SABL 2015a Handbook*

September, 2015

Abstract

This Handbook is an overview of the sequentially adaptive Bayesian learning (SABL) algorithm and a reference for users of SABL 2015a. Detailed aspects of the software, for example function and variable descriptions, are self-documenting; analytic treatments, including statements of conditions and theorems, are provided in working papers and publications. This Handbook is linked to those sources at appropriate points.

*SABL was created and is maintained with institutional support from University of Technology Sydney and the Australian Research Council Centre of Excellence for Mathematical and Statistical Frontiers. The team resonsible for SABL consists of John Geweke, Distinguished Professor; Huaxin Xu, Senior Scientific Programmer; Bin Peng, Postdoctoral Research Fellow; and Simon Yin, Scientific Programmer, all at the UTS School of Business. SABL is made possible with generous supprt from the ARC, ARC grant DP130103356, and through ARC sponsorship of ACEMS, ARC grant CE140100049.

Contents

1 Introduction

This Handbook is an introduction to the Matlab toolbox SABL, an implementation of the sequentially adaptive Bayesian learning algorithm; for brevity, the SABL toolbox and the SABL algorithm.

The SABL algorithm is a generalization of adaptive posterior simulators for Bayesian inference described in Durham and Geweke (2015). That work is motivated by the pleasingly parallel structure of sequential Monte Carlo algorithms, explained at the start of Section 2, in conjunction with the power of graphics processing unit (GPU) hardware and software that together provide inexpensive, massively parallel desktop scientific computing detailed in Section 3.2.2. The SABL algorithm builds on a substantial literature in particle filtering, as discussed in Durham and Geweke (2015).

The generalizations incorporated in the SABL toolbox include quite a few variants of the algorithm, and the toolbox readily accommodates the incorporation of more. The variants include the extension of sequential Monte Carlo to optimization problems (Sections 2.7 and 4.1.1), producing algorithms that can also be viewed as extensions of simulated annealing algorithms; see Geweke and Frischknecht (2014) and references there.

The SABL toolbox augments core Matlab functions as do all Matlab toolboxes, for example the Matlab Statistics and Matlab Parallel Computing Toolboxes. More important, the SABL toolbox exploits the modular structure of the algorithm. Incorporating new variants of the algorithm amounts to providing Matlab (or C) code that respects a simple interface. The same is true of new models and new prior distributions, which amount to code for prior densities and likelihood functions in the case of Bayesian inference and code for objective functions in the case of optimization. SABL is specifically designed to facilitate incorporation of new models by third parties referred to as *modelers* in this Handbook. SABL is also designed as a vehicle for applied scientific work drawing on models already incorporated in SABL. Going forward we refer to such applications as *projects* and to those who do this work as *users*.

SABL source code is open. It is freely available and may be used subject to the terms of the BSD license of the Open Software Initiative that protects it. The terms of this license are provided in the file `/Copyright_license/Copyright_software`.

2 The SABL algorithm

The SABL algorithm is a procedure for the controlled introduction of new information. It pertains to situations in which information can be represented as the probability distribution of a finite dimensional vector. SABL approximates this distribution by means of many (typically on the order of 10^4 to 10^6) alternative versions of the vector. These versions are called *particles*, reflecting some of SABL's connections to the particle filtering literature. In the SABL algorithm particles undergo a sequence of transformations as information is introduced. With minor exceptions accounting for a negligible frac-

tion of computing time in typical research applications, these transformations amount to identical instructions that operate on each particle in isolation. SABL is therefore a pleasingly parallel algorithm. This property is responsible for dramatic decreases in computing time for many research applications with GPU execution of SABL.

At its highest level the SABL algorithm looks like this:

- Represent initial information
- While information not entirely incorporated
  - Determine information increment and incorporate by weighting particles
  - Remove the weights by resampling
  - Modify the particles to represent the information more efficiently
- End

In the sequential Monte Carlo literature each pass through the loop While ... End is known as a *cycle*, and we will use ℓ to index cycles. The three steps in each cycle are *phases*. The first step is the *correction phase*, the second the *selection phase*, and the third is the *mutation phase*; for short, C phase, S phase and M phase.

Let $\theta \in \Theta \subseteq \mathbb{R}^d$ denote the vector whose probability distribution represents information. The notation reflects SABL's roots in Bayesian inference for a parameter vector. We develop the main ideas in this context and subsequently treat optimization as a variant, in Section 2.7. Denote the particles by θ_{jn}, the double subscripts indicating the J groups of N particles each employed by SABL. Initially θ has probability density $p^0(\theta)$; extension beyond absolutely continuous distributions is easy, and this streamlines the notation. In SABL the particles initially are

$$\theta_{jn}^{(0)} \overset{iid}{\sim} p^{(0)}(\theta) \quad (j = 1, \ldots, J; n = 1, \ldots, N). \tag{1}$$

In Bayesian inference $p^{(0)}(\theta)$ is a proper prior density and in optimization it is the probability density of an instrumental distribution (see Section 2.7). It must be practical to sample from the initial distribution (1) and to evaluate $p^{(0)}(\theta)$.

Denote the density incorporating all the information by $p^*(\theta)$. SABL requires that it be possible to evaluate a kernel $k(\theta)$ with the properties

$$k(\theta) \geq 0 \;\forall\; \theta \in \Theta, \quad \int_\Theta k(\theta)\, d\theta < \infty, \;\; p^*(\theta) \propto k^*(\theta) = p^{(0)}(\theta)\, k(\theta). \tag{2}$$

In Bayesian inference the kernel $k(\theta)$ is the likelihood function,

$$k(\theta) = p(y_{1:T} \mid \theta), \tag{3}$$

where T denotes sample size and $y_{1:T} = \{y_1, \ldots, y_T\}$ denotes the data. In the optimization problem $\max_{\theta \in \Theta} h(\theta)$,

$$k(\theta) = \exp[r \cdot h(\theta)], \tag{4}$$

where $r > 0$ and typically r is large.

Cycle ℓ begins with the kernel $k^{(\ell-1)}$ and ends with the kernel $k^{(\ell)}$. In the first and last cycles,

$$k^{(0)} = 1 \quad \text{and} \quad k^{(L)}(\theta) = k(\theta),$$

respectively. Correspondingly define

$$k^{*(\ell)}(\theta) = p^{(0)}(\theta) k^{(\ell)}(\theta), \tag{5}$$

implying

$$k^{*(0)} = p^{(0)}(\theta) \quad \text{and} \quad k^{*(L)}(\theta) = k^{*}(\theta). \tag{6}$$

The particles change in each cycle, and reflecting this let $\theta_{jn}^{(\ell)}$ denote the particles at the end of cycle ℓ. The initial particles $\theta_{jn}^{(0)}$ have the common distribution (1) and are independent. In succeeding cycles the particles $\theta_{jn}^{(\ell)}$ continue to be identically distributed but they are not independent. The theory underlying SABL, discussed further in this section and developed in detail by Durham and Geweke (2015) drawing on sequential Monte Carlo theory, assures that the final particles $\theta_{jn} = \theta_{jn}^{(L)} \xrightarrow{d} p^{*}(\theta)$. This convergence in distribution takes place in N, the number of particles per group. The result is actually stronger: the particles are ergodic in N, meaning that for any function g for which $E[g(\theta)] = \int_{\Theta} g(\theta) p^{*}(\theta) d\theta$ exists,

$$\lim_{N\to\infty} N^{-1} \sum_{n=1}^{N} g(\theta_{jn}) = E[g(\theta)] \tag{7}$$

with probability 1 in each group $j = 1, \ldots, J$.

A leading technical challenge in practical sequential Monte Carlo algorithms, which of course work with finite N, is to limit the dependence amongst particles, and in particular to keep dependence from increasing from one cycle to the next to the point that the final distribution of particles is an unreliable representation of any distribution at all. A further technical challenge is to provide a measure of the accuracy of the approximation implicit in the left side of (7) for finite N that is itself reliable. The SABL algorithm and toolbox do both in a way that makes minimal demands on users. The remainder of this section, and Section 3 that follows, provide the details.

2.1 C phase

For each cycle ℓ define the weight function

$$w^{(\ell)}(\theta) = k^{(\ell)}(\theta) / k^{(\ell-1)}(\theta).$$

The theory underlying the SABL algorithm requires that there exist an upper bound $\overline{w}^{(\ell)}$, that is,

$$w^{(\ell)}(\theta) < \overline{w}^{(\ell)} < \infty \ \forall\ \theta \in \Theta.$$

The C phase determines $w^{(\ell)}(\theta)$ explicitly and thereby defines

$$k^{(\ell)}(\theta) = w^{(\ell)}(\theta) \cdot k^{(\ell-1)}(\theta) \tag{8}$$

and

$$p^{*(\ell)}(\theta) = k^{*(\ell)}(\theta)\, d\theta / \int_{\Theta} k^{*(\ell)}(\theta)\, d\theta.$$

Thus (5) and (8) imply $k^{*(\ell)}(\theta) = w^{(\ell)}(\theta) \cdot k^{*(\ell-1)}(\theta)$ as well. In SABL the weight functions $w^{(\ell)}(\theta)$ are designed so that there exists $L < \infty$ for which $k^{(L)}(\theta) = k(\theta)$, although the value of L is in general not known at the outset.

One approach in designing the weight function is to use the functional form $w^{(\ell)}(\theta) = k(\theta)^{\Delta_\ell}$ and determine a sequence of positive increments $\{\Delta_\ell\}$ with $\sum_{\ell=1}^{L} \Delta_\ell = 1$. Thus at the end of cycle ℓ, $k^{(\ell)}(\theta) = k(\theta)^{r_\ell}$ where $r_\ell = \sum_{s=1}^{\ell} \Delta_s$. This variant of the C phase is known as *power tempering* or simply *tempering*. The term originates in the simulated annealing literature in which $T_\ell = r_\ell^{-1}$ is known as *temperature* and $\{T_\ell\}$ as the *cooling schedule*. Another approach originates in particle filtering and Bayesian inference: $k^{(\ell)}(\theta) = p(y_{1:t_\ell} \mid \theta)$, where $0 < t_1 \ldots < t_L = T$ for a sample of size T. The increments are therefore $w^{(\ell)}(\theta) = p\left(y_{t_{\ell-1}+1:t_\ell} \mid y_{1:t_{\ell-1}}, \theta\right)$. This variant of the C phase is known as *data tempering*.

The C phase can be motivated informally by analogy to importance sampling, a long-established Monte Carlo simulation method, interpreting $k^{*(\ell-1)}(\theta)$ as the kernel of the source density and $k^{*(\ell)}(\theta)$ as the kernel of the target density. (Recall the definition of $k^{*(\ell)}(\theta)$ in (5).) If it were the case that the particles $\theta_{jn}^{(\ell-1)}$ were independent and had common distribution indicated by the kernel density $k^{*(\ell-1)}(\theta)$, then

$$\begin{aligned} &\frac{\sum_{j=1}^{J}\sum_{n=1}^{N} w\left(\theta_{jn}^{(\ell-1)}\right) g\left(\theta_{jn}^{(\ell-1)}\right)}{\sum_{j=1}^{J}\sum_{n=1}^{N} w\left(\theta_{jn}^{(\ell-1)}\right)} \xrightarrow{a.s.} \frac{\int_{\Theta} k^{*(\ell)}(\theta)\, g(\theta)\, d\theta}{\int_{\Theta} k^{*(\ell)}(\theta)\, d\theta} \\ = &\int_{\Theta} p^{*(\ell)}(\theta)\, g(\theta)\, d\theta = E^{(\ell)}[g(\theta)] \end{aligned} \tag{9}$$

so long as $E^{(\ell)}[g(\theta)]$ exists. The convergence is in N, the number of particles per group.

The core of the argument for importance sampling is

$$\int_{\Theta} p^{*(\ell)}(\theta)\, g(\theta)\, d\theta = \frac{\int_{\Theta} w^{(\ell)}(\theta)\, k^{*(\ell-1)}(\theta)\, g(\theta)\, d\theta}{\int_{\Theta} w^{(\ell)}(\theta)\, k^{*(\ell-1)}(\theta)\, d\theta} = \frac{\int_{\Theta} w^{(\ell)}(\theta)\, p^{*(\ell-1)} g(\theta)\, d\theta}{\int_{\Theta} w^{(\ell)}(\theta)\, p^{*(\ell-1)}(\theta)\, d\theta}.$$

This result does not apply strictly, here, because while the particles $\theta_{jn}^{(\ell-1)}$ are identically distributed, they are not independent and $k^{*(\ell-1)}(\theta)$ is at best an approximation of the kernel density of the true common distribution of the particles $\theta_{jn}^{(\ell-1)}$ so long as $N < \infty$ (as it must be in practice). But many of the practical concerns in importance sampling carry over. In particular, success lies in $w(\theta)$ being "well-conditioned" – loosely speaking, variation in $w(\theta_{jn})$ must not be too great. For example, difficulties arise when

just a few weights $w(\theta_{jn})$ account for most of the sum. In this case the target density kernel $k^{*(\ell)}(\theta)$ is represented almost entirely by a small number of particles and the approximation of $E^{(\ell)}[g(\theta)]$ implicit in the left side of (9) is poor.

The C phase directly confronts the key question of how much information to introduce in cycle ℓ: too little and L will be larger than it need be; too much, and it becomes difficult for the other phases to convert ill-weighted particles from cycle $\ell-1$ into particles from cycle ℓ sufficiently independent that the representation of the distribution does not deteriorate from one cycle to the next into a state of gross unreliability. A conventional and effective way to monitor the quality of the weight function is by means of *relative effective sample size*

$$RESS^{(\ell)} = \frac{ESS^{(\ell)}}{JN} = \frac{\left[\sum_{j=1}^{J}\sum_{n=1}^{N} w^{(\ell)}\left(\theta_{jn}^{(\ell-1)}\right)\right]^2}{JN\sum_{j=1}^{J}\sum_{n=1}^{N} w^{(\ell)}\left(\theta_{jn}^{(\ell-1)}\right)^2}. \tag{10}$$

The *effective sample size* $ESS^{(\ell)}$ is an adjustment to the sample size (number of particles, JN) that accounts for lack of balance in the weights, and relative effective size is its ratio to sample size. Notice that if all weights are the same then $ESS^{(\ell)} = JN$ and $RESS^{(\ell)} = 1$, whereas if only one weight is positive then $ESS^{(\ell)} = 1$ and $RESS^{(\ell)} = 1/JN$.

In general $RESS^{(\ell)}$ is lower the more information is introduced in the C phase. This is always true for power tempering and as a practical matter is nearly always the case for data tempering. It suggests a strategy of introducing no further information after $RESS^{(\ell)}$ has attained or fallen below a target value $RESS^*$. The target $RESS^* = 0.5$ is usually reasonable, and it is the default value in the SABL toolbox. Practical experience shows that somewhat higher $RESS^*$ leads to more cycles but faster execution in the M phase, lower $RESS^*$ to fewer cycles but slower M phase execution, and as a result there is not much difference in execution time over the interval $(0.1, 0.9)$ for $RESS^*$.

Before any new information is introduced in the C phase $w^{(\ell)}(\theta) = 1$. Data tempering entails iterations $s = 1, 2, \ldots$ in which iteration s introduces $y_{t_{\ell-1}+s}$, updates

$$w^{(\ell)}\left(\theta_{jn}^{(\ell-1)}\right) = w^{(\ell)}\left(\theta_{jn}^{(\ell-1)}\right) \cdot p\left(y_{t_{\ell-1}+s} \mid y_{t_{\ell-1}+s-1}, \theta_{jn}^{(\ell-1)}\right),$$

and computes the corresponding $RESS^{(\ell)}$. Iterations terminate the first time $RESS^{(\ell)} < RESS^*$. This procedure has been well established in the sequential Monte Carlo particle filtering literature for years.

Such strategies have not been employed previously for power tempering. The first instance appears to be Geweke and Frischknecht (2014). Substituting $w^{(\ell)}(\theta) = k(\theta)^{\Delta_\ell}$ in (10),

$$RESS^{(\ell)} = \frac{\left[\sum_{j=1}^{J}\sum_{n=1}^{N} k\left(\theta_{jn}^{(\ell-1)}\right)^{\Delta_\ell}\right]^2}{JN\sum_{j=1}^{J}\sum_{n=1}^{N} k\left(\theta_{jn}^{(\ell-1)}\right)^{2\Delta_\ell}}. \tag{11}$$

Setting $RESS^{(\ell)} = RESS^*$ in (11) produces a nonlinear equation in the single variable Δ_ℓ that has a unique and easily computed solution so long as $RESS^* \in (0,1)$. If the solution implies $r^{(\ell)} > 1$ then $\Delta^{(\ell)} = 1 - r^{(\ell-1)}$ instead and the cycle $\ell = L$ is the last one.

2.2 S phase

The rest of cycle ℓ starts with the weighted particles $\theta_{jn}^{(\ell-1)}$ from the end of the C phase and produces unweighted particles $\theta_{jn}^{(\ell)}$ that that meet or exceed a mixing condition – a measure of lack of dependence described in the next section – at the end of the M phase. The S phase begins this process, removing weights by means of resampling. The principle behind resampling is to regard the weight function as the kernel of a discrete probability function defined over the particles and draw from this distribution with replacement. Hence the name selection phase. SABL performs this operation on each group of particles separately – that is, particles are always selected within groups and never across groups. This independence between the groups $j = 1, \ldots, J$ is essential in (1) proving the convergence of the algorithm, (2) assessing the mixing condition in the M phase, and (3) providing a numerical standard error for the approximation as discussed in Section 2.4.2. Resampling produces unweighted particles denoted $\theta_{jn}^{(\ell,0)}$.

The most elementary resampling method is to make N independent and identically distributed draws from the multinomial distribution with argument N and probabilities

$$p_{jn} = w^{(\ell)}\left(\theta_{jn}^{(\ell-1)}\right) / \sum_{i=1}^{N} w^{(\ell)}\left(\theta_{ji}^{(\ell-1)}\right) \quad (n = 1, \ldots, N).$$

This method is known as *multinomial resampling*. An alternative method, known as *residual resampling*, is to compute the same probabilities and collect an initial subsample of size $N^* \leq N$ consisting of $[N \cdot p_{jn}]$ copies of each particle θ_{jn}, where the function $[\cdot]$ is standard notation for what is variously known as the greatest whole integer, greatest integer not greater than, or floor function. Then draw the remaining $N - N^*$ particles by means of multinomial resampling with probabilities $p_{JN}^* \propto Np_{jn} - [N \cdot p_{jn}]$. Residual resampling results in lower dependence amongst the particles $\theta_{jn}^{(\ell,0)}$ $(n = 1, \ldots, N)$ than does multinomial resampling. For both methods there are central limit theorems that are essential to demonstrating convergence and interpreting numerical standard errors. There are other resampling methods that lead to even less dependence amongst the particles, but for these methods central limit theorems do not apply. These methods are all described in Douc et al. (2005).

The S phase is a simple but key part of the SABL algorithm. Resampling is also a key part of evolutionary (or, genetic) algorithms where it plays much the same role. The particles $\theta_{jn}^{(\ell,0)}$ $(n = 1, \ldots, N)$ are for this reason sometimes called the *children* of the *parent* particles $\left\{\theta_{jn}^{(\ell-1)}\right\}$ $(n = 1, \ldots, N)$, and also to emphasize the fact that for each child $\theta_{jn}^{(\ell,0)}$ there is a parent $\theta_{jn'}^{(\ell-1)}$. Parents with larger weights are likely to have more

children – it is not hard to work out the exact distribution of the number of children of a given parent for any one parent for multinomial resampling and then again for residual resampling. With both, the expected number of children, or fertility, of the parent $\theta_{jn}^{(\ell-1)}$is proportional to $w\left(\theta_{jn}^{(\ell-1)}\right)$, a measure of the parent's "success" in the environment of the information introduced in cycle ℓ.

2.3 M phase

If the algorithm were to continue in this way, the number of unique children would never increase and in general would decrease from cycle to cycle. Indeed, in the context of Bayesian inference it can be shown under mild regularity conditions that the number of unique particles converges almost surely to 1 as the number of observations increases. The same can be demonstrated in the context of optimization for a sufficiently large value of r in (4).

The M phase addresses this problem by creating diversity amongst sibling particles in a way that is faithful to the information kernel $k^{*(\ell)}(\theta)$. It does so using the same principle of invariance that is central to Markov chain Monte Carlo (MCMC) algorithms, drawing particles from a transition density $dQ^{(\ell)}(\theta \mid \theta^*)$ with the property

$$\int_{\Theta} k^{*(\ell)}(\theta^*)\, dQ^{(\ell)}(\theta \mid \theta^*)\, d\theta^* = k^{*(\ell)}(\theta) \;\; \forall\; \theta \in \Theta. \tag{12}$$

The transition density $dQ^{(\ell)}$ is invariant with respect to the kernel $k^{*(\ell)}(\theta)$, which preserves the original distribution of the children but introduces the prospect that they will be different. Notice that (12) implies that the successive application, or convolution, of a series of invariant transitions defines a transition that is itself invariant. The universe of invariant transition densities is large and manifest in the MCMC literature. Many of these transitions are model-specific, for example Gibbs sampling variants of MCMC. On the other hand a number of families of Metropolis-Hastings transitions apply quite generally and with problem-specific tuning of parameters can be computationally efficient.

SABL 2015a incorporates one of these variants, the Metropolis Gaussian random walk, and the structure of SABL accommodates addition of others in the future. The M phase applies the Metropolis random walk repeatedly in steps $s = 1, 2, \ldots$, each step generating a new set of particles $\theta_{jn}^{(\ell,s)}$ from the previous set $\theta_{jn}^{(\ell,s-1)}$. Following the familiar arithmetic, candidate new particles are generated $\theta_{jn}^{*(\ell,s)} \sim N\left(\theta_{jn}^{(\ell,s-1)}, \Sigma^{(\ell,s-1)}\right)$ and accepted with probability

$$\min\left[\frac{k^{*(\ell)}\left(\theta_{jn}^{*(\ell,s)}\right)}{k^{*(\ell)}\left(\theta_{jn}^{(\ell,s-1)}\right)},\; 1\right].$$

In SABL $\Sigma^{(\ell,s)}$ is proportional to the sample variance of $\theta_{jn}^{(\ell,0)}$ computed using all the particles. The factor of proportionality increases when the rate of candidate acceptance

in the previous step exceeds a specified threshold and is decreased otherwise. This draws on established practice in MCMC and works well in this context. Section 4.3 provides more detail about this threshold, as well as the initial value and increments of the scaling factor.

In some applications, especially those with a long parameter vector θ, the multivariate normal distribution is a sufficiently poor approximation of the local behavior of $k^{*(\ell)}(\theta)$ that the Metropolis Gaussian random walk can be quite inefficient. A straightforward way to address this contingency is the blocked Metropolis Gaussian random walk variant of the M phase. In this variant θ is partitioned into subvectors and the Gaussian random walk Metropolis algorithm is applied to the subvectors in turn. Section 4.3.2 provides more detail.

The objective of the M phase is to attain a degree of independence of the particles $\theta_{jn}^{(\ell)}$ at the end of each cycle sufficient to render the final set of particles $\theta_{jn} = \theta_{jn}^{(L)}$ a reliable representation of the distribution implied by the probability density function $p^*(\theta)$. The idea behind M phase termination in SABL is to measure the degree of mixing (lack of dependence) amongst the particles at the end of each Metropolis step s of cycle ℓ, and terminate when this measure meets or exceeds a certain threshold.

In SABL mixing is measured by the average relative numerical efficiency (RNE) of a group of functions chosen specifically for this purpose in each model. The RNE of the SABL approximation of a posterior moment $E[g(\theta)] = \int_\Theta g(\theta) p^*(\theta) d\theta$ is a measure of its numerical accuracy relative to that achieved by a hypothetical simulation $\theta_{ij} \overset{iid}{\sim} p^*(\theta)$. Section 2.4.2 explains how this measure is constructed. In the M phase the RNE of the particles $\left\{\theta^{(\ell,s)}\right\}$ tends to increase with the number of steps s, though not monotonically.

A simple stopping rule for the M phase is to terminate the iterations of the Metropolis random walk when the average RNE of a group of functions first exceeds a stated threshold. In any application there are practical limits to the average RNE that can be achieved through these iterations, and so it is also desirable to impose a limit on their number. Achieving greater independence of particles is especially important in the last cycle, because at the end of the M phase in that cycle the particles constitute the representation of $p^*(\theta)$. There are quite a few options for M phase termination, detailed in Section 4.3. The SABL core default criterion is average RNE 0.4 with 100 maximum iterations in cycles $1, \ldots, L-1$ and average RNE 0.9 with 300 maximum iterations in the final cycle L.

Mixing thoroughly is not the objective of the M phase. In MCMC that is essential in providing a workable representation of the distribution with kernel $k^*(\theta)$. In SABL the C and S phases take on this important task, whereas the function of the M phase is to place a lower bound on the dependence amongst particles.

2.4 Convergence, the two-pass variant of SABL and accuracy

Durham and Geweke (2015) shows that bounded likelihood

$$\max_{\theta\in\Theta} p\left(y_{1:T} \mid \theta\right) < \infty \tag{13}$$

and existence of the prior moment

$$\int_{\Theta} \left|g\left(\theta\right)\right| p^{(0)}\left(\theta\right) d\theta < \infty \tag{14}$$

respectively are sufficient for the essential condition (7) (page 7). (Weaker conditions exist but are more difficult to verify: see Durham and Geweke (2015) and references cited there.) In all posterior simulators the assessment of numerical accuracy is based on a central limit theorem, which in this context takes the form

$$N^{1/2}\left(\overline{g}^{(J,N)} - \overline{g}\right) \xrightarrow{d} N\left(0, \sigma_g^2\right) \tag{15}$$

where

$$\overline{g} = \int_{\Theta} g\left(\theta\right) p^*\left(\theta\right) d\theta \quad \text{and} \quad \overline{g}^{(J,N)} = N^{-1}\sum_{n=1}^{N} g\left(\theta_{jn}\right).$$

By itself (15) is not enough: it is essential to compute or approximation σ_g^2 as well. Section 2.4.2 explains how SABL does this.

2.4.1 Convergence and the two-pass variant

The theory developed in the sequential Monte Carlo literature provides a start. It posits a fixed pre-specified sequence of kernels $k^{(1)}, \ldots, k^{(L)}$ (see (8)) and a fixed pre-specified sequence of M phase transition densities $dQ^{(\ell)}$ (see (12)), together with side conditions (implied by conditions (13) and (14)), and proves (15). But in any practical application the kernels $k^{(\ell)}$ and transition densities $dQ^{(\ell)}$ are adaptive, relying on information in the particles $\theta_{jn}^{(\ell-1)}$ or $\theta_{jn}^{(\ell,s-1)}$, rather than fixed. The theory does not apply then because the kernels and transitions depend on the random particles, and the structure of this dependence is so complex as to preclude extension of the existing theory to this case – especially for the transition kernels $dQ^{(\ell)}$. Thus, this literature provides a theory of sequential Bayesian learning but not a theory of *sequentially adaptive* Bayesian learning. It is universally recognized that some form of adaptation is required, for it is impossible to pre-specify kernels $k^{(\ell)}$ and transition densities $dQ^{(\ell)}$ that provide reliable approximations in tolerable time without knowing a great deal about the posterior distribution – which, of course, is the goal and not the starting point.

Durham and Geweke (2015) deals with this issue by creating the two-pass variant of the algorithm. The first pass is exactly as described in this section, with the addition that the kernels $k^{(\ell)}$ and transitions $dQ^{(\ell)}$ are saved. For the specific variants described in Sections 2.1 and 2.3, this amounts to saving the sequence $\{r_\ell\}$ or $\{t_\ell\}$ from the C

phase and the doubly-indexed sequence of variance matrices $\Sigma^{(\ell,s-1)}$ from the M phase, but the idea generalizes to other variants of the C and M phases. The second pass re-executes the algorithm (with different seeds for the random number generator) and uses the kernels $k^{(\ell)}$ and transitions $dQ^{(\ell)}$ computed in the first pass, skipping the work required to compute these objects from the particles. The theory developed in the sequential Monte Carlo literature then applies directly to the second pass, because the kernels $k^{(\ell)}$ and transitions $dQ^{(\ell)}$ are in fact fixed in the second pass. The role of the first pass is to provide the knowledge of the posterior distribution required for sensible pre-specification of these objects.

Experience thus far is that substantial differences between the first and second passes do not arise, and can only be made to do so by specifying imprudently small values of N. Thus in practice it suffices to use the two-pass algorithm only occasionally – perhaps at the inception of a research project when the general character of the model(s), data and sample size are known, and then again prior to communicating findings.

2.4.2 Numerical accuracy

The sequential Monte Carlo literature provides abstract expressions for σ_g^2 in (15) but no means of evaluating or approximating σ_g^2. SABL provides the approximation using the particle groups. Consider the second pass of the two-pass algorithm where the convergence theory fully applies. In this setting there is no dependence of particles across groups. The M phase and the C phase are perfectly parallel: exactly the same operations applied to all the particles with no communication between particles. Resampling in the S phase, which introduces dependence amongst particles, takes place entirely within groups so as not to compromise independence across groups. Therefore the approximations $\overline{g}_{jN} = N^{-1} \sum_{n=1}^{N} g(\theta_{jn})$ of $\overline{g} = E[g(\theta)]$ are independent across the groups $j = 1, \ldots, J$. A central limit theorem (15) applies within each group so long as $g(\theta)$ has finite second moment. Computing the cross-group mean $\overline{g}_{J,N} = J^{-1} \sum_{j=1}^{J} \overline{g}_{jN}$, a conventional estimate of σ_g^2 in (15) is

$$\widehat{\sigma}_g^2 = N \cdot (J-1)^{-1} \sum_{j=1}^{J} \left(\overline{g}_{jN} - \overline{g}_{J,N} \right)^2 \tag{16}$$

and

$$(J-1)\, \widehat{\sigma}_g^2 / \sigma_g^2 \xrightarrow{d} \chi^2 (J-1), \tag{17}$$

the convergence in (17) being in particles per group N. In the limit $N \to \infty$, $\overline{g}_{J,N}$ and $\widehat{\sigma}_g^2$ are independent.

The corresponding numerical variance estimate for $\overline{g}_{J,N}$ is

$$\widehat{\sigma}_{g,JN}^2 = (JN)^{-1} \widehat{\sigma}_g^2. \tag{18}$$

This should not be confused with the approximation of the posterior variance,

$$\widehat{\mathrm{var}}(g) = (JN)^{-1} \sum_{j=1}^{J} \sum_{n=1}^{N} \left[g(\theta_{jn}) - \overline{g}_{J,N} \right]^2.$$

The *numerical standard error* corresponding to (18) is $\widehat{\sigma}_{g,JN} = \left[\widehat{\sigma}^2_{g,JN}\right]^{1/2}$. This is the measure of accuracy used in SABL. From (17) the formal interpretation of numerical standard error is

$$\frac{\overline{g}_{J,N} - \overline{g}}{\widehat{\sigma}_{g,JN}} \xrightarrow{d} t\left(J-1\right).$$

If particles within groups are independent then $\widehat{\sigma}^2_g \cong \widehat{\text{var}}\left(g\right)$, whereas if they are not then usually $\widehat{\sigma}^2_g > \widehat{\text{var}}\left(g\right)$, although $\widehat{\sigma}^2_g < \widehat{\text{var}}\left(g\right)$ may occur and is more likely with smaller numbers of particle groups J. The *relative numerical efficiency* of the approximation $\overline{g}_{J,N}$ is

$$RNE_g = \widehat{\text{var}}\left(g\right)/\widehat{\sigma}^2_g. \tag{19}$$

A useful interpretation of (19) is that a hypothetical simulator with $\theta_{jn} \overset{iid}{\sim} p^*\left(\theta\right)$ would achieve the same accuracy with $RNE_g \cdot JN$ particles.

This argument does not apply directly in the first pass because of the adaptation. In particular, recall that RNE is used in the M phase to assess mixing and determine the end of the sequence of iterations of the Metropolis random walk. This is an example of the complex feedback between particles and adaptation in the algorithm that has frustrated central limit theorems. This shortfall in theory is likely to persist. The two-pass procedure overcomes the problem and, moreover, provides the foundation for future variants of the algorithm without the overhead of establishing convergence for each variant.

2.5 Marginal likelihood

The SABL algorithm is particularly well suited to providing a numerical approximation of the marginal likelihood

$$ML = \int_\Theta p^{(0)}\left(\theta\right) p\left(y_{1:T} \mid \theta\right) d\theta = \int_\Theta k^*\left(\theta\right) d\theta. \tag{20}$$

The marginal likelihood, also called the marginal data density, is central in the theory and practice of Bayesian model comparison, as well as in Bayesian model averaging for combining models and decision-making. Marginal likelihood provides the bases for determining relative model probabilities, conditional on the data, and for combining models in forecasting or for the purpose of decision-making. Two models with the same likelihood function, but different prior distributions, are different models and thus marginal likelihood provides objective evidence on the suitability of one prior distribution relative to another. This is important: in some quarters choosing a prior distribution is still regarded as subjective and unscientific, but this is entirely false: all aspects of model specification are subjective, but for all aspects, including the prior distribution, evidence (science) can be brought to bear on the quality of the specification.

The approximation of marginal likelihood has posed a particularly difficult technical problem that has seen checkered resolution in the posterior simulation literature as well

as in practice: depending on the combination of posterior simulation method and model, approximation of ML can be easy to impossible, and reliably assessing the accuracy of the approximation poses further issues that are again specific to the situation.

The SABL algorithm overcomes these difficulties because it produces approximations of ML – more precisely $\log ML$ as is standard – as a by-product of the C phase. Here we will present the ideas behind the method, without going into full detail which requires considerable additional notation. Details are in Section 4 of Durham and Geweke (2015) and are reflected in the SABL toolbox code. From (6) and (8),

$$\begin{aligned}\int_{\Theta} k^{*}(\theta)\, d\theta &= \frac{\int_{\Theta} k^{*(L)}(\theta)\, d\theta}{\int_{\Theta} k^{*(0)}(\theta)\, d\theta} = \prod_{\ell=1}^{L} \frac{\int_{\Theta} k^{*(\ell)}(\theta)\, d\theta}{\int_{\Theta} k^{*(\ell-1)}(\theta)\, d\theta} \\ &= \prod_{\ell=1}^{L} \frac{\int_{\Theta} w^{(\ell)}(\theta)\, k^{*(\ell-1)}(\theta)\, d\theta}{\int_{\Theta} k^{*(\ell-1)}(\theta)\, d\theta} = \prod_{\ell=1}^{L} \int_{\Theta} w^{(\ell)}(\theta)\, p^{(\ell-1)}(\theta)\, d\theta. \qquad (21)\end{aligned}$$

In the C phase of cycle ℓ, as $N \to \infty$,

$$\overline{w}_{\ell,J,N} = (JN)^{-1} \sum_{j=1}^{J} \sum_{n=1}^{N} w^{(\ell)}\left(\theta_{jn}^{(\ell-1)}\right) \xrightarrow{a.s.} \int_{\Theta} w^{(\ell)}(\theta)\, p^{(\ell-1)}(\theta)\, d\theta,$$

Hence from (21),

$$\prod_{\ell=1}^{L} \overline{w}_{\ell,J,N} \xrightarrow{a.s.} \int_{\Theta} k^{*}(\theta)\, d\theta,$$

where the convergence is again in the number of particles per group N. This is the marginal likelihood (20) in a Bayesian inference context. Durham and Geweke (2015) discusses the approximation of $\log(ML)$ and computing the numerical standard error for that approximation.

2.6 Prediction

Bayesian inference implemented by posterior simulation provides a general practical approach to most prediction problems. The generic structure is that we wish to access the probability distribution of observable but not yet observed random vectors conditional on (a) our current knowledge about the process that produces observed values and (b) some known aspects of the environment in which the observables will be determined.

$$p(\text{conditional observables} \mid \text{conditional environment, process knowledge}). \qquad (22)$$

The idea is to simulate hypothetical future observables from this distribution.

The more structured version of this relationship in Bayesian statistics arises from several elements. With accompanying standard notation, they are

- A specified distribution of observables $\widetilde{y}$ conditional on a parameter vector θ and and environment $\widetilde{x}$, $p(\widetilde{y} \mid \theta, \widetilde{x})$;

- The environment $\widetilde{x} = x$ in which the observed values $\widetilde{y} = y$ were realized;
- The environment $\widetilde{x} = x^*$ in which the not yet observed values $\widetilde{y} = y^*$ are realized;
- The specified distribution $p(\widetilde{y} \mid \theta, \widetilde{x})$ applies both *ex ante*, $y \mid (\theta, x) \sim p(\widetilde{y} \mid \theta, x)$, and conditionally, $y^* \mid (\theta, x^*) \sim p(\widetilde{y} \mid \theta, x^*)$;
- There is additional information A about θ arising independently of x, y, x^* and y^*. Thus

$$p(y^* \mid A, x, y, x^*) = \int p(y^* \mid \theta, x)\, p(\theta \mid x, y, A)\, d\theta. \tag{23}$$

The SABL algorithm, like posterior simulators generally, produces a random sample

$$\theta_{ij} \sim p(\theta \mid x, y, A)\ (j = 1, \ldots, J; i = 1, \ldots, N) \tag{24}$$

for the specific case in which the additional information A about θ takes the form of a proper prior distribution $p(\theta \mid A)$. (The information A then takes the form of the hyperparameters of the prior distribution.) By then producing the accompanying random draw

$$y^*_{ij} \sim p(\widetilde{y} \mid \theta_{ij}.x^*) \tag{25}$$

the result is a random sample of size $J \cdot N$ drawn from (23).

Three cases are of particular interest.

1. In out-of-sample prediction the set of observed values y is not empty. The conditional observables may correspond to a hypothetical $x = x^*$. In this case $p(y^* \mid A, x, y, x^*)$ is a counter-factual or "what if" prediction. The conditional observables may arise from a temporal setting in which $\widetilde{x}$ in general includes past realized values of $\widetilde{y}$ implying that x^* includes some or all of y; x^* may also include deterministic aspects of the future like calendar and trend effects.

2. In prior predictive simulation the set of observed values y is empty, and

$$p(y^* \mid A, x, y, x^*) = p(y^* \mid A, x^*)$$

is then known as the prior predictive distribution. This is precisely the situation that arises when an investigator is contemplating model and prior specification before collecting data. It is also a useful – sometimes the only – way to assess the properties of model and prior specifications. The prior predictive distribution, used in these contexts, has well-established attractive formal Bayesian properties (Box, 1980; Geweke, 2005, Section 8.3.1).

3. In posterior predictive simulation the conditional environment is specified to be the same as the unconditional environment, $x^* = x$. The resulting sample y^*_{ij} then has the interpretation as a representation of what the observable would have been had it come precisely from the specified model and prior as opposed to the

process that actually generated it. The posterior predictive distribution does not fall within a strict Bayesian analysis, for it steps outside conditioning on the model. This precludes formal Bayesian use of the posterior predictive distribution (Box, 1980; Geweke, 2005, Section 8.3.2). Nevertheless it can indicate in an informal way the limitations of the specified model and prior, but one must exercise care in taking from the posterior distribution lessons for improved model specification, for this constitutes using the same data more than once and can give rise to problems like over-fitting.

To facilitate prediction, SABL models all include convenient functions that produces (25), and then (24) and (25) together provide the representation of the predictive distribution.

2.7 Optimization

Return now to the optimization problem, determining $\theta^* = \arg\max_{\theta\in\Theta} h(\theta)$. As discussed in Section 2, SABL approaches this problem using kernels of the form (4) in a manner analogous to the likelihood function $p(y_{1:T} \mid \theta)$ in Bayesian inference. There continues to be an initial density $p^{(0)}(\theta)$, and the corresponding distribution is sometimes call the instrumental distribution in this context. This might or might not be intended or interpreted as the expression of prior beliefs about the solution of the optimization problem. The density $p^{(0)}(\theta)$ is a technical device providing an initial condition for the algorithm and it suffices that $p^{(0)}(\theta) > 0 \;\forall\; \theta \in \Theta$.

If $h(\theta)$ is bounded above on Θ (the analogue of (13)) then SABL produces particles θ_{ij} whose distribution has kernel density $p^{(0)}(\theta)\exp[r \cdot h(\theta)]$. If h has a unique global mode θ^* then, under weak side conditions stated in Geweke and Frischknecht (2014), $\theta \xrightarrow{p} \theta^*$ as $r \to \infty$. It is typical to see a steady increase in power with each successive cycle. In fact for twice continuously differentiable objective functions $h(\theta)$ it can be shown that the rate of change $(r_\ell - r_{\ell-1})/r_{\ell-1}$ comes to depend only on the number of elements in θ. If the C phase power tempering criterion is $RESS^* = 0.5$ then for θ with 3 elements it is 1.153 , 10 elements 0.56, 20 elements 0.349, and 50 elements 0.198. Values are lower for higher $RESS^*$ and vice versa. Observed rates are often very close to the theoretical values in most cycles.

This leaves open the practical problem of terminating the algorithm. If the algorithm is simply left to run then differences in $p^{(0)}(\theta)\exp[r \cdot h(\theta)]$ become dominated by the limitations of 64-bit arithmetic rather than the shape of the function. This adversely affects the performance of the Metropolis search algorithm in the M phase and in consequence accepted candidates do little to increase effective sample size and so the increment to r decreases from the theoretical rates of change to values so low that the entire algorithm effectively "stalls". Performance for high values of r, and the point at which the approximation to the optimum suffices, are entirely problem-specific. Generally this decision can be made by joint consideration of some summary values of algorithm performance at the end of each cycle as follows:

1. Power of the algorithm, r; equivalently, the temperature $1/r$.

2. Current maximum, $p^* = \max_{i=1,...,N;j=1,...,J} p(\theta_{ij})$, especially in relation to the value in the previous cycle. If the value of the objective function has substance (e.g., measured in dollars) this may be especially attractive.

3. The distribution of $p(\theta_{ij})$ across particles; again, the interpretation is clear given a substantive value of the objective function.

4. The distribution of the particles θ_{ij}.

5. Fraction of particles θ_{ij} for which $p(\theta_{ij}) = p^*$. In simple problems this value can exceed 0.5 and even approach 1, indicating that the maximum has been computed up to the limits of 64-bit arithmetic. Geweke and Frischknecht (2014) find this for six canonical problems in the global optimization literature.

SABL does not use these criteria to provide a default algorithm stopping rule. Instead, it makes the summary values available to the user at the end of each cycle, and the user includes code in `p_monitor` that determines whether to continue with another cycle or terminate the algorithm. Section 4.1.3 indicates how to access to the summary values above.

3 The SABL toolbox

The SABL toolbox, like all other Matlab toolboxes, becomes an integral part of Matlab once it is stalled. Instructions for installing Matlab are given in the README file in the SABL parent directory. This directory shows that SABL code is organized into four separate directories. The user need not be concerned with this organization, but the distinction between code in these directories also separates SABL code by functionality.

Many subsections in this and the next section (models and priors) indicate, parenthetically, the corresponding Matlab control window `help` command the covers much of the same material.

3.1 Organization

SABL organizes code and memory in a way that facilitates the incorporate of a wide variety of models, priors and objective functions. It also enables the options for tailoring SABL that make the entire algorithm very flexible.

3.1.1 Functions

Core functions (`/code`) organize the SABL algorithm at the highest levels, and organizations common to all SABL models and applications. Core functions all have names of the form `c_*` with the exception of the root function `SABL`. This code should never be

edited because doing so could cause SABL not to perform at all, or lead to errors and compromises that are evident in its performance, or (worst of all) not evident. This is not necessary for two reasons:

- SABL provides ample documentation for all users, in several forms (this Handbook being one).
- SABL provides a specific interface with core code, explained in this section, through which users can tailor SABL in many ways to their needs.

The code is available consistent with the open architecture of SABL, and the those with a deep interest in SABL details or trying to address knotty programming issues in advance applications may find it useful.

Model code (`/models`) contains the main functions that implement each model in SABL. Code that organizes the specific computations for a given model is in a subdirectory specific to that model – for example, the code for the normal model (model `normal`) is in the directory `models/normal`. Functions in these subdirectories all have names of the form `m_*`, for example `models/normal/m_message.m`. The names of these functions that interface with SABL core functions have the same name regardless of the model. The user's specification of the model name tells SABL which model subdirectory to draw upon. A modeler contributing a new model to SABL does so by creating a subdirectory in `/models`. Section 5.6 provides details on how to do this. Users should not modify model code but consulting the code may prove helpful in some advanced applications.

Utility code (`/utilities`) includes functions that handle lower level but vital aspects of SABL. Functions in this subdirectory all have names of the form `u_*`, for example `utilities/u_mean`. The vital aspects addressed in these functions include the different computing environments in which SABL performs (CPUs or GPUs, either singly or explicitly multiple). They make it possible to write all other functions in SABL without incorporating logic that is conditional on the computing environment. These functions also enable users to write simple code that performs seamlessly in these different environments and without concern for the technical minutiae that accompany the organization of computation for GPUs or multiple cores. (Consult `help SABLfunctions`.) Users should not modify model code but consulting the code may prove helpful in some advanced applications.

Library code (`/library`) includes computationally intensive functions for which it is desirable to have both Matlab code – for example `l_egarch.m` for CPU environments. The subdirectory `cufiles` has code for GPU environments, for example `cufiles/l_egarch.CUDA.cu`. Functions in this subdirectory all have names of the form `l_*`. Users should not modify model code but consulting the code may prove helpful in some advanced applications.

Project code (`/projects`) provides some examples illustrating SABL models and methods. The entire interface between a specific project application and SABL is the function `p_monitor`. In actual application, the user can – and should – create a project

folder in a place that is most convenient given the way the user organizes work. For more advanced applications the user may find it useful to create one or more other functions, invoked by `p_monitor`, consistent with the principle of modularity. So long as these functions do not have names of the form `c_*`, `l_*`, `m_*` or `u_*` there will be no conflict with SABL functions. As always in Matlab, user functions take precedence in case of conflict with Matlab or toolbox functions.

3.1.2 Global structures and stages

`help algorithm`

All data (arrays, strings, structures) used in more than one invocation of a specific function exists in one of eight global structures in SABL. Section 3.4 discusses these structures in detail.

Two of these structures (`P` and `Ppar`) are reserved for project use, and it is essential for the user to understand how these structures work in order to get the most out of the flexibility that the single function `p_monitor` provides in tailoring the SABL algorithm, models and priors to the work at hand. This understanding is essential if the user plans to use multiple CPUs explicitly, or to use one or more GPUs. Placing all but the most transient data in these structures is also essential for the user to take advantage of the way that `SABL` exploits the pleasingly parallel structure of the SABL algorithm. Using `SABL` utility functions that accomplish the same thing is equally important if the user is to write code that need not be modified when, for example, the code is run using one GPU as opposed to one or more CPU cores.

Two of these structures (`M` and `Mpar`) are reserved for model use, and it is even more important for the creator of a new SABL model to understand how these structures work.

For purposes of interfacing with models and applications in projects, and to provide access to SABL intermediate computations that are useful for many substantive purposes, `SABL` passes down through a sequence of functions provided by the model and by the project, and then back up again. This happens at 15 distinct points known as *stages*. The string `stage` is the single input argument of the function `p_monitor`, and by interrogating this argument `p_monitor` can perform different tasks at different stages. This includes harvesting the products of intermediate SABL computations. It also enables the user to change the default values of many global fields that control the way SABL works in order to tailor the algorithm to the circumstances at hand.

3.1.3 Invoking SABL

`help SABL`

To invoke SABL, use the command

```
SABL([model],[project])
```
(26)

The argument [model] can take one of three forms:

- The name of a SABL model, e.g. `'normal'` or `'poisson'`;
- The directory (absolute or relative) for model code;
- If the current directory includes all model and project code, then both arguments can be omitted.

The argument [project] can take one of three forms:

- The directory for project code (absolute or relative);
- The name of one of the example projects in the SABL toolbox;
- If invoked from the project directory the argument can be omitted.

The command `allsturctures = SABL([model], [project])` results in a structure containing all eight SABL global structures `C`, `E`, `M`, `P`, `Cpar`, `Epar`, `Mpar`, `Ppar`. The command (26), followed by the declaration `global [SABL global structure 1], [SABL global structure 2],` makes the named structures accessible from the command window.

3.1.4 `SABL help`

SABL incorporates an extension of the Matlab `help` command that encompasses a wider array of documentation.

1. For Matlab functions and for SABL functions not specific to models, `help` works in the usual way: `help [functionname]`.
2. For SABL functions specific to models (equivalent to all functions of the form `m_*`) the form is `help [modelname] [functionname]`. The more verbose form is due to the fact that different model directories have functions with the same names.
3. The `help` command also applies to all Matlab global structures and their fields. Except for the `M` and `Mpar` global structures, the form is `help [field]`, for example `help C`, `help E.gpu` or `help C.Cphase.power`. For `M` and `Mpar` the form is `help [modelname] [field]`, for example `help normal M.x_pointer`. If the field is a structure then `help` lists all the fields of that structure, classifying each field as a structure, required field, monitor field or control field together with its default value in the last case. Otherwise, `help` provides a narrative description of the field as well as the same field classification. Users will find that the `help` command streamlines coding of `p_monitor`.

4. Finally, the `help` command provides access from the command window to convenient documentation that covers many of the same details in this section on the SABL toolbox and the section that follows on models and priors. These are arranged in a three-level branching hierarchy whose top level is `help SABL`. The result of that command points to the child nodes, and the rest of the hierarchy is organized in the same way. In every case the command takes the form `help [topic]`. The child nodes of the SABL parent are:

- `algorithm:` Useful reference for the entire SABL algorithm including `stage` definitions
- `SABLfunctions`: A guide to utility functions that users are most likely to employ
- `structures`: A concise guide to documentation of the many fields of SABL global structures
- `models`: A list of model names in SABL 2015a and a concise guide to their documentation
- `priors`: A list of prior names in SABL and some of their variations

3.2 Computing environments

`help environments`

Getting things done requires both software and hardware. The SABL toolbox functions in a very particular but widely available proprietary software environment. It operates on two different kinds of hardware, one of which (traditional single or multicore CPU) is available to anyone who is able to access Matlab. In addition, and key to the power of SABL for complex inference and optimization problems, it also operates using one or more graphics processing units (GPUs) that can be installed easily and relatively inexpensively. The full combination of hardware and software is available through cluster computing facilities in many universities and is increasingly available in academic and cloud computing environments.

3.2.1 Software

The SABL toolbox is written entirely in Matlab, and Matlab, version 2012b or later, is required. Model likelihood functions have two versions, one written in Matlab and the other in C/CUDA. Using the C/CUDA version requires a C/C++ compiler (e.g. GCC, Intel C++ Compiler or Microsoft Visual C++) as well as the CUDA Toolkit from Nvidia. The CUDA Toolkit is freely available at `https://developer.nvidia.com/cuda-toolkit`. It drives the GPUs on the system and operates on top of the C/C++ compiler in compiling C/CUDA code. If Matlab runs successfully on a given combination of hardware and operating system, then so will the SABL code. By implication the Parallel Computing Toolbox must be installed to exercise the GPU option (`E.gpu = true`) in SABL.

3.2.2 Hardware

All of the options and models in SABL are available for both CPU and GPU hardware; CPU is the default and is trivial to modify as described below. SABL exploits hardware in a CPU environment in exactly the same way as Matlab. By default, Matlab attempts to utilize all available CPU cores efficiently, and the extent to which this has been done can be gauged by the ratio of CPU time to elapsed time reported at the end of every SABL run. Experience suggests that this works to greater effect in SABL to the extent that the application demands more floating-point operations; ratios been 3 and 4 can be attained in these circumstances in conventional quadcore environments.

Through the Parallel Computing Toolbox, Matlab provides the facility to access GPU(s) and dispatch execution explicitly to individual GPUs, if more than one is present. In the context of the Matlab Parallel Toolbox each GPU constitutes a worker.

The Matlab Parallel Computing Toolbox enables users to access GPU memory conveniently and implements a subset of its instructions for the GPU. With a bit of effort one can also execute Matlab on more than one GPU simultaneously. SABL code is written respecting the limitations of the Matlab instruction set for the GPU, making it a trivial matter to execute SABL exploiting a GPU-enhanced platform. Thus the modeler and user can continue to code exclusively in Matlab and SABL will exploit the GPU. However, it does so relying only on the Matlab instruction subset for the GPU, and that is an important limitation. In the Matlab Parallel Computing Toolbox the organization of GPU computing is then function specific: it takes advantage of the GPU function by function in vector arithmetic and to some extent in linear algebra. It does in any way exploit the pleasingly parallel nature of the SABL algorithm, especially important in likelihood function evaluation. In this environment GPU execution can be faster than multicore CPU execution, but at best by a small factor – rarely more than 5 and certainly less than 10 compared with quadcore CPU.

The SABL toolbox organizes execution to exploit the pleasingly parallel nature of the SABL algorithm. This organization involves the entire algorithm and extends across the many functions that implement both current and prospective variants of the algorithm, models, and objective functions. Because of this organization, GPU execution in SABL can be faster than multicore CPU execution by much higher factors

To fully exploit these advantages SABL offers the option of providing code written using C/CUDA to evaluate likelihood functions. This provides for more explicit control of threads, and in particular the ability to specify that each thread evaluates the entire likelihood or objective function for a particular particle. As the number of floating point operations per particle required to evaluate these functions increases, the fraction of total SABL execution time devoted to the evaluation of these functions increases toward one. This enhances the reduction in execution time afforded by this organization of computations on the GPU. The number of floating point operations in the entire algorithm is close to proportional to the number of particles, especially in applications dominated by likelihood function evaluation. But, because there is also fixed overhead in GPU execution, returns to this organization of computations on the GPU increase

noticeably, typically to the point of several tens of thousands of particles. In the most favorable cases we have observed execution 40 times faster in this environment using a GPU with about 500 cores, compared with a quadcore CPU.

3.3 Layering, stages and passing control

`help algorithm`

Table 1 (page 65) provides an overview of the SABL algorithm described in Section 2, links it to the highest level core code of SABL (functions `SABL`, `c_sabl` and `c_SABL_spmd`) and indicates the stages in SABL.

SABL employs three layers: core, model and project. The core layer is the algorithm itself, including all of the options for the C, S and M phases. (SABL is designed to accommodate additional phase options in the future, including options provided by third parties.) Each layer consists of a set of functions. All function names in the core layer take the form `c_*`, except for the highest-level function `SABL`. All function names in the model layer take the form `m_*`. These functions are model specific. Each model has its own directory, and from the declaration of the model name at the outset, SABL builds the path that includes the right model directory.

A particular application of a model, including the data, specification of a prior distribution, and parameters controlling the algorithm (for example, the number of particles) is called a project. Each projects must include the function `p_monitor`, which defines and manages the application. Users may find it advantageous to write additional functions invoked directly or indirectly by `p_monitor`. As always it is important that these functions should not have names the same as those in Matlab or any Matlab toolbox being used, unless the intention is to provide a replacement for the function in question. For the SABL toolbox this simply means not having functions of the form `c_*`, `l_*`, `m_*`, or `u_*`, or writing a function `SABL`.

At each of the points indicated by "Stage:" in Table 1 (page 65) top level SABL code control passes to the function `c_monitor` in the core layer. This function executes certain commands and then passes control to the function `m_monitor` in the model layer. This model-specific function executes certain commands and then passes control to the `p_monitor`, the project-specific interface provided by the user. On return from `p_monitor` control passes back to `m_monitor`, which can execute certain further commands, then to `c_monitor` which can also perform additional tasks before control returns to the top level code.

The hierarchy has several purposes, one of which is simply to accommodate many models and projects in the same system. Somewhat more subtle but quite important is that the code in the `c_monitor` and `m_monitor` functions is written so as to enable the modeler (through `m_monitor`) and the user (through `p_monitor`) to exercise a great deal of control over the details of the SABL algorithm. The pattern is that `c_monitor` provides all of the details required for the SABL algorithm, using pre-defined default choices, before passing control to `m_monitor`. (For example, the default choice for the

number of particles is $J = 16$ groups with $N = 1024$ particles apiece: `C.J=16`, `C.N=1024`.) The function `m_monitor` then has the opportunity to modify these defaults before invoking `p_monitor`, as well as to make default choices for options specific to the model, for example the form of the prior distribution. In `p_monitor` the user has the opportunity to override defaults set in either `c_monitor` or `m_monitor` before control returns to `m_monitor`, and similarly `m_monitor` provides all of the model-specific details and defaults required before control returns to `c_monitor`.

Since control passes downward from `c_monitor` to `m_monitor` to `p_monitor` and back again at many points in the algorithm SABL must communicate to those functions the point at which control has been passed. To this end the entire algorithm is partitioned into stages (a concept) and at the end of each stage there is a round trip through the `*_monitor` functions (the implementation of the concept). This single input parameter of `c_monitor`, `m_monitor` and `p_monitor` is the stage name. The SABL code, including sample projects, uses `stage` as the variable name of this input string. The current stage can also be access through the field `C.stage`.

Thus the structure of `p_monitor` code must be

```
if strcmp(stage, [stagename_1])
      execute this ...
elseif strcmp(stage, [stagename_2])
     execute that ...
                ⋮
elseif strcmp(stage, [stagename_n])
     execute yet something different ...
end
```

While the ordering of the `if` and `elseif` blocks makes no difference, it may be helpful to keep them in the same order they occur in the algorithm. It is important to bear in mind that SABL makes the round trip through the `monitor` functions at 9 points in each cycle, and in general passes through two of them (`'whileCphase'` and `'whileMphase'` more than once in each cycle – often many times). Thus it is good practice that all execution in `m_monitor` and `p_monitor` be written within `if` / `elseif` branches of the kind just described. Anything executed outside those branches will be repeated thousands of times in a typical application of SABL. If data (e.g. an array or string) need be computed only once but is used later, it is best to place it in the `P` or `Ppar` structure and then access it as needed.

3.4 Global structures

`help structures`

Most constants and arrays in SABL are fields of structures, and these structures are global. SABL employs 8 structures, as follows:

- Core constants and arrays: `C` and `Cpar`. Examples are `C.J`, the number of particle groups, and `Cpar.theta`, the matrix of particles.

- Environment constants and arrays: `E` and `Epar.` Examples are `E.gpu`, indicator for execution using one or more GPUs, and `E.pus`, number of Matlab workers. `Epar` currently has no fields.

- Model constants and arrays: `M` and `Mpar`. Examples are `M.data`, the data array in most models, and `M.x_pointer`, which defines the covariates by pointing to columns of `M.data` in many models. `Mpar` has no fields in SABL 2015a models.

- Project constants and arrays: `P` and `Ppar`. These global structures are reserved for users who find it convenient to use global structures to communicate between user-defined functions or between SABL stages. These structures, as well as any `p_*` function other than `p_monitor`, are never accessed by SABL and so they are not required. However, SABL management of GPUs and multiple processors pertains only to the fields of its 8 global structures. As a consequence users must take advantage of `P` (virtually always) and `Ppar` (only if creating arrays in that correspond to particles) if their code is to function correctly in anything other than a single-CPU environment.

All vectors (i.e., two dimensional arrays in which one size dimension is 1) in SABL are column vectors, and some SABL functions rely on this fact. SABL automatically re-orients row vectors in global structures to be column vectors, implying that users can ignore this convention when providing fields of `M` structures in `p_monitor` in stage `'startrun'`. But the convention must be respected explicitly when a user invokes a utility function directly or indirectly from `p_monitor`.

With a single exception all communication between users and SABL, and between modelers and SABL, is managed through the fields of these global structures. The one exception is the string argument `stage`, which is the single input argument of `p_monitor` (for users) and is the single input argument of `m_monitor` (for modelers), but this string is also available in the field `C.stage`.

Every field of the `C` and `E` structures that is not itself a structure (for example, `C.Cphase`, which is a structure with its own fields) is either a control field or a monitor field. `M` structures have both kinds of fields, and in addition they have required fields. A required field is a control field for which there is no default value, whereas control fields (proper) always have default values. Failure to provide a required field always produces a terminating error. `M` structures have both kinds of fields, and in addition they have required fields. A required field is a control field for which there is no default value. Control fields, proper, always have default values.

Control fields and subfields with content in the `C` structure define the specific variant of the SABL algorithm; those in the `E` structure do the same for the computing environment; and those in the `M` structure define the specific model. The `P` structure is reserved for the user. The advantage of using the `P` structure is that its contents may be accessed after the algorithm is complete, by declaring `global P` in the command window, whereas the values of local arrays in functions are lost.

There are no required fields in the C and E structures. All control fields have default values and each of these control fields requires user access only if the user wishes to change its value. Control fields of the C structure must be set in p_monitor in stage 'startrun': for example, a user wishing to use $N = 512$ particles in each of $J = 8$ groups would specify C.J=8, C.N=512 because the default core values are C.J=16, C.N=1024.) Control fields of the E structure must be set in p_monitor in stage 'open': for example, in an environment with two GPUs the user would specify E.gpu=true and E.pus = 2 because the default core values are E.gup=false and E.pus=1. In each model the M structure has required fields that define the application (i.e., inference or optimization problem) that must be declared in p_monitor in stage 'startrun', for example M.data, M.x_pointer and M.y_pointer in most models. By virtue of the passing of control downward and back upward in each stage, user values take priority over either core or model default values, but if they do not respect the conventions of the field (e.g. M.y_pointer < 1 or M.y_pointer > size(M.data, 2) then a higher level will produce a terminating error. Most models also have a number of control fields with default values, which the user may change to tailor the model to the application.

As with any software, in learning SABL it is best to begin with the default values of the control fields and then experiment with alternatives. Experimentation will lead to a deeper understanding of the algorithm's behavior that will stand the user in good stead in approaching complex applications.

Monitor fields provide access to intermediate aspects of the SABL algorithm, and are very useful for specific purposes – for example, having access to predictive log likelihoods in the 'data_whole' variant of the C phase. In these applications it can be efficient to harvest (retrieve) the contents of these fields or functions of their contents. It is most efficient to copy the harvested information into a field of the P structure and then save the entire P structure at the termination of SABL.

Example 1 *Suppose that* `p_monitor` *invokes additional functions provided by the user. The user could pass* `stage` *as an input through a hierarchy of functions as needed, but it may prove simpler to declare* `global C` *in these functions and then access* `C.stage` *directly.*

There is an important distinction between C and Cpar. Cpar is reserved for arrays whose row dimension is the number of particles per Matlab worker (C.JNwork = C.JN/E.pus). These arrays are distinguished by placement in a different structure because when there are multiple Matlab workers or multiple GPUs all C.JN particles are partitioned into E.pus sets of equal size and each set is allocated to a worker. Thus each worker "sees" only part of any array whose rows correspond to particles, and core SABL code handles function of all particles in a computationally efficient way. (SABL has specific functions devoted to this task that are also available to users, for example u_mean and u_std – see Section 3.8 or invoke help SABLfunctions.) In the same situation the fields the entire C structure is copied to each worker. The structures Epar, Mpar and Ppar exist to handle the contingency that environment globals, model globals,

or project globals have arrays that have the same size and allocation properties as the structures that are the fields of `Cpar`.

3.5 Updating with new data

`help updating`

Given an ordering of the observations $t = 1, \ldots, T$, the posterior distribution for all T observations may be expressed as a sequence of recursive updates of posterior distributions at times t_s, with $0 < t_1 < \ldots < t_S = T$:

$$p(\theta \mid y_{1:T}) \propto p(\theta) \prod_{s=1}^{S} p\left(y_{t_{s-1}+1:t_s} \mid y_{1:t_{s-1}}, \theta\right).$$

SABL facilitates this updating by making it convenient to use the distribution $p(\theta \mid y_{1:t})$ in place of the prior distribution $p(\theta)$. If particles from $p(\theta \mid y_{1:t})$ are available in a file, it is faster to update than it is to execute the entire algorithm – often much faster. This is especially advantageous in real time, on-line settings in which data arrive rapidly.

The project specification `E.simulation_record = filename`, declared by `p_monitor` in stage `'open'`, causes SABL to record all 8 global structures in file `filename`.

In an updating run of SABL the project specifies `E.simulation_get = fname` (the same name assigned to `E.simulation_record` in the earlier run of SABL). This declaration must be made by `p_monitor` in stage `'open'`. SABL then begins using these previously recorded particles rather than particles drawn from the prior distribution in the updating run. It also uses all of the data, and for this reason SABL requires that `p_monitor` provide the full, updated data set in stage `'startup'`. (SABL identifies the part that is new from the value of `C.Cphase.tlast` read from `E.simulation_get`.)

Not all of the details of the SABL algorithm need be the same in an updating run as they were in the earlier run. For example, criteria for stopping the M phase can be changed based on experience in the earlier run, as can the effective sample size criterion `C.Cstop.ress` for stopping the C phase. But certain other details cannot, for example the number and organization of particles specified by `C.J` and `C.N`. SABL checks for this particular error, but SABL 2015a does not check all such possible errors, which could cause SABL to terminate with an error message that is not particularly illuminating. The SABL updating feature is tailored for ready implementation in environments in which new data arrive regularly but the specific variant of the SABL algorithm employed, as specified in the control fields of the `C` structure, does not evolve in any significant way.

Heuristically, updating in SABL is more efficient than re-execution from the beginning to the extent that the increment to information in the update relative to the previous sample is small compared to the increment in information in the entire sample relative to the prior distribution. If the information in the prior distribution is small compared to that in a typical observation (corresponding to the colloquial "diffuse prior") then this advantage is considerable. In this situation SABL bypasses the early cycles in which observations enter the sample slowly (in the `'data_whole'` variant of the C phase) or

the power of the likelihood function increases slowly from zero (in the `'anneal_Bayes'` variant). Each iteration of the M phase takes longer in updating than it typically did in the previous run because the likelihood function incorporates more observations, but this cost would also be incurred if the entire algorithm, rather than just the update, were executed in any event. In regular updating, for example the addition of the next quarter's macroeconomic data or the next day of financial data, the update often entails just one cycle. Moreover this cycle in turn involves a very small number of iterations, sometimes just one, in the M phase. The reason for this is that by default the final cycle of SABL uses a stronger termination criterion (`C.Mstop.rne_end`) than do the earlier cycles (`C.Mstop.rne`), implying that the particles at the start of an update are more thoroughly mixed than they are at the start of a typical cycle.

It is important to distinguish updating with additional data from revision of data employed in a previous run of SABL. The SABL algorithm cannot "undo" information once it is incorporated – this is fundamental. If data are revised it is necessary to begin with the prior distribution or a posterior distribution that uses only a subset of the unrevised portions of the data.

3.6 Two-pass variant of the SABL algorithm

`help pass`

As described in Section 2 the SABL algorithm builds the kernels $k^{(\ell)}$ in the C phase of successive cycles using information in the particles $\theta_{jn}^{(\ell-1)}$ and the transition kernels $dQ^{(\ell,s)}$ in the M phase using information in the succession of particles $\theta_{jn}^{(\ell,s-1)}$ $(s = 1, 2, \ldots)$. These critical adaptations, especially in the M phase, are precluded by the theorems for sequential Monte Carlo, a key underpinning of SABL. Indeed, all practical implementations of sequential Monte Carlo are subject to this limitation.

SABL resolves this problem and achieves analytical integrity by utilizing two passes. As explained in Section 2.4.1, the first pass is the adaptive variant in which the algorithm is self-modifying. The second pass fixes the design emerging from the first pass and then executes the fixed Bayesian learning algorithm to which the theorems in the sequential Monte Carlo literature directly apply. Experience strongly suggests that the second pass leads to qualitatively the same results as the first pass – in particular, to similar simulated posterior moments and associated numerical standard errors. Differences in posterior moment approximations, between the two passes, are typically consistent with numerical standard errors. Yet this is simply an observation, and it is important to be able to check this condition at least occasionally in the course of a research project.

All that is required for a project to execute the two-pass variant of the algorithm in SABL is for `p_monitor` to set `C.twopass = true` in stage `'startrun'`. The impact of this change is easy to trace in Table 1. (page 65) SABL executes both passes in sequence without pausing, displays to the screen in similar fashion in both passes, and records the results of the second pass to file if `E.simulation_record` is not empty. The logical field `C.passone` indicates whether SABL is in the first (`true`) or second (`false`) pass.

Example 2 *Suppose that a research project makes a formal comparison of results in the first and second passes, and to this end requires access to the particles of both passes. This can be accomplished by setting* `C.twopass = true` *in stage* `'startrun'`*, and then:*

```
elseif strcmp(stage, 'finish')
    if C.passone
            P.pass1.mean = u_mean(Cpar.theta);
        P.pass1.std = u_std(Cpar.theta);
        P.pass1.nse = u_nse(Cpar.theta);
        P.pass1.rne = u_rne(Cpar.theta)
    else
           P.pass2.mean = u_mean(Cpar.theta);
        P.pass2.std = u_std(Cpar.theta);
        P.pass2.nse = u_nse(Cpar.theta);
        P.pass2.rne = u_rne(Cpar.theta);
        p_passcompare(P.pass1, P.pass2);  %  Formal comparison
    end
end
```

Users contemplating this kind of work with multiple workers can test their preparation by understanding why this code would not run at stage `'endrun'`*. Would it work if* `E.gpu = true`*, and if not what changes should be made? If the* `p_passcompare` *code led to a terminal error, the results could still be saved with the line* `global P` *followed by the line* `save filename P` *from the console. Then* `p_passcompare` *could be modified, the file loaded, and* `p_passcompare` *executed. Either way,* `p_passcompare` *has access to all SABL functions. A more conservative strategy would simply save* `P` *at the end of execution without attempting* `p_passcompare` *in stage* `'finish'`*.*

3.7 Using multiple workers and GPUs

`help environments`

As detailed in Section 2 the SABL algorithm is pleasingly parallel, meaning that almost all steps execute the same instructions on different particles and there is very little communication between particles. The SABL toolbox exploits this property by organizing execution in one of four ways, corresponding in to the Cartesian product of (a) one (`E.pus = 1`) or more (`E.pus > 1`) workers, and (b) execution using CPU cores (`E.gpu = false`) or GPU cores (`E.gpu = true`). Alternatives (a) and (b) interact as follows.

1. With one worker and CPU cores (`E.pus = 1`, `E.gpu = false`) SABL exploits the pleasingly parallel structure of the algorithm by means of array arithmetic on any dimension corresponds to particles. This is merely a convenience for coding in Matlab: at the machine execution level all computation is serial, with the important exception that the Matlab compiler takes advantage of multiple CPU cores in translating Matlab code to C code, on a function-by-function basis. Matlab

can be quite efficient in organizing multiple CPU cores, but this is not due to exploiting the pleasingly parallel structure of the SABL algorithm.

2. With several workers and CPU cores (`E.pus > 1`, `E.gpu = false`) SABL partitions particles amongst cores. (`E.pus` cannot exceed the number of CPU cores available.) Each worker (in fact, a thread) utilizes a single CPU core and threads are synchronized using Matlab's single processor multiple data protocol. This often, but not necessarily, leads to faster execution than when `E.pus = 1`: the reason is that Matlab was already exploiting multiple CPU cores with `E.pus=1`, although with a different strategy.

3. With one worker and one GPU (`E.pus = 1`, `E.gpu = true`) SABL performs all but a few housekeeping functions on the GPU. GPU code exploits the hundreds or thousands of GPU cores available, defining threads to take advantage of the parallel structure of the algorithm and the single-instruction multiple-data nature of the GPU. This is accomplished in part through Matlab GPU kernels, but most importantly through C/CUDA code for likelihood or objective function evaluation, which in turn dominates floating point arithmetic in all but the simplest models and applications.

4. With several workers and GPUs (`E.pus > 1`, `E.gpu = true`) particles are partitioned across GPUs, and then on each GPU execution proceeds as in the single-GPU case. Because of the pleasingly parallel structure of the SABL algorithm there is little need for communication between GPUs; that which takes place utilizes Matlab utilities for this kind of transfer. In all but the simplest models and applications this overhead accounts for a very small fraction of the time required to complete the computations.

User code executed from `p_monitor` in stage `'startrun'` can ignore issues related to multiple workers and GPUs in assigning constants or arrays to fields of the `P` and `M` structures: SABL allocates the fields of these structures to CPU or GPU, and across workers, in an appropriate way. Thus users who have experience with Matlab but are unfamiliar with any of the extensions in the Matlab Parallel Computing Toolbox can simply write code to set control fields at this stage in their usual manner. In other stages more care is required, especially when harvesting monitor fields into fields of the `P` or `Ppar` structures. The remainder of this section goes into some of the details and considerations that are important for these operations to proceed smoothly and successfully.

3.7.1 Using multiple workers

By default SABL uses one worker (`E.pus = 1`). Using multiple workers requires that the Matlab Parallel Computing Toolbox be installed. The user manages the opening of the Matlab pool, including specification of number of workers, as well as its closing.

The user can also do this in `p_monitor`, opening a Matlab pool in stage `'open'` and closing it in stage `'close'`. However, this leads to the complication that the Matlab pool remains open if SABL never reaches stage `'endrun'` (for example, if SABL issues a terminal error or if the user interrupts SABL) and then the user must manually close the pool or else a Matlab error arises on the next execution of the code when it attempts to open a pool that is already open. For this reason SABL does not attempt to open or close pools directly, and leaves it up to the user to handle this in `p_monitor` or outside of SABL. If the user sets `E.pus` to an integer greater than 1 in stage `'startrun'` and there is no pool open, or if there are fewer than `E.pus` workers in the pool then SABL terminates with an error message to that effect.

SABL executes in parallel on multiple workers by means of a Matlab `spmd` block, which contains all stages between (but not including) `'startrun'` and `'endrun'`. If there are multiple workers then at the first bold **Copy** line in Table 1 (page 65) identical copies of the `C`, `E`, `M` and `P` global structures are passed to each worker. By contrast the arrays in `Cpar` (as well as any other `*par` global structures, currently unused) all have leading dimension $J \cdot N =$ `C.JN` and are split equally among `E.pus` workers, so that each array then has leading dimension `C.JNwork=C.JN/E.pus`. (SABL requires that `C.J` be wholly divisible by `E.pus`.) Once inside the `spmd` block there are `E.pus` workers (more technically, threads) that execute the same instructions. These instructions create and modify arrays and constants that have the same names but are in fact different. Control passes back out of the `spmd` block at the second bold **Copy** line in Table 1 (page 65).

This implies that if there are multiple workers then the nature of global structures accessed in `p_monitor` is quite different depending on the stage (but not if `E.pus=1`). To obviate this tedious dependence for the user in `p_monitor`, SABL provides utility functions to manipulate arrays in a way that is transparent to being in or out of the `spmd` block in the invoking function. To see a list and brief description of these utility functions invoke `help SABLfunctions` from the command window, followed by `help [functionname]` for the description required to use the function correctly. Because of these utility functions, SABL code outside these functions is not contingent on `E.pus`, `E.gpu`, or the status of the `spmd` block. Code written by users can do the same.

Example 3 *Suppose that a project works with quarterly macroeconomic data and one of its objectives is to illustrate how the posterior mean and standard deviation of several interesting functions of the parameters evolved over time. (For example, how they changed before, during and after the "great moderation" might be a particular focus.) To accomplish this in most approaches would require that inference be repeated for all quarters, a time-consuming task. The option* `C.Cphase_method = 'data_whole'` *introduces data one observation at a time in the C phase and in doing so provides access to the required posterior distributions. This access is transient: it exists in each step of the C phase within each cycle, but once that step is completed the posterior distribution is modified with the introduction of the next observation, either in the next step of the C phase in the current or next cycle. When the algorithm is complete the user would like to store the posterior means and standard deviations in a file, where they can be accessed to*

create tables or figures. Code of the following form will accomplish this for **nf** *functions of interest* **g**, *regardless of whether execution takes place with* **E.pus = 1** *or* **E.pus > 1**:

```
elseif strcmp(stage, 'startrun')
    P.funcmeans = zeros(C.tlast, nf);
    P.funcstds = zeros(C.tlast, nf);
          ⋮
elseif strcmp(stage, 'whileCphase')
    %  The function func evaluates the nf functions of interest:
      g = func(Cpar.theta);  % C.JNwork x nf on each worker
    P.funcmeans(C.Cphase.t, :)  = u_mean(g, 1, E.pus>1, Cpar.logw);
    P.funcstds(C.Cphase.t, :)  = u_std(g, 1, E.pus>1, Cpar.logw);
          ⋮
elseif strcmp(stage, 'endrun')
    funcmeans = P.funcmeans;
    funcstds = P.funcstds;
    save funcfile funcmeans funcstds
```

The first and last of the three blocks execute as a single thread whereas the second block executes as **E.pus** *threads. If* **E.pus > 1** *then in the middle block there are identical copies of the global* **C**, **E**, **M** *and* **P** *structures on each thread. The* **Cpar** *structure, and hence* **Cpar.theta and g**, *differ across workers. The last block references the* **P** *structure that SABL retrieved from one of the workers just before leaving the* **spmd** *block. (Use* **help** *for explicit descriptions of* **C.Cphase.t** *and* **Cpar.logw**.)

The functions **u_mean** *and* **u_std** *deal with a number of complications that arise regularly and would otherwise be inconvenient. One is that the particles are weighted in the C phase and this affects the computations – hence the input* **Cpar.logw**. *A second one is that the computations require communication across the workers. The functions* **u_mean** *and* **u_std** *handle this in an efficient fashion, minimizing the information transferred between threads.*

Observe that if any of the arrays created as fields of the **P** *structure had instead been local, or persistent, or had been a field of any structure other than the eight defined by SABL, the computations would have failed if* **E.pus** *> 1. The only local constant or array is* **g** *in the middle block, which exists only to avoid executing* **func** *twice. This is satisfactory because* **g** *is transient – the* **E.pus** *versions of* **g** *are used only on this occasion. Some care must be exercised in writing the code in* **func** *if is to execute successfully when* **E.pus** *> 1 and/or when* **E.gpu** *is true. Since* **Cpar.theta** *can be quite large and exists on the GPU if* **E.gpu** *is true, there is the potential for the code to move particles between GPU and CPU inadvertently, thereby adding possibly substantial overhead in each iteration of the C phase. SABL provides utility functions that assist with this (Section 3.8 and* **help SABLfunctions**), *but they do not entirely preclude the possibility of mistakes of this kind.*

The efficiency of using multiple CPU cores explicitly in this way increases as problems become larger and more complex: the dominant relevant dimensions are number

of particles along with the number of floating point operations in the evaluation of the likelihood function (in Bayesian inference) or objective function (in optimization). For Bayesian inference in models that cannot exploit sufficient statistics the number of observations is a third relevant dimension. This is complicated by the fact that under the basis for comparison, `E.pus = 1`, Matlab already exploits the multiple CPU cores found in most hardware. The implications can be seen directly in SABL, which indicates both elapsed time and CPU seconds at the end: the latter is typically a multiple of the former that can be a respectable fraction (e.g., 75%) of the number of cores. The returns to explicit multicore CPU execution by assigning a value greater than 1 to `E.pus` are application-specific, but generally will be maximized by making this value the total number of CPU cores available.

In hardware with multiple GPUs, `E.pus` indicates the number of GPUs to use (in conjunction, of course, with `E.gpu = true`). Gains here can be proportional to `E.pus`. Limited experiments to date with four K80 chips on a single server reduce execution time by very close to 50% in moving from `E.pus = 1` to `E.pus = 2`, and a reduction not quite as great in moving from `E.pus = 2` to `E.pus = 4`.

3.7.2 Using graphics processing units

The Matlab Parallel Computing Toolbox also integrates computation on graphics processing units (GPUs) into Matlab code. By default, SABL executes on the CPU (`E.gpu = false`). To execute with one or more GPUs, the user declares `E.gpu = true` in stage `'open'` of `p_monitor`. `SABL` will then use `E.pus` GPUs so long as there are at least `E.pus` physical GPUs installed and available – if not, SABL terminates with an error message. If `E.pus` $>$ `1` then the user must also manage a Matlab pool in the same way described at the start of Section 3.7.1.

Just as Matlab uses the compiled language C to execute code on CPU cores, with the Parallel Computing Toolbox it uses the extension C/CUDA to execute code on GPUs. It enables the Matlab programmer to create constants, arrays and certain structures in GPU memory and transfer data from CPU to GPU and vice versa. Many Matlab commands have been extended to GPUs and have same options and variants (e.g., `svd` and `kron`); some other commands have been extended to GPUs but with particular options (e.g., `rand`, and `log` with complex output) and others have not been extended (e.g., `pinv` and `poissrnd`). These extensions are still actively underway and grow with each new edition of the Matlab Parallel Computing Toolbox, so documentation for the installed version of the Parallel Computing Toolbox should be consulted. As with any documentation the contingencies are not exhaustive and some learning by trial and error may be required. However, it is not necessary for the SABL user to master these Matlab conventions.

GPU arrays have precedence over all other types, and within GPU types precedence is the same as within CPU types (e.g., `double` takes precedence over `single`). Thus if a type `double` CPU array `A` and a type `double` GPU array `GA` are conformable, then `B = A + GA` is a type `double` GPU array. If `GA` had been a type `single` GPU array then

`B` would be type `single` GPU array. The primary tool for moving data from the CPU into the GPU array is `gpuArray`, for example `GA = gpuArray(A)`, and the primary tool for moving data from the GPU to the CPU is `gather`, for example `A = gather(GA)`. Other data types supported on the GPU include `int` and `unit` types as well as `logical`. The arguments of cell arrays and the field names of structures are pointers that exist in CPU memory. Different fields in a structure can contain both GPU and CPU data types. The SABL utility functions listed under the subheadings "Matrix arithmetic" and "Management" in `help SABLfunctions` manage the intricacies in GPU use and in writing code that performs well on both the GPU and CPU. This enables the SABL user to write code that does not branch on either `E.gpu` or `E.pus`.

GPUs work to advantage when arrays are large and operations are at least pleasingly parallel. This includes many operations in linear algebra. Because there is substantial overhead in execution on the GPU, the problem must be sufficiently large and amenable to parallel execution for net gains to occur. When these conditions are not met then execution on the GPU can take even longer than it does on CPU cores. The SABL algorithm is inherently pleasingly parallel, and it is in large and complex problems that efficiency gains matter: the gain from reducing computing time from 12 hours to 36 minutes (95%) is generally more valuable than reducing it from 2 minutes to 6 seconds (also 95%). In these large problems gains are nearly proportional to the number of GPU cores as well as to the number of GPUs, and gains of a factor of 50 to 100 compared with conventional "single CPU" implementation can be achieved with a 1,000 - processor GPU. In considering the value of this increase in speed, keep in mind that the basis of comparison is a CPU environment in which 4 or more cores are already being used efficiently, and that the cost of GPU hardware, now well under one US dollar per core, is small relative to CPU installation and this relative cost has been falling.

Whether or not gains of this order are realized is strongly affected by the way the log likelihood or objective function is evaluated. In large problems the vast majority of floating point operations are devoted to repeated evaluations of this function, which is in turn heavily concentrated in the M phase. It is because of this concentration that the parts of the SABL algorithm that are not perfectly parallel, chiefly resampling in the S phase and the assessment of acceptance rates in the M phase, are unimportant in large problems – in the relevant sense that reducing their execution time to nothing would have a proportionately negligible effect on total execution time. For likelihood or objective function evaluation to realize anything like its potential efficiency in SABL running on the GPU threads, it is important that the threads be organized to carry out function evaluation at a particle from start to finish. At least in its current incarnation, Matlab code running on the GPU cannot be instructed to do this. However, if likelihood function evaluation is carried out using a function coded in C/CUDA (more properly termed a kernel) this barrier can be overcome. More precisely, C/CUDA code organizes the threads and defines a single thread as a complete evaluation of the likelihood or objective function.

Communication between Matlab and kernels running on GPUs (for example, code in `*.cu` and files) is straightforward, just as it always has been between conventional

Matlab and functions written in C (Matlab `*.mex` files). SABL 2015a contains a full implementation of a C/CUDA kernel for likelihood functions in each model, complementing functions written in Matlab. This code, especially for `egarch`, is designed in part as a template for the implementation of C/CUDA kernels for other models. Modelers, especially, will find it helpful to consult the `cufiles` directory and `compile_CUDA.m` file in the various `models` subdirectories.

The principle that parallel operations arise because each particle is a thread is central to the efficiency of SABL in exploiting GPUs. Most existing GPU code, for example the GPU functions in the Matlab Parallel Toolbox, do not exploit parallelism arising from application of the function to thousands of different input values, which is what SABL requires. Instead, they exploit instances of parallel operations that are inherent in the arithmetic of the function as applied to a single input. (And, in fact, SABL speed-up factors often exceed those of functions in the Matlab Parallel Toolbox GPU function library.) Attempts to use GPU-enabled functions like these, which use a different principle for defining threads, operates at cross-purposes to the organization of GPU threads in SABL. Thus these functions should not be used in likelihood function evaluation: coding a likelihood function using the Matlab Parallel Toolbox GPU functions may well result in less efficient computation than with a single CPU. Since there are, indeed, a variety of situations in which it is sensible to organize threads by different function input data, such functions are currently being developed in some leading GPU libraries in the public domain and these may turn out to be useful for SABL GPU code.

In this environment, one of the greatest inefficiencies that can be introduced is frequent movement of all particles, or any array with size `C.JN` in some dimensions, from GPU to CPU or vice versa. This never happens in the core SABL code: in particular all fields of the `Cpar` global structure are GPU arrays when `E.gpu` is true. This convention should be respected by the user in creating any `Ppar` fields if the efficiency of SABL running on GPUs is to be maintained. Particles are born (S phase), mutate (M phase) and die (S phase) in GPU memory. The precedence of the `gpuArray` data type implies that natural ways of writing project code avoid transfer from GPU to CPU.

Example 4 *`x = gpuArray(randn(10,1))` or `x = gpuArray.randn([10,1])` creates `x` on the GPU. The subsequent command `y=x.^2` places `y` on the GPU as well and entails no intervening movement between CPU and GPU. This is reinforced by the fact that Matlab prohibits conversion from `gpuArray` to `double` in assignment statements: continuing the example, the subsequent command `z = zeros(10,1)` followed by `z(1) = x(1)` generates a terminal error. But with the SABL toolbox installed, if the user has declared `E` global and set `E.gpu`, then the sequence `x=u_allocate([10,1], 'randn'); y = x.^2; z = u_allocate([10,1]); z(1) = x(1)` will execute without error regardless of the value of the logical field `E.gpu`.*

Example 5 *Return to Example 3, executing `E.gpu = true` in stage `'open'`. Then `Cpar.theta` is a GPU array, and so is the vector `g` created in stage `'whileCphase'`. However, `P.funcmeans` and `P.funcstds` are CPU arrays, created in stage `'startrun'`.*

Direct conversion of a GPU array to a CPU array is prohibited, and so the lines `P.funcmeans...` *and* `P.funcstds...` *in stage* `'whileCphase'` *cause a terminal error. Matlab specifically provides the command* `gather` *in its parallel toolbox to cope with this situation, and since code using* `E.gpu = true` *successfully must have the Matlab toolbox installed, the command*

```
P.funcmeans(C.Cphase.t,:)  = gather(u_mean(g, 1, E.pus>1, Cpar.logw))
```

and similarly for `P.funcstds` *will successfully address this problem.*

But now suppose the user provides the code to a colleague using Matlab without the Parallel Computing Toolbox installed. Matlab will not recognize `gather`*, leading to an error the colleague may not recognize. Multiply this by many such occurrences in more complex code, and it's clear a major portability problem to Matlab without the Parallel Computing Toolbox installed has arisen. For this reason, all SABL code is designed to execute without the Parallel Computing Toolbox installed (and, for that matter, without the Statistics Toolbox installed) so long as* `E.gpu = false` *and* `E.pus = 1`*. The substitute for* `gather` *is* `u_setdatatype`*:*

```
P.funcmeans(C.Cphase.t,:)  = ...
u_setdatatype(u_mean(g, 1, E.pus>1, Cpar.logw), 'cpu')
```

This command causes the transfer of `16*length(g)` *bytes from GPU to CPU per observation if* `E.gpu = true`*, but acts as a simple assignment statement if* `E.gpu = false`*.. In any problem large enough that GPU computation is more efficient than CPU computation this overhead in this transfer is so negligible as to be difficult to measure.*

3.8 Utilities

`help SABLfunctions`

SABL employs many utility functions that facilitate code writing and testing. Many of them provide a single command for array creation and common mathematical operations regardless of whether the computing environment is one or multiple workers, and CPU or GPU. For example, when execution takes place on one or multiple GPUs, these functions minimize transfer of data across GPUs and between CPU and GPU.

This section organizes and lists these functions, providing very brief descriptions. The Matlab help command provides full technical detail. The actual code can be found in the utilities directory of the SABL toolbox, along with some other functions that are important to SABL but less likely to be relevant for modelers and users.

- Array manipulation
  - `u_allocate` Allocate an array on CPU or GPU memory
  - `u_gcat` Aggregate data from multiple workers to one worker
  - `u_setdatatype` Move data to CPU or GPU memory

- Matrix arithmetic
  - `u_logmeanlog` Replicate `log(mean(exp(x))` avoiding overflow
  - `u_logmomlog` Also replicate `log(std(exp(x))` avoiding overflow
  - `u_logsumlog` Replicate `log(sum(exp(x))` avoiding overflow
  - `u_max` Replicate `max(x)` with efficient handling of multiple workers
  - `u_mean` Replicate `mean(x)` with efficient handling of multiple workers
  - `u_min` Replicate `min(x)` with efficient handling of multiple workers
  - `u_mvmom` Replicate `mean(x)` and `cov(x)` with efficient handling of multiple workers
  - `u_nse` Compute numerical standard error
  - `u_rne` Compute relative numerical efficiency
  - `u_std` Replicate `std(x)` with efficient handling of multiple workers
  - `u_sum` Replicate `sum(x)` with efficient handling of multiple workers
- Management
  - `u_islab1` Replicate `labindex == 1` independent of Parallel Toolbox
  - `u_numlabs` Replicate `numlabs` independent of Parallel Toolbox
- Code testing
  - `u_pfcheck` Test probability function and random sample
  - `u_pdfcheck` Test probability density function and random sample

4 Variants of the algorithm in the SABL toolbox

`help algorithm`

The SABL code incorporates variants of tempering in the C phase, resampling in the S phase, and mutation employing an invariant transition kernel in the M phase. Similarly there are variants on procedures for determining the degree of incremental information in the C phase and degree of mixing mixing in the M phase, referred to collectively as stopping criteria.

For each of these variants this section provides a short description, often by reference to other sections of this Handbook; the protocol for invoking the option; a list of the control fields or subfields of the global structure `C` that define the variant precisely and may be set by the user; and a list of the monitor fields or subfields of the global structure `C` that may be useful in accessing the state of the algorithm and should not be changed by the user. The `help` command, in turn, documents the fields, and in the case of control fields indicates the default setting employed by SABL if the user or modeler does not set the field explicitly.

4.1 C phase

`help Cphase`

SABL 2015a has three variants of the C phase: data tempering, power tempering, and optimization. The default method is power tempering.

SABL 2015a has two variants of the C phase stopping criterion: effective sample size and optimization. Both terminate the C phase if relative effective sample size (10) reaches or falls below threshold ($RESS^{(\ell)} \leq RESS^*$). In the case of power tempering this is guaranteed at the end of the first iteration of the C phase by solving (11) for Δ_ℓ, but checking takes place regardless because it is almost costless and simplifies flow control in the code.

The following fields are common to all variants.

Control fields: `C.Cphase_method`, `C.Cstop_method`, `C.Cstop.ress`

Monitor fields: `C.Cphase.count`, `C.Cphase.total`

4.1.1 Power tempering

Description: See power tempering in Section 2.1. This is the core default.

Invocation: `C.Cphase_method = 'anneal_Bayes'`

Additional control fields: `C.Cphase.crit`

Additional monitor fields: `C.Cphase.power`, `C.Cphase.logpowers`

4.1.2 Data tempering

Description: See data tempering in Section 2.1.

Invocation: `C.Cphase_method = 'data_whole'`

Additional control fields: None

Additional monitor fields: `C.Cphase.t`, `C.logpl`

4.1.3 Optimization

Description: See power tempering in Section 2.1 and optimization Section 2.7

Invocation: `C.Cphase_method = 'anneal_optimize'`

Additional control fields: `C.Cphase.crit`

Additional monitor fields: In the structure `C.optstatus`

4.2 S phase

`help Sphase`

SABL has four variants of the S phase: multinomial, residual, systematic and stratified resampling. The default method is residual resampling. In the S phase there is no analogue of the stopping rule in the C phase or M phase. Control fields and monitoring fields are common to all variants of the S phase.

Control fields: `C.Sphase_method`

Monitor fields: `C.Sphase.nunique`, `Cpar.select`

4.2.1 Residual resampling

Description: See Section 2.2.

Invocation: `C.Sphase_method = 'residual'`. This is the core default.

4.2.2 Multinomial resampling

Description: See Section 2.2.

Invocation: `C.Sphase_method = 'multinomial'`

4.2.3 Systematic resampling

Description: This variant is more efficient than residual resampling, but there is no central limit theorem for numerical accuracy. This option is intended for research only and should not be used in applications.

Invocation: `C.Sphase_method = 'systematic'`

4.2.4 Stratified resampling

Description: This variant is more efficient than residual resampling, but there is no central limit theorem for numerical accuracy. This option is intended for research only and should not be used in applications.

Invocation: `C.Sphase_method = 'stratified'`

4.3 M phase

`help Mphase`

SABL 2015a has two variants of the M phase: Metropolis random walk and blocked Metropolis random walk. The default method is Metropolis random walk. Both variants are described in Section 2.3. In complex or less tractable applications, the specification of the M phase can be critical to SABL execution that respects practical time and resource constraints. This is an acquired skill that involves a solid understanding of the algorithm and mastering the many options for control of the M phase that SABL affords. Those options are present precisely because they have proven useful in these cases.

4.3.1 Metropolis random walk

The control field `C.Mphase_method = 'MGRW'` invokes this option, which is the core default. The M phase proceeds as described in Section 2.3. The following fields are common to all variants of the M phase and the M phase stopping rule.

Control fields: `C.Mphase_method`, `C.Mstop_method`,

C.Mphase.step_inc, C.Mphase.step_initial, C.Mphase.step_lower, C.Mphase.step_upper, C.Mphase.acceptgoal, C.Mphase.total

Monitor fields: C.Mphase.accept_rate, C.Mphase.count, C.Mphase.step, C.Mphase.rne, C.Mphase.total

Consult the respective help commands for detailed descriptions of these fields and their relationship to the description in Section 2.3.

SABL 2015a has four variants of the M phase stopping rule. Each rule is a function two criteria: the number of iterations elapsed (iterations) and the average relative numerical efficiency of several functions of the parameters (RNE).

1. When C.Mphase_method = 'steps' the stopping rule is the fixed number of iterations C.Mstop.steps (core default 100), except in the last cycle where it is C.Mstop.steps_end (core default 300).

2. When C.Mphase_method = 'RNE' the stopping rule is the minimum value of RNE C.Mstop.rne (core default 0.4), except in the last cycle where it is C.Mstop.rne_end (core default 0.9).

3. When C.Mphase_method = 'RNE&steps' the stopping rule is the maximum value of iterations C.Mstop.steps or the minimum value of RNE C.Mstop.rne, whichever comes first. In the last cycle these criteria are replaced by C.Mstop.steps_end and C.Mstop.rne_end, respectively.

4. When C.Mphase_method = 'hybrid' the iterations of the M phase continue so long as

   C.Mphase.rne $>$ C.Mstop.rne*C.Mphase.count/C.Mstop.steps.

   In the last cycle C.Mstop.rne_end replaces C.Mstop.rne and C.Mstop.steps_end replaces C.Mstop.steps in this expression. This limits the number of further iterations if RNE improvement stalls as M phase iterations continue, a phenomenon that can arise with complex models, casual formulation of strong prior distributions, and errors in code. This is the core default.

4.3.2 Blocked Metropolis random walk

The control field C.Mphase_method = 'MRGW_blocked' invokes this option, which in turn has two variants: fixed blocks and random blocks. The default is random blocks unless the user or modeler creates the field C.Mphase.blocks, a cell array with one entry per block, each entry pointing to the entries of the parameter vector θ defining that block. If the user or modeler does not create this field but creates the field C.Mphase.nblocks, each iteration of the M phase randomly allocates θ to blocks, making the block lengths as close as possible. If the field C.Mphase.nblocks is also absent then it is taken to be one-six the number of parameters rounded to the nearest integer with a minimum of 2.

An attempt to use the blocked variant of the Metropolis random walk terminates with error if the model has only one parameter.

The stopping criteria are computed and the stopping rule is imposed accordingly at the end of each iteration. The iteration stopping criterion is the total number of blocks executed in the M phase of the current cycle: thus, for example, if `C.Mstop.steps = 100` and `C.Mphase.nblocks = 4`, the M phase will execute exactly 25 complete iterations with 4 blocks in each iteration.

Invocation: `C.Mphase_method = 'MRGW_blocked'`

Control fields: `C.Mphase.ranblock`, `C.Mphase.blocks`, `C.Mphase.nblocks`

Monitor fields: `C.Mphase.blocks`, `C.Mphase.iblock`, `C.Mphase.blocks`, `C.Mphase.nblocks`

4.4 Adding a new variant of the algorithm to the SABL toolbox

Algorithm variants are manifest in the code as `if` / `elseif` blocks in the functions `c_Cphase`, `c_Sphase` and `c_Mphase`. In each case there is a field with a string indicating the variant (e.g., `C.Cphase_method = 'anneal_Bayes'`) and a function that invokes the variant, e.g. `c_Cphase_anneal_Bayes(stage)`). The functions `c_Cphase`, `c_Sphase` and `c_Mphase` recognize the string `'custom'` for the field, and control then passes to a function for that variant, e.g. `p_Cphase_custom(stage)` for the C phase and similarly for the S and M phase. The `p_` prefix recognizes the fact that a customized variant might be project specific. It also facilitates initial work on any variant that is a candidate for future inclusion in the SABL algorithm and toolbox, at which point an additional `elseif` block would be added at the appropriate place in `c_Cphase`, `c_Sphase` or `c_Mphase`.

Stopping rule variants are manifest in the code as `if` / `elseif` blocks in the functions `c_Cphase` and `c_Mphase`. The string `'custom'` is recognized for the C and M phase stopping rules, and control then passes to `p_Cstop_custom` and `p_Mstop_custom` respectively.

5 Models in the SABL toolbox

The SABL algorithm provides for Bayesian inference in any likelihood function and maximization of any objective function, in each case subject to some weak conditions described early in Section 2. The SABL toolbox exploits this structure to minimize the effort required for new models and objective functions to be added to SABL, in a manner that in turn lets users apply these models or objective functions. SABL provides a generic structure that results in a common "look and feel" across different models and objective functions. Models, in turn, provide default structures that are simple and handle many common applications by users, while also allowing users to construct variants that include a wide variety of model and prior specifications.

5.1 Model specifications and parameter maps

SABL 2015a offers a small core group of models that will be substantially expanded in the future editions of SABL.

5.1.1 SABL models

`help models`

To use a particular model, the user invokes `SABL([modelname], [project folder])` where `[project folder]` is the absolute or relative address of the directory that contains the function `p_monitor.m` and any other functions invoked by `p_monitor`. If `pwd = [project folder]` then the second input argument can be omitted. SABL generates both intermediate and final output that may be of interest to the user, all contained in the global structures discussed in Section 3.4. The final output can be accessed by declaring the relevant globals before after invoking `SABL`, and the intermediate output can be accessed by making the same declarations in `p_monitor` and then harvesting the quantities of interest to the generic project global structure `P` at the appropriate stage(s). These will then be available after invoking `SABL` once `P` has been declared global.

Each model requires specific inputs in the global structure `M` and these requirements are documented for each model in Section 5.5. Every model requires that the field `M.data`, a data matrix with observations by rows and variables by columns, be created in stage `'startrun'`. Typically data are read from file and then following any required transformations or rearrangement of rows and/or columns `p_monitor` places the result in `M.data` in stage `'startrun'`.

SABL uses rows `C.tfirst` through `C.tlast` of `M.data`. It takes as default values `C.tfirst = 1` and `C.tlast = size(M.data, 1)`. The user can designate an alternative range directly in stage `'startrun'`.

Each model also requires the user to designate the columns in `M.data` corresponding to the observables of that model. For example, model `normal` is

$$y_t \sim N\left(\beta' x_t, \gamma' z_t\right) \quad (t = 1, \ldots, T)$$

where the outcomes y_t are independent conditional on $(x_t, z_t; t = 1, \ldots, T)$. In stage `'startrun'` the user must designate the scalar `M.y_pointer` pointing to the column of `M.data` containing y_t $(t = 1, \ldots, T)$, the column vector `M.x_pointer` pointing to the columns of `M.data` containing x_t $(t = 1, \ldots, T)$ and the column vector `M.z_pointer` pointing to the columns of `M.data` containing z_t. $(t = 1, \ldots, T)$. All models use these naming conventions for pointers corresponding to the notation that describes the model.

Users or others who need access to model code (or are simply curious) can find it in the `models/[modelname]` directory of the SABL toolbox. Because Matlab places user code above toolbox code in the path it searches to find invoked functions the user can modify model code by copying the relevant model file to an appropriate directory and making sure that directory is in the Matlab path. Doing this successfully requires a solid understanding of how the model code works, since otherwise it can lead to errors that are difficult to trace or (even worse) detect.

5.1.2 Parameter maps

`help parametermaps`

Each model employs a map from the SABL algorithm vector θ described at the start of Section 2 to the model parameter vectors (for example, β and γ in the case of model `normal`). Many models, including the simpler ones, have a default map and when this is the case then this map is described in the detailed model description in Section 5.5. For example, in model `normal` the default map is from the first `length(M.x_pointer)` elements of θ to β and from the next `length(M.z_pointer)` elements of θ to γ. If this structure is also suited to the user's specification then the user need do noting further about parameter maps.

More complex models, for example full information inference (`M.specification = 'FI'`) in the multivariate normal model `MVN`, require the user to construct an explicit map, and this is an option in any model when the user's specification does not match the default specification. The map must be declared by the user in stage `'startrun'` of `p_monitor`. For each parameter vector of the model there are one or more arrays that provide the map from θ to that parameter vector. These arrays are all in the structure `M`, and each model has names of the structures in `M` that specify the map(s).

For each model parameter map(s) are required if the user does not employ the default model map. Each model parameter vector or matrix has an associated vector or matrix `map`. In the case of model `normal` the required instances are `M.betmap` and `M.gammap`. There are four degrees of flexibility in map specification. In what follows, if the parameter matrix is a vector then it is a column vector, meaning that arguments `(i,j)` in the description could be replaced with `(i)` or `(i,1)`.

1. In the simplest specification `map(i,j)` designates the column of θ that corresponds to element (i, j) of the parameter matrix. This is how model `normal` handles the default case. Notice that this simplest specification can also be used to specify cross-restrictions of the form $\beta_i = \gamma_j$: for example, `M.betmap(2) = 3`, `M.gammap(3) = 3` in model `normal`. This requires that the elements of `map` all be integers between 1 and `C.parameters` (the number of elements of θ; equivalently, the number of columns of `Cpar.theta`) inclusive.

2. If `map(i,j) = 0` then element i of the parameter vector is identically zero. This is unnecessary in a simple model like `normal` because the same result can be achieved by omitting element i of the corresponding `M.*_pointer` vector. In other models, for example the full information specification of the multivariate normal model (`MVN`) it is critical because this implements the use of identifying and over-identifying exclusion restrictions in the model parameters.

3. If `map(i,j)` $<$ `0` then element i of the parameter vector is a linear combination of the elements of θ. The linear combinations are stored in an array `linc`, and column `-map(i)`contains the linear combination for element i. (If all elements of `map` are nonnegative then the array `linc` need not exist and will be ignored if

it does. The number of columns in `linc` must be at least `-min(min(map))`. The number of rows in `linc` is one more than the number of elements of θ; equivalently,

$$\texttt{size(linc, 1)} = \texttt{size(Cpar.theta, 2)} + 1 = \texttt{C.parameters} + 1.$$

SABL constructs the particles corresponding to element (i, j) of the parameter matrix as

```
Cpar.theta*linc(1:C.parameters,-map(i,j)) + ...
        linc(C.parameters+1,-map(i,j).
```

4. If `map(i,j)` $>$ `C.parameters` then element i is a nonlinear function of the elements of θ. If `map(i,j)` $>$ `C.parameters` for any i, then SABL provides the following 6 inputs to a Matlab or C function written by the user that computes the nonlinear transformations:

   - `theta` The `C.JN` $\times$ `C.parameters` matrix of particles
   - `name` This is a name assigned by the model indicating which map is being constructed. The documentation for each model provides the names. For example, in model `normal`, `name = 'bet'` for the map `M.betmap` and `name = 'gam'` for the map `M.gammap`.
   - `ii` is a vector with one entry for each model parameter that is a nonlinear function of θ; its length is `sum(sum(map>npars))`. The entry `ii(r)` is the row i of `map` leading to the need to compute the r'th nonlinear transformation.
   - `jj` is a vector with one entry for each model parameter that is a nonlinear function of θ. The entry `jj(r)` is the column j of `map` leading to the need to compute the r'th nonlinear transformation.
   - `columns` is a vector with one entry for each model parameter that is a nonlinear function of θ. The entry `columns(r)` is the column of `A_in` that is missing (i.e., is entirely zeros) on input that will be filled out using the r'th nonlinear transformation.
   - `A_in` is the matrix of model parameters with `C.JNwork` rows. At input, the columns of `A` that result from linear transformations of θ are already filled. The rest – those requiring nonlinear transformations – are identically zero.

   The output `A` of the function is the same as `A_in` but with the columns requiring nonlinear transformations now filled. Good practice in coding the function is to set `A=A_in` at the start and then fill in the nonlinear transformations.

   (a) If computations are carried out using CPU cores (i.e., `E.gpu = false`) then the function must consist of Matlab code and take the name `p_thetapar.m`. Thus,

   ```
   A = p_thetapar(theta, name, ii, jj, columns, A_in).
   ```

(b) If the computations are carried out using one or more GPUs (i.e., `E.gpu = true`) then the function may consist of C/CUDA code and take the name `p_thetapar_CUDA`. Thus,

```
A = p_thetapar_CUDA(theta, name, ii, jj, columns, A_in).
```

The user must set `C.nlcuda = true` if using `p_thetapar_CUDA`. (The default value is `false`.)

(c) If `E.gpu = true` the user may, alternatively, provide the function evaluations using the Matlab function `p_thetapar.m`. This facility exists so that users who are not experienced with C/CUDA may still use GPUs when nonlinear transformations are included. However, this alternative is generally less efficient than (b). This difference may become quite important if the map from θ to model parameters is computationally intensive and dominates the computatons required to evaluate the likelihood functions. Examples include more involved dynamic stochastic general equilibrium models, game-theoretic models with complex solutions, and complex state space models.

Example 6 *In the SABL toolbox* ***projects/normal_AR3*** *provides an example of a nonlinear map from algorithm parameters θ to the parameters of the normal model in a study of business cycle dynamics. It includes code for* ***p_thetapar*** *and* ***p_thetapar_CUDA****.*

5.1.3 Prediction

Each model in SABL provides the facility for sampling from the model's distribution of observables conditional on covariates and parameters, which can be expressed generically as (25), page 16 in Section 2.6: $y_{ij}^* \sim p\left(\widetilde{y} \mid \theta_{ij}.x^*\right)$. The function `[modelname]sim` (e.g., `normalsim`) provides the code in each case. The input and output arguments are similar from one model to another, but there are some exceptions, so the user should consult the `help` documentation in each case, e.g. `help normal normalsim`. The SABL algorithm does not require this facility. SABL includes these functions because they are useful in prediction and simulating from the prior and posterior predictive distributions as explained in Section 2.6.

5.1.4 On model specification

In all but the simplest cases, it can be difficult to determine what events a model renders more and less probable (or impossible) simply from the analytical form of the conditional distribution of observables (25) and the prior density $p\left(\theta\right)$. In principle, these probabilities for functions of observables can be determined from the prior predictive distribution represented discussed in Section 2.6 by repeating the sequence

$$\theta_{ij} \sim p\left(\theta_{ij}\right), \;\; y_{ij}^* \sim p\left(\widetilde{y} \mid \theta_{ij}.x^*\right), \;\; g_{ij} = g\left(y_{ij}^*\right)$$

for $i = 1, \ldots, N$; $j = 1, \ldots, J$, g being an interesting and relevant vector function. For example, in a macroeconomic application g might include the number of business cycles per century and the relative lengths of upturns and downturns; in an industrial organization application g might include post-merger price increases. For fuller discussion see Geisser (2004) and Geweke (2005, Section 8.3.1).

The user may construct these distributions directly in SABL. This is especially useful at the start of a project in examining prospective model specifications and alternative priors. In stage `'initialize'` the particles θ_{ij} represent the prior distribution of θ. In `p_monitor`, stage `'initialize'` the user executes the following steps.

1. Select either a subset of the particles (`randperm` is useful here) or all the particles.
2. For each of the selected particles simulate one (or more, if all the particles were selected) draws from the distribution of observables conditional on the sample covariates using the function `[modelname]sim`.
3. Compute the functions of interest of the simulated observables that will be used to study the implications of the specified model, and present them in a format useful for understanding and iterating on model specification (graphical presentations are a favorite).

An instance of this procedure is included in `projects/poisson_example/p_monitor`. On completion of the steps it may be useful to insert the command `pause`, to provide time to study the results. The user may then proceed with the SABL algorithm or halt execution to reconsider the model specification. This procedure can be especially useful in complex models in determining whether the specified model precludes aspects of observed behavior that are deemed important for the purposes at hand: if so, better to stop at this point than to proceed with resource-demanding computing, analysis, or perhaps even data collection.

5.2 Prior and initial distributions

`help priors`

`SABL` is structured to work with generic prior and initial distributions, facilitating both the future expansion of SABL priors and the incorporation of custom priors by modellers and users.

5.2.1 Priors in SABL 2015a

`SABL 2015a` provides seven different prior distributions directly and makes it straightforward for modellers or users to incorporate others – The distributes currently provided are

- the beta distribution (`beta`, univariate);

- the Dirichlet distribution (`dirichlet`, multivariate);
- the gamma distribution (`gamma` , univariate);
- the Laplace distribution (`laplace` , univariate);
- the linear distribution comprising both the Gaussian and Student-t distributions (`linear`, univariate and multivariate);
- the uniform distribution (`uniform`, univariate and multivariate);
- the Wishart distribution (`wishart`, multivariate).

The subsections of Section 5.3 provide detailed descriptions.

5.2.2 Customized and extended prior distributions

SABL includes two other named prior distributions, `model` and `custom`. These are handles to code directly provided by a modeler (`model`) or a user (`custom`) that can be used just like the named prior distributions in `SABL 2015a`. See Section 5.4.2 for details; alternatively, `help modelbrief` or `help custombrief`.

Any univariate distribution can be mixed with a discrete distribution: i.e., a distribution for which $P(x = a_i) = p_i$ for a finite set of points $(i = 1, \ldots n)$ and then the balance the distribution (i.e., with probability $1 - \sum_{i=1}^{n} p_i$) is the continuous univariate distribution. See Section 5.4.4 for details; alternatively, `help priormixed`.

Any univariate distribution and any multivariate linear distribution can be truncated by linear constraints. See Section 5.4.5 for details; alternatively, `help priorconstraints`.

5.2.3 Organization of the prior distributions

Associated with each prior distribution there are three functions
`u_prior_[name]setup` (creates the prior distribution)
`u_prior_[name]sim` (simulates from the prior distribution)
`u_prior_[name]` (evaluates the prior probability density function)
For example in the case of the gamma distribution the three functions are

`u_prior_gammasetup`,`u_prior_gammasim` and `u_prior_gamma`.

The SABL user need only be concerned with the first one, invoked in `p_monitor` stage `'startrun'` as

```
prior = u_prior_[name]setup(prior).
```

Specification of the exact form of the prior distribution entails creating and filling some fields of `prior` before invoking `u_prior_[name]setup`. On output `prior` contains additional fields with values that are subsequently used in the execution of `SABL`. Invoking `help u_prior_[name]setup` provides detailed description of both inputs and

outputs. The input fields may include the structures `constraints` and/or `mixed` (see Sections 5.4.4 and 5.4.5). The other input fields are specific to the distribution, and the following section provides some mathematical details that are not practical to include in the SABL `help` facility.

5.3 Prior distributions in SABL 2015a

`help priors`

This section links the input fields of prior to the probability density functions.

5.3.1 Beta prior distribution

`help betabrief`

This distribution is univariate. The prior density has support on $x \in (0, 1)$,

$$p(x \mid a, b) = [B(a, b)]^{-1} x^{a-1} (1 - x)^{b-1}.$$

On input `prior` specifies either

- a and b (scalar fields `a` and `b`) or
- the mean and standard deviation of the distribution (scalar fields `mean` and `std`).

If the beta prior distribution is not part of a default model specification then the scalar control field `columns` provides the column of `Cpar.theta` to which the prior distribution pertains.

5.3.2 Dirichlet prior distribution

`help dirichletbrief`

This distribution is multivariate. The prior density has support on the n-dimensional unit simplex $x_i > 0$ $(i = 1, \ldots, n)$ and $\sum_{i=1}^{n} x_i = 1$,

$$p(x \mid a) = [B(a)]^{-1} \prod_{j=1}^{n} x_i^{a_i - 1}$$

where $B(a)$ is the multivariate beta function

$$B(a) = \left[\Gamma \left(\sum_{j=1}^{n} a_j \right) \right]^{-1} \prod_{i=1}^{n} \Gamma(a_i).$$

defined for $a_i > 0$ $(i = 1, \ldots, n)$. On input `prior` specifies this distribution by providing either

- the $n \times 1$ vector a (vector field `a`) or

- n and a single number a for which $a_1 = \ldots a_n = a$ (scalar fields `n` and `a`)

If the Dirichlet prior distribution is not part of a default model specification then the vector control field `columns` provides the n columns of `Cpar.theta` to which the prior distribution pertains.

5.3.3 Gamma prior distribution

(`help gammabrief`

This distribution is univariate. The prior density has support $x \in (0, \infty)$,

$$p(x \mid k, \theta) = \left[\Gamma(k)\,\theta^k\right]^{-1} x^{k-1} \exp(-x/\theta),$$

where k is the shape parameter and θ is the scale parameter. On input `prior` specifies this distribution in one of four ways by providing either

- the shape parameter k and scale parameter θ (scalar fields `shape` and `scale`),
- the shape parameter k and rate parameter $b = 1/\theta$ (scalar fields `shape` and `rate`),
- the mean and standard deviation of the distribution (scalar fields `mean and std`), or
- the degrees of freedom parameter ν and scale factor s^2 in the alternative representation of the prior distribution

  $$s^2 x \sim \chi^2(\nu)$$

  (the fields `chi2df` and `scale`).

If the gamma prior distribution is not part of a default model specification then the scalar control field `columns` provides the column of
`Cpar.theta` to which the prior distribution pertains.

5.3.4 Laplace prior distribution

`help laplacebrief`

This distribution is univariate. The prior density has support $x \in (-\infty, \infty)$,

$$p(x \mid \mu, b) = (\lambda/2) \exp(-\lambda |x - \mu|),$$

where μ is the location (or mean) parameter and λ is the diversity parameter. On input `prior` provides either

- μ and λ (the scalar fields `mean` and `diversity`) or
- the mean and standard deviation of the distribution (scalar fields `mean` and `std)`.

If the Laplace prior distribution is not part of a default model specification then the scalar control field `columns` provides the column of `Cpar.theta` to which the prior distribution pertains.

5.3.5 Linear prior distribution

`help linearbrief`

This distribution is univariate or multivariate. The prior distribution is either normal (indicated by the absence of the field `df`) with prior probability density

$$p\left(x \mid \mu, \Sigma\right) = \left(2\pi\right)^{-n/2} \left|\Sigma\right|^{-1/2} \exp\left[-\left(x-\mu\right)' \Sigma^{-1} \left(h-\mu\right)/2\right] \tag{27}$$

or Student-t (indicated by the presence of the scalar field `df` set to a positive finite value) with prior probability density

$$\begin{aligned} p\left(x \mid \mu, \Sigma, \nu\right) &= \left[\Gamma\left(\nu/2\right)\right]^{-1} \Gamma\left[\left(\nu+n\right)/2\right] \nu^{-(p/2)} \pi^{-p/2} \left|\Sigma\right|^{-1/2} \\ &\quad \cdot \left[1+\nu^{-1}\left(x-\mu\right)' \Sigma^{-1}\left(x-\mu\right)\right]^{-(\nu+n)/2} \end{aligned} \tag{28}$$

where in each case x is $n \times 1$; μ is the mean parameter vector in (27) and the location parameter vector in (28); Σ is the variance matrix in (27) and the scale matrix in (28); and ν is the degrees of freedom parameter in (28).

In both cases the distribution can be specified by providing either

- the $n \times 1$ vector μ (vector field `mean`) and the $n \times n$ positive definite matrix Σ (matrix field `variance`),
- the $n \times 1$ vector μ (vector field `mean`) and the positive definite matrix Σ^{-1} (matrix field `precision`),
- the $n \times 1$ vector μ (vector field `mean`) and the $n \times 1$ positive vector $\left(\sigma_{11}^{1/2}, \ldots, \sigma_{nn}^{1/2}\right)$ (vector field `std`) with the understanding that $\Sigma = \mathrm{diag}\left(\sigma_{11}, \ldots, \sigma_{nn}\right)$, or
- the $n \times n$ nonsingular matrix R (matrix field `R`), the $n \times 1$ vector r (vector field `r`) such that $Rx - r$ either has a probability density of the form (27) or (28) with $\mu = 0$, and variance $\Sigma = \mathrm{diag}\left(\sigma_{11}, \ldots, \sigma_{nn}\right)$ with $\left(\sigma_{11}^{1/2}, \ldots, \sigma_{nn}^{1/2}\right)$ in vector field `std`.

If the linear prior distribution is not part of a default model specification then the vector control field `columns` provides the n columns of `Cpar.theta` to which the prior distribution pertains.

5.3.6 Uniform prior distribution

`help uniformbrief`

This distribution is univariate or multivariate. If the prior distribution is uniform on the n-dimensional hypercube $\{x : a \leq x \leq b\}$ then the structure `prior` specifies either

- the $n \times 1$ vectors a and b with $b > a$ ($n \times 2$ matrix field `endpoints` $= [a, b]$) or
- the mean of the distribution $(a+b)/2$ and the n-dimensional width of the distribution $b - a$ (vector field `mean` and positive vector field `width`).

5.3.7 Wishart prior distribution

`help wishartbrief`

This distribution is multivariate. The prior density has support on the space of $n \times n$ positive definite matrices X,

$$p(X \mid V) = 2^{(-n\nu)/2} |V|^{-\nu/2} [\Gamma_n(\nu/2)]^{-1} |X|^{(\nu-n-1)/2} \exp\left[\mathrm{tr}\left(V^{-1}X\right)/2\right]$$

in which V is the scale matrix, ν is the degrees of freedom parameter with $\nu > n-1$, and Γ_n is the multivariate gamma function,

$$\Gamma_n(\nu/2) = \pi^{n(n-1)/4} \prod_{j=1}^{n} \Gamma\left(\frac{\nu+1-j}{2}\right).$$

The structure `prior` specifies the degrees of freedom parameter ν (positive scalar field `df`) and either

- the $n \times n$ positive definite scale matrix V (matrix field `variance`),
- the $n \times n$ positive definite matrix V^{-1} (matrix field `precision`), or
- the positive vector $\left(v_{11}^{1/2}, \ldots, v_{nn}^{1/2}\right)$ (vector field `std`) with the understanding that $V = \mathrm{diag}(v_{11}, \ldots, v_{nn})$.

SABL uses an internal vector representation of X, and any other random positive definite matrix, that may be described as follows:

- Let B be the unique Choleski factorization of X with the properties (a) $X = B'B$, (b) B is upper triangular, and (c) the diagonal elements of B are all positive.
- Construct C from B by taking $c_{ij} = b_{ij}$ $(j > i;\ i = 1, \ldots, n)$ and $c_{ii} = \log(b_{ii})$ $(i = 1, \ldots, n)$.
- Let θ be the row-major vectorization of C omitting the below-diagonal elements (all identically zero).

For compactness and subsequent reference denote this sequence of transformations by

$$\theta = c(X), \tag{29}$$

where X is any $n \times n$ positive definite matrix and θ is $n(n+1)/2 \times 1$.

If the Wishart prior distribution is not part of a default model specification then the vector control field `columns` provides the $n(n+1)/2$ columns of `Cpar.theta` providing this internal representation of X to which the prior distribution pertains.

5.4 Specifying model prior distributions

`help priors`

The used can specify prior distributions in three levels of increasing generality. As with any software, less generality implies more compact and simpler specification; more generality implies more verbose and complex specification.

5.4.1 Model specific default prior specifications

Some models in SABL, mostly simpler ones, have default structures for the map from the algorithm parameter vector θ to the model parameter vector (Section 5.1.2) and for the prior distribution. The documentation for each model details these default structures, which minimize the amount of work required by the user but (of course) provide limited flexibility in model and prior specifications.

For example the model `normal` is $y_t \mid (x_t, z_t) \sim N\left(\beta' x_t,\ \exp\left(\gamma' z_t\right)\right)$ where the outcome variables y_t are mutually independent conditional on (x_t, z_t) $(t = 1, \ldots, T)$. In the default structure $\theta' = (\beta', \gamma')$ with the length of β inferred from the user's specification of `M.x_pointer`, the columns of `M.data` corresponding to x_i, and similarly for γ, `M.z_pointer` and z_t. The prior distribution has two independent components, one for β and one for γ. Each can be linear or Laplace. The user specifies these distributions by creating the structures `M.betprior and M.gamprior` as described for the normal distribution 5.3.5 or the Laplace distribution in Section 5.3.4 For example, if β has two independent elements and in the prior distribution $\beta_1 \sim N(1, 1)$ and $\beta_2 \sim N(0, 1)$ then

```
M.betprior.distribution = 'normal';
M.betprior.mean = [1; 0];
M.betprior.std = ones(2, 1);
M.betprior = u_prior_linearsetup(M.betprior)
```

5.4.2 Tailoring prior specifications using prior provided in SABL

Internally, SABL represents prior distributions by means of a cell array `M.prior`; the name is the same for all models. Each cell entry is a structure, whose fields include all of those discussed for specific prior distributions in Section 5.3. The `u_prior_[priorname]setup` functions create additional fields. SABL adds to this the field `columns`, a vector indicating the elements of θ (columns of `Cpar.theta`) to which the prior distribution applies. For the default prior in model `normal` SABL uses `M.prior{1}` for β, adding the field `columns = 1:length(M.x_pointer)`, and uses `M.prior{2}` for γ, adding the field `columns = length(M.x_pointer)+1:length(M.x_pointer)+length(M.y_pointer)`.

The user may access this internal representation directly – and must, unless he/she creates all default prior fields (`M.betprior` and `M.gamprior` in the case of model `normal`); or, if the model does not have default prior distributions. The structure for an entry `M.prior{j}` may be created by means of the relevant `u_prior_[priorname]setup` func-

tion if one of the prior distributions described in Section 5.3 is used. The user must add the `columns` field.

5.4.3 Customizing all or some of the prior distribution

Prior distributions are not limited to those of Section 5.3. Models may supplement these with their own specific distributions, providing a function `m_prior_modelsetup` to create the prior distribution. Using these specifications is very much like using the specifications in Section 5.3. In general the user takes advantage of these model-specific priors by providing other fields as indicated in the model description`u_prior[name]_setup` and then invoking `u_prior[name]_setup` . This makes it straightforward for modelers to exploit the existing infrastructure for management of prior distributions, providing only what is unique to their prior distribution(s).

But the user also has the ability to create any prior distribution at all, so long as the user can provide code that simulates from the prior distribution and evaluates its prior density. The user does this by specifying `M.prior{j}.name = 'custom'` together with the following two functions written in Matab:

- `theta = p_prior_customsim(ipart, a)`: `ipart` is the index of `M.prior{ipart}`; `a` is a prior specification structure created by the user in `p_monitor`, stage `'startrun'`; and the output `theta` has `C.JNwork` rows, each row an independent draw from the prior distribution. SABL will move `theta` to the GPU if `E.gpu = true`. If `E.pus>1` then `p_prior_custom` is executed on each of the `E.pus` threads and SABL does the attendant bookkeeping.

- `logp = p_prior_custom(theta, ipart, a)` evaluates the prior density: `theta` is the matrix providing the points of evaluation; the input arguments `ipart` and `a` are as for `p_prior_customsim`.

The user may wish to write an accompanying function `p_prior_customsetup` for their own convenience, but this is not necessary. SABL never invokes a function with this name directly.

5.4.4 Specifying mixed prior distributions and constraints

`help priormixed`

Any univariate prior distributions becomes mixed discrete-continuous if the prior structure `prior` contains the field `mixed`. The mixture is described by two fields of the structure `prior.mixed`:

- The scalar field `a` is the vector a with mass points $a_1, \ldots, a_n$. They must all be in the support of the distribution.

- The scalar field `p` provides the vector of corresponding probabilities $p_1, \ldots, p_n$, specifying $P(x = a_i) = p_i$. They must be positive with $\sum_{i=1}^{n} p_i < 1$.

The continuous part of the distribution is then given by the univariate prior distribution and the total probability associated with the continuous part is $1 - \sum_{i=1}^{n} p_i$. Violation of these conditions on the a_i and p_i, or an attempt to make a multivariate distribution mixed discrete-continuous produces a terminating error.

5.4.5 Specifying constraints for prior distributions

`help priorconstraints`

Any univariate prior distribution, as well as the multivariate `linear` prior distribution becomes truncated if the prior structure `prior` contains the field `constraints`. The constraints have the form

$$a \leq Dx \leq b$$

where x is $n \times 1$, a and b are $r \times 1$ with $a_i < b_i$ $(i = 1, \ldots r)$ and D is $r \times n$ of rank r. These are in the fields `a`, `b` and `D`, respectively, of the structure `prior.constraints`. Values $a_i = -\infty$ and $b_i = +\infty$ are permitted. For univariate distributions it is simplest to set the field `D = 1`. Failure to respect $a < b$ or $\text{rank}\,(D) = r$ produces a terminating error.

5.5 Models in SABL 2015a

These are the models currently offered in SABL. Future editions will include substantially more.

5.5.1 The normal model

To invoke the normal model, use `SABL('normal')` from the project directory, or

```
SABL('normal', [project_directory])
```

from any directory.

Observables distribution Let x_t and z_t be vectors of covariates and y_t the corresponding outcome. The outcomes y_t $(t = 1, \ldots, T)$ are mutually independent conditional on (x_t, z_t) $(t = 1, \ldots, T)$:

$$y_t \sim N\left(\beta' x_t,\ \exp\left(\gamma' z_t\right)\right).$$

The user provides the model data through the required model specification control fields

- `M.data`: $T \times d$ data matrix;
- `M.x_pointer`: A length k_x vector pointing to columns of `M.data` (defines x_t);
- `M.z_pointer`: A length k_z vector pointing to columns of `M.data` (defines z_t);
- `M.y_pointer`: A scalar pointing to a column of `M.data` (defines y_t).

Parameter map (See Section 5.1.2.) In general β and γ are each an arbitrary function of the algorithm parameter vector θ. The default map is $\theta' = (\beta', \gamma')$. To override the default the user creates *both* the model specification control fields

- `M.betmap`: Part of the parameter map for component β;
- `M.gammap`: Part of the parameter map for component γ.

Consistent with all parameter maps the user may also need to specify

- `M.betlinc`: Part of the parameter map for component β;
- `M.gamlinc`: Part of the parameter map for component γ.

For nonlinear parameter maps model `normal` provides the name `'bet'` for β and `'gam'` for γ as input to `p_thetapar`.

Prior distributions In conjunction with the default parameter map for β and γ the user may invoke either or both of the two default prior distributions

`M.betprior`: Prior distribution for β, must be linear;

`M.gamprior`: Prior distribution for γ, must be linear.

The user may also create the prior distribution for θ directly in the cell array `M.prior` described in Section 5.4.2.

M phase RNE functions By default there are two M phase RNE functions, $\beta'\overline{x}$ and $\gamma'\overline{z}$. The user may substitute custom RNE functions by setting the field `M.rnecustom = true` and providing a function `g = p_rne(theta)` that has as input the particle matrix `theta` and output the corresponding RNE function values `g`.

5.5.2 The Poisson model

To invoke the Poisson model, use `SABL('poisson')` from the project directory, or `SABL('poisson', [project_directory])` from any directory.

Observables distribution Let x_t be a vector of covariates and y_t be the corresponding outcome. The outcomes y_t $(t = 1, \ldots, T)$ are independent conditional on x_t $(t = 1, \ldots, T)$:

$$y_t \sim \text{Poisson}\left(\lambda_t\right), \quad \log\left(\lambda_t\right) = \beta' x_t.$$

The user provides the model data through the required model specification control fields

- `M.data`: $T \times d$ data matrix;
- `M.x_pointer`: A length k_x vector pointing to columns of `M.data` (defines x_t);
- `M.y_pointer`: A scalar pointing to a column of `M.data` (defines y_t.).

Parameter map (See Section 5.1.2) In general β is an arbitrary function of the algorithm parameter vector θ. The default map is $\theta = \beta$. To override the default the user creates the model specification control field

- `M.betamap`: Part of the parameter map for β.

Consistent with all parameter maps the user may also need to specify

- `M.betalinc`: Part of the parameter map for β.

For nonlinear parameter maps model `normal` provides the name `'beta'`as input to `p_thetapar`.

Prior distributions In conjunction with the default parameter map for β the user may invoke the distribution

`M.betaprior`: Prior distribution for β, must be linear;

The user may also create the prior distribution for θ directly in the cell array `M.prior` described in Section 5.4.2.

M phase RNE functions By default there is one M phase RNE function, $\beta'\overline{x}$. The user may substitute custom RNE functions by setting the field `M.rnecustom = true` and providing a function `g = p_rne(theta)` that has as input the particle matrix `theta` and output the corresponding RNE function values `g`.

5.5.3 The negative binomial model

To invoke the Poisson model, use `SABL('negative_binomial')` from the project directory, or `SABL('negative_binomial', [project_directory])` from any directory.

Observables distribution Let x_t be a vector of covariates and y_t be the corresponding outcome. The outcomes y_t $(t = 1, \ldots, T)$ are mutually independent conditional on (x_t, z_t) $(t = 1, \ldots, T)$:

$$\begin{aligned} y_t &\sim \mathrm{NB}\left(r_t, p_t\right), \\ \log E\left(y_t\right) &= \log\left(\frac{r_t p_t}{1-p_t}\right) = \beta' x_t, \\ \log\left[\frac{\mathrm{var}\left(y_t\right)}{\mathrm{E}\left(y_t\right)} - 1\right] &= \log\left(\frac{p_t}{1-p_t}\right) = \gamma' z_t. \end{aligned}$$

The function of moments $\mathrm{var}\left(y_t\right)/\mathrm{E}\left(y_t\right) - 1$ is the overdispersion coefficient, which has support $(0, \infty)$ in the negative binomial distribution. (In the Poisson distribution the overdispersion coefficient is 0.) The inverse map is therefore $r_t = \exp\left(\beta' x_t - \gamma' z_t\right)$, $p_t = \left[1 + \exp\left(-\gamma' z_t\right)\right]^{-1}$.

The user provides the model data through the required model specification control fields

- `M.data`: $T \times d$ data matrix;
- `M.x_pointer`: A length k_x vector pointing to columns of `M.data` (defines x_t);
- `M.z_pointer`: A length k_z vector pointing to columns of `M.data` (defines z_t);
- `M.y_pointer`: A scalar pointing to a column of `M.data` (defines y_t.).

Parameter map (See Section 5.1.2.) In general β and γ are each an arbitrary function of the algorithm parameter vector θ. The default map is $\theta' = (\beta', \gamma')$. To override the default the user creates *both* the model specification control fields

- `M.betmap`: Part of the parameter map for component β;
- `M.gammap`: Part of the parameter map for component γ.

Consistent with all parameter maps the user may also need to specify

- `M.betlinc`: Part of the parameter map for component β;
- `M.gamlinc`: Part of the parameter map for component γ.

For nonlinear parameter maps model `normal` provides the name `'bet'` for β and `'gam'` for γ as input to `p_thetapar` (see Section 5.1.2).

Prior distributions In conjunction with the default parameter map for β and γ the user may invoke either or both of the two default prior distributions

`M.betprior`: Prior distribution for β, must be linear;

`M.gamprior`: Prior distribution for γ, must be linear.

The user may also create the prior distribution for θ directly in the cell array `M.prior` described in Section 5.4.

M phase RNE functions By default there are two M phase RNE functions, $\beta'\overline{x}$ and $\gamma'\overline{z}$. The user may substitute custom RNE functions by setting the field `M.rnecustom = true` and providing a function `g = p_rne(theta)` that has as input the particle matrix `theta` and output the corresponding RNE function values `g`.

5.5.4 The multivariate normal model, full information specification

To invoke the MVN model specification FI (hereafter MVN-FI), use `SABL('MVN')` from the project directory, or `SABL('MVN', [project_directory])` from any directory, and set `M.specification = 'FI'` in `p_monitor` stage `'startrun'`.

Observables distribution Let x_t and z_t be vectors of covariates and let y_t be the corresponding vector of outcomes. In MVN-FI

$$y_t = Ay_t + Bx_t + \varepsilon_t,\ \varepsilon_t \sim N(0, \Sigma). \tag{30}$$

The vectors y_t and ε_t are $n \times 1$ and the vector x_t is $k \times 1$; y_t and x_t are observable. The disturbances ε_t $(t = 1, \ldots, T)$ are mutually independent conditional on x_t $(t = 1, \ldots, T)$. The matrix A is $n \times n$ and the matrix B is $n \times k$. Usually the diagonal elements of A are all zero, but they need not be; the prior distribution must imply that the matrix $I_n - A$ is nonsingular with probability 1. Note that MVN-FI implies that

$$y_t \sim N\left((I - A)^{-1} Bx_t,\ \ (I - A)^{-1} \Sigma (I - A)^{-1\prime}\right) \tag{31}$$

and that conditional on x_t $(t = 1, \ldots, T)$ the observables y_t are mutually independent. As with all positive definite matrices in SABL Σ^{-1} is represented by $c(\Sigma)$ (29) in (30) and (31), and priors for Σ must address $c(\Sigma)$. The Wishart prior distribution for Σ^{-1} (Section 5.3.7), equivalently the inverse Wishart prior distribution for Σ, does this directly and conveniently.

The user provides the model data through the required model specification control fields

- `M.data`: $T \times d$ data matrix;
- `M.x_pointer`: A length k vector pointing to columns of `M.data` (defines x_t);
- `M.y_pointer`: A length n vector pointing to columns of `M.data` (defines y_t).

Parameter map (See Section 5.1.2.) There are no default parameter maps in MVN-FI. The model parameter vector consists of all of the elements of A, B and H that are not identically zero. Each of these can be an arbitrary function of the algorithm parameter vector θ, implemented using the parameter map capability of SABL. The parameter map arrays are

- `M.Amap` (an $n \times n$ matrix) for A together with `M.Alinc` if required. For nonlinear maps the name is `'A'`;
- `M.Bmap` (an $n \times k$ matrix) for B together with `M.Blinc` if required. For nonlinear maps the name is `'B'`;
- `M.Hmap` (an $n \times n$ upper triangular matrix) for H together with `M.Hlinc` if required. For nonlinear maps the name is `'H'`.

Thus SABL can be applied to any model of the form

$$y_t = A(\theta)\, y_t + B(\theta)\, x_t + \varepsilon_t,\ \varepsilon_t \overset{iid}{\sim} N\left(0,\ \ (c(f(\theta)))^{-1}\right). \tag{32}$$

This includes the very large class of structural models that are linearized when applied to data. Of course, one must be able to code the functions in (32), and for these models this may be a demanding analytical and programming task in its own right using SABL or any other econometric approach. The practicality and efficiency of SABL in this dimension can only be determined on a case-by-case basis.

Prior distributions Since there is no default parameter map, there is no default prior specification. Perhaps the simplest situation arises when $\theta' = (\theta_1', \theta_2')$, A and B are linear functions of θ_1 ($n_1 \times 1$), both with linear prior distributions, and Σ is independent with an inverse Wishart distribution. Then `M.prior` may have a few as two cells: the first cell pertains to θ_1, which has a linear prior distribution, and the $n_1 \times 1$ vector `columns` field points to θ_1. The second cell pertains to $\theta_2 = c(\Sigma^{-1})$, for which the prior distribution is Wishart and the $n(n+1)/2 \times 1$ `columns` field points to the internal representation of Σ^{-1} (Section 5.3.7).

Of course this simple situation need not apply, and specifically does not at the level of generality in (32). The ability to employ prior distributions that go beyond those described in Section 5.3 by means of the functions `m_prior_custom` and `p_prior_custom` described in Section 5.4.3 implies that SABL is limited mainly by the creativity and coding skills of the modeler or user. The practicality and efficiency of SABL in this dimension can only be determined on a case-by-case basis.

M phase RNE functions By default the M phase RNE functions are the algorithm parameters θ themselves. The user can provide `g = p_rne(theta)`that has as input the particle matrix `theta` and output the corresponding RNE function values `g`, if a different set of RNE functions is desired. It is best to avoid intensive computations in `p_rne`, since the results are not used elsewhere in SABL.

5.5.5 The EGARCH model

To invoke the EGARCH model, use `SABL('egarch')` from the project directory, or `SABL('egarch', [project_directory])` from any directory.

Observables distribution Let y_t ($t = 1, \ldots, T$) denote an observed sequence of asset returns. The EGARCH model is

$$y_t = \mu_Y + \sigma_Y \exp\left(\sum_{k=1}^{K} v_{kt}/2\right) \varepsilon_t$$

where $\mu_Y = \mathrm{E}(y_t)$ and $\sigma_Y^2 = \mathrm{var}(y_t)$. The K volatility factors are

$$v_{kt} = \alpha_k v_{k,t-1} + \beta_k \left(|\varepsilon_{t-1}| - (2/\pi)^{1/2}\right) + \gamma_k \varepsilon_{t-1} \quad (k = 1, \ldots, K).$$

The disturbance terms ε_t are independent and identically distributed as a full mixture of I normal distributions

$$p(\varepsilon_t) = \sum_{i=1}^{I} p_i \phi(\varepsilon_i; \mu_i, \sigma_i^2); \;\; \mathrm{E}\,(\varepsilon_t) = 0, \;\; \mathrm{var}\,(\varepsilon_t) = 1.$$

The user must specify the fields `M.K` $= K$ and `M.I` $= I$, as well as `M.y_pointer` indicating the column of `M.data` containing the time series y_t $(t = 1, \ldots, T)$.

Parameter map (See Section 5.1.2.) The default parameter map in the `egarch` model consists of the transformations used in Durham and Geweke (2014), Section 4. In these transformations the algorithm parameter vector is $\theta' = \left(\theta^{(1)}, \ldots, \theta^{(8)\prime}\right)$. The map from algorithm parameters to model parameters begins with

$$\mu_Y = \theta^{(1)}/1000, \quad \sigma_Y = \exp\left(\theta^{(2)}\right),. \tag{33}$$

Then for $k = 1, \ldots, K$,

$$\alpha_k = \tanh\left(\theta_k^{(3)}\right), \quad \beta_k = \exp\left(\theta_k^{(4)}\right), \quad \gamma_k = \theta_k^{(5)}. \tag{34}$$

For $i = 1, \ldots, I$,

$$p_i^* = \tanh\left(\theta_i^{(6)}\right), \quad \mu_i^* = \theta_i^{(7)}, \quad \sigma_i^* = \exp\left(\theta_i^{(8)}\right), \tag{35}$$

and then

$$\begin{aligned} p_i &= \frac{p_i^*}{\sum_{j=1}^{I} p_i^*}, \;\; \mu_i^{**} = \mu_i^* - \sum_{j=1}^{I} p_j \mu_j^*, \\ c &= \left\{ \sum_{j=1}^{I} p_j \left[\left(\mu_j^*\right)^2 + \left(\sigma_j^*\right)^2 \right] \right\}^{-1/2}, \\ \mu_i &= c\mu_i^{**}, \;\; \sigma_i = c\sigma_i^*. \end{aligned}$$

The alternative to the default parameter map entails the specification the following 8 fields by the user:

`M.muYmap` (a scalar) for μ_Y together with `M.muYlinc` if required. For nonlinear maps the name is `'muY'`;

`M.sigmaYmap` (a scalar) for σ_Y together with `M.sigmaYlinc` if required. For nonlinear maps the name is `'sigmaY'`;

`M.alphamap` (a $K \times 1$ vector) for α together with `M.alphalinc` if required. For nonlinear maps the name is `'alpha'`;

`M.betamap` (a $K \times 1$ vector) for β together with `M.betalinc` if required. For nonlinear maps the name is `'beta'`;

`M.gammamap` (a $K \times 1$ vector) for γ together with `M.gammalinc` if required. For nonlinear maps the name is `'gamma'`;

`M.pmap` (an $I \times 1$ vector) for p together with `M.plinc` if required. For nonlinear maps the name is `'p'`;

`M.mumap` an $I \times 1$ vector) for μ together with `M.mulinc` if required. For nonlinear maps the name is `'mu'`;

`M.sigmamap` (an $I \times 1$ vector) for σ together with `M.sigmalinc` if required. For nonlinear maps the name is `'sigma'`.

The user must specify all 8 fields or none of them. If none of the fields are specified then `egarch` uses the default parameter map. Users creating their own map may find it useful to review the code in `models/egarch/p_thetapar` for the logic employed.

Prior distributions The `egarch` model has a default prior structure that can be used only in conjunction with the default parameter map – but need not be: the user can employ an alternative prior by constructing the cell array of prior structures `M.prior`. In the default prior, each of the vectors $\theta^{(i)}$ in (33) - (35) normal and independent of $\theta^{(j)}$ $(j \neq i)$, each element having the same mean and standard deviation, and a diagonal variance matrix. Correspondingly there are eight structure fields `M.muYprior`, `M.sigmaYprior`, `M.alphaprior`, `M.betaprior`, `M.gammaprior`, `M.muprior`, `M.ppriior`, and `M.sigmaprior`, each of these structures with fields `mean` and `std`.

M phase RNE functions The default RNE functions are μ_Y and $\log \sigma_Y$.

5.6 Adding a new model to the SABL toolbox

The SABL toolbox accommodates the incorporation of new models. It is straightforward to develop a local research library of models that have full interface with the SABL toolbox and can take advantage of all of its features, so long as the conventions common to all models are respected. This section describes those conventions.

5.6.1 Required functions

The SABL toolbox interface requires an interface of four functions for each model:

- `m_message`: Log likelihood or objective function.
- `m_monitor`: Interface between `c_monitor` and `p_monitor` (see Section 3.3).
- `m_[modelname]sim`: Simulate from the distribution of observables conditional on parameters.
- `m_rnefunctions`: RNE test functions use in the M phase.

New model developers should consult the model directories in the SABL toolbox. Files in these directories provide insight into the management of execution, especially code that uses GPUs or multiple workers efficiently. As with all code development projects, there are substantial advantages to modularity, and in general this entails developing other functions that are invoked directly or indirectly by these four functions.

5.6.2 Interaction with SABL

Before adding a new model to SABL – ideally, at the time the prospect is conceived – the modeler should digest the many commonalities in the four interface functions for current SABL models. This includes conventions for field names in the `M` structure not spelled out explicitly in this document or in the `help` system.

Specific attention must be given to modification of core default values in the `C` structure, documentation of the `M` structure in a corresponding `helpfiles` directory, at least one example project, and tests provided in a `testcode` subdirectory.

The last requirement is the most important. SABL provides checks for analytical and coding errors in `u_pfcheck` and `u_pdfcheck`, as well as functions `u_runtest` and `u_runtest_study` that test the correct integration of the model code with the most important control fields in the `C` and `E` structures. The developers of the SABL toolbox actively encourage such contributions and are happy to consider contributions that meet these standards.

5.6.3 CUDA code

CUDA code for likelihood evaluation is essential to efficient GPU performance with all but the simplest models (in which GPU computing provides little if any advantage over CPU in the first place). CUDA code is contained in a subdirectory `cufiles` of each model directory. The subdirectory contains several similar `mex` files, one each for linux, mac and windows operating systems. Consult the existing CUDA model files for more details. New models may also entail the development of additional library functions: note that the directory `library` also has subdirectories specific to CUDA code.

5.6.4 Code documentation

Each model function should include:

1. Comment lines at the start that then constitute the response to the corresponding `help` command, in the format common to all SABL functions;

2. A model subdirectory that contains

   (a) One file for each field of the `M` structure in that model, using the convention that the symbol `Q` replaces the dot (`.`) in any field name, in a subdirectory `helpfiles`;

   (b) The file `helpfiles/modelname]brief.m` that provides the structure of the model description returned in response to `help [modelname]brief`.

These conventions assure that the model will be fully incorporated in the `SABL help` system.

References

Box GEP 1980. Sampling and Bayes inference in scientific modelling (with discussion and rejoinder). Journal of the Royal Statistical Society Series A 143: 383 - 430.

Douc R, Cappe P, Moulines E. 2005. Comparison of resampling schemes for particle filtering. 4th International Symposium on Image and Signal Processing and Analysis (`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.126.9118`)

Durham G, Geweke J. 2015. Adaptive sequential posterior simulators for massively parallel computing environments. In: Ivan Jeliazkov, Dale J. Poirier (eds.) Bayesian Model Comparison (Advances in Econometrics, Volume 34) Emerald Group Publishing Limited, Chapter 1, 1-44.

Geisser S. 2004. Predictive Analysis. Encyclopedia of Statistical Sciences.

Geweke J. 2005. Contemporary Bayesian Econometrics and Statistics. Wiley.

Geweke J, Frischknecht B. 2014. Exact optimization by means of sequentially adaptive Bayesian learning. Working paper

Table 1: SABL algorithm

Retrieve particles to be updated (if `E.simulation_get` specified)
Set `SABL` environment (`E` fields)
Stage: `'open'`
`for pass = first, [second if C.twopass = true]`
  Define the problem, including priors and data Stage: `'startrun'`
  Copy global structures from one to multiple workers if E.pus > 1
  Draw particles from prior if not updating Stage: `'initialize'`
  `while SABL cycles = 1, 2, 3, ...`
    Stage: `'startcycle'`
    Initialize C (Correction) phase weight function
    Stage: `'startCphase'`
    `while C phase steps incomplete`
      Update the weight function Stage: `'whileCphase'`
    `end`
    Stage: `'endCphase'`
    Stage: `'startSphase'`
    Execute S (Selection) phase
    Stage: `'endSphase'`
    Stage: `'startMphase'`
    `while M (Mutation) phase steps incomplete`
      Execute the next step Stage: `'whileMphase'`
    `end`
    Stage: `'endMphase'`
  `end`
  Stage: `'finish'`
  Stage: `'endrun'`
  Copy global structures from multiple workers to one if E.pus > 1
`end`
Stage: `'close'`
Save particles for future updating (if `E.simulation_record` specified)