Julia for R programmers

Douglas Bates

University of Wisconsin - Madison
`<Bates@Wisc.edu>`

useR!2012
June 14, 2012

Outline

Outline

The Julia language

According to its developers (I have itemized what was a paragraph)

- *Julia is a high-level, high-performance dynamic programming language for technical computing, with syntax that is familiar to users of other technical computing environments.*
- *It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library.*
- *The library, mostly written in Julia itself, also integrates mature, best-of-breed C and Fortran libraries for linear algebra, random number generation, FFTs, and string processing.*
- *Julia programs are organized around defining functions, and overloading them for different combinations of argument types, which can also be user-defined.*

Similarities to R

- "**high-level** . . . **dynamic** programming language for **technical computing**".

high-level	can work on the level of vectors, matrices, structures, etc.
dynamic	values have types, identifiers don't. Functions can be defined during an interactive session.
technical computing	these folks know about floating point arithmetic

- "organized around defining **functions**, and overloading them for different combinations of argument types". The "overloading . . ." part means generic functions and methods.
- "syntax that is familiar to uses of other technical computing environments". Julia code looks very much like R code and/or Matlab/octave code. It is, of course, not identical but sufficiently similar to be readable.

Major differences - advantage Julia

- The ways in which R and Julia are different are too many to enumerate. These just skim the surface.
- "high-performance", "sophisticated compiler", "distributed parallel execution". We will see all of these in examples. The compiler, based on LLVM, creates dynamically-loaded machine code from method instantiations - type inference is used to create very fast code.
- "types, which can be user-defined". Julia has a sophisticated type system, simultaneously more specific and more general than classes in R. Scalar types exist (convenient for interfacing to C or Fortran compiled libraries).
- All method dispatch is S4-like multiple dispatch. All functions are generic.
- Distribution of base system and of packages is based on *git*.

Major differences - advantage R

- Named actual arguments
- Default argument values
- Handling of NA's is built-in to R at a very low level
- R is a mature language with large and active developer and user communities and an extensive infrastructure (CRAN, Bioconductor, mailing lists, The R Journal, JSS, hundreds of books describing R and its applications, . . .)

The Simple Gibbs example

- To get a flavor of the language, consider an example that has been used in many language comparisons (admittedly an example not well suited to R).
- Generate a sample from the (unscaled) bivariate density

$$f(x,y) \propto x^2 \exp(-xy^2 - y^2 + 2y - 4x)$$

  using direct Gibbs sampling from the conditional distributions

$$\mathcal{X}|\mathcal{Y} \sim \Gamma(3, y^2 + 4)$$
$$\mathcal{Y}|\mathcal{X} \sim \mathcal{N}\left(\frac{1}{1+x}, \frac{1}{2(1+x)}\right)$$

  with the Γ distribution in the shape/rate formulation
- Create a function of two arguments, N and *thin*, that returns a matrix of size $N \times 2$ containing N samples after thinning by *thin*.

The R function, Rgibbs

```
Rgibbs <- function(N,thin) {
    mat <- matrix(0,ncol=2,nrow=N)
    x <- 0
    y <- 0
    for (i in 1:N) {
        for (j in 1:thin) {
            x <- rgamma(1,3,rate=y*y+4)
            y <- rnorm(1,1/(x+1),1/sqrt(2*(x+1)))
        }
        mat[i,] <- c(x,y)
    }
    mat
}
```

As stated earlier, this function will not perform well because of the inherently sequential nature of the calculation. Even the byte-compiled version is relatively slow.

JGibbs1, using Rmath.jl

```
load("extras/Rmath.jl")
function JGibbs1(N::Int, thin::Int)
    mat = Array(Float64, (N, 2))
    x   = 0.
    y   = 0.
    for i = 1:N
        for j = 1:thin
            x = rgamma(1,3,1/(y*y + 4))[1]
            y = rnorm(1, 1/(x+1),1/sqrt(2(x + 1)))[1]
        end
        mat[i,:] = [x,y]
    end
    mat
end
```

The arguments to `rgamma` here are shape and scale (not rate).

JGibbs2, direct calls to libRmath

```
function JGibbs2(N::Int, thin::Int)
    mat = Array(Float64, (N, 2))
    x   = 0.
    y   = 0.
    for i = 1:N
        for j = 1:thin
            x = ccall(dlsym(_jl_libRmath, :rgamma),
                      Float64, (Float64, Float64),
                      3., 1/(y*y + 4))
            y = ccall(dlsym(_jl_libRmath, :rnorm),
                      Float64, (Float64, Float64),
                      1/(x+1), 1/sqrt(2(x + 1)))
        end
        mat[i,:] = [x,y]
    end
    mat
end
```

JGibbs3, the Julia randn and randg samplers

```
function JGibbs3(N::Int, thin::Int)
    mat = Array(Float64, (N, 2))
    x   = 0.
    y   = 0.
    for i = 1:N
        for j = 1:thin
            x = randg(3) / (y*y + 4)
            y = 1/(x + 1) + randn()/sqrt(2(x + 1))
        end
        mat[i,:] = [x,y]
    end
    mat
end
```

Distributed versions, dJGibbs3a and dJGibbs3b

- Julia allows for distributed parallel execution by specifying the number of processes at start-up.
- One appealing abstraction for parallel execution is a "distributed array" where each process works on a part of an array.
- The `dJGibbs3a` function leaves the result as a distributed array, suitable for further distributed processing. `dJGibbs3b` returns the result as an ordinary array. The results are from multiple chains not a single long chain as in the other functions.

```julia
function dJGibbs3a(N::Int, thin::Int)
    darray((T,d,da)->JGibbs3(d[1],thin),
           Float64, (N, 2), 1)
end
function dJGibbs3b(N::Int, thin::Int)
    convert(Array{Float64,2}, dJGibbs3a(N, thin))
end
```

Timings

- Detailed timings (and the code if you want to try yourself) are at `https://gist.github.com/2656226`.
- Roughly the results are:
  - `JGibbs1` is within a factor of 2 of `RcppGibbs`.
  - `JGibbs2` is nearly the same speed as `RcppGibbs`.
  - `JGibbs3` is faster than `RcppGibbs` (which uses the R samplers) and `GSLGibbs` (using the GSL samplers). Differences are attributable to different samplers.
  - On a 4-core processor using 4 processes (and no conflicting jobs), `dJGibbs3a` is nearly 4x faster than `JGibbs3`. `dJGibbs3b` is about 3x faster than `JGibbs3`, due to the communication overhead of converting from distributed to non-distributed.

Recall the comment about “library written in Julia itself”

```
# Generating gamma variables - Marsaglia and Tsang
function randg(a::Real)
    d = a - 1.0/3.0
    c = 1.0 / sqrt(9*d)
    while(true)
        v = 0.
        while (v <= 0.0)
            x = randn()
            v = 1.0 + c*x
        end
        v = v*v*v
        U = rand()
        x2 = x*x
        if U < 1.0 - 0.331*x2*x2; return d*v; end
        if log(U) < 0.5*x2 + d*(1.0 - v + log(v))
            return d*v
        end
    end
end
```