Psychon Bull Rev (2018) 25:77–101
https://doi.org/10.3758/s13423-017-1394-5

Bayesian inference for psychology, part III: Parameter estimation in nonstandard models

Dora Matzke[1] · Udo Boehm[2] · Joachim Vandekerckhove[3]

Published online: 13 November 2017

Abstract We demonstrate the use of three popular Bayesian software packages that enable researchers to estimate parameters in a broad class of models that are commonly used in psychological research. We focus on WinBUGS, JAGS, and Stan, and show how they can be interfaced from R and MATLAB. We illustrate the use of the packages through two fully worked examples; the examples involve a simple univariate linear regression and fitting a multinomial processing tree model to data from a classic false-memory experiment. We conclude with a comparison of the strengths and weaknesses of the packages. Our example code, data, and this text are available via https://osf.io/ucmaz/.

Keywords WinBUGS · JAGS · Stan · Bayesian estimation · Bayesian inference

Introduction

In this special issue, Dienes (this issue) has argued that Bayesian methods are to be preferred over classical methods, Kruschke (this issue) and Etz and Vandekerckhove (this issue) have introduced the Bayesian philosophy and associated mathematics, and Love et al. (this issue; see also Love et al. (2015)) and Wagenmakers et al. (this issue) described software that implements standard hypothesis tests within the Bayesian framework. In the present paper, we demonstrate the use of three popular software packages that enable psychologists to estimate parameters in formal models of varying complexity.

The mathematical foundations of Bayesian parameter estimation are not especially difficult—all that is involved are the elementary laws of probability theory to determine the posterior distribution of parameters given the data. Once the posterior distribution has been defined, the final hurdle of Bayesian parameter estimation is to compute descriptive statistics on the posterior. In order to obtain these descriptive statistics, one widely applicable strategy is to draw random samples from the posterior distribution using Markov chain Monte Carlo methods (MCMC; van Ravenzwaaij, this issue)—with sufficient posterior samples, descriptives on the sample set can substitute for actual quantities of interest.

In this article, we describe the use of three popular, general-purpose MCMC engines that facilitate the sampling process. We will focus on WinBUGS, JAGS, and Stan, and illustrate their use for parameter estimation in two popular models in psychology. The development of these software packages has greatly contributed to the increase in the prevalence of Bayesian methods in psychology over the past decade (e.g., Lee & Wagenmakers, 2013). The packages owe their popularity to their flexibility and usability; they allow researchers to build a large number of models of varying complexity using a relatively small set of sampling statements and deterministic transformations. Moreover, the packages have a smooth learning curve, are well documented, and are supported by a large community of users both within and outside of psychology. Their popularity notwithstanding, WinBUGS, JAGS, and Stan represent only a subclass of the many avenues to Bayesian analysis; the different avenues implement a trade–off between flexibility and accessibility. At one end of the spectrum,

✉ Joachim Vandekerckhove
joachim@uci.edu

[1] University of Amsterdam, Amsterdam, Netherlands

[2] University of Groningen, Groningen, Netherlands

[3] University of California, Irvine, CA, USA

researchers may use off–the–shelf Bayesian software packages, such as JASP (Love et al. this issue; see also Love et al., 2015). JASP has an attractive and user-friendly graphical user interface, but presently it only supports standard hypothesis tests (see also Morey et al., 2015). At the other end of the spectrum, researcher may implement their own MCMC sampler, one that is tailored to the peculiarities of the particular model at hand (e.g., van Ravenzwaaij, this issue; Rouder & Lu, 2005). This approach provides tremendous flexibility, but it is time-consuming, labor-intensive, and requires expertise in computational methods. General-purpose MCMC engines—such as WinBUGS, JAGS, and Stan—are the middle-of-the-road alternatives to Bayesian analysis that provide a large degree of flexibility at a relatively low cost.

We begin with a short introduction of formal models as generative processes using a simple linear regression as an example. We then show how this model can be implemented in WinBUGS, JAGS, and Stan, with special emphasis on how the packages can be interacted with from R and MATLAB. We then turn to a more complex model, and illustrate the basic steps of Bayesian parameter estimation in a multinomial processing tree model for a false-memory paradigm. The WinBUGS, JAGS, and Stan code for all our examples is available in the Supplemental Materials at https://osf.io/ucmaz/. The discussion presents a comparison of the strengths and weaknesses of the packages and provides useful references to hierarchical extensions and Bayesian model selection methods using general-purpose MCMC software.

An introduction with linear regression

Specification of models as generative processes

Before we continue, it is useful to consider briefly what we mean by a formal *model*: A formal model is a set of formal statements about how the data come about. Research data are the realizations of some stochastic process, and as such they are draws from some random number generator whose properties are unknown. In psychology, the random number generator is typically a group of randomly selected humans who participate in a study, and the properties of interest are often differences in group means between conditions or populations (say, the difference in impulsivity between schizophrenia patients and controls) or other invariances and systematic properties of the data generation process. A formal model is an attempt to emulate the unknown random number generator in terms of a network of basic distributions.

Consider, for example, simple linear regression, with its three basic assumptions of *normality*, *linearity*, and *homoskedasticity*. This common technique implies a stochastic process: the data are assumed to be random draws from a normal distribution (normality), whose mean is a linear function of a predictor (linearity), and whose variance is the same (homoskedasticity) for all units, where "units" can refer to participants, items, conditions, and so on. A regression model in which we predict y from x may be written as a follows:

$$y_i|\mu_i,\tau \sim \mathcal{N}(\mu_i,\tau) \tag{1}$$
$$\mu_i|\beta_1,\beta_2,x_i = \beta_1+\beta_2 x_i. \tag{2}$$

The tilde ($\sim$) may be read as "is a random sample from". These two statements encode the assumptions of normality (1), homoskedasticity across units i (1), and linearity (2). Usually omitted, but implied, is that these statements hold true for all values that the subscript i can take:

$$\forall i,\ i=1,\ldots,N. \tag{3}$$

We use τ to indicate the *precision*—the inverse of the variance—because that is how WinBUGS and JAGS parameterize the Gaussian distribution.

In the Bayesian framework, we must further specify our prior assumptions regarding the model parameters β_1, β_2, and τ. Let us use the following[1] forms for the priors:

$$\beta_1 \sim \mathcal{N}(0,0.001) \tag{4}$$
$$\beta_2 \sim \mathcal{N}(0,0.001) \tag{5}$$
$$\tau \sim \Gamma(0.001,0.001). \tag{6}$$

This simple model also helps to introduce the types of variables that we have at our disposal. Variables can be *stochastic*, meaning that they are draws from some distribution. Stochastic variables can be either observed (i.e., data) or unobserved (i.e., unknown parameters). In this model, y, β_1, β_2, and τ are stochastic variables. Variables can also be *deterministic*, which means their values are completely determined by other variables. Here, μ_i is determined as some combination of β_1, β_2, and x_i. N is a constant.

Taken together, a Bayesian model can be thought of as a data-generation mechanism that is conditional on parameters: Bayesian models make *predictions*. In particular, the *sampling statements*— including the priors—in Eqs. 1, 4, 5, and 6 and the deterministic transformation in Eq. 2, fully define a *generative model*; this set of statements fully defines the model because they are all that is needed to

[1]We chose values for the parameters of the prior distributions that fit the introductory example. In general, these values should depend on the application at hand (see Vanpaemel & Lee, this issue; and Morey, this issue).

generate data from the model. The generative model thus formalizes the presumed process by which the data in an empirical study were generated.

A toy data set

As our introductory example, we will use a small data set containing (a) the observed number of attendees at each session of a recent conference (the data y) and (b) the number of attendees that was expected by the organizers (the predictor x). Table 1 shows the data set.

Implementing a generative model

The generative specification is the core of the BUGS modeling language (Lunn et al., 2000) that is used by WinBUGS and dialects of which are used by JAGS and Stan. In all of these programs, the model definition consists of a generative specification. In many cases, the model code is almost a point-to-point translation of a suitable generative specification. Consider this BUGS implementation of the linear regression model:

```
model {
   # linear regression
   for (i in 1:N) {                          # Eq. 3
      y[i] ~ dnorm(mu[i],  tau)              # Eq. 1
      mu[i] <- beta[1] + beta[2] * x[i]      # Eq. 2
   }
   # prior definitions
   beta[1] ~ dnorm(0, 0.001)                 # Eq. 4
   beta[2] ~ dnorm(0, 0.001)                 # Eq. 5
   tau ~ dgamma(0.001, 0.001)                # Eq. 6
}
```

Table 1 Example data set for linear regression

Expected (x)			Observed (y)			
51	24	32	33	35	32	x < – c(51, 44, 57, 41, 53, 56,
44	21	42	55	18	31	49, 58, 50, 32, 24, 21,
57	23	27	49	14	37	23, 28, 22, 30, 29, 35,
41	28	38	56	31	17	18, 25, 32, 42, 27, 38,
53	22	32	58	13	11	32, 21, 21, 12, 29, 14)
56	30	21	61	23	24	y < – c(33, 55, 49, 56, 58, 61,
49	29	21	46	15	17	46, 82, 53, 33, 35, 18,
58	35	12	82	20	5	14, 31, 13, 23, 15, 20,
50	18	29	53	20	16	20, 33, 32, 31, 37, 17,
32	25	14	33	33	7	11, 24, 17, 5, 16, 7)

Attendance at each session of a conference, as predicted by the organizers (*left*) and as observed (*middle*), with the corresponding "S-style" data file (*right*)

The parameter `beta[1]` denotes the intercept (i.e., observed number of attendees for 0 expected attendees), `beta[2]` denotes the slope of the regression line (i.e., the increase in the observed number of attendees associated with a one-unit increase in the expected number of attendees), and `tau` represents the inverse of the error variance. This short piece of code maps exactly to the generative model for linear regression that we specified. Of course, since there is much more freedom in mathematical expression than there is in computer code, the point-to-point translations will not always be perfect, but it will typically be an excellent starting point.

In the code, deterministic variables are followed by the `<-` assignment operator. For instance, the line `mu[i] <- beta[1] + beta[2] * x[i]` specifies that the `mu` parameters are given by a linear combination of the of the stochastic `beta` variables and the observed data `x`. The # symbol is used for comments. The complete list of distributions, functions, logical operators, and other programming constructs that are available in WinBUGS, JAGS, and Stan, is listed in their respective user manuals. BUGS is a declarative language, which means that the order of the statements in the model file is largely irrelevant. In contrast, in Stan, the order of statements matters. With the model translated from formal assumptions to BUGS language, the next step is to interact with the software and sample from the posterior distribution of the parameters.

WinBUGS graphical user interface

WinBUGS (Bayesian inference Using Gibbs Sampling for Windows; Lunn et al., 2000, 2009, 2012; Spiegelhalter et al., 2003; for an introduction see Kruschke, 2010, and Wagenmakers, 2013) is a stand-alone piece of software that is freely available at http://www.mrc-bsu.cam.ac.uk/bugs/. In this section, we give a brief description of the WinBUGS graphical user interface (GUI) using the linear regression model introduced above; later we illustrate how WinBUGS can be called from other software, such as R and MATLAB. For a detailed step-by-step introduction to the WinBUGS GUI, the reader is referred to Lee and Wagenmakers (2013).

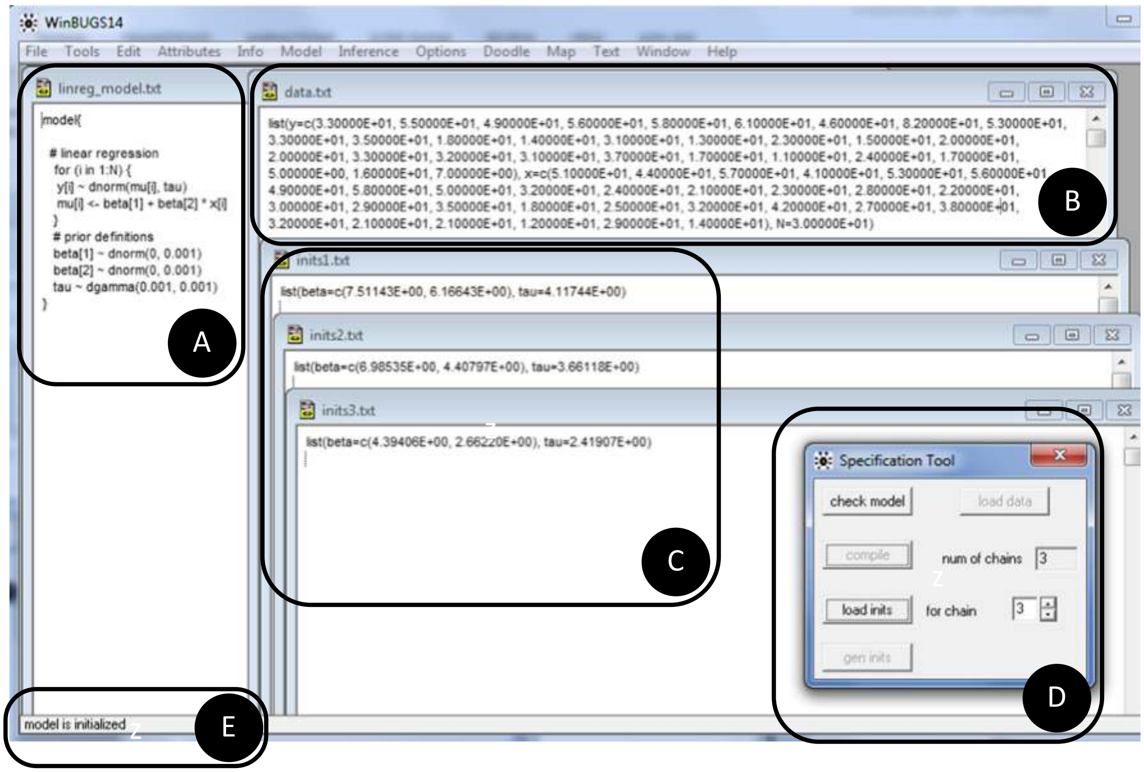
To interact with WinBUGS via the GUI, users have to create a number of files. First, there is a *model file* that describes the generative specification of the model, second is the *data file* that contains the raw data, and third is an *initial values file* that contains some starting values for the sampling run.

Panel A in Fig. 1 shows the model file `linreg_model.txt` that describes the generative model for the linear regression example. Panel B shows the data file `data.txt`. The data specification follows S-plus object notation, where vectors are encapsulated in the concatenation operator `c(...)` and matrices are defined as structures with a dimension field, such as `structure(.Data = c(...),.Dim = c(R, C))`, where `R` stands for the number of rows and `C` for the number of columns. In the linear regression example, the data consist of the vector of observations `y` corresponding to the observed number of attendees, a vector of observations `x` corresponding to the predicted number of attendees, and a scalar `N` corresponding to the number of sessions.

The same data format is used to store the (optional, but strongly recommended) set of initial values for the unobserved stochastic variables. If initial values are not supplied, WinBUGS will generate these automatically by sampling from the prior distribution of the parameters. Automatically generated initial values can provide poor starting points for the sampling run and may result in numerical instability. If multiple MCMC chains are run in order to diagnose convergence problems, we encourage users to create a separate file for each set of initial values. As shown in Fig. 1c, we will run three chains, each with a different set of initial values, and store these in `inits1.txt`, `inits2.txt`, and `inits3.txt`.

Once the model file, the data file, and the files containing the initial values are created, follow the steps outlined below to sample from the posterior distribution of the parameters.

1. Load the model file and check the model specification. To open the model file, go to `File -> Open` and select `linreg_model.txt` in the appropriate directory. To check the syntax of the model specification, go to `Model -> Specification` and open the `Specification Tool` window (Panel D in Fig. 1), activate the model file by clicking inside `linreg_model.txt`, click on `check model`, and wait for the message "model is syntactically correct" to appear in the status bar.
2. Load the data file. To open the data file, go to `File -> Open` and select `data.txt` in the appropriate directory. To load the data, activate the data file, click



```
model{

  # linear regression
   for (i in 1:N) {
    y[i] ~ dnorm(mu[i], tau)
    mu[i] <- beta[1] + beta[2] * x[i]
   }
   # prior definitions
   beta[1] ~ dnorm(0, 0.001)
   beta[2] ~ dnorm(0, 0.001)
   tau ~ dgamma(0.001, 0.001)
}
```

```
list(y=c(3.30000E+01, 5.50000E+01, 4.90000E+01, 5.60000E+01, 5.80000E+01, 6.10000E+01, 4.60000E+01, 8.20000E+01, 5.30000E+01,
3.30000E+01, 3.50000E+01, 1.80000E+01, 1.40000E+01, 3.10000E+01, 1.30000E+01, 2.30000E+01, 1.50000E+01, 2.00000E+01,
2.00000E+01, 3.30000E+01, 3.20000E+01, 3.10000E+01, 3.70000E+01, 1.70000E+01, 1.10000E+01, 2.40000E+01, 1.70000E+01,
5.00000E+00, 1.60000E+01, 7.00000E+00), x=c(5.10000E+01, 4.40000E+01, 5.70000E+01, 4.10000E+01, 5.30000E+01, 5.60000E+01
4.90000E+01, 5.80000E+01, 5.00000E+01, 3.20000E+01, 2.40000E+01, 2.10000E+01, 2.30000E+01, 2.80000E+01, 2.20000E+01,
3.00000E+01, 2.90000E+01, 3.50000E+01, 1.80000E+01, 2.50000E+01, 3.20000E+01, 4.20000E+01, 2.70000E+01, 3.80000E+01,
3.20000E+01, 2.10000E+01, 2.10000E+01, 1.20000E+01, 2.90000E+01, 1.40000E+01), N=3.00000E+01)
```

```
list(beta=c(7.51143E+00, 6.16643E+00), tau=4.11744E+00)
```

```
list(beta=c(6.98535E+00, 4.40797E+00), tau=3.66118E+00)
```

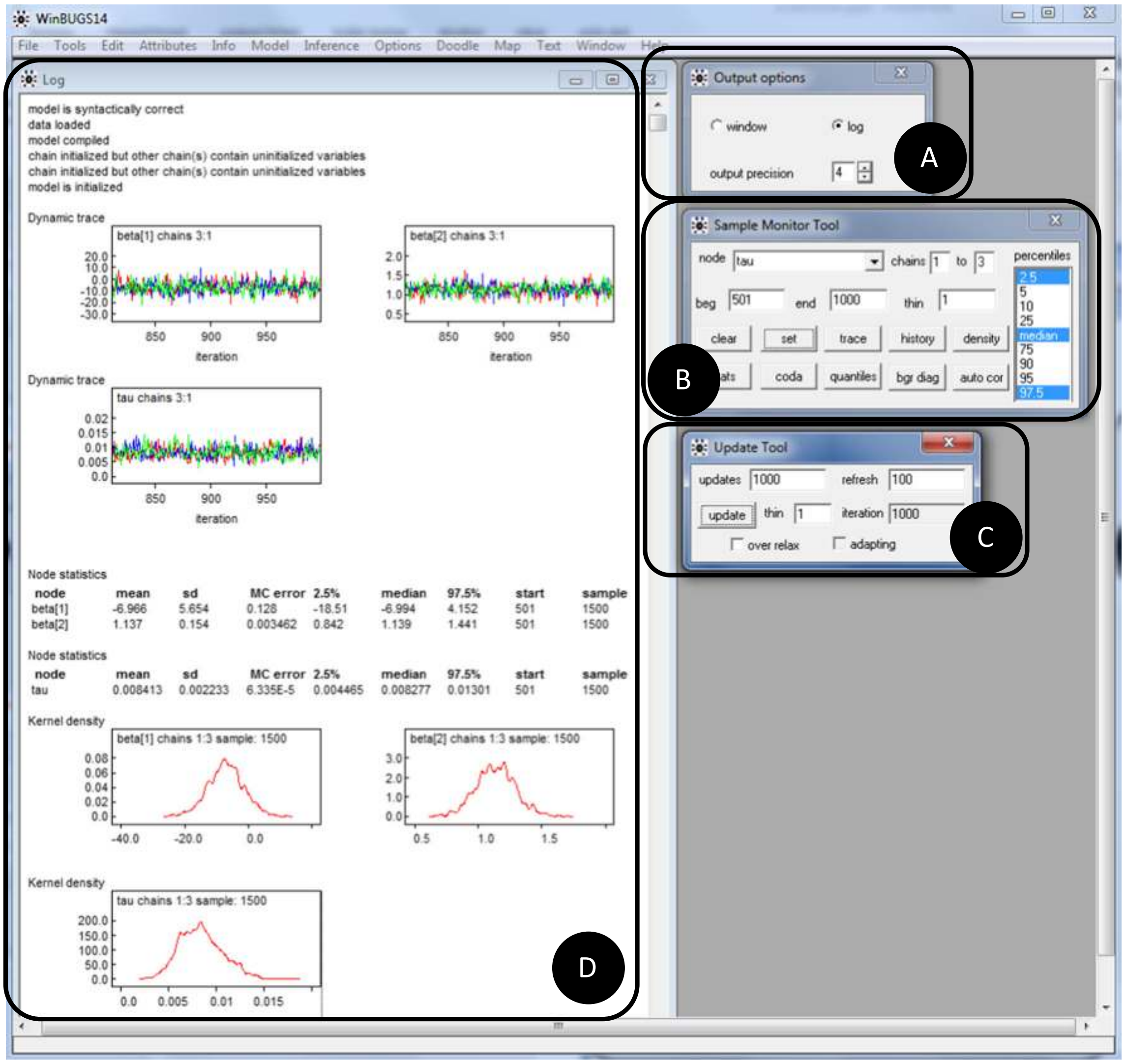
```
list(beta=c(4.39406E+00, 2.66220E+00), tau=2.41907E+00)
```

```
model is initialized
```

Fig. 1 *The WinBUGS graphical user interface.* **a** The model file. **b** The data file. **c** The initial values. **d** The Specification Tool window. **e** The status bar

on `load data` in the `Specification Tool` window, and wait for the message "data loaded" to appear in the status bar.

3. Compile the model. To compile the model, specify the number of MCMC chains in the box labeled `num of chains` in the `Specification Tool` window, click on `compile`, and wait for the message "model compiled" to appear in the status bar. In the linear regression example, we will run three MCMC chains, so we type "3" in the `num of chains` box.
4. Load the initial values. To open the file that contains the initial values for the first chain, go to `File -> Open` and select `inits1.txt` in the appropriate directory. To load the first set of initial values, activate `inits1.txt`, click on `load inits` in the `Specification Tool` window, and wait for the message "chain initialized but other chain(s) contain uninitialized variables". Repeat these steps to load the initial values for the second and third MCMC chain. After the third set of initial values is loaded, wait for the message "model is initialized" to appear in the status bar (Fig. 1e).
5. Choose the output type. To ensure that WinBUGS pastes all requested output in a single user-friendly log file, go to `Output -> Output options`, open the `Output options` window, and select the `log` option (Fig. 2a).
6. Specify the parameters of interest. To specify the parameters that you want to draw inference about, go to `Inference -> Samples`, open the `Sample Monitor Tool` window, type one by one the name of the parameters in the box labeled `node`, and click on



```
model is syntactically correct
data loaded
model compiled
chain initialized but other chain(s) contain uninitialized variables
chain initialized but other chain(s) contain uninitialized variables
model is initialized
```

```
Node statistics
 node      mean      sd        MC error  2.5%      median    97.5%    start  sample
 beta[1]   -6.966    5.654     0.128     -18.51    -6.994    4.152    501    1500
 beta[2]   1.137     0.154     0.003462  0.842     1.139     1.441    501    1500

Node statistics
 node      mean      sd        MC error  2.5%      median    97.5%    start  sample
 tau       0.008413  0.002233  6.335E-5  0.004465  0.008277  0.01301  501    1500
```

Output options: ( ) window [✓ log]; output precision 4

Sample Monitor Tool: node tau; chains 1 to 3; beg 501; end 1000; thin 1; percentiles: [✓ 2.5], 5, 10, 25, [✓ median], 75, 90, 95, [✓ 97.5]

Update Tool: updates 1000; refresh 100; thin 1; iteration 1000

Fig. 2 *The WinBUGS graphical user interface continued.* **a** The Output Options window. **b** The Sample Monitor Tool window. **c** The Update Tool window. **d** The log file

`set` (Fig. 2b). In the linear regression example, we will monitor the `beta[1]`, `beta[2]`, and `tau` parameters. To request dynamic trace plots of the progress of the sampling run, select the name of the parameters in the drop-down menu in the `Sample Monitor Tool` window and click on `trace`. WinBUGS will start to display the dynamic trace plots once the sampling has begun.

7. Specify the number of recorded samples. To specify the number of recorded samples per chain, fill in the boxes labeled `beg`, `end`, and `thin` in the `Sample Monitor Tool` window. In our linear regression example, we will record 500 posterior samples for each parameter. We will discard the first 500 samples as burn-in and start recording samples from the 501^{th} iteration (`beg=501`); we will draw a total of 1,000 samples (`end=1000`); and we will record each successive sample without thinning the chains (`thin=1`).
8. Sample from the posterior distribution of the parameters. To sample from the posteriors, go to `Model -> Update`, open the `Update Tool` window (Fig. 2c), fill in the total number of posterior samples per chain (i.e., 1,000) in the box labeled `updates`, specify the degree of thinning (i.e., 1) in the box labeled `thin`, click on `update`, and wait for the message "model is updating" to appear in the status bar.
9. Obtain the results of the sampling run. To obtain summary statistics and kernel density plots of the posterior distributions, select the name of the parameters in the drop-down menu in the `Sample Monitor Tool` window and click on `stat` and `density`. WinBUGS will print all requested output in the log file (Panel D in Fig. 2). The figures labeled "Dynamic trace" show trace plots of the monitored parameters; the three MCMC chains have mixed well and look identical to one another, indicating that the chains have converged to the stationary distribution and that the successive samples are largely independent. The table labeled "Node statistics" shows summary statistics of the posterior distribution of the parameters computed based on the sampled values. For each monitored parameter, the table displays the mean, the median, the standard deviation, and the upper-and lower bound of the central 95% credible interval of the posterior distribution. The central tendency of the posterior, such as the mean, can be used as a point estimate for the parameter. This 95% credible interval ranges from the 2.5^{th} to the 97.5^{th} percentile of the posterior and encompasses a range of values that contains the true value of the parameter with 95% probability; the narrower this 95% credible interval, the more precise the parameter estimate. The figures labeled "Kernel density" show density plots of the posterior samples for each parameter.

As the reader might have noticed by now, running analyses via the GUI is inflexible and labor-intensive; the GUI does not allow for data manipulation and visualization and requires users to click through a large number of menus and options. Later we therefore illustrate how WinBUGS can be called from standard statistical software, such as R and MATLAB.

JAGS and Stan command-line interface

Both JAGS and Stan are based on a command-line interface. Although this type of interface has fallen out of fashion, and it is strictly speaking not required to use either of these programs, we introduce this low-level interface here—using JAGS as the example—in order to provide the reader with an appreciation of the inner workings of other interfaces. Readers who are not interested in this can skip to either one of the next two sections.

Before launching the program, it is again useful to make a set of text files containing the model, data, and initial values. The model file should contain the code in the listing above; for this example, we saved the model in `linreg_model.txt`.

The data file should contain the data, formatted as in the right column of Table 1. The data format in Table 1 is sometimes referred to as "S-style"; each variable name is given in double quotation marks, followed by the assignment operator `<-` and the value to be assigned to the variable. Vectors are encapsulated in the concatenation operator `c(...)` and matrices are defined as structures with a dimension field: `struct(c(...),.Dim=c(R,C))`, where the $R \times C$ matrix is entered in column-major order. Our data file is called `linreg_data.txt`.

The same data format is used to store the (optional, but strongly recommended) set of initial values. For at least some of the unknowns nodes (i.e., nodes which in the BUGS code are followed by the sampling operator $\sim$), initial values should be provided. If multiple chains will be run, one unique file for each chains is recommended. Our initial values files are called `inits1.txt`, `inits2.txt`, and `inits3.txt`.

Once all these files are in place, start JAGS by opening a command window and typing `jags`. Below is the complete interaction with JAGS, in which user input is preceded by the period (`.`) prompt. Comments are preceded by a pound sign `#`.

```
~$ jags
Welcome to JAGS 3.4.0 on Mon Jul 20 14:02:50 2015
JAGS is free software and comes with ABSOLUTELY NO WARRANTY
Loading module: basemod: ok
Loading module: bugs: ok
. model in "linreg_model.txt"       # loads the model
. data in "linreg_data.txt"         # loads the data
Reading data file linreg_data.txt
. compile, nchains(3)               # compiles the model for 3 chains
Compiling model graph
   Resolving undeclared variables
   Allocating nodes
   Graph Size: 117
. parameters in "inits1.txt"        # loads initial values for chain 1
Reading parameter file inits1.init
. parameters in "inits2.txt"        # loads initial values for chain 2
Reading parameter file inits2.txt
. parameters in "inits3.txt"        # loads initial values for chain 3
Reading parameter file inits3.txt
. initialize                        # sets up the sampling algorithms
Initializing model
. update 500                        # draws 500 samples for burn-in
Updating 500
-------------------------------------------------| 500
************************************************** 100%
. monitor set beta, thin(1)         # indicates a variable to save
. monitor set tau, thin(1)          # indicates a variable to save
. update 500                        # draws 500 samples from posterior
Updating 500
-------------------------------------------------| 500
************************************************** 100%
. coda *, stem('samples_')          # saves posterior samples to files
. exit                              # exits
```

This will produce a set of files starting with `samples_chain` and an index file starting with `samples_index`. These files can be loaded into a spreadsheet program like Microsoft Excel or LibreOffice Calc (or command line tools like awk and perl) to compute summary statistics and do inference. However, this approach is both tedious and labor-intensive, so there exist convenient interfaces from programming languages such as R, MATLAB, and Python.

Working from MATLAB

MATLAB is a commercial software package that can be obtained via http://www.mathworks.com/. Just like Python or R, MATLAB can be used to format data, generate initial values, and visualize and save results of a sampling run. In this section we outline how users can interact with WinBUGS, JAGS, and Stan using MATLAB. R users can skip this section; in the next sections, we will describe how to use R for the same purposes.

To interact with the three computational engines from MATLAB, we will use the Trinity toolbox (Vandekerckhove, 2014), which is developed as a unitary interface to the Bayesian inference engines WinBUGS, JAGS, and Stan. Trinity is a work-in-progress that is (and will remain) freely available via http://tinyurl.com/matlab-trinity. The MATLAB code needed to call these three engines from Trinity is essentially identical.

To start Trinity, download the toolbox, place it in your MATLAB path, and then call:

```
>> trinity install
>> trinity new
```

The first line will cause the Trinity files to be detected by MATLAB and the second line will create a bare-bones MATLAB script with programming instructions. For example, one line reads:[2]

```
% Write the model into a variable
(cell variable)
model = {
   %$ MODEL GOES HERE $%
   };
```

The user can then enter the model code directly into the MATLAB script, using cell string notation (note the single quotes around each line):

```
model = {
   'model {'
   '   # linear regression'
   '   for (i in 1:N) {'
   '      y[i] ~ dnorm(mu[i], tau)'
   '      mu[i] <- beta[1] + beta[2] * x[i]'
   '   }'
   '   # prior definitions'
   '   beta[1] ~ dnorm(0, 0.001)'
   '   beta[2] ~ dnorm(0, 0.001)'
   '   tau ~ dgamma(0.001, 0.001)'
   '}'
    };
```

It is also possible to write the model in a separate file and provide the file name here instead of the model code. One advantage of writing model code directly into the MATLAB script is that the script can be completely self-contained. Another is that the model code, when treated as a MATLAB variable, could be generated on-the-fly if tedious or repetitive code is required to define a model or if the model file needs to be adapted dynamically (e.g., if variable names need to change from one run to another).

Next, we need to list the parameters of interest (i.e., for which variables we should save posterior samples). For our current application we could list all variables but choose to omit `mu` (which is particularly useful if `N` is large and the vector `mu` takes up much memory):

```
% List all the parameters of interest
(cell variable)
params = {
    'beta' 'tau'
  };
```

[2]It is likely that the exact appearance of this code will vary a little over successive versions of the Trinity toolbox, but the requirements will remain broadly the same.

Next, we collect the data variables that MATLAB will send to the computational engine. Again, it is possible to do this by providing the name to a properly formatted data file, but it is more practical to make a MATLAB variable that contains the data. To collect the data, make a *structure variable* as follows (for the example, `x` and `y` should first be defined with the values given in Table 1):

```
% Make a structure with the data (note
that the name of the field needs to
% match the name of the variable in
the model)
data = struct(...
    'x', 'x', ...
    'y', 'y', ...
    'N', numel(x) ...
    );
```

Each field name (in single quotes) is the name of the variable as it is used in the model definition.[3] Note that Trinity will not permit the model code to have variable names containing underscores, as this symbol is reserved for internal use. Following each field name is the value that this variable will take; this value is taken from the MATLAB workspace, so it can be an existing variable with any name, or it can be a MATLAB expression that generates the correct value, as we did here with `N`. Of course, before making this data structure, the data may need to be parsed, read into MATLAB, and possibly pre-processed (outliers removed, etc.).

The final block to complete is a little more involved and requires understanding of MATLAB's *anonymous functions* construct. Anonymous functions are in-line function definitions that are saved as variables. A typical command to define an anonymous function has the following structure:

$$\underbrace{\texttt{anonfun}}_{\textcircled{1}} = \overbrace{\texttt{@(a,b)}}^{\textcircled{2}}\ \underbrace{\texttt{3*a + sqrt(b)}}_{\textcircled{3}};$$

In this example, `anonfun` (part (1)) is the name given to the new function—this can be anything that is a valid MATLAB variable name. Part (2) indicates the start of an anonymous function with the `@` symbol and lists the input variables of the function between parentheses. Part (3) is a

[3]Because MATLAB does not differentiate between vectors and single-column or single-row matrices, but some of the computational engines do, it is sometimes convenient to pass variables explicitly as a matrix or explicitly as a vector. For this situation, Trinity allows the flags `AS_MATRIX_` and `AS_VECTOR_` to be prepended to any variable name. A common situation in which this is useful is when matrix multiplication is applied in the model, but one of the matrices has only one column. JAGS, for example, will treat that matrix as a vector and throw a "dimension mismatch" error unless the flag is applied. In our example, the data structure would then be defined as `struct('AS_MATRIX_x', x)`.

single MATLAB expression that returns the output variable, computed from inputs `a` and `b`. This anonymous function could be invoked with: `anonfun(1,4)`, which would yield 5.

It is possible for an anonymous function to take no (zero) input arguments. For example, `nrand = @()-rand` will create a function called `nrand` that generates uniformly distributed variates between −1 and 0. In order to supply the computational engine with initial values for the sampling process, we will define an anonymous function that draws a sample from the prior distribution of all or part of the parameter set. An example is:

```
% Write a function that generates a
structure with one random value for
% each parameter in a field
generator = @()struct(...
    'beta', randn(2, 1) * 10 + 0, ...
    'tau' , rand * 5 ...
    );
```

Here, a structure is generated with one field for each parameter, and a random initial value for each. The initial value for each of the two `betas` is generated from a normal distribution with mean 0 and standard deviation 10, and `tau` is generated from a uniform distribution between 0 and 5. The function `generator()` can now be called from MATLAB:

```
>> generator()
ans =
    beta: [2x1 double]
    tau: 0.6349
```

Note that either of these variables can be validly omitted, but at least one must be given. If one of the random number generators draws a value that is not allowed by the model (e.g., where the prior or likelihood is zero), the engines will throw errors (e.g., JAGS will call them "invalid parent values"). If no initial values are given, both the engine and Trinity will proceed without error, *but in some engines all MCMC chains will have the same starting point*, rendering any convergence statistics invalid. It is always prudent to provide at least some initial values. Initial values can be scalars, vectors, or matrices, as needed.

Once all of these variables are prepared, they can be handed off to the main function of Trinity, `callbayes`. This function can take a large number of input fields to control the behavior of the engine, which can be WinBUGS, JAGS, or Stan (WinBUGS is currently limited to Windows operating systems, and Stan is limited to Unix-based systems). To select the computational engine, set `engine` to `'bugs'`, `'jags'`, or `'stan'`. (Note that if Stan is selected, the model code should be changed to the Stan code provided in the next section.) More detail regarding the use of `callbayes` can be found in its help documentation (`doc callbayes`). These default inputs are generally sufficient:

```
[stats, chains, diagnostics, info] = callbayes(engine, ...
    'model'          ,     model , ...   % the model as a cell
    'data'           ,      data , ...   % the data as a struct
    'outputname'     , 'samples' , ...   % any character string
    'init'           , generator , ...   % an anonymous function
    'datafilename'   ,   proj_id , ...   % any character string
    'initfilename'   ,   proj_id , ...   % any character string
    'scriptfilename' ,   proj_id , ...   % any character string
    'logfilename'    ,   proj_id , ...   % any character string
    'nchains'        ,        3  , ...   % the number of chains
    'nburnin'        ,     1000  , ...   % the burnin period
    'nsamples'       ,    10000  , ...   % how many saved samples?
    'monitorparams'  ,    params , ...   % the cell string
    'thin'           ,        1  , ...   % the thinning factor
    'workingdir'     ,    ['/tmp/' proj_id]  , ...   % a temp dir
    'verbosity'      ,        0  , ...   % higher is more verbose
    'saveoutput'     ,     true  , ...   % save JAGS log file?
    'parallel'       ,    false  , ...   % use multiple cores?
    'modules'        ,   {'dic'}  );     % use extra modules?
```

and operating system. The first input selects the engine. The various `'*filename'` inputs on lines 6–9 serve to organize the temporary files in a readable fashion, so that the user can easily access them for debugging or reproduction purposes.[4]

The input values on lines 10–14 determine how many independent chains should be run, how many samples should be used for burn-in, how many samples should be saved per chain, which parameters should be saved, and by how much the chains should be thinned (n means every n^{th} sample is saved). Line 15 determines a working directory, which is currently set to a value that will work well on UNIX systems; Windows users might want to change this. Line 16 determines how much output Trinity gives while it is running. Line 17 decides whether the text output given by the engine should be saved.

Line 18 determines if parallel processing should be used—if this is set to `true`, all the chains requested on line 10 will be started simultaneously.[5] Note that for complex models, this may cause computers to become overburdened as all the processing power is used up by Trinity. Users who want to run multiple chains than they have computing cores available can use the optional input pair `'numcores', C, ...`, where `C` is the maximum number of cores Trinity is allowed to use. Finally, line 19 lists optional extra modules (JAGS only). By default, the `dic` module is called because this facilitates tracking of the model deviance as a variable. Users with programming experience can create their own modules for inclusion here (e.g., `'wiener'`; see Wabersich & Vandekerckhove, 2014).

A successful `callbayes` call will yield up to four output arguments. `stats` contains summary statistics for each saved parameter (mean, median, standard deviation, and the mass of the posterior below 0). These can be used for easy access to parameter estimates. `chains` contains all the posterior samples saved. The usefulness of this is discussed below. `diagnostics` provides quick access to the convergence metric $\hat{R}$ and the number of effective samples (Gelman & Rubin, 1999). `info` gives some more information, in particular the model variable and a list of all the options that were set for the analysis (combining the user-provided and automatically generated settings).

The most important output variable is `chains`, which contains the saved posterior samples that are the immediate goal of the MCMC procedure. This variable is used by practically all functions in Trinity that do post-processing, summary, and visualization. The default Trinity script contains the line `grtable(chains, 1.05)`. The `grtable` function prints a table with a quick overview of the sampling results, such as the posterior mean, the number of samples drawn, the number of effective samples (`n_eff`) and the $\hat{R}$ convergence metric. The second input to `grtable` can be either a number, in which case only parameters which an $\hat{R}$ larger than that number will be printed (or a message that no such parameters exist); or it can be a string with a *regular expression*, in which case only parameters fitting that pattern will be shown.[6]

Another useful function that relies on the `chains` variable and on regular expressions is `codatable`, which prints a table with user-selected statistics for selected parameters. For example, to see the posterior mean and standard deviation of the `beta` parameters:

```
>> codatable(chains, 'beta', @mean, @std)
  Estimand          mean           std
    beta_1        -6.937         5.624
    beta_2         1.139        0.1554
```

Finally, Trinity contains a set of functions for visualizing MCMC chains and posterior distributions, but for the present application, a simple scatter plot and regression line suffice (Fig. 3):

```
scatter(x, y)
line(xlim, stats.mean.beta_1 + stats.
mean.beta_2 * xlim)
```

Note that the posterior distributions of the regression parameters contain the first bisector ($\beta_1 \approx 0$, $\beta_2 \approx 1$).

Working from R

R (R Development Core Team, 2004) is a free statistical software package that can be downloaded from http://

[4]When using JAGS or Stan, the working directory will contain a file with a cryptic name that starts with `tp` and ends in a sequence of random characters, with no file extension. This is the entry point script that Trinity uses to call the engine. It can be used to reproduce the analysis outside of MATLAB, if desired—the files in that directory that do not have the `.txt` extension are all that is needed for reproduction. The `*.txt` files are output, containing the posterior samples and the log file. When using WinBUGS, data files, initial values files, and model files will be available in the working directory where they can be accessed with the WinBUGS GUI.

[5]On UNIX systems, this requires the installation of the free program GNU parallel (Tange, 2011). On Windows systems, it currently requires the MATLAB Parallel Computing Toolbox, but we are working to resolve this dependency.

[6]Regular expressions are an extremely powerful and flexible programming constructs. To give some examples: if the expression is `'beta'`, all parameters with the string `beta` in their name will be shown. If it is `'^beta'`, only parameters starting with that string will be shown. `'beta$'` will show only those ending in that string. `'.'` will match any variable, and `'be|ta'` will match anything containing `be` *or* `ta`. A complete overview to regular expressions in MATLAB can be found via the documentation for the function `regexp`.

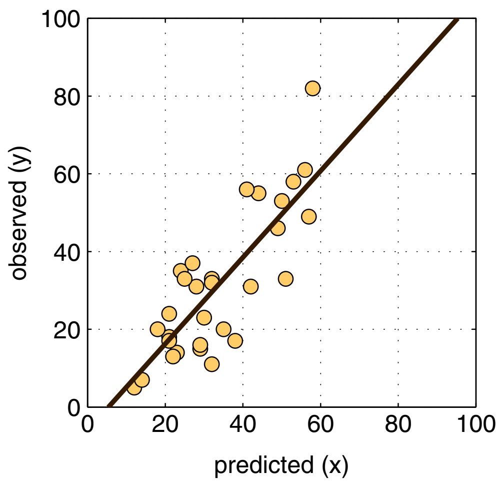


Fig. 3 Results of the linear regression example. The best fitting regression line is very close to the first bisector $y = x$

www.r-project.org/. In this section, we outline how users can interact with WinBUGS, JAGS, and Stan using R. As with MATLAB, using R to run analyses increases flexibility compared to working with these Bayesian engines directly; users can use R to format the data, generate the initial values, and visualize and save the results of the sampling run using simple R commands.

Interacting with WinBUGS: R2WinBUGS

To interact with WinBUGS, users have to install the R2WinBUGS package (Sturtz et al., 2005). The R2WinBUGS package allows users to call WinBUGS from within R and pass on the model specification, the data, and the initial values to WinBUGS using the bugs() function. WinBUGS then samples from the posterior distribution of the parameters and returns the MCMC samples to R.

The following R code can be used to sample from the posterior distribution of the model parameters in the linear regression example using WinBUGS.

```
# set working directory
setwd("C:/Dropbox/My
Documents/Bayesian_estimation/ WinBUGS")
# load R2WinBUGS package
library (R2WinBUGS)
```

The setwd() function specifies the working directory where R will look for the model file and will save the results. The library() function loads the R2WinBUGS package.

```
# create vector that contains the expected number of attendees
x <- c(51, 44, 57, 41, 53, 56, 49, 58, 50, 32,
       24, 21, 23, 28, 22, 30, 29, 35, 18, 25,
       32, 42, 27, 38, 32, 21, 21, 12, 29, 14)
# create vector that contains the observed number of attendees
y <- c(33, 55, 49, 56, 58, 61, 46, 82, 53, 33,
       35, 18, 14, 31, 13, 23, 15, 20, 20, 33,
       32, 31, 37, 17, 11, 24, 17,  5, 16,  7)
# create a scalar that contains the number of sessions
N <- 30
# create a list that contains the data and will be passed on to WinBUGS
mydata <- list("y", "x", "N")
```

Here we create a list named mydata that contains the data (i.e., x, y, and N) and will be passed on to WinBUGS.

```
# create the initial values for the
unobserved stochastic nodes
myinits=function(){
  list(beta=rnorm(2, 0, 10), tau=runif
(1, 0, 5))
}
```

Here we create the initial values for the unobserved stochastic nodes. The initial values for beta[1] and beta[2] are random deviates from a zero-centered normal distribution with a standard deviation of 10.0 generated using the rnorm() function. The initial values for tau are generated from a uniform distribution with lower bound of 0 and upper bound of 5 using the runif() function. The code generates a unique set of initial values for each chain.

```
# specify parameters of interest
myparameters <- c("beta", "tau")
```

Here we create a vector that contains the names of the model parameters that we want to draw inference about.

```
# call WinBUGS
samples <- bugs(data=mydata, inits=myinits, parameters=myparameters,
    model.file="linreg_model.txt", n.chains=3, n.iter=1000, n.burnin=500,
    n.thin=1, DIC=FALSE, bugs.directory="C:/WinBUGS14", codaPkg=FALSE,
    debug=FALSE)
```

The `bugs()` function calls WinBUGS and passes on the model specification, the data, and the start values using the following arguments:

- `data` specifies the list object that contains the data.
- `inits` specifies the list object that contains the initial values.
- `parameters` specifies the vector that lists the names of the parameters of interest.
- `model.file` specifies the text file that contains the model specification. The `model.file` argument can also refer to an R function that contains the model specification that is written to a temporary file.
- `n.chain` specifies the number of MCMC chains.
- `n.iter` specifies the total number of samples per chain.
- `n.burnin` specifies the number of samples per chain that will be discarded at the beginning of the sampling run.
- `n.thin` specifies the degree of thinning.
- `DIC` specifies whether WinBUGS should return the Deviance Information Criterion (DIC; Spiegelhalter et la., 2002) measure of model comparison.
- `bugs.directory` specifies the location of `WinBUGS14.exe`.
- `codaPkg` specifies the output that is returned from WinBUGS. Here `codaPkg` is set to `FALSE` to ensure that WinBUGS returns the posterior samples in the `samples` object. If `codaPkg` is set to `TRUE`, WinBUGS returns the paths to a set of files that contains the WinBUGS output.
- `debug` specifies whether WinBUGS will be automatically shut down after sampling. Here `debug` is set to `FALSE` to ensure that WinBUGS shuts down immediately after sampling and returns the results to R. If `debug` is set to `TRUE`, WinBUGS will not shut down after sampling and will display summary statistics and trace plots of the monitored parameters. As the name suggests, setting `debug` to `TRUE` can also provide—often cryptic—cues for debugging purposes.

For more details on the use of `bugs()`, the reader is referred to the help documentation.

Once WinBUGS has finished sampling, it returns the posterior samples to R in the `samples` object. The results of the sampling run can be accessed, visualized, and summarized using, for instance, the following code:

```
# display the first 15 samples in the
first chain for tau
samples$sims.array[1:15,1,"tau"]
# plot a histogram of the posterior
distribution of tau
hist(samples$sims.array[,,"tau"])
# display summary statistics of the
posterior distributions
print(samples)
```

The posterior samples for `beta[1]`, `beta[2]`, and `tau` are stored in `samples$sims.array` (or `samples$sims.list`). The `hist()` function can be used to plot histograms of the posterior distribution of the parameters based on the samples values. The `print(samples)` command displays a useful summary of the posterior distribution of each model parameter, including the mean, the standard deviation, and the quantiles of the posteriors, and (if multiple chains are run) the $\hat{R}$ convergence metric.

Interacting with JAGS: R2jags

To interact with JAGS, users have to install the `R2jags` package (Su & Yajima, 2012). The `R2jags` package allows users to call JAGS from within R and pass on the model specification, the data, and the start values to JAGS using the `jags()` function. JAGS then samples from the posterior distribution of the parameters and returns the MCMC samples to R.

The R code for running the MCMC routine for the linear regression example in JAGS is similar to the R code for running the WinBUGS analysis outlined in the previous section, with the following modifications. Instead of loading the `R2WinBUGS` package, load the `R2jags` package by typing:

```
# load R2jags
library(R2jags)
```

Once the `mydata`, `myinits`, and `myparameters` objects are created in R, use the `jags()` function to call

JAGS and sample from the posterior distribution of the parameters:

```
# call JAGS
samples <- jags(data=mydata, inits=myinits,
    parameters.to.save=myparameters, model.file="linreg_model.txt",
    n.chains=3, n.iter=1000, n.burnin=500, n.thin=1, DIC=FALSE)
```

The `jags()` function takes as input the following arguments:

- `data` specifies the list object that contains the data.
- `inits` specifies the list object that contains the initial values.
- `parameters.to.save` specifies the vector that lists the names of the parameters of interest.
- `model.file` specifies the file that contains the model specification. The `model.file` argument can also refer to an R function that contains the model specification that is written to a temporary file.
- `n.chains` specifies the number of MCMC chains.
- `n.iter` specifies the total number of samples per chain.
- `n.burnin` specifies the number of samples per chain that will be discarded at the beginning of the sampling run.
- `n.thin` specifies the degree of thinning.
- `DIC` specifies whether JAGS should return the DIC.

For more details on the use of `jags()`, the reader is referred to the help documentation.

Once JAGS has finished sampling, it returns the posterior samples to R in the `samples` object. The results of the sampling run can be accessed, visualized, and summarized using, for instance, the following code:

```
# display the first 15 samples in the first chain for tau
samples$BUGSoutput$sims.array[1:15,1,"tau"]
# plot a histogram of the posterior distribution of tau
hist(samples$BUGSoutput$sims.array[,,"tau"])
# display summary statistics of the posterior distributions
print(samples)
# plot traceplot; press ENTER for page change
traceplot(samples)
```

The posterior samples for `beta[1]`, `beta[2]`, and `tau` are stored in `samples$BUGSoutput$sims.array` (or `samples$BUGSoutput$sims.list`), and can be visualized and summarized using the `hist()` and `print()` functions, respectively. As the name suggests, the `traceplot(samples)` command displays trace plots of the model parameters, which provide useful visual aids for convergence diagnostics.

Interacting with Stan: rstan

To interface R to Stan, users need to install the `rstan` package (Guo et al., 2015). The `rstan` package allows users to call Stan from within R and pass the model specification, data, and starting values to Stan using the `stan()` function. The MCMC samples from the posterior distribution generated by Stan are then returned and can be further processed in R.

There are a few differences between WinBUGS/JAGS and Stan that are worth noting when specifying Stan models. While JAGS and WinBUGS simply interpret the commands given in the model, Stan compiles the model specification to a C++ program. Consequently, Stan differentiates between a number of different variable types, and variables in a model need to be declared before they can be manipulated. Moreover, model code in Stan is split into a number of blocks, such as *"data"* and *"model"*, each of which serves a specific purpose. Finally, unlike in WinBUGS and JAGS, the order of statements in a Stan model matters and statements cannot be interchanged with complete liberty.

To run the R code for the linear regression example in Stan, begin by loading the `rstan` package:

```
# load rstan package
library(rstan)
```

The `mydata`, `myinits`, and `myparameters` are created in R as illustrated before. However, as Stan relies on a somewhat different syntax than WinBUGS and JAGS, we need to rewrite the model file so it can be parsed by Stan.

Here we chose to specify the Stan model as a vector string in R and pass it directly to Stan's sampling function. Note, however, that we could get the same result by simply saving the code as, say, `linreg_model.stan`.

```
# specify Stan model as a string vector
linreg_model <- "data{
    int<lower=1> N;
    vector[N]    x;
    vector[N]    y;
}
parameters{
    vector[2]     beta;
    real<lower=0> sigma2;
}
transformed parameters{
    real<lower=0>  tau;
    tau <- pow(sigma2, -1);

    real<lower=0>  sigma;
    sigma <- pow(sigma2, 0.5);
}
model{
    // prior definitions
    beta[1] ~ normal(0, sqrt(1000));                          // Eq. 4
    beta[2] ~ normal(0, sqrt(1000));                          // Eq. 5
    // inverse gamma prior for the variance
    sigma2 ~ inv_gamma(0.001, 0.001)                          // Eq. 6
    // linear regression
    for(i in 1:N){                                            // Eq. 3
        y[i] ~ normal(beta[1] + beta[2] * x[i], sigma);       // Eqs. 1-2
    }
}"
```

There are a number of very obvious ways in which this model specification differs from that in WinBUGS and JAGS. The model code is split into four blocks and all variables that are mentioned in the "model" block are defined in the preceding blocks. The "data" block contains the definition of all observed data that are provided by the user. The "parameters" block contains the definition of all stochastic variables, and the "transformed parameters" block contains the definition of all transformations of the stochastic variables. The difference between these latter two parts of the code is rather subtle and has to do with the number of times each variable is evaluated during the MCMC sampling process; a more elaborate explanation can be found in the Stan reference manual (Stan Development Team, 2015).

We will not discuss the specifics of all the variable definitions here (see Stan Development Team (2015), for details) but will rather illustrate a few important points using as example the `tau` variable. As in the model specification for WinBUGS and JAGS, `tau` is the precision of the Gaussian distribution. Defining a variable for the precision of the Gaussian is, strictly speaking, not necessary because distribution functions in Stan are parameterized in terms of their standard deviation. Nevertheless, we retain `tau` for easy comparability of the Stan MCMC samples with the output of WinBUGS or JAGS. The first line of the definition of `tau` states that it is a real number that is not smaller than 0, and Stan will return an error message should it encounter a negative value for `tau` during the sampling process. The next line states that `tau` is the inverse of the variance of the Gaussian. If we were to reverse the order of these last two lines, due to Stan's line-by-line evaluation of the code, we would get an error message stating that the variable `tau` is not defined.

The specification of the actual sampling statements in the "model" block begins, in line with Stan's line-by-line evaluation style, with the prior distributions for the regression coefficients beta[1] and beta[2] and the variance of the Gaussian. Note that the prior for sigma2 is an inverse gamma distribution—this is equivalent to the prior specification in the WinBUGS/JAGS model where the inverse of the variance was given a gamma prior. Finally, we summarized (1) and (2) into a single line, which is another way in which the Stan model specification differs from the WinBUGS/JAGS code. While WinBUGS does not allow users to nest statements within the definition of a stochastic node, Stan (and also JAGS) users can directly specify the mean of the Gaussian to be a function of the regression coefficients and observed data x, without needing to define mu[i].

To sample from the posterior distribution of the parameters, call the stan() function:

```
# call Stan
samples <- stan(data = mydata, init =
myinits, pars = myparameters,
    model_code = linreg_model,
    chains = 3, iter = 1000,
    warmup = 500, thin = 1)
```

The stan() function takes as input the following arguments:

- data specifies the list object that contains the data.
- init specifies the list object that contains the initial values.
- pars specifies the vector that lists the names of the parameters of interest.
- model_code specifies the string vector that contains the model specification. Alternatively, the name of a .stan file that contains the model specification can be passed to Stan using the file argument.
- chains specifies the number of MCMC chains.
- iter specifies the total number of samples per chain.
- warmup specifies the number of samples per chain that will be discarded at the beginning of the sampling run.
- thin specifies the degree of thinning.

For more details on the use of stan(), we refer readers to the corresponding R help file.

Once sampling is finished, Stan returns the posterior samples to R in the samples object. The results of the sampling run can be accessed, visualized, and summarized using the following code:

```
# display the first 15 samples for tau
extract(samples, pars="tau",
inc_warmup=F)$tau[1:15]
```

The posterior samples in the samples object can most easily be accessed using the extract() function, which takes as input arguments:

- samples object containing the posterior samples from Stan.
- pars character vector with the names of the parameters for which the posterior samples should be accessed.
- inc_warmup logical value indicating whether warm-up samples should be extracted too.

```
# plot a histogram of the posterior
distribution of tau
hist(extract(samples, pars="tau")$tau)
# display summary statistics of the
posterior distributions
print(samples)
# plot traceplot; press ENTER for page
change traceplot(samples)
```

The posterior samples for beta[1], beta[2], and tau can be visualized and summarized using the hist() and print() functions, respectively. As the name suggests, the traceplot(samples) command displays trace plots of the model parameters, which provide useful visual aids for convergence diagnostics.

Example: Multinomial processing tree for modeling false-memory data

In this section, we illustrate the use of WinBUGS, JAGS, and Stan for Bayesian parameter estimation in the context of multinomial processing trees, popular cognitive models for the analysis of categorical data. As an example, we will use data reported in Wagenaar and Boer (1987). The data result from an experiment in which misleading information was given to participants who were asked to recall details of a studied event. The data were previously revisited by Vandekerckhove et al. (2015), and our discussion of Wagenaar and Boer (1987)'s experiment and their three possible models of the effect of misleading postevent information on memory closely follows that of Vandekerckhove et al. (2015).

The experiment proceeded in four phases. Participants were first shown a sequence of drawings involving a pedestrian-car collision. In one particular drawing, a car was shown at an intersection where a traffic light was either red, yellow, or green. In the second phase, participants were asked questions about the narrative, such as whether they remembered a pedestrian crossing the road as the car approached the "traffic light" (in the consistent-information condition), the "stop sign" (in the inconsistent-information condition) or the "intersection" (the neutral group). In the third phase, participants were given a recognition test. They were shown pairs of pictures from phase I, where one of the pair had been slightly altered (e.g., the traffic light had been replaced by a stop sign), and asked to pick out the unaltered version. In the final phase, participants were informed that there had indeed been a traffic light, and were then asked to recall the color of the light.

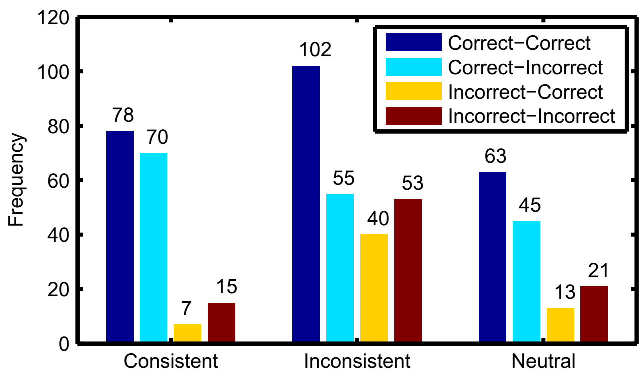


Fig. 4 The data from the Wagenaar and Boer (1987) experiment. Correct–Correct: Both phase III and phase IV answers are correct; Correct–Incorrect: phase III answer is correct but phase IV answer is incorrect; Incorrect–Correct: phase III answer is incorrect but phase IV answer is correct; Incorrect–Incorrect: Both phase III and phase IV answers are incorrect. The data are grouped by condition

The data consist of the frequency with which participants' responses fall into each of the four response categories, where each response category is characterized by a distinct response pattern: both phase III and phase IV answers are correct (Correct–Correct), phase III answer is correct but phase IV answer is incorrect (Correct–Incorrect), phase III answer is incorrect but phase IV answer is correct (Incorrect–Correct), and both phase III and phase IV answers are incorrect (Incorrect–Incorrect). The data from the Wagenaar and Boer (1987) experiment are shown in Fig. 4; the figure shows the frequency of participants in each of the four response categories in the consistent, inconsistent, and neutral conditions.

The first theoretical account on the effect of misleading postevent information is Loftus' *destructive–updating* model. This model predicts that when conflicting information is presented, it replaces and destroys the original information. Second is the *coexistence* model, under which the initial memory is suppressed by an inhibition mechanism. However, the suppression is temporary and can revert. The third model is the *no–conflict* model, under which misleading postevent information cannot replace or suppress existing information, so that it only has an effect if the original information is somehow missing (i.e., was not encoded or is forgotten).

Multinomial processing tree models

The three theoretical accounts can be cast as *multinomial processing tree models* (MPT), which translate a decision tree like the one in Fig. 5 into a multinomial distribution (Batchelder & Riefer, 1980; Chechile, 1973; Riefer & Batchelder, 1988). Figure 5 shows the tree associated with the no-conflict model. In phase I of the experiment, the presence of the traffic light is correctly stored with probability p. If this phase is successful, the color is encoded next, with success probability c. In phase II, the false presence of the stop sign is stored with probability q. In phase III, the answer is either known or guessed correctly with probability $1/2$, and in phase IV the answer is either known or guessed correctly with probability $1/3$.

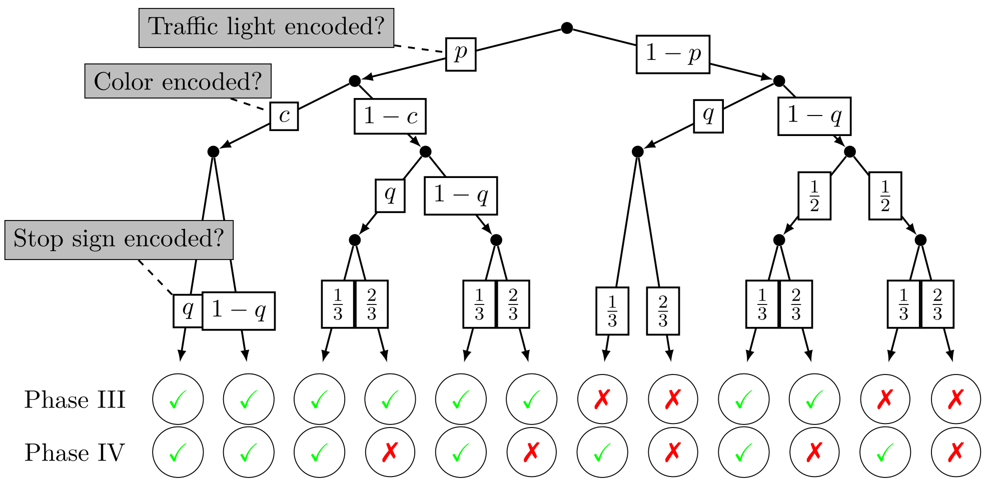


Fig. 5 Multinomial processing tree representation of the inconsistent condition according to the no-conflict model (adapted from Wagenaar & Boer, 1987). The probability of each of the four response patterns (i.e., correct vs. error in phase III and correct vs. error in phase IV) is given by adding the probabilities of each branch leading to that data response pattern. The probability of each branch is given by the product of the individual probabilities encountered on the path

To calculate the probability of the four possible response patterns (i.e., correct vs. error in phase III and correct vs. error in phase IV), we add together the probabilities of each branch leading to that response pattern. The probability of each branch being traversed is given by the product of the individual probabilities encountered on the path. For example, under the no-conflict model, the probability (and hence, expected proportion) of getting phase III correct but phase IV wrong is (adding the paths in Fig. 5 from left to right and starting at the bottom from those cases where phase III was correct but phase IV was not): $\frac{2}{3} \times q \times (1-c) \times p + \frac{2}{3} \times (1-q) \times (1-c) \times p + \frac{2}{3} \times \frac{1}{2} \times (1-q) \times (1-p)$.

The two competing models both add one parameter to the no-conflict model. In the case of the destructive-updating model, we add one parameter d for the *probability that the traffic light information is destroyed upon encoding the stop sign*. In the case of the coexistence model, we instead add one parameter s for the *probability that the stop sign encoding causes the traffic light information to be suppressed, not destroyed*, so that it remains available in phase IV.

Here we focus on the no-conflict model, but implementing the other models would involve only small changes to our code. The generative specification of the no-conflict model for the consistent (cons), inconsistent (inco) and neutral (neut) conditions is as follows:

$$\text{cons} \sim \mathcal{M}(\theta_{(1,\cdot)}, N_1) \tag{7}$$
$$\text{inco} \sim \mathcal{M}(\theta_{(2,\cdot)}, N_2) \tag{8}$$
$$\text{neut} \sim \mathcal{M}(\theta_{(3,\cdot)}, N_3), \tag{9}$$

where $\mathcal{M}$ denotes that the data follow a multinomial distribution and N refers to the number of participants in the n^{th}, $n = 1, 2, 3$, condition. The 3×4 matrix θ contains the category probabilities of the multinomial distributions in the three conditions, where $\theta_{(n,\cdot)}$ refers to the n^{th} row of θ. For each condition, the four category probabilities are expressed in terms of the three model parameters p, q, and c. As shown in Fig. 5, the category probabilities map onto the four response categories and the corresponding response patterns, and are obtained by following the paths in the tree representation of the model. In particular, the category probabilities in the three conditions are given by:

$$\theta_{(1,1)} = (1 + p + q - pq + 4pc)/6 \tag{10}$$
$$\theta_{(1,2)} = (1 + p + q - pq - 2pc)/3 \tag{11}$$
$$\theta_{(1,3)} = (1 - p - q + pq)/6 \tag{12}$$
$$\theta_{(1,4)} = (1 - p - q + pq)/3 \tag{13}$$
$$\theta_{(2,1)} = (1 + p - q + pq + 4pc)/6 \tag{14}$$
$$\theta_{(2,2)} = (1 + p - q + pq - 2pc)/3 \tag{15}$$
$$\theta_{(2,3)} = (1 - p + q - pq)/6 \tag{16}$$
$$\theta_{(2,4)} = (1 - p + q - pq)/3 \tag{17}$$
$$\theta_{(3,1)} = (1 + p + 4pc)/6 \tag{18}$$
$$\theta_{(3,2)} = (1 + p - 2pc)/3 \tag{19}$$
$$\theta_{(3,3)} = (1 - p)/6 \tag{20}$$
$$\theta_{(3,4)} = (1 - p)/3 \tag{21}$$

Finally, our priors are flat beta distributions $\mathcal{B}(1, 1)$; these distributions imply equal prior probability for all values between 0 and 1 (i.e., $\mathcal{B}(1, 1)$ is the same as a standard uniform distribution):

$$p \sim \mathcal{B}(1, 1) \tag{22}$$
$$q \sim \mathcal{B}(1, 1) \tag{23}$$
$$c \sim \mathcal{B}(1, 1) \tag{24}$$

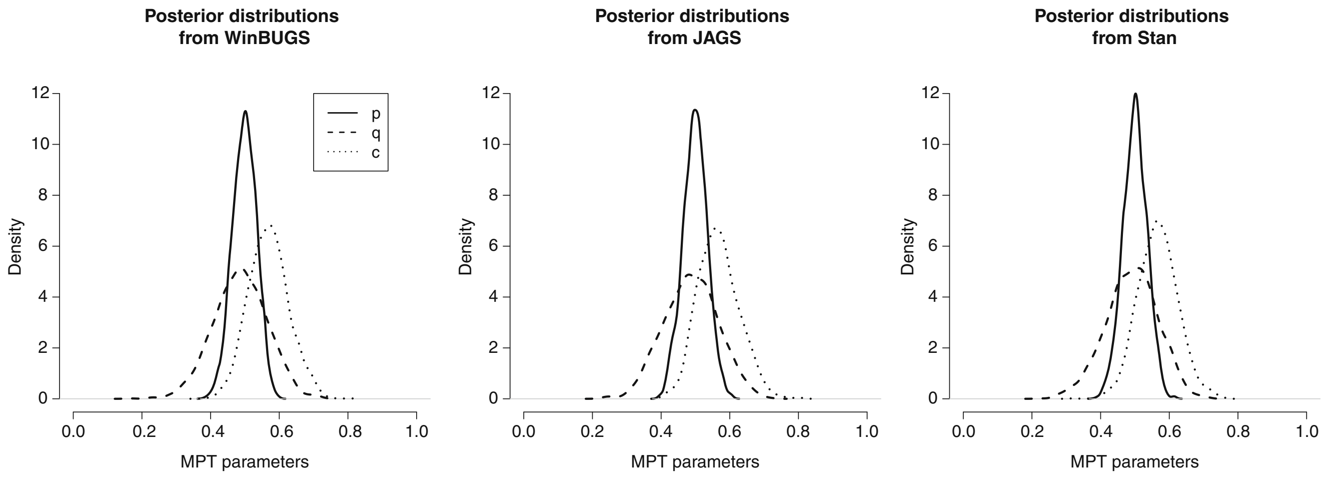


Fig. 6 The posterior distribution of the parameters of the no-conflict MPT model obtained from WinBUGS, JAGS, and Stan in combination with R. The *solid*, *dashed*, and *dotted lines* show the posterior distribution of the p, q, and c parameters, respectively

We will now fit the no-conflict model to the Wagenaar and Boer (1987) data using WinBUGS, JAGS and Stan in combination with both MATLAB and R. Obtaining parameter estimates for the destructive–updating and the coexistence models requires only minor modifications to the code. In particular, we would have to modify the category probabilities (10–21) to reflect the tree architecture of the alternative models and define an additional parameter (i.e., parameter *d* for the destructive-updating and parameter *s* for the coexistence model) with the corresponding uniform prior distribution. As an illustration, the Supplemental Material presents the WinBUGS, JAGS and Stan model files and the corresponding R code that allows users to estimate the parameters of the no-conflict as well as the destructive-updating and coexistence models.

Working from R using R2WinBUGS

The WinBUGS code for the generative specification of the no-conflict model is given below. Note here that since the generative model specification is just a list of declarative statements, the order of statements does not matter for the specification. We write the statements here in the order in which they appear in the text. This intentionally violates the usual "programmer logic" in which variables need to be declared before they are used. We emphasize that such restriction is not needed in WinBUGS code.

```
model {
  # ---- Data --------------------------------------------- #
  cons[1:4] ~ dmulti(theta[1,1:4], N[1])          #  Eq. 7
  inco[1:4] ~ dmulti(theta[2,1:4], N[2])          #  Eq. 8
  neut[1:4] ~ dmulti(theta[3,1:4], N[3])          #  Eq. 9

  # ---- Consistent condition ----------------------------- #
  theta[1,1] <- ( 1 + p + q - pq + 4 * pc ) / 6      # Eq. 10
  theta[1,2] <- ( 1 + p + q - pq - 2 * pc ) / 3      # Eq. 11
  theta[1,3] <- ( 1 - p - q + pq ) / 6               # Eq. 12
  theta[1,4] <- ( 1 - p - q + pq ) / 3               # Eq. 13

  # ---- Inconsistent condition --------------------------- #
  theta[2,1] <- ( 1 + p - q + pq + 4 * pc ) / 6      # Eq. 14
  theta[2,2] <- ( 1 + p - q + pq - 2 * pc ) / 3      # Eq. 15
  theta[2,3] <- ( 1 - p + q - pq ) / 6               # Eq. 16
  theta[2,4] <- ( 1 - p + q - pq ) / 3               # Eq. 17

  # ---- Neutral condition-------------------------------- #
  theta[3,1] <- ( 1 + p + 4 * pc ) / 6               # Eq. 18
  theta[3,2] <- ( 1 + p - 2 * pc ) / 3               # Eq. 19
  theta[3,3] <- ( 1 - p ) / 6                        # Eq. 20
  theta[3,4] <- ( 1 - p ) / 3                        # Eq. 21

  # ---- Priors ------------------------------------------- #
  p ~ dbeta(1,1)                                     # Eq. 22
  q ~ dbeta(1,1)                                     # Eq. 23
  c ~ dbeta(1,1)                                     # Eq. 24

  # ---- Some useful transformations ---------------------- #
  pq  <- p * q
  pc  <- p * c
}
```

Once the model specification is saved to a text file (e.g., noconflict.txt), the following R code can be used to create the data and the initial values, and call WinBUGS using the R2WinBUGS package:

```
# load R2WinBUGS package
library(R2WinBUGS)

# create the data
cons <- c( 78, 70,  7, 15)
inco <- c(102, 55, 40, 53)
neut <- c( 63, 45, 13, 21)
N <- c(170, 250, 142)
mydata <- list("cons", "inco", "neut", "N")

# create the initial values
myinits <- function(){
    list(p=runif(1), q=runif(1), c=runif(1))
}

# specify the parameters of interest
myparameters <- c("p", "q", "c")

# call WinBUGS
samples <- bugs(data=mydata, inits=myinits, parameters=myparameters,
    model.file="noconflict.txt",
    n.chains=3, n.iter=3500, n.burnin=500, n.thin=5,
    DIC=FALSE, bugs.directory="C:/WinBUGS14",
    codaPkg=FALSE, debug=TRUE)
```

Note that we ran 3500 iterations per chain (n.iter=3500) and retained only every 5th sample (n.thin=5). As the parameters in cognitive models are often strongly correlated, it is typically necessary to run relatively long MCMC chains and thin the chains to reduce auto-correlation. When the sampling run has finished, WinBUGS returns the posterior samples for the three model parameters in the samples object. The posterior distribution of the parameters—plotted using the sampled values—is shown in the first column of Fig. 6.

Working from R using R2jags

The JAGS code for the generative specification of the no-conflict model is identical to the WinBUGS code presented in the previous section, and so is the R code for creating the data and generating the initial values. Once the R2jags package is loaded by typing library(R2jags), the following R code can be used to call JAGS and sample from the posterior distribution of the parameters:

```
samples <- jags(data=mydata,inits=myinits,parameters.to.save=myparameters,
    model.file ="noconflict.txt",
    n.chains=3, n.iter=3500, n.burnin=500, n.thin=5, DIC=FALSE)
```

JAGS returns the posterior samples for the three model parameters in the `samples` object. The posterior distribution of the parameters is shown in the second column of Fig. 6. The posteriors obtained with JAGS are essentially indistinguishable from the ones obtained with WinBUGS.

Working from R using rstan

The Stan code for the non-conflict model again differs somewhat from the WinBUGS/JAGS code:

```
data{
    int cons[4];
    int inco[4];
    int neut[4];
}
parameters{
    real<lower=0,upper=1> p;
    real<lower=0,upper=1> q;
    real<lower=0,upper=1> c;
}
transformed parameters{
    real<lower=0,upper=1> pq;
    real<lower=0,upper=1> pc;
    simplex[4] theta1;
    simplex[4] theta2;
    simplex[4] theta3;

    pq <- p * q;
    pc <- p * c;

    // consistent condition
    theta1[1] <- ( 1 + p + q - pq + 4 * pc ) / 6;  // Eq. 10
    theta1[2] <- ( 1 + p + q - pq - 2 * pc ) / 3;  // Eq. 11
    theta1[3] <- ( 1 - p - q + pq ) / 6;           // Eq. 12
    theta1[4] <- ( 1 - p - q + pq ) / 3;           // Eq. 13

    // inconsistent condition
    theta2[1] <- ( 1 + p - q + pq + 4 * pc ) / 6;  // Eq. 14
    theta2[2] <- ( 1 + p - q + pq - 2 * pc ) / 3;  // Eq. 15
    theta2[3] <- ( 1 - p + q - pq ) / 6;           // Eq. 16
    theta2[4] <- ( 1 - p + q - pq ) / 3;           // Eq. 17

    // neutral condition
    theta3[1] <- ( 1 + p + 4 * pc ) / 6;           // Eq. 18
    theta3[2] <- ( 1 + p - 2 * pc ) / 3;           // Eq. 19
    theta3[3] <- ( 1 - p ) / 6;                    // Eq. 20
    theta3[4] <- ( 1 - p ) / 3;                    // Eq. 21
}
model{
    // priors
    p ~ beta(1,1);                                 // Eq. 22
    q ~ beta(1,1);                                 // Eq. 23
    c ~ beta(1,1);                                 // Eq. 24

    cons ~ multinomial(theta1);                    // Eq. 7
    inco ~ multinomial(theta2);                    // Eq. 8
    neut ~ multinomial(theta3);                    // Eq. 9
}
```

Once the model specification is saved as noconflict.stan, the rstan package has been loaded by typing library(rstan), and R objects have been created that contain the data, initial values, and parameters of interest, the following code can be used to obtain samples from the posterior distributions of the parameters:

```
samples <- stan(data = mydata, init = myinits, pars = myparameters,
        file = 'noconflict.stan',
        chains = 3, iter = 3500, warmup = 500, thin = 5)
```

The posterior samples for the three model parameters are returned in the samples object. The third column of Fig. 6 shows estimates of the posterior densities based on the sampled values; the posteriors closely resemble those obtained with WinBUGS and JAGS.

Working from MATLAB using trinity

The code to fit the no-conflict model from MATLAB using Trinity is again very formulaic, and differs very little between the three computational engines. In the bare-bones script automatically generated by trinity new, we first enter the data:

```
cons = [  78 ,  70 ,   7 ,  15 ] ;
inco = [ 102 ,  55 ,  40 ,  53 ] ;
neut = [  63 ,  45 ,  13 ,  21 ] ;
N = [sum(cons) sum(inco) sum(neut)];
```

After the data are entered, the model definition needs to be provided as a cell string. We omit the model specification here because both the WinBUGS/JAGS and Stan versions are fully given in the previous sections.

Next, we list the parameters of interest in a cell variable:

```
parameters = {
    'c' 'p' 'q'
    };
```

and we write a function that generates a structure containing one random value for each parameter in a field:

```
generator = @()struct(...
    'c', rand, ...
    'p', rand, ...
    'q', rand  ...
    );
```

We also enter the data into a structure where we match the names of the fields to the variable names in the model definition:

```
data = struct(...
    'cons', cons, ...
    'inco', inco, ...
    'neut', neut, ...
    'N'   , N     ...
    );
```

After selecting an engine, the callbayes function is called with mostly default settings:

```
%% Run Trinity with the CALLBAYES
() function
tic
[stats, chains, diagnostics, info] =
callbayes(engine, ...
'model'          ,          model , ...
'data'           ,           data , ...
'outputname'     ,      'samples' , ...
'init'           ,      generator , ...
'modelfilename'  ,        proj_id , ...
'datafilename'   ,        proj_id , ...
'initfilename'   ,        proj_id , ...
'scriptfilename' ,        proj_id , ...
'logfilename'    ,        proj_id , ...
'nchains'        ,              4 , ...
'nburnin'        ,            1e4 , ...
'nsamples'       ,            1e4 , ...
'monitorparams'  ,     parameters , ...
'thin'           ,              5 , ...
'refresh'        ,           1000 , ...
'workingdir'     ,[/tmp/ proj_id] , ...
'verbosity'      ,              0 , ...
'saveoutput'     ,           true , ...
'parallel'       ,       isunix() , ...
'modules'        ,        {'dic'} );
```

The engine will return, among others, the chains variable containing posterior samples for all three parameters of interest. We can inspect the results, and we can use the codatable function to give qualitative feedback about the convergence of the MC chains:

```
if any(codatable(chains, @gelmanrubin) > 1.1)
    grtable(chains, 1.1)
    warning('Some chains were not converged!')
else
    disp('Convergence looks good.')
end
```

Finally, we can inspect the posterior means by chain using the stats structure:

```
disp('Posterior means by chain:')
disp(stats.mean)
```

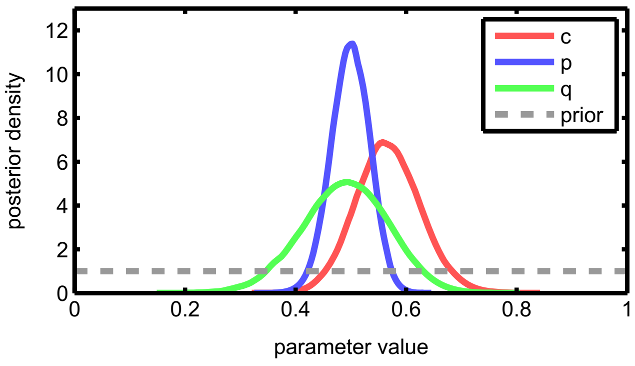


Fig. 7 The posterior distribution of the parameters of the no-conflict MPT model obtained from JAGS in combination with Trinity

as well as check some basic descriptive statistics averaged over all chains:

```
disp('Descriptive statistics for all chains:')
codatable(chains)
```

and visually inspect the posterior distributions using the `smhist` function:

```
smhist(chains, '^c$|^q$|^p$');
```

where the regular expression may be read as "match only variables whose name is exactly `c` or exactly `q` or exactly `p`". The output of the last command—the posterior distribution of the parameters—is shown in Fig. 7.

Testing hypotheses

After the posterior samples have been drawn, and posterior distributions possibly visualized as above, there remains the issue of testing hypotheses relating to parameters. With the current false-memory data set, one hypothesis of interest might be that the probability p of encoding the traffic light is greater (versus lower) than chance (Hypothesis 1). The same question might be asked of the probability c of encoding the light color (Hypothesis 2).

Given samples from the posterior, a convenient way of computing the posterior probability that a hypothesis is true is by computing the proportion of posterior samples in which the hypothesis holds. To test Hypothesis 1, we would calculate the proportion of cases in which $p > 0.5$. To test Hypothesis 2, we calculate the proportion of cases in which $c > 0.5$.

The `codatable` command is useful in this regard. Custom statistics on the posterior samples can be computed by providing anonymous functions as secondary input variables. A quick way of counting the proportion of cases in which a condition is true is to make use of the fact that MATLAB represents logically true statements as 1 and false statements as 0. Hence, the anonymous function `@(x)mean(x>.5)` will return the proportion of cases where the input is greater than 0.5:

```
>> codatable(chains, '^c$|^p$', @mean, @(x)mean(x>.5))
  Estimand          mean  @(x)mean(x>.5)
         c         0.558          0.8441
         p        0.4956          0.4547
```

As it turns out, the probability of Hypothesis 1 given the data is about 84% and that of Hypothesis 2 is about 45%. In other words, neither of the hypotheses is strongly supported by the data. In fact, as Fig. 7 shows, most of the posterior mass is clustered near 0.5 for all parameters.

Conclusions

Bayesian methods are rapidly rising from obscurity and into the mainstream of psychological science. While Bayesian equivalents of many standard analyses, such as the t test and linear regression, can be conducted in off–the-shelf software such as JASP (Love et al., 2015), custom models will continue to require a flexible programming framework and, unavoidably, some degree of software MacGyverism. To implement specialized models, researchers may write their own MCMC samplers, a process that is time-consuming and labor-intensive, and does not come easy to investigators untrained in computational methods.

Luckily, general-purpose MCMC engines—such as WinBUGS, JAGS, and Stan—provide easy-to-use alternatives to custom MCMC samplers. These software packages hit the sweet spot for most psychologists; they provide a large degree of flexibility at a relatively low time cost.

In this tutorial, we demonstrated the use of three popular Bayesian software packages in conjunction with two scientific programming languages, R and MATLAB. This combination allows researchers to implement custom Bayesian analyses from already familiar environments. As we illustrated, models as common as a linear regression can be easily implemented in this framework, but so can more complex models, such as multinomial processing trees (MPT; Batchelder & Riefer, 1980; Chechile, 1973; Riefer & Batchelder, 1988).

Although the tutorial focused exclusively on non-hierarchical models, the packages may also be used for modeling hierarchical data structures (e.g., Lee 2011). In hierarchical modeling, rather than estimating parameters separately for each unit (e.g., participant), we model the between-unit variability of the parameters with group-level distributions. The group-level distributions are used as priors to "shrink" extreme and poorly constrained estimates to more moderate values. Hierarchical estimation can provide more precise and less variable estimates than non-hierarchical estimation, especially in data sets with relatively few observations per unit (Farrell & Ludwig, 2008; Rouder et al., 2005). Hierarchical modeling is rapidly gaining popularity in psychology, largely by virtue of to the availability of accessible MCMC packages. The WinBUGS, JAGS, and Stan implementation of most hierarchical extensions is very straightforward and often does not require more than a few additional lines of code. For the hierarchical WinBUGS implementation of regression models, the reader is referred to Gelman and Hill (2007). For the hierarchical implementation of custom models, such as multinomial processing trees, signal detection, or various response time models, the reader is referred to Lee and Wagenmakers (2013), Matzke et al. (2015), Matzke and Wagenmakers (2009), Nilsson et al. (2011), Rouder et al. (2008) and Vandekerckhove et al. (2011).

Although the goal of our tutorial was to demonstrate the use of general-purpose MCMC software for Bayesian *parameter estimation*, our MPT-example has also touched on Bayesian hypothesis testing. Various other Bayesian methods are available that rely on MCMC-output to test hypotheses and formally compare the relative predictive performance of competing models. For instance, Wagenmakers et al. (2010) and Wetzels et al. (2009) discuss the use of the Savage-Dickey density ratio, a simple procedure that enables researchers to compute Bayes factors (Jeffreys, 1961; Kass and Raftery, 1995) for nested model comparison using the height of the prior and posterior distributions obtained from WinBUGS. Vandekerckhove et al. (2015) shows how to use posterior distributions obtained from WinBUGS and JAGS to compute Bayes factors for non-nested MPTs using importance sampling. Lodewyckx et al. (2011) outline a WinBUGS implementation of the product-space method, a transdimensional MCMC approach for computing Bayes factors for nested and non-nested models. Most recently, Gronau et al. (2017) provide a tutorial on bridge sampling—a new, potentially very powerful method that is under active development. It is important to note, however, that these methods are almost all quite difficult to use and can be unstable, especially for high-dimensional problems.

Throughout the tutorial, we have advocated WinBUGS, JAGS, and Stan as flexible and user-friendly alternatives to homegrown sampling routines. Although the MCMC samplers implemented in these packages work well for the majority of models used in psychology, they may be inefficient and impractical for some. For instance, models of choice and response times, such as the linear ballistic accumulator (Brown and Heathcote, 2008) or the lognormal race (Rouder et al., 2015), are notoriously difficult to sample from using standard MCMC software. In these cases, custom-made MCMC routines may be the only solution. For examples of custom-made and non-standard MCMC samplers, the reader is referred to Rouder and Lu (2005) and Turner et al. (2013), respectively.

Their general usefulness notwithstanding, the three packages all have their own set of limitations and weaknesses. WinBUGS, as the name suggests, was developed specifically for Windows operating systems. Although it is possible to run WinBUGS under OS X and Linux using emulators such as Darwine and CrossOver or compatibility layers such as Wine, user experience is often jarring. Even under Windows, software installation is a circuitous process and requires users to decode a registration key and an upgrade patch via the GUI. Once installed, users typically find the GUI inflexible and labor-intensive. In interaction with R, user experience is typically more positive. Complaints focus mostly on WinBUGS' cryptic error messages and the limited number of built-in functions and distributions. Although the WinBUGS Development Interface (WBDev; Lunn, 2003) enables users to implement custom-made functions and distributions, it requires experience with Component Pascal and is poorly documented. Matzke et al. (2013) provide WBDev scripts for the truncated-normal and ex-Gaussian distributions; Wetzels et al. (2010) provide an excellent WBDev tutorial for psychologists, including a WBDev script for the shifted-Wald distribution. Importantly, the BUGS Project has shifted development away from WinBUGS; development now focuses on OpenBUGS (http://www.openbugs.net/w/FrontPage).

Stan comes equipped with interfaces to various programming languages, including R, Python and MATLAB, and only requires the installation of the specific interface package, which is easy and straightforward under most common operating systems. In terms of computing time, Stan seems a particularly suitable choice for complex models with many parameters and large posterior sample sizes. This advantage in computing time is due to the fact that Stan compiles the sampling model to a `C++` program before carrying out the sampling process. The downside of this compilation step is that, particularly for small models as used in the present tutorial, compilation of the model might require more time than the sampling process itself, in which case WinBUGS or JAGS seem a more advantageous choice.

Finally, we will highlight two advantages of JAGS over Stan. First, as illustrated in our example code, Stan code requires variable declaration and as a result can be somewhat more complicated than JAGS code. Second, as a consequence of Stan's highly efficient Hamiltonian Monte Carlo sampling algorithm, some model specifications are not allowed—in particular, Stan does not easily allow model specifications that require inference on discrete parameters, which reduces its usefulness if the goal is model selection rather than parameter estimation.

We demonstrated the use of three popular Bayesian software packages that enable researchers to estimate parameters in a broad class of models that are commonly used in psychological research. We focused on WinBUGS, JAGS, and Stan, and showed how they can be interfaced from R and MATLAB. We hope that this tutorial can serve to further lower the threshold to Bayesian modeling for psychological science.

Acknowledgements The authors thank Eric-Jan Wagenmakers for helpful comments during the writing of this article. DM was supported by a Veni grant #451-15-010 from the Netherlands Organization of Scientific Research (NWO). UB was supported by an NWO Research Talent grant #406-12-125. JV was supported by NSF grants #1230118 and #1534472 from the Methods, Measurements, and Statistics panel and John Templeton Foundation grant #48192.

References

Batchelder, W. H., & Riefer, D. M. (1980). Separation of storage and retrieval factors in free recall of clusterable pairs. *Psychological Review*, *87*, 375–397.

Brown, S. D., & Heathcote, A. J. (2008). The simplest complete model of choice reaction time: Linear ballistic accumulation. *Cognitive Psychology*, *57*, 153–178.

Chechile, R. A. (1973). *The relative storage and retrieval losses in short-term memory as a function of the similarity and amount of information processing in the interpolated task (Unpublished doctoral dissertation)*. Pittsburgh: University of Pittsburgh.

Farrell, S., & Ludwig, C. J. H. (2008). Bayesian and maximum likelihood estimation of hierarchical response time models. *Psychonomic Bulletin & Review*, *15*, 1209–1217.

Gelman, A., & Hill, J. (2007). *Data analysis using regression and multilevel/hierarchical models*. Cambridge: Cambridge University Press.

Gelman, A., & Rubin, D. B. (1999). Evaluating and using statistical methods in the social sciences. *Sociological Methods & Research*, *27*, 403–410.

Gronau, Q. F., Sarafoglou, A., Matzke, D., Ly, A., Boehm, U., Marsman, M., & Steingroever, H. (2017). A tutorial on bridge sampling. arXiv:1703.05984

Guo, J., Lee, D., Goodrich, B., de Guzman, J., Niebler, E., Heller, T., & Goodrich, B. (2015). rstan: R interface to stan [Computer software manual]. Retrieved from https://cran.r-project.org/web/packages/rstan/index.html

Jeffreys, H. (1961). *Theory of probability*, 3rd edn. Oxford: Oxford University Press.

Kass, R. E., & Raftery, A. E. (1995). Bayes factors. *Journal of the American Statistical Association*, *90*, 773–795.

Kruschke, J. K. (2010). *Doing Bayesian data analysis: A tutorial introduction with R and BUGS*. Burlington: Academic Press.

Lee, M. D. (2011). How cognitive modeling can benefit from hierarchical Bayesian models. *Journal of Mathematical Psychology*, *55*, 1–7.

Lee, M. D., & Wagenmakers, E.-J. (2013). *Bayesian modeling for cognitive science: A practical course*. Cambridge: Cambridge University Press.

Lodewyckx, T., Kim, W., Tuerlinckx, F., Kuppens, P., Lee, M. D., & Wagenmakers, E.-J. (2011). A tutorial on Bayes factor estimation with the product space method. *Journal of Mathematical Psychology*, *55*, 331–347.

Love, J., Selker, R., Marsman, M., Jamil, T., Dropmann, D., Verhagen, A. J., & Wagenmakers, E.-J. (2015). JASP [computer software]. https://jasp-stats.org/

Lunn, D. J. (2003). WinBUGS development interface (WBDev). *ISBA Bulletin*, *10*, 10–11.

Lunn, D. J., Jackson, C., Best, N., Thomas, A., & Spiegelhalter, D. (2012). *The BUGS book: A practical introduction to Bayesian analysis*. Boca Raton: Chapman & Hall/CRC.

Lunn, D. J., Spiegelhalter, D., Thomas, A., & Best, N. (2009). The BUGS project: Evolution, critique and future directions. *Statistics in Medicine*, *28*, 3049–3067.

Lunn, D. J., Thomas, A., & Best, N. (2000). WinBUGS—a Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, *10*, 325–337.

Matzke, D., Dolan, C. V., Batchelder, W. H., & Wagenmakers, E.-J. (2015). Bayesian estimation of multinomial processing tree models with heterogeneity in participants and items. *Psychometrika*, *80*, 205–235.

Matzke, D., Dolan, C. V., Logan, G. D., Brown, S. D., & Wagenmakers, E.-J. (2013). Bayesian parametric estimation of stop-signal reaction time distributions. *Journal of Experimental Psychology: General*, *142*, 1047–1073.

Matzke, D., & Wagenmakers, E.-J. (2009). Psychological interpretation of the ex-Gaussian and shifted Wald parameters: A diffusion model analysis. *Psychonomic Bulletin & Review*, *16*, 798–817.

Morey, R. D., Rouder, J. N., & Jamil, T. (2015). Package Bayes factorâĂŹ. http://cran.r-project.org/web/packages/BayesFactor/BayesFactor.pdf

Nilsson, H., Rieskamp, J., & Wagenmakers, E.-J. (2011). Hierarchical Bayesian parameter estimation for cumulative prospect theory. *Journal of Mathematical Psychology*, *55*, 84–93.

R Development Core Team (2004). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from http://www.R-project.org (ISBN 3-900051-00-3).

Riefer, D. M., & Batchelder, W. H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, *95*, 318–399.

Rouder, J. N., & Lu, J. (2005). An introduction to Bayesian hierarchical models with an application in the theory of signal detection. *Psychonomic Bulletin & Review*, *12*, 573–604.

Rouder, J. N., Lu, J., Morey, R. D., Sun, D., & Speckman, P. L. (2008). A hierarchical process dissociation model. *Journal of Experimental Psychology: General*, *137*, 370–389.

Rouder, J. N., Lu, J., Speckman, P. L., Sun, D., & Jiang, Y. (2005). A hierarchical model for estimating response time distributions. *Psychonomic Bulletin & Review*, *12*, 195–223.

Rouder, J. N., Province, J. M., Morey, R. D., Gomez, P., & Heathcote, A. (2015). The lognormal race: A cognitive-process model of choice and latency with desirable psychometric properties. *Psychometrika*, *80*, 491–513.

Spiegelhalter, D. J., Best, N. G., Carlin, B. P., & van der Linde, A. (2002). Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society B*, *64*, 583–639.

Spiegelhalter, D. J., Thomas, A., Best, N., & Lunn, D. (2003). *WinBUGS version 1.4 user manual*. Cambridge: Medical Research Council Biostatistics Unit.

Stan Development Team (2015). Stan modeling language: User's guide and reference manual. version 2.7.0 [Computer software manual]. Retrieved from https://github.com/stan-dev/stan/releases/download/v2.7.0/stan-reference-2.7.0.pdf

Sturtz, S., Ligges, U., & Gelman, A. (2005). R2WinBUGS: A package for running Win-BUGS from R. *Journal of Statistical Software*, *12*, 1–16.

Su, Y.-S., & Yajima, M. (2012). R2jags: A package for running JAGS from R [Computer software manual]. Retrieved from http://CRAN.R-project.org/package=R2jags

Tange, O. (2011). Gnu parallel - the command–line power tool. *;login: The USENIX Magazine*, *36*(1), 42–47. Retrieved from http://www.gnu.org/s/parallel

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.

Vandekerckhove, J. (2014). Trinity: A MATLAB interface for Bayesian analysis. http://tinyurl.com/matlab-trinity

Vandekerckhove, J., Matzke, D., & Wagenmakers, E.-J. (2015). Model comparison and the principle of parsimony. In Busemeyer, J. R., Townsend, J. T., Wang, Z. J., & Eidels, A. (Eds.) *Oxford handbook of computational and mathematical psychology*. Retrieved from http://p.cidlab.com/vandekerckhove2014model.pdf (pp. 300–317). Oxford: Oxford University Press.

Vandekerckhove, J., Tuerlinckx, F., & Lee, M. D. (2011). Hierarchical diffusion models for two-choice response times. *Psychological Methods*, *16*, 44–62. Retrieved from http://p.cidlab.com/vandekerckhove2011hierarchical.pdf

Wabersich, D., & Vandekerckhove, J. (2014). Extending JAGS: A tutorial on adding custom distributions to JAGS (with a diffusion model example), (Vol. 46. Retrieved from http://p.cidlab.com/wabersich2014extending.pdf

Wagenaar, W. A., & Boer, J. P. (1987). A Misleading postevent information: Testing parameterized models of integration in memory. *Acta Psychologica*, *66*, 291–306.

Wagenmakers, E.-J., Lodewyckx, T., Kuriyal, H., & Grasman, R. (2010). Bayesian hypothesis testing for psychologists: A tutorial on the Savage–Dickey method. *Cognitive Psychology*, *60*, 158–189.

Wetzels, R., Lee, M. D., & Wagenmakers, E.-J. (2010). Bayesian inference using WBDev: A tutorial for social scientists. *Behavior Research Methods*, *42*, 884–897.

Wetzels, R., Raaijmakers, J. G. W., Jakab, E., & Wagenmakers, E.-J. (2009). How to quantify support for and against the null hypothesis: A flexible WinBUGS implementation of a default Bayesian *t* test. *Psychonomic Bulletin & Review*, *16*, 752–760.