R/C Glue in SamplerCompare

Madeleine Thompson

2012–01–28

1 Introduction

The `SamplerCompare` R package allows both samplers and distributions to be implemented in either R or C. This document describes the interface for implementing them in C and documents the internals of the glue holding it together. It assumes familiarity with the interface for implementing samplers and distributions in R; for more on those interfaces, see the help pages for `make.dist` and `compare.samplers` and the vignette "Introduction to SamplerCompare."

Section 2 describes how to compile and link C code into R so that it can extend `SamplerCompare`; this procedure is the same whether you are implementing a distribution or a sampler. Sections 3 and 4 describe the specifics of implementing distributions and samplers. The remaining sections are intended for a narrower audience: Sections 5 and 6 describe the internals of the glue code. Section 7 describes an alternative interface for implementing samplers, useful when Markov chain iterations are expressible as a sequence of updates.

2 Compiling and linking

Whether you're implementing a sampler or distribution, you must compile it and link it into R before it can be used. If you are not using the R package system, you can do this is with `R CMD SHLIB` and `dyn.load`. Suppose your implementation is in `mycode.c`. At the Unix command line, type:

```
SC_INCLUDE=`echo 'suppressPackageStartupMessages(library(SamplerCompare)) ;
  cat(system.file("include", package="SamplerCompare"))' | R --vanilla --slave`
MAKEFLAGS="CPPFLAGS=-I$SC_INCLUDE" R CMD SHLIB mycode.c
```

The first command locates the `SamplerCompare/include` directory containing the required header file, `SamplerCompare.h`. The second generates a shared object, `mycode.so`. Then, in R, type:

```
dyn.load('mycode.so')
```

At this point, from within R, you can use `wrap.c.sampler` if you've implemented a sampler or `make.c.dist` if you've implemented a distribution. For an example of this procedure that works on both Unix and Windows, you can read `tests/test-indep-mh.R`.

Alternatively, suppose you are using the R package system and the code with your sampler or distribution is in the `src` directory of `MyPackage`. If you are not using a custom `Makefile` and are targeting Unix and MacOS, you can add the following lines to `MyPackage/src/Makevars` so that `R CMD build` can find `SamplerCompare.h`:

```
PKG_CPPFLAGS=-I`echo 'suppressPackageStartupMessages(library(SamplerCompare)) ; \
  cat(system.file("include", package="SamplerCompare"))' | R --vanilla --slave`
```

Be sure no whitespace follows the backslash. Then, to ensure that the shared library created by `R CMD build` is loaded, add the following to `MyPackage/NAMESPACE`:

```
useDynLib(MyPackage)
```

Then, any code in `MyPackage/R` can call `wrap.c.sampler` or `make.c.dist`.

`Makevars` is not used under Windows, so if you want your package to work under Windows, you will also need to create a `Makevars.win` like the following:

```
PKG_CPPFLAGS=-I.../path/to/SamplerCompare/include
PKG_LIBS=$(BLAS_LIBS)
```

3 Implementing a distribution in C

A distribution is defined in C by writing a log density function that follows the `log_density_t` interface, defined in `SamplerCompare.h`. The function has the type:

```
typedef double log_density_t(dist_t *dist, double *x, int compute_grad, double *grad);
```

The distribution itself is represented by a `dist_t`:

```
typedef struct {
  log_density_t *log_dens;   /* log density declared above    */
  SEXP context;              /* opaque context object         */
  int ndim;                  /* dimension of the distribution */
} dist_t;
```

When a sampler wants to evaluate the log density (and optionally its gradient), it calls the `log_density_t` with the first argument set to the `dist_t`, the second equal to an `ndim`-long array of doubles indicating the point at which to evaluate the log density, and the third a boolean indicating whether the gradient is needed. If `compute_grad` is nonzero, the fourth argument, `grad`, is a pointer to an `ndim`-long array of doubles to be filled in with the gradient. The log density itself is returned by the `log_density_t`.

An example density function implementing a one-dimensional standard normal is:

```
#include <R.h>
#include <SamplerCompare.h>

double normal_log_dens(dist_t *dist, double *x, int compute_grad, double *grad) {
  double log_dens;
  if (dist->ndim!=1)
    error("Dimension must be one.");
  log_dens = -0.5 * sqrt(2.0*M_PI) - 0.5 * x[0]*x[0];
  if (compute_grad)
    grad[0] = -x[0];
  return log_dens;
}
```

If the distribution requires external data defined at runtime, the `context` element of `dist_t` may be used; it is defined when calling `make.c.dist`. See the file `distributions.c` in the SamplerCompare source for more examples, including ones that use `context`.

If one cannot or does not want to write code to compute the gradient, the log density function should call the R `error` function if `compute_grad` is nonzero. The resulting distribution will not be usable by samplers that require gradients. Similarly, distributions should not access the memory pointed to by `grad` unless `compute_grad` is nonzero.

Once the log density function has been written in C, it is compiled and linked as described in section 2. An R `scdist` object can then be created with `make.c.dist`, which takes as arguments (among others) the name of the C function and an R object to be passed as the `context` element of the `dist_t` passed as the first argument to the `log_density_t`. An example invocation for the function above is:

```
std.normal.dist <- make.c.dist("Std. Normal", "normal_log_dens",
                               mean=0, cov=as.matrix(1))
```

`std.normal.dist` could then be sampled from as if it had been implemented in R and defined with `make.dist`.

4 Implementing a sampler in C

The other side of the interface of section 3 is the sampler that calls the log density function. The SamplerCompare glue code ensures that R distributions also appear to implement this interface, so if one wants to implement a sampler in C, one only need target that interface.

A C sampler implements the `sampler_t` type, defined in `SamplerCompare.h`:

```
typedef void sampler_t(SEXP sampler_context, dist_t *ds, double *x0,
                       int sample_size, double tuning, double *X_out);
```

`sampler_context` is an opaque context object. `ds` is a `dist_t` representing the distribution to sample from. `x0` is an array of doubles of length `ds->ndim` containing the initial state of the Markov chain. `X_out` is an array of doubles of length `ds->ndim * sample_size`. It should be filled in column-major with the result of the simulation.

To obtain the log density at `x0`, for example, the sampler would call:

```
double y0 = ds->log_dens(ds, x0, 0, NULL);
```

Or, if the sampler wanted a gradient as well:

```
double g0[ds->ndim];
double y0 = ds->log_dens(ds, x0, 1, g0);
```

Then, when it wants to store a new state, using `dcopy_` from BLAS:

```
// ... x_k is (zero-based) state k of the Markov chain
const int one = 1;
dcopy_(&ds->ndim, x_k, &one, X_out+k, &sample_size);
```

As with distributions, samplers implemented in C must be compiled and linked as described in section 2. Then, an R sampler object can be defined with the function `wrap.c.sampler`, which takes arguments specifying the name of a the C function and a context object containing tuning parameters other than the first one. The context object may be any R object; it is the sampler's responsibility to interpret it.

The glue code calls `GetRNGstate` and `PutRNGstate` before and after invoking the sampler, so it is not necessary for the sampler to do so itself. The sampler must, however, call `R_CheckUserInterrupt` periodically to check for user interrupts; once per state transition is usually appropriate.

For a simple example of a complete sampler, implementing independence Metropolis–Hastings with univariate updates, see the file `indep-mh-sampler.c`, which defines a `sampler_t` with the name `indep_mh`. After compiling it as described in section 2, it can be accessed from R by typing:

```
indep.mh.sample <- wrap.c.sampler(sampler.symbol='indep_mh',
                                  sampler.context=0,
                                  name='Independence MH')
```

This method has two tuning parameters: the proposal mean, specified by the sampler context, and the proposal standard deviation, specified as the `tuning` argument to `indep.mh.sample`.

5 Glue used by R samplers calling C distributions

When `make.c.dist` is called to define an R object that represents a log density function implemented in C, it uses `.Call` to call the C function `raw_symbol`, which uses the R internal function `R_FindSymbol` to obtain a function pointer for the log density function. If it cannot be found, an error is reported. Otherwise, `raw_symbol` returns this function pointer so that the linker does not need to be called every invocation. The function pointer is stored in the `sym` element of the `scdist` object as an R raw.

The `log.density` and `grad.log.density` functions in the `scdist` object returned by `make.c.dist` are stubs that call `log.density.and.grad`, which itself uses `.Call` to invoke `R_invoked_C_glue`, a C function that calls the function pointer located by `raw_symbol`.

6 Glue used by C samplers

`wrap.c.sampler` creates an R wrapper function for a `sampler_t`. When this R wrapper is called, it checks whether the distribution has a `c.log.density.and.grad` element, as would occur if the distribution were defined with `make.c.dist`. If so, it uses `.Call` to invoke `sampler_glue_C_dist` to call the sampler. Otherwise, it invokes `sampler_glue_R_dist` to call the sampler.

`sampler_glue_C_dist` uses the `c.log.density.and.grad` string to obtain a function pointer to the C log density function. It then creates a `dist_t` pointing not to that function, but to `C_log_density_stub_func`, which itself obeys the `log_density_t` interface, storing the function pointer and the context object to be passed to the actual sampler in a `C_stub_context_t` object. It then calls the `sampler_t` with this stub distribution function. When invoked by the sampler, the stub distribution function increments a counter for the number of evaluations and possibly for the number of gradients, then invokes the log density function named by the user in `make.c.dist`. This way, C samplers, unlike R samplers, do not need to count function and gradient evaluations.

`sampler_glue_R_dist` works in a similar way, but instead of being passed a string naming a C function, it is passed a `SEXP` referencing an R function implementing the `log.density.and.grad` interface. (See the help page for `make.dist` for more information.) Like `sampler_glue_C_dist`, it passes the sampler a stub `log_density_t`, using the stub function `R_log_density_stub_func`, which in addition to tracking evaluations, deals with packing the C state into R arguments, invoking the R `log.density.and.grad`, and unpacking the results.

When the `sampler_t` invoked from either glue function returns, `X_out`, the evaluations counter, and the gradient counter are packed up into an R list object by their glue function and returned to R wrapper, which returns this list to the user.

7 The transition function interface

One can often express an MCMC update as a sequence of updates by simpler methods. The transition function interface provides an alternative to the `sampler_t` interface that makes it more convenient to implement these simple updates and combine them flexibly.

This interface specifies a single transition of the Markov chain using functions that implement the type:

```
typedef void transition_fn(SEXP sampler_context, dist_t *ds, double *x0,
                           double tuning, double *x1);
```

`sampler_context` is in a special format. `ds` and `tuning` are the target distribution and tuning parameter, as with `sampler_t`. `x0` is the current state. `x1` is a vector in which the next state is returned.

To define a sampler that implements this interface, pass `"transition_sample"` to `wrap.c.sampler` as the sampler symbol. The transition function is specified by passing a function pointer returned by `raw.symbol` or a list whose first element is a function pointer returned by `raw.symbol` to `wrap.c.sampler` as the context. This context will be passed along to the transition function. If the transition function needs additional context, such as extra tuning parameters, it can use the elements of the context list after the first.

Then, if a user wants to perform updates with multiple transition functions each iteration of a Markov chain, they can write a wrapper that calls each of them in sequence and pass a pointer to the wrapper to `wrap.c.sampler`. This way, the individual functions do not have a loop over sample size, so they can be called both individually and in sequence without changes, which I have found useful when experimenting with combinations of methods.