

Julia 科学计算开源工具链 - 从了解到入门

刘贵欣

通常从 MATLAB 或 Python 转过来的用户有以下疑问：

这个函数为什么报错了，怎么跟我预期的不一样？

对于XXX问题，该用什么函数？

- 多维数组
- 基本线性代数
- 运行时间测试
- 数据读写
- 运筹优化
- 张量网络

- Julia 具有原生的数组实现，并提供了许多方便的基本函数。

函数	描述
<code>eltype(A)</code>	A 中元素的类型
<code>length(A)</code>	A 中元素的数量
<code>ndims(A)</code>	A 的维数
<code>size(A)</code>	一个包含 A 各个维度上元素数量的元组
<code>size(A,n)</code>	A 第 n 维中的元素数量
<code>axes(A)</code>	一个包含 A 有效索引的元组
<code>axes(A,n)</code>	第 n 维有效索引的范围
<code>eachindex(A)</code>	一个访问 A 中每一个位置的高效迭代器
<code>stride(A,k)</code>	在第 k 维上的间隔 (stride) (相邻元素间的线性索引距离)
<code>strides(A)</code>	包含每一维上的间隔 (stride) 的元组

```
julia> A = randn(4,4)
4×4 Matrix{Float64}:
-0.0653551  0.800785  0.399488 -1.12979
 0.280816 -1.70278  0.422952  0.560208
-0.259381 -0.630157 -0.624698 -1.08722
-0.413353  0.172939  0.210217  0.973692

julia> length(A)
16

julia> ndims(A)
2

julia> size(A)
(4, 4)

julia> strides(A)
(1, 4)
```

- Julia 具有原生的数组实现，可用于数组遍历的函数有：eachindex, eachrow, eachcol 等

```
julia> A = collect(reshape(1:16, 4, 4))
4×4 Matrix{Int64}:
 1  5  9 13
 2  6 10 14
 3  7 11 15
 4  8 12 16

julia> for xi in eachindex(A)
    println(A[xi])
end

1
2
3
4
5
6
7
8
```

```
julia> for row_i in eachrow(A)
    println(row_i)
end

[1, 5, 9, 13]
[2, 6, 10, 14]
[3, 7, 11, 15]
[4, 8, 12, 16]

julia> for col_i in eachcol(A)
    println(col_i)
end

[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 16]
```

- Julia 具有原生的数组实现，还可用于数组遍历的函数有：length

```
julia> A = collect(reshape(1:16, 4, 4))
4×4 Matrix{Int64}:
 1  5  9 13
 2  6 10 14
 3  7 11 15
 4  8 12 16

julia> length(A)
16

julia> for xi in 1:length(A)
           println(A[xi])
       end

1
2
3
4
5
6
7
8
```

MATLAB

`L = length(X)` 返回 X 中最大数组维度的长度，等价于 `max(size(X))`

```
X = zeros(3,7);
L1 = length(X)
```

L1 = 7

```
L2 = max(size(X))
```

L2 = 7

- Julia 拥有真正的一维数组。列向量的大小为 N ，而不是 $N \times 1$ 。

```
julia> mat = reshape(collect(1:16), (4,4))
4×4 Matrix{Int64}:
 1  5  9 13
 2  6 10 14
 3  7 11 15
 4  8 12 16

julia> vec_slice = mat[1, :]
4-element Vector{Int64}:
 1
 5
 9
13

julia> vec_mat = reshape([1 9;5 13], :, 1)
4×1 Matrix{Int64}:
 1
 5
 9
13

julia> vec_slice == vec_mat
false
```

MATLAB

```
mat = reshape(1:16,[4, 4]);
```

```
row_slice = mat(1, :);
```

```
col_slice = mat(:, 1);
```

名称	值	大小	类
col_slice	[1;2;3;4]	4x1	double
mat	4x4 double	4x4	double
row_slice	[1,5,9,13]	1x4	double

- Julia 数组在分配给另一个变量时不会被复制。在 $A=B$ 之后，改变 B 的元素也会改变 A 的元素。

```
julia> x = y = z = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> z[1] = 4
4

julia> y
3-element Vector{Int64}:
 4
 2
 3

julia> x
3-element Vector{Int64}:
 4
 2
 3
```

MATLAB

1

2

3

```
x = [1,2,3];
y = x;
x(1) = 4;
```

名称	值	大小
x	[4,2,3]	1x3
y	[1,2,3]	1x3

1. 与其他语言的显著差异: <https://cn.julialang.org/JuliaZH.jl/latest/manual/noteworthy-differences/>

基本线性代数

- Julia 包含了绝大多数常见基本线性代数操作。在调用函数前请: `using LinearAlgebra`

```
julia> using LinearAlgebra

julia> A = [4 9 2; 3 5 7; 8 1 6]
3×3 Matrix{Int64}:
 4  9  2
 3  5  7
 8  1  6

julia> tr(A)
15

julia> det(A)
360.0

julia> inv(A)
3×3 Matrix{Float64}:
 0.0638889 -0.144444  0.147222
 0.105556  0.0222222 -0.0611111
-0.102778  0.188889 -0.0194444
```

```
julia> diagvec = diag(A)
3-element Vector{Int64}:
 4
 5
 6

julia> diag(diagvec)
ERROR: ArgumentError: use
Stacktrace:
 [1] diag(A::Vector{Int64})
      @ LinearAlgebra D:\Prog
 [2] top-level scope
      @ REPL[15]:1

julia> diagm(diagvec)
3×3 Matrix{Int64}:
 4  0  0
 0  5  0
 0  0  6
```

注意与 `np.diag()` 和 MATLAB 中的 `diag` 函数的区别, 这两者的行为是

- 如果是二维数组, 返回对角元素;
- 如果是一维数组, 返回对角为该一维数组, 其余元素为 0 的稠密矩阵;

本征值分解

```
julia> using LinearAlgebra

julia> A = [5 3 4; 3 2 6; 4 6 8]
3×3 Matrix{Int64}:
 5  3  4
 3  2  6
 4  6  8

julia> F = eigen(A)
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 -1.7492969445703608
  2.388393961699327
 14.360902982871064
vectors:
3×3 Matrix{Float64}:
 0.0910829  0.877897  0.470107
-0.864349  -0.164756  0.475138
 0.494575   -0.449614  0.743803

julia> F.v ← 按Tab 补全元素
values  vectors
```

```
julia> eigvecs(A) ←
3×3 Matrix{Float64}:
 0.0910829  0.877897  0.470107
-0.864349  -0.164756  0.475138
 0.494575   -0.449614  0.743803

julia> eigvals(A) ←
3-element Vector{Float64}:
 -1.7492969445703783
  2.3883939616993146
 14.360902982871064

julia> eigmin(A) ←
-1.7492969445703783

julia> eigmax(A) ←
14.360902982871064
```

只取本征向量

只取本征值

只取最小本征值

只取最大本征值

- Julia 具有丰富的矩阵类型，同时也为这些类型开发了专用的高效实现。以对称三角矩阵和对角矩阵为例，

```
julia> n=4; dv=rand(n); ev=rand(n-1);

julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Float64, Vector{Float64}}:
 0.246372  0.115546  .          .
 0.115546  0.633327  0.112879  .
 .         0.112879  0.347459  0.518732
 .         .         0.518732  0.0279973
```

```
julia> n=2000; a=rand(n); b=rand(n-1);

julia> mat_tri = SymTridiagonal(a,b);

julia> @btime eigmin(mat_tri)
 727.500 μs (13 allocations: 376.03 KiB)
-1.2351301154307668

julia> mat_all = Matrix(mat_tri);

julia> @btime eigmin(mat_all)
 683.822 ms (13 allocations: 31.24 MiB)
-1.2351301154307754
```

对称三角矩阵

```
julia> I(4) ← 单位矩阵
4×4 Diagonal{Bool, Vector{Bool}}:
 1  .  .  .
 .  1  .  .
 .  .  1  .
 .  .  .  1
```

```
julia> n=1000; vec1 = rand(n); mat1 = randn(n,n);

julia> diag_mat = diagm(vec1); ← 全矩阵
julia> diagonal = Diagonal(vec1); ← 稀疏矩阵

julia> @btime diag_mat * mat1;
 11.487 ms (2 allocations: 7.63 MiB)

julia> @btime diagonal * mat1;
 1.194 ms (2 allocations: 7.63 MiB)
```

对角矩阵

- Julia 具有丰富的矩阵类型，同时也为这些类型开发了专用的高效实现。

Matrix type	+	-	*	\	Other functions with optimized methods
Symmetric				MV	inv, sqrt, exp
Hermitian				MV	inv, sqrt, exp
UpperTriangular			MV	MV	inv, det, logdet
UnitUpperTriangular			MV	MV	inv, det, logdet
LowerTriangular			MV	MV	inv, det, logdet
UnitLowerTriangular			MV	MV	inv, det, logdet
UpperHessenberg				MM	inv, det
SymTridiagonal	M	M	MS	MV	eigmax, eigmin
Tridiagonal	M	M	MS	MV	
Bidiagonal	M	M	MS	MV	
Diagonal	M	M	MV	MV	inv, det, logdet, /
UniformScaling	M	M	MVS	MVS	/

- M : matrix-Matrix
- V : matrix-Vector
- S : matrix-Scalar

- Julia 中也提供了诸多常见的矩阵分解方法。

Julia 中的矩阵分解方法（不完全统计）

矩阵 A 的特性	矩阵分解方法
正定	Cholesky (cholesky)
稠密的对称 / 厄米	Bunch-Kaufman (bunchkaufman)
稀疏的对称 / 厄米	<u>LDLt</u> (ldlt)
三角	LU (lu)
实对称的三对角	<u>LDLt</u> (ldlt)
一般方阵	LU (lu)
一般非方阵	QR (qr)

Bunch-Kaufman 分解: $A = P' * U * D * U' * P$

```
julia> using LinearAlgebra

julia> A = [15 3 4; 3 22 1; 4 1 8]
3×3 Matrix{Int64}:
 15   3   4
   3  22   1
   4   1   8

julia> F = bunchkaufman(A); # F is a BunchKaufman object

julia> F.P' * F.U * F.D * F.U' * F.P
3×3 Matrix{Float64}:
15.0  3.0  4.0
 3.0 22.0  1.0
 4.0  1.0  8.0
```

通常我们需要查看一段代码的运行时间，一个直接的想法是使用 @time 宏

```
julia> a = randn(2^22);

julia> @time sum(a);
0.041319 seconds (36.73 k allocations: 2.431 MiB, 91.93% compilation time)

julia> @time sum(a);
0.003185 seconds (1 allocation: 16 bytes)

julia> @time sum(a);
0.001774 seconds (1 allocation: 16 bytes)

julia> @time sum(a);
0.001813 seconds (1 allocation: 16 bytes)
```

但是，由于著名的编译延迟问题，@time 第一次得到的结果并不准确

为了避免编译延迟带来的影响，一个可行的方案是会通过多次运行取平均（不含第一次），这就需要用到 BenchmarkTools


```
julia> using BenchmarkTools

julia> a = rand(2^22);

julia> @btime sum(a);
1.006 ms (1 allocation: 16 bytes)

julia> @benchmark sum(a)
BenchmarkTools.Trial: 3902 samples with 1 evaluation.
Range (min ... max): 1.007 ms ... 4.140 ms | GC (min ... max): 0.00% ... 0.00%
Time (median): 1.211 ms | GC (median): 0.00%
Time (mean ± σ): 1.262 ms ± 191.235 μs | GC (mean ± σ): 0.00% ± 0.00%

Histogram: frequency by time
1.01 ms 2 ms <
```



Memory estimate: 16 bytes, allocs estimate: 1.

但是，对于一些比较大型的问题，采用这种多次运行的方案将会非常耗时。此时推荐 TimerOutputs.jl

```
1 using TimerOutputs
2
3 const to = TimerOutput();
4
5 function time_test()
6     @timeit to "nest 1" begin
7         sleep(0.1)
8         # 3 calls to the same label
9         @timeit to "level 2.1" sleep(0.03)
10        @timeit to "level 2.1" sleep(0.03)
11        @timeit to "level 2.1" sleep(0.03)
12        @timeit to "level 2.2" sleep(0.2)
13    end
14    @timeit to "nest 2" begin
15        @timeit to "level 2.1" sleep(0.3)
16        @timeit to "level 2.2" sleep(0.4)
17    end
18 end
19
20 time_test()
21 show(to)
```

		Time			Allocations		
Tot / % measured:		1.58s / 75.0%			21.5MiB / 0.0%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
nest 2	1	762ms	64.1%	762ms	2.05KiB	48.5%	2.05KiB
level 2.2	1	435ms	36.6%	435ms	144B	3.3%	144B
level 2.1	1	327ms	27.5%	327ms	448B	10.4%	448B
nest 1	1	426ms	35.9%	426ms	2.17KiB	51.5%	2.17KiB
level 2.2	1	218ms	18.4%	218ms	144B	3.3%	144B
level 2.1	3	100ms	8.4%	33.4ms	432B	10.0%	144B

如果临时需要将小矩阵输出到 txt 文件中，那么可以考虑使用 DelimitedFiles (stdlib) 或 Printf (stdlib)

```
1 using DelimitedFiles
2
3 a = 1:10
4 b = a.^2 + a
5
6 open("result1.dat", "w") do io
7     println(io, "a    b")
8     writedlm(io, [a b])
9 end
```

≡ result1.dat

1	a	b
2	1	2
3	2	6
4	3	12
5	4	20
6	5	30
7	6	42
8	7	56
9	8	72
10	9	90
11	10	110

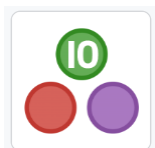
```
12 using Printf
13
14 x = randn(10)
15 y = x.^2 + x
16
17 ∨ open("result2.dat", "w") do io
18     println(io, "x      y")
19     ∨ for (ix, iy) in zip(x, y)
20         @printf(io, "%.4f    %.4f \n", ix, iy)
21     end
22 end
23
```

≡ result2.dat

1	x	y
2	-0.1889	-0.1532
3	-0.3538	-0.2286
4	0.6474	1.0666
5	-2.1650	2.5224
6	-2.5139	3.8058
7	-1.5465	0.8452

大规模数据读写

HDF5（Hierarchical Data Format 5）是一种用于高性能计算领域的文件格式



JuliaIO

- HDF5.jl: Julia 语言封装的HDF5接口
- JLD2.jl: 纯 Julia 实现的数据读写包, 兼容 HDF5 格式

```
c = h5open("mydata.h5", "r") do file
    read(file, "A")
end
```

HDF5.jl

```
jldopen("example.jld2", "w") do file
    file["bigdata"] = randn(5)
end
```

JLD2.jl

源代码

```
1  using JLD2
2  using Printf
3  using BenchmarkTools
4
5  n = 10
6  a = rand(n, n)
7
8  function largeio(a)
9      jldopen("example.jld2", "w") do file
10         file["bigdata"] = a
11     end
12 end
13
14 function largeio2(a)
15     open("example2.dat", "w") do io
16         for ix in eachindex(a)
17             @printf(io, "%.4f\n", a[ix])
18         end
19     end
20 end
```

性能比较表

n	largeio	largeio2	加速比
10	3.7ms	140 μs	0.03x
100	3.8ms	1.14ms	0.3x
1000	12ms	116ms	9.7x
2000	37ms	508ms	13.7x
4000	121ms	2.02s	16.7x

总结：当矩阵规模较大时，JLD2 写入加速效果显著

接下来，我们将开始在开源世界的冒险！



是 Julia 官方提供的科学计算商业服务平台。Julia 用户可以使用它来探索整个生态

Packages

Register package

All Packages

Q Search Packages

Topics

optimization x

Licences

Q Licenses...

Showing 209 of 10248 packages

下载量

GitHub 星标数

点击筛选该团队所有的包

jump-dev / JuMP

Modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear)

conic-programs

julia

linear-programming

mathematical-modelling

mathematical-programming

mixed-integer-programming

modeling-language

nonlinear-programming

optimization

semidefinite-programming

[Documentation](#)

 Source

↓ 6.0k

☆ 2058

MPL-2.0

1.16.0

JuliaNLSolvers / Optim

📄 9.8k  ☆ 1025 MIT 1.7.8

Optimization functions for Julia

Optim —— 单变量及多变量的无约束优化

jump-dev / JuMP

📄 6.0k  ☆ 2058 MPL-2.0 1.16.0

Modeling language for Mathematical Optimization (linear, mixed-integer, conic, semidefinite, nonlinear)

JuMP—— 求解器无关的代数建模语言+各大求解器接口

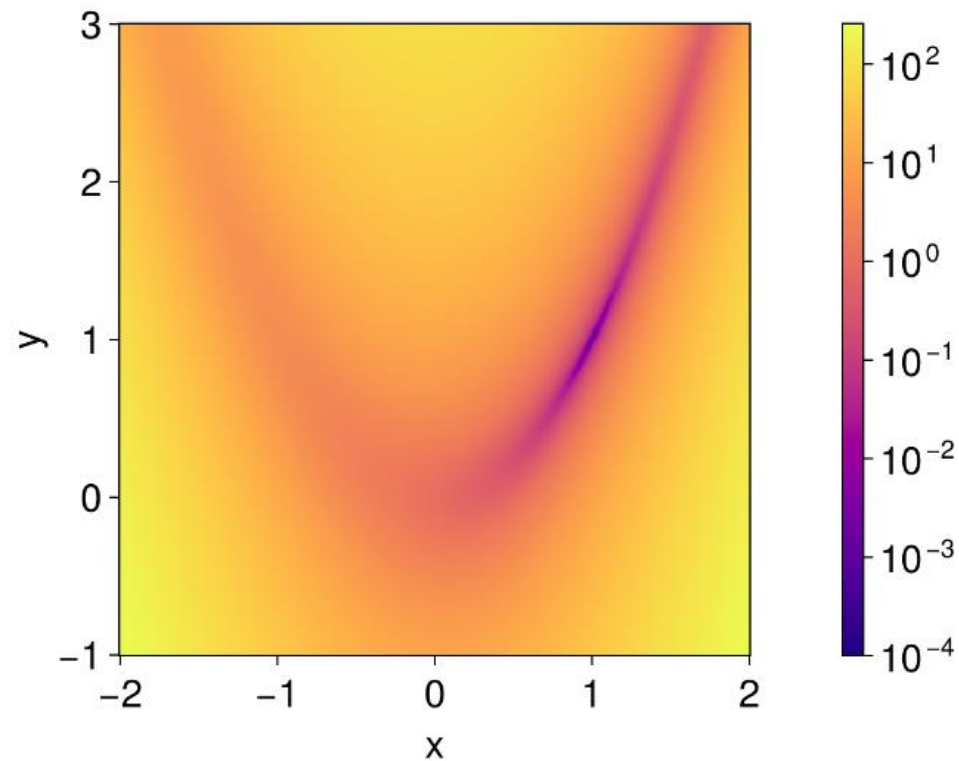
无约束优化两者都可， 约束优化多用 JuMP

Rosenbrock 函数

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

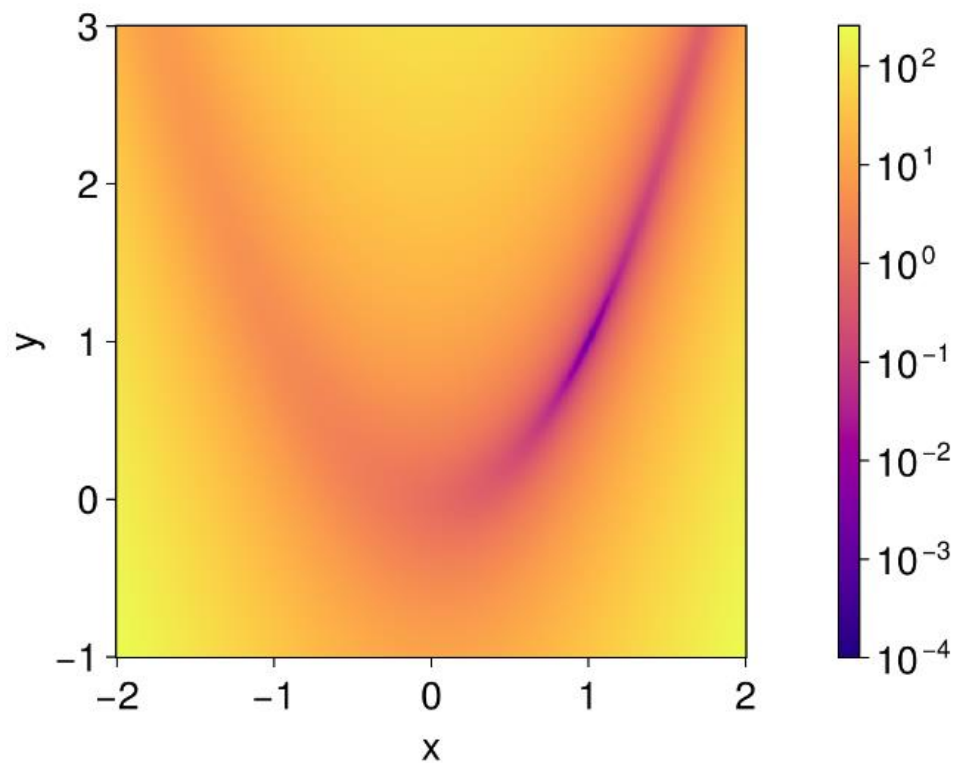
性质:

1. 当 $(x, y) = (a, a^2)$ 时, 取全局最小值: $f(x, y) = 0$
2. 全局最小值位于一处抛物线形谷内, 但全局最小值附近函数比较平缓, 故难以精确寻得全局最小值;



$a = 1, b = 10$

Rosenbrock 函数最小值



$a = 1, b = 10$

```
1 using Optim
2
3 rosenbrock(x) = (1.0 - x[1])^2 + 10.0 * (x[2] - x[1]^2)^2
4 result = optimize(rosenbrock, zeros(2), BFGS())
5 x, y = Optim.minimizer(result)
6 fmin = minimum(result)
7 println("x = $x, y = $y")
8 println("fmin = $fmin") | ✓
```

```
* Work counters
  Seconds run:   0 (vs limit Inf)
  Iterations:   11
  f(x) calls:   32
  ∇f(x) calls:  32
```

```
x = 0.9999999992667803, y = 0.999999998533542
fmin = 5.376111424228077e-19
```


JuMP

- 求解器无关的代数建模语言
- 各大求解器 wrapper (Ipopt, Gurobi)
- 线性/非线性规划
- 混合整数规划
- second-order conic
- semidefinite

```
1 using JuMP
2 import Ipopt
3 using Test
4
5 function example_rosenbrock()
6     model = Model{Ipopt.Optimizer}
7     set_silent(model)
8     @variable(model, x)
9     @variable(model, y)
10    @objective(model, Min, (1 - x)^2 + 10 * (y - x^2)^2)
11    optimize!(model)
12    Test.@test termination_status(model) == LOCALLY_SOLVED
13    Test.@test primal_status(model) == FEASIBLE_POINT
14    Test.@test objective_value(model) ≈ 0.0 atol = 1e-10
15    Test.@test value(x) ≈ 1.0
16    Test.@test value(y) ≈ 1.0
17    println("x = $(value(x)), y = $(value(y))")
18 end
19
20 example_rosenbrock()
```

```
x = 0.9999999999999999, y = 0.9999999999999999
```

爱因斯坦求和规则

向量 $A_\alpha = \overset{\alpha}{\text{---}} \textcircled{A}$

矩阵乘法 $\sum_\beta A_{\alpha\beta} B_{\beta\gamma} = \overset{\alpha}{\text{---}} \textcircled{A} \text{---}^\beta \textcircled{B} \text{---}^\gamma$

矩阵 $A_{\alpha\beta} = \overset{\alpha}{\text{---}} \textcircled{A} \text{---}^\beta$

矩阵求迹 $\text{Tr}(A_{\alpha\beta}) = \overset{\alpha}{\text{---}} \textcircled{A} \text{---}^\alpha$

3阶张量 $B_{\alpha\beta\gamma} = \overset{\alpha}{\text{---}} \textcircled{B} \text{---}^\beta \underset{\gamma}{\text{---}}$

张量缩并 $\sum_{\alpha,\gamma} A_{\alpha\delta} B_{\alpha\beta\gamma} C_{\gamma\epsilon} = \text{Diagram}$

首先，我们还是利用 **JuliaHub** 查找要用到的软件包，预备关键词：tensor 和 einsum，并根据搜索框返回结果多次尝试，筛选出的包如下：

ITensor / ITensors

📄 570 📈 ☆ 442 Apache-2.0 0.3.52
A Julia library for efficient tensor computations and tensor network calculations

chakravala / Grassmann

📄 24 📈 ☆ 429 AGPL-3.0 0.8.6
{Grassmann-Clifford-Hodge} multilinear differential geometric algebra

Jutho / TensorKit

📄 19 📈 ☆ 152 MIT 0.12.0
A Julia package for large-scale tensor computations, with a hint of category theory

[Documentation](#)

[Source](#)

under-Peter / OMEinsum

📄 53 📈 ☆ 150 MIT 0.7.5
One More Einsum for Julia! With runtime order-specification and high-level adjoints for AD

Jutho / TensorOperations

📄 180 📈 ☆ 374 MIT 4.0.7
Julia package for tensor contractions and related operations

mcabbott / Tullio

Σ

[automatic-differentiation](#) [einsum](#) [julia](#) [notation](#) [tensor](#)

 **bsc-quantic / EinExprs.jl** [Public](#)

首先，我们还是利用 **JuliaHub** 查找要用到的软件包，预备关键词：tensor 和 einsum，并根据搜索框返回结果多次组合尝试，筛选出的包如下：

ITensor / ITensors

↓ 570

☆ 442

Apache-2.0

0.3.52

A Julia library for efficient tensor computations and tensor network calculations

chakravala / Grassmann

↓ 24

☆ 429

AGPL-3.0

0.8.6

⟨Grassmann-Clifford-Hodge⟩ multilinear differential geometric algebra

Jutho / TensorKit

↓ 19

☆ 152

MIT

0.12.0

A Julia package for large-scale tensor computations, with a hint of category theory

Documentation

Source

under-Peter / OMEinsum

↓ 53

☆ 150

MIT

0.7.5

One More Einsum for Julia! With runtime order-specification and high-level adjoints for AD

Jutho / TensorOperations

↓ 180

☆ 374

MIT

4.0.7

Julia package for tensor contractions and related operations

mcabbott / Tullio

↓ 400

☆ 566

MIT

0.3.7

Σ

automatic-differentiation

einsum

julia

notation

tensor

[Jutho / TensorOperations](#)

↓ 180



☆ 374

MIT

4.0.7

Julia package for tensor contractions and related operations

[under-Peter / OMEinsum](#)

↓ 53



☆ 150

MIT

0.7.5

One More Einsum for Julia! With runtime order-specification and high-level adjoints for AD

```
1 using OMEinsum
2 using TensorOperations
3 using TimerOutputs
4
5 # Create a TimerOutput, this is the main type that keeps track of everything.
6 const to = TimerOutput()
7
8 a_cpu = randn(Float64, 4, 1000, 1000)
9 b_cpu = randn(Float64, 4, 1000, 1000)
10
11 @timeit to "einsum_cpu" @ein block_cpu[p, b, n, m] := a_cpu[p, n, o] * b_cpu[b, o, m]
12 @timeit to "tensor_cpu" @tensor block_cpu[p, b, n, m] := a_cpu[p, n, o] * b_cpu[b, o, m]
13
```

↑
:= 指生成新的张量

如果要做基于MPS (Matrix Product State) 量子多体模拟工作, 首选 ITensors

- Intelligent Indices
- Quantum Physics Tools
- Quantum Number Conserving ITensors
- MPS and MPO Algorithms

```
1 using ITensors
2
3 let
4     # For example:
5     i = Index(2,"i")
6     j = Index(3,"j")
7     T = randomITensor(i,j)
8     @show T
9 end
```

```
1 using ITensors
2 let
3     N = 100
4     sites = siteinds("S=1",N)
5
6     os = OpSum()
7     for j=1:N-1
8         os += "Sz",j,"Sz",j+1
9         os += 1/2,"S+",j,"S-",j+1
10        os += 1/2,"S-",j,"S+",j+1
11    end
12    H = MPO(os,sites)
13
14    psi0 = randomMPS(sites,10)
15
16    nsweeps = 5
17    maxdim = [10,20,100,100,200]
18    cutoff = [1E-10]
19
20    energy, psi = dmrg(H,psi0; nsweeps, maxdim, cutoff)
21
22    return
23 end
```



JuliaGPU 为多种GPU 平台实现了支持, 其中最好用的是 CUDA.jl。

CUDA.jl 不仅可以用于编写 CUDA内核, 也可直接使用Julia 语法操作多维数组。



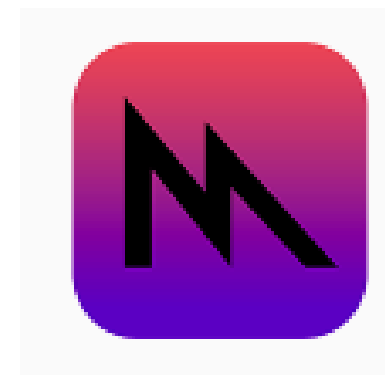
CUDA.jl



AMDGPU.jl



oneAPI.jl



Metal.jl

CUDA.jl 已经实现了Julia 中的绝大部分多维数组操作

```
julia> using CUDA

julia> CUDA.zeros(4,4)
4×4 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> mat = CUDA.randn(4,4)
4×4 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 1.42798      1.62652    -2.33491     1.83148
-1.32568      0.400546    2.15678     0.84477
 0.000137211  0.111303    0.922282    0.201765
 1.57575     -0.253458   -1.61895     0.80799

julia> CUDA.sum(mat)
6.3743014f0
```

```
julia> CUDA.svd(mat)
LinearAlgebra.SVD{Float32, Float32, CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}}
U factor:
4×4 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
-0.736945  0.592402  0.305386  0.112738
 0.443447  0.763351 -0.268347 -0.385539
 0.13807   0.217108 -0.410857  0.874642
-0.491124 -0.138634 -0.816042 -0.27139

singular values:
4-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 4.622511
 2.268726
 0.9148959
 0.3834122

Vt factor:
4×4 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
-0.522244 -0.190629  0.778703 -0.290763
-0.169453  0.58562  0.30319  0.732401
-0.54007  0.601525 -0.38213 -0.447738
 0.637865  0.5088  0.394556 -0.422583
```


cuTENSOR

NVIDIA 官方开发的高性能张量计算库，支持：

- 混合精度张量计算
- 复数-实数乘法操作
- 最高支持 64 维张量
- 常用张量操作（缩并，逐元素计算）
- ...

cuTensorNet

NVIDIA 官方基于cuTENSOR 开发的高性能张量网络库

- 创建张量网络结构
- 对任意张量网络搜索优化缩并路径
- 基于内存约束搜索sliced 缩并路径
- 创建张量网络态（量子电路模拟）
- 张量分解（QR & SVD）
- ...

先安装CUDA.jl。考虑到 cuTENSOR 和 cuTensorNet 的兼容性，将CUDA 版本设置为 11.8

```
julia> using CUDA

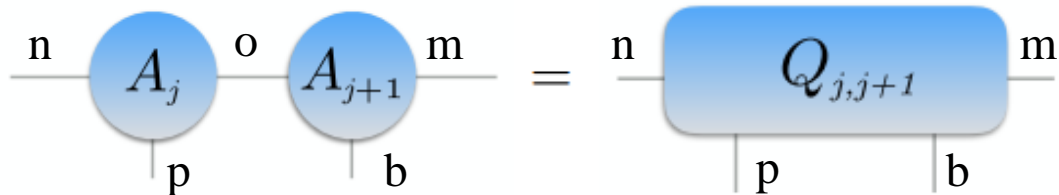
julia> CUDA.set_runtime_version!(v"11.8")
[ Info: Configure the active project to use CUDA 11.8; please re-start Julia for this to take effect.
```

然后安装 cuTENSOR 和 cuTensorNet

```
(example2023) pkg> add cuTENSOR
Resolving package versions...
```

```
(example2023) pkg> add cuTensorNet
Resolving package versions...
Installed cuTensorNet - v1.1.1
Updating `~/workspace/example2023/Project.toml`
[448d79b3] + cuTensorNet v1.1.1
Updating `~/workspace/example2023/Manifest.toml`
[448d79b3] + cuTensorNet v1.1.1
⚡ [b75408ef] + cuQuantum_jll v22.11.13+0
```

基于 cuTENSOR 的缩并



p b 维度为 4

n o m 维度为 dim

CPU: Tensoroperations.@tensor

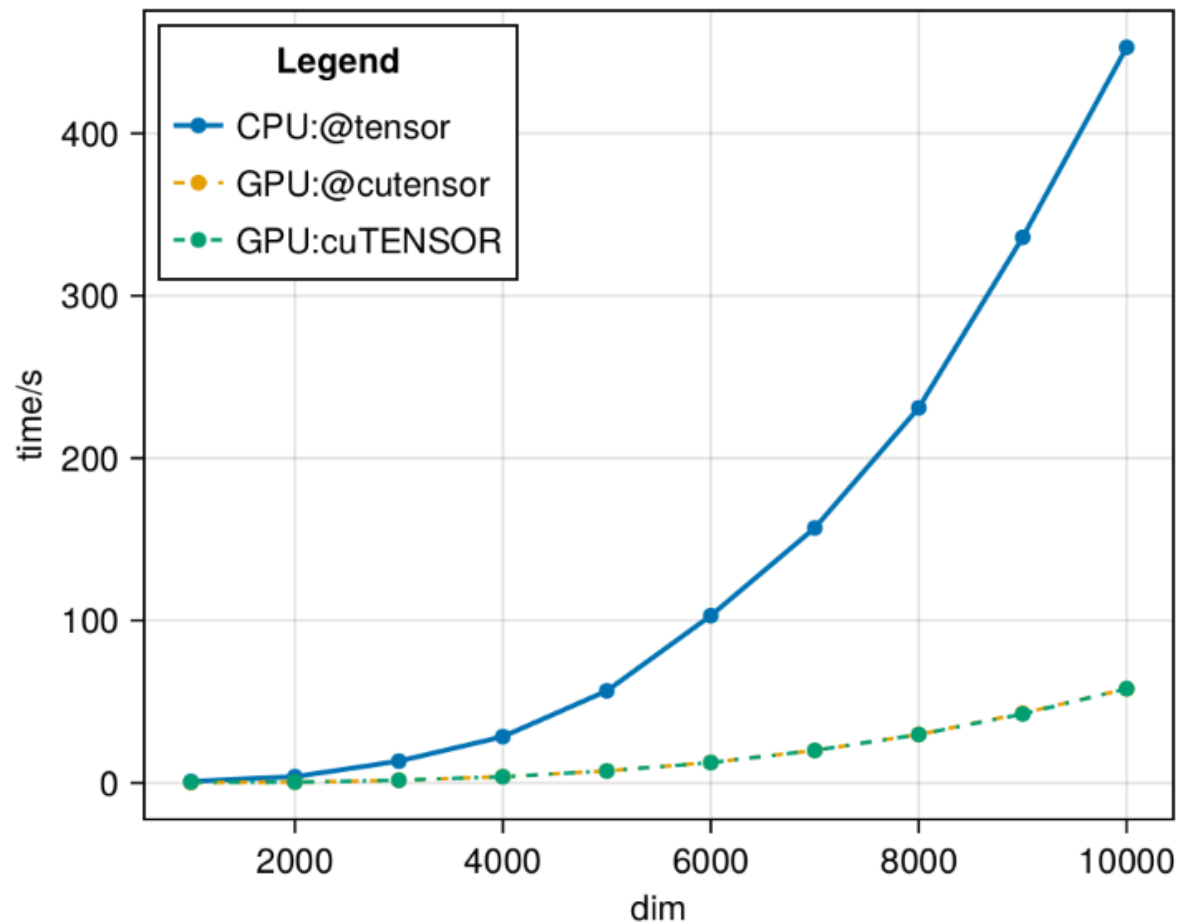
GPU: Tensoroperations.@cutensor

GPU: using cuTENSOR

```
modesA = ['p', 'b', 'n', 'm']
modesU = ['p', 'n', 'o']
modesV = ['b', 'o', 'm']

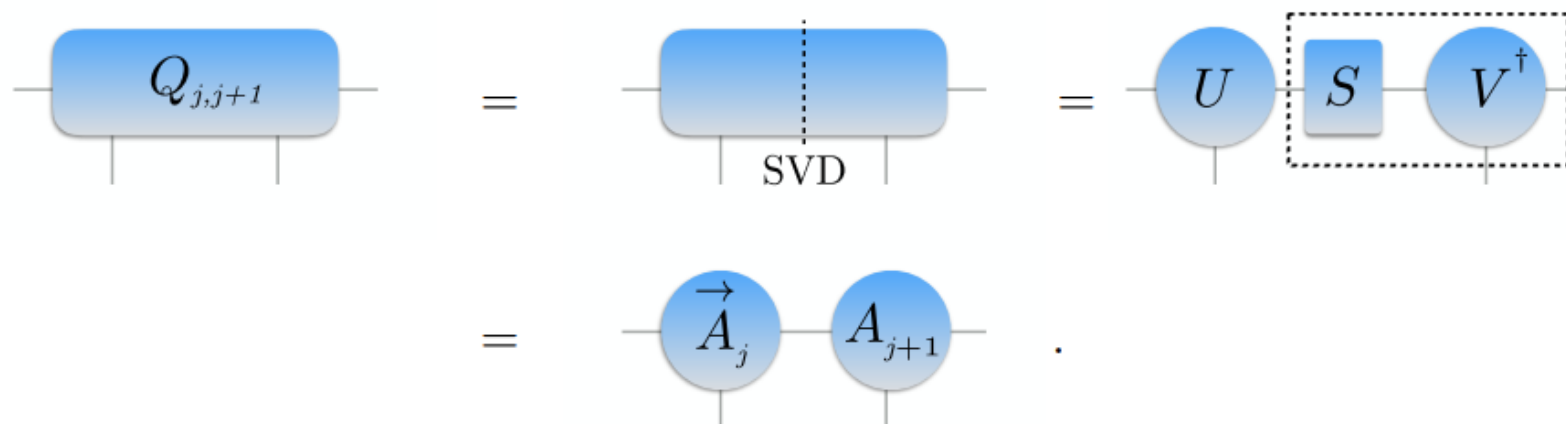
At = CuTensor(block_gpu, modesA)
Vt = CuTensor(b_gpu, modesV)

print("cuTENSOR einsum: ")
CUDA.@time At = Ut * Vt
```



1. 用 CUDA 做缩并更快
2. 在自己的包可性能无损地使用 cuTENSOR

基于cuTensorNet 的张量奇异值分解



```
22 block_gpu = CuArray(convert(Array, block_cpu))
23 u_gpu = CuArray{Float64}(undef, (4, dim_i, dim_i))
24 s_gpu = CuVector{Float64}(undef, (dim_i,))
25 v_gpu = CuArray{Float64}(undef, (4, dim_i, dim_i))
26
27 modesA = ['p', 'b', 'n', 'm']
28 modesU = ['p', 'n', 'o']
29 modesV = ['b', 'o', 'm']
30
31 print("svd_gpu_$dim_i: ")
32 CUDA.@time CUDA.@sync cuTensorNet.svd!(block_gpu, modesA, u_gpu, modesU, s_gpu, v_gpu, modesV)
```

- 基本线性代数: LinearAlgebra (stdlib)
- 运行时间测试: BenchmarkTools (精准测试) , TimerOutputs (耗时程序)
- 数据读写: DelimitedFiles (stdlib) , HDF5 & JLD2 (大规模数据读写)
- 运筹优化: Optim (单变量及多变量的无约束优化) , JuMP (代数建模语言+ 约束优化)
- 张量网络: ITensors (MPS) , TensorOperations , OMEinsum (高维张量网络收缩)