0.1 Introduction

The `harvestr` package is a new approach to simulation studies that facilitates parallel execution. It builds off the structures available in the `plyr`, `foreach` and `rsprng` packages. What `harvestr` brings to the picture is abstractions of the process of performing simulation.

0.2 Process

The theme of `harvestr` is that of gardening, stemming from the idea that the pseudo-random numbers generated (RNG) in replicable simulation come from initial states called seeds. Figure 1 shows the basic process for `harvestr`.

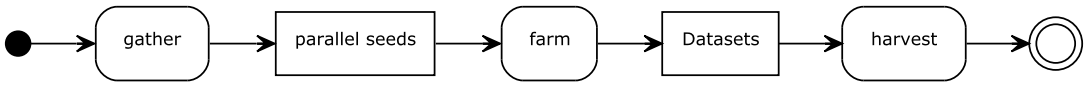


Figure 1: The basic `harvestr` process

The ideas are simple.

1. `gather(n, [seed])` takes an integer for the number of seeds to generate.
   Optionally, the seed can be set for replicable simulations. This uses the `rsprng` library to initialize independent parallel random number streams.

2. The seeds that are returned from `gather` are then fed into the `farm` function along with an expression to be generate data. `farm` returns a list of data frames each independently generated under each of the rng streams.

3. The final step is to apply a function to analyze the data. This is done with the `harvest` command, which takes the data from `farm` and applies an analysis function to the dataset. In the case that the analysis is deterministic `harvest` is equivalant to `llply` from the `plyr` package. The difference is with stochastic analysis, such as Markov Chain Monte Carlo (MCMC), where `harvest` resumes the RNG stream where `farm` left off when generating the data.

The effect is the results can be taken in any order and independently, and the final results are the same as if each analysis was taken from start to end with setting a single seed for each stream.

0.3 Example 1 - Basic Example

Some learn best by example. Here I will show a simple example for the basic process. Here we will perform simple linear regression for 100 data sets. First off we gather the seeds. This step is separate to facilitate storing the seeds to be distributed along with research if necessary.

```
> library(harvestr)
> library(plyr)
> seeds <- gather(100, seed=12345)
```

Second, we generate the data.

```
> datasets <- farm(seeds, {
+   x <- rnorm(100)
+   y <- rnorm(100, mean=x)
+   data.frame(y,x)
+ })
```

Then we analyze the data.

```
> analyses <-  harvest(datasets, lm)
```

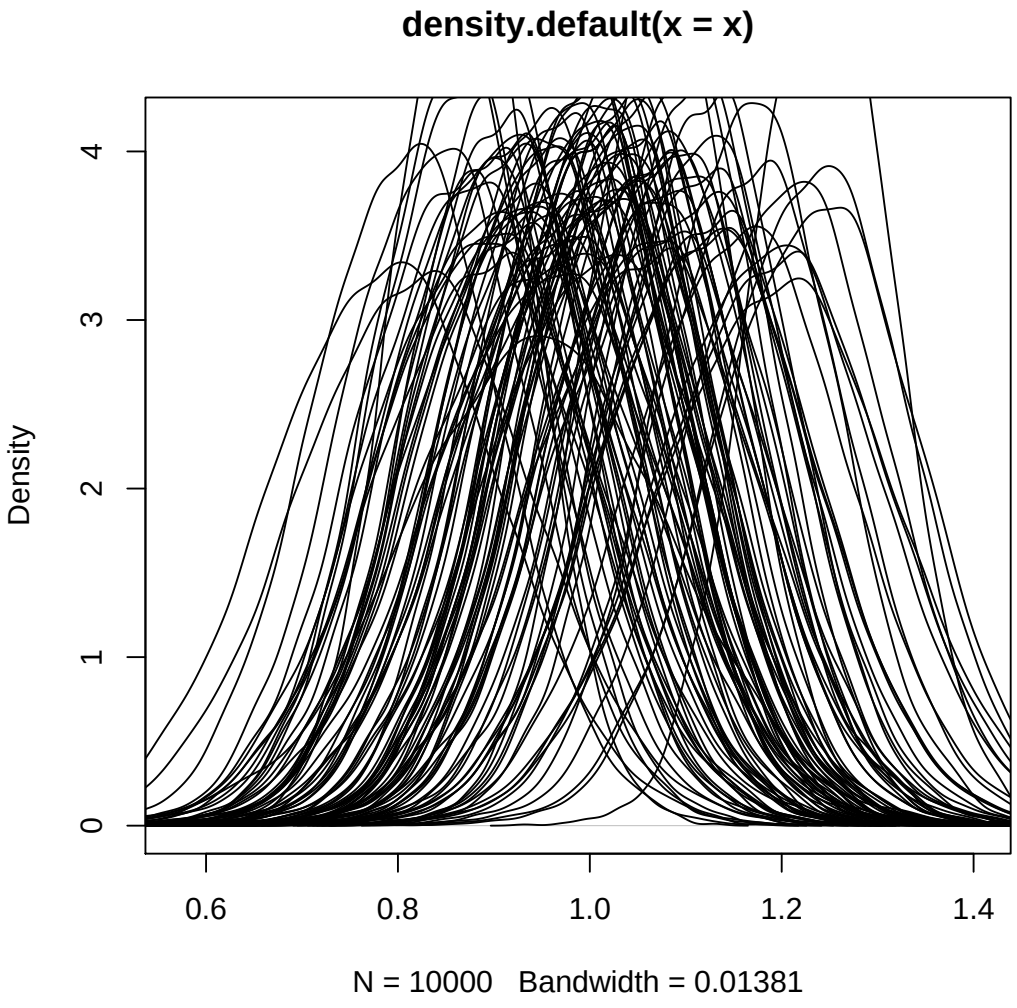
So what do we have in `analyses`? We have whatever `lm` returned. In this case we have a list of `lm` objects containg the results of a linear regression. Ussually we will want to do more to summarize the results.

```
> library(dostats)
> coefs <- t(sapply(analyses, coef))
> adply(coefs,2, dostats, mean, sd)

          X1        mean        sd
1 (Intercept) 0.009484538 0.1030893
2           x 1.006747591 0.1007218
```

0.4 Example 2 - Stochastic Analysis

That is very nice, but rather simple as far ananalyses go. What might be more interesting is to perform an analysis with a stochastic component such as Markov Chain Monte Carlo.

```
> library(MCMCpack)
> library(plyr)
> posteriors <- harvest(datasets, MCMCregress, formula=y~x)
> dataframes <- harvest(posteriors, as.data.frame)
> X.samples  <- harvest(dataframes, `[[`, "x")
> densities  <- harvest(X.samples, density)

> plot(densities[[1]])
> l_ply(densities, lines)
```



0.5 Example 3 - Caching

To ease longer analyses with many steps caching is available.

```
> unlink("harvestr-cache", recursive=TRUE)  # reset cache
> system.time({
+     posteriors1 <- harvest(datasets, MCMCregress, formula=y~x, cache=TRUE)
+ })

   user  system elapsed
  6.890   0.020   6.913
```

and when we run it again.

```
> system.time({
+     posteriors2 <- harvest(datasets, MCMCregress, formula=y~x, cache=TRUE)
+ })

   user  system elapsed
   0.21    0.02    0.23
```

To maintain integrity `harvestr` functions take use `digest` to create hashes of the seed, data, and function so that if any element changes, out of data cache results will not be used.