

Notes 9: Numerics

Chris Paciorek

2025-02-20

Table of contents

Introduction	1
Random number generation	2
The seed	2
Generators	2
Distributions	3
Floating point issues	4
Integer and floating point types	4
Floating point precision	5
Floating point details	6
Overflow	6
Implications for comparisons and calculations	7
Avoid multiplying/dividing many numbers	8
Linear algebra	9
Arithmetic	9
Solving systems of equations / inversion	9
Spectral (eigen) decomposition	11
Don't forget floating point issues	11
Smart factorization	13
Use known structure!	14
Optimization	15

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, 0.9531246272848422]
```

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 when working in parallel that I have not absorbed yet.

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

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

```
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).

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

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

Distributions

As with distributions in SciPy in Python, we first define the distribution of interest and then carry out distributional operations with it.

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

Precompiling Distributions

QuadGK

StatsFuns → StatsFunsChainRulesCoreExt

Distributions

3 dependencies successfully precompiled in 6 seconds. 41 already precompiled.

10-element Vector{Float64}:

```
0.4542940512832609
0.27004146470644036
0.3870672630509891
0.21116529046475746
0.5041249425689261
0.2784321400052276
0.6829933994934323
0.1295297689806119
0.43795698919772685
0.39395847238977155
```

using Plots

```
grid = 0:.01:1
plot(grid, pdf.(beta_dist, grid))
pdf.(beta_dist, beta_samples[1:5])
```

5-element Vector{Float64}:

```
1.2086291289041646
2.300088115277408
```

```
1.6389245754521693
2.4529451430316644
0.9144258514892248
```

```
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 $-2^{63} \dots 2^{63}$. 32 bit integers can represent $-2^{31} \dots 2^{31}$.

Similarly for 16 and 128 bit integers.

Values outside that range *overflow*.

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

```
4611686018427387904
```

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

```
0
```

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

```
-9223372036854775808
```

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

```
-9223372036854775808
```

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

```
9223372036854775808
```

```
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
```



```
"0.33333333333333337034"
```

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

```
"0.3333333333333331483"
```

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

```
"0.3333333333333331483"
```

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

```
"0.0999999999999997780"
```

```
prf(0.1)
```

```
"0.1000000000000000555"
```

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

```
false
```

Here's an example of catastrophic cancellation:

```
prf(123456781234.56)
```

```
"123456781234.55999755859375000000"
```

```
prf(123456781234.00)
```

```
"123456781234.00000000000000000000"
```

```
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"
```

```
1.0 + 1e-17 == 1.0
```

```
true
```

Avoid multiplying/dividing many numbers

How large does n need to be to see underflow here? In other cases it could overflow.

```
using Distributions  
normd = Normal(0, 1)
```

```

samples = rand(normd, n);
prod(pdf.(normd, samples)) # Log-likelihood/log-density

sum(logpdf.(normd, samples))

```

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 addition
A .+ B; # Element-wise addition
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}:
-1.3254878978759739
 5.5034206079487324
 0.30748299868104345

```

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

```

3-element Vector{Float64}:
-1.3254878978759748
 5.503420607948734
 0.30748299868104345

```

```
eigvals(AtA)
```

```

3-element Vector{Float64}:
 0.03992762277861558
 0.22181922851410624
 2.0138442890779333

```

```

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

0.017836043877867285
logdet(AtA)

-4.026533932146638
tr(A)

1.5834783240781707
chol = cholesky(AtA)

Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 0.957996  0.309353  0.873741
           0.213918  0.168095
                    0.651687

We have $ (U^{\text{top}}U)^{-1}b = U^{-1}(U^{\text{top}})^{-1}b $, which in Julia is implemented internally.
chol \ b      # Automatically exploits triangularity.

3-element Vector{Float64}:
-1.3254878978759743
 5.503420607948734
 0.30748299868104323
chol.U \ (chol.L \ b) # Manual equivalent solution.

3-element Vector{Float64}:
-1.325487897875974
 5.503420607948734
 0.3074829986810431
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}:
 0.917756  0.296359  0.83704
 0.296359  0.14146  0.306253
 0.83704   0.306253  1.21638

```

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

```

3×3 Matrix{Float64}:
 0.917756  0.296359  0.83704
 0.296359  0.14146  0.306253
 0.83704   0.306253  1.21638

```

```
result == AtA
```

```
false
```

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

```
true
```

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

```
true
```

Don't forget floating point issues

```

n=500;
A = rand(n, n);
det(A'A)

```

Inf

```
logdet(A'A)
```

```
1364.3425695000474
```

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 [inlined]
[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] chollesky!
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/chollesky.jl:267 [inlined]
[4] chollesky!(A::Matrix{Float64}, ::NoPivot; check::Bool)
   @ LinearAlgebra /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/chollesky.jl:
[5] chollesky! (repeats 2 times)
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/chollesky.jl:295 [inlined]
[6] chollesky
   @ /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/chollesky.jl:401 [inlined]
[7] chollesky(A::Matrix{Float64})
   @ LinearAlgebra /system/linux/julia-1.10.4/share/julia/stdlib/v1.10/LinearAlgebra/src/chollesky.jl:
[8] top-level scope
   @ In[75]: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;
```

6.012 ms (3 allocations: 39.12 KiB)

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

14.518 ms (2 allocations: 39.11 KiB)

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

105.739 s (2 allocations: 32 bytes)

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

644.817 ms (5 allocations: 190.77 MiB)

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

100.387 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) # 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;
```

9.412 ms (2 allocations: 39.11 KiB)

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

20.386 s (2 allocations: 39.11 KiB)

Optimization

A good place to start for a variety of standard optimization algorithms is [Optim.jl](#).

The main function is `optimize()`, and you provide the optimization method you want to use as an argument.

Some of the optimizers include:

- Derivative-free
 - Nelder-Mead
 - Simulated annealing
 - Particle swarm
- Gradient-based
 - Adam and AdaMax
 - Conjugate gradient
 - Gradient-descent
 - BFGS and LBFGS (limited-memory BFGS)
- Hessian-based
 - Newton
 - Newton with trust region (recommended by the package developers)
 - Interior point Newton

This collection is somewhat similar to the optimizers available with `optim` in R and with `scipy.optimize` in Python.