MATH 3406 Diary

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!

Basic operations

In [257…]:
```julia
using LinearAlgebra
using RowEchelon
using BenchmarkTools # For benchmarking, alternative to tic; tac; in MATLAB
```

In [190…]:
```julia
A = [1 2 3; 4 1 6; 7 8 1]
```

Out[190…]:
```
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1
```

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

Out[4]:
```
3
```

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

Out[5]:
```
104.0
```

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

Out[6]:
```
3×3 Array{Float64,2}:
 -0.451923   0.211538    0.0865385
  0.365385  -0.192308    0.0576923
  0.240385   0.0576923  -0.0673077
```

In [191…]:
```julia
A * inv(A)
```

Out[191…]:
```
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

In [184…]:
```julia
rref(A)
```

Out[184…]:
```
2×2 Array{Float64,2}:
 0.0  1.0
 0.0  0.0
```

Solving a linear system

In [187…]:
```julia
A = [2 1 1; 4 -6 0; -2 7 2]
b = [5, -2, 9]
A\b
```

Out[187…]:
```
3-element Array{Float64,1}:
 1.0
 1.0
 2.0
```

Lets try it out with a singular matrix

In [188…]:
```julia
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.jl:1116
 [7] top-level scope at In[188]: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).

LU Decomposition

In [192…]:
```julia
# Tri-diagonal matrix
A =
[
    1 -1 0 0;
    -1 2 -1 0;
    0 -1 2 -1;
    0 0 -1 2;
]

F = lu(A)
```

Out[192…]:
```
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
```

In [149…]:
```julia
A[F.p, :] == F.L * F.U
```

Out[149…]:
```
true
```

The above decomposition did not require any row-exchanges.

In [146…]:
```julia
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]
```

In [148…]:
```julia
A[F.p, :] == F.L * F.U
```

Out[148…]:
```
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

In [151…]:
```julia
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[151]: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

In [156…]:
```julia
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)

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.

In [167…]:
```julia
A = [0 1; 0 2]
```

Out[167…]:
```
2×2 Array{Int64,2}:
 0  1
 0  2
```

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

In [170…]:
```julia
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
```

In [179…]:
```julia
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
```

In [181…]:
```julia
L1 * U1 == A
```

Out[181…]:
```
true
```

In [182…]:
```julia
L2 * U2 == A
```

Out[182…]:
```
true
```

HW 1

Section 1.3

30

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

In [11]:
```julia
# part 1

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

Out[11]:
```
3-element Array{Float64,1}:
 1.0
 3.0
 2.0
```

In [12]:
```julia
# part 2

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

Out[12]:
```
3-element Array{Float64,1}:
 4.0
 1.0
 2.0
```

32

In [204…]:
```julia
L,U,p = lu(rand(3,3))
```

Out[204…]:
```
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
 1.0        0.0       0.0
 0.0138241  1.0       0.0
 0.614814   0.778507  1.0
U factor:
3×3 Array{Float64,2}:
 0.779435  0.5035     0.403196
 0.0       0.246302   0.88074
 0.0       0.0       -0.601532
```

In [205…]:
```julia
U[p][1,1]
```

Out[205…]:
```
0.7794346449731999
```

In [134…]:
```julia
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
```

Out[134…]:
```
3-element Array{Float64,1}:
 0.538352456101028
 0.5404040696892417
 0.5395553022227193
```

The average values for each pivot are about 0.5

Section 1.4

21

In [198…]:
```julia
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]
```

In [199…]:
```julia
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]
```

In [200…]:
```julia
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

In [58]:
```julia
A = I(3)
```

Out[58]:
```
3×3 Diagonal{Bool,Array{Bool,1}}:
 1  ⋅  ⋅
 ⋅  1  ⋅
 ⋅  ⋅  1
```

In [63]:
```julia
v = [3;4;5]
```

Out[63]:
```
3-element Array{Int64,1}:
 3
 4
 5
```

In [65]:
```julia
v'
```

Out[65]:
```
1×3 Adjoint{Int64,Array{Int64,1}}:
 3  4  5
```

In [66]:
```julia
A*v
```

Out[66]:
```
3-element Array{Int64,1}:
 3
 4
 5
```

In [67]:
```julia
v' * v
```

Out[67]:
```
50
```

In [68]:
```julia
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[68]:1
 [12] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091
```

60

In [69]:
```julia
A = ones(4,4)
```

Out[69]:
```
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
```

In [70]:
```julia
v = ones(4,1)
```

Out[70]:
```
4×1 Array{Float64,2}:
 1.0
 1.0
 1.0
 1.0
```

In [71]:
```julia
A * v
```

Out[71]:
```
4×1 Array{Float64,2}:
 4.0
 4.0
 4.0
 4.0
```

In [73]:
```julia
B = I(4) + ones(4,4)
```

Out[73]:
```
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
```

In [74]:
```julia
w = zeros(4,1) + 2 * ones(4,1)
```

Out[74]:
```
4×1 Array{Float64,2}:
 2.0
 2.0
 2.0
 2.0
```

In [75]:
```julia
B * w
```

Out[75]:
```
4×1 Array{Float64,2}:
 10.0
 10.0
 10.0
 10.0
```

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.

In [208…]:
```julia
# Calculation for part (c)
A =
[
    1 1 0 0;
    1 1 1 0;
    0 1 1 1;
    0 0 1 1
]

inv(A)
```

Out[208…]:
```
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
```

In [214…]:
```julia
# Calculation for part (d)
A =
[
    1 0;
    0 2;
]
inv(A)
```

Out[214…]:
```
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  0.5
```

32

In [76]:
```julia
A = 5 * I(4) - ones(4,4)
```

Out[76]:
```
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
```

In [78]:
```julia
inv(A)
```

Out[78]:
```
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 $b = 0.2$

In [80]:
```julia
A = 6 * I(5) - ones(5,5)
```

Out[80]:
```
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
```

In [82]:
```julia
inv(A)
```

Out[82]:
```
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

In [84]:
```julia
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.jl:1116
 [7] top-level scope at In[84]:3
 [8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091
```

In [91]:
```julia
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.jl:1116
 [7] top-level scope at In[91]: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

In [265…]:
```julia
A = rand(500, 500)

@btime inv(A)
```

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

Out[265…]:
```
500×500 Array{Float64,2}:
 -0.232059   -0.305865    0.204706    …  -0.41236     0.848234   -0.158316
 -0.274561   -0.377009    0.77494         0.145578    0.738137    0.14288
 -0.277316   -0.545251    0.300348       -0.68765     1.20534    -0.320808
 -0.0616053  -0.119039    0.329892        0.238535    0.0353979   0.112729
  0.0672354  -0.0878137  -0.635576       -1.0874      0.708982   -0.597247
  0.0721308  -0.0716083   0.0177225   …  -0.496813    0.484021   -0.203767
 -0.154627   -0.523618   -0.269109       -1.62106     1.71222    -0.836199
  0.245155    0.126372   -0.915456       -0.95324     0.268777   -0.586257
 -0.174877    0.164628    0.879089        1.73179    -1.10885     0.969218
  0.0773281   0.0235047  -0.311736       -0.363162    0.264814   -0.238866
 -0.218165   -0.229869    0.665426    …   0.405718    0.331481    0.287417
  0.267832   -0.01115    -1.09659        -1.67441     0.793391   -0.897267
  0.137153    0.12134    -0.320981       -0.0961022  -0.209312   -0.134629
  ⋮                                   ⋱
 -0.152681   -0.232722    0.490734       -0.0268052   0.611991    0.039019
 -0.0889187  -0.0395822  -0.16075         0.185663   -0.327413    0.105083
 -0.0738991   0.0302181   0.740914    …   1.01515    -0.409348    0.622269
  0.0760725   0.0807437  -0.0703095      -0.0287242  -0.0360732  -0.0989715
  0.141314   -0.0245286  -0.909044       -1.50095     1.04564    -0.806416
 -0.127831   -0.0944111   0.835114        0.639098    0.200252    0.358462
  0.461889    0.636846   -0.780058        0.546031   -1.55416     0.292297
 -0.175867   -0.0144654  -0.00940815  …  -0.071834    0.313137   -0.0472551
  0.132841    0.176066   -0.119316        0.23224    -0.405544    0.109026
 -0.020849    0.0328693   0.516699        0.807197   -0.361508    0.472888
 -0.189136   -0.185462    0.382118        0.288165    0.110801    0.192848
  0.0630003  -0.107217   -0.211561       -0.610211    0.662377   -0.360026
```

In [266…]:
```julia
A = rand(1000, 1000)

@btime inv(A)
```

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

Out[266…]:
```
1000×1000 Array{Float64,2}:
  0.62497    -0.458387   -0.134949    …   0.512443    1.53798    -1.16083
  0.17291    -0.211562   -0.142715        0.347272    0.857607   -0.893295
 -0.543666    0.431714    0.155863       -0.398559   -1.35658     1.10232
 -0.141046    0.239169    0.0773037      -0.228223   -0.859179    0.688993
  1.01181    -0.793356   -0.238073        0.796961    2.39993    -1.87481
  0.709584   -0.569739   -0.156784    …   0.597084    1.86341    -1.53175
 -0.294739    0.358514    0.125133       -0.488936   -1.13131     0.934847
 -0.122143    0.211812    0.0807415      -0.335547   -0.796785    0.728623
 -0.931092    0.920988    0.312525       -0.958089   -2.85479     2.3639
 -0.300808    0.328408   -0.00589672     -0.26086    -0.849034    0.529699
  0.623683   -0.527229   -0.210475    …   0.538549    1.80466    -1.56758
  0.782074   -0.536589   -0.144162        0.58887     1.85478    -1.44964
 -0.358967    0.183945    0.0633861      -0.131894   -0.528916    0.352792
  ⋮                                   ⋱
  0.606286   -0.532536   -0.087893        0.452189    1.55982    -1.08221
 -0.151548    0.0575045   0.00587092     -0.0033015  -0.229861    0.137279
 -0.527934    0.449225    0.242075    …  -0.51754    -1.56565     1.30969
  0.119437   -0.0113074  -0.0525644       0.0882518   0.132097   -0.136744
  0.490483   -0.453626   -0.0186042       0.354675    1.28757    -0.989587
  0.369254   -0.207609   -0.10821         0.265296    0.570155   -0.393563
 -0.409215    0.531732    0.116981       -0.542399   -1.50832     1.04914
 -0.632348    0.324128    0.102967    …  -0.273332   -1.07183     0.807916
  0.313073   -0.316972    0.00103988      0.256204    0.905331   -0.579594
 -0.432383    0.274486    0.110858       -0.372461   -1.08097     0.881925
 -0.987632    0.92885     0.321396       -0.976204   -2.86724     2.35738
 -0.220388    0.250722    0.0531822      -0.309715   -0.88543     0.65921
```

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

In [260…]:
```julia
_I = I(1000)
A = rand(1000, 1000)
B = triu(A)
```

Out[260…]:
```
1000×1000 Array{Float64,2}:
 0.668256  0.500095  0.198305  0.603398  …  0.901597   0.239783   0.147324
 0.0       0.349062  0.166038  0.687794     0.277482   0.71834    0.0785509
 0.0       0.0       0.64725   0.399254     0.443962   0.237851   0.391373
 0.0       0.0       0.0       0.766339     0.286818   0.270388   0.654782
 0.0       0.0       0.0       0.0          0.455284   0.297385   0.194704
 0.0       0.0       0.0       0.0       …  0.210236   0.103857   0.838948
 0.0       0.0       0.0       0.0          0.0476031  0.642534   0.471228
 0.0       0.0       0.0       0.0          0.911331   0.844099   0.54737
 0.0       0.0       0.0       0.0          0.365171   0.328597   0.308367
 0.0       0.0       0.0       0.0          0.460295   0.583315   0.0987463
 0.0       0.0       0.0       0.0       …  0.239453   0.503526   0.292078
 0.0       0.0       0.0       0.0          0.838518   0.569672   0.229836
 0.0       0.0       0.0       0.0          0.859378   0.376304   0.463611
 ⋮                                       ⋱
 0.0       0.0       0.0       0.0          0.987199   0.213412   0.260112
 0.0       0.0       0.0       0.0          0.243775   0.857218   0.767155
 0.0       0.0       0.0       0.0       …  0.243985   0.0759308  0.818727
 0.0       0.0       0.0       0.0          0.092791   0.144529   0.500437
 0.0       0.0       0.0       0.0          0.145906   0.747882   0.0426047
 0.0       0.0       0.0       0.0          0.966844   0.664243   0.170233
 0.0       0.0       0.0       0.0          0.496949   0.293253   0.00311442
 0.0       0.0       0.0       0.0       …  0.968147   0.309902   0.22273
 0.0       0.0       0.0       0.0          0.710106   0.653583   0.233324
```

```
0.0       0.0       0.0       0.0          0.816063  0.571443  0.106854
0.0       0.0       0.0       0.0          0.0       0.672444  0.233029
0.0       0.0       0.0       0.0          0.0       0.0       0.863416
```

In [261…

```
@btime inv(B)
```

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

Out[261…

```
1000×1000 Array{Float64,2}:
 1.49643  -2.14392   0.0914981   0.698251  …   5.40099e198  -2.33354e198
 0.0       2.86482  -0.73491    -2.18832      -1.62421e198   7.01753e197
 0.0       0.0       1.545      -0.804926      1.40368e198  -6.0647e197
 0.0       0.0       0.0         1.3049       -2.01067e199   8.68723e198
 0.0       0.0       0.0         0.0          -2.59761e198   1.12231e198
 0.0       0.0       0.0         0.0       …   6.45524e198  -2.78903e198
 0.0       0.0       0.0         0.0           1.77419e199  -7.66551e198
 0.0       0.0       0.0         0.0          -2.62962e198   1.13615e198
 0.0       0.0       0.0         0.0          -1.65743e198   7.16104e197
 0.0       0.0       0.0         0.0           4.54799e197  -1.96499e197
 0.0       0.0       0.0         0.0       …  -2.01685e198   8.71394e197
 0.0       0.0       0.0         0.0          -3.64057e197   1.57293e197
 0.0       0.0       0.0         0.0           2.94919e198  -1.27422e198
 ⋮                                         ⋱
 0.0       0.0       0.0         0.0         -20.5555       11.6857
 0.0       0.0       0.0         0.0           7.70743      -2.63851
 0.0       0.0       0.0         0.0       …  11.9014       -4.67512
 0.0       0.0       0.0         0.0          -8.03518       1.03479
 0.0       0.0       0.0         0.0          -4.22291       1.72643
 0.0       0.0       0.0         0.0          -5.28121       1.88249
 0.0       0.0       0.0         0.0          -0.375153      0.398597
 0.0       0.0       0.0         0.0       …   6.17229      -2.30802
 0.0       0.0       0.0         0.0          -0.308206     -0.132299
 0.0       0.0       0.0         0.0          -1.04134       0.129397
 0.0       0.0       0.0         0.0           1.48711      -0.401359
 0.0       0.0       0.0         0.0           0.0           1.15819
```

In [262…

```
@btime B\I
```

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

Out[262…

```
1000×1000 Array{Float64,2}:
 1.49643  -2.14392   0.0914981   0.698251  …   5.40099e198  -2.33354e198
 0.0       2.86482  -0.73491    -2.18832      -1.62421e198   7.01753e197
 0.0       0.0       1.545      -0.804926      1.40368e198  -6.0647e197
 0.0       0.0       0.0         1.3049       -2.01067e199   8.68723e198
 0.0       0.0       0.0         0.0          -2.59761e198   1.12231e198
 0.0       0.0       0.0         0.0       …   6.45524e198  -2.78903e198
 0.0       0.0       0.0         0.0           1.77419e199  -7.66551e198
 0.0       0.0       0.0         0.0          -2.62962e198   1.13615e198
 0.0       0.0       0.0         0.0          -1.65743e198   7.16104e197
 0.0       0.0       0.0         0.0           4.54799e197  -1.96499e197
 0.0       0.0       0.0         0.0       …  -2.01685e198   8.71394e197
 0.0       0.0       0.0         0.0          -3.64057e197   1.57293e197
 0.0       0.0       0.0         0.0           2.94919e198  -1.27422e198
 ⋮                                         ⋱
 0.0       0.0       0.0         0.0         -20.5555       11.6857
 0.0       0.0       0.0         0.0           7.70743      -2.63851
 0.0       0.0       0.0         0.0       …  11.9014       -4.67512
 0.0       0.0       0.0         0.0          -8.03518       1.03479
 0.0       0.0       0.0         0.0          -4.22291       1.72643
 0.0       0.0       0.0         0.0          -5.28121       1.88249
 0.0       0.0       0.0         0.0          -0.375153      0.398597
 0.0       0.0       0.0         0.0       …   6.17229      -2.30802
 0.0       0.0       0.0         0.0          -0.308206     -0.132299
 0.0       0.0       0.0         0.0          -1.04134       0.129397
 0.0       0.0       0.0         0.0           1.48711      -0.401359
 0.0       0.0       0.0         0.0           0.0           1.15819
```

In [263…

```
@btime inv(A)
```

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

Out[263…

```
1000×1000 Array{Float64,2}:
 -0.523667   -0.482268    -0.431259    …   0.337907    -0.177193
  0.0302209   0.0881947    0.0795895       0.125973     0.136548
 -0.2323     -0.175653    -0.0670776       0.272359    -0.0551474
 -0.221193   -0.241093    -0.33921         0.46704     -0.159117
 -0.117256   -0.218634    -0.166742        0.116869    -0.10111
  0.100683    0.165526     0.251768    …  -0.0164263    0.146091
 -0.117831   -0.221348    -0.221403        0.0552234   -0.179337
 -0.497778   -0.493654    -0.430099        0.506457    -0.300345
 -0.0535493  -0.100699     0.00524921     -0.0385063   -0.000566493
  0.187716    0.172202     0.189148       -0.186028    -0.0132704
 -0.0657512   0.0135414   -0.0727692   …  -0.20124     -0.183351
  0.186675    0.237737     0.239209       -0.00271675   0.111346
 -0.299828   -0.227547    -0.278998        0.141411    -0.240707
  ⋮                                    ⋱
  0.108108    0.150003     0.218897       -0.223295     0.0817466
  0.181362    0.202014     0.318621       -0.244351     0.196917
 -0.496035   -0.483929    -0.643866    …   0.393983    -0.269871
 -0.231752   -0.198725    -0.378463        0.367829    -0.0990437
 -0.491347   -0.535643    -0.607982        0.384223    -0.289614
  0.280651    0.210103     0.365192       -0.243575     0.222415
 -0.31649    -0.26873     -0.13673         0.152516    -0.21772
 -0.312868   -0.335007    -0.294355    …   0.131205    -0.248601
 -0.0514909  -0.0730094    0.00637453     -0.00946273  -0.00663144
  0.06681    -0.00347339   0.0198718       0.121122    -0.0265237
 -0.311042   -0.2784      -0.159137        0.179849    -0.226274
 -0.164406   -0.218984    -0.216931        0.234302     0.0177381
```

In [264…

```
@btime A\I
```

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

Out[264…

```
1000×1000 Array{Float64,2}:
 -0.523667   -0.482268    -0.431259    …   0.337907    -0.177193
  0.0302209   0.0881947    0.0795895       0.125973     0.136548
 -0.2323     -0.175653    -0.0670776       0.272359    -0.0551474
 -0.221193   -0.241093    -0.33921         0.46704     -0.159117
 -0.117256   -0.218634    -0.166742        0.116869    -0.10111
  0.100683    0.165526     0.251768    …  -0.0164263    0.146091
 -0.117831   -0.221348    -0.221403        0.0552234   -0.179337
 -0.497778   -0.493654    -0.430099        0.506457    -0.300345
 -0.0535493  -0.100699     0.00524921     -0.0385063   -0.000566493
  0.187716    0.172202     0.189148       -0.186028    -0.0132704
 -0.0657512   0.0135414   -0.0727692   …  -0.20124     -0.183351
  0.186675    0.237737     0.239209       -0.00271675   0.111346
 -0.299828   -0.227547    -0.278998        0.141411    -0.240707
  ⋮                                    ⋱
  0.108108    0.150003     0.218897       -0.223295     0.0817466
  0.181362    0.202014     0.318621       -0.244351     0.196917
 -0.496035   -0.483929    -0.643866    …   0.393983    -0.269871
 -0.231752   -0.198725    -0.378463        0.367829    -0.0990437
 -0.491347   -0.535643    -0.607982        0.384223    -0.289614
  0.280651    0.210103     0.365192       -0.243575     0.222415
 -0.31649    -0.26873     -0.13673         0.152516    -0.21772
 -0.312868   -0.335007    -0.294355    …   0.131205    -0.248601
 -0.0514909  -0.0730094    0.00637453     -0.00946273  -0.00663144
  0.06681    -0.00347339   0.0198718       0.121122    -0.0265237
 -0.311042   -0.2784      -0.159137        0.179849    -0.226274
 -0.164406   -0.218984    -0.216931        0.234302     0.0177381
```

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.

In [220…

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

Out[220…

```
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
```

In [221…

```
rationalize.(inv(L))
```

Out[221…

```
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

In [217…

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

Out[217…

```
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
```

In [227…

```
rationalize.(inv(L))
```

Out[227…

```
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
```

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

In [250…

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

Out[250… `q71 (generic function with 1 method)`

In [234…

```
L = q71(6)
rationalize.(L)
```

Out[234…

```
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
```

In [235…

```
rationalize.(inv(L))
```

Out[235…

```
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
```

In [244…

```
L = q71(7)
rationalize.(L)
```

Out[244…

```
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
```

In [245…

```
rationalize.(inv(L))
```

Out[245…

```
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.

In [249…

```
float(428914250225764//1000799917193449) - (3/7)
```

Out[249… `1.6653345369377348e-16`