Distributions in Julia

Simon Byrne

Department of Statistical Science
Centre for Computational Statistics and Machine Learning
University College London

26th June 2014

Outline

About me

I'm a research fellow in Statistics: my research focuses on computational statistics, particularly Markov chain Monte Carlo (MCMC).

- Mostly theoretical analysis of computational techniques.
- but I occasionally need to write some code.
- Sequential algorithms: can't be vectorised.

I want a language that is

- Quick to write: easy to try out ideas.
- Reasonably performant, but representative of the speed of the algorithm.
- Easy to understand: if I need to look at the code 6 months later.
- Able to peek under the hood.

How I met Julia

A blog post by Justin Domke (September 2012)

> *Just to never write another .mex file, I'll very seriously consider Julia for new projects*

I had Matlab code full of:

```
R = chol(A)
y = R \ ( R' \ x)
```

I really like being able to write Julia code:

```
C = cholfact(A)
y = C \ x
```

I filed my first pull request three weeks later. . .

Outline

1. Distributions.jl
2. Examples
3. Current work
4. Future ideas

Distributions

Distributions.jl is a library for working with probability distributions.

- One of the oldest julia packages (5th package added to METADATA).
- Started as a wrapper for Rmath library (more on this later).
- 568 commits, 6747 lines of code, 27 contributors.
- Other main contributors: Dahua Lin, John Myles White, Douglas Bates and Andreas Noack Jensen.
- MIT licensed (eventually).

Structure

Distributions are types, *e.g.*

```
Gamma <: Distribution{Univariate,Continuous}
```

- Allows a consistent interface: we don't have to remember function prefixes, or argument order.
- Capitalisation of types avoids `Gamma` (distribution)/`gamma` (function) confusion.
- Immutable types means there is no overhead in creating/destroying types.

Properties and functions

Properties accept a distribution argument

```
julia> mean(Gamma(3,5))
15.0
julia> kurtosis(Beta(2,8))
0.49038461538461536
```

Functions accept a distribution and an argument

```
julia> pdf(Gamma(3,5),1.0)
0.0032749230123119257
julia> cdf(Normal(0,1),10)
1.0
julia> ccdf(Normal(0,1),10) # 1 - cdf(Normal(0,1))
7.619853024160593e-24
julia> logccdf(Normal(0,1),10) # log(1 - cdf(Normal(0,1)))
-53.23128515051247
```

Sampling and fitting

`rand` has been extended to allow `Distribution` arguments

```
julia> rand(Binomial(10000,0.2))
2025
julia> rand(Binomial(10000,0.2),1000)
1000-element Array{Int64,1}:
 2064
 1993
 1946
 ...
```

Maximum likelihood estimation

```
julia> fit_mle(Gamma, rand(Gamma(3,5),1000))
Gamma( shape=2.9489146726658464 scale=5.24658602972775 )
```

Outline

Kernel density estimation

A kernel density estimator (KDE) estimates a pdf from a sample $x_1, \ldots, x_n$,

$$\hat{f}_\theta(x) = \sum_{i=1}^{n} \frac{1}{n} k_\theta(x - x_i)$$

where k_θ is a pdf of some symmetric probability distribution centred at 0.
A simple approach:

```
function kde(X,k)
    m = 1024 # grid size
    M = linspace(minimum(X),maximum(X),m)
    y = zeros(m)
    for x in X
        y += pdf(k, M.-x) / length(x)
    end
    M,y
end
```

- Requires $O(nm)$ operations.

KDEs via FFTs

We are convolving the “empirical” density with that of the kernel

- Use Fourier transforms!

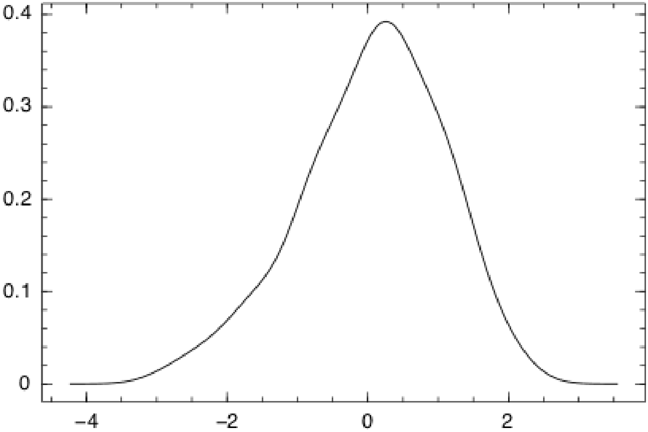
1. Tabulate the data to the grid: $O(n)$ operations.
2. Compute the fft of the table: $O(m \log m)$
3. Convolve by multiplying by Fourier transform of kernel.
   - Can be computed directly from the *characteristic function* `cf(k,x)`: $O(m)$.
4. Compute inverse fft: $O(m \log m)$

This has the further benefit of being able to re-use the table

- Useful for selecting kernel bandwidth (*e.g.* via cross-validation)

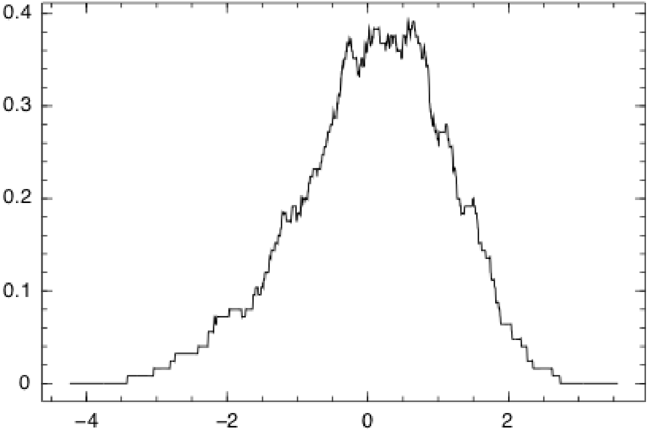
Example: KernelDensity.jl

```
using KernelDensity
plot(kde(X)) # uses Normal kernel by default
```



Example: KernelDensity.jl (cont.)

```
plot(kde(X,kernel=Uniform)) # Uniform kernel
```



Outline

Replacing Rmath

Rmath is a library of numerical C routines developed for use in R.

- GPL licensed
- Possibly the most widely used open-source library of such functions
- 15+ years of extensive use.
- Reasonably efficient
- Provides log-space functions for extreme tails.
- Reliable implementations of many tricky functions: incomplete gamma and beta functions, non-central χ^2 cdf, etc.

Being GPL, we can't just translate the C code.

Other sources

- Common methods often have other implementations available:
  - Naval Surface Warfare Center (NSWC) library (Fortran 77)
  - Cephes (C)
  - Boost (C++)
  - SciPy (Python/C)
- Otherwise it means deciphering journal appendices.
- Licence status not always clear
  - ACM have a lot to answer for.
- Old code
  - Often predates IEEE-754 standard (exact round-to-nearest arithmetic, gradual underflow, etc.)
  - Lots of `Q = 0.5 + (0.5 - P)`
- No analysis of floating-point error.
- Often okay for “reasonable” values, but can be wildly inaccurate in the extremes.

Example: Poisson density

Consider the Poisson density function

$$\frac{\lambda^x e^{-\lambda}}{x!}$$

- For large values of x, the λ^x can easily overflow (or underflow for $\lambda < 1$).

Can re-write as

$$\frac{\exp\{x \log(\lambda) - \lambda\}}{x!}$$

- If x and λ are of a similar magnitude: amplifies relative error of $[x \times \log(\lambda)]$.
- Amplified further by taking exponent.

Example: Poisson density (cont.)

Using Stirling's asymptotic approximation to the gamma function,

$$x! = \sqrt{2\pi x}\exp\{x\log x - x + s(x)\} \quad \text{where } s(x) \approx \frac{1}{12x} - \frac{1}{360x^3} + \dots,$$

we can write

$$\frac{\lambda^x e^{-\lambda}}{x!} = \frac{\exp\{x \operatorname{logmxp1}(\lambda/x) - s(x)\}}{\sqrt{2\pi x}}$$

where $\operatorname{logmxp1}(x) = \log(x) - x + 1$.

- $s(x)$ accurate < 2 ulps for $x > 10$
  - ulps = "units in last place" = multiples of `eps(x)`
- Both terms in exponent of same sign: no explosion of relative error
- As a bonus: we don't need to evaluate a separate gamma function.
- Yet more alternatives:
  - Rmath uses a slightly different formulation (Loader, 2000)
  - Boost uses a method based Lanczos' approximation to the gamma function

But what about this $\operatorname{logmxp1}(x)$?

Utility functions

Need lots of supporting routines: at the moment we have the following variants of log:

- $\text{log1mexp}(x) = \log(1 - \exp(x))$
- $\text{log2mexp}(x) = \log(2 - \exp(x))$
- $\text{log1pexp}(x) = \log(1 + \exp(x))$
- $\text{logexpm1}(x) = \log(\exp(x) - 1)$
- $\text{log1psq}(x) = \log(1 + x^2)$
- $\text{log1pmx}(x) = \log(1 + x) - x$
- $\text{logmxp1}(x) = \log(x) - x + 1$

Naive versions exhibit numerical instability:

- overflow and underflow when taking exponents.
- catastrophic cancellation when subtracting two quantities of a similar magnitude.

Anatomy of log1pmx

Recall the Taylor expansion

$$\log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots, \quad |x| < 1$$

For small x, $\log(1+x) \sim x$: this is why we have a log1p function.

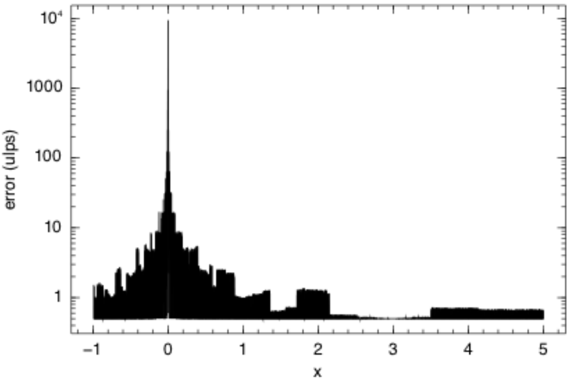
```
julia> log(1 + 1e-20)
0.0
julia> log1p(1e-20)
1e-20
```

For small x, $\log(1+x) - x \sim -\frac{1}{2}x^2$

```
julia> x = 1e-20; log1p(x) - x
0.0
julia> x = big(1e-20); log1p(x) - x |> float64
-5.0e-41
```

- Catastrophic cancellation

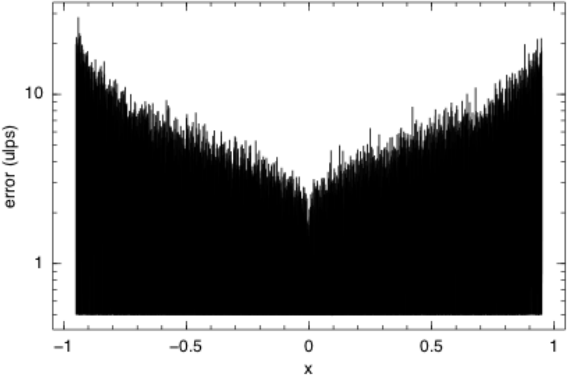
How bad is it?



- Most libm functions are accurate to < 1 ulp.
- Need to do something better for $|x| < -0.95$.

Simple series summation

We keep adding terms until the summation stops changing (used by Boost)



- Accrues large round-off error (we're summing from largest to smallest)
- Series is slow to converge: can require up to 618 terms.
- Frequent branching: difficult for compiler to optimise.

A better series

Let $r = \frac{x}{x+2}$, then

$$\log(1+x) = 2r + \frac{2}{3}r^3 + \frac{2}{5}r^5 + \dots$$

- Used by log1p in openlibm.
- Half as many terms
- All of the same sign: no cancellation error.

Can be rearranged to obtain useful expressions for log1pmx

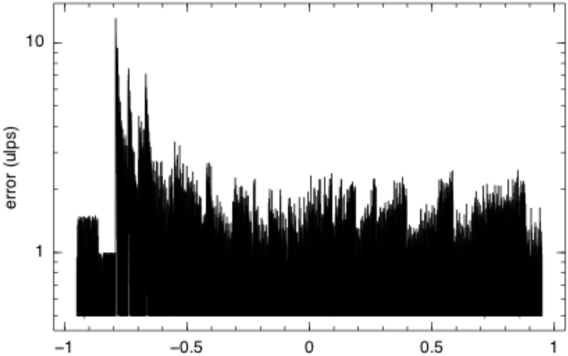
Rmath

Rmath uses this series:

- For small values: converges in a known number of terms, can be evaluated using a Horner expansion:

  $$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = a_0 + x \times (a_1 + x \times (a_2 + \ldots (a_n \times x) \ldots))$$

- For other $|x| < 1$, use a continued fraction representation: lower error than the naive approach.

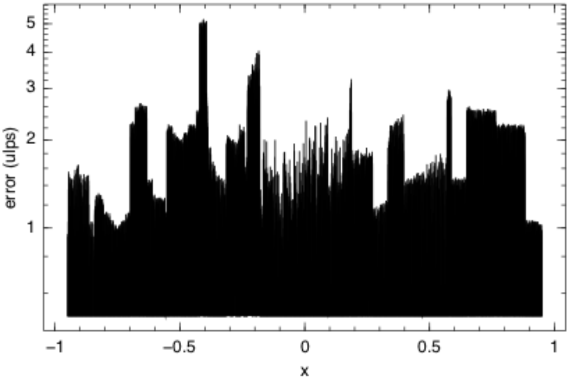


NSWC library

The NSWC function is based on the same series:

- For small x use the first few terms, plus a rational approximation of the remainder
- For other values, do a range reduction:

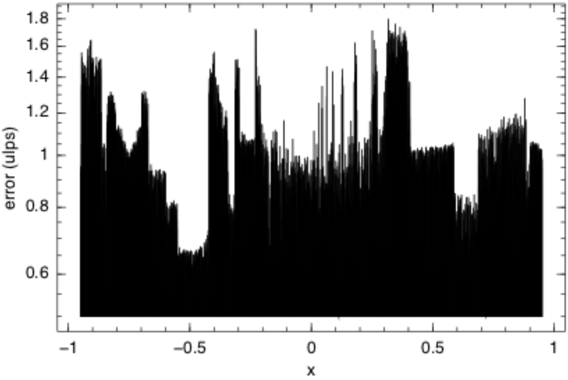
$$u = \frac{x-\alpha}{1+\alpha}, \quad \log(1+x) - x = [\log(1+u) - u] + [\log(1+\alpha) - \alpha] - \alpha u$$



Some more tweaks

As good as this is, we can still improve further

- The rational approximation is unnecessary.
- Rearrange and use different reductions to exploit exact IEEE floating-point operations: *e.g.*
  - Multiplying by powers of 2
  - Subtracting numbers of the same magnitude.



Still more to go

Still lots more to do:

- Incomplete gamma functions: Gamma distribution cdf

$$\frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}dt, \quad \frac{1}{\Gamma(a)}\int_x^\infty e^{-t}t^{a-1}dt$$

- Incomplete Beta function: Beta distribution cdf

$$\frac{1}{B(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}dt$$

- Inverses of these functions (quantiles)
- Non-central distributions.

Sampling random numbers

- Often different algorithms for different parameters
- *e.g.* Binomial(n,p)
  - Sampling n Bernoulli variables infeasible for large n.
  - For very small (or large) probabilities:

```
y = 0; x = 0
while true
    y += rand(Geometric(p)) + 1
    if y > n
        return x
    end
    x += 1
end
```

  - Otherwise: use a normal approximation with appropriate corrections.
- Can re-use constants for multiple samples.
  - Rmath uses global variables: not thread safe.
  - We define `Sampler` types, containing appropriate values
- Polyalgorithm then chooses appropriate method.

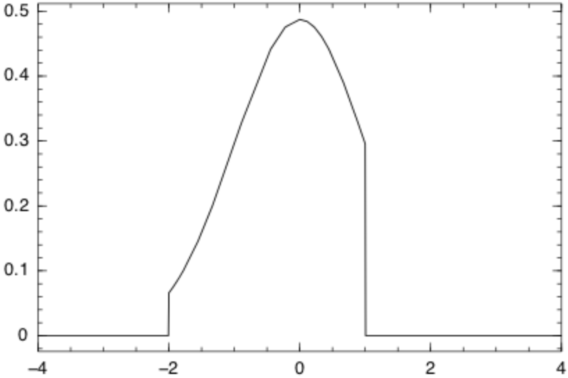
Outline

Truncated types

Currently have a `Truncated` parametric type for distributions constrained to an interval.

- Functions (pdf, cdf, quantile) are derived from the underlying distribution.

```
julia> plot(x -> pdf(Truncated(Normal(0,1),-2,1), x),
  xrange=(-4,4))
```



Variable transformations

Often require monotonic transformations of variables (*e.g.* `log`, `exp`, `sqrt`):

```
immutable TransformDist{D<:UnivariateDistribution,
  F <: Functor}
  dist::D
end
```

- `Functor` is a type representing a function (from Dahua's `NumericFuns.jl`).

Can derive methods:

```
rand{D,F}(d::TransformDist{D,F}) = evaluate(F,rand(d.dist))
cdf{D,F}(d::TransformDist{D,F},x::Real) =
  cdf(d.dist, evaluate(inv(F),x))
pdf{D,F}(d::TransformDist{D,F},x::Real) =
  pdf(d.dist, evaluate(inv(F),x))*evaluate(grad(inv(F)),x)
```

Sufficient statistics

Many common distributions are *exponential families*

$$p(x \mid \theta) = h(x) \exp\{s(x)^\top t(\theta) - A(\theta)\}$$

$s(x)$ is a *sufficient statistic*

- contains all the information about θ from x
- linear: we can summarise data $x_1, \ldots, x_n$ by $\hat{s} = \sum_{i=1}^n s(x_i)$

Define `suffstats` methods for each distribution:

```
s = suffstats(Gamma,x)
```

Can then be used instead of the data for computing likelihoods, fitting, *etc.*

- Currently all defined manually for each distribution.
- Seems ripe for some meta-programming trickery.

Orthogonal polynomials

Any univariate probability distribution P defines a Hilbert space with inner product

$$\langle f, g \rangle_P = \int_{\mathbb{R}} f(x) g(x) dP(x)$$

Applying Gram–Schmidt to the monomials $1, x, x^2, \ldots$ gives a sequence of *orthogonal polynomials*. *e.g.*

Normal(0,1) : *Hermite polynomials*: 1, x, $x^2 - 1$, $x^3 - 3x$, $\ldots$

Exponential(1) : *Laguerre polynomials*: 1, $-x + 1$, $\frac{1}{2}x^2 - 2x + 1$, $\ldots$

Quadrature rules (Gauss, Clenshaw–Curtis) evaluate numerical integrals via projections onto orthogonal polynomials at a finite number of evaluation points

- Could be used for computing expectations:

```
expectation(Normal(0,1), f)
```

and more . . .

- Constrained estimation
- Conjugate updating
- Graphical models
- Gradients and Hessians
- `Float32` methods
- Non-Euclidean sample spaces:
  - Circles
  - Spheres
  - Stiefel manifolds (orthogonal matrices)
  - Orthogonal group (orientation matrices)
  - Combinatorial spaces

Final thoughts

- A great way to learn numerical analysis.
- Convenient `BigFloat` arithmetic is invaluable.
- Strict IEEE arithmetic with predictable rounding is extremely useful for understanding error.

Challenges/future features

- Access to extended precision arithmetic (`Float80`/`Float128`) could improve accuracy.
  - Difficult to implement consistently across platforms.
  - How accurate do we need to be?
- Convenient and consistent syntax for in-place operations.