. Lines 18 through 20 of the R code (Box 2) show how the posterior predictive can be obtained by plotting the data (`dat`) as a histogram and then overlaying the density of predicted values, `pred`. The left panel of Fig. 3 shows that the posterior predictive density (solid line) fits the data (histogram bars) quite well. Lastly, line 22 of the R code shows how to plot the posterior distribution of λ with the following command: `plot(density(mcmc_chain[, 'lambda']))`. There are two steps to this command. First, the `density` function, a base function in R, is called. This function estimates the density of the posterior distribution of λ from the MCMC samples held in `mcmc_chain[, 'lambda']`. Second, the `plot` function, also part of the base R distribution, is called, which outputs a plot of the density plot of the MCMC samples. It is also possible to call the histogram function, `hist`, and plot

the samples as a histogram. The right panel of Fig. 3 shows the posterior distribution of λ . The 95 % highest density interval (HDI) is also depicted. The HDI is the smallest interval that can be obtained in which 95 % of the mass of the distribution rests; this interval can be obtained from the summary statistics output by `print(fit)`. The HDI is different from a confidence interval because values closer to the center of the HDI are “more credible” than values farther from the center (e.g., Kruschke, 2011). As the HDI increases, uncertainty about the parameter value also increases. As the HDI decreases, the range of credible values also decreases, thereby decreasing the uncertainty. Figure 3 shows that 95 % of the mass of λ is between 0.92 and 1.09, indicating that parameter recovery was successful, since the simulated data were generated with $\lambda = 1$.

Fig. 2 Samples from each chain as a function of iteration generated by Stan (left) and Metropolis–Hastings (right)

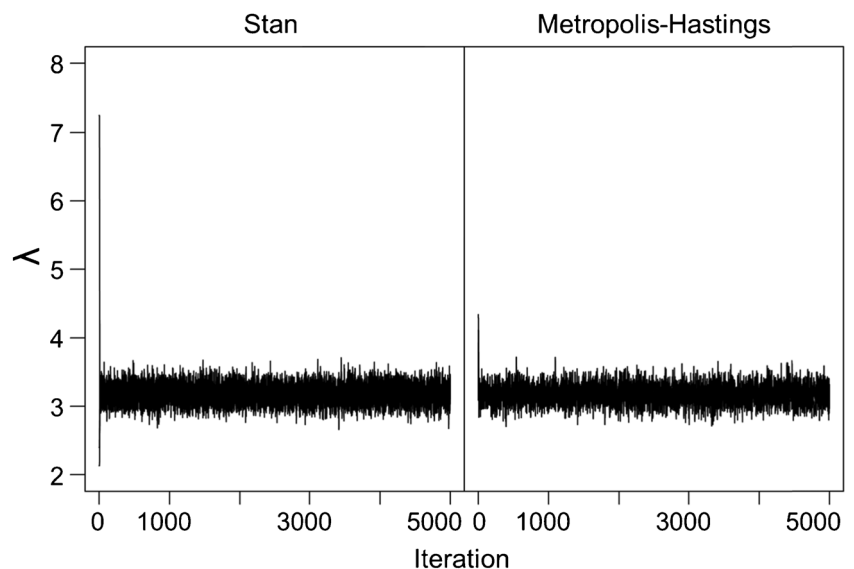
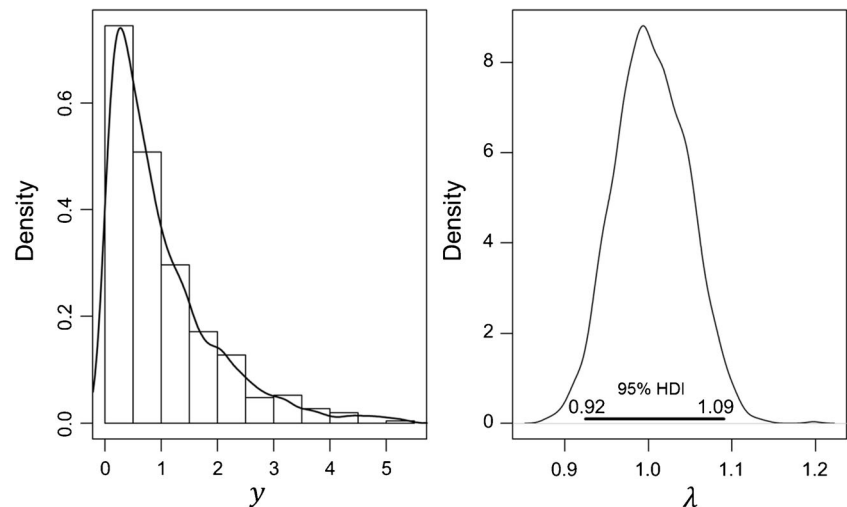


Fig. 3 A posterior predictive check (left), where the solid line represents the predictions and the histogram bars represent the data, and the posterior distribution of λ (right)



As we have just seen, the Stan model successfully recovered the single parameter value of λ that was used to generate the exponentially distributed data. Oftentimes, parameter recovery is more rigorous, testing recovery over a *range* of parameter values. The parameter recovery process can be repeated many times, each time storing the actual and recovered parameter values. A plot can then be made of the actual parameter values as a function of the recovered parameter values. The values should fall close to the diagonal (i.e., the recovered parameters should be close to the actual parameters). This also lets us explore how well Stan does over a range of parameterizations of the exponential.

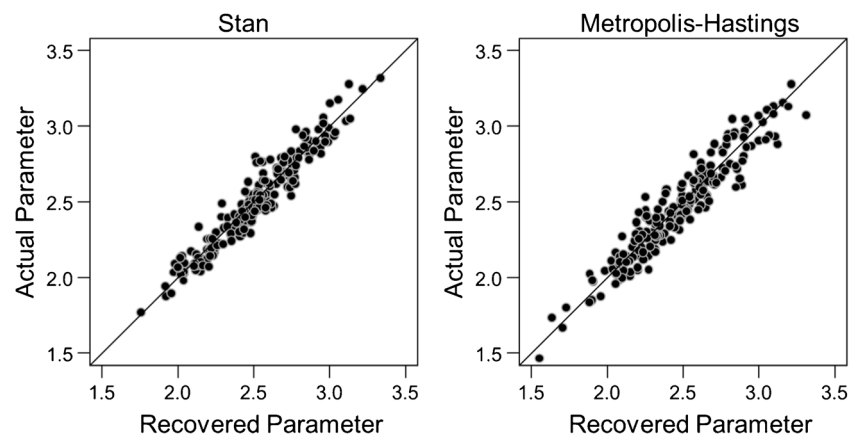
To better test the Stan model in this way, we simulated 200 sets of data over a range of values of λ . The λ parameter values were drawn from a truncated normal distribution with mean 2.5 and standard deviation 0.25. Each data set contained 500 data points. The Stan model was fit to each data set, saving the mean of the posterior of lambda for each fit. The left panel of Fig. 4 shows the parameter recovery for the exponential distribution implemented in Stan. We can see that

the parameter recovery was successful, since most values fall close to the diagonal. If we use the classic Metropolis–Hastings algorithm, we can see in the right panel of Fig. 4 that its performance is very similar to our parameter recovery in Stan.

User-defined distributions in Stan

Thus far we have implemented an exponential model in Stan using built-in probability distributions for the likelihood and the prior. Although there are dozens of built-in probability distributions in Stan (as in other Bayesian applications), sometimes the user requires a distribution that might not already be implemented. This will often be the case for specialized distributions of the kind assumed in many cognitive models. But before moving on to complicated cognitive models, we first want to present an example using the exponential model, but without the benefit of using Stan’s built-in probability distribution function.

Fig. 4 Actual parameter values plotted as a function of the recovered parameter values for Stan (left) and Metropolis–Hastings (right)



An example with the exponential distribution, `redux`

The exponential distribution is a built-in distribution in Stan, and therefore it is not necessary to implement it as a user-defined function. We do so here for tutorial purposes.

To begin, the likelihood function of the exponential distribution is

$$P(y|\lambda) = \prod_{i=1}^N \lambda e^{-\lambda y_i}, \quad (3)$$

where λ is the rate parameter of the exponential, y is the vector of data points, each data point $y_i \in [0, \infty)$, and N is the number of data points in y . Stan requires the

log likelihood, so we simply take the log of Eq. 3 in our Stan implementation.

Having mathematically defined the (log) likelihood function, we can now implement it in Stan. Once we implement the user-defined function, we can then call it just as we would call a built-in function. In this example, we will replace the built-in distribution, `exponential`, in the sampling statement $Y \sim \text{exponential}(\text{lambda})$ in line 14 of Box 1 with a user-defined exponential distribution.

Box 3 shows how this is accomplished. To add a user-defined function, it is first necessary to define a `functions` code block. The `functions` block must come before all other blocks of Stan code, but when there are no user-defined functions this block may be omitted entirely.

Box 3 Stan code of exponential model as a user-defined function

```

1  functions{
2      real newexp_log(vector x, real lam){
3          vector[num_elements(x)] prob;
4          real lprob;
5          for (i in 1:num_elements(x)){
6              prob[i] <- lam*exp(-lam*x[i]);
7          }
8          lprob <- sum(log(prob));
9          return lprob;
10     }
11 }
12 data{
13     int LENGTH;
14     vector[LENGTH] Y;
15 }
16 parameters{
17     real<lower=0> lambda;
18 }
19 model{
20     real alpha;
21     real beta;
22     alpha <- 1.0;
23     beta <- 1.0;
24     lambda ~ gamma(alpha,beta);
25     Y ~ newexp(lambda);
26 }
27 generated quantities{
28     real pred;
29     pred <- exponential_rng(lambda);
30 }

```

When dealing with functions that implement probability distributions, three important rules must be

considered. First, Stan requires the name of any function that implements a probability distribution to end

with `_log`;³ the `_log` suffix permits access to the `increment_log_prob` function (an internal function that can be ignored for the purposes of this tutorial), where the total log probability of the model is computed. Second, when *calling* such defined functions, the `_log` suffix must be dropped. Lastly, when naming a user-defined function, the name must be different from any built-in function when defining it in the `functions` block *and* it must differ from any built-in function when the `_log` suffix is dropped. For example, suppose that we named our user-defined function `exp_log`. When called, this would be different from the built-in `exponential()` function, but unfortunately it would conflict with another built-in function, `exp`, and result in an error. With these rules in mind, we can now properly name our user-defined exponential likelihood function. Line 2 of Box 3 shows that we have named the exponential likelihood function `newexp_log`. This name works because there are no built-in function called `newexp` and no built-in distribution `newexp_log`.

The `newexp_log` function returns a real number of type `real`. The first argument of the function is the data vector `x`, and the second argument is the rate parameter `lam`. Note that the scope of the variables within each function is local. Within the function itself, another local variable is defined called `prob`, which is a vector of the same length as the data vector `x`. For each element in the data vector, a probability density will be computed and stored in `prob`, implementing the elements in Eq. 3. As we noted previously, Stan requires the log likelihood, so instead of multiplying the probability densities, we take the natural logs and sum them. The sum of the log densities will be assigned to the variable `lprob`.⁴ Then `lprob` value—representing the log likelihood of the exponential distribution (log of Eq. 3)—is then returned by the function. After this, lines 12 through 30 are identical to the code shown in Box 1 lines 1 through 19, with the exception of the call to `newexp` (Box 3 line 25) instead of the built-in Stan `exponential` distribution (Box 1 line 14).

We found that this implementation produced exactly the same results as the implementation using the built-in

³ This naming convention holds for user-defined and as well as built-in functions. For example, in line 14 of Box 1, in the sampling statement $Y \sim \text{exponential}(\text{lambda})$ we are actually calling the built-in function `exponential_log` by dropping the `_log` suffix.

⁴ Stan includes C++ libraries designed for efficient vector and matrix operations, and therefore it is often more efficient to use the vectorized form of a function. For example, the log likelihood can be computed in a single line with `lprob <- sum(log(lam) - x*lam);`. For simplicity, we do not consider vectorization any further, and instead refer readers to the Stan manual.

distribution (Box 1). This is not surprising, given that the built-in and user-defined exponential distributions are mathematically equivalent.

An example with the LBA model

In this section, we briefly describe the LBA model and how it can be utilized in a Bayesian framework, before describing how it can be implemented in Stan. Accumulator models attempt to describe how the evidence for one or more decision alternatives is accumulated over time. LBA predicts response probabilities as well as distributions of the response times, much like other accumulator models. Unlike some models, which assume a noisy accumulation of evidence to threshold within a trial, LBA instead assumes a linear and continuous accumulation to threshold—hence, the “ballistic” in LBA. LBA assumes that the variabilities in response probabilities and response times are determined by between-trial variability in the accumulation rate and other parameters.

LBA assumes a separate accumulator for each response alternative i . A response is made when the evidence accrued for one of these alternatives exceeds some predetermined threshold, b . The rate of accumulation of evidence is referred to as the *drift rate*. The LBA model assumes that the drift rate, d_i , is sampled on each trial from a normal distribution with mean v_i and standard deviation s . Figure 5 illustrates an example in which the drift for response m_1 is greater than that

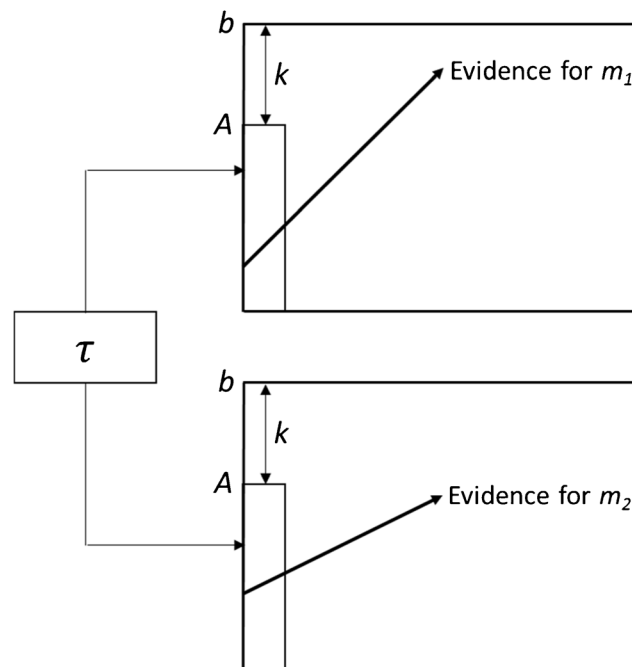


Fig. 5 Graphical depiction of the linear ballistic accumulator (LBA) model (Brown & Heathcote, 2008)

for response m_2 . In this example trial, the participant will make response m_1 because that accumulator reaches its threshold, b , before the other accumulator.

Each accumulator starts with some a priori amount of evidence. This start point is assumed to vary across trials. The start-point variability is assumed to be uniformly distributed between 0 and A (A must be less than the threshold b).

Like other accumulator models, LBA also makes the assumption that there is a period of nondecision time, τ , that occurs before evidence begins to accumulate (as well as after, leading to whatever motor response is required). In this implementation, as in some other LBA implementations, we assume that the nondecision time is fixed across trials.

The following equations are a formalization of these processing assumptions, showing the likelihood function for the LBA (see Brown & Heathcote, 2008, for derivations). Given the processing assumptions of the LBA, the response time, t , on trial j is given by

$$t_j = \tau + \min_i \left(\frac{b-a_i}{d_i} \right). \quad (4)$$

Let us assume that θ is the full set of LBA parameters $\theta = \{v_1, v_2, b, A, s, \tau\}$. Then the joint probability density function of making response m_1 at time t (referred to as the *defective PDF*) is

$$LBA(m_1, t|\theta) = f(t-\tau|v_1, b, A, s)[1-F(t-\tau|v_2, b, A, s)], \quad (5)$$

and the joint density for making response m_2 at time t is

$$LBA(m_2, t|\theta) = f(t-\tau|v_2, b, A, s)[1-F(t-\tau|v_1, b, A, s)], \quad (6)$$

where f and F are the probability density function (PDF) and the cumulative distribution function (CDF) of the LBA distribution, respectively. We refer the reader to Brown and Heathcote for the full mathematical descriptions and justifications of the LBA's CDF $F(t)$ and PDF $f(t)$. For the Stan implementation details of these distributions, please see the [Appendix](#).

Negative drift rates may arise in this model, because drifts across trials are drawn from a normal distribution. If both drift rates are negative, this will lead to an undefined response time.

The probability of this happening is $\prod_{i=1}^2 \Phi\left(\frac{-v_i}{s}\right)$. If we assume that at least one drift rate will be positive, then we can truncate the defective PDF:

$$LBA_{\text{trunc}}(m_i, t|\theta) = \frac{LBA(m_i, t|\theta)}{1 - \prod_{i=1}^2 \Phi\left(\frac{-v_i}{s}\right)}. \quad (7)$$

Thus, this model assumes there are zero undefined response times. This will be the model we implement in

Stan. If we consider the vector of binary responses, R , and response times, T , for each trial i and a total of N trials, the likelihood function is given by

$$P(T, R|\theta) = \prod_{i=1}^N LBA_{\text{trunc}}(T_i, R_i|\theta). \quad (8)$$

To implement the model in a Bayesian framework, priors are placed on each of the parameters of the LBA model. We chose priors that one might encounter in real-world applications and based them on Turner et al. (2013). First, to make the model identifiable, we set s to a constant value (Donkin, Brown, & Heathcote, 2009). Here, we fix s at 1. We then assume that the priors for the drift rates are truncated normal distributions:

$$v_i \sim \text{Normal}(2, 1) \epsilon(0, \infty). \quad (9)$$

We assume a uniform prior on nondecision time:

$$\tau \sim \text{Uniform}(0, 1). \quad (10)$$

The prior for the maximum starting evidence A is a truncated normal distribution:

$$A \sim \text{Normal}(.5, 1) \epsilon(0, \infty). \quad (11)$$

To ensure that the threshold, b , is always greater than the starting point a , we reparameterize the model by shifting b by k units away from A . We refer to k as the *relative threshold*. Thus, we do not model b directly, but as the sum of k and A , and assume that the prior for k is a truncated normal:

$$k \sim \text{Normal}(.5, 1) \epsilon(0, \infty). \quad (12)$$

Stan code The Stan code for the LBA likelihood function is shown in Box 4. Note that lines 2 through 102 are omitted from the code for the sake of brevity. These omitted lines contain the code implementing the PDF and CDF functions of the LBA, and can be found in the [Appendix](#). The likelihood function of the LBA is named `lba_log` (recall that the `_log` suffix is only used in its definition and is dropped when the function is called in the `model` block) and accepts the following arguments: a matrix, `RT`, whose first column contains the observed response times, and whose second column contains the observed responses; the relative threshold, `k`; the maximum starting evidence, `A`; a vector holding the drift rates, `v`; the standard deviation of the drift rates, `s`; and the nondecision time, `tau`. Note that the Stan implementation of the LBA differs from other Bayesian implementations of accumulator models, which treat negatively coded response times as errors and positively coded response times as correct responses (e.g., Wabersich & Vandekerckhove, 2014). This is a major advantage of the LBA and the implementation that we present here, in that it allows for any number of response choices.

Box 4 Likelihood function of the LBA implemented in Stan

```

1  functions{
*snip*
103 real lba_log(matrix RT, real k, real A, vector v, real s, real tau){
104
105     real t;
106     real b;
107     real cdf;
108     real pdf;
109     vector[rows(RT)] prob;
110     real out;
111     real prob_neg;
112
113     b <- A + k;
114     for (i in 1:rows(RT)){
115         t <- RT[i,1] - tau;
116         if (t > 0){
117             cdf <- 1;
118             for (j in 1:num_elements(v)){
119                 if (RT[i,2] == j){
120                     pdf <- lba_pdf(t, b, A, v[j], s);
121                 }else{
122                     cdf <- (1-lba_cdf(t, b, A, v[j], s)) * cdf;
123                 }
124             }
125             prob_neg <- 1;
126             for (j in 1:num_elements(v)){
127                 prob_neg <- Phi(-v[j]/s) * prob_neg;
128             }
129             prob[i] <- pdf*cdf;
130             prob[i] <- prob[i]/(1-prob_neg);
131             if (prob[i] < 1e-10){
132                 prob[i] <- 1e-10;
133             }
134
135             }else{
136                 prob[i] <- 1e-10;
137             }
138         }
139     out <- sum(log(prob));
140     return out;
141 }

```

First, the local variables to be used in the function are defined (lines 105–111 in Box 4). Then, to obtain the decision threshold b , k is added to A . On each iteration of the `for` loop, the decision time t is obtained by subtracting the nondecision time τ from the response time RT . If the decision time is greater than zero, then the defective PDF is computed as in Eqs. 5 and 6, and the CDF and PDF functions described earlier accordingly are called on lines 120 and 122 (see the Appendix for the Stan implementation of each). The defective PDF associated with each row in RT is stored in the `prob` array. If the value of the defective PDF is less than 1×10^{-10} , then the value stored in `prob` is set to 1×10^{-10} ; this is to avoid underflow problems

arising from taking the natural logarithm of extremely small values of the defective PDF. Once all of the densities are computed, the likelihood is obtained by taking the sum of the natural logarithms of the densities in `prob` and returning the result.

Box 5 continues the code from Box 4 and, as in our earlier example, shows the `data` block defining the data variables that are to be modeled. The `LENGTH` variable defines the number of rows in RT , whose first column contains response times and whose second column contains responses. A response coded as 1 corresponds to the first accumulator finishing first, and a response coded as 2 corresponds to the second

accumulator finishing first. One of the advantages of the LBA is that it can be applied to tasks with more than two choices. The `NUM_CHOICES` variable defines the

number of choices in the task and must be equal to the length of the drift rate vector defined in the `parameters` block.

Box 5 Continuation of Stan code for the LBA model

```

142 data{
143   int LENGTH;
144   matrix[LENGTH,2] RT;
145   int NUM_CHOICES;
146 }
147 parameters{
148   real<lower=0> k;
149   real<lower=0> A;
150   real<lower=0> tau;
151   vector<lower=0>[NUM_CHOICES] v;
152 }
153 model{
154   k ~ normal(.5,1)T[0,];
155   A ~ normal(.5,1)T[0,];
156   tau ~ normal(.5,.5)T[0,];
157   for (n in 1:NUM_CHOICES){
158     v[n] ~ normal(2,1)T[0,];
159   }
160   RT ~ lba(k,A,v,1,tau);
161 }
162 generated quantities{
163   vector[2] pred;
164   pred <- lba_rng(k,A,v,1,tau);
165 }

```

The `parameters` block shows that the parameters are all real numbers of type `real` and include the relative threshold `k`, the maximum starting evidence `A`, the nondecision time `tau`, and the vector of drift rates `v`. All parameters have normal priors truncated at zero, and therefore are constrained with `<lower=0>`.

The Bayesian LBA model is implemented in the `model` block, which shows that the priors for the relative threshold `k` and the maximum threshold `A` are both assumed to be normally distributed with a mean of `.5` and standard deviation `1`. The prior for nondecision time is assumed to be normally distributed with a mean of `.5` and standard deviation `.5`, and the priors for drift rates are distributed normally with means of `2` and standard deviations of `1`. The data, `RT`, is assumed to be distributed according to the LBA distribution, `lba`.

The implementation of the `generated quantities` block for the LBA uses a user-defined function called `lba_rng`, which generates random simulated samples from the LBA model given the posterior parameter estimates. The function is also defined in the

`functions` block, along with all of the other user-defined functions, but has been omitted in Box 4 for brevity. The code and explanation for this function can be found in the [Appendix](#). This code is based on the “`rtstats`” package for R (Singman et al., 2016), which can be found online (<https://cran.r-project.org/web/packages/rtstats/index.html>). We note that porting code from R to Stan is relatively straightforward, as they both are geared toward vector and matrix operations and transformations.

R code Box 6 shows the R code that runs the LBA model in Stan. This should look very similar to the code we used for the simple exponential example earlier. We again begin by clearing the workspace, setting the working directory, and loading the `RStan` library. After simulating the data from the LBA distribution using a file called “`lba.r`” (see the website listed above for the code), which contains the `rlba` function that generates random samples drawn from the LBA distribution, the model is then run on line 10.

Box 6 R code that runs the LBA model in Stan

```

1  rm(list=ls())
2  setwd("~/LBA/")
3  source('lba.r')
4  library(rstan)
5  #make simulated data
6  out = rlba(500,1,1,c(2,1),1,.5)
7  rt = cbind(out$rt,out$resp)
8  len = length(rt[,1])
9  #run the Stan model
10 fit <- stan(file = 'lba.stan',
              data = list(RT=rt,LENGTH=len,NUM_CHOICES=2),
              warmup = 750,
              iter = 1500,
              chains = 3)

```

As we noted in our earlier example, in real-world applications of the model, the data would not be simulated but would be collected from a behavioral experiment. We use simulated data here for convenience of the tutorial and because we are interested in determining whether the Bayesian model can recover the known parameters used to generate the simulated data (parameter recovery). With just some minor modification, the code we provide using simulated data can be generalized to an application to real data. For example, real data stored in a text file or spreadsheet can be read into R and then formatted and coded in the same way as the simulated data.

The Bayesian LBA model can be validated in a fashion similar to that for the Bayesian exponential model. Figure 6 shows the autocorrelation function for each parameter. For Stan, autocorrelation across all parameters became undetectable after approximately 15 iterations. The right panels shows that the Metropolis–Hastings algorithm had high autocorrelation for long lags, indicating that the sampler was not taking independent samples from the posterior distribution. This high autocorrelation leads to lower numbers of effective samples and longer convergence times. The N_{eff} values returned by Metropolis–Hastings across all chains were on average 27 for each parameter. This means that running 4,500 iterations (three chains of 1,500 samples) is equivalent to drawing only 27 independent samples. On the other hand, Stan returned on average 575 effective samples for each parameter after 4,500 iterations. In addition, \hat{R} for all parameters was above 1.1 for Metropolis–Hastings, and below 1.1 for Stan, indicating that the chains converged for Stan but not for Metropolis–Hastings.

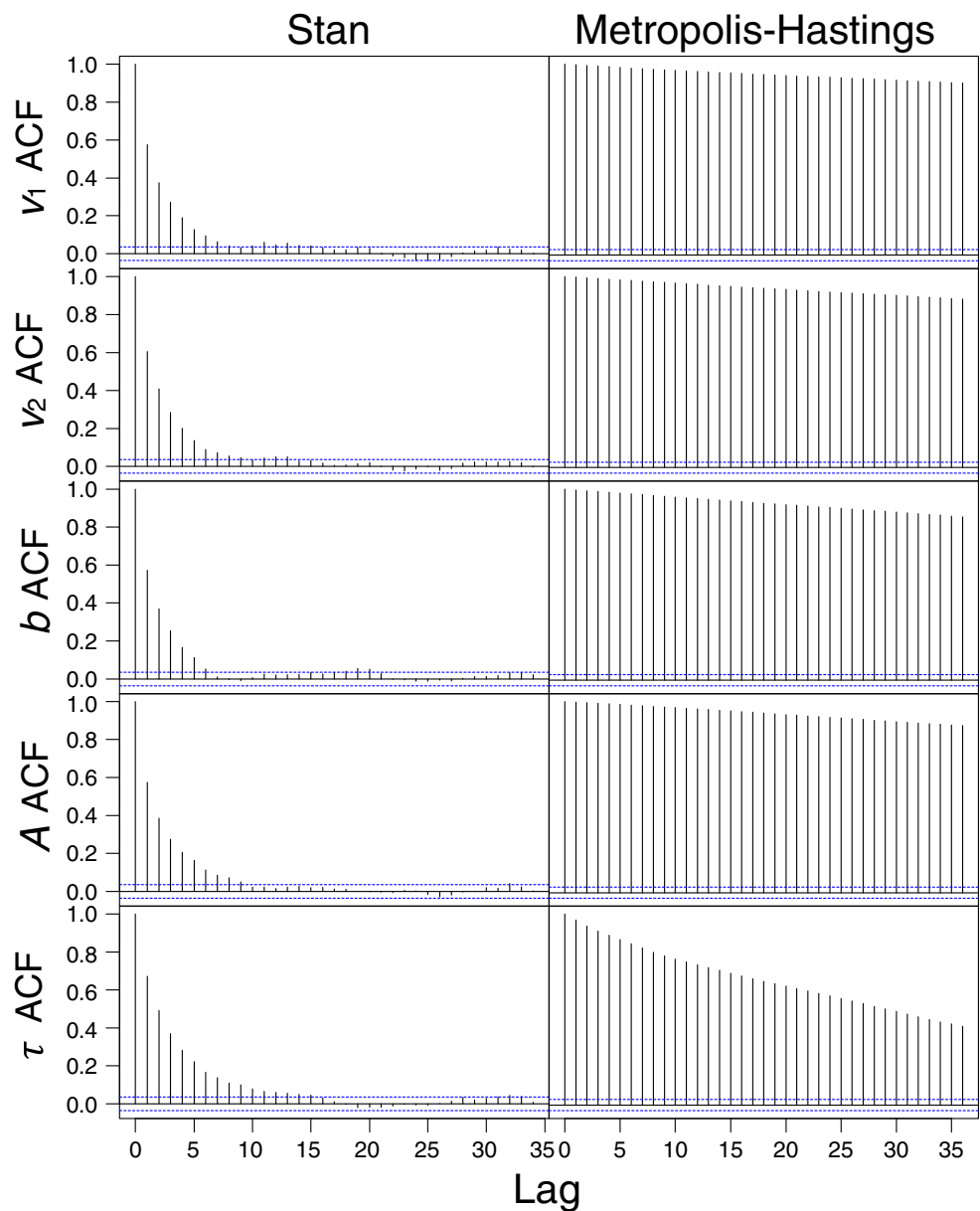
The deleterious effect of the high autocorrelation of Metropolis–Hastings in comparison to the low autocorrelation of Stan is apparent in Fig. 7. The left panels show the chains produced by Stan, and the right panels show the chains produced

by Metropolis–Hastings. In the left panel, the Stan chains show good convergence: They look like “fuzzy caterpillars,” it is difficult to distinguish one chain from the others, and the chains do not drift up and down. In the right panels of Fig. 7, the Metropolis–Hastings chains clearly do not meet any of the necessary criteria for convergence. The only way we found to correct for this was to thin by at least 50 or more steps.

Figure 8 shows the results of a larger parameter recovery study for Stan and the Metropolis–Hastings algorithm. In this exploration, 200 simulated data sets containing 500 data points each were generated, each with a different set of parameter values. The parameter values were drawn randomly from a truncated normal with a lower bound of 0, a mean of 1, and a standard deviation of 1. The Stan model was fit to each data set, and the resulting mean of the posterior distribution for each parameter was saved. Figure 8 shows the actual parameter values plotted against the recovered parameter values. For Stan, most of the points fall along the diagonal, indicating that parameter recovery was successful. For Metropolis–Hastings, it is clear visually that parameter recovery is poorer—this is due to the aforementioned difficulty this algorithm has with the inherent correlations between parameters in the LBA model.

We note that parameter recovery in sequential-sampling models is often difficult if the experimental design is unconstrained, like the one we present here, which benefited from a large number of data points for each data set as well as priors that were similar to the actual parameters that had generated the data. We present this parameter recovery as a sanity check to ensure that Stan can recover sensible parameter values under optimal conditions. Obviously, this will not be the case in real-world applications, and therefore, great care must be taken when designing experiments to test sequential-sampling models like the LBA.

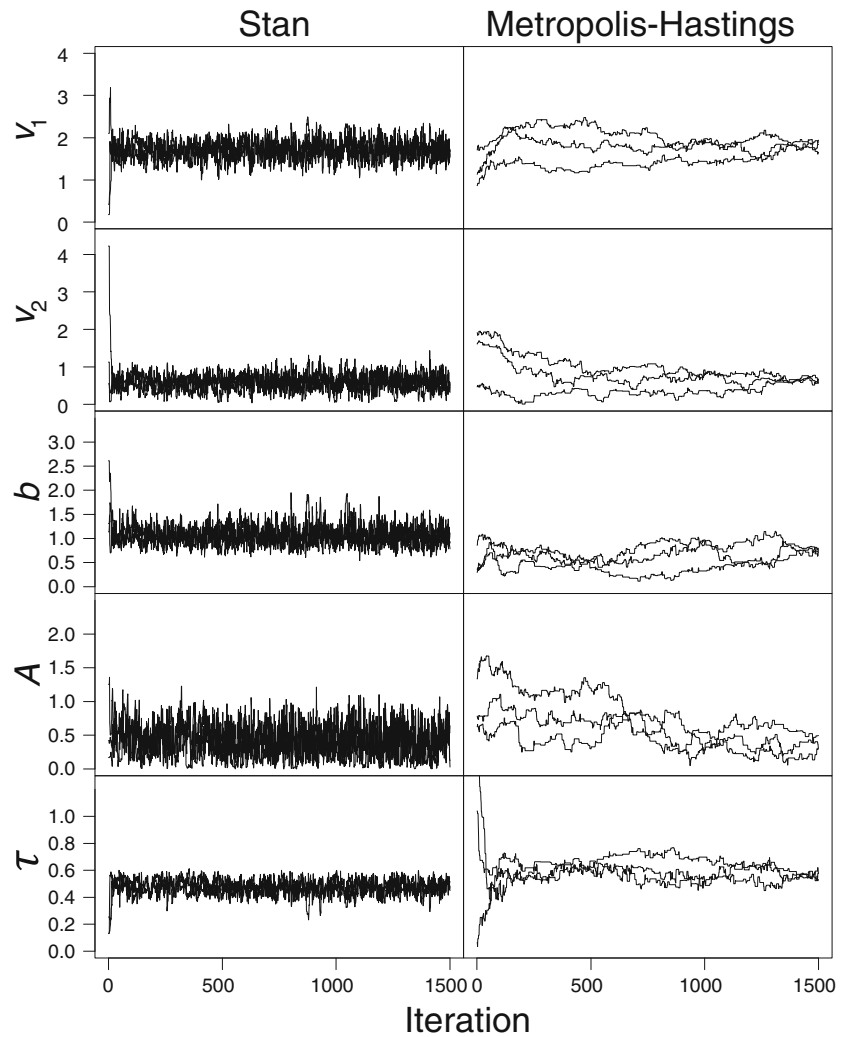
Fig. 6 Autocorrelation function (ACF) of each parameter, plotted as a function of lag, for Stan (left) and Metropolis–Hastings (right)



Better Metropolis–Hastings sampling might be achieved by careful adjustment and experimentation with the proposal step process. Here, the proposal step was generated by sampling from a normal distribution with a mean equal to the current sample and a standard deviation of .05. Increasing the standard deviation increases the average distance between the current sample and the proposal, but decreases the probability of accepting the proposal. We found that different settings of the standard deviation largely led to autocorrelations similar to those we have presented here. The only thing we found that led to improvements in autocorrelation was thinning. Thinning by 50 steps led to autocorrelation dropping to nonsignificant values at around lags of 40. At 75 steps, Metropolis–Hastings’s performance was similar to that of Stan, resulting in similar autocorrelation, N_{eff} , and \hat{R} values.

A reason behind the poor sampling of Metropolis–Hastings is the correlated parameters of the LBA. The half below the diagonal of Fig. 9 shows the joint posterior distribution for each parameter pair of the LBA for a given set of simulated data, and the half above the diagonal gives the corresponding correlations. Each point in each panel in the lower half of the grid represents a posterior sample from the joint posterior probability distribution of a particular parameter pair for the LBA model. For example, the bottom left corner panel shows the joint posterior probability distribution between τ and v_1 . We can see that this distribution has negatively correlated parameters. The upper right corner panel of the grid confirms this, showing the correlation between τ and v_1 to be $-.45$. Five joint distributions have correlations with absolute values well above .50 (v_1-v_2 , v_1-b , v_2-b , $v_2-\tau$, and $A-\tau$). These correlations in parameter values cause some

Fig. 7 Samples of each parameter for each iteration of each chain for Stan (left) and Metropolis–Hastings (right)



MCMC algorithms such Gibbs sampling and the Metropolis–Hastings to perform poorly. On the other hand, Stan does not drastically suffer from the model’s correlated parameters.

In summary, the Stan implementation of the LBA model shows successful parameter recovery and efficient sampling of

the posterior distribution when compared to Metropolis–Hastings, due in large part to the correlated parameters of the LBA model. Whereas Stan was designed with the intention to handle these situations properly, standard MCMC techniques such as the Metropolis–Hastings algorithm were not, and they

Fig. 8 Actual parameter values, plotted as a function of the parameters recovered by Stan (left) and Metropolis–Hastings (right)

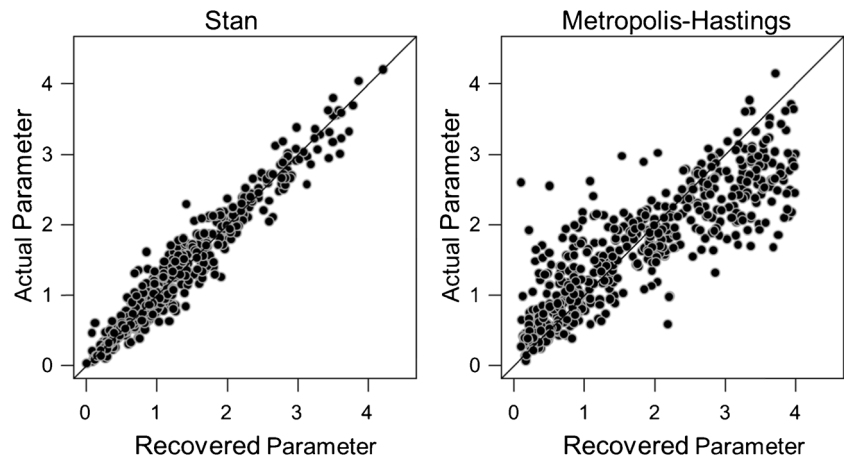
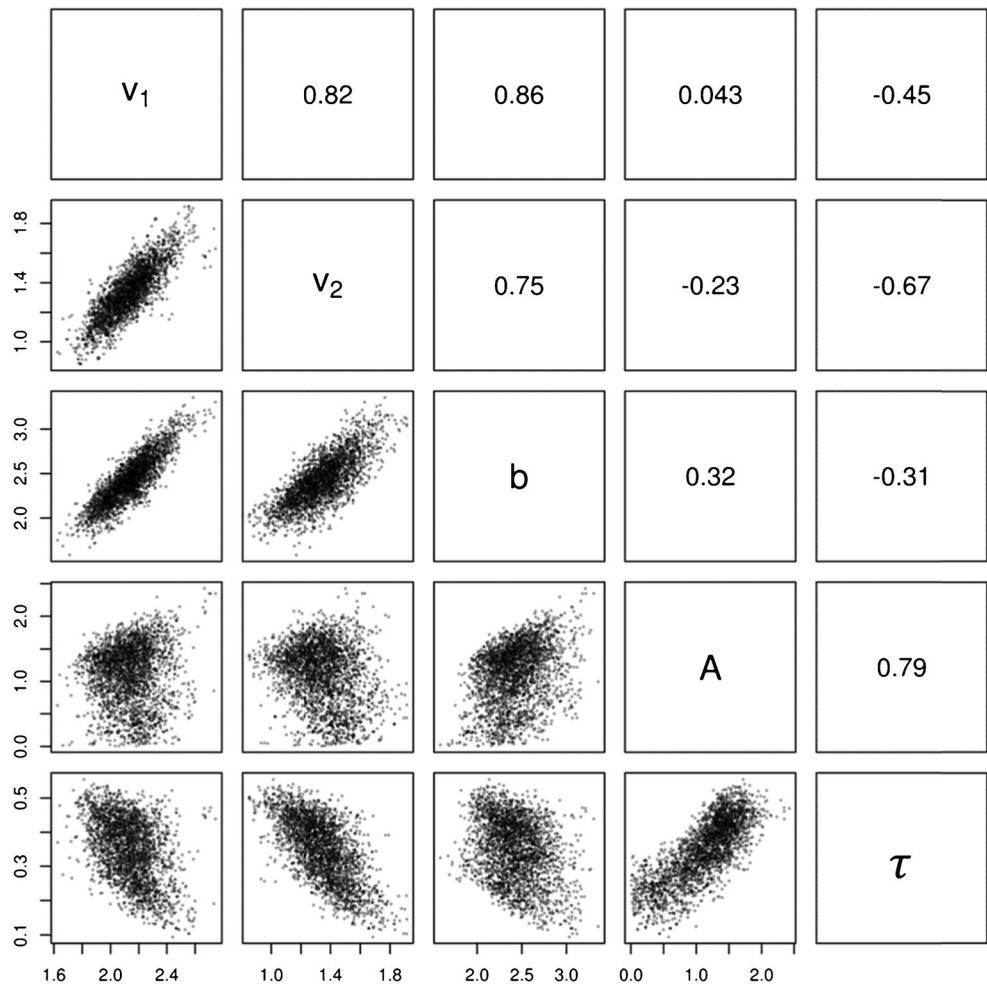


Fig. 9 The lower left of the grid shows the joint posterior probability distributions for each pair of key parameters in the LBA model fitted to a simulated set of data. Each point in each panel represents a posterior sample from the joint posterior probability distribution of a particular parameter pair for the LBA model. For example, the bottom left corner panel shows the joint posterior probability distribution between τ and v_1 . The upper right of the grid gives the correlation between each parameter pair. For example, the upper right corner shows the correlation between τ and v_1 to be -0.45



do not converge to the posterior distribution in any sort of reliable manner.

Fitting multiple subjects in multiple conditions: a hierarchical extension of the LBA model

The simple LBA model just described was designed for a single subject in a single condition. This is never the case in any real-world application of the LBA model. In this section, we describe and implement an LBA model that is designed for multiple subjects in multiple conditions. The model will include parameters that model performance at both the group and individual levels. The Bayesian approach allows both the group- and individual-level parameters to be estimated simultaneously. This type of model is called a *Bayesian hierarchical model* (e.g., Kruschke, 2011; Lee & Wagenmakers, 2014). In a Bayesian hierarchical model, the parameters for individual participants are informed by the group parameter estimates. This reduces the potential for the individual parameter estimates to be sensitive to outliers, and decreases the overall number of participants necessary to achieve reliable parameter estimates.

The model we consider assumes that the vector of response times for each participant i in condition j is distributed according to the LBA:

$$RT_{i,j} \sim LBA(k_i, A_i, v_{i,j}^1, v_{i,j}^2, s, \tau_i), \tag{13}$$

where, as before, the responses are coded as 1 and 2, corresponding to each accumulator; k_i is the relative threshold; A_i is the maximum starting evidence; $v_{i,j}^1$ and $v_{i,j}^2$ are the mean drift rates for each accumulator; s is the standard deviation, held constant across participants and conditions; and τ_i is the nondecision time. As before, we assume that s is fixed at 1.0 and that the prior on each parameter follows a truncated normal distribution with its own group mean μ and standard deviation σ .

$$k_i \sim \text{Normal}(u^k, \sigma^k) \in (0, \infty) \tag{14}$$

$$A_i \sim \text{Normal}(u^A, \sigma^A) \in (0, \infty) \tag{15}$$

$$v_{i,j}^1 \sim \text{Normal}(\mu_j^{v_1}, \sigma_j^{v_1}) \in (0, \infty) \tag{16}$$

$$v_{i,j}^2 \sim \text{Normal}(\mu_j^{v_2}, \sigma_j^{v_2}) \in (0, \infty) \tag{17}$$

$$\tau_i \sim \text{Normal}(u^\tau, \sigma^\tau) \in (0, \infty) \quad (18)$$

Having defined the priors at the individual level, the next step in designing a hierarchical model is to define the group-level priors on the parameters in Eqs. 14–18. The group-level priors that we choose are grounded on what one might encounter in real-world situations and are based on Turner et al. (2013). The priors for the group-level means for k , A , and τ are assumed be constant across conditions:

$$u^k, u^A \sim \text{Normal}(.5, 1) \in (0, \infty), \quad (19)$$

$$u^\tau \sim \text{Normal}(.5, .5) \in (0, \infty). \quad (20)$$

The priors for the group-level drift rates for each condition are given by the following:

$$v_j^1, v_j^2 \sim \text{Normal}(2, 1) \in (0, \infty). \quad (21)$$

Following Turner et al., the priors for the group-level standard deviations are each assumed to be weakly informative:

$$\sigma^k, \sigma^A, \sigma_j^{v_1}, \sigma_j^{v_2}, \sigma^\tau \sim \text{Gamma}(1, 1) \in (0, \infty). \quad (22)$$

This model is conceptually easy to implement in Stan, as is shown in Box 7, even if it requires a good bit more coding than the simple version. Lines 3 through 16 define the group-level priors shown in Eqs. 19 through 22. Lines 13 through 16 define a `for` loop in which the group-level mean drift rates are estimated for each condition j and each accumulator n . The individual-level parameters are defined in lines 18 through 28; for example, line 19 indicates that for each subject i , the prior on $k[i]$ is distributed according to a truncated normal distribution on the interval $(0, \infty)$ with mean `k_mu` and standard deviation `k_sigma`. Likewise, lines 23 through 25 define the mean drift rates on each accumulator n for each subject i in condition j ; for example, line 24 indicates that the prior on the individual-level mean drift rates, $v[i, j, n]$, is distributed according to a normal distribution with a group-level mean `v_mu[j, n]` and standard deviation `v_sigma[j, n]`. Thus, the individual-level drift rates are drawn from a normal distribution with the group-level means. This is what makes the model hierarchical and allows for the simultaneous fitting of group-level and individual-level parameters.

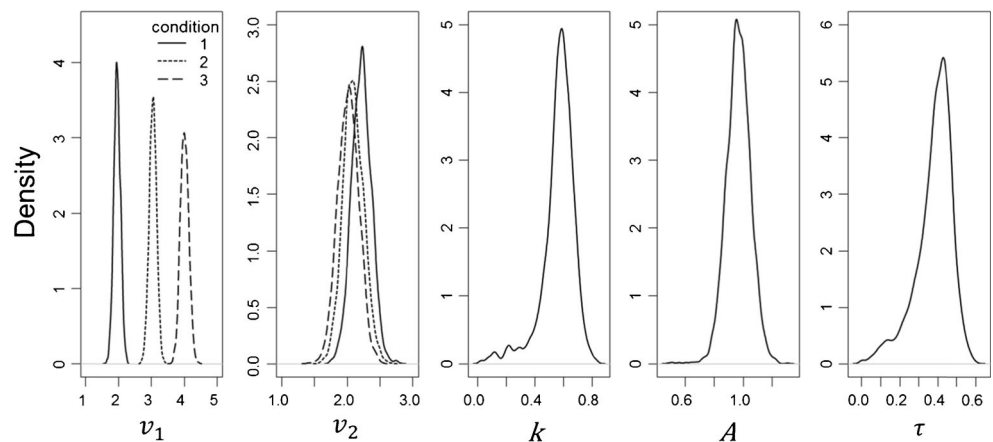
Box 7 Hierarchical LBA model implemented in Stan

```

1  model {
2
3    k_mu ~ normal(.5,1)T[0,];
4    A_mu ~ normal(.5,1)T[0,];
5    tau_mu ~ normal(.5,.5)T[0,];
6
7    k_sigma ~ gamma(1,1);
8    A_sigma ~ gamma(1,1);
9    tau_sigma ~ gamma(1,1);
10
11   for (j in 1:NUM_COND){
12     for (n in 1:NUM_CHOICES){
13       v_mu[j,n] ~ normal(2,1)T[0,];
14       v_sigma[j,n] ~ gamma(1,1);
15     }
16   }
17
18   for (i in 1:NUM_SUBJ){
19     k[i] ~ normal(k_mu,k_sigma)T[0,];
20     A[i] ~ normal(A_mu,A_sigma)T[0,];
21     tau[i] ~ normal(tau_mu,tau_sigma)T[0,];
22     for(j in 1:NUM_COND){
23       for(n in 1:NUM_CHOICES){
24         v[i,j,n] ~ normal(v_mu[j,n],v_sigma[j,n])T[0,];
25       }
26       RT[i,j] ~ lba(k[i],A[i],v[i,j],1,tau[i]);
27     }
28   }
29 }

```

Fig. 10 Group-level parameter estimates of the hierarchical LBA model for simulated data. For the panels plotting v_1 and v_2 , solid lines indicate Condition 1, dotted lines indicate Condition 2, and dashed lines indicate Condition 3



To test whether the model could successfully recover the parameters, we simulated 20 subjects, each with 100 responses and response times. Each simulated subject's parameters were drawn from a group-level distribution. Specifically, the maximum starting evidence parameter, A , relative threshold parameter, k , and nondecision time parameter, τ , were drawn from a truncated normal distribution with a mean of .5, standard deviation of .5, and lower bound of 0. We then varied the drift rates across three conditions. The drift rates of the first accumulator were drawn from a truncated normal with means of 2 (Condition 1), 3 (Condition 2), and 4 (Condition 3), respectively, all with standard deviations of 1 and lower bounds of 0. The mean drift rate of the second accumulator for all three conditions was drawn from a truncated normal with a mean of 2 and standard deviation of 1. In applications to real data, the distribution of the drift rates corresponding to the incorrect choice will usually have a lower mean and larger standard deviation than the distribution of the drift rates corresponding to the correct choice.

We then fit the hierarchical LBA model to the simulated data. The group-level parameter estimates are shown in Fig. 10. For the panels plotting v_1 and v_2 , solid lines indicate Condition 1, dotted lines indicate Condition 2, and dashed lines indicate Condition 3. All other parameters were held constant across conditions. The group-level parameter estimates for the hierarchical model shown in Fig. 10 closely align with the group-level distribution parameters used to generate the simulated data.

To further illustrate the advantages of the hierarchical LBA model, we also fit the nonhierarchical LBA model shown in Box 5 to the same set of simulated data. The nonhierarchical model assumed that for each subject, the priors on k and A were normally distributed with a mean of .5 and standard deviation of 1. The prior on τ for each subject was normal with mean .5 and standard deviation .5. Lastly, the prior on the drift rate for each accumulator was drawn from a normal distribution with a mean of 2 and standard deviation 1. Thus, the priors on the parameters for each subject in the nonhierarchical model mirrored the group-level priors in the hierarchical model.

Fig. 11 Hierarchical model parameter estimates (solid lines) versus nonhierarchical parameter estimates (dashed lines) for a single subject in a single condition

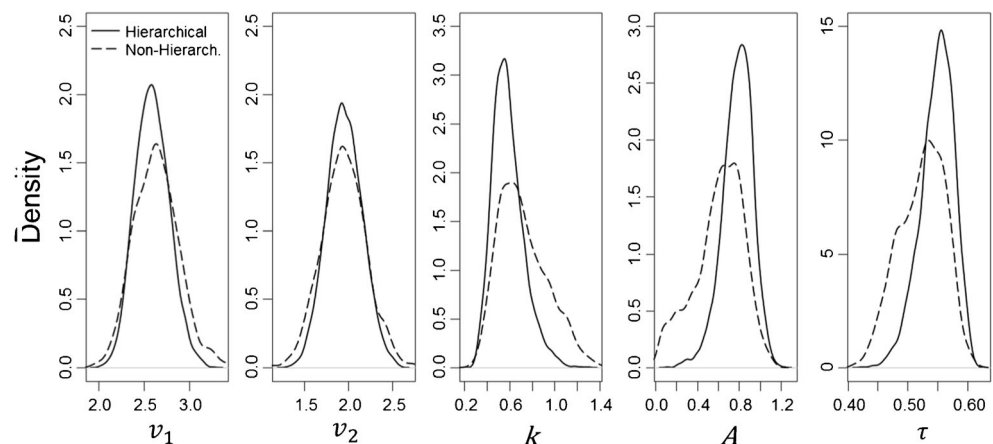


Figure 11 shows the hierarchical model (solid line) and the nonhierarchical model (dotted line) individual-level parameter estimates for the first subject in the first condition. It is clear that there is far less uncertainty in the parameter estimates of the hierarchical model. This decrease in uncertainty is due to a property of hierarchical models called *shrinkage*, through which the group estimates inform the individual-level parameter estimates (Kruschke, 2011). Therefore, increases in sample size will usually result in better parameter estimates for *both* the group and individual levels. This is in contrast to nonhierarchical models, which treat subjects independently, so that increases in sample size will only result in better group-level estimates.

Speeding up runtimes

Depending on the design of the experiment and the number of observed data points per condition, hierarchical LBA implementations within Stan can have runtimes that are quite long. For 20 subjects, each with 100 data points per condition, runtimes for a single chain were approximately 3–6 h on an Intel Xeon 2.90-GHz processor with more than sufficient RAM. If one is equipped with a multicore machine, speed-ups can be achieved by running multiple chains in parallel. This is a built-in option in Stan and can be achieved in a single line after the RStan library is loaded, with `options(mc.cores = parallel::detectCores())`. If Stan is prohibitively slow, we also recommend using a method in Stan that approximates the posterior, called *automatic differentiation variational inference* (ADVI; Kucukelbir, Tran, Ranganath, Gelman, & Blei, 2016). All models coded in Stan can be run using ADVI. Models that might take days to run in Stan using conventional methods might take less than an hour to run using ADVI. This can result in massive speed-ups during the initial stages of model development, when many iterative versions of a model need to be tested. If possible, it is recommended that the NUTS algorithm be used to draw final inferences using starting points based on samples drawn using ADVI. Although ADVI is beyond the scope of this tutorial, we have included example R code to run the hierarchical LBA with ADVI.

Discussion

Because parameters in cognitive models have psychological interpretations, accurately estimating those parameters is crucial for theoretical development. Traditional parameter estimation methods find the set of parameter values that maximize the likelihood of the data or maximize or minimize some other fit statistic. These methods often result in point estimates that do not take into account the uncertainty of the estimate. The Bayesian approach to parameter estimation, on the other hand, treats parameters as probability distributions, naturally encompassing the estimate's uncertainty. Because the estimated uncertainty of parameters factors into the complexity of the underlying model, Bayesian analysis is also particularly well-suited to aiding model selection between models of differing complexities. This advantage comes at a computational cost: Bayesian parameter estimation beyond very simple models with only one or two parameters requires the use of MCMC algorithms to create samples from the posterior distribution. Some of these algorithms, such as Metropolis–Hastings, require careful tuning to ensure that the sampling process converges to the posterior distribution. Metropolis–Hastings and other algorithms, such as Gibbs sampling, often show poor performance when the parameters of the model are correlated (as in cognitive models like the LBA). Moreover, implementing custom models, such as the LBA, can be technically challenging in many Bayesian analysis packages.

Stan (Stan Development Team, 2015) was developed to solve these issues by utilizing HMC (Duane, Kennedy, Pendleton, & Roweth, 1987; Neal, 2011), which can efficiently sample from distributions with correlated dimensions, making it particularly easy to implement custom distributions. We wrote this tutorial to show how to use Stan and how to develop custom distributions in it. As compared to some other packages, it is relatively easy to implement a custom model distribution, so long as the likelihood function is known (see Turner & Sederberg, 2012). It is relatively easy to extend model implementations to more complex scenarios that involve multiple subjects and multiple conditions by using hierarchical models.

The computation involved in NUTS is fairly expensive and can be slow for complex models. It should be noted that this lowered speed is, by design, traded off for greater effective sample rates. Other techniques, such as MCMC-DE (Turner et al., 2013), which approximate some of the more expensive computations involved in NUTS, may offer an alternative if the sampling rate becomes an issue.

Although Bayesian parameter estimation has many advantages over traditional methods, implementing the MCMC algorithm can be technically challenging. Turnkey Bayesian inference applications allow the researcher to work at the level of the model and not of the sampler, but they are likewise not without issues. Stan is a viable alternative to other applications that do automatic Bayesian inference, especially when the researcher is interested in distributions that are uncommon and require user implementation or when the model's parameters are correlated.

Box A1 Probability density function of the LBA, snipped from Box 4 in the main text.

Author note This work was supported by Grant Nos. NEI R01-EY021833 and NSF SBE-1257098, the Temporal Dynamics of Learning Center (NSF SMA-1041755), and the Vanderbilt Vision Research Center (NEI P30-EY008126).

Appendix

The Stan implementations of the PDF and CDF of the LBA are given in Boxes A1 and A2, respectively. These functions are used in the calculation of the likelihood function of the LBA given in Box 4 of the main text and are nothing more than implementations of the equations that Brown and Heathcote (2008) provided. Here, we simply note some implementation details of each.

```

22  real lba_cdf(real t, real b, real A, real v, real s){
23
24      real b_A_tv;
25      real b_tv;
26      real ts;
27      real term_1;
28      real term_2;
29      real term_3;
30      real term_4;
31      real cdf;
32
33      b_A_tv <- b - A - t*v;
34      b_tv <- b - t*v;
35      ts <- t*s;
36      term_1 <- b_A_tv/A * Phi(b_A_tv/ts);
37      term_2 <- b_tv/A * Phi(b_tv/ts);
38      term_3 <- ts/A * exp(normal_log(b_A_tv/ts,0,1));
39      term_4 <- ts/A * exp(normal_log(b_tv/ts,0,1));
40      cdf <- 1 + term_1 - term_2 + term_3 - term_4;
41
42      return cdf;
43
44  }
```

Box A2 Cumulative distribution function of the LBA,
continued from Box A1

```

22 real lba_cdf(real t, real b, real A, real v, real s){
23
24     real b_A_tv;
25     real b_tv;
26     real ts;
27     real term_1;
28     real term_2;
29     real term_3;
30     real term_4;
31     real cdf;
32
33     b_A_tv <- b - A - t*v;
34     b_tv <- b - t*v;
35     ts <- t*s;
36     term_1 <- b_A_tv/A * Phi(b_A_tv/ts);
37     term_2 <- b_tv/A * Phi(b_tv/ts);
38     term_3 <- ts/A * exp(normal_log(b_A_tv/ts,0,1));
39     term_4 <- ts/A * exp(normal_log(b_tv/ts,0,1));
40     cdf <- 1 + term_1 - term_2 + term_3 - term_4;
41
42     return cdf;
43
44 }
```

The first thing to note is that both are real-valued functions of type `real`. They both take as arguments the decision time, t , the decision threshold, b , the maximum starting evidence, A , the drift rate, v , and the standard deviation, s . Lines 4 through 10 in Box A1 and lines 24 through 31 in Box A2 define all local variables that will be used in each computation. After defining the local variables, the PDF or CDF is computed and the result is returned. Some built-in functions allow for an easier computation of the PDF and CDF. The `Phi` function is a built-in Stan function that implements the normal cumulative distribution function. The `exp` function is the exponential function, and the `normal_log` function is the natural logarithm of the PDF of the normal distribution, where the last two arguments are the mean and standard deviation, respectively.

Box A3 implements the LBA model in Stan. This code is based on the “`rtdists`” package for R (Singman et al., 2015), which can be found online (<https://cran.r-project.org/web/packages/rtdists/index.html>). All Stan functions that generate samples from a given distribution are called random number generators (RNGs). To distinguish between functions, RNGs must contain the `_rng` suffix. For example, the RNG for the exponential distribution is called `exponential_rng`. Here, we

name the function that generates samples from the LBA model `lba_rng`. After defining the local variables, the drift rates for each accumulator are drawn from the normal distribution. Negative drift rates result in negative response times, and drift rates of zero result in undefined response times. The LBA model that we implement assumes that at least one accumulator has a positive drift rate, and therefore no negative or undefined response times. This is achieved in the `while` loop beginning on line 64, in which drift rates are drawn from the normal distribution until at least one drift rate is positive. The loop terminates after a maximum of 1,000 iterations if at least one positive drift rate has not been drawn. If this is the case, a negative value is returned, denoting an undefined response time (lines 79 and 80). In practice, we have found this works very well and will not return negative or undefined response times given a reasonable model and data. After drawing the drift rates, the start points for each accumulator are drawn (line 84). The finishing times of each accumulator are computed according to the processing assumptions of the LBA on line 85. Lastly, the response alternative and lowest positive response time are stored in the `pred` vector and returned.

Box A3 LBA random number generator (RNG). The model assumes that on every trial at least one of the drift rates is positive. Code is continued from Box A2.

```

45  vector lba_rng(real k, real A, vector v, real s, real tau){
46
47      int get_pos_drift;
48      int no_pos_drift;
49      int get_first_pos;
50      vector[num_elements(v)] drift;
51      int max_iter;
52      int iter;
53      real start[num_elements(v)];
54      real ttf[num_elements(v)];
55      int resp[num_elements(v)];
56      real rt;
57      vector[2] pred;
58      real b;
59
60      get_pos_drift <- 1;
61      no_pos_drift <- 0;
62      max_iter <- 1000;
63      iter <- 0
64      while (get_pos_drift){ //try to get a positive drift rate
65          for (j in 1:num_elements(v)){
66              drift[j] <- normal_rng(v[j],s);
67              if (drift[j] > 0){
68                  get_pos_drift <- 0;
69              }
70          }
71          iter <- iter + 1;
72          if (iter > max_iter){
73              get_pos_drift <- 0;
74              no_pos_drift <- 1;
75          }
76      }
77      //return a negative response time if all < 0
78      if (no_pos_drift){
79          pred[1] <- -1;
80          pred[2] <- -1;
81      }else{
82          b <- A + k;
83          for (i in 1:num_elements(v))
84              start[i] <- uniform_rng(0,A); //start point
85              ttf[i] <- (b-start[i])/drift[i]; //finish times
86          }
87          //rt is the minimum non-negative finish time
88          resp <- sort_indices_asc(ttf);
89          ttf <- sort_asc(ttf);
90          get_first_pos <- 1;
91          iter <- 1;
92          while (get_first_pos){
93              if (ttf[iter] > 0){
94                  pred[1] <- ttf[iter] + tau;
95                  pred[2] <- resp[iter];
96                  get_first_pos <- 0;
97              }
98              iter <- iter + 1;
99          }
100     }
101     return pred;
102 }

```

References

- Brooks, S., Gelman, A., Jones, G. L., & Meng, X. L. (2011). *Handbook of Markov chain Monte Carlo*. Boca Raton, FL: Chapman & Hall/CRC.
- Brown, S. D., & Heathcote, A. (2008). The simplest complete model of choice reaction time: Linear ballistic accumulation. *Cognitive Psychology*, *57*, 153–178. doi:10.1016/j.cogpsych.2007.12.002
- Donkin, C., Brown, S. D., & Heathcote, A. (2009). The overconstraint of response time models: Rethinking the scaling problem. *Psychonomic Bulletin & Review*, *16*, 1129–1135. doi:10.3758/PBR.16.6.1129
- Duane, S., Kennedy, A. D., Pendleton, B. J., & Roweth, D. (1987). Hybrid Monte Carlo. *Physics Letters B*, *195*, 216–222.
- Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian data analysis* (3rd ed.). Boca Raton, FL: Chapman & Hall/CRC.
- Gelman, A., & Rubin, D. B. (1992). Inference from iterative simulation using multiple sequences. *Statistical Science*, *7*, 457–511.
- Gilks, W. R., Richardson, S., & Spiegelhalter, D. J. (1996). *Markov chain Monte Carlo in practice*. Boca Raton, FL: Chapman & Hall/CRC.
- Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, *57*, 97–109.
- Hoffman, M. D., & Gelman, A. (2014). The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, *15*, 1351–1381.
- Kruschke, J. K. (2011). *Doing Bayesian data analysis: A tutorial with R and BUGS*. Burlington, MA: Academic Press.
- Kucukelbir, A., Tran, D., Rajesh, R., Gelman, A., & Blei, D. M. (submitted). Automatic differentiation variational inference. Retrieved May 5, 2016, from <http://arxiv.org/pdf/1603.00788v1.pdf>
- Lee, M. D., & Wagenmakers, E.-J. (2014). *Bayesian cognitive modeling: A practical course*. New York, NY: Cambridge University Press.
- Lewandowsky, S., & Farrell, S. (2011). *Computational modeling in cognition: Principles and practice*. Thousand Oaks, CA: Sage.
- Link, W. A., & Eaton, M. J. (2011). On thinning of chains in MCMC. *Methods in Ecology and Evolution*, *3*, 112–115.
- Lunn, D. J., Thomas, A., Best, N., & Spiegelhalter, D. (2000). WinBUGS—A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, *10*, 325–337.
- Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, *21*, 1087–1092.
- Neal, R. M. (2011). MCMC using Hamiltonian dynamics. In S. Brooks, A. Gelman, G. L. Jones, & X. L. Meng (Eds.), *Handbook of Markov chain Monte Carlo* (pp. 113–162). Boca Raton, FL: Chapman & Hall/CRC.
- Palmeri, T. J., Schall, J. D., & Logan, G. D. (2015). Neurocognitive modelling of perceptual decisions. In J. R. Busemeyer, Z. Wang, J. T. Townsend, & A. Eidels (Eds.), *Oxford handbook of computational and mathematical psychology* (pp. 320–340). Oxford, UK: Oxford University Press. doi:10.1093/oxfordhb/9780199957996.013.15
- Plummer, M. (2003, March). *JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling*. Paper presented at the 3rd International Workshop on Distributed Statistical Computing, Vienna, Austria.
- Purcell, B. A., Schall, J. D., Logan, G. D., & Palmeri, T. J. (2012). Gated stochastic accumulator model of visual search decisions in FEF. *Journal of Neuroscience*, *32*, 3433–3446.
- Ratcliff, R., & McKoon, G. (2008). The diffusion decision model: Theory and data for two-choice decision tasks. *Neural Computation*, *20*, 873–922. doi:10.1162/neco.2008.12.06.420
- Ratcliff, R., Philiastides, M. G., & Sajda, P. (2009). Quality of evidence for perceptual decision making is indexed by trial-to-trial variability of the EEG. *Proceedings of the National Academy of Sciences*, *106*, 6539–6544.
- Ratcliff, R., Thapar, A., & McKoon, G. (2010). Individual differences, aging, and IQ in two-choice tasks. *Cognitive Psychology*, *60*, 127–157. doi:10.1016/j.cogpsych.2009.09.001
- Robert, C., & Casella, G. (2004). *Monte Carlo statistical methods*. New York, NY: Springer.
- Singman, H., Brown, S., Gretton, M., Heathcote, A. Voss, A., Voss, J., & Terry, A. (2016). rtdists: Response time distributions (Version 0.4-9).
- Stan Development Team. (2015). Stan: A C++ library for probability and sampling (Version 2.8.0).
- Thomas, A., O'Hara, B., Ligges, U., & Sturtz, S. (2006). Making BUGS Open. *R News*, *6*(1), 12–17.
- Turner, B. M., & Sederberg, P. B. (2012). Approximate Bayesian computation with differential evolution. *Journal of Mathematical Psychology*, *56*, 375–385.
- Turner, B. M., Sederberg, P. B., Brown, S. D., & Steyvers, M. (2013). A method for efficiently sampling from distributions with correlated dimensions. *Psychological Methods*, *18*, 368–384. doi:10.1037/a0032222
- Turner, B. M., Van Maanen, L., & Forstmann, B. U. (2015). Combining cognitive abstractions with neurophysiology: The neural drift diffusion model. *Psychological Review*, *122*, 312–336.
- van Maanen, L., Brown, S. D., Eichele, T., Wagenmakers, E.-J., Ho, T., Serences, J. T., & Forstmann, B. U. (2011). Neural correlates of trial-to-trial fluctuations in response caution. *Journal of Neuroscience*, *31*, 17488–17495. doi:10.1523/JNEUROSCI.2924-11.2011
- Wabersich, D., & Vandekerckhove, J. (2014). Extending JAGS: A tutorial on adding custom distributions to JAGS (with a diffusion model example). *Behavior Research Methods*, *46*, 15–28.