Notes 9: Numerics

Chris Paciorek

2025-02-20

Table of contents

Introduction

This document is the ninth of a set of notes, this document focusing on numerical questions, random number generation, and linear algebra. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

Random number generation

The seed

As usual, it's best to set the random number seed.

```
using Random
Random.seed!(1234);      # Seed number 1234
println(rand(3))
```

```
[0.32597672886359486, 0.5490511363155669, 0.21858665481883066]
```

```
println(rand(5))
```

```
[0.8942454282009883, 0.35311164439921205, 0.39425536741585077, 0.9531246272848422, 0.7955469475347194
```

```
Random.seed!(1234);     # Re-seed with same number - will give the same sequence of random numbers
println(rand(2))
```

```
[0.32597672886359486, 0.5490511363155669]
```

```
println(rand(6))
```

```
[0.21858665481883066, 0.8942454282009883, 0.35311164439921205, 0.39425536741585077, 0.953124627284842
```

`rand` has a variety of methods.

```
rand(1:10, 3)
```

```
3-element Vector{Int64}:
 5
 8
 6
```

```
rand(['a','b','c'], 5)
```

```
5-element Vector{Char}:
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
```

Generators

The manual discusses the available generators.

The default RNG is `Xoshiro256++`, but there are details related to random number streams for a given Task that I haven't delved into.

```
Random.default_rng();
Random.seed!(1234)
println(rand(3))
```

```
[0.32597672886359486, 0.5490511363155669, 0.21858665481883066]
```

```julia
rng = Random.Xoshiro(1234)
println(rand(rng, 3))
```

```
[0.32597672886359486, 0.5490511363155669, 0.21858665481883066]
```

The default used to be the Mersenne Twister (still the default in R and formerly the default in Python/numpy).

```julia
rng = Random.MersenneTwister(1234);
println(rand(rng, 3))
```

```
[0.5908446386657102, 0.7667970365022592, 0.5662374165061859]
```

Distributions

```julia
using Distributions
beta_dist = Beta(2, 5)
beta_samples = rand(beta_dist, 10)
```

```
10-element Vector{Float64}:
 0.4542940512832609
 0.27004146470644036
 0.3870672630509891
 0.21116529046475746
 0.5041249425689261
 0.2784321400052276
 0.6829933994934323
 0.1295297689806119
 0.43795698919772685
 0.39395847238977155
```

```julia
beta_density = pdf.(beta_dist, beta_samples[1:5])
```

```
5-element Vector{Float64}:
 1.2086291289041646
 2.300088115277408
 1.6389245754521693
 2.4529451430316644
 0.9144258514892248
```

```julia
beta_logDensity = logpdf.(beta_dist, beta_samples[1:5])
```

```
5-element Vector{Float64}:
  0.1894867660152757
  0.8329474331914342
  0.4940402800647119
  0.8972894018105007
```

```
-0.08945889536877027
```

Floating point issues

Integer and floating point types

64 bit integers can represent $-2^63 \ldots 2^63$. 32 bit integers can represent $-2^31 \ldots 2^31$.

Similarly for 16 and 128 bit integers.

Values outside that range *overflow*.

```julia
xi::Int64 = 2^62
```

```
4611686018427387904
```

```julia
xi::Int64 = 2^70  # Overflows
```

```
0
```

```julia
xi::Int64 = 2^63  # Just overflows.
```

```
-9223372036854775808
```

```julia
yi::Int128 = 2^63  # Hmmm.
```

```
-9223372036854775808
```

```julia
yi::Int128 = Int128(2)^63
```

```
9223372036854775808
```

```julia
Int64(yi)
```

```
LoadError: InexactError: trunc(Int64, 9223372036854775808)
InexactError: trunc(Int64, 9223372036854775808)

Stacktrace:
 [1] throw_inexacterror(f::Symbol, ::Type{Int64}, val::Int128)
   @ Core ./boot.jl:634
 [2] checked_trunc_sint
   @ ./boot.jl:656 [inlined]
 [3] toInt64
   @ ./boot.jl:705 [inlined]
 [4] Int64(x::Int128)
   @ Core ./boot.jl:784
 [5] top-level scope
   @ In[19]:1
```

```julia
xi::Int64 = 2^63 - 1  # Shouldn't this overflow when calculating 2^63?
```

```
9223372036854775807
```

```julia
x = parse(BigInt, "1234567890123456789012345678901234567890")
```

```
1234567890123456789012345678901234567890
```

```julia
x+1
```

```
1234567890123456789012345678901234567891
```

There are 16, 32, and 64 bit floating point numbers, as well as `BigFloat`s.

More a bit later.

Floating point precision

Let's consider how much precision we have with real-valued numbers because of limited floating point precision.

For 64-bit floats, 53 bits are used for precision, which translates to approximately 16 digits of accuracy in base 10, regardless of the magnitude of the number. How many digits of accuracy do we have with 32-bit and 16-bit floats?

```julia
using Printf
@sprintf("%.20f", Float64(1/3))
```

```
"0.33333333333333331483"
```

```julia
@sprintf("%.20f", Float32(1/3))
```

```
"0.33333334326744079590"
```

```julia
@sprintf("%.20f", Float16(1/3))
```

```
"0.33325195312500000000"
```

```julia
BigFloat(1/3)   # Hmmm.
```

```
0.333333333333333314829616256247390992939472198486328125
```

```julia
BigFloat(1) / BigFloat(3)
```

```
0.3333333333333333333333333333333333333333333333333333333333333333333333333333348
```

```julia
BigFloat("0.3")
```

```
0.3000000000000000000000000000000000000000000000000000000000000000000000000000017
```

```julia
BigFloat("0.3", precision=500)
```

```
0.30000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Computation with `BigFloat`s will be slow, so you wouldn't want to do matrix operations with a matrix of them.

Floating point details

With Float64, any number is stored as a base 2 number of the form:

$$(-1)^S \times 1.d \times 2^{e-1023} = (-1)^S \times 1.d_1 d_2 \ldots d_{52} \times 2^{e-1023}$$

where the computer uses base 2, $b = 2$, (so $d_i \in \{0, 1\}$) because base-2 arithmetic is faster than base-10 arithmetic. The leading 1 normalizes the number; i.e., ensures there is a unique representation for a given computer number. This avoids representing any number in multiple ways, e.g., either $1 = 1.0 \times 2^0 = 0.1 \times 2^1 = 0.01 \times 2^2$. For a double, we have 8 bytes=64 bits. Consider our representation as (S, d, e) where S is the sign. The leading 1 is the *hidden bit* and doesn't need to be stored because it is always present. In general e is represented using 11 bits ($2^{11} = 2048$), and the subtraction takes the place of having a sign bit for the exponent. (Note that in our discussion we'll just think of e in terms of its base 10 representation, although it is of course represented in base 2.) This leaves $p = 52 = 64 - 1 - 11$ bits for d.

Overflow

Integer numbers can be represented exactly by Float64 up to 2^53.

```
function pri(x)
  @sprintf("%.20i", x)
end

pri(2.0^52)
```

```
"4503599627370496"
```

```
pri(2.0^52 + 1)
```

```
"4503599627370497"
```

```
pri(2.0^53)
```

```
"9007199254740992"
```

```
pri(2.0^53 + 1)
```

```
"9007199254740992"
```

```
pri(2.0^53 + 2)
```

```
"9007199254740994"
```

```
pri(2.0^63)
```

```
"9223372036854775808"
```

```
pri(2.0^70)     # No overflow here unlike Int64.
```

```
"1180591620717411303424"
```

```julia
pri(12345678123456781234.0)  # Not exact.
```

```
"12345678123456780288"
```

Float64 overflow is not until ~$2^{1023} \approx 10^{308}$.

```julia
function prf(x)
  @sprintf("%.20f", x)
end

prf(2.0^1022)
```

```
"449423283715578976932326297697256183404494244735576643183575202894331689513752407831771193306018840
```

```julia
prf(2.0^1023)
```

```
"898846567431157953864652595394512366808988489471153286367150405788663379027504815663542386612037680
```

```julia
prf(2.0^1024)
```

```
"Inf"
```

But Int64s that big do overflow.

```julia
pri(10^307)
```

```
"00000000000000000000"
```

```julia
pri(10^310)
```

```
"00000000000000000000"
```

Implications for comparisons and calculations

```julia
prf(1-2/3)
```

```
"0.33333333333333337034"
```

```julia
prf(1/3)
```

```
"0.33333333333333331483"
```

```julia
prf(2/3-1/3)
```

```
"0.33333333333333331483"
```

```julia
prf(0.3 - 0.2)
```

```
"0.09999999999999997780"
```

```julia
prf(0.1)
```

```
"0.10000000000000000555"
```

```
0.3 - 0.2 == 0.1
```

```
false
```

Here's an example of catastrophic cancellation:

```
prf(123456781234.56 - 123456781234.00)
```

```
"0.55999755859375000000"
```

And here the precision is that of the larger magnitude number:

```
prf(1.0 + 1e-8)
```

```
"1.00000000999999993923"
```

```
prf(1.0 + 1e-17)
```

```
"1.00000000000000000000"
```

Avoid multiplying/dividing many numbers

```
using Distributions
normd = Normal(0, 1)
samples = rand(normd, 1000);
prod(pdf.(normd, samples))  # Log-likelihood/log-density
```

```
0.0
```

```
sum(logpdf.(normd, samples))
```

```
-1446.1968973565636
```

```
sum(logpdf.(normd, samples[1:100]))
```

```
-137.28793148618522
```

We see that a likelihood with only as many as 1000 terms underflows. (In other cases it could overflow.)

Linear algebra

We'll see that linear algebra operations heavily exploit Julia's multiple dispatch system, calling the most appropriate method for different kinds of input matrices.

Arithmetic

```
A = rand(3, 3);
B = rand(3, 3);
A + B   # Element-wise
A .+ B  # Element-wise
```

```
A * B   # Matrix multiplication
A .* B  # Element-wise multiplication
```

Solving systems of equations / inversion

```
using LinearAlgebra

A = rand(3, 3);

AtA = A'A;    # A' * A
b = rand(3);
AtA \ b       # Solve system of equations (AtA^{-1} b)
```

```
3-element Vector{Float64}:
 -5839.367752001243
   237.62992428763377
  1388.1477149501256
```

```
inv(AtA) * b  # Not as efficient
```

```
3-element Vector{Float64}:
 -5839.367752001244
   237.62992428763377
  1388.1477149501254
```

```
eigvals(AtA)
```

```
3-element Vector{Float64}:
 1.0406188999502348e-5
 0.20302101952602597
 2.8527433608929575
```

```
det(AtA)      # Be careful of over/underflow!
```

```
6.026919865436598e-6
```

```
logdet(AtA)
```

```
-12.019274479480647
```

```
tr(A)
```

```
1.3719983709419976
```

```
chol = cholesky(AtA)
```

```
Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 0.290417  1.25712    1.00661
```

```
        0.605775  -0.103239
                   0.0139545
```

```
chol \ b                # Automatically exploits triangularity.
```

```
3-element Vector{Float64}:
 -5839.3677520089905
   237.62992428794826
  1388.1477149519676
```

```
chol.U \ (chol.L \ b)   # Manual equivalent solution.
```

```
3-element Vector{Float64}:
 -5839.3677520089905
   237.62992428794826
  1388.1477149519676
```

```
typeof(chol)            # Special type of object
```

```
Cholesky{Float64, Matrix{Float64}}
```

```
typeof(chol.U)          # Special kind of matrix.
```

```
UpperTriangular{Float64, Matrix{Float64}}
```

This works in Julia but is having problems via Quarto, so I'll just paste in some timings.

```
using BenchmarkTools

n = 5000
A = rand(n, n);
AtA = A'A;
b = rand(n);
@btime AtA \ b;       # Solve system of equations (AtA^{-1} b) via Gaussian elimination (LU)
# 633.046 ms (6 allocations: 190.81 MiB)
@btime inv(AtA) * b;  # Not as efficient
#  2.538 s (8 allocations: 193.25 MiB)
@btime cholesky(AtA) \ b;  # Best if matrix is positive definite.
#  546.957 ms (5 allocations: 190.77 MiB)
@btime chol = cholesky(AtA); chol.U \ (chol.L \ b);  # As good but verbose.
#  520.021 ms (3 allocations: 190.73 MiB)
```

In principle, the Cholesky approach should involve $n^3/6$ calculations and the Gaussian elimination $n^3/3$, but we don't see a two-fold difference here in practice.

Spectral (eigen) decomposition

```
Γ = eigvecs(AtA);
  = eigvals(AtA);
result = Γ * diagm( ) * Γ';
```

```
result[1:3,1:3]
```

```
3×3 Matrix{Float64}:
 0.0843421  0.365089  0.292337
 0.365089   1.94732   1.20289
 0.292337   1.20289   1.02412
```

```
AtA[1:3,1:3]
```

```
3×3 Matrix{Float64}:
 0.0843421  0.365089  0.292337
 0.365089   1.94732   1.20289
 0.292337   1.20289   1.02412
```

```
result == AtA
```

```
false
```

```
isapprox(result, AtA)
```

```
true
```

```
result   AtA   # \approx TAB
```

```
true
```

Don’t forget floating point issues

```
det(AtA)      # Could easily over/underflow for large matrices.
```

```
6.026919865436598e-6
```

```
logdet(AtA)
```

```
-12.019274479480647
```

Here’s a positive definite matrix (mathematically) that has all real, positive eigenvalues. On a computer, it’s not positive definite, and therefore not invertible/full rank.

```
xs = 0:99
# Compute distance matrix.
dists = abs.(xs .- xs')  # Using broadcasting with ' (an "outer" operation).
# Create correlation matrix.
corr_matrix = exp.(-(dists/10).^2)
# Compute eigenvalues and get last 20
eigvals(corr_matrix)[1:20]
```

```
20-element Vector{Float64}:
 -5.050107193681261e-15
 -3.975288045003424e-15
 -3.0343039148117795e-15
```

```
 -2.3309913036821875e-15
 -1.5093549661154687e-15
 -1.3511621460923302e-15
 -1.2848549714671506e-15
 -1.0472866233731733e-15
 -8.644575073124488e-16
 -8.364136467422076e-16
 -7.989065965702987e-16
 -6.829697931404996e-16
 -6.091008178138255e-16
 -6.030845253775599e-16
 -5.700968077566563e-16
 -5.37906121536584e-16
 -4.615826595950015e-16
 -4.4365869197860045e-16
 -4.0894133652379844e-16
 -3.580464402267892e-16
```

```
chol = cholesky(corr_matrix);
```

```
LoadError: PosDefException: matrix is not positive definite; Cholesky factorization failed.
PosDefException: matrix is not positive definite; Cholesky factorization failed.

Stacktrace:
 [1] checkpositivedefinite
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/factorization.jl:67 [inlin
 [2] cholesky!(A::Hermitian{Float64, Matrix{Float64}}, ::NoPivot; check::Bool)
   @ LinearAlgebra /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/cholesky.jl:
 [3] cholesky!
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/cholesky.jl:267 [inlined]
 [4] cholesky!(A::Matrix{Float64}, ::NoPivot; check::Bool)
   @ LinearAlgebra /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/cholesky.jl:
 [5] cholesky! (repeats 2 times)
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/cholesky.jl:295 [inlined]
 [6] cholesky
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/cholesky.jl:401 [inlined]
 [7] cholesky(A::Matrix{Float64})
   @ LinearAlgebra /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/cholesky.jl:
 [8] top-level scope
   @ In[74]:1
```

One should be able to use an approximate pivoted Cholesky that sets diagonal elements to zero corresponding to the rank deficiency. I'm having trouble seeing how to do that.

This does seem to work to solve the system of equations. We'd have to investigate to know what Julia is doing behind the scenes, but it's probably using a pivoted LU decomposition.

```
b = rand(100);
out = corr_matrix \ b;
```

Smart factorization

```
n = 500
A = rand(n, n);
```

```
typeof(factorize(A'A))
```

```
Cholesky{Float64, Matrix{Float64}}
```

```
typeof(factorize(A))           # I wouldn't have guessed that A is invertible!
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
```

```
typeof(factorize(A[:,1:10]))
```

```
QRPivoted{Float64, Matrix{Float64}, Vector{Float64}, Vector{Int64}}
```

The orthogonal matrices generated by certain factorizations can most efficiently be worked with by having them treated as "matrix-backed, function-based linear operators".

```
QRresult = qr(A[:,1:10]);
typeof(QRresult.Q)
```

```
LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
```

```
QRresult.Q * rand(10);
```

Use known structure!

```
using BenchmarkTools

n = 5000
A = rand(n, n);
AtA = A'A;
b = rand(n);
chol = cholesky(AtA);
typeof(chol.U)
@btime chol.U \ b;
```

```
  6.227 ms (3 allocations: 39.12 KiB)
```

```
U_dense = Matrix(chol.U);
@btime U_dense \ b;
```

```
  14.987 ms (2 allocations: 39.11 KiB)
```

```
@btime logdet(chol.U);
```

```
  104.467  s (2 allocations: 32 bytes)
```

```
@btime logdet(U_dense);
```

```
  603.875 ms (5 allocations: 190.77 MiB)
```

```
@btime sum(log.(diag(chol.U)));
```

```
  100.240  s (7 allocations: 78.27 KiB)
```

```
using SparseArrays

n = 5000;
A = Matrix(1.0I, n, n);
A[1,3] = 7.7;
A[5,9] = 2.3;
sA = sparse(A);

sizeof(A)
```

```
200000000
```

```
sizeof(sA)  # Ok, so that's not helpful given the pointers involved.
```

```
40
```

```
# sA. TAB  # What are the components of `sA`?
sizeof(sA.colptr) + sizeof(sA.nzval) + sizeof(sA.rowval)
```

```
120040
```

```
b = rand(n);
@btime A * b;
```

```
  9.456 ms (2 allocations: 39.11 KiB)
```

```
@btime sA * b;
```

```
  10.823  s (2 allocations: 39.11 KiB)
```