math3406_diary

October 31, 2021

1 MATH 3406 Diary

1.1 Introduction

In this diary I will be experimenting with the concepts learned in MATH 3406 and doing the assigned problems. I will be using the Julia programming language. The diary is maintained through a Jupyter notebook which lets me combine prose and code in a user-friendly way.

Julia is similar to MATLAB in some regards. It draws inspiration from dynamic languages like Lisp, Perl, Python, and R. Under the hood, Julia operates on the multiple-dispatch paradigm and supports optional typing.

I decided to use this language because I thought it would be interesting to try it out!

1.2 Basic operations

[2]:
```
using LinearAlgebra
using RowEchelon
using BenchmarkTools # For benchmarking, alternative to tic; tac; in MATLAB
```

[3]:
```
A = [1 2 3; 4 1 6; 7 8 1]
```

[3]:
```
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1
```

[4]:
```
tr(A)
```

[4]:
```
3
```

[5]:
```
det(A)
```

[5]:
```
104.0
```

[6]:
```
inv(A)
```

[6]:
```
3×3 Array{Float64,2}:
 -0.451923   0.211538   0.0865385
  0.365385  -0.192308   0.0576923
```

```
  0.240385    0.0576923  -0.0673077
```

[7]:
```
A * inv(A)
```

[7]:
```
3×3 Array{Float64,2}:
  1.0           0.0          -2.77556e-17
 -2.22045e-16   1.0           0.0
  3.88578e-16  -1.38778e-17    1.0
```

Note how this is not exactly I_3 but is very close to it

[8]:
```
rref(A)
```

[8]:
```
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

1.3 Solving a linear system

[9]:
```
A = [2 1 1; 4 -6 0; -2 7 2]
b = [5, -2, 9]
A\b
```

[9]:
```
3-element Array{Float64,1}:
 1.0
 1.0
 2.0
```

Lets try it out with a singular matrix

[10]:
```
A = [1 2 3; 4 5 6; 7 8 9]
b = [5, -2, 9]
A\b
```

```
SingularException(3)

Stacktrace:
 [1] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:19 [inlined]
 [2] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at /Users/julia/buildbot
 ↪worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu
 ↪jl:85
 [4] #lu#136 at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
 ↪julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
```

```
[6] \(::Array{Int64,2}, ::Array{Int64,1}) at /Users/julia/buildbot/worker/
↪package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/generic.j :
↪1116
[7] top-level scope at In[10]:3
[8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
↪1091
```

Note how Julia throws a `SingularException`. This is because it checks if the matrix is singular before attempting to perform the calculation. Under the hood, Julia uses an LU decomposition for non-triangular square matrices (which it fails to compute becase A is singular).

1.4 LU Decomposition

[11]:
```
# Tri-diagonal matrix
A =
[
    1 -1 0 0;
    -1 2 -1 0;
    0 -1 2 -1;
    0 0 -1 2;
]

F = lu(A)
```

[11]:
```
LU{Float64,Array{Float64,2}}
L factor:
4×4 Array{Float64,2}:
  1.0   0.0   0.0  0.0
 -1.0   1.0   0.0  0.0
  0.0  -1.0   1.0  0.0
  0.0   0.0  -1.0  1.0
U factor:
4×4 Array{Float64,2}:
 1.0  -1.0   0.0   0.0
 0.0   1.0  -1.0   0.0
 0.0   0.0   1.0  -1.0
 0.0   0.0   0.0   1.0
```

[12]:
```
A[F.p, :] == F.L * F.U
```

[12]:
```
true
```

The above decomposition did not require any row-exchanges.

[13]:
```
A = [0 1; 3 4]
F = lu(A)
display(F)
println("p = ", F.p)
```

```
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
U factor:
2×2 Array{Float64,2}:
 3.0  4.0
 0.0  1.0

p = [2, 1]
```

[14]:
```
A[F.p, :] == F.L * F.U
```

[14]:
```
true
```

Note how in the above case, we needed to permuate the original matrix A. The permutation specifically is a row-exchange between R_1 and R_2

[15]:
```
A = [0 0 1; 0 0 2; 0 3 0]
F = lu(A)
```

```
SingularException(1)

Stacktrace:
 [1] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:19 [inlined]
 [2] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at /Users/julia/buildbot
 ↪worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu
 ↪jl:85
 [4] #lu#136 at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
 ↪julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] top-level scope at In[15]:2
 [7] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

Here we tried to perform an LU decomposition on a singular matrix. Note how Julia raises a `SingularException`. This is because by default Julia checks if the matrix is singular before performing a decomposition.

However, we know that every matrix admits an LUP decomposition, where P is the permutation matrix. We can disable the checking by passing the `check=false` parameter

[16]:
```
F = lu(A, check=false)
display(F.L)
```

```
display(F.U)
display(F.P)
display(F)
```

```
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
3×3 Array{Float64,2}:
 0.0  0.0  1.0
 0.0  3.0  0.0
 0.0  0.0  2.0
```

```
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  0.0  1.0
 0.0  1.0  0.0
```

```
Failed factorization of type LU{Float64,Array{Float64,2}}
```

Here we see the L, U, and P factors. It even prints a message saying that the factorization failed (which is expected for this example)

1.4.1 Uniqueness of LU factorization

If the matrix A is square and invertible, then there exists a unique LU decomposition provided we restrict L to be uni-triangular (i.e., 1s on the diagonal).

However, if A is not invertible, we cannot make such a guarantee. Here is an example.

[17]:
```
A = [0 1; 0 2]
```

[17]:
```
2×2 Array{Int64,2}:
 0  1
 0  2
```

A is not singular. Here are two possible LU decompositions:

[18]:
```
L1, U1 = lu(A, check=false)
display(L1)
display(U1)
```

```
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

```
2×2 Array{Float64,2}:
 0.0  1.0
 0.0  2.0
```

[19]:
```
L2 = [1 0; 3 1]
U2 = [0 1; 0 -1]
display(L2)
display(U2)
```

```
2×2 Array{Int64,2}:
 1  0
 3  1
```

```
2×2 Array{Int64,2}:
 0   1
 0  -1
```

[20]:
```
L1 * U1 == A
```

[20]:
```
true
```

[21]:
```
L2 * U2 == A
```

[21]:
```
true
```

1.5 HW 1

1.5.1 Section 1.3

30 For this question, I modeled the system as `Ax = b` and used the left division operator `\` to solve it

[22]:
```
# part 1

A = [1 1 1; 1 2 2; 2 3 -4]
b = [6, 11, 3]
x = A\b
```

[22]:
```
3-element Array{Float64,1}:
 1.0
 3.0
 2.0
```

[23]:
```
# part 2

A = [1 1 1; 1 2 2; 2 3 -4]
b = [7, 10, 3]
x = A\b
```

[23]:
```
3-element Array{Float64,1}:
 4.0
 1.0
 2.0
```

32

[24]:
```
L,U,p = lu(rand(3,3))
```

[24]:
```
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
 1.0        0.0       0.0
 0.181466   1.0       0.0
 0.865863  -0.030839  1.0
U factor:
3×3 Array{Float64,2}:
 0.970819  0.896182   0.897219
 0.0       0.681361  -0.0631938
 0.0       0.0       -0.312108
```

[25]:
```
U[p][1,1]
```

[25]:
```
0.0
```

[26]:
```
res = [0, 0, 0]
n = 100000

for i = 1:n
    A = rand(3,3)
    L,U,p = lu(A)
    res += abs.(diag(U)[p])
end

res/n
```

[26]:
```
3-element Array{Float64,1}:
 0.5383421368268668
 0.5395923172615149
 0.5379058415340402
```

The average values for each pivot are about 0.5

1.5.2 Section 1.4

21

[27]:
```
A = [0.5 0.5; 0.5 0.5]

println("A^2 = ", A^2)
println("A^3 = ", A^3)
```

```
A^2 = [0.5 0.5; 0.5 0.5]
A^3 = [0.5 0.5; 0.5 0.5]
```

[28]:
```
B = [1 0; 0 -1]

println("B^2 = ", B^2)
println("B^3 = ", B^3)
```

```
B^2 = [1 0; 0 1]
B^3 = [1 0; 0 -1]
```

[29]:
```
C = A*B
println("C = ", C)
println("C^2 = ", C^2)
println("C^3 = ", C^3)
```

```
C = [0.5 -0.5; 0.5 -0.5]
C^2 = [0.0 0.0; 0.0 0.0]
C^3 = [0.0 0.0; 0.0 0.0]
```

$$A^k = A$$

$$B^k = \begin{bmatrix} 1 & 0 \\ 0 & (-1)^k \end{bmatrix}$$

$$C^k = \begin{cases} \begin{bmatrix} 0.5 & -0.5 \\ 0.5 & -0.5 \end{bmatrix} & \text{if k=1} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \text{otherwise} \end{cases}$$

$k \in \mathbb{N}$

59

[30]:
```
A = I(3)
```

[30]:
```
3×3 Diagonal{Bool,Array{Bool,1}}:
 1
    1
       1
```

[31]:
```
v = [3;4;5]
```

[31]:
```
3-element Array{Int64,1}:
 3
 4
 5
```

[32]:
```
v'
```

[32]:
```
1×3 Adjoint{Int64,Array{Int64,1}}:
 3  4  5
```

[33]:
```
A*v
```

[33]:
```
3-element Array{Int64,1}:
 3
 4
 5
```

[34]:
```
v' * v
```

[34]:
```
50
```

[35]:
```
v*A
```

```
DimensionMismatch("array could not be broadcast to match destination")

Stacktrace:
 [1] check_broadcast_shape at ./broadcast.jl:520 [inlined]
 [2] check_broadcast_shape at ./broadcast.jl:521 [inlined]
 [3] check_broadcast_axes at ./broadcast.jl:523 [inlined]
 [4] check_broadcast_axes at ./broadcast.jl:527 [inlined]
 [5] instantiate at ./broadcast.jl:269 [inlined]
 [6] materialize! at ./broadcast.jl:848 [inlined]
 [7] materialize! at ./broadcast.jl:845 [inlined]
 [8] rmul!(::Array{Int64,2}, ::Diagonal{Bool,Array{Bool,1}}) at /Users/julia/
 ↪buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.5/
 ↪LinearAlgebra/src/diagonal.jl:192
 [9] * at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/diagonal.jl:185 [inlined]
 [10] *(::Array{Int64,1}, ::Diagonal{Bool,Array{Bool,1}}) at /Users/julia/
 ↪buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.5/
 ↪LinearAlgebra/src/matmul.jl:63
 [11] top-level scope at In[35]:1
 [12] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

60

[36]:
```
A = ones(4,4)
```

[36]:
```
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

[37]:
```
v = ones(4,1)
```

[37]:
```
4×1 Array{Float64,2}:
 1.0
 1.0
```

```
 1.0
 1.0
```

[38]:
```
A * v
```

[38]:
```
4×1 Array{Float64,2}:
 4.0
 4.0
 4.0
 4.0
```

[39]:
```
B = I(4) + ones(4,4)
```

[39]:
```
4×4 Array{Float64,2}:
 2.0  1.0  1.0  1.0
 1.0  2.0  1.0  1.0
 1.0  1.0  2.0  1.0
 1.0  1.0  1.0  2.0
```

[40]:
```
w = zeros(4,1) + 2 * ones(4,1)
```

[40]:
```
4×1 Array{Float64,2}:
 2.0
 2.0
 2.0
 2.0
```

[41]:
```
B * w
```

[41]:
```
4×1 Array{Float64,2}:
 10.0
 10.0
 10.0
 10.0
```

1.5.3 Section 1.6

12

(a) True

In both cases, we are calculating the inverse of a matrix A using the Gauss-Jordan method.

Case 1: A is upper triangular

Here, A is already in the row-echelon form. We do not need to perform forward elimination. During the first round of backward elimination, we will do a row transformation that substracts a multiple of the last row from the penultimate row. In the last row, the only non-zero entry is the value on the diagonal on both the LHS (if this entry were zero, then A would not be invertible) and RHS (which is the identity matrix). This means that when performing the row transformation, the

entries below the diagonal still remain zero, since subtracting zero from a number gives back the number. WLOG, we can apply the same reasoning to conclude that the entries below the diagonal remain zero throughout the entire backward elimination process on both the LHS and RHS. This means that the RHS (which is A^{-1}) is upper-triangular.

$\therefore A^{-1}$ is upper-triangular.

Case 2: A is lower triangular

Here, we only need to perform forward elimination, since the entries above the diagonal are zero. WLOG, we can apply the same reasoning as in the previous case on backward elimination to conclude that the entries above the diagonal remain zero throughout the forward elimination process on both the LHS and RHS.

$\therefore A^{-1}$ is lower-triangular.

From both these cases, we conclude that A^{-1} is triangular.

(b) True

If A is symmetric then,

$A^T = A$

But,

$(A^T)^{-1} = (A^{-1})^T$

$\Rightarrow A^{-1} = (A^{-1})^T$

$\therefore A^{-1}$ is symmetric.

(c) False

Example:

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 1 \end{bmatrix}$$

(d) False

Example:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{bmatrix}$$

(e) True

The entries of A are given to be fractions. In other words, the entries of A are rational (since every rational number can be expressed in the form $\frac{p}{q}$ where q is non-zero). We know that the set of rational numbers are closed under addition, multiplication, and subtraction. The set of row-transformations (as explained below) we need to apply to find the inverse using the Gauss-Jordan process precisely use addition, multiplication, and subtraction.

Lets say we want to make the entry $\frac{p}{q}$ on row R_i zero by applying a row-transformation that subtracts this row from a row R_j (with $j < i$) whose corresponding column entry is $\frac{a}{b}$ (that is non-zero, it wouldn't make sense to subtract a zero entry). We can write this transformation as

$$R_i \to R_i - \frac{p}{q}\frac{b}{a}R_j$$

Observe that this transformation takes rational numbers to rational numbers. $\frac{b}{a}$ exists since $\frac{a}{b}$ is non-zero.

The corresponding operation that takes place on the RHS is also rational, since we start off with the identity matrix.

Row-exchanges do not change the rationality of the entries.

By performing a series of the above-mentioned transformations, we will be able to calculate the inverse of A using the Gauss-Jordan process.

[42]:
```
# Calculation for part (c)
A =
[
    1 1 0 0;
    1 1 1 0;
    0 1 1 1;
    0 0 1 1
]

inv(A)
```

[42]:
```
4×4 Array{Float64,2}:
  1.0   0.0  -1.0   1.0
  0.0   0.0   1.0  -1.0
 -1.0   1.0   0.0  -0.0
  1.0  -1.0   0.0   1.0
```

[43]:
```
# Calculation for part (d)
A =
[
    1 0;
    0 2;
]
inv(A)
```

[43]:
```
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  0.5
```

32

[44]:
```
A = 5 * I(4) - ones(4,4)
```

[44]:
```
4×4 Array{Float64,2}:
  4.0  -1.0  -1.0  -1.0
 -1.0   4.0  -1.0  -1.0
 -1.0  -1.0   4.0  -1.0
 -1.0  -1.0  -1.0   4.0
```

[45]:
```
inv(A)
```

[45]:
```
4×4 Array{Float64,2}:
 0.4  0.2  0.2  0.2
 0.2  0.4  0.2  0.2
 0.2  0.2  0.4  0.2
 0.2  0.2  0.2  0.4
```

$a = 0.4$ and $b = 0.2$

[46]:
```
A = 6 * I(5) - ones(5,5)
```

[46]:
```
5×5 Array{Float64,2}:
  5.0  -1.0  -1.0  -1.0  -1.0
 -1.0   5.0  -1.0  -1.0  -1.0
 -1.0  -1.0   5.0  -1.0  -1.0
 -1.0  -1.0  -1.0   5.0  -1.0
 -1.0  -1.0  -1.0  -1.0   5.0
```

[47]:
```
inv(A)
```

[47]:
```
5×5 Array{Float64,2}:
 0.333333  0.166667  0.166667  0.166667  0.166667
 0.166667  0.333333  0.166667  0.166667  0.166667
 0.166667  0.166667  0.333333  0.166667  0.166667
 0.166667  0.166667  0.166667  0.333333  0.166667
 0.166667  0.166667  0.166667  0.166667  0.333333
```

$a = 0.3333$ and $b = 0.1667$

47

[48]:
```
A = ones(4,4)
b = rand(4,1)
x = A\b
```

```
SingularException(2)

Stacktrace:
 [1] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:19 [inlined]
 [2] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at /Users/julia/buildbot
 ↪worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu
 ↪jl:85
 [4] #lu#136 at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
 ↪julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \(::Array{Float64,2}, ::Array{Float64,2}) at /Users/julia/buildbot/worker/
 ↪package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/generic.j :
 ↪1116
 [7] top-level scope at In[48]:3
 [8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

[49]:
```
b = ones(4,1)
x = A\b
```

```
SingularException(2)

Stacktrace:
 [1] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:19 [inlined]
 [2] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at /Users/julia/buildbot
 ↪worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu
 ↪jl:85
 [4] #lu#136 at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
 ↪julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \(::Array{Float64,2}, ::Array{Float64,2}) at /Users/julia/buildbot/worker/
 ↪package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/generic.j :
 ↪1116
 [7] top-level scope at In[49]:2
 [8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

Julia throws a `SingularException` in both cases. This is because the language does not attempt

to solve the equation if the matrix is singular.

Running the same example in Matlab, you get the following output

```
>> A = ones(4,4)
A =
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

>> b = rand(4,1)
b =
    0.8147
    0.9058
    0.1270
    0.9134

>> x = A\b
Warning: Matrix is singular to working
precision.

x =
   NaN
   NaN
   NaN
   Inf

>> b = ones(4,1)
b =
     1
     1
     1
     1

>> x = A\b
Warning: Matrix is singular to working
precision.

x =
   NaN
   NaN
   NaN
   NaN
```

Instead of throwing an exception, MATLAB issues a warning.

69

[50]:
```
A = rand(500, 500)

@btime inv(A);
```

```
  7.201 ms (5 allocations: 2.16 MiB)
```

[51]:
```
A = rand(1000, 1000)

@btime inv(A);
```

```
  37.743 ms (5 allocations: 8.13 MiB)
```

We are using the `@btime` macro from the `BenchmarkTools` package which lets us benchmark our functions easily.

For the above two cases, computing the inverse of a random 1000x1000 matrix took longer than a random 500x500 matrix.

In our case, our random `A` matrices were invertible. This may not always be the case since `rand` may output a singular matrix (which is not invertible). However, the chance of this happening is very low.

70

[52]:
```
_I = I(1000)
A = rand(1000, 1000)
B = triu(A);
```

[53]:
```
@btime inv(B);
```

```
  15.000 ms (2 allocations: 7.63 MiB)
```

[54]:
```
@btime B\I;
```

```
  15.023 ms (2 allocations: 7.63 MiB)
```

[55]:
```
@btime inv(A);
```

```
  38.384 ms (5 allocations: 8.13 MiB)
```

[56]:
```
@btime A\I;
```

```
  39.796 ms (5 allocations: 8.13 MiB)
```

The running times are almost the same.

\ performs slightly better than `inv`

71 In Julia, rational numbers are created using the `//` operator. We can display floating point numbers are fractions using the `rationalize` function. In the following code blocks, we broadcast the `rationalize` function to the matrix.

[57]:
```
L = [1 0 0 0; -0.5 1 0 0; 0 -2/3 1 0; 0 0 -3/4 1]
rationalize.(L)
```

[57]:
```
4×4 Array{Rational{Int64},2}:
  1//1   0//1   0//1  0//1
 -1//2   1//1   0//1  0//1
  0//1  -2//3   1//1  0//1
  0//1   0//1  -3//4  1//1
```

[58]:
```
rationalize.(inv(L))
```

[58]:
```
4×4 Array{Rational{Int64},2}:
 1//1  0//1  0//1  0//1
 1//2  1//1  0//1  0//1
 1//3  2//3  1//1  0//1
 1//4  1//2  3//4  1//1
```

Testing the pattern

[59]:
```
L = I(5) - diagm(1:5)\diagm(-1=>1:4)
rationalize.(L)
```

[59]:
```
5×5 Array{Rational{Int64},2}:
  1//1   0//1   0//1   0//1  0//1
 -1//2   1//1   0//1   0//1  0//1
  0//1  -2//3   1//1   0//1  0//1
  0//1   0//1  -3//4   1//1  0//1
  0//1   0//1   0//1  -4//5  1//1
```

[60]:
```
rationalize.(inv(L))
```

[60]:
```
5×5 Array{Rational{Int64},2}:
 1//1  0//1  0//1  0//1  0//1
 1//2  1//1  0//1  0//1  0//1
 1//3  2//3  1//1  0//1  0//1
 1//4  1//2  3//4  1//1  0//1
 1//5  2//5  3//5  4//5  1//1
```

The general formula for L is

`L = I(x) - diagm(1:x)\diagm(-1=>1:(x-1))`

Let's test it out for the next few matrices in the sequence

[61]:
```
function q71(x)
    I(x) - diagm(1:x)\diagm(-1=>1:(x-1))
end
```

[61]:
```
q71 (generic function with 1 method)
```

[62]:
```
L = q71(6)
rationalize.(L)
```

[62]:
```
6×6 Array{Rational{Int64},2}:
  1//1   0//1   0//1   0//1   0//1  0//1
 -1//2   1//1   0//1   0//1   0//1  0//1
  0//1  -2//3   1//1   0//1   0//1  0//1
  0//1   0//1  -3//4   1//1   0//1  0//1
  0//1   0//1   0//1  -4//5   1//1  0//1
  0//1   0//1   0//1   0//1  -5//6  1//1
```

[63]:
```
rationalize.(inv(L))
```

[63]:
```
6×6 Array{Rational{Int64},2}:
 1//1  0//1  0//1  0//1  0//1  0//1
 1//2  1//1  0//1  0//1  0//1  0//1
 1//3  2//3  1//1  0//1  0//1  0//1
 1//4  1//2  3//4  1//1  0//1  0//1
 1//5  2//5  3//5  4//5  1//1  0//1
 1//6  1//3  1//2  2//3  5//6  1//1
```

[64]:
```
L = q71(7)
rationalize.(L)
```

[64]:
```
7×7 Array{Rational{Int64},2}:
  1//1   0//1   0//1   0//1   0//1   0//1  0//1
 -1//2   1//1   0//1   0//1   0//1   0//1  0//1
  0//1  -2//3   1//1   0//1   0//1   0//1  0//1
  0//1   0//1  -3//4   1//1   0//1   0//1  0//1
  0//1   0//1   0//1  -4//5   1//1   0//1  0//1
  0//1   0//1   0//1   0//1  -5//6   1//1  0//1
  0//1   0//1   0//1   0//1   0//1  -6//7  1//1
```

[65]:
```
rationalize.(inv(L))
```

[65]:
```
7×7 Array{Rational{Int64},2}:
 1//1  0//1                 0//1                 0//1  0//1  0//1  0//1
 1//2  1//1                 0//1                 0//1  0//1  0//1  0//1
 1//3  2//3                 1//1                 0//1  0//1  0//1  0//1
 1//4  1//2                 3//4                 1//1  0//1  0//1  0//1
 1//5  2//5                 3//5                 4//5  1//1  0//1  0//1
 1//6  1//3                 1//2                 2//3  5//6  1//1  0//1
 1//7  2//7  428914250225764//1000799917193449  4//7  5//7  6//7  1//1
```

Interesting point to note: Julia fails to correctly output `3/7` in the inverse. Instead, it outputs a number that is very close to it.

[66]:
```
float(428914250225764//1000799917193449) - (3/7)
```

[66]:
```
1.6653345369377348e-16
```

1.6 Matrix Inverse

[67]:
```
# Non-singular matrix
A =
[
    2 4;
    4 12;
]
inv(A)
```

[67]:
```
2×2 Array{Float64,2}:
  1.5  -0.5
 -0.5   0.25
```

[68]:
```
# Singular matrix
A =
[
    1 2 3;
    4 5 6;
    7 8 9;
]
inv(A)
```

```
SingularException(3)

Stacktrace:
 [1] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:19 [inlined]
 [2] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at /Users/julia/buildbot
 ↪worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu
 ↪jl:85
 [4] #lu#136 at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
 ↪julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] inv(::Array{Int64,2}) at /Users/julia/buildbot/worker/package_macos64/buil
 ↪usr/share/julia/stdlib/v1.5/LinearAlgebra/src/dense.jl:781
 [7] top-level scope at In[68]:8
 [8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

$A = LU$ via the LU decomposition

$AB = I_n$

A is invertible iff U is invertible and $A^{-1} = U^{-1}L^{-1}$

1.7 Transpose

[69]:
```
A =
[
    1 2 3;
    4 5 6;
    7 8 9;
]

A'
```

[69]:
```
3×3 Adjoint{Int64,Array{Int64,2}}:
 1  4  7
 2  5  8
 3  6  9
```

Theorem: $(A^{-1})^T = (A^T)^{-1}$

[70]:
```
A =
[
    2 4;
    0 1;
]
inv(A)
```

[70]:
```
2×2 Array{Float64,2}:
 0.5  -2.0
 0.0   1.0
```

[71]:
```
A'
```

[71]:
```
2×2 Adjoint{Int64,Array{Int64,2}}:
 2  0
 4  1
```

[72]:
```
inv(A)' == inv(A')
```

[72]:
```
true
```

1.8 Symmetric Matrices

[73]:
```
A =
[
    1 2 3;
    2 1 2;
    3 2 1;
]
```

```
A' == A
```

[73]: true

[74]:
```
issymmetric(A)
```

[74]: true

If A is symmetric and invertible, and $A = LDU$ using the Cholesky decomposition, then $A = LDL^T$

[75]:
```
A =
[
    2 1 1;
    1 2 0;
    1 0 2;
]

cholesky(A)
```

[75]:
```
Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 1.41421   0.707107   0.707107
          1.22474   -0.408248
                     1.1547
```

[76]:
```
A =
[
    1 2 3;
    2 1 2;
    3 2 1;
]

cholesky(A)
```

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

Stacktrace:
 [1] checkpositivedefinite at /Users/julia/buildbot/worker/package_macos64/buil
 ↪usr/share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:18 [inlined]
 [2] cholesky!(::Hermitian{Float64,Array{Float64,2}}, ::Val{false}; check::Bool
 ↪at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/
 ↪v1.5/LinearAlgebra/src/cholesky.jl:219
 [3] cholesky!(::Array{Float64,2}, ::Val{false}; check::Bool) at /Users/julia/
 ↪buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.5/
 ↪LinearAlgebra/src/cholesky.jl:251
 [4] #cholesky#130 at /Users/julia/buildbot/worker/package_macos64/build/usr/
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/cholesky.jl:344 [inlined]
```

```
 [5] cholesky at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
↪julia/stdlib/v1.5/LinearAlgebra/src/cholesky.jl:344 [inlined] (repeats 2 time
[6] top-level scope at In[76]:8
[7] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
↪1091
```

Interesting. A is symmetrix and invertible. However, Julia fails to compute the cholesky decomposition. The error message tells us that the matrix is not positive definite.

1.9 Vector Spaces

A vector space has two operations defined: - $(+)$ addition - $(\cdot)$ scalar multiplication

Examples: - $\mathbb{R}^\infty$: Addition and scalar multiplication are the same as $\mathbb{R}^n$ - Space of functions f: $(f+g)(x) = f(x) + g(x)$, $f(\alpha x) = \alpha f(x)$

1.9.1 Subspaces

A subspace of a vector space is a nonempty subset that satisfies the requirements of a vector space: Linear combinations of its members stay in the subspace.

Important subspaces: - Column space: Range of A. All possible linear combinations of the columns of A. $span(A)$ - Null space: Set of solutions to $Ax = 0$

[77]:
```
# Finding the null space

A =
[
    1 0 0;
    0 1 0;
    0 0 0;
]

nullspace(A)
```

[77]:
```
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0
```

This makes sense. The vectors x along the third dimension make $Ax = 0$ since it is identity for the other two dimensions.

[78]:
```
A =
[
    1 0 1;
    5 4 9;
    2 4 6;
]
```

```
nullspace(A)
```

[78]:
```
3×1 Array{Float64,2}:
  0.5773502691896256
  0.577350269189626
 -0.5773502691896257
```

[79]:
```
A =
[
    1 0 0;
    0 0 0;
    0 0 0;
]
nullspace(A)
```

[79]:
```
3×2 Array{Float64,2}:
 0.0  0.0
 0.0  1.0
 1.0  0.0
```

[80]:
```
A =
[
    1 0 0;
    0 0 0;
    0 0 0;
]
rank(A)
```

[80]:
```
1
```

[81]:
```
A =
[
    1 0 0;
    0 1 0;
    0 0 0;
]
rank(A)
```

[81]:
```
2
```

[82]:
```
A = I(3)
rank(A)
```

[82]:
```
3
```

Section 1.7, q8

[83]:
```
H =
[
    1 1//2 1//3;
    1//2 1//3 1//4;
    1//3 1//4 1//5
]
inv(H)
```

[83]:
```
3×3 Array{Rational{Int64},2}:
   9//1   -36//1    30//1
 -36//1   192//1  -180//1
  30//1  -180//1   180//1
```

[84]:
```
H =
[
    1 0.5 0.333;
    0.5 0.333 0.25;
    0.333 0.25 0.2;
]
inv(H)
```

[84]:
```
3×3 Array{Float64,2}:
   9.67066   -39.5082    33.2836
 -39.5082    210.186   -196.951
  33.2836   -196.951    195.772
```

[85]:
```
H =
[
    1 1//2 1//3;
    1//2 1//3 1//4;
    1//3 1//4 1//5
]
float(H * [1;1;1])
```

[85]:
```
3-element Array{Float64,1}:
 1.8333333333333333
 1.0833333333333333
 0.7833333333333333
```

[86]:
```
H * [0; 6; -3.6]
```

[86]:
```
3-element Array{Float64,1}:
 1.8
 1.1
 0.7799999999999999
```

[87]:
```
rref(H)
```

[87]:
```
3×3 Array{Rational{Int64},2}:
 1//1  0//1  0//1
 0//1  1//1  0//1
 0//1  0//1  1//1
```

Note that the value of b are almost the same. But $\begin{bmatrix}1\\1\\1\end{bmatrix} \neq \begin{bmatrix}0\\6\\-3.6\end{bmatrix}$.

But H is Hermitian. As we can see above, the RREF of H is I_3, which means that for every b, there must be a unique solution. But due to rounding errors, we get approximately the same value of b.

[88]:
```
A =
[
    2 1 0;
    1 2 1;
    0 1 2;
]
inv(A)
```

[88]:
```
3×3 Array{Float64,2}:
  0.75  -0.5   0.25
 -0.5    1.0  -0.5
  0.25  -0.5   0.75
```

1.10 Unitary Matrices

A matrix U is said to be unitary if $UU^\dagger = U^\dagger U = I$. The † stands for conjugate transpose.

Unitary matrices are extensively used in describing the theory of quantum mechanics (and by relation quantum computing).

In quantum computing, operations are carried out on a state vector ψ. These operations can be represented as a unitary matrix. Quantum gates have to be reversible. In other words, the matrix representing the operation must have an inverse. Unitary matrices satisfy this condition.

The unitary matrix has several interesting properties: - $|det(U)| = 1$ - $(Ux) \cdot (Uy) = x \cdot y$, i.e., it preserves the inner product - U can be written in the form e^{iH}, where H is a Hermitian matrix

[10]:
```
# The identity matrix is unitary!

U = [
    1 0;
    0 1;
]

U == U'
```

[10]:
```
true
```

[11]:
```
function is_unitary(U)
    U == U'
end
```

[11]:
```
is_unitary (generic function with 1 method)
```

Note that the ' operator in Julia computes a complex conjugate transpose.

[15]:
```
# The set of Pauli gates X, Y, and Z are unitary

X = [
    0 1;
    1 0;
]

Y = [
    0 -im;
    im 0;
]

Z = [
    1 0;
    0 -1;
]

println(is_unitary(X))
println(is_unitary(Y))
println(is_unitary(Z))
```

```
true
true
true
```

[107]:
```
# The Hadamard matrix is also unitary

H = [
    1/√2 1/√2;
    1/√2 -1/√2;
]
is_unitary(H)
```

[107]:
```
true
```

[24]:
```
x = [2; 3]
y = [4; 7]

U = [
    1 0;
```

```
    0 -1
]

# Unitary matrices preserve the inner product
dot(x,y) == dot((U*x), (U*y))
```

[24]:
```
true
```

1.10.1 Matrix Exponential

Lets say X is a matrix. What does e^X mean? This operation is known as the matrix exponential and is a generalization of the Taylor series expansion of e^x but where each x is a matrix.

$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k$

Recall that for $c \in \mathbb{C}$, we defined the exponential function to be

$e^c = \sum_{k=0}^{\infty} \frac{1}{k!} c^k$

[44]:
```
methods(exp)
```

[44]:
```
# 14 methods for generic function "exp":
[1] exp(a::Float16) in Base.Math at math.jl:1144
[2] exp(x::BigFloat) in Base.MPFR at mpfr.jl:603
[3] exp(::Missing) in Base.Math at math.jl:1197
[4] exp(a::Complex{Float16}) in Base.Math at math.jl:1145
[5] exp(z::Complex) in Base at complex.jl:613
[6] exp(x::T) where T<:Union{Float32, Float64} in Base.Math at special/exp.jl:74
[7] exp(x::AbstractFloat) in Base.Math at math.jl:399
[8] exp(x::Real) in Base.Math at special/exp.jl:73
[9] exp(A::StridedArray{var"#s828", 2} where var"#s828"<:Union{Complex{Float32},
Complex{Float64}, Float32, Float64}) in LinearAlgebra at /Applications/Julia-1.5
.app/Contents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/dense.jl
:523
[10] exp(A::StridedArray{var"#s828", 2} where var"#s828"<:Union{Integer,
Complex{var"#s827"} where var"#s827"<:Integer}) in LinearAlgebra at /Application
s/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/s
rc/dense.jl:524
[11] exp(A::Hermitian{var"#s827",S} where S<:(AbstractArray{var"#s828",2} where
var"#s828"<:var"#s827") where var"#s827"<:Complex) in LinearAlgebra at /Applicat
ions/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebr
a/src/symmetric.jl:922
[12] exp(A::Union{Hermitian{var"#s828",S}, Symmetric{var"#s828",S}} where S
where var"#s828"<:Real) in LinearAlgebra at /Applications/Julia-1.5.app/Contents
/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/symmetric.jl:918
[13] exp(D::Diagonal) in LinearAlgebra at /Applications/Julia-1.5.app/Contents/R
esources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/diagonal.jl:576
[14] exp(J::UniformScaling) in LinearAlgebra at /Applications/Julia-1.5.app/Cont
ents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/uniformscaling.jl
```

```
:139
```

Observe how Julia has different implementations of `exp` depending on the type of the matrix. Computing the matrix exponential of a diagonalizable matrix is a lot simpler than the general case. Julia uses different types like `Hermitian`, `StridedArray`, `Diagonal` to disambiguate the function call. This is a prime example of how Julia uses multiple dispatch to optimize certain cases.

[46]:
```
A = Hermitian([0 -im; im 0])
exp(A)
```

[46]:
```
2×2 Hermitian{Complex{Float64},Array{Complex{Float64},2}}:
 1.54308+0.0im         0.0-1.1752im
     0.0+1.1752im  1.54308+0.0im
```

[49]:
```
A = [1 2 3; 4 5 6; 7 8 9]
A = Diagonal(A)
```

[49]:
```
3×3 Diagonal{Float64,Array{Float64,1}}:
 2.71828
          148.413
                   8103.08
```

1.10.2 Hermitian Matrices

A matrix H is said to be Hermitian if it is equal to its conjugate transpose.

$H = H^\dagger$, where † stands for conjugate tranpose.

Note that here the entries of H may be complex. A corollary of this is a matrix that has only real entries is symmetric if and only if it is Hermitian.

A Hermitian matrix may not be symmetric.

[28]:
```
H = [
    0 -im;
    im 0;
]

# H is not symmetric but it is hermitian
print(issymmetric(H))
H == H'
```

```
false
```

[28]:
```
true
```

Now that we know what the matrix exponential and Hermitian matrices are, let's try to find H such that $U = e^{iH}$, where U is a unitary matrix.

Since U is unitary, we can diagonalize it.

$U = VDV^\dagger$

This is the spectral decomposition. By the definition of unitary matrices, the eigenvalues (along the diagonal matrix) must have modulus 1. Therefore, we can represent the diagonal as $\lambda_j = e^{i\theta_j}$

$\therefore H = \frac{1}{i} ln(U) = V diag(\theta_1, \theta_2, \ldots) V^\dagger$.

Note that there can be many such H.

[33]:
```
U = [
    0 -im;
    im 0;
]

H = log(U)/im
```

[33]:
```
2×2 Array{Complex{Float64},2}:
 1.5708-0.0im        0.0+1.5708im
    0.0-1.5708im  1.5708-0.0im
```

[39]:
```
exp(im*H)
```

[39]:
```
2×2 Array{Complex{Float64},2}:
 -1.19059e-17+4.81586e-16im   -4.996e-16-1.0im
  4.44089e-16+1.0im          3.98213e-17+3.91023e-16im
```

We are very close to equality. Let's round the result and compare

[40]:
```
U == round.(exp(im*H))
```

[40]:
```
true
```

1.11 Four Fundamental Subspaces

As we learned in class, every $m \times n$ matrix A has four fundamental subspaces associated with it: - $Col(A)$ - also known as the range - $Nul(A)$ - also known as the kernel - $Col(A^T)$ - also known as the row space - $Nul(A^T)$ - also known as the left null space

Let's try to explore these different subspaces by trying out some operations on matrices!

[94]:
```
A = [
    1 2;
    3 6;
]
```

[94]:
```
2×2 Array{Int64,2}:
 1  2
 3  6
```

[95]:
```
# To find out the column space, we need the RREF form
rref(A)
```

[95]:
```
2×2 Array{Float64,2}:
 1.0  2.0
 0.0  0.0
```

Clearly, the above matrix has only one pivot.

$\therefore Col(A) = span\{\begin{pmatrix}1\\2\end{pmatrix}\}$

[96]:
```
# This function computes a basis for the nullspace of A
nullspace(A)
```

[96]:
```
2×1 Array{Float64,2}:
 -0.894427190999916
  0.447213595499958
```

Interesting... this is different than what I got when computing the null space by hand.

I was expecting to get $\begin{pmatrix}-2\\1\end{pmatrix}$

Let's see if the vector we got satisfies $Ax = 0$

[97]:
```
A * nullspace(A)
```

[97]:
```
2×1 Array{Float64,2}:
 0.0
 0.0
```

It does!

Note that the Rank theorem is satisifed!

$rank = dim(Col(A)) = 1$

$nullity = dim(Nul(A)) = 1$

$n = rank + nullity = 2$

[98]:
```
rref(A')
```

[98]:
```
2×2 Array{Float64,2}:
 1.0  3.0
 0.0  0.0
```

The RREF of A^T also has only one pivot (we were expecting this anyways since the dimension of the column space and row space are the same).

$\therefore Col(A^T) = span\{\begin{pmatrix}1\\2\end{pmatrix}\}$

[99]:
```
nullspace(A')
```

[99]:
```
2×1 Array{Float64,2}:
 -0.9486832980505138
  0.316227766016838
```

[100]:
```
A' * nullspace(A')
```

[100]:
```
2×1 Array{Float64,2}:
 2.220446049250313e-16
 4.440892098500626e-16
```

The result is not quite zero, but close to it. If we round it we'll get zero.

[106]:
```
round.(A' * nullspace(A'))
```

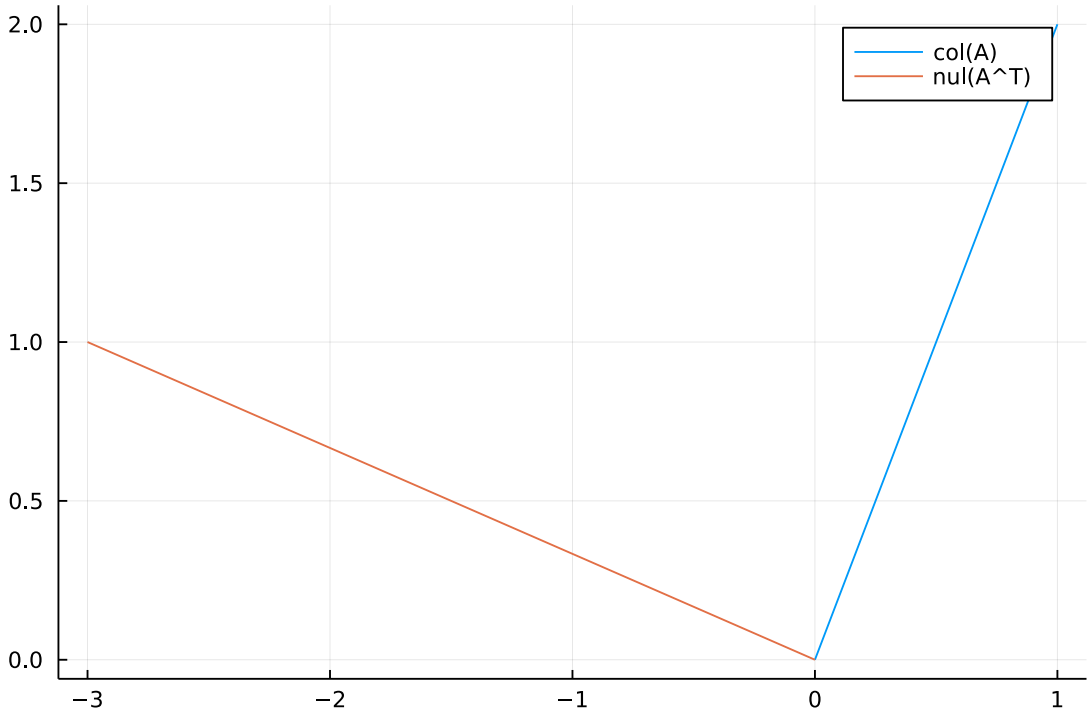
[106]:
```
2×1 Array{Float64,2}:
 0.0
 0.0
```

Let's try to plot these spaces

[65]:
```
using Plots
```

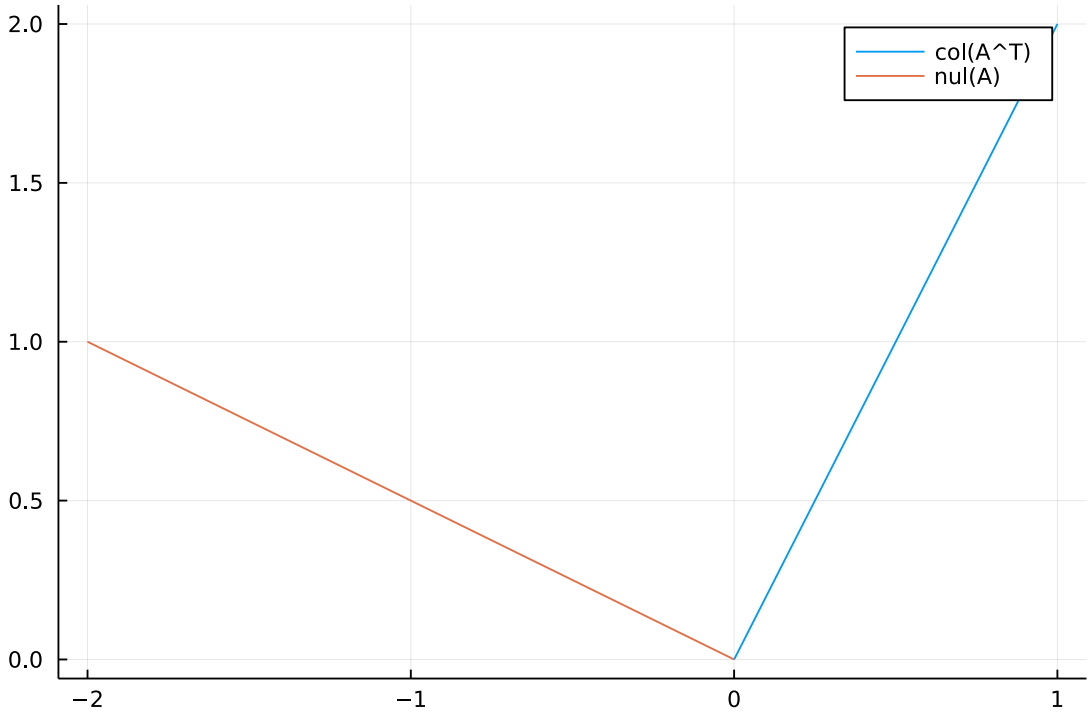
[73]:
```
x = [[0, 1], [0, -3]]
y = [[0, 2], [0, 1]]
plot(x, y, label=["col(A)" "nul(A^T)"])
```

[73]:



[74]:
```
x = [[0, 1], [0, -2]]
y = [[0, 2], [0, 1]]
plot(x, y, label=["col(A^T)" "nul(A)"])
```

[74]:



We can see that the spaces are orthogonal to each other

1.11.1 Left inverse, right inverse, full inverse

If A is an $m \times n$ matrix, then the following inverses are defined: - Right inverse: full row rank (i.e., $rank = m$) - Left inverse: full column rank (i.e., $rank = n$) - Full inverse: $m = n = rank$

A right inverse is only possible if $m \leq n$

A left inverse is only possible if $m \geq n$

Julia does not have a built-in function to compute the left and right inverse (only the full inverse is supported). We can, however, easily define those functions.

A right inverse exists if AA^T is invertible. A left inverse exists if A^TA is invertible.

[76]:
```
function inv_right(A)
    A'*inv(A*A')
end
```

[76]:
```
inv_right (generic function with 1 method)
```

[77]:
```
function inv_left(A)
    inv(A'*A)*A'
end
```

[77]:
```
inv_left (generic function with 1 method)
```

[82]:
```
A = [
    4 0 0;
    0 5 0;
]
inv_right(A)
```

[82]:
```
3×2 Array{Float64,2}:
 0.25  0.0
 0.0   0.2
 0.0   0.0
```

[83]:
```
A * inv_right(A)
```

[83]:
```
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

[84]:
```
A = [
    4 0;
    0 5;
    0 0;
]
inv_left(A)
```

[84]:
```
2×3 Array{Float64,2}:
 0.25  0.0  0.0
 0.0   0.2  0.0
```

[85]:
```
inv_left(A) * A
```

[85]:
```
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

[88]:
```
A = [
    4 0;
    0 5;
]
inv(A)
```

[88]:
```
2×2 Array{Float64,2}:
 0.25  0.0
```

```
 0.0   0.2
```

The existence of a right inverse implies that there exists **at least one solution** to $Ax = b$ for all $b \in \mathbb{R}^n$

The existence of a left inverse implies that there exists **at most one solution** to $Ax = b$ for all $b \in \mathbb{R}^n$

1.12 Tensor Product

Quantum circuits are composed by computing the tensor products between different gates. For example, $H \otimes X$ can be viewed as a circuit that applies a Hadamard gate to the first qubit and an X gate to the second qubit. I never really understood the tensor product before taking this class. I knew how to mechanically compute the Kronecker product, but didn't attempt to understand it beyond it.

Here is my attempt at exploring the concept in Julia!

[91]:
```
A = [
    0 1;
    1 0;
]
B = [
    1 0;
    0 -1;
]
kron(A, B)
```

[91]:
```
4×4 Array{Int64,2}:
 0   0  1   0
 0   0  0  -1
 1   0  0   0
 0  -1  0   0
```

The `kron` function computes the Kronecker product. The Kronecker product $A \otimes B$ is defined conveniently by the block matrix $\begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \\ a_{m1}B & & a_{mm}B \end{bmatrix}$.

Observation: The dimension of the matrices *multiply* together (and not add like in the case of the Cartesian product).

In the above example, $dim(A) = 2$, $dim(B) = 2$.

$dim(A \otimes B) = 2 * 2 = 4$

The tensor product offers a mathematical explanation for why quantum computing offers an exponentially larger computational space for the same number of computing units.

1.13 HW 2

1.13.1 Section 2.2

33

[1]:
```
A = [
    1 3 3;
    2 6 9;
    -1 -3 3;
]
b = [
    1;
    5;
    5;
]

x = A \ b
```

```
LinearAlgebra.SingularException(2)

Stacktrace:
 [1] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:19 [inlined]
 [2] checknonsingular at /Users/julia/buildbot/worker/package_macos64/build/usr
 ↪share/julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:21 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at /Users/julia/buildbot
 ↪worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu
 ↪jl:85
 [4] #lu#136 at /Users/julia/buildbot/worker/package_macos64/build/usr/share/
 ↪julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at /Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/
 ↪stdlib/v1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \(::Array{Int64,2}, ::Array{Int64,1}) at /Users/julia/buildbot/worker/
 ↪package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/generic.j
 ↪1116
 [7] top-level scope at In[1]:12
 [8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

The columns of A are not linearly independent. Therefore we run into `SingularException`.

[18]:
```
F = lu(A, check=false)
F.P*F.L*F.U == A
```

[18]:
```
true
```

[16]:
```
F.L
```

[16]:
```
3×3 Array{Float64,2}:
  1.0  0.0  0.0
  0.5  1.0  0.0
 -0.5  0.0  1.0
```

[17]:
```
F.U
```

[17]:
```
3×3 Array{Float64,2}:
 2.0  6.0   9.0
 0.0  0.0  -1.5
 0.0  0.0   7.5
```

Using this, we can easily read off the solutions.

$x_1 = -3x_2 - 4.5x_3$

$x_2 = x_2$

$x_3 = 0$

$Nul(A) = \{ \begin{pmatrix} -3 \\ 1 \\ 0 \end{pmatrix} \}$

[23]:
```
inv(F.P*F.L)*b
```

[23]:
```
3-element Array{Float64,1}:
  5.0
 -1.5
  7.5
```

Particular solution:

$x_p = \begin{pmatrix} -2 \\ 0 \\ 1 \end{pmatrix}$

Therefore, the set of all solutions is given by:

$x = x_n + x_p$, where $x_n \in Nul(A)$

35 TODO

36

[84]:
```
# (a)

A = [
    1 2 1;
    2 6 3;
    0 2 5
]
```

```
rref(A)
```

[84]:
```
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

[85]:
```
lu(A)
```

[85]:
```
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
 1.0   0.0  0.0
 0.0   1.0  0.0
 0.5  -0.5  1.0
U factor:
3×3 Array{Float64,2}:
 2.0  6.0  3.0
 0.0  2.0  5.0
 0.0  0.0  2.0
```

Therefore, $Col(A) = \mathbb{R}^{\not\vDash}$ and $Nul(A) = \{0\}$

[86]:
```
# (b)

A = [
    1 1 1;
    1 2 4;
    2 4 8;
]

rref(A)
```

[86]:
```
3×3 Array{Float64,2}:
 1.0  0.0  -2.0
 0.0  1.0   3.0
 0.0  0.0   0.0
```

Therefore, $Col(A) = span\{\begin{pmatrix}1\\1\\2\end{pmatrix}, \begin{pmatrix}1\\2\\4\end{pmatrix}\}$ and $Nul(A) = span\{\begin{pmatrix}2\\-3\\1\end{pmatrix}\}$

1.14 Linear Programming

Linear programming (LP) is an optimization technique used to find the best outcome given a list of linear constraints. Many real-world problems (scheduling, supply-chain logistics, network flow) can be solved using LP.

Here's an example I found on the internet (https://www.upgrad.com/blog/linear-programming-

problems-solutions/):

Suppose you are a manufacturer of toys and you only produce two toys: A and B. Roughly speaking, your toys require two resources X and Y to manufacture. Here are the requirements of your toys:

- One unit of toy A requires you one unit of resource X and three units of resource Y
- One unit of toy B requires one unit of resource X and two units of resource Y
- You have five units of resource X and 12 units of resource Y.

Your profit margins on the sale of these toys are:

- \$6 on each sold unit of toy A
- \$5 on each sold unit of toy B

How many units of each toy would you produce to get the maximum profit?

We can represent the objective as:

$$\text{maximize } 6a + 5b$$

The constraints are:

$$\begin{aligned} a + b &\leq 5 \\ 3a + 2b &\leq 12 \\ a &\geq 0 \\ b &\geq 0 \end{aligned}$$

This is an LP problem! Note how each constraint is linear.

In general, we can convert any LP problem into the following canonical form:

$$\text{maximize } c^T x$$

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \end{aligned}$$

where A is an $m \times n$ matrix, $x \in \mathbb{R}^n$. There are m constraints (excluding the non-negativity constraint)

Julia has the JuMP package which lets us model LP problems and solve them. It also has support for other kinds of mathematical optimizations. Let's try to solve the previous problem using it.

[1]:
```
using JuMP
using GLPK
```

```
Info: Precompiling JuMP [4076af6c-e467-56ae-b986-b466b2749572]
@ Base loading.jl:1278
Info: Precompiling GLPK [60bf3e95-4087-53dc-ae20-288a0d20c6a6]
@ Base loading.jl:1278
```

[3]:
```
model = Model(GLPK.Optimizer)
@variable(model, a >= 0)
@variable(model, b >= 0)
@objective(model, Max, 6a + 5b)
@constraint(model, c1, a + b <= 5)
@constraint(model, c2, 3a + 2b <= 12)

print(model)
optimize!(model)

@show value(a)
@show value(b)
@show objective_value(model)
```

$$\begin{aligned} \max \quad & 6a + 5b \\ \text{Subject to} \quad & a + b \leq 5.0 \\ & 3a + 2b \leq 12.0 \\ & a \geq 0.0 \\ & b \geq 0.0 \end{aligned}$$

```
value(a) = 2.0000000000000004
value(b) = 2.9999999999999996
objective_value(model) = 27.0
```

[3]:
```
27.0
```

Julia lets programmers define macros which can help in removing a lot of boilterplate code. Note how JuMP provides us with macros like `@variable`, `@objective` and `@constraint`. The description of the LP problem very closely resembles how we would write it out in LaTeX

The optimal values are $a = 2$ and $b = 3$ (we can safely round up the values since we know that a and b have to be integers.

Let's solve another problem!

TODO insert problem

[5]:
```
# TODO problem solution
```

Now let's take a look at a max-flow problem.

TODO insert theory of max-flow

TODO insert problem

[4]:
```
# TODO problem solution
```

1.14.1 Simplex Algorithm

Simplex is a very popular algorithm for solving LP problems. We will implement this algorithm from scratch and run it on some inputs.

Earlier we saw how any LP can be put into a canonical form. The constratin $Ax \leq b$ represents a system of linear constraints. Each constraint is a region that lives in $\mathbb{R}^n$. These regions can intersect with each other.

Claim: The optimal value must be one of the points of intersection.

Simplex uses this fact to solve the LP. The algorithm starts at one of the points of intersection (which can be 0 in the first iteration). It then tries to find the neighboring points of intersection such that those points are more optimal. If it can't find any such points, then the algorithm halts.

[6]:
```
# TODO understand what Canonical Tableau form is
# TODO understand how to convert to this form
# TODO figure out iteration step
```

1.15 Least-Squares

1.16 QR Decomposition

1.17 Discrete Fourier Transform

1.18 HW 3

[ ]:
```

```

1.19 Determinants

1.20 Quantum Gates