

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

In [190]: 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]: tr(A)

Out [4]: 3

In [5]: det(A)

Out [5]: 104.0

In [6]: inv(A)

Out [6]: 3×3 Array{Float64,2}:
-0.45923  0.211338  0.086385
 0.365385 -0.132308  0.057623
 0.140385  0.057623 -0.0673077

In [191]: A * inv(A)

Out [191]: 3×3 Array{Float64,2}:
 1.0  0.0  0.0
-2.02045e-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]: rref(A)

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

Solving a linear system

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

Out [187]: 3-element Array{Float64,1}:
 1.0
 0.0
 2.0

Lets try it out with a singular matrix

In [188]: A = [1 2 3; 4 5 6; 7 8 9]
b = [5; 2; 9]
A\b

Stacktrace:
 [1] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:119 [inlined]
 [2] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:121 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at ./Users/julia/buildbot/work
e/r/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:185
 [4] #lu#136 at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/std
lib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v
1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \{::Array{Float64,2}, ::Array{Float64,1}) at ./Users/julia/buildbot/worker/packag
e_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 because  $A$  is singular).
```

LU Decomposition

```
In [192]: # Tri-diagonal matrix
A = triu(ones(4,4))
L = factorize(A)
U = L[1,2,3]

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 [148]: A[F,P,:], :: F = L * F * U

true

The above decomposition did not require any row-exchanges.
```

```
In [146]: 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 = [7; 8]
```

```
In [148]: A[F,P,:] == F * L * F * U

true
```

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

```
In [151]: A = [0 0 1; 0 0 2; 0 0 3]
F = lu(A)
display(F)

SingularException{1}

Stacktrace:
 [1] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:119 [inlined]
 [2] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:121 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at ./Users/julia/buildbot/work
e/r/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:185
 [4] #lu#136 at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/std
lib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v
1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \{::Array{Float64,2}, ::Array{Float64,1}) at ./Users/julia/buildbot/worker/packag
e_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/generic.jl:1116
 [7] top-level scope at in[151]:2
 [8] 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 *LU* *P* decomposition, where P is the permutation matrix. We can avoid the checking by passing the `check=false` parameter

```
In [156]: F = lu(A, check=false)
display(F.L)
display(F.U)
display(F.P)

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}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  2.0
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
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]: A = [0 1; 0 0]

Out [167]: 2×2 Array{Float64,2}:
 0  1
 0  0

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

```
In [170]: L1, U1 = lu(A, check=false)
display(U1)

2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
2×2 Array{Float64,2}:
 0  1
 0  0
```

```
In [179]: L2 = [1 0; 0 1]
U2 = [0 1; 0 -1]
display(U2)

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

```
In [181]: L1 * U1 == A

true
```

```
In [182]: L2 * U2 == A

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 [111]: # part 1
A = [1 1 1; 1 2 2; 2 3 -4]
b = [6; 11; 3]
x = A\b
```

```
Out [111]: 3-element Array{Float64,1}:
 1.0
 2.0
 3.0
```

```
In [112]: # part 2
A = [1 1 1; 1 2 2; 2 3 -4]
b = [7; 10; 3]
x = A\b
```

```
Out [112]: 3-element Array{Float64,1}:
 4.0
 1.0
 2.0
```

32

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

Out [204]: LU{Float64,Array{Float64,2}}
L factor:
 3×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.138241  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
 0.8  0.0  0.603532
```

```
In [205]: U[p][1,1]

Out [205]: 0.7794346449731999
```

```
In [134]: 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}:
 0.53835256101029
 0.540404069682417
 0.539553302227193
```

The average values for each pivot are about 0.5

Section 1.4

21

```
In [198]: 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]: 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]: 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]: A = I(3)
```

```
Out [58]: 3×3 Adjoint{Bool,Array{Bool,1}}:
 1  .  .
 .  1  .
 .  .  1
```

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

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

```
In [65]: v*
```

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

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

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

```
In [67]: v + v
```

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

```
In [68]: v*A
```

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

Stacktrace:
 [1] check_broadcast_shape at ./broadcast.jl:520 [inlined]
 [2] check_broadcast_axes at ./broadcast.jl:523 [inlined]
 [3] check_broadcast_axes at ./broadcast.jl:527 [inlined]
 [4] instantiate at ./broadcast.jl:529 [inlined]
 [5] materialize! at ./broadcast.jl:640 [inlined]
 [6] #materialize! at ./broadcast.jl:645 [inlined]
 [7] #materialize! at ./broadcast.jl:645 [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:132
 [9] * at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/diagonal.jl:185 [inlined]
 [10] \{::Array{Int64,2}, ::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]: 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]: v = ones(4,1)
```

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

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

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

```
In [73]: 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]: 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]: 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 row of backward elimination, we will do a row transformation that subtracts a multiple of the first 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:

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

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

(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 $\frac{p}{q}$ (that is non-zero, it wouldn't make sense to subtract a zero entry). We can write this transformation as

$R_i \rightarrow R_i - \frac{p}{q} \cdot R_j$

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]: # Calculation for part (c)
A = [
 1 1 1 0 0;
 1 1 0 1 0;
 1 1 1 0 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  0.0  0.0
-1.0  1.0  0.0 -1.0
 0.0 -1.0  0.0  1.0
```

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

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

32

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

```
Out [76]: 4×4 Array{Float64,2}:
 4.0  0.0  0.0  0.0
-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]: inv(A)
```

```
Out [78]: 4×4 Array{Float64,2}:
 0.4  0.2  0.2  0.2
 0.4  0.2  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]: 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]: 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]: A = ones(4,4)
b = rand(4,1)
x = A\b
```

```
SingularException{2}

Stacktrace:
 [1] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:119 [inlined]
 [2] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:121 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at ./Users/julia/buildbot/work
e/r/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:185
 [4] #lu#136 at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/std
lib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v
1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \{::Array{Float64,2}, ::Array{Float64,1}) at ./Users/julia/buildbot/worker/packag
e_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]: b = ones(4,1)
x = A\b

SingularException{2}

Stacktrace:
 [1] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:119 [inlined]
 [2] checknon_singular at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/
julia/stdlib/v1.5/LinearAlgebra/src/factorization.jl:121 [inlined]
 [3] lu!(::Array{Float64,2}, ::Val{true}; check::Bool) at ./Users/julia/buildbot/work
e/r/package_macos64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/lu.jl:185
 [4] #lu#136 at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/std
lib/v1.5/LinearAlgebra/src/lu.jl:273 [inlined]
 [5] lu at ./Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v
1.5/LinearAlgebra/src/lu.jl:272 [inlined] (repeats 2 times)
 [6] \{::Array{Float64,2}, ::Array{Float64,1}) at ./Users/julia/buildbot/worker/packag
e_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
 NaN
```

```
>> b = ones(4,1)
Warning: Matrix is singular to working precision.
x =
 NaN
 NaN
 NaN
 NaN
```

Instead of throwing an exception, MATLAB issues a warning.

69

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

@time inv(A)
```

```
Out [265]: 500×500 Array{Float64,2}:
 0.0232059  -0.305865  0.204706
```


		0.0	0.0	0.0	0.0	-	0.496949	0.293253	0.00311442
0.0	0.0	0.0	0.0	0.0	0.0		0.968147	0.309902	0.222233
0.0	0.0	0.0	0.0	0.0	0.0		0.816063	0.571443	0.106854
0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.672444	0.233029
0.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.63616
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		1.40366e198	-6.0647e197	
		0.0	0.0	0.0	1.30493		-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		-2.62962e198	1.13615e198	
		0.0	0.0	0.0	0.0		-1.65743e197	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		-6.37513	0.38897	
		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	0.401359	
		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		1.40366e198	-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		-2.62962e198	1.13615e198	
		0.0	0.0	0.0	0.0		-1.65743e197	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		-6.37513	0.38897	
		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	0.401359	
		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}:							
		1.49643	-2.14392	-0.0914981	0.698251	-	5.40099e198	-2.33354e198	
		-0.0302209	0.0881947	0.0795895	0.125973		0.136548	-0.0551474	
		-0.232152	-0.175653	-0.0670776	0.272359		-0.051474		
		-0.231756	-0.218634	-0.166742	0.116869		-0.10111		
		0.100683	0.165526	0.251768	-	-0.0164263	0.146091		
		-0.117831	-0.221348	0.221403	0.46704		-0.177229	-2.30802	
		-0.497778	-0.493654	-0.430099	0.506457		-0.303445		
		-0.0353493	-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	-0.649866	-	0.393983	-0.269871		
		-0.231752	-0.198725	-0.378463	0.272359		-0.0551474		
		-0.491347	-0.535643	-0.607982	0.384223		-0.289614		
		-0.280651	-0.335643	-0.607982	0.367829		-0.0990437		
		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	-0.00347339	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		
		-0.232152	-0.175653	-0.0670776	0.272359		-0.0551474		
		-0.231756	-0.241093	-0.33921	0.46704		-0.159117		
		-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		-0.303445		
		-0.0353493	-0.100699	0.00524921	-0.0385063		-0.000566493		
		-0.187716	-0.172202	0.189148	-0.186028		-0.0132704		
		-0.0657512	0.0135						