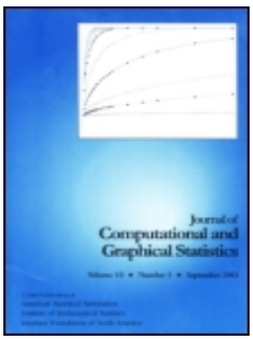

Taylor & Francis
Taylor & Francis Group

Journal of Computational and Graphical Statistics

ISSN: 1061-8600 (Print) 1537-2715 (Online) Journal homepage: http://amstat.tandfonline.com/loi/ucgs20

Programming with models: writing statistical algorithms for general model structures with NIMBLE

Perry de Valpine, Daniel Turek, Christopher J. Paciorek, Clifford Anderson-Bergman, Duncan Temple Lang & Rastislav Bodik

To cite this article: Perry de Valpine, Daniel Turek, Christopher J. Paciorek, Clifford Anderson-Bergman, Duncan Temple Lang & Rastislav Bodik (2016): Programming with models: writing statistical algorithms for general model structures with NIMBLE, Journal of Computational and Graphical Statistics, DOI: 10.1080/10618600.2016.1172487

To link to this article: http://dx.doi.org/10.1080/10618600.2016.1172487

Programming with models: writing statistical algorithms for general model structures with NIMBLE

Perry de Valpine[1], Daniel Turek[1,2], Christopher J. Paciorek[2], Clifford Anderson-Bergman[1,2], Duncan Temple Lang[3], and Rastislav Bodik[4]

[1]University of California, Berkeley, Department of Environmental Science, Policy and Management
[2]University of California, Berkeley, Department of Statistics
[3]University of California, Davis, Department of Statistics
[4]University of California, Berkeley, Department of Electrical Engineering and Computer Science

Abstract

We describe NIMBLE, a system for programming statistical algorithms for general model structures within R. NIMBLE is designed to meet three challenges: flexible model specification, a language for programming algorithms that can use different models, and a balance between high-level programmability and execution efficiency. For model specification, NIMBLE extends the BUGS language and creates model objects, which can manipulate variables, calculate log probability values, generate simulations, and query the relationships among variables. For algorithm programming, NIMBLE provides functions that operate with model objects using two stages of evaluation. The first stage allows specialization of a function to a particular model and/or nodes, such as creating a Metropolis-Hastings sampler for a particular block of nodes. The second stage allows repeated execution of computations using the results of the first stage. To achieve efficient second-stage computation, NIMBLE compiles models and functions via C++, using the Eigen library for linear algebra, and provides the user with an interface to compiled objects. The NIMBLE language represents a compilable domain-specific language (DSL) embedded within R. This paper provides an overview of the design and rationale for NIMBLE along with illustrative examples including importance sampling, Markov chain Monte Carlo (MCMC) and Monte Carlo expectation maximization (MCEM).

Keywords: domain-specific language; hierarchical models; probabilistic programming; R; MCEM; MCMC

1 Introduction

Rapid advances in many statistical application domains are facilitated by computational methods for estimation and inference with customized hierarchical statistical models. These include such diverse fields as ecology and evolutionary biology, education, psychology, economics, epidemiology, and political science, among others. Although each field has different contexts, they share the statistical challenges that arise from non-independence among data – from spatial, temporal, clustered or other sources of shared variation – that are often modeled using unobserved (often unobservable) random variables in a hierarchical model structure (e.g., Banerjee et al., 2003; Royle and Dorazio, 2008; Cressie and Wikle, 2011).

Advancement of analysis methods for such models is a major research area, including improved performance of Markov chain Monte Carlo (MCMC) algorithms (Brooks et al., 2011), development of maximum likelihood methods (e.g., Jacquier et al., 2007; Lele et al., 2010; de Valpine, 2012), new approximations (e.g., Rue et al., 2009), methods for model selection and assessment (e.g., Hjort et al., 2006; Gelman et al., 2014), combinations of ideas such as sequential Monte Carlo and MCMC (Andrieu et al., 2010), and many others. However, the current state of software for hierarchical models leaves a large gap between the limited methods available for easy application and the newer ideas that emerge constantly in the statistical literature. In this paper we introduce a new approach to software design for programming and sharing such algorithms for general model structures, implemented in the NIMBLE package.

The key idea of NIMBLE is to combine flexible model specification with a system for programming functions that can adapt to model structures. This contrasts with two common statistical software designs. In the most common approach, a package provides a fairly narrowly constrained family of models together with algorithms customized to those models. A fundamentally different approach has been to provide a language for model specification, thereby allowing a much wider class of models. Of these, the BUGS language (Gilks et al., 1994) has been most widely used, with dialects implemented in WinBUGS, OpenBUGS, and JAGS (Lunn et al., 2000, 2012; Plummer, 2003). Other tools with their own modeling language (or similar system) include AD Model Builder and its newer version, Template Model Builder (Fournier et al., 2012; Kristensen et al., 2015); Stan (2015); BayesX (Belitz et al., 2013); and PyMC (Patil et al., 2010). All of these pack-

ages have been successful for providing specific target algorithms, such as Laplace approximation and specific kinds of MCMC, but none provide a high-level way to write many different kinds of algorithms that can use the flexibly-defined models. NIMBLE aims to do that via a compilable domain specific language (DSL; Elliott et al., 2003) embedded within R.

The design of NIMBLE uses several approaches that we think are new for statistical software. To get started, we needed a general language for model specification, for which we adopted and extended BUGS because it has been so widely used. NIMBLE processes BUGS code into a model object that can be used by programs: it can be queried for variable relationships and operated for simulations or probability calculations. R was a natural fit for implementing this idea because of its high syntactic compatibility with BUGS and its ability to modify and evaluate parsed code as a first-class object, owing to its roots in Lisp and Scheme (Ihaka and Gentleman, 1996). Second, to allow model-generic programming, we needed a way for functions to adapt to different model structures by separating one-time "setup" steps, such as querying a model's structure, from repeated "run-time" steps, such as running a Metropolis-Hastings sampler. This was accomplished by allowing these steps to be written separately and using the concepts of specialization and staged evaluation from computer science (Taha and Sheard, 1997; Rompf and Odersky, 2010). Third, we needed a way to allow high-level programming of algorithms yet achieve efficient computation. This was done by creating a compiler to translate the model and run-time functions to corresponding C++ code and interfacing to the resulting objects from R.

NIMBLE includes a domain specific language (DSL) embedded within R. "Run-time" code can be thought of as a subset of R with some special functions for handling models. Programming in NIMBLE is a lot like programming in R, but the DSL formally represents a distinct language defined by what is allowed for compilation. NIMBLE stands for Numerical Inference for statistical Models using Bayesian and Likelihood Estimation.

The rest of this paper is organized as follows. First we give an overview of NIMBLE's motivation and design, discussing each major part and how they interact, without specific implementation details. Then we give examples of three algorithms – importance sampling, MCMC, and Monte Carlo expectation maximization (MCEM; Wei and Tanner, 1990; Levine and Casella, 2001) – operating on one model to illustrate how the pieces fit together to provide a flexible system. These

examples, and the more complete code available in the online supplement, provide an introduction to NIMBLE's implementation. A complete user manual is available at the project web site (R-nimble.org).

2 Overview of NIMBLE

NIMBLE comprises three main components (Fig. 1): a new implementation of BUGS (with extensions) as a model declaration language (Fig. 1:A-C); the `nimbleFunction` system for programming with models (Fig. 1:D-G); and the NIMBLE compiler for `model` objects and `nimbleFunction`s (Fig. 1:H-J).

2.1 Design rationale

The goal of NIMBLE is to make it easier to implement and apply a variety of algorithms to any model defined as a directed acyclic graph (DAG). For example, we might want to use (i) several varieties of MCMC to see which is most efficient (Brooks et al., 2011), including programmatic exploration of valid MCMC samplers for a particular model; (ii) other Monte Carlo methods such as sequential Monte Carlo (SMC, a.k.a. "particle filters"; Doucet et al., 2001) or importance sampling; (iii) modular combinations of methods, such as combination of particle filters and MCMC in state-space time-series models (Andrieu et al., 2010) or combination of Laplace approximation and MCMC for different levels of the model; (iv) algorithms for maximum likelihood estimation such as MCEM and data cloning (Lele et al., 2007; Jacquier et al., 2007; de Valpine, 2012); (v) methods for model criticism, model selection, and estimation of prediction error (Vehtari and Ojanen, 2012) such as Bayesian cross-validation (Gelfand et al., 1992; Stern and Cressie, 2000), calibrated posterior predictive p-values (Hjort et al., 2006) or alternatives to DIC such as WAIC (Watanabe, 2010; Spiegelhalter et al., 2014); (vi) "likelihood free" or "plug-and-play" methods such as synthetic likelihood (Wood, 2010), approximate Bayesian computation (ABC; Marjoram et al., 2003), or iterated filtering (Ionides et al., 2006); (vii) parametric bootstrapping of any of the above ideas; or (viii) the same model and algorithm for multiple data sets. These are just some of many ideas that could be listed.

There are several reasons the above kinds of methods have been difficult to handle in general software. First, if one has wanted to write a package providing a new general method, one has had

to "reinvent the wheel" of model specification. This means deciding on a class of allowed models, writing a system for specifying the models, and writing the algorithm to use that system. Creating model specification systems for each package is difficult and tangential to the statistical algorithms themselves. It also results in multiple different systems for specifying similar classes of models. For example, lme4 (Bates et al., 2014), MCMCglmm (Hadfield, 2010), R-INLA (Martins et al., 2013), and others each use a different system for GLMM specification. We desired a system with the flexibility of BUGS for declaring a wide range of models, while allowing different algorithms to use the same representation of a given model.

A second limitation of current designs arises from the tension between expressing algorithms easily in a high-level language and obtaining good computational performance. High-level languages, especially R, can be slow, but low-level languages like C++ require much greater implementation effort and customization to different problems. A common solution to this problem has been to write computationally intensive steps in a low-level language and call them from the high-level language. This results in code that is less general and less accessible to other developers. Most of the general MCMC packages represent an extreme case of this phenomenon, with the algorithms hidden in a "black box" unless one digs into the low-level code. We wanted to keep more programming in a high-level language and use compilation to achieve efficiency.

2.2 Specifying models: Extending the BUGS language

We chose to build upon the BUGS language because it has been widely adopted (Lunn et al., 2009). Many books use BUGS to teach Bayesian statistical modeling (e.g., Lancaster, 2004; Kery and Schaub, 2011; Vidakovic, 2011), and domain scientists find that it helps them to reason clearly about models (Kery and Schaub, 2011). Many users of the BUGS packages think of BUGS as nearly synonymous with MCMC, but we distinguish BUGS as a DSL for model specification from its use in MCMC packages. The differences between BUGS dialects in JAGS, OpenBUGS, and WinBUGS are not important for this paper.

2.2.1 BUGS, model definitions, and models

When NIMBLE processes BUGS code (Fig. 1:A), it extracts all semantic relationships in model declarations and builds two primary objects from them. The first is a `model definition` object (Fig. 1:B), which includes a representation of graph nodes (also called *vertices* in graph theory)

and edges. The second is a `model` object (Fig. 1:C), which contains functions for investigating model structure (Fig. 1:C1), objects to store values of model variables (Fig. 1:C2) and sets of functions for model calculations and simulations (Fig 1:C3). One `model definition` can create multiple `model`s with identical structure. Normally a user interacts only with the `model` object, which may use its `model definition` object internally (Fig. 1: brown arrow from C1 to B).

At this point, it will be useful to introduce several concepts of `model definition` objects and `model` objects designed to accommodate the flexibility of BUGS. Each BUGS declaration creates a *node*, which may be stochastic or deterministic ("logical" in BUGS). For example, node `y[3]` may be declared to follow a normal distribution with mean `mu` and standard deviation `sigma` (Fig 1:A). That would make `mu` and `sigma` *parents* of `y[3]` and `y[3]` a *dependent* (or *dependency*) of `mu` and `sigma`. NIMBLE uses *variable* to refer to a possibly multivariate object whose elements represent one or more nodes. For example, the variable `y` includes all nodes declared for one or more elements of `y` (e.g., `y[1], y[2], y[3]`). A node can be multivariate, and such nodes can be occur arbitrarily in contiguous scalar elements of a variable. Groups of nodes in a variable may be declared by iteration, such that their role in the model follows a pattern, but they may also be declared separately, so it cannot be assumed in later processing that they do follow a pattern. The `model definition` uses abstractions for variables, nodes, and their graph relationships that supports handling of interesting cases. For example, a program may need to determine the dependencies of just one element of a multivariate node, even though that element is not itself a node.

Processing BUGS code in a high-level language like R facilitates some natural extensions to BUGS. First, NIMBLE makes BUGS extensible by allowing new functions and distributions to be provided as `nimbleFunction`s. Second, NIMBLE can transform a declared graph into different, equivalent graphs that may be needed for different implementation contexts. For example, NIMBLE implements alternative parameterizations for distributions by automatically inserting nodes into the graph to transform from one parameterization to another. If the function that ultimately executes gamma probability density calculations needs the `rate` parameter but the BUGS code declares a node to follow a gamma distribution using the `scale` parameter (related by rate = 1/scale), a new node is inserted to calculate `1/scale`, which is then used as the needed gamma `rate`. If

any other declaration invokes the same reparameterization, it will use the same new node. Another important, optional, graph transformation occurs when the parameter of a distribution is an expression. In that case a separate node can be created for the expression's value and inserted for use where needed. This is useful when an algorithm needs access to the value of a parameter for a particular node, such as for conjugate distribution relationships used in Gibbs sampling and other contexts. A third extension is that BUGS code can be used to define a set of alternative models by including conditional statements (i.e., `if-then-else`) that NIMBLE evaluates (in R) when the `model definition` is created. This avoids the need to copy and modify entire BUGS model definitions for each alternative model, the standard practice when using previous BUGS packages.

NIMBLE uses a more general concept of *data* than previous BUGS packages. In previous packages, a model cannot be defined without its data. In NIMBLE, *data* is a label for the role played by certain nodes in a model. For example, nodes labeled as data are excluded from calls to simulate new values into the model by default, to avoid over-writing observed values, but this default can be over-ridden by a programmer who wishes to simulate fake data sets from the model. The data label is distinct from the actual *values* of nodes labeled as data, which can be programmatically changed. For example, one might want to iterate over multiple data sets, inserting each one into the data nodes of a model and running an algorithm of interest for each.

At the time of this writing, some BUGS features are not implemented. Most notably, NIMBLE does not yet allow stochastic indexing, i.e., indices that are not constants.

2.2.2 How `model` objects are used

A `model` object is used in two ways from R and/or `nimbleFunction`s. First, one may need to query node relationships, a common step in setup code (Fig 1: brown arrow from E to C1). For example, consider a `nimbleFunction` for a Metropolis-Hastings MCMC sampler (shown in detail later). In one instance, it may be needed to sample a node called `mu[2]`, in another to sample a node called `x[3, 5]`, and so on. We refer to the node to be sampled as the *target node*. The setup stage of the `nimbleFunction` can query the `model` object to determine what stochastic nodes depend on the target node and save that information for repeated use by run-time code. Or it may be that an R function needs to query a `model` object, for example to determine if it conforms to the requirements for a particular algorithm. The implementation of the `model` uses its `model`

`definition` to respond to such queries, but the `nimbleFunction` programmer is protected from that detail.

Other examples of model queries include determining:

- Topologically sorted order of nodes, which means an order valid for sequential calculations or simulations.
- All nodes or variables in the model of a particular type, such as stochastic, deterministic, and/or data nodes.
- The position of nodes in the model: e.g., *top* nodes have no stochastic parents; *end* nodes have no stochastic dependents; and *latent* nodes have stochastic parents and dependents.
- The nodes contained in an arbitrary subset of variable elements. For example, `x[3:5]` may represent the three scalar nodes `x[3]`, `x[4]`, and `x[5]`, or it may represent one scalar node `x[3]` and one multivariate node `x[4:5]`, or other such combinations.
- Nodes or expressions with certain semantic relationships, such as the node or expression for the `rate` parameter of a gamma distribution.
- A variety of kinds of dependencies from a set of nodes. For example, stochastic dependencies (also called "Markov blankets") include all paths through the graph terminating at, and including, stochastic nodes. These are needed for many algorithms. In other cases, stochastic dependencies without data nodes are needed, such as for one time-step of a particle filter. Deterministic dependencies are like stochastic dependencies but omit the stochastic nodes themselves. This kind of dependency is useful following the assignment of a value to a node to ensure descendent stochastic nodes use updated parameter values.

The second way `model` objects are used is to manage node values and calculations, both of which are commonly needed in run-time functions (Fig 1:F2). A `model` object contains each model variable and any associated log probabilities (Fig 1:C2). It also can access functions for calculating log probabilities and generating simulations for each node (Fig 1:C3). These functions are constructed as `nimbleFunction`s from each line of BUGS code. Specifically, each node has a `nimbleFunction` with four run-time functions:

- `calculate`: For a stochastic node, this calculates the log probability mass or density function, stores the result in an element of the corresponding log probability variable (Fig. 1: C2), and returns it. For a deterministic node, `calculate` executes its computation, stores the result as the value of the node, and returns 0.
- `calculateDiff`: This is like `calculate` except that for a stochastic node it returns the difference between the new log probability value and the previously stored value. This is useful for iterative algorithms such as Metropolis-Hastings-based MCMC.
- `simulate`: For a stochastic node, this generates a draw from the distribution and stores it as the value of the node. `simulate` has no return value. For a deterministic node, `simulate` is identical to `calculate` except that it has no return value.
- `getLogProb`: For a stochastic node, this returns the currently stored log probability value corresponding to the node. For a deterministic node, this returns 0.

A `model` object has functions of the same names to call each of these node functions for an ordered sequence of nodes. With the exception of `simulate`, these return the sum of the values returned by the corresponding node functions (e.g., the sum of log probabilities for `calculate`). A typical idiom for model-generic programming is to determine a vector of nodes by inspecting the model in setup code and then use it for the above operations in run-time code.

2.2.3 `modelValues` objects for storing multiple sets of model values

A common need for hierarchical model algorithms is to store multiple sets of values for multiple model variables, possibly including their associated log probability variables. NIMBLE provides a `modelValues` data structure for this purpose. When a `model definition` is created, it builds a specification for the related `modelValues` class. When a `model` object is created, it includes an object of the `modelValues` class as a default location for model values. New `modelValues` classes and objects can be created with whatever variables and types are needed. Examples of uses of `modelValues` objects are: storing the output of MCMC; storing a set of simulated node values for input to importance sampling; and storing a set of "particle" values and associated log probabilities for a particle filter.

2.3 Programming with models

One can use model objects arbitrarily in R, but NIMBLE's system for model-generic programming is based on `nimbleFunction`s (Fig 1:D). Separate function definitions for the two evaluation stages – one `setup` function and one or more run-time functions – are written within the `nimbleFunction` (Fig 1:D1,D2). The purpose of a `setup` function is to *specialize* a `nimbleFunction` to a particular `model` object, nodes, or whatever other arguments are taken by the `setup` function. This typically involves one-time creation of objects that can be used repeatedly in run-time code. Such objects could be results from querying the model about node relationships, specializations of other `nimbleFunction`s, new `modelValues` objects, or results from arbitrary R code. When a `nimbleFunction` is called, the arguments are passed to the `setup` function, which is evaluated in R (Fig 1:E). The `nimbleFunction` saves the evaluation environment and creates the return object. The return object is an instance of a custom-generated class whose member functions are the run-time function(s) (Fig 1:F).

The two-stage evaluation of `nimbleFunction`s is similar to a function object (functor) system: the `nimbleFunction` is like an implicit class definition, and calling it is like instantiating an object of the class with initialization steps done by the `setup` function. However the `nimbleFunction` takes care of steps such as determining which objects created during `setup` evaluation need to become member data in a corresponding class definition and determining their types from specialized instances of the `nimbleFunction`. As a result, the programmer can focus on higher level logic.

The run-time functions include a default-named `run` function and arbitrary others. These are written in the NIMBLE DSL, which allows them to be evaluated natively in R (Fig 1:G) or compiled into C++ class methods (Fig 1:H). The former allows easier debugging of algorithm logic, while the latter allows much faster execution. It is also possible to omit the `setup` function and provide a single `run` function, which yields a simple function in the NIMBLE DSL that can be compiled to C++ but has no first-stage evaluation and hence no specialization. For both `model`s and `nimbleFunction`s, the R objects that use compiled or uncompiled versions provide a largely identical interface to the R user.

The NIMBLE DSL supports control of `model` and `modelValues` objects, common math operations, and basic flow control. Control of `model` objects includes accessing values of nodes and

variables as well as calling `calculate`, `calculateDiff`, `simulate`, and `getLogProb` for vectors of nodes. With these basic tools, a run-time function can *operate* a model: get or set values, simulate values, and control log probability calculations. Use of `modelValues` objects includes setting and accessing specific values and copying arbitrary groups of values between `model` and/or other `modelValues` objects using the special `copy` operation. Together these uses of `modelValues` facilitate iteration over sets of values for use in a `model` object. For example, a `modelValues` object might contain the "particle" sample of a particle filter, and the run function could iterate over them, using each one in the model for some simulation or calculation. Supported math operations include basic (vectorized) math, linear algebra, and probability distribution calculations.

The two-stage evaluation system works naturally when one `nimbleFunction` needs to use other `nimbleFunction`s. One `nimbleFunction` can specialize another `nimbleFunction` in its `setup` code, or take it as a `setup` argument, and then use it in run-time code. In addition one can create vectors of `nimbleFunction`s. There is a simple `nimbleFunction` class inheritance system that allows labeling of different `nimbleFunction`s that have the same run-time function prototype(s). For example, a `nimbleFunction` for MCMC contains a vector of `nimbleFunction`s, each of which updates (samples) some subset of the model. The latter `nimbleFunction`s inherit from the same base class. This is a light burden for the NIMBLE programmer and allows the NIMBLE compiler to easily generate a simple C++ class hierarchy. One can also create numeric objects, lists of same-type numeric objects, and customized `modelValues` objects in `setup` code for use in run-time code.

The `nimbleFunction` system is designed to look and feel like R in many ways, but there are important differences. The `setup` function does not have a programmer-defined return value because the `nimbleFunction` system takes charge by returning a specialized `nimbleFunction` (ready for run-time function execution) after calling its `setup` function. More importantly, the run-time function(s) have some highly non-R-like behavior. For efficient C++ performance, they pass arguments by reference, opposite to R's call-by-value semantics. To support the static typing of C++, once an object name is used it cannot subsequently be assigned to a different-type object. And type declarations of arguments and the return value are required in order to simplify compiler implementation. To a large extent, other types are inferred from the code.

2.4 The NIMBLE compiler

A thorough description of the NIMBLE compiler is beyond the scope of this paper, but we provide a brief overview of how `nimbleFunction`s and `model`s are mapped to C++ and how NIMBLE manages the use of the compiled C++. The NIMBLE compiler generates a C++ class definition for a `nimbleFunction`. Results of `setup` code that are used in run-time code are turned into member data. The default-named `run` member function and other explicitly defined run-time functions are turned into C++ member functions. Once the C++ code is generated, NIMBLE calls the C++ compiler and loads the resulting shared object into R. Finally, NIMBLE dynamically generates an R reference class definition to provide an interface (using active bindings) to all member data and functions of objects instantiated from compiled C++ (Fig 1:J). This creates an object with identical interface (member functions) as its uncompiled counterpart for the R user. When there are multiple instances (specializations) of the same `nimbleFunction`, they are built as multiple objects of the same C++ class. If a `nimbleFunction` is defined with no `setup` code, then there is no first-stage evaluation, and the corresponding C++ is a function rather than a class.

Compilation of `model`s involves two components. Each line of BUGS code is represented as a custom-generated `nimbleFunction` with `calculate`, `calculateDiff`, `simulate`, and `getLogProb` run-time functions. These are compiled like any other `nimbleFunction`, the only difference being inheritance from a common base class. This facilitates NIMBLE's introduction of extensibility for BUGS by allowing new functions and distributions to be provided as `nimbleFunction`s. The variables of a `model` and `modelValues` are implemented by generating simple C++ classes with appropriate member objects. Like `nimbleFunction`s, both `model`s and `modelValues` objects are automatically interfaced via R objects that have similar interfaces to their uncompiled counterparts (Fig 1:I).

For the most part, the compiler infers types and dimensionality of numeric variables and generates code for run-time size-checking and resizing. The exceptions include required declaration of run-time argument types and the return type as well as situations where size inference is not easy to implement. NIMBLE includes a library of functions and classes used in generated C++. Vectorized math and linear algebra are implemented by generating code for the Eigen C++ library (Guennebaud, Jacob, et al., 2010). Basic `for`-loops for numeric iterators and basic flow control

using `if-then-else` and `do-while` constructs are supported. The actual compilation processing converts run-time code into an abstract syntax tree (AST) with an associated symbol table, which are annotated and transformed into a C++ syntax tree. A set of R classes for representing C++ code was developed for this purpose.

Compilation of `nimbleFunction`s harnesses completed first-stage evaluation (specialization) in several ways. First, contents of objects created during `setup` evaluation can be directly inspected to determine types. Second, the compiler uses *partial evaluation* to simplify the C++ code and types needed. For example, the compiler resolves nodes in `model` objects at compile time so that the C++ code can find the right object by simple pointer dereferencing. It also converts vectors of nodes into different kinds of objects depending on how they are used in run-time code. Such partial evaluation is done in the `setup` environment, essentially as a compiler-generated extension to the `setup` code.

3 Examples

In this section we present some examples of model-generic programming and the algorithm composition it supports. This section includes more implementation details, including some code for discussion. Specifically, we show how importance sampling and Metropolis-Hastings sampling are implemented as `nimbleFunction`s. Then we show how an MCMC is composed of multiple samplers that can be modified programmatically from R. Finally we show an example of composing an algorithm that uses MCMC as one component, for which we choose MCEM. Complete code to replicate the examples is provided in the supplement.

As a model for illustration of these algorithms, we choose the pump model from the WinBUGS/OpenBUGS suite of examples (Lunn et al., 2012) because it is simple to explain and use. We assume some familiarity with BUGS. The BUGS code is:

```
pumpCode <- nimbleCode({
  for (i in 1:N){
      theta[i] ~ dgamma(shape = alpha, rate = beta) ## random effects
      lambda[i] <- theta[i]*t[i]     ## t[i] is explanatory data
      x[i] ~ dpois(lambda[i])        ## x[i] is response data
```

```
  }
  alpha ~ dexp(1.0)                    ## priors for alpha and beta
  beta ~ dgamma(0.1, 1.0)
})
```

Here `x` and `t` are to be provided as data (not shown), `theta` are random effects, and `alpha` and `beta` are the parameters of interest. We have written the gamma distribution for `theta[i]` using named parameters to illustrate this extension of BUGS. Creation of a model object called `pumpModel` from the `pumpCode` is shown in the supplement.

3.1 Importance sampling

Importance sampling is a method for approximating an expected value from a Monte Carlo sample (Givens and Hoeting, 2012). It illustrates the glaring gap between algorithms and software: although it is an old and simple idea, it is not easily available for general model structures. It involves sampling from one distribution and weighting each value so the weighted sample represents the distribution involved in the expected value. It can be used to approximate a normalizing constant such as a likelihood or Bayes factor. (It can also be combined with a resampling step to sample from a Bayesian posterior, i.e., Sampling/Importance Resampling.)

For the pump model, suppose we want to use importance sampling to approximate the marginal likelihood of `x[1:3]`, which requires integrating over the first three random effects, `theta[1:3]`, given values of `alpha` and `beta`. This is an arbitrary subset of the model for illustration. To do so one simulates a sample $\texttt{theta[1:3]}_k \sim P_{\text{IS}}(\texttt{theta[1:3]})$, $k = 1 \ldots m$, where P_{IS} is a known distribution. For mathematical notation, we are mixing the code's variable names with subscripts, so that $\texttt{theta[1:3]}_k$ is the k^{th} simulated value of `theta[1:3]`. Then the likelihood is approximated as

$$P(\texttt{x[1:3]}) \approx \frac{1}{m} \sum_{k=1}^{m} P(\texttt{x[1:3]} | \texttt{theta[1:3]}_k) \frac{P(\texttt{theta[1:3]}_k)}{P_{\text{IS}}(\texttt{theta[1:3]}_k)} \tag{1}$$

where $P(\cdot)$ indicates the part of the model's probability density or mass labeled by its argument. The ratio on the right is the importance weight for the k^{th} value of `theta[1:3]`. To keep the example concise, we assume the programmer already has a function (in R or NIMBLE) to sample

from P_{IS} and calculate the denominator of the weights. Our example shows the use of NIMBLE to calculate (1) from those inputs.

Model-generic NIMBLE code for calculation of (1) is as follows:

```
importanceSample <- nimbleFunction(
    setup = function(model, sampleNodes, mvSample) {
        calculationNodes <- model$getDependencies(sampleNodes)
    },
    run = function(simulatedLogProbs = double(1)) {
        ans <- 0.0                                              # (1)
        for(k in 1:getsize(mvSample)) {                         # (2)
            copy(from = mvSample, to = model,                   # (3)
                 nodes = sampleNodes, row = k)
            logProbModel <- model$calculate(calculationNodes)  # (4)
            if(!is.nan(logProbModel))                           # (5)
              ans <- ans + exp(logProbModel - simulatedLogProbs[k])
        }
        return(ans/getsize(mvSample))                           # (6)
        returnType(double(0))                                   # (7)
    }
)
```

The specialization step for our example would be `pumpIS <- importanceSample(model = pumpModel, sampleNodes = "theta[1:3]", mvSample = ISsample)`. Note that the arguments to `importanceSample` are defined in its `setup` function. `model` is given as the `pumpModel` object created above. `sampleNodes` – the set of nodes over which we want to integrate by importance sampling – is provided as a character vector using R's standard variable subset notation. The `ISsample` object passed as the `mvSample` argument is a `modelValues` object for providing the values sampled from P_{IS}. It does not need to be populated with sample values until the `run` function is called. Rather, at the `setup` stage, it just binds `mvSample` to (a reference to) `ISsample` for use in the `run` code.

The only processing done in the `setup` code is to query the model for the vector of ordered stochastic dependencies of the `sampleNodes`. These are needed in the `run` code to calculate the necessary part of the model in topologically sorted order. The model is queried using `getDependencies`, and the result saved in `calculationNodes`. In this case, `calculationNodes` will turn out to be (`theta[1], theta[2], theta[3], lambda[1], lambda[2], lambda[3], x[1], x[2], x[3]`). This means that `model$calculate(calculationNodes)` will return $\log(P(\texttt{x[1:3]}|\texttt{theta[1:3]}_k)P(\texttt{theta[1:3]}_k))$.

The `run` code illustrates several features of the NIMBLE DSL. It shows type declaration of the `simulatedLogProbs` argument as a vector of doubles (double-precision numbers) and (7) the return type as a scalar double. `simulatedLogProbs` represents the vector of $P_{\text{IS}}(\texttt{theta[1:3]}_k)$ values. In another implementation of importance sampling, this could be included in the `mvSample` object, but we use it here to illustrate a run-time argument. The body of the `run` function (1) initializes the answer to zero; (2) iterates over the samples in `mvSample`; (3) copies values of the `sampleNodes` from `mvSample` into the model; (4) `calculate`s the sum of log probabilities of `calculationNodes`; and (5) uses basic `if-then` logic and math to accumulate the results.

The most important insight about `importanceSample` is that it is model-generic: nothing in the `setup` code or `run` code is specific to the pump model or nodes `theta[1:3]`.

3.2 Metropolis-Hastings samplers

Next we illustrate a Metropolis-Hastings sampler with a normally-distributed random-walk proposal distribution. The model-generic code for this is:

```
simple_MH <- nimbleFunction(
  setup = function(model, currentState, targetNode) {
    calculationNodes  <- model$getDependencies(targetNode)
  },
  run = function(scale = double(0)) {
    logProb_current <- model$getLogProb(calculationNodes)                # (1)
    proposalValue <- rnorm(1, mean = model[[targetNode]], sd = scale)    # (2)
    model[[targetNode]] <<- proposalValue                                # (3)
    logProb_proposal <- model$calculate(calculationNodes)                # (4)
```

```
    log_Metropolis_Hastings_ratio <- logProb_proposal - logProb_current # (5)
    accept <- decide(log_Metropolis_Hastings_ratio)                     # (6)
    if(accept)
      copy(from = model, to = currentState, row = 1,                    # (7a)
           nodes = calculationNodes, logProb = TRUE)
    else
      copy(from = currentState, to = model, row = 1,                    # (7b)
           nodes = calculationNodes, logProb = TRUE)
    return(accept)
    returnType(integer(0))
  })
```

Suppose we want a sampler for `theta[4]` in the pump model. An example specialization step would be `theta4sampler <- simple_MH(model = pumpModel, currentState = mvState, targetNode = "theta[4]")`. Here `mvState` is a `modelValues` object with variables that match those in the model, with only one of each. This is used to store the current state of the `model`. We assume that on entry to the `run` function, `mvState` will contain a copy of all model variables and log probabilities, and on exit the `run` function must ensure that the same is true, reflecting any updates to those states. As in the importance sampling example, the only real work to be done in the `setup` function is to query the model to determine the stochastic dependencies of the `targetNode`. In this case `calculationNodes` will be `(theta[4], lambda[4], x[4])`.

The `run` function illustrates the compactness of expressing a Metropolis-Hastings algorithm using language elements like `calculate`, `getLogProb`, `copy`, and list-like access to a model node. The `scale` run-time argument is the standard deviation for the normally distributed proposal value. In the full, released version of this algorithm (`sampler_RW`), the `setup` code includes some error trapping, and there is additional code to implement adaptation of the `scale` parameter (Haario et al., 2001) rather than taking it as a run-time argument. The simplified version here is less cluttered for illustration. In addition the full version is more efficient by using `calculateDiff` instead of both `getLogProb` and `calculate`, but here we use the latter to illustrate the steps more clearly.

The lines of `run` (1) obtain the current sum of log probabilities of the stochastic dependents of the target node (including itself); (2) simulate a new value centered on the current value (`model[[targetNode]]`); (3) put that value in the model; (4) calculate the new sum of log probabilities of the same stochastic dependents; (5) determine the log acceptance probability; (6) call the utility function `decide` that determines the accept/reject decision; and (7) copy from the `model` to the `currentState` for (7a) an acceptance or (7b) vice-versa for a rejection. Again, the `setup` and `run` functions are fully model-generic.

This example illustrates natural R-like access to nodes and variables in models, such as `model[[targetNode]]`, but making this model-generic leads to some surprising syntax. Every node has a unique character name that includes indices, such as `"theta[4]"`. This leads to the syntax `model[["theta[4]"]]`, rather than `model[["theta"]][4]`. The latter is also valid, but it is not model-generic because, in another specialization of `simple_MH`, `targetNode` may have a different number of indices. For example, if `targetNode` is `"y[2, 3]"`, `model[[targetNode]]` accesses the same value as `model[["y"]][2,3]`. The NIMBLE DSL also provides vectorized access to groups of nodes and/or variables.

3.3 MCMC

To illustrate a full set of MCMC samplers for a model, we do not provide `nimbleFunction` code as above but rather illustrate the flexibility provided by managing sampler choices from R. The first step in creating an MCMC is to inspect the model structure to decide what kind of sampler should be used for each node or block of nodes. An R function (`configureMCMC`) does this and returns an object with sampler assignments, which can be modified before creating the `nimbleFunction`s to execute the MCMC. Since this is all written in R, one can control its behavior, modify the code, or write a completely new MCMC system. Once the user is happy with the MCMC configuration, the corresponding suite of specialized `nimbleFunction`s can be built, compiled, and executed.

In the case of the pump model (see supplement), we choose for illustration to start with normal adaptive random walk samplers rather than Gibbs samplers. It is apparent from Figure 2 (left panel) that the posterior is correlated between `alpha` and `beta`. One might then customize the sampler choices using this knowledge. For example, one can insert a bivariate (block) adaptive random walk sampler and then re-compile the MCMC. This results in improved mixing, reflected

as lower autocorrelations of the chain (Fig. 2, middle panel) and higher effective sample size per second of computation (Fig. 2, right panel).

3.4 Monte Carlo Expectation Maximization

MCEM is a widely known algorithm for maximum likelihood estimation for hierarchical models. It is used instead of the EM algorithm when the "expectation" step cannot be determined analytically. To our knowledge, there has been no previous implementation of MCEM that can automatically be applied to the range of model structures provided by BUGS. MCEM works by iterating over two steps: (1) MCMC sampling of the latent states given fixed parameters (top-level nodes); and (2) optimization with respect to (non-latent) parameters of the average log probability of the MCMC sample. NIMBLE provides a `buildMCEM` function in which step (1) is implemented by creating an MCMC configuration with samplers only for latent states, and step (2) is implemented by calling one of R's optimizers with a compiled `nimbleFunction` as the objective function. The top level of control of the algorithm is an R function that alternates between these steps. For the pump model, the MCEM quickly settled within 0.01 of the published values of 0.82 and 1.26 for `alpha` and `beta` (George et al., 1993), which we consider to be within Monte Carlo error.

4 Discussion

We have introduced a system for combining a flexible model specification language with a high-level algorithm language for model-generic programming, all embedded within R. Numerous other algorithms can be envisioned for implementation with this system, such as those listed in section (2.1) above.

However, several important challenges remain for building out the potential of NIMBLE. First, not all features of BUGS, or of graphical models in general, have so far been incorporated. A particular challenge is efficient handling of stochastically indexed dependencies, such as when discrete mixture components are latent states. This represents a dynamic graph structure and so will require a more flexible system for representing dependencies. Second, several packages have made great use of automatic differentiation, notably ADMB/TMB and Stan. Because the NIMBLE compiler generates C++ code, it would be possible to extend it to generate code that uses an automatic differentiation library. Third, there is a need to include more compilable functionality in

the NIMBLE DSL, such as use of R's optimization library from generated C++. An algorithm like Laplace approximation would be most natural if optimization and derivatives are available in the DSL. Finally, there is potential to extend the NIMBLE compiler in its own right as a useful tool for programming efficient computations from R even when there is no BUGS code involved.

The choice to embed a compilable domain-specific language within R revealed some benefits and limitations. R's handling of code as an object facilitates processing of BUGS models and `nimbleFunction` code. It also allows the dynamic construction and evaluation of class-definition code for each `model` and `nimbleFunction` and their C++ interfaces. And it provides many other benefits, perhaps most importantly that it allows NIMBLE to work within such a popular statistical programming environment. On the negative side, NIMBLE needs some fundamentally different behavior than R, such as call-by-reference and functions that work by "side effects" (e.g., modifying an object without copying it). Such inconsistencies make NIMBLE something of a conceptual hybrid, which could be viewed as practical and effective by some or as inelegant or confusing by others. And for large models, NIMBLE's compilation processing suffers from R's slow execution.

We built upon BUGS as a model specification language because it has become so widely used, but it has limitations. First, BUGS uses distribution notation slightly different from R, so combining BUGS and R syntaxes in the same system could be confusing. In particular some BUGS distributions use different default parameterizations than R's distributions of the same or similar name. Second, BUGS does not support modular model programming, such as compactly declaring common model substructures in a way that re-uses existing code. It also does not support vectorized declarations of scalar nodes that follow the same pattern (it requires `for`-loops instead). These are extensions that could be built into NIMBLE in the future. Other extensions, such as declaration of single multivariate nodes for vectorized calculations, were implemented almost automatically as a result of NIMBLE's design. Third, one could envision powerful uses of programmatically generating model definitions rather than writing them in static code. This could be done via NIMBLE's model definition system in the future.

Other quite distinct lines of research on software for graphical models come from "probabilistic programming" efforts by computer scientists, such as Church (Goodman et al., 2008) and BLOG (Milch et al., 2006). Their motivations are somewhat different, and their programming style and

concepts would be new to many applied statisticians. It will be interesting to see where these two distinct motivations for similar programming language problems lead in the future.

SUPPLEMENTARY MATERIAL

R code for examples: R code to run examples with NIMBLE package installed. (R code file)

5 Acknowledgements

We thank Jagadish Babu for contributions to an early, pre-release version of NIMBLE. This work was supported by grant DBI-1147230 from the US National Science Foundation and by support to DT from the Berkeley Institute for Data Science.

References

Andrieu, C., A. Doucet, and R. Holenstein (2010). Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 72*(3), 269–342.

Banerjee, S., B. P. Carlin, and A. E. Gelfand (2003). *Hierarchical Modeling and Analysis for Spatial Data* (1st ed.). Boca Raton, Fla: Chapman and Hall/CRC.

Bates, D., M. Maechler, B. Bolker, and S. Walker (2014). *lme4: Linear mixed-effects models using Eigen and S4*. R package version 1.1-7.

Belitz, C., A. Brezger, T. Kneib, S. Lang, and N. Umlauf (2013). *BayesX: Software for Bayesian Inference in Structured Additive Regression Models*. Version 2.1.

Brooks, S., A. Gelman, G. Jones, and X.-L. Meng (Eds.) (2011). *Handbook of Markov Chain Monte Carlo* (1st ed.). Boca Raton: Chapman and Hall/CRC.

Cressie, N. and C. K. Wikle (2011). *Statistics for Spatio-Temporal Data* (1 ed.). Wiley.

de Valpine, P. (2012). Frequentist analysis of hierarchical models for population dynamics and demographic data. *Journal of Ornithology 152*, 393–408.

Doucet, A., N. De Freitas, and N. Gordon (2001). *Sequential Monte Carlo methods in practice*. New York: Springer.

Elliott, C., S. Finne, and O. de Moor (2003). Compiling embedded languages. *Journal of Functional Programming 13*(2).

Fournier, D. A., H. J. Skaug, J. Ancheta, J. Ianelli, A. Magnusson, M. N. Maunder, A. Nielsen, and J. Sibert (2012). AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software 27*(2), 233–249.

Gelfand, A. E., D. K. Dey, and H. Chang (1992). Model determination using predictive distributions with implementation via sampling-based methods. In *Bayesian statistics, 4*, pp. 147–167. Oxford Univ. Press, New York.

Gelman, A., J. Hwang, and A. Vehtari (2014). Understanding predictive information criteria for Bayesian models. *Statistics and Computing 24*(6), 997–1016.

George, E. I., U. E. Makov, and A. F. M. Smith (1993). Conjugate likelihood distributions. *Scandinavian Journal of Statistics 20*(2), 147–156.

Gilks, W. R., A. Thomas, and D. J. Spiegelhalter (1994). A language and program for complex Bayesian modelling. *Journal of the Royal Statistical Society. Series D (The Statistician) 43*(1), 169–177.

Givens, G. H. and J. A. Hoeting (2012). *Computational Statistics* (2 edition ed.). Hoboken, N.J: Wiley.

Goodman, N., V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum (2008). Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-08)*Corvallis, Oregon, pp. 220–229. AUAI Press.

Guennebaud, G., B. Jacob, et al. (2010). Eigen v3. http://eigen.tuxfamily.org.

Haario, H., E. Saksman, and J. Tamminen (2001). An adaptive Metropolis algorithm. *Bernoulli 7*(2), 223–242.

Hadfield, J. D. (2010). MCMC methods for multi-response generalized linear mixed models: The MCMCglmm R package. *Journal of Statistical Software 33*(2), 1–22.

Hjort, N. L., F. A. Dahl, and G. Hognadottir (2006). Post-processing posterior predictive p values. *Journal of the American Statistical Association 101*(475), 1157–1174.

Ihaka, R. and R. Gentleman (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics 5*(3), 299–314.

Ionides, E., C. Breto, and A. King (2006). Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the United States of America 103*(49), 18438–18443.

Jacquier, E., M. Johannes, and N. Polson (2007). MCMC maximum likelihood for latent state models. *Journal of Econometrics 137*(2), 615–640.

Kery, M. and M. Schaub (2011). *Bayesian Population Analysis using WinBUGS: A hierarchical perspective* (1st ed.). Boston: Academic Press.

Kristensen, K., A. Nielsen, C. W. Berg, H. J. Skaug, and B. Bell (2015). TMB: Automatic differentiation and Laplace approximation. ArXiv e-print; in press, *Journal of Statistical Software*.

Lancaster, T. (2004). *Introduction to Modern Bayesian Econometrics* (1st ed.). Malden, MA: Wiley-Blackwell.

Lele, S., B. Dennis, and F. Lutscher (2007). Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte carlo methods. *Ecology Letters 10*(7), 551–563.

Lele, S. R., K. Nadeem, and B. Schmuland (2010). Estimability and likelihood inference for generalized linear mixed models using data cloning. *Journal of the American Statistical Association 105*(492), 1617–1625.

Levine, R. and G. Casella (2001). Implementations of the Monte Carlo EM algorithm. *Journal of Computational and Graphical Statistics 10*(3), 422–439.

Lunn, D., C. Jackson, N. Best, A. Thomas, and D. Spiegelhalter (2012). *The BUGS Book: A Practical Introduction to Bayesian Analysis* (1st ed.). Boca Raton, FL: Chapman and Hall/CRC.

Lunn, D., D. Spiegelhalter, A. Thomas, and N. Best (2009). The BUGS project: Evolution, critique and future directions. *Statistics in Medicine 28*(25), 3049–3067.

Lunn, D. J., A. Thomas, N. Best, and D. Spiegelhalter (2000). WinBUGS - A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing 10*(4), 325–337.

Marjoram, P., J. Molitor, V. Plagnol, and S. Tavar (2003). Markov chain Monte Carlo without likelihoods. *Proceedings of the National Academy of Sciences of the United States of America 100*(26), 15324 –15328.

Martins, T. G., D. Simpson, F. Lindgren, and H. Rue (2013). Bayesian computing with INLA: New features. *Computational Statistics & Data Analysis 67*, 68–83.

Milch, B., B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov (2006). BLOG: Probabilistic models with unknown objects. In L. D. Raedt, T. Dietterich, L. Getoor, and S. H. Muggleton (Eds.), *Probabilistic, Logical and Relational Learning - Towards a Synthesis*, Number 05051 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Patil, A., D. Huard, and C. J. Fonnesbeck (2010). PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software 35*(4), 1–81.

Plummer, M. (2003). JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling.

Rompf, T. and M. Odersky (2010). Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, New York, NY, USA, pp. 127–136. ACM.

Royle, J. and R. Dorazio (2008). *Hierarchical modeling and inference in ecology*. London: Academic Press.

Rue, H., S. Martino, and N. Chopin (2009). Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 71*(2), 319–392.

Spiegelhalter, D. J., N. G. Best, B. P. Carlin, and A. van der Linde (2014). The deviance information criterion: 12 years on. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 76*(3), 485–493.

Stan (2015). Stan: A C++ library for probability and sampling, version 2.9.0.

Stern, H. S. and N. Cressie (2000). Posterior predictive model checks for disease mapping models. *Statistics in Medicine 19*(17-18), 2377–2397.

Taha, W. and T. Sheard (1997). Multi-stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, New York, NY, USA, pp. 203–217. ACM.

Vehtari, A. and J. Ojanen (2012). A survey of Bayesian predictive methods for model assessment, selection and comparison. *Statistics Surveys 6*, 142–228.

Vidakovic, B. (2011). *Statistics for Bioengineering Sciences - With MATLAB and WinBUGS Support*. Springer.

Watanabe, S. (2010). Asymptotic equivalence of Bayes cross validation and widely applicable information criterion in singular learning theory. *The Journal of Machine Learning Research 11*, 3571–3594.

Wei, G. and M. Tanner (1990). A Monte-Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms. *Journal of the American Statistical Association 85*(411), 699–704.

Wood, S. N. (2010). Statistical inference for noisy nonlinear ecological dynamic systems. *Nature 466*(7310), 1102–1104.

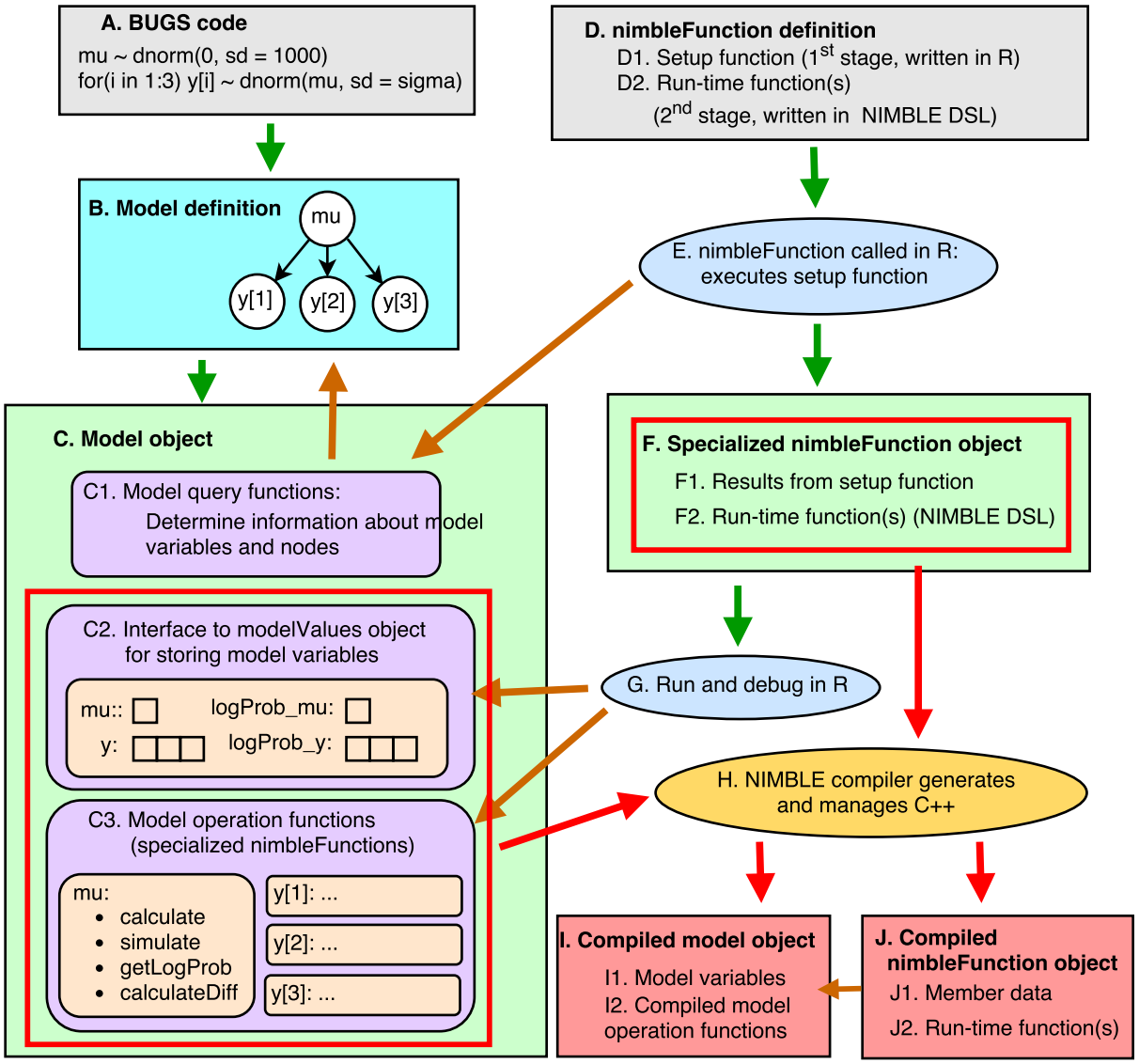

Figure 1: Overview of NIMBLE. Left side: A model starts as BUGS code (A), which is turned into a `model definition` object (B), which creates an uncompiled `model` object (C). Right side: A `nimbleFunction` starts as model-generic code (D). It is specialized to a model and/other arguments by executing its `setup` function (E), which may inspect the `model` structure (brown arrow, using C1). This returns an uncompiled, specialized `nimbleFunction` object (F). Its run-time function(s) can be executed in R, using the uncompiled model (brown arrows), to debug algorithm logic (G). Parts of the `model` and `nimbleFunction` (red boxes) can be compiled (H), creating objects (I, J) that can be used from R similarly to their uncompiled counterparts. Gray = code. Blue = R execution. Green, purple & tan = Uncompiled objects that run in pure R. Green arrows = pre-compilation workflow. Red boxes & arrows = compilation workflow.

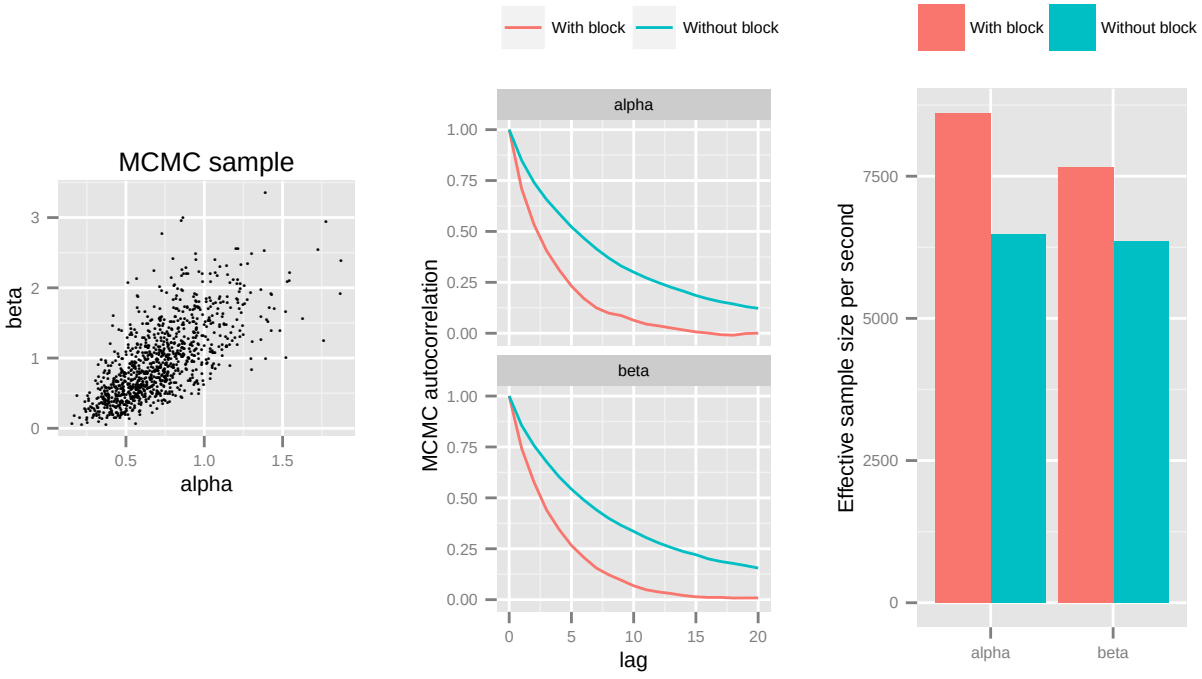

Figure 2: Example of how high-level programmability and compilation allow flexible composition of efficient algorithms. This uses the "pump" model from the classic BUGS examples. Left panel: Parameters α and β show posterior correlation. Middle panel: MCMC mixing is summarized using the estimated autocorrelation function. When a bivariate (block) adaptive random walk sampler is added to the suite of univariate adaptive random walk samplers, the chain autocorrelation decreases, reflecting better mixing. Right panel: Computational performance measured as the effective sample size per second of computation time is greater with the block sampler included.