RChurch

Jonathan Malmaud

August 24, 2011

1 Introduction

`RChurch` provide an R interface to Church. Church is a set of Scheme libraries for automatically performing conditional inference on arbitrary models, using rejection sampling or Markov Chain Monte Carlo (MCMC) sampling algorithms. The ability to condition on arbitrary predicates rather than explicitly conditioning on certain nodes in a graphical model taking on particular values, plus the ability to write the data-generating function in a full-featured programming language, gives Church much more flexibility than WinBUGS/JAGS. However, using Church directly requires a knowledge of Scheme and the installation of the Church library and a Scheme interpreter, which may frustrate potential users. Additionally, the output of a Church program can be difficult to analyze and visualize since Scheme's standard library does not include many functions for this purpose and the set of Scheme graphing libraries is very limited.

`RChurch` sets out to solve both these problems. Installing the `RChurch` package requires only a single command on the command line. It automatically installs and configures a scheme interpreter and the Church libraries. It also allows the predicate and data-generating function to be specified in R instead of Scheme, and returns the results of a Church program in a parsed form that is easy to then analyze in R. You are free to use the rich set of R statisical and visualization routines to process the results.

The interface to `RChurch` is modeled after RJags, an interface between R and the JAGS sampler. You need only familiarize yourself with two functions: church.model sets up a Church model, and church.sample draws samples from that model. Various options can be set through these functions, such as whether to use Bher or MIT-Church. Consult the help files of these functions for a comprehensive treatment.

1.1 Installation

In the terminal, execute the command `R CMD INSTALL RChurch.tar.gz` from the directory where the package was downloaded to. You might need root permissions. Example:

```
cd('~/Downloads')
sudo R CMD INSTALL RChurch.tar.gz
```

```
(mh-query 100 10
        (define x (gaussian 0 1))
        (define y (gaussian 0 2))
        (define z (+ x y))
        (list x y)
        (> z 0)
)
```

Figure 1: A simple Church model

`RChurch` is not yet on CRAN. In the interrum, you can clone the git repository. From the command line, run

```
git clone git@github.com:/malmaud/RChurch.git -b develop
R CMD INSTALL RChurch
```

You might optionally also want to install the coda package, which is a set of diagnostic functions for analyziing MCMC chains; and the doMC and parfor packages, which allows multicore machines to run multiple chains in parallel. `RChurch` will automatically utilize these libraries if they are installed. You can install them from within R:

```
install.packages('coda')
install.packages('doMC')
install.packages('parfor')
```

2 Running Church models from R

2.1 Example

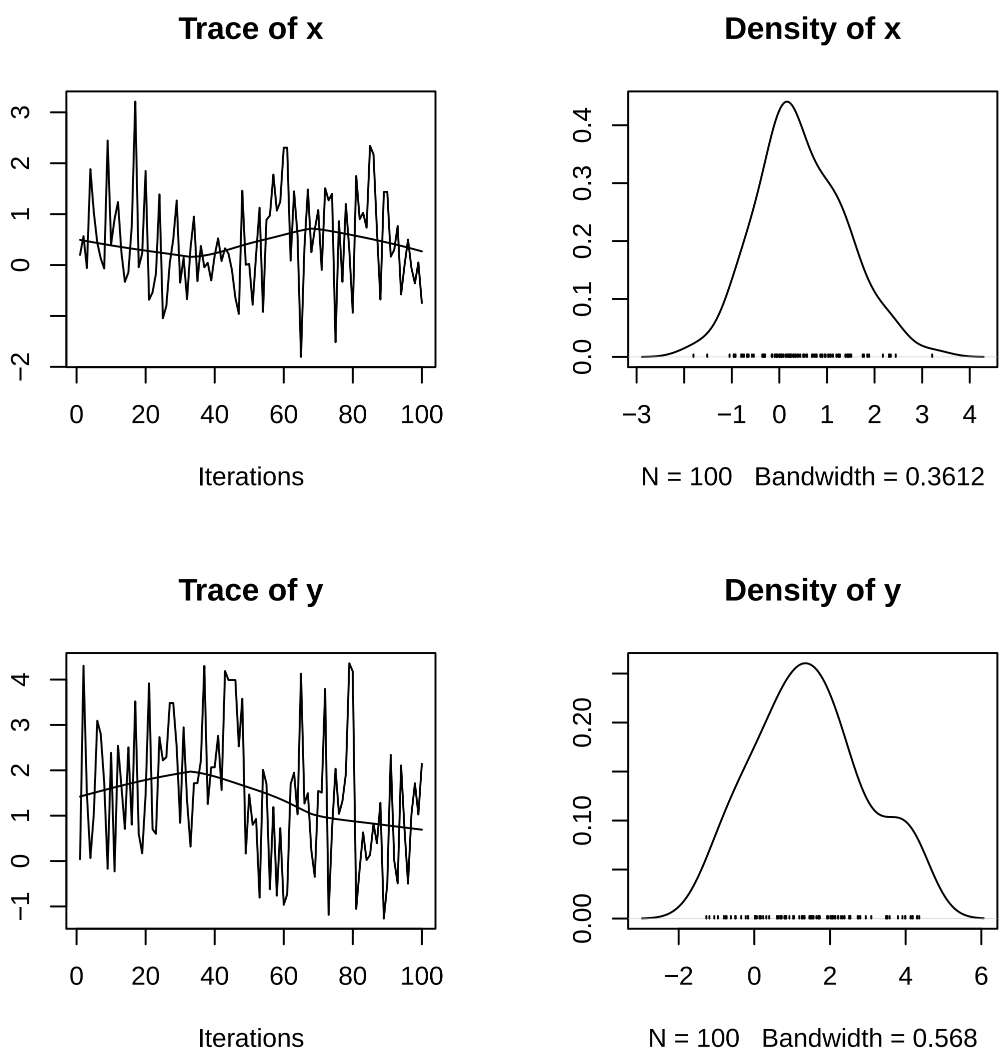
As a simple example, say we have the Church program in Figure 1 saved in a file called "simple.church" in R's working directory. We draw samples from the model and plot some diagnostics:

```
> library(RChurch)
> my.model = church.model(filename='simple.church')
> my.samples = church.samples(my.model, variable.names=c('x', 'y'))
> summary(my.samples)

Iterations = 1:100
Thinning interval = 1
Number of chains = 1
Sample size per chain = 100
```

```
1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

    Mean     SD Naive SE Time-series SE
x 0.4603 0.9216  0.09216         0.1076
y 1.4795 1.4536  0.14536         0.1853

2. Quantiles for each variable:

    2.5%      25%    50%   75% 97.5%
x -1.005 -0.07508 0.3129 1.072 2.324
y -1.012  0.54807 1.4630 2.352 4.247
```

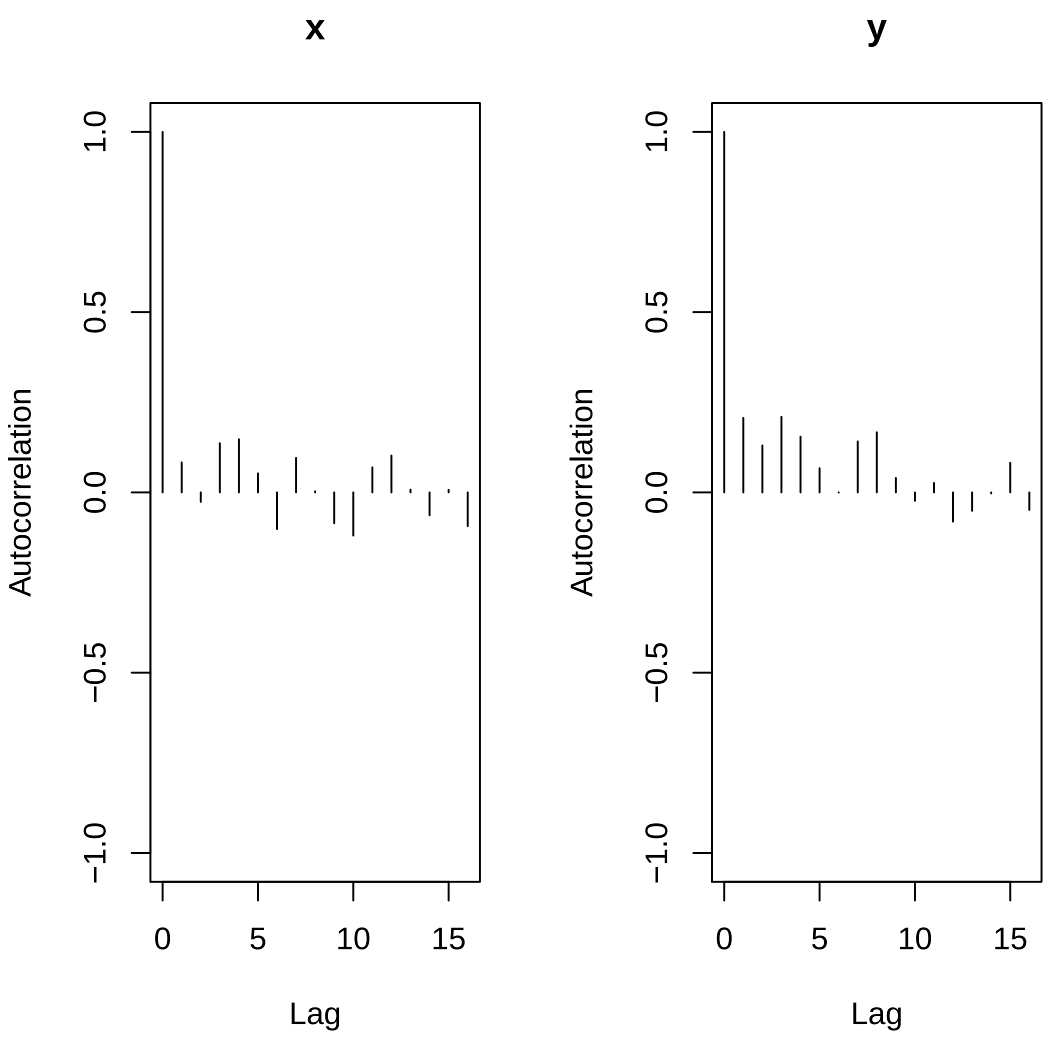
The distributions and traces of the monitored variables can be plotted:

```
> plot(my.samples)
```

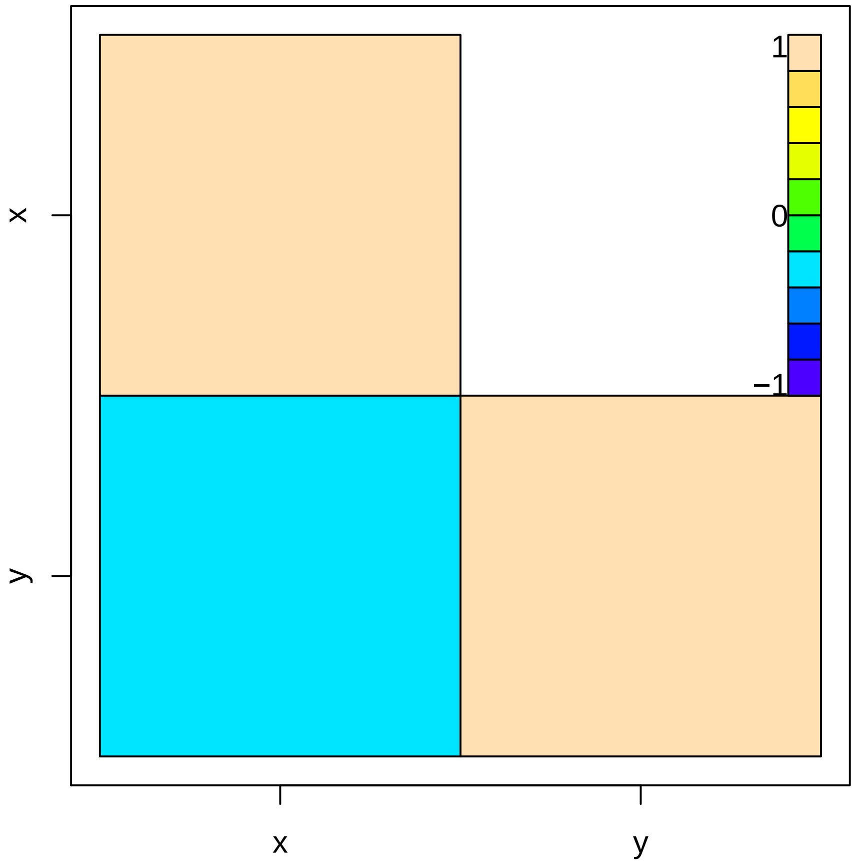


We can look at the auto-correlation and cross-correlation of the traces to get a sense of how quickly the chain mixes:

```
> diagnostics = church.diag(my.samples)

> autocorr.plot(diagnostics)
```



```
> crosscorr.plot(diagnostics)
```



`church.diag` returns an object of class MCMC, which the coda package knows how to analyze.

2.2 Multiple chains

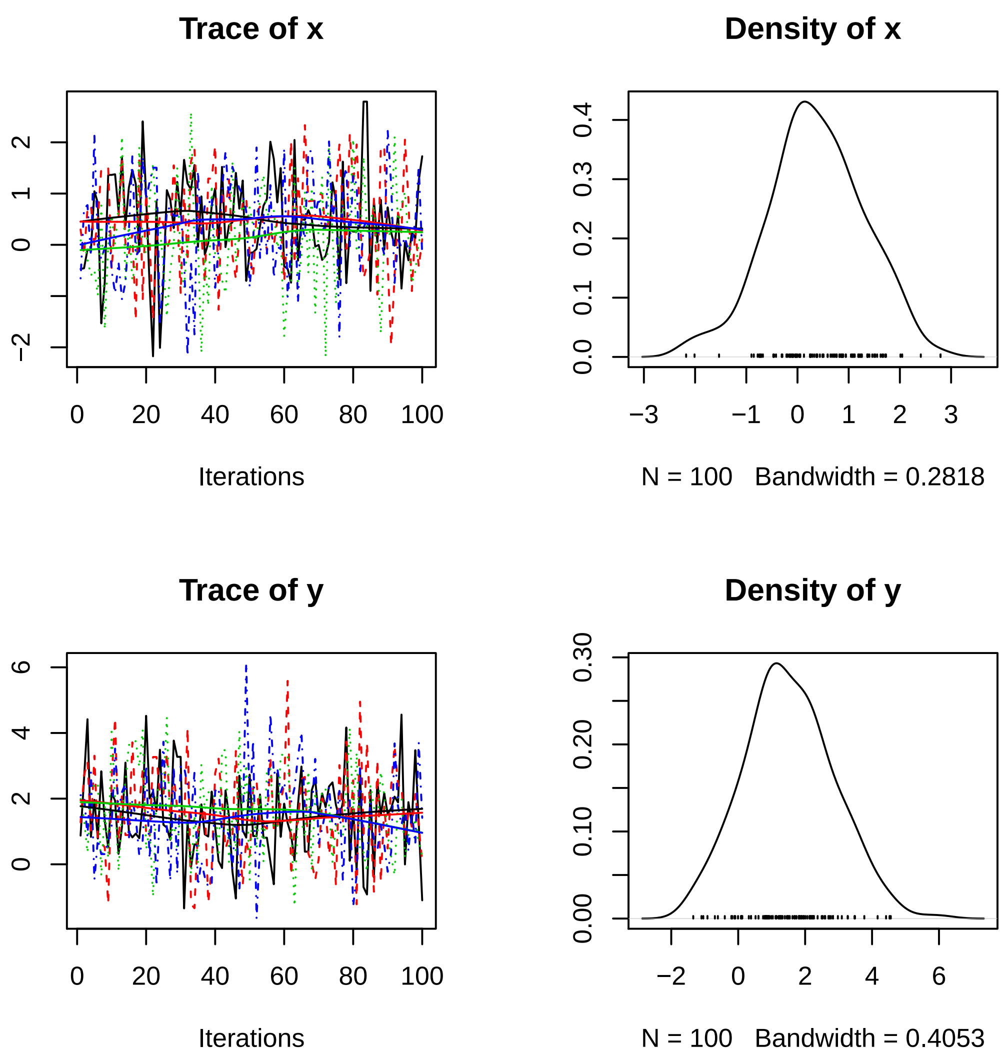
`RChurch` can also run multiple MCMC chains at different starting points and aggregate the results. It can also compare the chains to each to assess convergence. On a multi-core computer, the chains can also be run in parallel. Here is an example, again using the model in Figure 1.

```
> parallel.samples = church.samples(my.model, variable.names=c('x','y'),
+   n.chains=4, parallel=TRUE)
> summary(parallel.samples)
```

```
Iterations = 1:100
Thinning interval = 1
Number of chains = 4
Sample size per chain = 100

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

    Mean     SD Naive SE Time-series SE
x 0.3686 0.9303  0.04652        0.04885
y 1.5226 1.3310  0.06655        0.06664

2. Quantiles for each variable:

    2.5%     25%    50%    75% 97.5%
x -1.648 -0.1975 0.3247 0.9832 2.067
y -1.044  0.6631 1.4375 2.3613 4.147
```

The trace shows all the chains overlayed on each other:

```
> plot(parallel.samples)
```

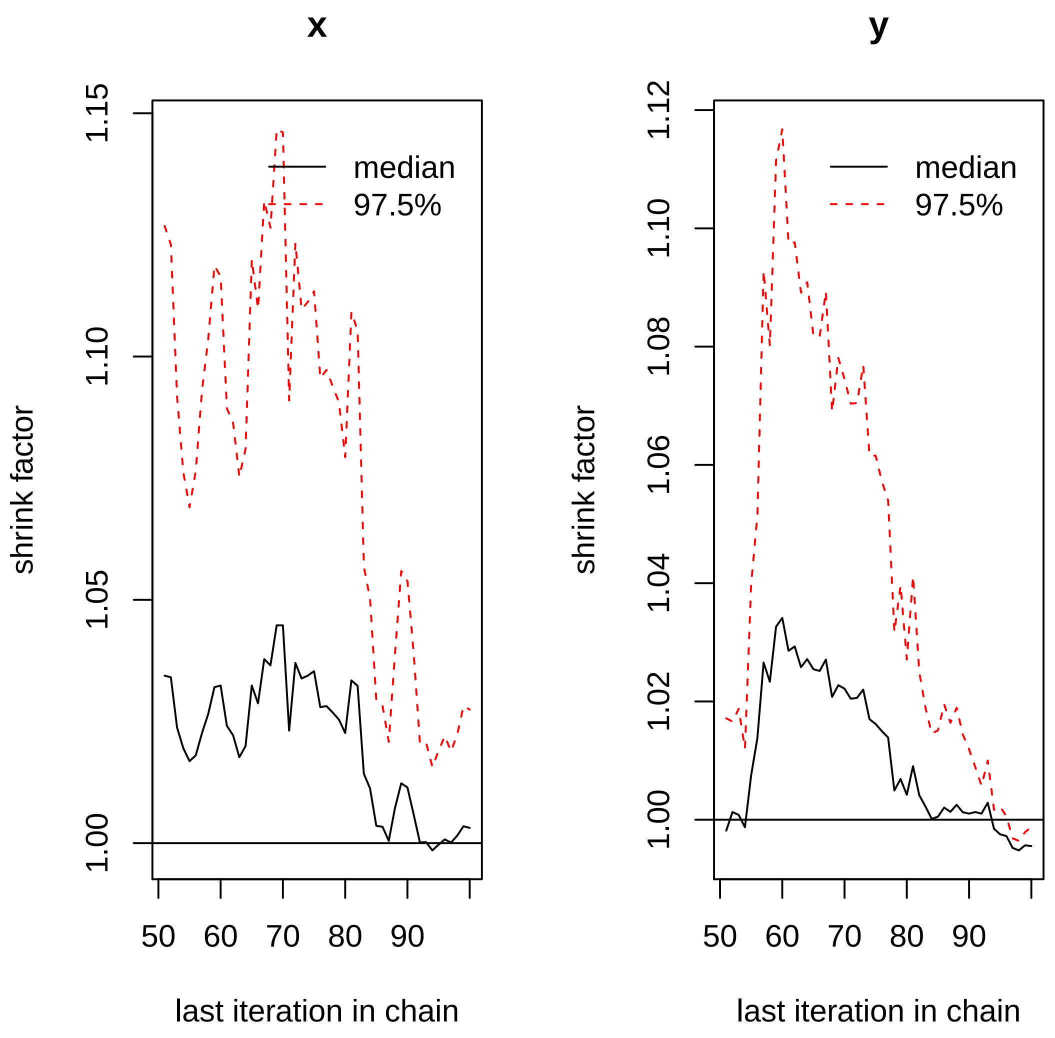


A Gelman diagnostic can assess convergence by comparing the variance within a chain to the variance between chains. An upper limit far away from 1 indicates a lack of convergence.

```
> parallel.diagnostics = church.diag(parallel.samples)
> gelman.diag(parallel.diagnostics)
Potential scale reduction factors:

  Point est. Upper C.I.
x      1.003      1.027
y      0.996      0.999

Multivariate psrf

1.01
```

We can also make a Gelman plot, showing how the Gelman diagnostic varies with iteration number:

```
> gelman.plot(parallel.diagnostics)
```



See the `coda` documentation for a full list of diagnostics.

2.3 Limitations

- The final output of the Church program must be a list of observed variables. The observed variables must either be a symbol/character string, numeric value, or a list of observed variables[1]. They can't, e.g., be an entire lambda expression. All `RChurch` cares

[1]The lists can be arbitrarily nested, however

about is that the result of executing the Scheme code through the Scheme interpreter is a list–it doesn't care whether it was actually generated by a Church query.

- A consequence is the output cannot be a single scalar, but must be a list with one element. So this won't work:

  ```
  (mh-query 100 10
    (define x (gaussian 0 1))
    x
    (> x 0)
  )
  ```

  But this will:

  ```
  (mh-query 100 10
    (define x (gaussian 0 1))
    (list x)
    (> x 0)
  )
  ```

- For character observed variables, the value of the variable cannot conflict with an R symbol.

- Logical returned variables (those whole value is #f or #t) are converted to 0 and 1 numeric values, respectively. String outputs with content #f or #t will also be mistakenly converted, so avoid using those.

3 Translate R into Scheme

3.1 Language translation

In addition to providing an interface to load and process Church code written in Scheme, `RChurch` can also automatically translate (certain) R programs into Scheme behind the scenes. This allows you to both write and analyze Church models entirely in the R environment without knowing any Scheme.

3.2 Example

We will convert the model in Figure 1. The equivalent code using just R is

```
> simple.model <- function() {
+   x <- rnorm(1, 0, 1)
+   y <- rnorm(1, 0, 2)
+   z <- x+y
+ }
```

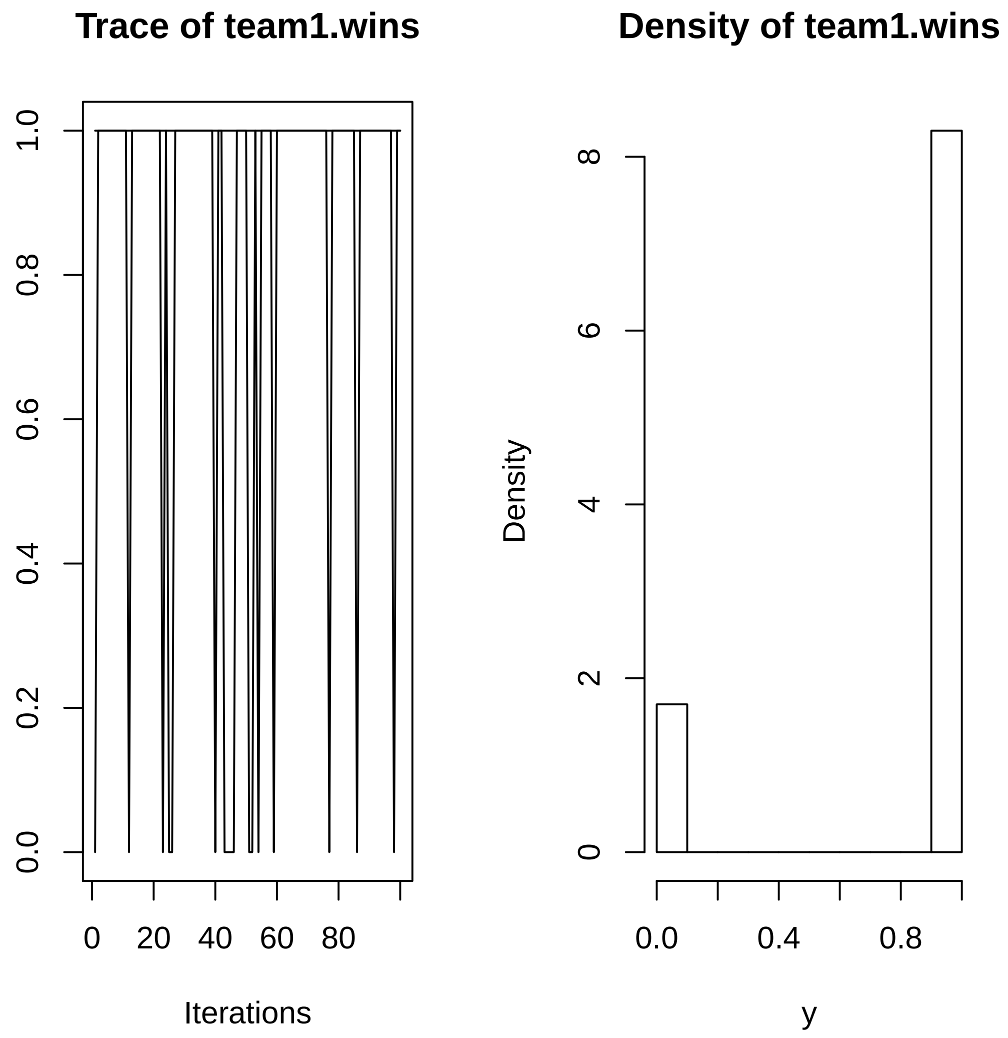
```
> R.model = church.model(model=simple.model, predicate=function() {z>0})
> R.samples = church.samples(R.model, n.iter=100,
+   thin=10, variable.names=c('x', 'y'))
> summary(R.samples)

Iterations = 1:100
Thinning interval = 1
Number of chains = 1
Sample size per chain = 100

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

    Mean     SD Naive SE Time-series SE
x 0.3772 0.8232  0.08232        0.08661
y 1.4071 1.3554  0.13554        0.14754

2. Quantiles for each variable:

     2.5%     25%    50%    75% 97.5%
x -0.8924 -0.2433 0.2906 0.9654 1.967
y -1.0113  0.3598 1.3976 2.2819 4.004
```

You don't explicitly reference mh-query or return a list of observed variables in the model. `RChurch` automatically generates the appropropriate query and list call.

For a more complicated example, we consider the "Tug of War" example from the Church wiki, reproduced in Figure 2. The R equivalent is

```
> model = function() {
+   strength = mem(function(person) if(flip()) 10 else 5)
+   lazy = function(person) flip(1/3)
+   total.pulling = function(team) {
+     realized.strength = function(person)
+       if(lazy(person)) strength(person)/2 else strength(person)
+     sum(sapply(team, realized.strength))
+   }
+   winner = function(team1, team2)
+     if(total.pulling(team1) < total.pulling(team2)) 'team2' else 'team1'
+   team1.wins = 'team1' == winner(c('bob', 'mary'), c('jim', 'sue'))
+ }
> predicate = function() {
+   strength('mary')>=strength('sue') &&
+    'team1' == winner(c('bob', 'francis'), c('tom', 'jim'))}
```

```
> tug.of.war.model = church.model(model, predicate)
> tug.of.war.samples = church.samples(tug.of.war.model,
+   variable.names=c('team1.wins'), n.iter=100, thin=10)

> plot(tug.of.war.samples)
```



Here the model makes reference to several symbols defined in Church but not R, such as flip and mem. Since the R-to-Scheme translator doesn't recognize these symbols, they are parsed through to Church unchanged. There are also some symbols defined in R but not Church, such as sapply. The translator recognizes a stock set of such R symbols and translates them into equivalent Scheme code. See Table 1 for a list of recognized R symbols.

```
(mh-query 100 1
  (define strength (mem (lambda (person) (if (flip) 10 5))))
  (define lazy (lambda (person) (flip (/ 1 3))))

  (define (total-pulling team)
    (sum
       (map (lambda (person) (if (lazy person) (/ (strength
          person) 2) (strength person)))
             team)))

  (define (winner team1 team2) (if (< (total-pulling team1) (
     total-pulling team2)) 'team2 'team1))

  (list (eq? 'team1 (winner '(bob mary) '(jim sue))))

  (and (>= (strength 'mary) (strength 'sue))
       (eq? 'team1 (winner '(bob francis) '(tom jim)))))
```

Figure 2: Tug of War model, in Scheme

R command	Church equivalent	Example
`<-` or `==`	define or let	`x<-5`
c	list	`x<-c(1, 3, 5)`
:	iota	`x<-1:5`
sapply	map	sapply(c(1,2,3), function(x) x+1)
`[`	vector-ref	`x<-c('jon', 'bob'); y<-x[2];`
replicate	repeat	`replicate(5, function() rnorm(1,0,1))`
function	lambda	`f<-function(x) 1+x`
`{`	|let*|	`f<-function(x) {y=2*x; y+1}`
sample	`uniform-draw`	`x<-c(1,6,12); y<-sample(x, 2)`
rnorm	`gaussian`	`x<-rnorm(1,0,1)`
if	if	`x<-if(y>0) 1 else -1`
`&&`, `||`	and, or	`predicate<-function() {a>b && (b<c || c<d)}`

Table 1: A list of R symbols that the translator knows how to translate into Church equivalents.

3.3 Limitations

- The model cannot call any R function or reference an R object that is not defined in the model or context.
- Arithmetic expressions in the model will not automatically vectorize. You have to explicitly use sapply or map. So this won't work:

```
function() {
  x = c(1,2,3)
  y = 5*x
}
```

  But this will:

```
function() {
  x = c(1,2,3)
  y = sapply(x, function(p) p*5)
}
```

- From within a lower scope, you cannot assign to a variable in a higher scope. So this won't work:

```
function() {
  x = 0
  y = rnorm(1,0,1)
  if(y>0) x <- 1 else x <- -1
}
```

  But this will:

```
function() {
  y = rnorm(1, 0, 1)
  x = if(y>0) 1 else -1
}
```

- The variables you want to monitor must be explicitly assigned to a variable within the model. You cannot monitor an expression. Note how in the "Tug of War" example, `team1.wins` is used to store the result of the logical expression we want to monitor.