Julia for *R* programmers

Douglas Bates, U. of Wisconsin-Madison

July 18, 2013

What does Julia provide that R doesn't?

The *Julia* language

To quote its developers,

- *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.

R is great

- Open source, freely available, used in many disciplines
- Allows for user contributions through the package system.
- Package repositories, *CRAN* and *Bioconductor*, are growing rapidly. Over 3000 packages on *CRAN*.
- Packages can contain *R* code and sample data, which must be documented. They can also contain code in languages like C, C++ and Fortran for compilation and vignettes (longer forms of documentation).
- Many, many books about *R*. Journals like *Journal of Statistical Software* and the *R Journal* devoted nearly entirely to *R* packages.
- Widely used, a recent *coursera.org* MOOC on "Computing for Data Analysis" by Roger Peng had over 40,000 registrants.

R is great, but ...

- The language encourages operating on the *whole object* (i.e. vectorized code). However, some tasks (e.g. MCMC) are not easily vectorized.
- Unvectorized R code (*for* and *while* loops) is slow.
- Techniques for large data sets – parallelization, memory mapping, database access, map/reduce – can be used but not easily. *R* is single threaded and most likely will stay that way.
- *R* functions should obey *functional semantics* (not modify arguments). Okay until you have very large objects on which small changes are made during parameter estimation.
- Sort-of object oriented using generic functions but implementation is casual. Does garbage collection but not based on reference counting.
- The real work is done in underlying C code and it is not easy to trace your way through it.

Fast development vs. fast execution - Can we have both?

- The great advantage of *R*, an interactive language with dynamic types, is ease of development. High level language constructs, ease of testing small pieces of code, a read-eval-print loop (REPL) versus an edit-compile-run loop.
- Compilation to machine code requires static types. *C++* allows templates instead of dynamic types, and recent libraries like *STL*, *Boost*, *Rcpp*, *Armadillo*, *Eigen* use template metaprogramming for flexibility. But those who value their sanity leave template metaprogramming to others.
- *Julia* has a wide range of types, including user-defined types and type hierarchies, and uses multiple dispatch on generic functions with sophisticated type inference to emit code for the *LLVM* JIT.
- In my opinion *Julia* provides the best of both worlds and is the technical programming language of the future.

An example, a (very) simple Gibbs sampler

- The Gibbs sampler discussed on Darren Wilkinson's blog and also on Dirk Eddelbuettel's blog has been implemented in several languages, the first of which was R.
- The task is to create a Gibbs sampler for the density

$$f(x, y) = k\,x^2 \exp(-xy^2 - y^2 + 2y - 4x), \quad x > 0$$

using the conditional distributions

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

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

(Gamma parameters are shape, scale. Normal parameters are mean, variance.)

R version of simple Gibbs sample

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

Julia version using the Distributions package

```
using Distributions
function jgibbs(N::Integer, thin::Integer)
    mat = Array(Float64,(N,2))
    x = y = 0.
    for i in 1:N
        for j in 1:thin
            x = rand(Gamma(3.,1./(y*y+4.))) #shape/scale
            y = rand(Normal(1./(x+1.),1./sqrt(2.(x+1.))))
        end
        mat[i,1] = x; mat[i,2] = y
    end
    mat
end
```

- In *Julia* `0` is an integer and `0.` is floating point. *R* has the peculiar convention that `0` is floating point and `0L` is an integer.

Comparative timings

- In *R* generating a bivariate sample of size 10,000 with a thinning of 500 takes about 97 sec. on this laptop

```
> system.time(Rgibbs(10000,500))
   user  system elapsed
 96.740   0.004  97.027
```

- The corresponding *Julia* function runs in less than 1 second.

```
julia> jgibbs(10000,500);  # warm-up
julia> @elapsed jgibbs(10000,500)
0.867915085
```

- In *Julia* the first call to a method invokes the JIT so we time the second call.

Distributed version in Julia

```
djgibbs(N::Integer, thin::Integer) =
    DArray(I->jgibbs(map(length,I)[1],thin),(N,2))
```

- This rather cryptic function creates a distributed array (i.e. different parts are stored and accessed on different processors) using an anonymous function, which is the first argument and uses the `->` operator to create a function.
- This laptop has a 4-core processor. Starting `julia` with the `-p 4` flag (4 processes) provides

```
julia> djgibbs(10000,500); @elapsed djgibbs(10000,500)
0.218568565
```

- Almost exactly a 4-fold increase in speed using 4 processes.

Details of the syntax of these functions

Functions (algorithms) and types (data structures)

- Like *R* and *Matlab*, programming in *Julia* is based on defining *functions*.
- Like *R*, names of *generic functions* can be applied to different argument *signatures*, creating different *methods* for the generic.
- Unlike *R*, *all* functions in *Julia* are generic. In a sense, you don't define functions, you only define methods. The first time a function name occurs in a definition, the generic is automatically created.
- Data structures are called *types* in *Julia*. The language itself defines many types including abstract types (e.g. `Integer`) that include several concrete types.
- Users can define types to represent data as they choose.

Defining functions

- Function (actually method) definitions are usually the header followed by the body, a block of code
  - the header gives the formal argument names and, optionally, types
  - argument types can be templated
  - as in *R*, the last expression in the block is the return value

```
function jgibbs(N::Integer, thin::Integer)
 ...
end
```

- An alternative syntax can be used for *one-liners*, usually a single expression.

```
jgibbs(N::Integer) = jgibbs(N, 100)
```

- An anonymous function can be defined using the -> syntax.

Functions are really methods

- As mentioned above, in *Julia* all functions are generic so any function definition is actually a method definition. (Well, not exactly. Anonymous functions aren't generic because they're, well, anonymous.)
- There is nothing special about defining an additional method for a function. Just use the same function name and a different signature for the arguments
- Part of becoming fluent in *Julia* is learning to think in terms of methods.
- Default argument values can now be specified for *Julia* functions, as in *R*. Alternatively you can write a trivial, pass-through, method that has a truncated signature, as in the one-argument form of `jgibbs`. (The effect of specifying default values is to create such pass-through methods.)

Templated methods and data types

- The types Vector, Matrix and Array can have different element types; Vector{Float64}, Vector{Int}, ...
- Sometimes you want to define algorithms on the abstract type with minor variations for, say, the element type.

```
function sumsq{T <: Number}(V::Vector{T})
    s = zero(T)
    for v in V; s += v*v; end
    s
end
```

- The notation T <: Number means any type T that inherits from the abstract type Number.
- Note that some method signatures take the type itself as an argument. e.g. zero(T), one(T), zeros(T,dim...), ones(T,dim...)

Examples of types

```
julia> xdump(Number)
Number
  Real
  Complex
  ComplexPair = Complex
  ImaginaryUnit
  Complex128 = Complex
  Complex64 = Complex
```

- Check for yourself which types inherit from `Real`, `Integer`, `Signed`.

Using templated functions

- `Vector{T}` is a typealias for `Array{T,1}` (one-dimensional array with element type `T`).

```
julia> sumsq
# methods for generic function sumsq
sumsq{T<:Number}(V::Array{T<:Number,1}) at none:2

julia> typeof([1:3])
Array{Int64,1}

julia> sumsq([1:3])
14

julia> typeof(sumsq([1:3]))
Int64
```

A function can modify its arguments

- It is important to realize that a *Julia* function can modify its arguments. If you are not careful this can result in bugs.
- However, sometimes you *want* to modify in-place and this is exactly what you need. Often the key to speeding up an algorithm is cutting down on the copies being created, as we will see later.
- By convention, such *mutating* functions in the standard library have names ending in `"!"`, indicating that users should be careful when using such functions. Thus `copy!` is mutating, `copy` is not.

```
julia> src = [1:3]; dest = [0:2]; println(dest)
[0,1,2]

julia> copy!(dest, src); println(dest)
[1,2,3]
```

Defining types

- The Cholesky decomposition of a positive definite symmetric matrix, `A`, is written by numerical analysts as the lower triangular matrix, `L` such that `A = LL'` and by statisticians as the upper triangular R such that `A = R'R`.
- LAPACK can work with either form and with element types of `Float32`, `Float64`, `Complex64` or `Complex128`, collectively called `BlasFloat`.

```
abstract Factorization{T}

type Cholesky{T<:BlasFloat} <: Factorization{T}
    UL::Matrix{T}
    uplo::Char
end
```

Documentation

Finding documentation, asking questions

- The project's web site is http://julialang.org/
- The `docs` link takes you to the documentation for version 0.1, which is out of date. The 0.2 release is in the works but no release date yet. When downloading use the 0.2-pre versions.
- Use the documentation at http://julia.readthedocs.org/en/latest (a.k.a. http://docs.julialang.org). Notice that there is a list of available packages.
- Mailing lists `julia-dev` and `julia-users` are linked on the project web site. As always, it is best to read several conversations on the archives before posting, just so you can get a feel for the culture.

Finding function names

- As with any language, *Julia* will initially be frustrating because you don't know the name of the function that does ...
- It is best to start with a quick perusal of the manual to familiarize yourself.
- The documentation on the standard library is organized in topic groups which can be helpful (if you can decide which topic applies).
- Interactive *Julia*, (i.e the REPL), provides tab completion, after which you can check likely names with `help`
- Read the code written by people who seem to know what they are doing and then experiment. You have the REPL and can check things quickly.

Using the REPL

- By default the value of an expression, even an assignment, is shown (unlike the behavior of *R* where printing of assigned values is suppressed).
- Large vectors and other arrays have only the first and last few rows, columns, etc. printed.
- Printing of other objects may blurt thousands of numbers, etc. onto your screen.
- Vectors are printed vertically. The `println` function's output is sometimes more compact. Another trick is to transpose a vector so it prints horizontally.
- A semicolon (`;`) at the end of an expression suppresses printing.
- There is no default printing in batch mode. You must use explicit output function calls.
- The last value evaluated is available as `ans`.

The Distributions package

R's approach to distributions

- The `p-q-r-d` functions, providing the cumulative probability function (e.g. `pnorm`), the quantile function (e.g. `qnorm`), a random generator function (e.g. `rnorm`) and the density or probability mass function (e.g. `dnorm`), are an important part of the *R* language.
- However, the convention of creating distinct functions for each distribution results in dozens of such functions and some confusion (`pgeom` is the c.d.f and `dgeom` is the p.m.f.).
- Every such function takes some, possibly overspecified, set of parameters for the distribution. For safety, each such function should do a consistency check on the arguments.

Distributions as types

- In the `Distributions` package for Julia we define distributions as types. For efficiency most distributions are labelled as `immutable`, indicating that you can't change the parameter values after you generate an instance. This also allows the compiler to take some short cuts when working with instances of such types.
- A function of the same name as the type declared within the type definition is called an *internal constructor*. Usually these check validity and perform conversions, if needed. They end in a call to `new`.
- An *external constructor* has the same name as the type. It can be used to create default values.

An example, constructors for the Normal distribution

```
immutable Normal <: ContinuousUnivariateDistribution
    mu::Float64
    sd::Float64
    function Normal(mu::Real, sd::Real)
        sd > zero(sd) || error("sd must be positive")
        new(float64(mu), float64(sd))
    end
end
Normal(mu::Real) = Normal(float64(mu), 1.0)
Normal() = Normal(0.0, 1.0)
```

- The check on `sd > 0.` in the internal constructor could be written with an `if` statement. This idiom is read "either sd $> 0.$ or throw an error".
- The external constructors set the defaults for `sd` to 1. and `mu` to 0.

Methods for distributions

- There are dozens of generics that can be applied to `Distribution` types.
- Install the package

```
julia> Pkg.add("Distributions")
```

and check the types and methods shown in the file

```
julia> Pkg.dir("Distributions", "src", "Distributions.jl")
"/home/bates/.julia/Distributions/src/Distributions.jl"
```

Vectorization, etc.

- Often we want to apply one of the generic functions to vector or array arguments instead of scalars.
- Defining distributions as types and providing abstract types (UnivariateDistribution, ContinuousDistribution, ContinuousUnivariateDistribution) allows for easy vectorization of methods. Check the result of, for example,

```
julia> using Distributions
julia> methods(pdf)
# methods for generic function pdf
pdf{T<:Real}(d::Dirichlet,x::Array{T<:Real,1}) at /home/bates/
pdf{T<:Real}(d::Multinomial,x::Array{T<:Real,1}) at /home/bate
pdf{T<:Real}(d::MultivariateNormal,x::Array{T<:Real,1}) at /ho
pdf(d::MultivariateDistribution,X::AbstractArray{T,2}) at /hom
...
```

Some examples

```
julia> n1=Normal(3, 0.25); (xvals = linspace(2.,4.,9))'
1x9 Float64 Array:
 2.0  2.25  2.5  2.75  3.0  3.25  3.5  3.75  4.0
julia> [mean(n1) median(n1) modes(n1) std(n1) var(n1)]
1x5 Float64 Array:
 3.0  3.0  3.0  0.25  0.0625
julia> [skewness(n1) kurtosis(n1)]
1x2 Float64 Array:
 0.0  0.0
julia> [pdf(n1, xvals) cdf(n1,xvals) quantile(n1,linspace(0.1,
9x3 Float64 Array:
 0.000535321  3.16712e-5  2.67961
 0.0177274    0.0013499   2.78959
 0.215964     0.0227501   2.8689
 0.967883     0.158655    2.93666
 1.59577      0.5         3.0
 0.967883     0.841345    3.06334
```

Linear algebra

Linear algebra operations

- The names of functions and symbols for operators are similar to those of *Matlab*.
- For vector and matrix arguments, `*` and `/` and `\` denote matrix multiplication and solutions of linear systems. The (conjugate) transpose operator is `'`.
- A leading `.` on an operator denotes elementwise operations. e.g. `.*`, `./`, `.<`
- Names of generators of particular array forms are patterned on *Matlab*; `zeros`, `ones`, `rand`, `randn`, `linspace`, `eye`, `speye`
- Sparse matrices are part of the base language.
- Various extractors like `diag`, `triu` and `tril` are available.
- Predicates (functions that check for a characteristic and return a Boolean value) usually begin with `"is"`. E.g. `istriu`, `issym`.

Getting and setting elements or subarrays

- When indexing arrays, *all of a dimension* is denoted by `:`. Thus the `j`th column of matrix `A` is `A[:,j]`.
- The last value in a dimension can be written **`end`**. Thus `v[2:end]` is everything in the vector `v` beyond the first element.
- The `size` generic returns dimensions. With no additional arguments the value is a `"tuple"`. Specifying a specific dimension returns an integer. The `length` generic returns the overall length (i.e. the product of the dimensions).

```
julia> I = eye(4);
julia> println("$(size(I)), $(size(I,1)), $(length(I))")
(4,4), 4, 16
```

- Note the use of expression interpolation in a string

Assigning multiple values

- Having shown that `size` returns a tuple, it is a good time to mention that the elements of the tuple can be assigned to multiple names in a single statement.
- A common idiom is

```
function matprod{T<:Number}(A::Matrix{T}, B::Matrix{T})
    m,n = size(A); p,q = size(B)
    n == p || error("incompatible dimensions")
    res = Array(T, m, q)
    ...
end
```

Comprehensions

- It is common to create a vector or array by applying a scalar function to the values of another vector or array or to indices.
- In *Julia* you do not need to "fear the loop", especially within a method definition.
- A *comprehension* is an implicit loop that creates the result array and fills it. Suppose we want to apply a logit function to an array.

```
logit(p::Float64) = 0<p<1?log(p/(1.-p)):error("p not in (0,1)'
logit(V::Vector{Float64}) = [logit(v) for v in V]
logit(linspace(0.1,0.9,9))'

1x9 Float64 Array:
 -2.19722  -1.38629  -0.847298  -0.405465  0.0  0.405465  0.84
```

- The scalar `logit` method uses the ternary operator (`cond ? trueres : falseres`) and the ability to chain comparisons (`0. < p < 1.`)

Using BLAS and LAPACK

- Dense matrix operations are based on the well-known *LAPACK* (Linear Algebra Package) and *BLAS* (Basic Linear Algebra Subroutines) facilities. *Julia* is usually built against an accelerated, multi-threaded BLAS package such as *OpenBLAS* or *MKL* providing very high performance.
- Occasionally you need to throttle the number of threads being used with, e.g.

```
blas_set_num_threads(1)
```

- Operations such as `X'X`, `X'y`, and `norm(y)` on a vector `y` and matrix `X` will automatically call a suitable BLAS routine. These allocate space for the result.

Calling BLAS and LAPACK directly

- Sometimes you want the result of a linear algebra calculation to overwrite existing storage, thereby saving on storage allocation and freeing (called "garbage collection") in an iterative algorithm.
- Most BLAS and LAPACK operations can be called directly. Templated Julia functions like `gemm` (general matrix-matrix product) allocate storage whereas `gemm!` overwrites one of its arguments.
- Because the Julia functions are templated, their names are the BLAS or LAPACK names without the first character.
- The calling sequence for the `!` version is the BLAS or LAPACK version without all the dimension arguments.

Factorizations

- The backslash, \, operator applied to a matrix and vector analyzes the form of the matrix to decide on a suitable factorization (e.g. Cholesky, LU, QR) to solve a linear system or a least squares problem.
- Explicitly creating the factorization allows for its re-use or for more specific forms. Available factorizations include
    - `Cholesky` (functions `cholfact` and `cholfact!`)
    - `CholeskyPivoted` (functions `cholpfact` and `cholpfact!`)
    - `LU` (functions `lufact` and `lufact!`)
    - `QR` (functions `qrfact` and `qrfact!`)
    - `QRPivoted` (functions `qrpfact` and `qrpfact!`)
    - `Hessenberg` (functions `hessfact` and `hessfact!`)
    - Eigenvalue-eigenvector and generalized eigenvalue (functions `eigfact`, `eigfact!`, `eigvals`, `eigvecs`, `eig`, `eigmax`, `eigmin`)
    - SVD and generalized SVD (`svdfact`, `svdfact!`, `svd`, `svdvals`)
    - Schur (functions `schurfact` and `schurfact!`)

Use of the backslash

```
julia> srand(1234321)        # set the random seed
julia> X = [ones(100) rand(100,2)]; X'
3x100 Float64 Array:
 1.0         1.0        1.0        1.0        1.0         …  1.0
 0.0944218   0.936611   0.258327   0.930924   0.555283       0.79928
 0.942218    0.042852   0.658443   0.933942   0.493509       0.00561
julia> beta = rand(Normal(),3); beta'
1x3 Float64 Array:
 -0.197287  -0.86977  -1.55077
julia> y = X*beta + rand(Normal(0.,0.4), size(X,1)); y'
1x100 Float64 Array:
 -1.89246  -1.06572  -1.92631  -2.00036  -1.19763  …  -1.2715
julia> betahat = X\y; betahat'
1x3 Float64 Array:
 -0.225888  -0.913171  -1.4425
```

Use of factorizations

```
julia> using NumericExtensions
julia> ch = cholpfact!(X'X)
CholeskyPivoted{Float64}(3x3 Float64 Array:
 10.0  5.26508  4.88059
  0.0  3.03731  0.289294
  0.0  0.0      2.91872 ,'U',[1,2,3],3,-1.0,0)
julia> beta = ch\X'y; beta'
1x3 Float64 Array:
 -0.225888  -0.913171  -1.4425
julia> df = length(y) - rank(ch)
julia> fitted = X*beta; ssqr = sqdiffsum(y, fitted)/df
0.15681475390926472
julia> vcov = ssqr * inv(ch)
3x3 Float64 Array:
  0.00981028  -0.00818202  -0.00806099
 -0.00818202   0.0171654   -0.00175329
 -0.00806099  -0.00175329   0.0184078
```

Julia packages

Using packages created by others

- Although only recently available, the *Julia* package system is growing rapidly. Check the list of available packages on the documentation site.
- Packages of interest to statisticians include
  - DataFrames — facilities similar to *R* data frames and formula language
  - Distributions — described above
  - GLM — fitting linear and generalized linear models
  - MixedModels — mixed-effects models - currently LMMs but GLMMs are coming
  - NLopt — high-quality optimizers for constrained and unconstrained problems
  - NumericExtensions - highly optimized matrix reductions etc.
  - Winston — 2D plotting
  - various MCMC packages

An example - overdetermined ridge regression

Overdetermined ridge regression

- It is not uncommon now to work with data sets that have fewer observations (rows) than variables (columns).
- For example, GWAS (genome-wide association studies) data may determine status at millions of SNP sites for tens of thousands of individuals.
- Paulino Perez and Gustavo de la Campos performed a centered, scaled ridge regression using *R* on such data where at each SNP position a binary response was measured. Here $p > n$ and a p-dimensional coefficient vector is estimated by regularizing the system to be solved.
- On modern computers the system could be solved in memory when p is in the thousands or tens of thousands but not for p in the hundreds of thousands. A *back-fitting* algorithm was used.

R code

```
backfit <- function(Xstd, yc, h2 = 0.5, nIter = 20L) {
    n <- nrow(Xstd); p <- ncol(Xstd)
    stopifnot(length(yc) == n)
    lambda <- p * (1 - h2)/h2
    SSx <- colSums(X^2)   # the diagonal elements of crosspro
    bHat <- rep(0,p)      # initial values bj=zero
    e <- y                # initial model residuals
    for(i in 1:nIter){    # loop for iterations of the algori
        for(j in 1:p){    # loop over predictors
            yStar <- e+X[,j] * bHat[j] # forming offsets
            bHat[j] <- sum(X[,j] * yStar)/(SSx[j]+lambda) # e
            e <- yStar-X[,j] * bHat[j]   # updates residuals
        }
    }
    bHat
}
```

Direct *Julia* translation

```julia
function backfit(Xstd::Matrix, yc::Vector, h2=0.5, nIter=20)
    n,p = size(Xstd)
    lambda = p*(1-h2)/h2
    SSx = sum(Xstd .^ 2, 1) # all n-1 after standardizing
    bHat = zeros(p)
    e = copy(yc)
    for i in 1:nIter, j in 1:p
        yStar = e + Xstd[:,j]*bHat[j]      # offsets
        bHat[j] = dot(Xstd[:,j],yStar)/(SSx[j] + lambda)
        e = yStar - Xstd[:,j]*bHat[j]
    end
    bHat
end
```

Julia version with in-place updates

```julia
using Base.LinAlg.BLAS.axpy!
function backfit2(Xstd::Matrix, yc::Vector, h2=0.5, nIter=20)
    n,p = size(Xstd)
    lambda = p*(1 - h2)/h2
    SSx = sum(Xstd .^ 2, 1) # all n-1 after standardizing
    bHat = zeros(p)
    e = copy(yc)
    for i in 1:nIter, j in 1:p
        axpy!(bHat[j], Xstd[:,j], e)
        bHat[j] = dot(Xstd[:,j], e)/(SSx[j] + lambda)
        axpy!(-bHat[j], Xstd[:,j], e)
    end
    bHat
end
```

Julia version using NumericExtensions

```
using NumericExtensions
function backfit3(Xstd::Matrix, yc::Vector, h2=0.5, nIter=20)
    n,p = size(Xstd)
    length(yc) == n || error("Dimension mismatch")
    SSx = sum(Abs2(), Xstd, 1) #  all n-1 after standardizing
    bHat = zeros(p); lambda = p*(1 - h2)/h2
    e = copy(yc)
    for i in 1:nIter, j in 1:p
        Xj = unsafe_view(Xstd, :, j:j)
        fma!(e, Xj, bHat[j]) # fused multiply and add
        bHat[j] = mapreduce(Multiply(),Add(),Xj,e)/(SSx[j]+lan
        fma!(e, Xj, -bHat[j])
    end
    bHat
end
```

Timing results

- On a moderate-sized data set (600 by 1280) `backfit2` is twice as fast as `backfit` in *Julia* and about 3 times as fast as the *R* version.

```
> system.time(backfit(X,y))
   user  system elapsed
  1.604   0.012   1.623
```

```
julia> @time backfit(Xstd, yc);
elapsed time: 1.093174015 seconds
julia> @time backfit2(Xstd, yc);
elapsed time: 0.460209712 seconds
```

- Profiling the execution of *backfit2* showed that most of the time was spent in indexing operations, leading to the `backfit3` version.

```
julia> @time backfit3(Xstd, yc);
elapsed time: 0.103352683 seconds
```

Timing results (cont'd)

- A direct calculation using the 1280 by 1280 system matrix with an inflated diagonal was nearly as fast as `backfit3`.

```
function directfit(Xstd::Matrix, yc::Vector, h2=0.5)
    n,p = size(Xstd); lambda = p*(1-h2)/h2; XtX = Xstd'Xstd
    SSx = sum(Abs2(), Xstd, 1) #  all n-1 after standardizing
    for i in 1:size(XtX,1); XtX[i,i] += SSx[i] + lambda; end
    cholfact!(XtX)\(Xstd'yc)
end
```

```
julia> @time directfit(Xstd, yc);
elapsed time: 0.132523019 seconds
```

Timing results (cont'd)

- On a 1000 by 200000 example `backfit3` took 25 seconds, `backfit2` 125 seconds and `backfit` nearly 300 seconds in *Julia*.

```
julia> size(Xstd)
(1000,200000)
julia> @time backfit(Xstd,yy);
elapsed time: 296.172161458 seconds
julia> @time backfit2(Xstd, yy);
elapsed time: 125.506957544 seconds
julia> @time backfit3(Xstd, yy);
elapsed time: 25.27092502 seconds
```

- The direct solution with a 200000 by 200000 system matrix is not on.
- If the matrix `X` is a binary matrix, it is more effective to store the matrix as a `BitArray` and evaluate the two possible values in each column separately, customizing the multiplication operation for this data type

Workflow in *Julia* as described by Tim Holy

- The workflow “quickly write the simple version first” (which is really pleasant thanks to Julia’s design and the nice library that everyone has been contributing to) -> “run it” -> “ugh, too slow” -> “profile it” -> “fix a few problems in places where it actually matters” -> “ah, that’s much nicer!” is quite satisfying.
- If I had to write everything in C from the ground up it would be far more tedious. And doing a little hand-optimization gives us geeks something to make us feel like we’re earning our keep.
- Key points as summarized by Stefan Karpinski
  1. You can write the easy version that works, and
  2. You can usually make it fast with a bit more work

Calling compiled code

Calling compiled functions through *ccall*

- *R* provides functions `.C`, `.Fortran` and `.Call` to call functions (subroutines) in compiled code.
- The interface can be tricky, whole chapters of the manual “Writing R Extensions” are devoted to this topic.
- The *Rcpp* package provides a cleaner interface to *C++* code but using it is still no picnic.
- In *Julia* it is uncommon to need to write code in *C/C++* for performance.
- The *ccall* facility allows for calling functions in existing *C* libraries directly from *Julia*, usually without writing interface code.

Example of *ccall*

- In the *Rmath* library the *pbinom* function for evaluating the *cdf* of the *binomial* distribution has signature

```
double pbinom(double x, double n, double p, int lower_tail, ir
```

- We can call this directly from *Julia* as

```
function cdf(d::Binomial, q::Number)
    ccall((:pbinom,:libRmath), Cdouble,
          (Cdouble, Cdouble, Cdouble, Cint, Cint),
          q, d.size, d.prob, 1, 0)
end
```

- *Cdouble* is a typealias for `Float64`, *Cint* for `Int32` making it easier to translate the signature. Pointers and, to some extent, C structs can be passed as well.

Julia for "Big Data"

Julia for “wide data”

- Data sets like that from a GWAS study is sometimes called “wide data” because the number of “variables” (SNP locations in this case) can vastly excede the number of observations (test subjects).
- We showed a simulated example with 1000 subjects and 200,000 SNP locations but much larger studies are common now. Millions of SNP locations and perhaps 10’s of thousands of subjects.
- In these cases we must store the data compactly. The combinations of the `.bed`, `.bim` and `.fam` files generated by plink are common.
- The data are a (very) large matrix of values that can take on 1 of 4 different values (including the missing value). They are stored as 4 values per byte.

GenData2 data type

```julia
type GenData2
    gendat::Matrix{Uint8}
    nsubj::Int
    counts::Matrix{Int}
end
```

GenData2 constructor

```
function GenData2(bnm::ASCIIString)     # bnm = basename
    nsnp = countlines(string(bnm,".bim"))
    nsubj = countlines(string(bnm,".fam"))
    bednm = string(bnm,".bed"); s = open(bednm)
    read(s,Uint8) == 0x6c && read(s,Uint8) == 0x1b || error("w
    read(s,Uint8) == 1 || error(".bed file, $bednm, is not in
    m = div(nsubj+3,4) # of bytes per col., nsubj rounded up t
    bb = mmap_array(Uint8, (m,nsnp), s); close(s)
    GenData2(bb,nsubj,bedfreq(bb,nsubj)')
end
```

Counter for column frequencies

```
function bedfreq(b::Matrix{Uint8},ns::Integer)
    m,n = size(b)
    div(ns+3,4) == m || error("ns = $ns should be in [$(4m-3)
    cc = zeros(Int, 4, n)
    bpt = convert(Ptr{Uint8},b)
    for j in 0:(n-1)
        pj = bpt + j*m
        for i in 0:(ns-1)
            cc[1+(unsafe_load(pj,i>>2+1)>>(2i&0x06))&0x03,
               1+j] += 1
        end
    end
    cc
end
```

Performance the current version

- On the “consensus” data set from the “HapMap 3” samples (296 subjects, 1,440,616 SNPs) the time to generate the frequency counts was about 5 seconds on a single process

```
julia> @time gd = GenData2("/var/tmp/hapmap3/consensus");
elapsed time: 5.662568297 seconds
julia> gd.counts'
4x1440616 Int64 Array:
    3     0    15     0  245  230  203  147   77  …     4    3
   16     3     0     0    0   19   28    3   17        0
   11    66   123     1  459  433  410  438  290        0
 1154  1115  1046  1183  480  502  543  596  800     1180  115
```

- Using distributed arrays is an obvious next step.

Julia on "long" data

- I hear of people needing to fit models like logistic regression on very large data sets but getting the data are usually proprietary. Simulating such data

```
julia> using GLM
julia> n = 2_500_000; srand(1234321)
julia> df2 = DataFrame(x1 = rand(Normal(), n),
                       x2 = rand(Exponential(), n),
                       ss = pool(rand(DiscreteUniform(50), n)));
julia> beta = unshift!(rand(Normal(),52), 0.5);
julia> eta = ModelMatrix(ModelFrame(Formula(:(~ (x1 + x2 + ss
                                                   df2)).m * beta;
julia> mu = linkinv!(LogitLink, similar(eta), eta);
julia> df2["y"] = float64(rand(n) .< mu); df2["eta"] = eta;
julia> df2["mu"] = mu;
```

Julia on “long” data (cont’d)

```
julia> head(df2)
                x1        x2 ss   y       eta       mu
[1,]     -0.160767 0.586174  6 1.0   1.92011 0.872151
[2,]      -1.48275 0.365982 46 0.0 -0.206405 0.448581
[3,]     -0.384419  1.13903 37 1.0   1.22092 0.772226
[4,]       -1.3541  1.02183 10 1.0  0.534944 0.630635
[5,]       1.03842   1.1349 34 1.0   3.33319  0.96555
[6,]      -1.16268    1.503  6 0.0   2.21061 0.901198
```

Julia on “long” data (cont’d)

```
julia> @time gm6 = glm(:(y ~ x1 + x2 + ss), df2, Bernoulli(),
elapsed time: 19.711060758 seconds

Formula: y ~ :(+(x1,x2,ss))
Coefficients:
           Estimate  Std.Error  z value     Pr(>|z|)
[1,]       0.410529  0.0122097  33.6232 7.69311e-248
[2,]       0.550263 0.00214243   256.84          0.0
[3,]        1.01559 0.00370327  274.241          0.0
[4,]       0.591439  0.0206309  28.6676 9.66624e-181
...
```

- I had plans for making the numerical work faster but then I profiled the execution and discovered that the time was being spent creating the model matrix.

Julia summary

Good points

- Speed of a compiled language in a dynamic, REPL-based language. (They said it couldn't be done.)
- Core developers are extremely knowledgable (and young).
- Syntax "not unlike" *R* and/or *Matlab*. Many functions have the same names.
- generic functions, multiple dispatch and parallelization built into the base language.

Pitfalls

- Need to learn yet another language. Syntax and function names are similar to *R* but not the same.
- Language, package system and packages are still in their infancy (but development is at an amazing pace).