

math3406_diary

December 1, 2021

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!

2 Basic operations

```
[417]: using LinearAlgebra  
       using RowEchelon  
       using BenchmarkTools # For benchmarking, alternative to tic; tac; in MATLAB  
       using Plots  
       gr()
```

```
[417]: Plots.GRBackend()
```

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

```
[48]: 3x3 Array{Int64,2}:  
      1  2  3  
      4  1  6  
      7  8  1
```

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

```
[49]: 3
```

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

```
[50]: 104.0
```

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

```
[51]: 3×3 Array{Float64,2}:
-0.451923   0.211538   0.0865385
 0.365385  -0.192308   0.0576923
 0.240385   0.0576923  -0.0673077
```

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

```
[52]: 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

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

```
[53]: 3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

3 Solving a linear system

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

```
[54]: 3-element Array{Float64,1}:
 1.0
 1.0
 2.0
```

Lets try it out with a singular matrix

```
[55]: 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[55]: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).

3.1 LU Decomposition

```
[56]: # Tri-diagonal matrix
A =
[
    1 -1 0 0;
    -1 2 -1 0;
    0 -1 2 -1;
    0 0 -1 2;
]
F = lu(A)
```

```
[56]: 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
```

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

```
[57]: true
```

The above decomposition did not require any row-exchanges.

```
[58]: 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]
```

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

```
[59]: 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

```
[60]: 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[60]: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

```
[61]: 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)

3.2 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.

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

```
[62]: 2×2 Array{Int64,2}:
```

```
0 1
0 2
```

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

```
[63]: 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
```

```
[64]: 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
```

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

```
[65]: true
```

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

```
[66]: true
```

4 Chapter 1 HW

4.1 Section 1.3

4.1.1 30

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

```
[67]: # part 1
```

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

```
[67]: 3-element Array{Float64,1}:
 1.0
 3.0
 2.0
```

```
[68]: # part 2
```

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

```
[68]: 3-element Array{Float64,1}:
 4.0
 1.0
 2.0
```

4.1.2 32

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

```
[69]: LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
 1.0      0.0      0.0
 0.491171  1.0      0.0
 0.873102  0.788027  1.0
U factor:
3×3 Array{Float64,2}:
 0.814736  0.184385  0.953432
 0.0        0.34116   0.448749
 0.0        0.0       -0.610216
```

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

```
[70]: 0.0
```

```
[71]: 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
```

```
[71]: 3-element Array{Float64,1}:
 0.5389110954089431
 0.5393148888069205
 0.5403582697144874
```

The average values for each pivot are about 0.5

4.2 Section 1.4

4.2.1 21

```
[72]: 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]
```

```
[73]: 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]
```

```
[74]: 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]
```

$$A^k = A$$

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

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

$$k \in \mathbb{N}$$

4.2.2 59

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

```
[75]: 3x3 Diagonal{Bool,Array{Bool,1}}:
```

```
1
```

```
1
```

```
1
```

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

```
[76]: 3-element Array{Int64,1}:
```

```
3  
4  
5
```

```
[77]: v'
```

```
[77]: 1×3 Adjoint{Int64,Array{Int64,1}}:
```

```
3 4 5
```

```
[78]: A*v
```

```
[78]: 3-element Array{Int64,1}:
```

```
3  
4  
5
```

```
[79]: v' * v
```

```
[79]: 50
```

```
[80]: 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[80]:1
[12] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:
 ↪1091
```

4.2.3 60

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

```
[81]: 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
```

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

```
[82]: 4×1 Array{Float64,2}:
 1.0
 1.0
 1.0
 1.0
```

```
[83]: A * v
```

```
[83]: 4×1 Array{Float64,2}:
 4.0
 4.0
 4.0
 4.0
```

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

```
[84]: 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
```

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

```
[85]: 4×1 Array{Float64,2}:
 2.0
 2.0
 2.0
 2.0
```

```
[86]: B * w
```

```
[86]: 4×1 Array{Float64,2}:
 10.0
 10.0
 10.0
```

10.0

4.3 Section 1.6

4.3.1 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 subtracts 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

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

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.

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

```
[87]: 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
```

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

```
[88]: 2×2 Array{Float64,2}:
 1.0  0.0
 0.0  0.5
```

4.3.2 32

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

```
[89]: 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
```

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

```
[90]: 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 $b = 0.2$

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

```
[91]: 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
```

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

```
[92]: 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 $b = 0.1667$

4.3.3 47

```
[93]: 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[93]:3
[8] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091
```

```
[94]: 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[94]: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.

4.3.4 69

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

@btime inv(A);
```

7.813 ms (5 allocations: 2.16 MiB)

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

@btime inv(A);
```

39.475 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.

4.3.5 70

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

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

15.520 ms (2 allocations: 7.63 MiB)

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

15.759 ms (2 allocations: 7.63 MiB)

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

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

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

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

The running times are almost the same.

\ performs slightly better than inv

4.3.6 71

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

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

```
[102]: 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
```

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

```
[103]: 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

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

```
[104]: 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
```

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

```
[105]: 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

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

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

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

```
[107]: 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
```

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

```
[108]: 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
```

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

```
[109]: 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
```

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

```
[110]: 7×7 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  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.

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

```
[111]: 1.6653345369377348e-16
```

5 Matrix Inverse

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

```
[112]: 2×2 Array{Float64,2}:
 1.5  -0.5
 -0.5   0.25
```

```
[113]: # 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/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/src/dense.jl:781
[7] top-level scope at In[113]: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 $A^{-1} = U^{-1}L^{-1}$

6 Transpose

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

A'
```

```
[114]: 3×3 Adjoint{Int64,Array{Int64,2}}:
  1  4  7
  2  5  8
  3  6  9
```

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

```
[115]: A =
[
  2 4;
  0 1;
]

inv(A)
```

```
[115]: 2×2 Array{Float64,2}:
  0.5  -2.0
  0.0   1.0
```

```
[116]: A'
```

```
[116]: 2×2 Adjoint{Int64,Array{Int64,2}}:  
2 0  
4 1
```

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

```
[117]: true
```

7 Symmetric Matrices

```
[118]: A =  
[  
    1 2 3;  
    2 1 2;  
    3 2 1;  
]  
  
A' == A
```

```
[118]: true
```

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

```
[119]: true
```

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

```
[120]: A =  
[  
    2 1 1;  
    1 2 0;  
    1 0 2;  
]  
  
cholesky(A)
```

```
[120]: 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
```

```
[121]: 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/build/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 times)
[6] top-level scope at In[121]:8
[7] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

```

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

8 Vector Spaces

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

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

8.1 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 . $\text{span}(A)$ - Null space: Set of solutions to $Ax = 0$

[122]: # Finding the null space

```

A =
[
    1 0 0;

```

```
    0 1 0;  
    0 0 0;  
]  
  
nullspace(A)
```

[122]: 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.

[123]: A =

```
[  
    1 0 1;  
    5 4 9;  
    2 4 6;  
]  
nullspace(A)
```

[123]: 3×1 Array{Float64,2}:

```
0.5773502691896256  
0.577350269189626  
-0.5773502691896257
```

[124]: A =

```
[  
    1 0 0;  
    0 0 0;  
    0 0 0;  
]  
nullspace(A)
```

[124]: 3×2 Array{Float64,2}:

```
0.0 0.0  
0.0 1.0  
1.0 0.0
```

[125]: A =

```
[  
    1 0 0;  
    0 0 0;  
    0 0 0;  
]  
rank(A)
```

```
[125]: 1
```

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

```
[126]: 2
```

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

```
[127]: 3
```

8.1.1 Section 1.7, q8

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

```
[128]: 3×3 Array{Rational{Int64},2}:
 9//1   -36//1   30//1
 -36//1   192//1  -180//1
 30//1  -180//1   180//1
```

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

```
[129]: 3×3 Array{Float64,2}:
 9.67066   -39.5082    33.2836
 -39.5082    210.186   -196.951
 33.2836   -196.951    195.772
```

```
[130]: H =
[
    1 1//2 1//3;
```

```

1//2 1//3 1//4;
1//3 1//4 1//5
]
float(H * [1;1;1])

```

[130]: 3-element Array{Float64,1}:

```

1.833333333333333
1.083333333333333
0.783333333333333

```

[131]: H * [0; 6; -3.6]

[131]: 3-element Array{Float64,1}:

```

1.8
1.1
0.7799999999999999

```

[132]: rref(H)

[132]: 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 .

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

[133]: 3×3 Array{Float64,2}:

```

0.75 -0.5 0.25
-0.5 1.0 -0.5
0.25 -0.5 0.75

```

9 Unitary Matrices

A matrix U is said to be unitary if $UU^\dagger = U^\dagger U = I$. The \dagger 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

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

```
U = [
    1 0;
    0 1;
]

U == U'
```

```
[134]: true
```

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

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

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

```
[136]: # 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
```

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

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

```
[137]: true
```

```
[138]: 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))
```

```
[138]: true
```

10 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

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

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

```
[139]: # 15 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 /Applications/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/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 /Applications/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/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/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/diagonal.jl:576
[14] exp(J::UniformScaling) in LinearAlgebra at /Applications/Julia-1.5.app/Contents/Resources/julia/share/julia/stdlib/v1.5/LinearAlgebra/src/uniformscaling.jl
:139
[15] exp(A::StaticArrays.StaticArray{Tuple{N,M},T,2} where T where M where N) in
StaticArrays at
/Users/eltonpinto/.julia/packages/StaticArrays/0WJK7/src/expm.jl:1

```

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.

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

```
[140]: 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
```

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

```
[141]: 3×3 Diagonal{Int64, Array{Int64, 1}}:
 1
 5
 9
```

11 Hermitian Matrices

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

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

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.

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

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

false

```
[142]: 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 $\lambda_j = e^{i\theta_j}$

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

Note that there can be many such H .

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

H = log(U)/im
```

```
[143]: 2×2 Array{Complex{Float64},2}:
 1.5708-0.0im      0.0+1.5708im
 0.0-1.5708im   1.5708-0.0im
```

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

```
[144]: 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

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

```
[145]: true
```

12 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!

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

```
[146]: 2×2 Array{Int64,2}:  
       1 2  
       3 6
```

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

```
[147]: 2×2 Array{Float64,2}:  
       1.0 2.0  
       0.0 0.0
```

Clearly, the above matrix has only one pivot.

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

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

```
[148]: 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 $Ax = 0$

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

```
[149]: 2×1 Array{Float64,2}:
 0.0
 0.0
```

It does!

Note that the Rank theorem is satisfied!

$$\text{rank} = \dim(\text{Col}(A)) = 1$$

$$\text{nullity} = \dim(\text{Nul}(A)) = 1$$

$$n = \text{rank} + \text{nullity} = 2$$

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

```
[150]: 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).

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

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

```
[151]: 2×1 Array{Float64,2}:
 -0.9486832980505138
 0.316227766016838
```

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

```
[152]: 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.

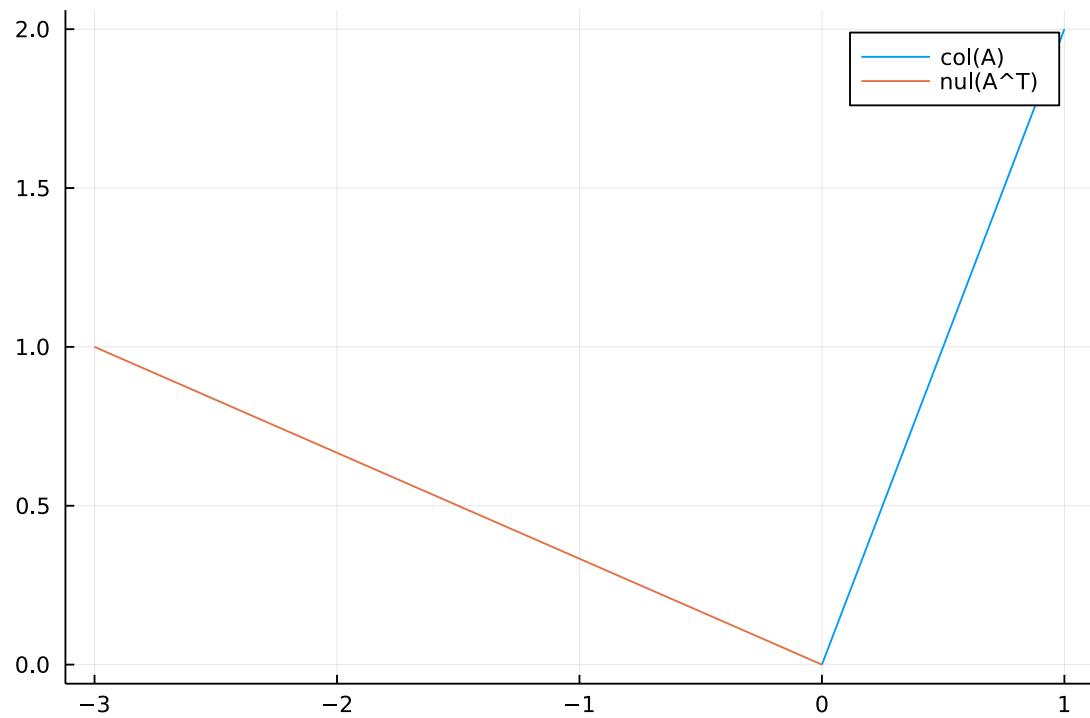
```
[153]: round.(A' * nullspace(A'))
```

```
[153]: 2×1 Array{Float64,2}:
 0.0
 0.0
```

Let's try to plot these spaces

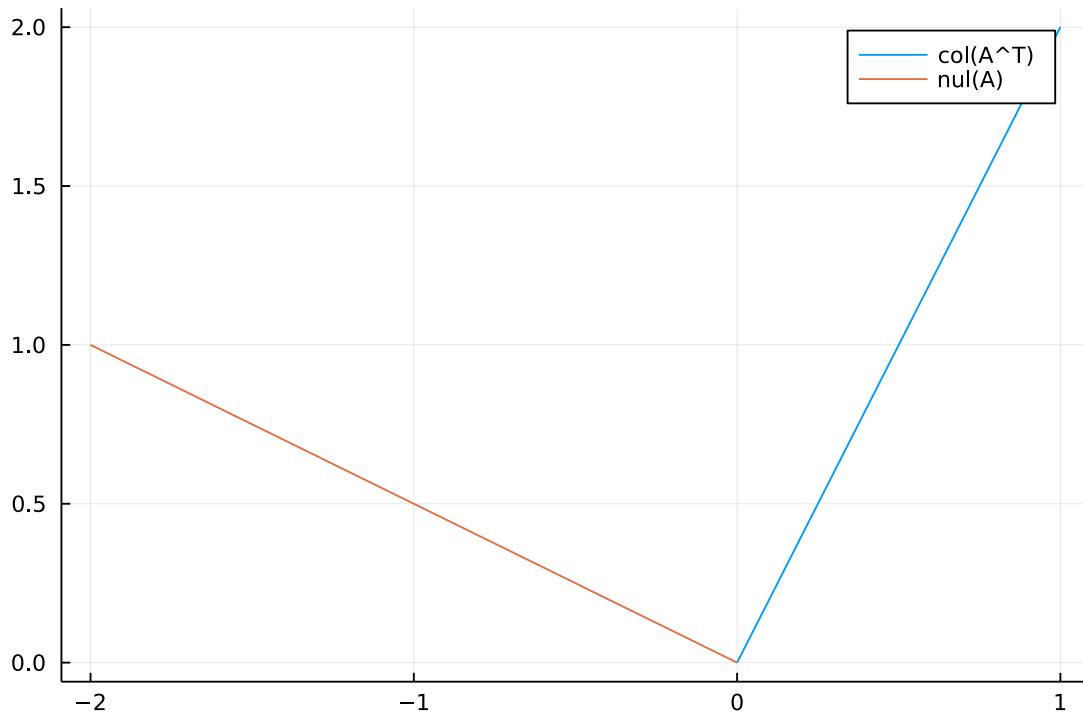
```
[154]: x = [[0, 1], [0, -3]]
y = [[0, 2], [0, 1]]
plot(x, y, label=["col(A)" "nul(A^T)"])
```

```
[154]:
```



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

```
[155]:
```



We can see that the spaces are orthogonal to each other

12.1 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., $\text{rank} = m$)
- Left inverse: full column rank (i.e., $\text{rank} = n$)
- Full inverse: $m = n = \text{rank}$

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.

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

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

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

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

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

```
[158]: 3×2 Array{Float64,2}:
0.25  0.0
0.0   0.2
0.0   0.0
```

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

```
[159]: 2×2 Array{Float64,2}:
1.0  0.0
0.0  1.0
```

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

```
[160]: 2×3 Array{Float64,2}:
0.25  0.0  0.0
0.0   0.2  0.0
```

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

```
[161]: 2×2 Array{Float64,2}:
1.0  0.0
0.0  1.0
```

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

```
[162]: 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 $b \in \mathbb{R}^n$

13 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!

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

[163]: 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.

14 Chapter 2 HW

14.1 Section 2.2

14.1.1 33

```
[164]: A = [
    1 3 3;
    2 6 9;
    -1 -3 3;
]
b = [
    1;
    5;
    5;
]
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{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[164]: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.

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

```
[165]: true
```

```
[166]: F.L
```

```
[166]: 3×3 Array{Float64,2}:
```

```
1.0  0.0  0.0  
0.5  1.0  0.0  
-0.5 0.0  1.0
```

```
[167]: F.U
```

```
[167]: 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.

$$x_1 = -3x_2 - 4.5x_3$$

$$x_2 = x_2$$

$$x_3 = 0$$

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

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

```
[168]: 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:

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

14.1.2 35

TODO

14.1.3 36

```
[169]: # (a)
```

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

```
]
```

```
rref(A)
```

[169]: 3×3 Array{Float64,2}:

```
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
```

[170]: lu(A)

[170]: 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, $Col(A) = \mathbb{R}^{\mathbb{M}}$ and $Nul(A) = \{0\}$

[171]: # (b)

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

```
rref(A)
```

[171]: 3×3 Array{Float64,2}:

```
1.0 0.0 -2.0
0.0 1.0 3.0
0.0 0.0 0.0
```

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

15 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:

$$a + b \leq 5 \tag{1}$$

$$3a + 2b \leq 12 \tag{2}$$

$$a \geq 0 \tag{3}$$

$$b \geq 0 \tag{4}$$

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

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

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

$$Ax \leq b \tag{5}$$

$$x \geq 0 \tag{6}$$

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.

```
[172]: using JuMP  
using GLPK
```

```
[173]: 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)
```

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

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

[173]: 27.0

Julia lets programmers define macros which can help in removing a lot of boilerplate 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!

(From DPV Chapter 7, 7.3)

A cargo plane can carry a maximum weight of 100 tons and a maximum volume of 60 cubic meters. There are three materials to be transported, and the cargo company may choose to carry any amount of each, up to the maximum available limits given below.

- Material 1 has density 2 tons/cubic meter, maximum available amount 40 cubic meters, and revenue \$1,000 per cubic meter.
- Material 2 has density 1 ton/cubic meter, maximum available amount 30 cubic meters, and revenue \$1,200 per cubic meter.
- Material 3 has density 3 tons/cubic meter, maximum available amount 20 cubic meters, and revenue \$12,000 per cubic meter.

Write a linear program that optimizes revenue within the constraints.

```
[174]: model = Model(GLPK.Optimizer)
# Let v1, v2, v3 be the volumes of Material 1, Material 2, and Material 3
@variable(model, v1 >= 0)
@variable(model, v2 >= 0)
@variable(model, v3 >= 0)
@objective(model, Max, 1000v1 + 1200v2 + 12000v3)
# Density = Mass / Volume
@constraint(model, c1, 2v1 + v2 + 3v3 <= 100)
@constraint(model, c2, v1 + v2 + v3 <= 60)
@constraint(model, c3, v1 <= 40)
@constraint(model, c4, v2 <= 30)
@constraint(model, c5, v3 <= 20)
print(model)
optimize!(model)
@show value(v1)
@show value(v2)
@show value(v3)
@show objective_value(model)
```

$$\begin{aligned} \text{max } & 1000v_1 + 1200v_2 + 12000v_3 \\ \text{Subject to } & 2v_1 + v_2 + 3v_3 \leq 100.0 \\ & v_1 + v_2 + v_3 \leq 60.0 \\ & v_1 \leq 40.0 \\ & v_2 \leq 30.0 \\ & v_3 \leq 20.0 \\ & v_1 \geq 0.0 \\ & v_2 \geq 0.0 \\ & v_3 \geq 0.0 \\ \\ \text{value}(v1) & = 5.0 \\ \text{value}(v2) & = 30.0 \\ \text{value}(v3) & = 20.0 \\ \text{objective_value}(model) & = 281000.0 \end{aligned}$$

```
[174]: 281000.0
```

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

Let $G = (V, E, s, t, c)$ be a network where s is the source vertex, t is the sink vertex, and c is the edge-capacities.

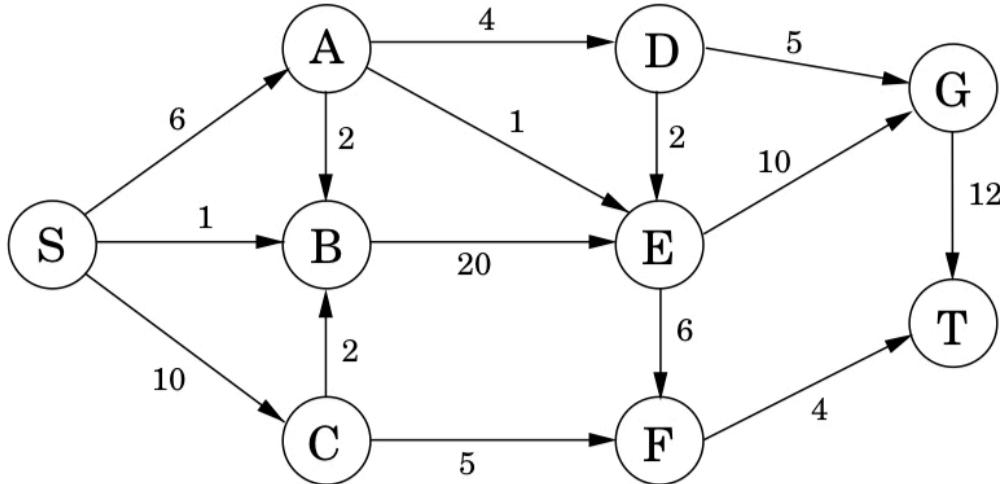
Every flow f of the network must obey the following two laws:

- Conservation of Flow: $\sum_{(u,v) \in E} f_{uv} = \sum_{(v,t) \in E} f_{vt}$, for $v \in V \setminus \{s, t\}$
- Capacity constraint: $f_e \leq c_e \forall e \in E$

Objective: Given G , we wish to find the maximum flow in the network.

We can see that each constraint is linear. This means that we can formulate this problem as an LP.

Here is an example max-flow problem:



[175]: # Encode the graph

```

V = Set(["S", "A", "B", "C", "D", "E", "F", "G", "T"])
E = Set([
    ("S", "A"),
    ("S", "B"),
    ("S", "C"),
    ("A", "B"),
    ("A", "E"),
    ("A", "D"),
    ("B", "E"),
    ("C", "B"),
    ("C", "F"),
    ("D", "E"),
    ("D", "G"),
    ("E", "F"),
    ("E", "G"),
    ("F", "T"),
    ("G", "T")
])
s = "S"
t = "T"
V_s = setdiff(V, Set([s,t]))
E_s = setdiff(E, Set([s,t]))
c = Dict(
    ("S", "A") => 6,
    ("S", "B") => 1,
    ("S", "C") => 10,
    ("A", "B") => 2,
    ("A", "E") => 1,
    ("A", "D") => 4,
    ("B", "E") => 20,
    ("B", "D") => 5,
    ("C", "B") => 2,
    ("C", "F") => 5,
    ("D", "E") => 2,
    ("D", "G") => 5,
    ("E", "F") => 6,
    ("E", "G") => 10,
    ("F", "T") => 4,
    ("G", "T") => 12
)
  
```

```

("S", "B") => 1,
("S", "C") => 10,
("A", "B") => 2,
("A", "E") => 1,
("A", "D") => 4,
("B", "E") => 20,
("C", "B") => 2,
("C", "F") => 5,
("D", "E") => 2,
("D", "G") => 5,
("E", "F") => 6,
("E", "G") => 10,
("F", "T") => 4,
("G", "T") => 12
)

```

[175]: Dict{Tuple{String,String},Int64} with 15 entries:

```

("B", "E") => 20
("D", "G") => 5
("S", "A") => 6
("S", "C") => 10
("C", "F") => 5
("F", "T") => 4
("A", "B") => 2
("S", "B") => 1
("G", "T") => 12
("A", "E") => 1
("D", "E") => 2
("C", "B") => 2
("E", "F") => 6
("A", "D") => 4
("E", "G") => 10

```

```

[176]: model = Model(GLPK.Optimizer)
@variable(model, flow[e in E] >= 0)
@objective(model, Max, sum(flow[(s,u)] for u in V if (s,u) in E))
@constraint(model, cap[e in E], flow[e] <= c[e])
@constraint(
    model,
    cons[v in V_s],
    sum(flow[(u,v)] for u in V if (u,v) in E) == sum(flow[(v,t)] for t in V if (v,t) in E)
)
optimize!(model)
outgoing_flow = sum(value(flow[(s,u)]) for u in V if (s,u) in E)

```

[176]: 13.0

The max-flow is 13!

Note that there are several other ways to find the s - t max-flow. One popular algorithm is the Edmunds-Karps algorithm which has a running time of $O(|V||E|^2)$.

We haven't covered the notion of a dual LP in our presentation. An important consequence of the dual in the context of max-flows is that the max-flow is equal to the min-cut.

15.1 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 constraint $Ax \leq b$ 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.

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

16 Least-Squares

We've learned about least-squares in class. Let's try to solve some least-squares problem in Julia and plot the results.

16.1 Part 1: New York Stock Exchange data

I downloaded a [Medical Cost Personal Dataset](#) from Kaggle which has data on insurance costs and BMI.

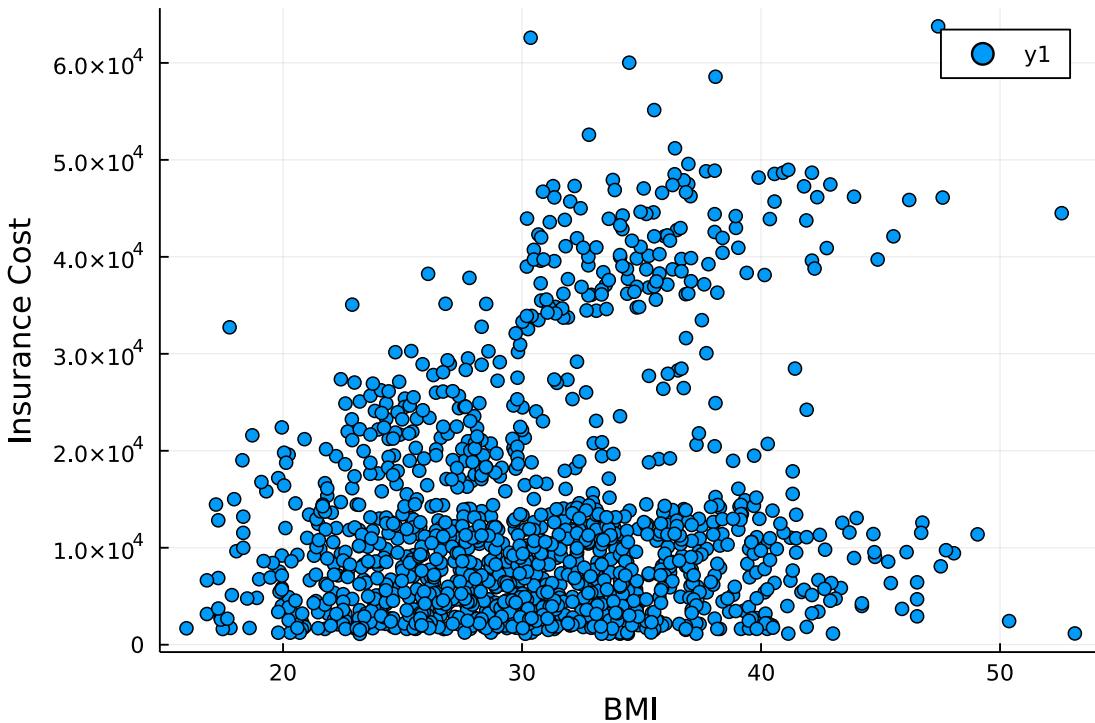
I'm going to try to fit a straight line $y = mx + b$ onto "BMI vs Insurance Cost"

```
[178]: using CSV
using DataFrames
```

```
[179]: df = DataFrame(CSV.File("assets/medical_insurance/insurance.csv"));
```

```
[180]: p = plot(df."bmi", df."charges", seriestype = :scatter, xlabel="BMI",
              ylabel="Insurance Cost")
```

```
[180]:
```



```
[181]: A = hcat(ones(Float64, length(df."bmi")), df."bmi")
b = df."charges"

x = (inv(A'A))A'b
```

```
[181]: 2-element Array{Float64,1}:
1192.9372089610724
393.873030797398
```

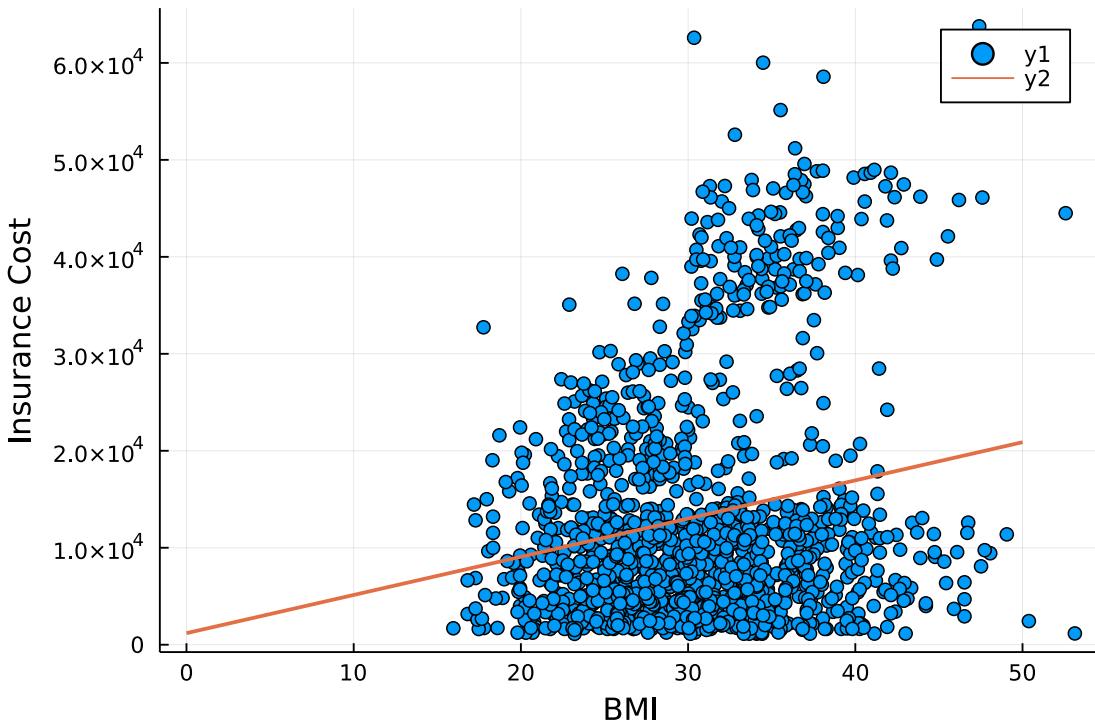
Therefore, the best-fit line is $y = 393.87x + 1192.93$

```
[182]: b,m = x
y = a -> m*a + b
```

```
[182]: #13 (generic function with 1 method)
```

```
[183]: plot!(p, [0, 50], [y(0), y(50)], linewidth=2)
```

```
[183]:
```



16.2 Part 2: Fitting an ellipse

First, let's generate some random datapoints for the ellipse.

The parametric representation of an ellipse:

- $x = a \cos(t)$
- $y = b \sin(t)$

Also, $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$

```
[184]: using Random
```

```
[185]: n = 1000

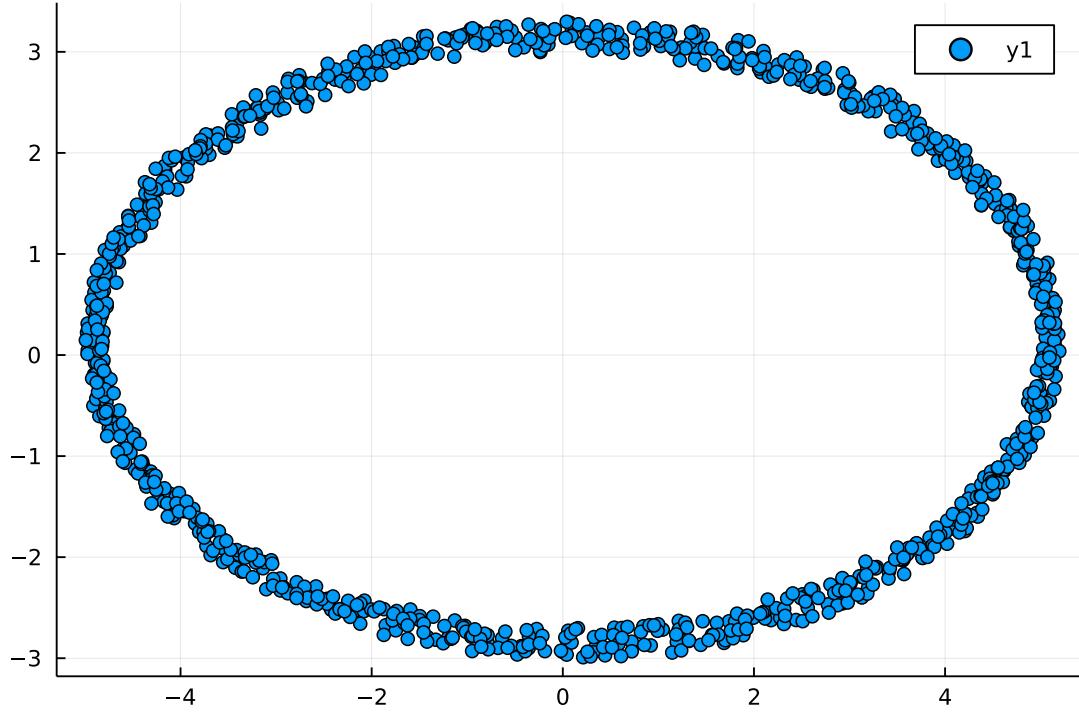
a = 5
da = 0.2

b = 3
db = 0.3

t = 1:n
x = a * cos.(t) + da * rand(Float64, n)
y = b * sin.(t) + db * rand(Float64, n)
```

```
p = plot(x, y, seriestype=:scatter)
```

[185]:



```
A = hcat(x.^2, y.^2)
b = hcat(ones(n))
x = inv(A'A)A'b
```

[186]: 2×1 Array{Float64,2}:

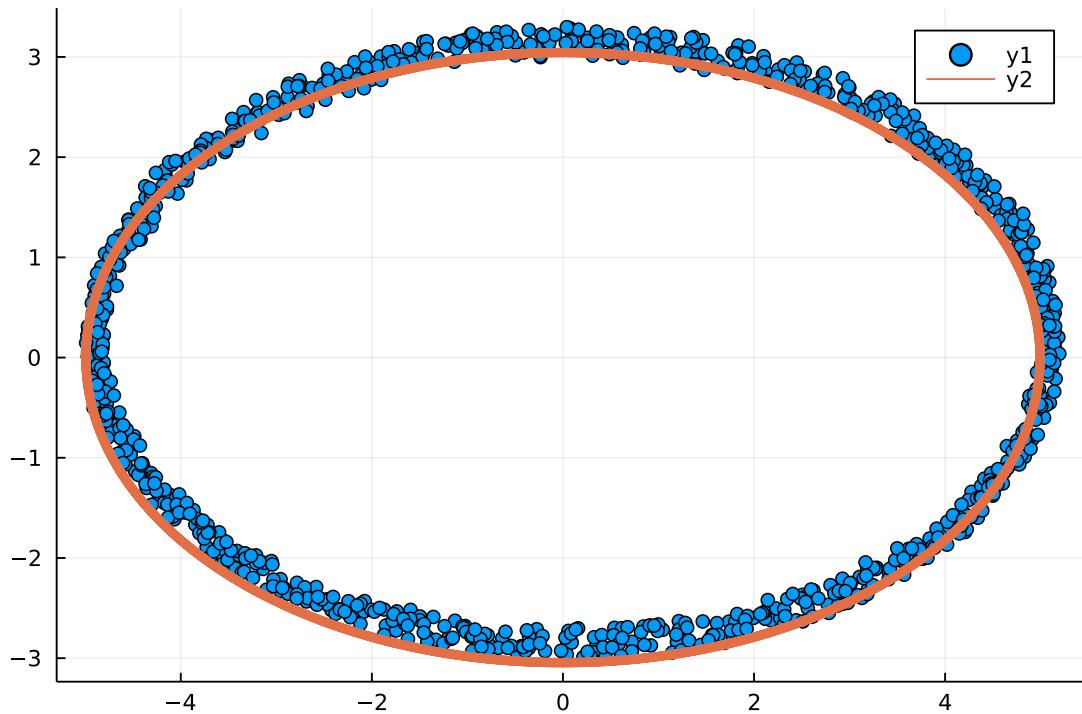
```
0.04010651105222368
0.10789327225330449
```

```
a_inv, b_inv = x
a = √(1/a_inv)
b = √(1/b_inv)
```

[187]: 3.0444077411940507

```
tvec = range(0, 4, length = n)
plot!(p, a*cos.(tvec), b*sin.(tvec), linewidth=5)
```

[188]:



17 Discrete Fourier Transform

```
[189]: using WAV
using FFTW
```

```
[190]: fft([0; 1; 2; 1])
```

```
[190]: 4-element Array{Complex{Float64},1}:
 4.0 + 0.0im
-2.0 + 0.0im
 0.0 + 0.0im
-2.0 + 0.0im
```

```
[191]: y_all, fy = wavread("assets/chord_progression.wav")
```

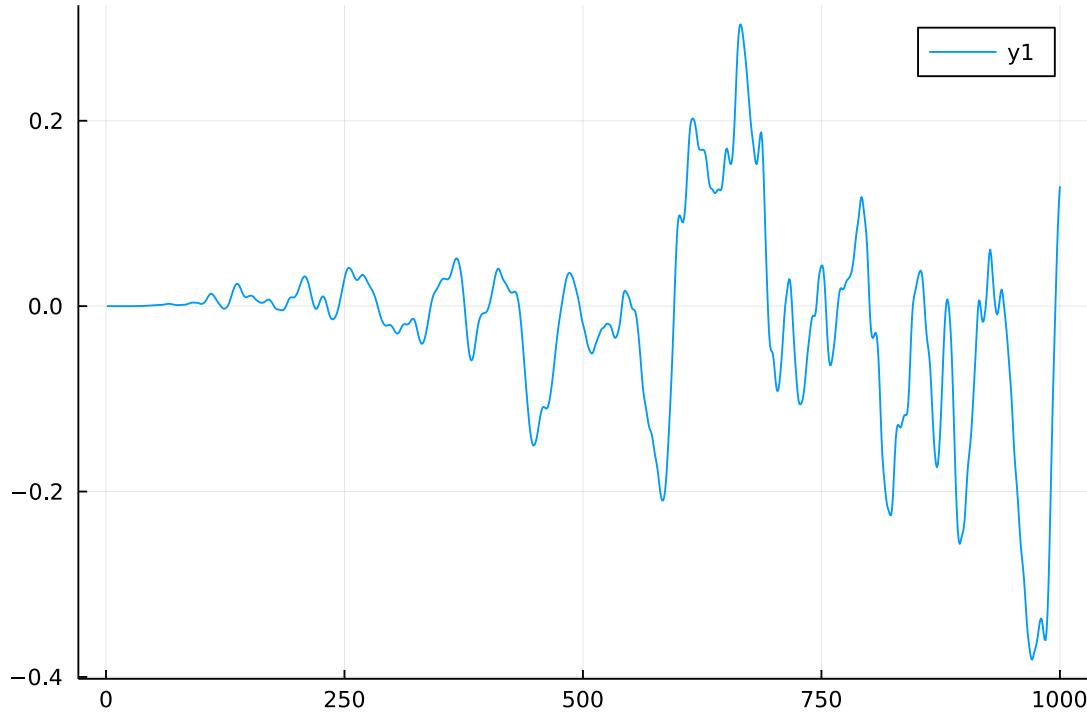
```
[191]: ([0.0 0.0; 0.0 0.0; ... ; -0.005859553819391461 -0.00598162785729545;
-0.0030823694570757164 -0.0031128879665517136], 44100.0f0, 0x0010,
WAVChunk[WAVChunk(:JUNK, UInt8[0x08, 0xfc, 0x2e, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00 ... 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]), 
WAVChunk(Symbol("fmt "), UInt8[0x10, 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00,
0x44, 0xac, 0x00, 0x00, 0x10, 0xb1, 0x02, 0x00, 0x04, 0x00, 0x10, 0x00]),
WAVChunk(Symbol("cue "), UInt8[0x02, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
0x00, 0x00 ... 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]),
```

```
WAVChunk(:LIST, UInt8[0x61, 0x64, 0x74, 0x6c, 0x6c, 0x61, 0x62, 0x6c, 0x12, 0x00  
... 0x53, 0x65, 0x63, 0x74, 0x69, 0x6f, 0x6e, 0x20, 0x41, 0x00]))
```

```
[192]: y = [first(row) for row in eachrow(y_all)];
```

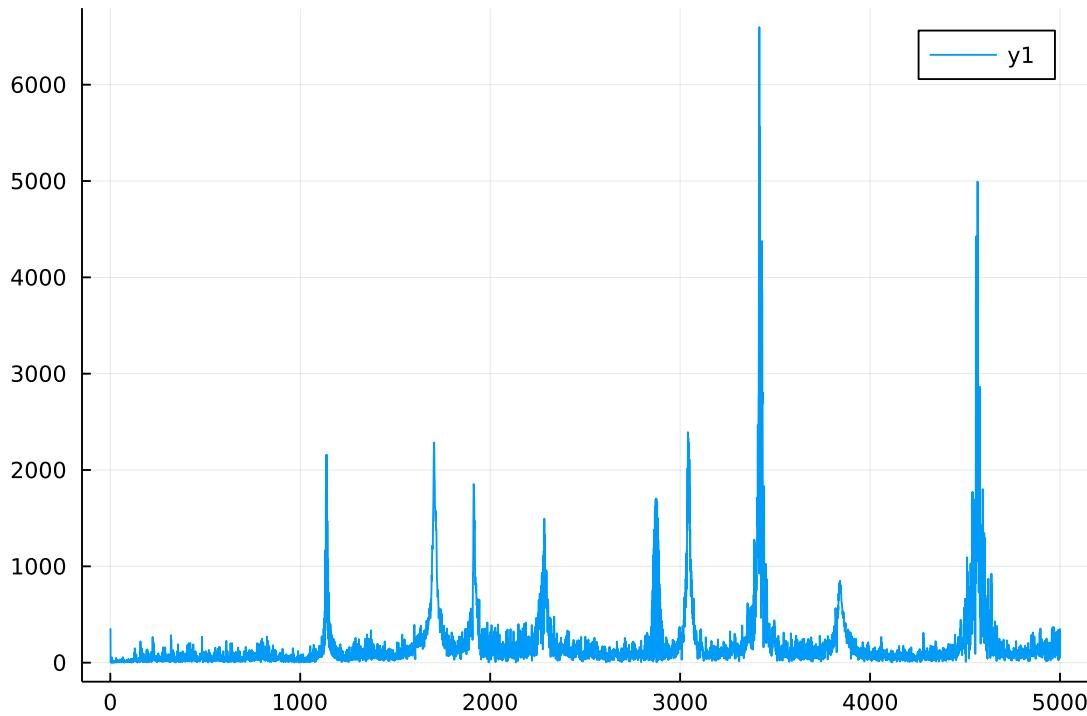
```
[193]: plot(y[1:1000])
```

```
[193]:
```



```
[194]: plot(abs.(fft(y)[1:5000]))
```

```
[194]:
```



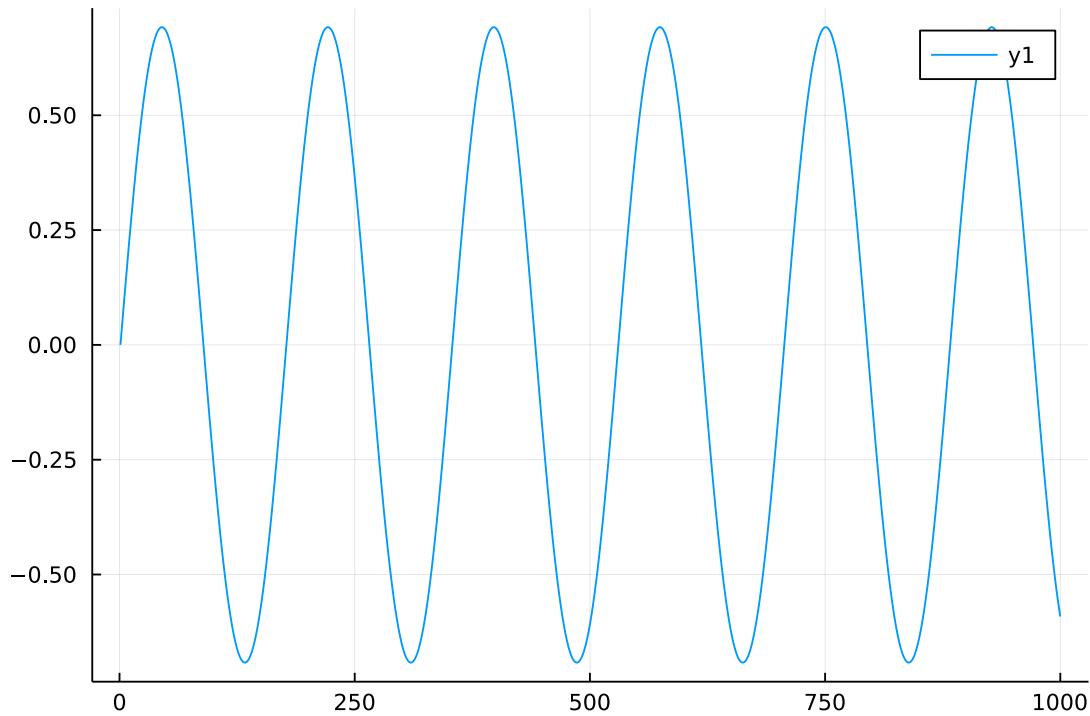
Now let's look at sounds of specific frequencies and try to combine the plots together.

```
[195]: files = ["assets/250hz.wav", "assets/440hz.wav", "assets/1khz.wav"]
ys = first.(wavread.(files));
```

17.1 250Hz

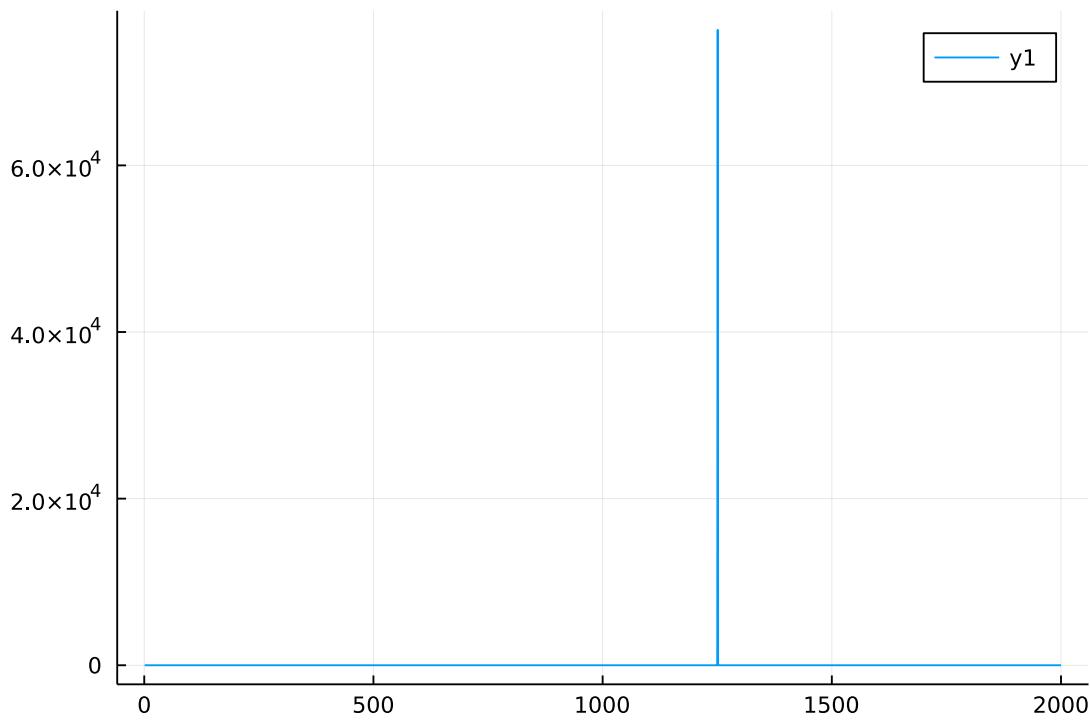
```
[196]: plot(ys[1][1:1000])
```

```
[196]:
```



```
[197]: plot(abs(fft(ys[1]))[1:2000])
```

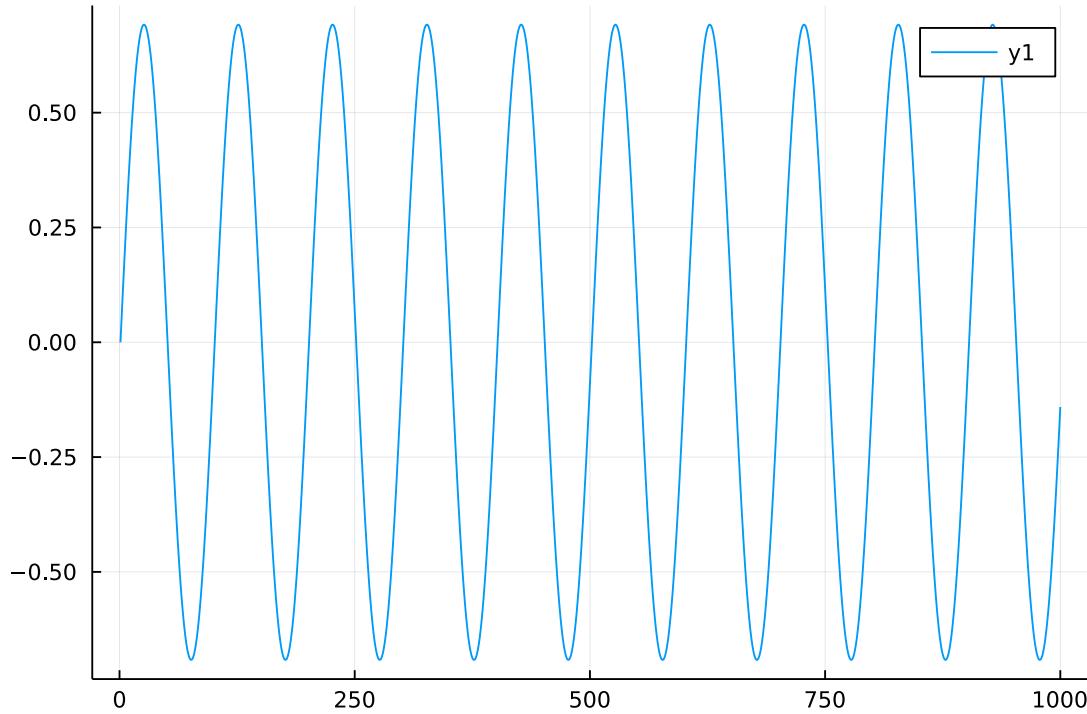
[197]:



17.2 440Hz

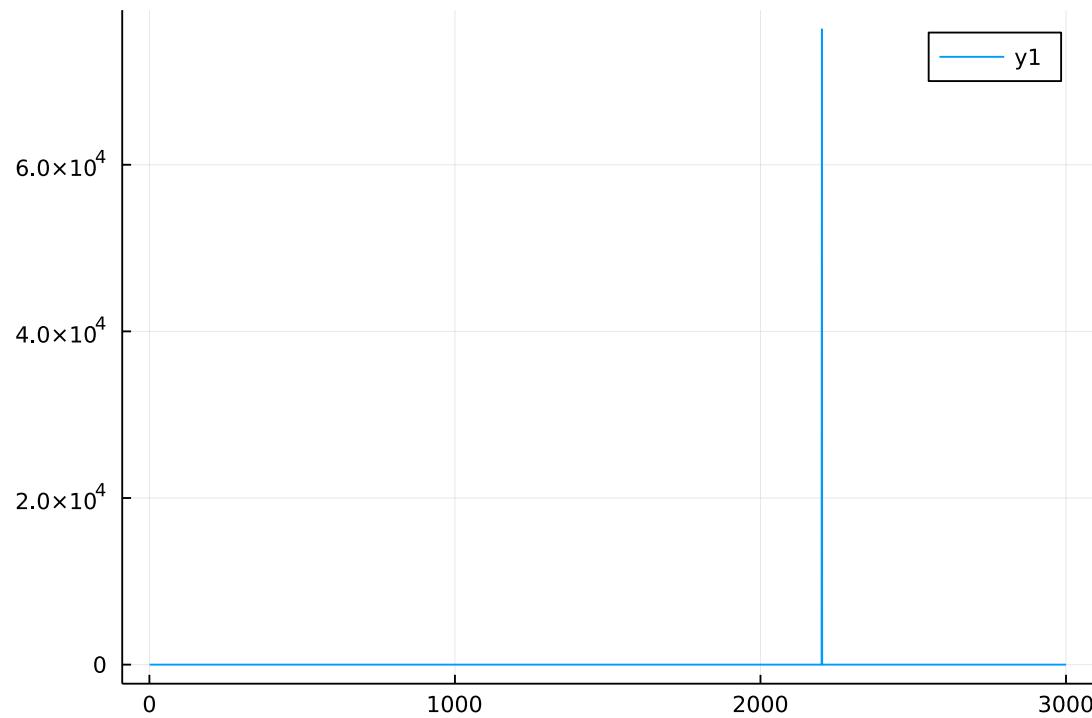
```
[198]: plot(ys[2][1:1000])
```

```
[198]:
```



```
[199]: plot(abs(fft(ys[2]))[1:3000])
```

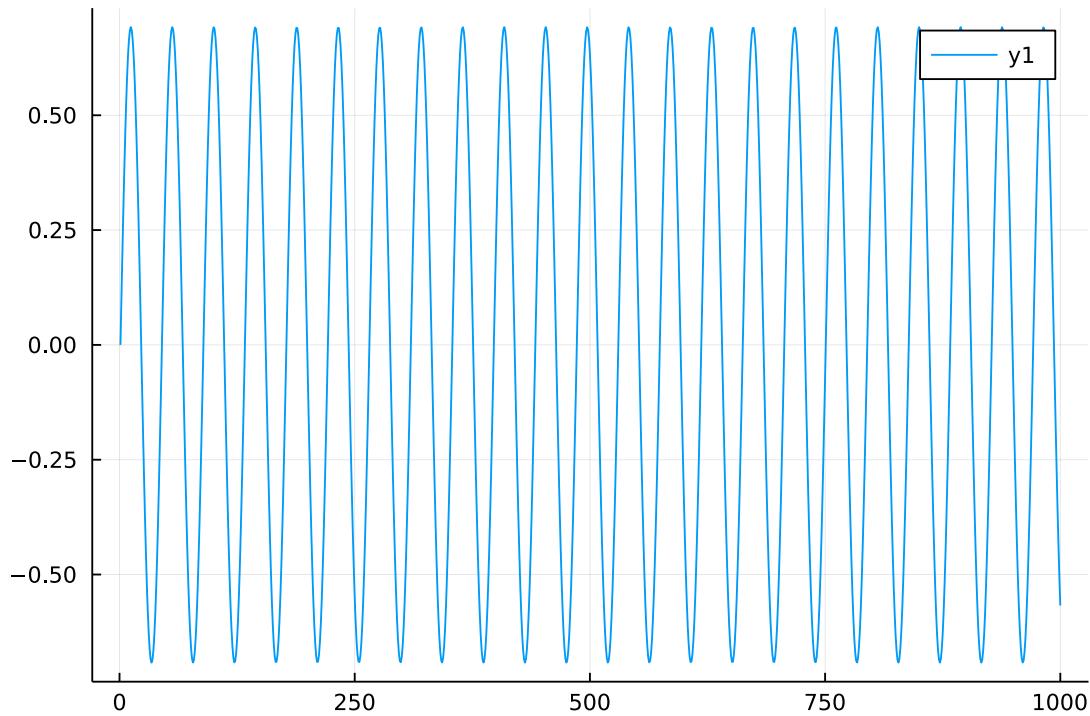
```
[199]:
```



17.3 1kHz

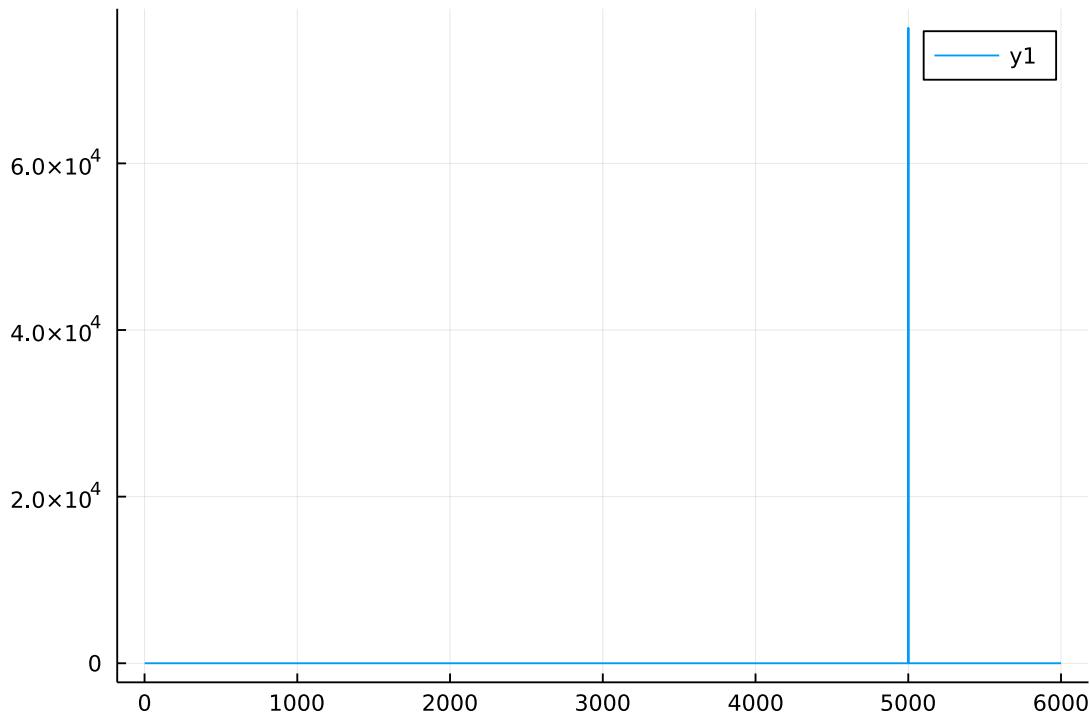
```
[200]: plot(ys[3][1:1000])
```

```
[200]:
```



```
[201]: plot(abs.(fft(ys[3]))[1:6000])
```

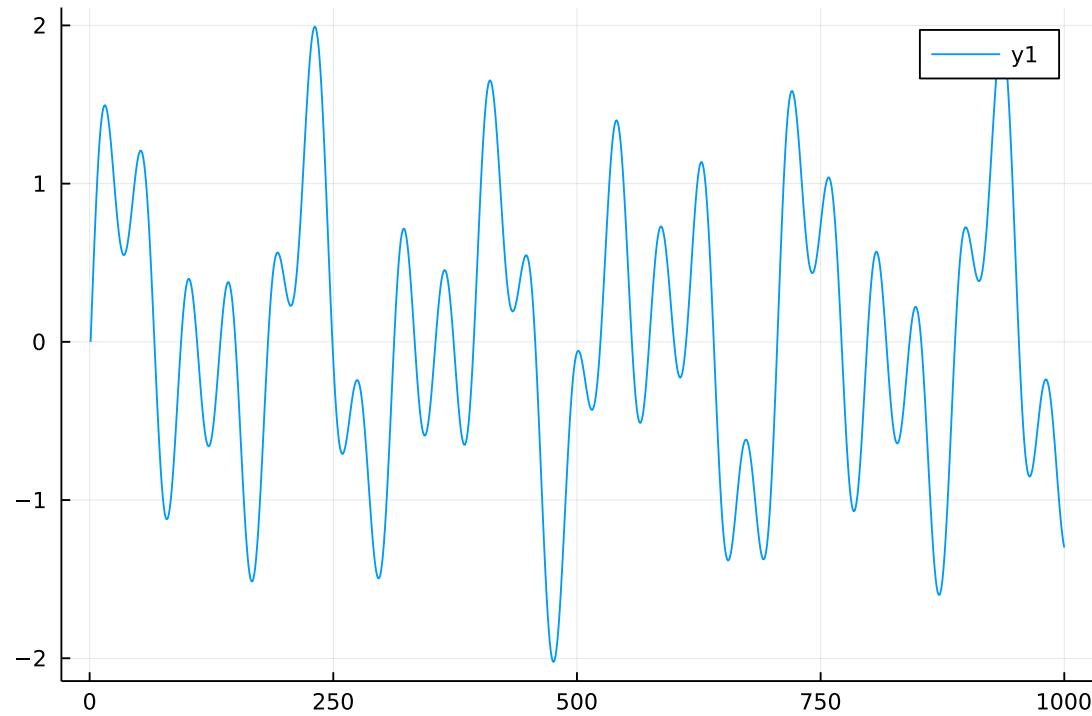
[201]:



17.4 Combination

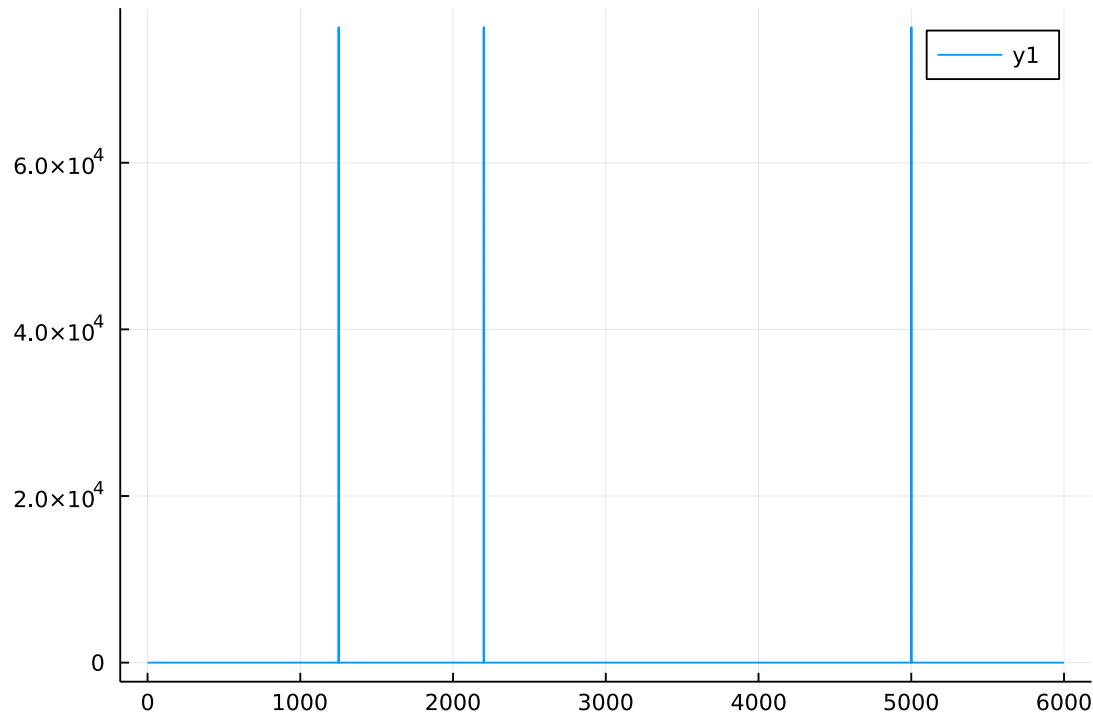
```
[202]: plot(sum(ys)[1:1000])
```

```
[202]:
```



```
[203]: plot(abs(.fft(sum(ys)))[1:6000])
```

```
[203]:
```



Note how the FFT graph matches the combination of the individual graph. We can very clearly see the peaks. On the other hand, the waveform graph appears a lot more complicated!

17.5 Chapter 3 HW

17.6 Section 3.3

17.6.1 32

```
[204]: A = [
    1 0;
    1 1;
    1 3;
    1 4;
]
b = [
    0;
    8;
    8;
    20
]
x = (A'*A)^-1 * A' * b
```

```
[204]: 2-element Array{Float64,1}:
0.9999999999999991
4.0000000000000001
```

```
[205]: p = A*x
```

```
[205]: 4-element Array{Float64,1}:
0.9999999999999991
5.0
13.000000000000004
17.000000000000004
```

```
[206]: error = b - A*x
```

```
[206]: 4-element Array{Float64,1}:
-0.9999999999999991
3.0
-5.0000000000000036
2.9999999999999964
```

```
[207]: E_squared = sum(error.^2)
```

```
[207]: 44.00000000000014
```

17.6.2 33

$$C = 0 \quad (7)$$

$$C + D = 8 \quad (8)$$

$$C + 3D = 8 \quad (9)$$

$$C + 4D = 20 \quad (10)$$

```
[208]: p = [
1;
5;
13;
17
]
x = A\p
```

```
[208]: 2-element Array{Float64,1}:
1.000000000000004
4.0
```

Note that there are rounding errors (in both q32 and q33). The exact answer is (1, 4).

17.6.3 34

```
[209]: e = [
    -1;
    3;
    -5;
    3
]

A' * e
```

[209]: 2-element Array{Int64,1}:

```
0
0
```

```
[210]: norm(e)
```

[210]: 6.6332495807108

$\therefore \|e\| = 6.6332495807108$

17.6.4 35

$$C = 0 \quad (11)$$

$$C + D + E = 8 \quad (12)$$

$$C + 3D + 9E = 8 \quad (13)$$

$$C + 4D + 16E = 20 \quad (14)$$

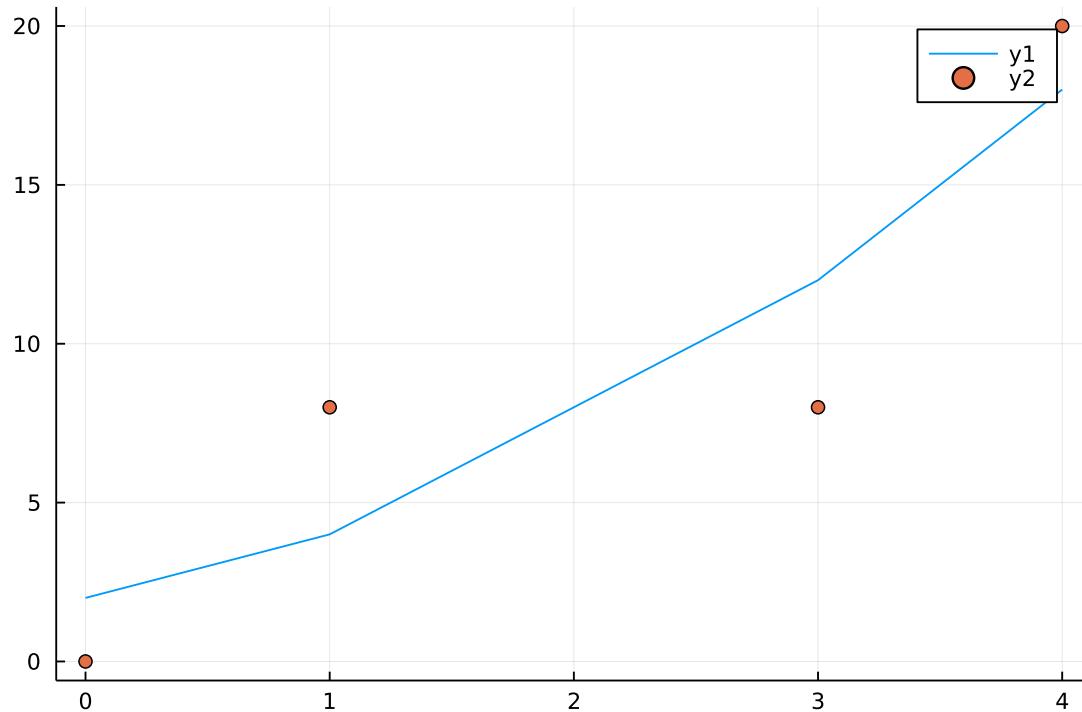
```
[211]: A = [
    1 0 0;
    1 1 1;
    1 3 9;
    1 4 16;
]
b = [
    0;
    8;
    8;
    20
]
x = (A'*A)^-1 * A' * b
```

[211]: 3-element Array{Float64,1}:

```
2.000000000000006
1.333333333333325
0.6666666666666712
```

```
[212]: xs = [0; 1; 3; 4]
p = plot(xs, A * x)
plot!(p, xs, b, seriestype = :scatter)
```

[212]:



We can see that Fig 3.9b is no longer a straight line but is starting to look like a parabola.

17.6.5 36

$$C = 0 \quad (15)$$

$$C + D + E + F = 8 \quad (16)$$

$$C + 3D + 9E + 27F = 8 \quad (17)$$

$$C + 4D + 16E + 64F = 20 \quad (18)$$

```
[213]: A = [
    1 0 0 0;
    1 1 1 1;
    1 3 9 27;
    1 4 16 64;
]

x = A^-1 * b
```

```
[213]: 4-element Array{Float64,1}:
```

```
0.0  
15.666666666666666  
-9.333333333333334  
1.666666666666667
```

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

```
[214]: 4×4 Array{Float64,2}:
```

```
1.0 0.0 0.0 0.0  
0.0 1.0 0.0 0.0  
0.0 0.0 1.0 0.0  
0.0 0.0 0.0 1.0
```

A is invertible. There exists a unique solution for the above system.

```
[215]: p = A*x
```

```
[215]: 4-element Array{Float64,1}:
```

```
0.0  
7.999999999999999  
8.0  
19.999999999999986
```

```
[216]: round.(p)
```

```
[216]: 4-element Array{Float64,1}:
```

```
0.0  
8.0  
8.0  
20.0
```

```
[217]: e = b - p
```

```
[217]: 4-element Array{Float64,1}:
```

```
0.0  
8.881784197001252e-16  
0.0  
1.4210854715202004e-14
```

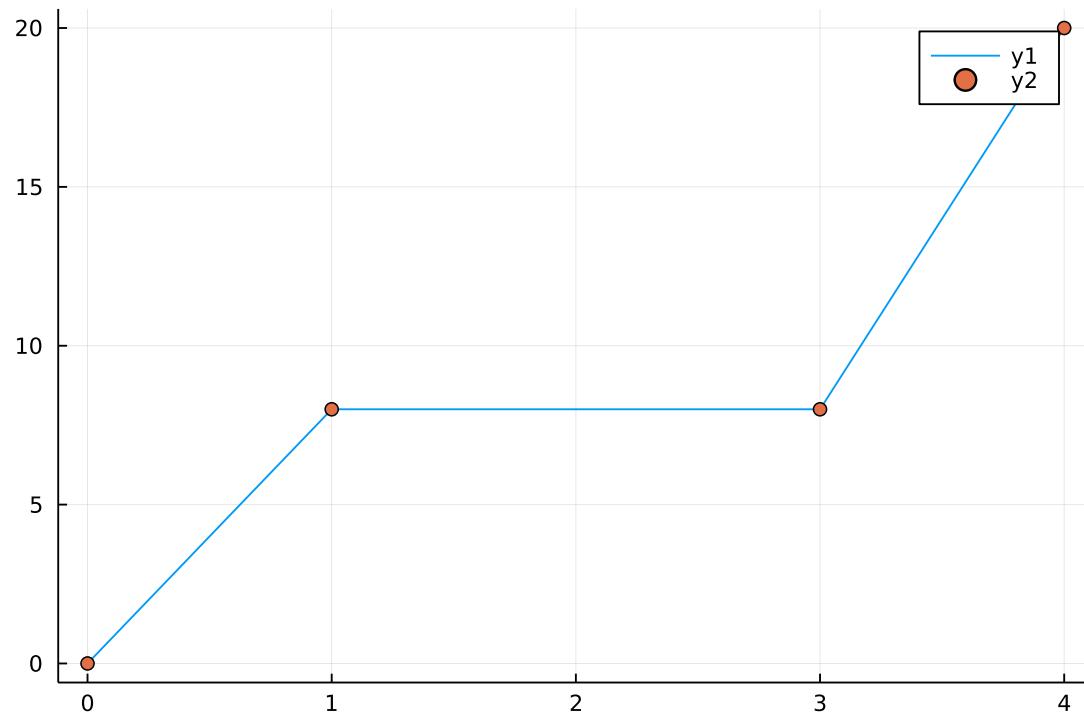
```
[218]: round.(e)
```

```
[218]: 4-element Array{Float64,1}:
```

```
0.0  
0.0  
0.0  
0.0
```

```
[219]: xs = [0;1;3;4]
pl = plot(xs, A*x)
plot!(pl, xs, b, seriestype=:scatter)
```

[219]:

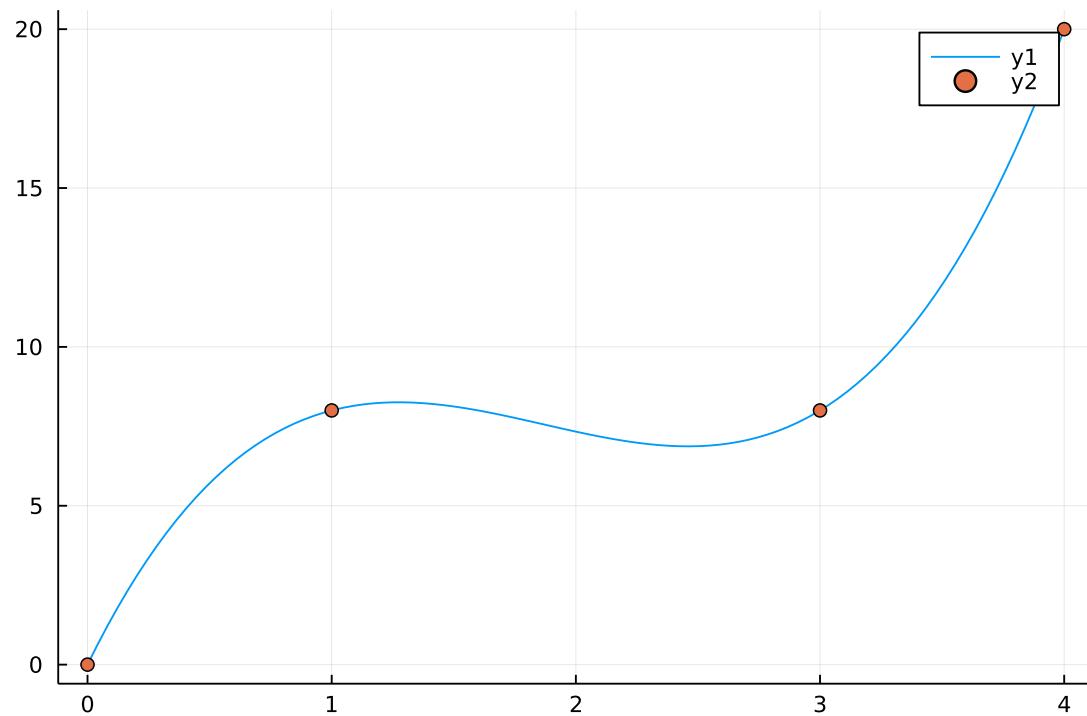


Here's a smooth version of the curve

```
[220]: function soln_poly(t)
    sum(x' * [1; t^1; t^2; t^3])
end

xs = range(0, 4, length=100)
pl = plot(xs, soln_poly.(xs))
plot!(pl, [0;1;3;4], b, seriestype=:scatter)
```

[220]:



17.6.6 37

Best line is obtained by setting $C = 1$ and $D = 4$

```
[221]: 1 + 4 * 2 == 9
```

```
[221]: true
```

```
[222]: A = [
    1 0;
    1 1;
    1 3;
    1 4;
]
b = [
    0;
    8;
    8;
    20
]

display(A'A)
display(A'b)
```

```

2×2 Array{Int64,2}:
4   8
8  26

2-element Array{Int64,1}:
36
112

```

The first equation is given by:

$$\left(\sum_i^m 1\right)C + \left(\sum_i^m t_i\right)D = \left(\sum_i^m b_i \cdot 1\right) \quad (19)$$

$$mC + \left(\sum_i^m t_i\right)D = \left(\sum_i^m b_i\right) \quad (20)$$

$$C + D\hat{t} = \hat{b} \quad (21)$$

(22)

18 Chapter 5 HW

18.1 Section 5.2

18.1.1 29

It approaches zero as $k \rightarrow \infty$ if and only if λ has absolute value less than 1.

```
[223]: A = [
    0.6 0.4;
    0.4 0.6
]

eigvals(A)
```

```
[223]: 2-element Array{Float64,1}:
0.1999999999999996
1.0
```

```
[224]: B = [
    0.6 0.9;
    0.1 0.6
]

eigvals(B)
```

```
[224]: 2-element Array{Float64,1}:
0.2999999999999993
0.8999999999999999
```

Only $B^k \rightarrow 0$

18.1.2 30

```
[225]: A = [  
    0.6 0.4;  
    0.4 0.6  
]  
  
vals, S = eigen(A)
```

```
[225]: Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}  
values:  
2-element Array{Float64,1}:  
0.1999999999999996  
1.0  
vectors:  
2×2 Array{Float64,2}:  
-0.707107  0.707107  
0.707107  0.707107
```

$$\Lambda = \begin{bmatrix} 0.19 & 0 \\ 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} -0.707107 & 0.707107 \\ 0.707107 & 0.707107 \end{bmatrix}$$

$$\Lambda^k \text{ as } k \rightarrow \infty = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\therefore S\Lambda^k S^{-1} \text{ as } k \rightarrow \infty = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

```
[226]: S * [0 0; 0 1] * S^-1
```

```
[226]: 2×2 Array{Float64,2}:  
0.5 0.5  
0.5 0.5
```

Each column of this matrix is an eigenvector for the eigenvalue $\lambda = 1$

```
[227]: A * [0.5;0.5] == [0.5;0.5]
```

```
[227]: true
```

18.1.3 31

```
[228]: B = [  
    0.6 0.9;  
    0.1 0.6  
]
```

```
vals, S = eigen(B)
```

```
[228]: Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
2-element Array{Float64,1}:
 0.2999999999999993
 0.899999999999999
vectors:
2×2 Array{Float64,2}:
 -0.948683  0.948683
 0.316228  0.316228
```

$$\Lambda = \begin{bmatrix} 0.299 & 0 \\ 0 & 0.899 \end{bmatrix}$$

$$S = \begin{bmatrix} -0.948683 & 0.948683 \\ 0.316228 & 0.316228 \end{bmatrix}$$

```
[229]: u0_a = [3;1]
u0_b = [3;-1]
u0_c = [6;0]
```

```
B10 = B^10
```

```
display(B10*u0_a)
display(B10*u0_b)
display(B10*u0_c)
```

```
2-element Array{Float64,1}:
```

```
 1.0460353203000001
 0.3486784401
```

```
2-element Array{Float64,1}:
```

```
 1.771469999989339e-5
 -5.904900000019975e-6
```

```
2-element Array{Float64,1}:
```

```
 1.046053035
 0.34867253519999997
```

Note how these values are approaching zero.

```
[230]: B^10
```

```
[230]: 2×2 Array{Float64,2}:
 0.174342  0.523009
 0.0581121 0.174342
```

```
[231]: B^100
```

```
[231]: 2×2 Array{Float64,2}:
 1.32807e-5  3.98421e-5
 4.4269e-6   1.32807e-5
```

```
[232]: B^1000
```

```
[232]: 2×2 Array{Float64,2}:
 8.73936e-47  2.62181e-46
 2.91312e-47  8.73936e-47
```

```
[233]: round.(B^1000)
```

```
[233]: 2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

18.1.4 45

```
[234]: A = [
    0.6 0.2;
    0.4 0.8
]

vals, S = eigen(A)
```

```
[234]: Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
2-element Array{Float64,1}:
 0.3999999999999997
 1.0
vectors:
2×2 Array{Float64,2}:
 -0.707107  -0.447214
 0.707107  -0.894427
```

Let's find A^k as $k \rightarrow \infty$ as well

```
[235]: S * [0 0; 0 1] * S^-1
```

```
[235]: 2×2 Array{Float64,2}:
 0.333333  0.333333
 0.666667  0.666667
```

This is A^∞ . Therefore, A^∞ is the steady-state matrix of the Markov chain.

```
[236]: A_infty = [
    1//3 1//3;
    2//3 2//3
]
```

```
vals, S = eigen(A_infny)
```

```
[236]: Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
2-element Array{Float64,1}:
 0.0
 1.0
vectors:
2×2 Array{Float64,2}:
 -0.707107  -0.447214
  0.707107  -0.894427
```

We can read the eigenvalues and eigenvectors off the results

```
[237]: A^100
```

```
[237]: 2×2 Array{Float64,2}:
 0.333333  0.333333
 0.666667  0.666667
```

A^∞ is the steady-state matrix for the Markov Chain.

Every Markov Chain has a transition matrix P and the distribution of probabilities at time t given an initial state μ_0 is given by the following equation:

$$\mu_t = \mu_0 P^t$$

We can compare A^{100} and A^∞ by looking at their eigen decompositions.

```
[238]: A100 = A^100
```

```
vals, S = eigen(A100)
```

```
[238]: Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}
values:
2-element Array{Float64,1}:
 0.0
 1.0000000000000089
vectors:
2×2 Array{Float64,2}:
 -0.707107  -0.447214
  0.707107  -0.894427
```

The eigenvalues and eigenvectors are nearly identical (ignoring rounding errors). This result makes sense because we are expecting A to ultimately converge to A^∞

18.2 Section 5.3

18.2.1 3

[239]: A = [1 1; 1 0]

```
display(A^2)
display(A^3)
display(A^4)
```

2×2 Array{Int64,2}:

```
2 1
1 1
```

2×2 Array{Int64,2}:

```
3 2
2 1
```

2×2 Array{Int64,2}:

```
5 3
3 2
```

[240]: A^20

[240]: 2×2 Array{Int64,2}:

```
10946 6765
6765 4181
```

[241]: A^20 * [1;0]

[241]: 2-element Array{Int64,1}:

```
10946
6765
```

.: $F_{20} = 6765$

Let's verify this by writing a classical fibonacci sequence function.

[242]:

```
function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

[242]: fib (generic function with 1 method)

[243]: fib(20)

[243]: 6765

The structure of A^k is interesting.

$$A^k = \begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix}$$

7

[244]: A = [1 1; 1 0]
res = A^10 * [1;2]

[244]: 2-element Array{Int64,1}:

199
123

[245]: exact = res[2]

[245]: 123

[246]: l1 = 0.5 * (1 + √5)
approx = l1^10

[246]: 122.99186938124426

[247]: abs(exact-approx)

[247]: 0.008130618755743058

[248]: l2 = 0.5 * (1 - √5)
l1^10 + l2^10

[248]: 123.00000000000004

18.2.2 16

[249]: A = [
0 -1;
1 0;
]

[249]: 2×2 Array{Int64,2}:
0 -1
1 0

[250]: # (F)
eigvals(I+A)

```
[250]: 2-element Array{Complex{Float64},1}:
 1.0 - 1.0im
 1.0 + 1.0im
```

```
[251]: # (B)
eigvals((I-A)^-1)
```

```
[251]: 2-element Array{Complex{Float64},1}:
 0.5 - 0.5000000000000001im
 0.5 + 0.5000000000000001im
```

```
[252]: # (C)
eigvals((I-0.5*A)^-1 * (I+0.5*A)^-1)
```

```
[252]: 2-element Array{Float64,1}:
 0.8000000000000002
 0.8000000000000002
```

```
[253]: # TODO figure out which solution stays on the circle
```

18.2.3 29

```
[254]: A = [
 3 2;
 1 4
]

B = [
 3 2;
 -5 -3;
]

C = [
 5 7;
 -3 -4;
]

D = [
 5 6.9;
 -3 -4
]
```

```
[254]: 2x2 Array{Float64,2}:
 5.0   6.9
 -3.0  -4.0
```

```
[255]: opnorm(A^1024)
```

```
[255]: 1.2066571869539525e19
```

I'm not getting something $> 10^{700}$. The opnorm function computes the matrix norm. I wonder if this is a typesetting error in the textbook

```
[256]: B^1024
```

```
[256]: 2×2 Array{Int64,2}:
```

```
 1  0  
 0  1
```

```
[257]: C^1024
```

```
[257]: 2×2 Array{Int64,2}:
```

```
-5  -7  
 3   4
```

```
[258]: opnorm(D^1024)
```

```
[258]: 4.656336537257555e-79
```

```
[259]: round.(eigvals(B))
```

```
[259]: 2-element Array{Complex{Float64},1}:
```

```
0.0 - 1.0im  
0.0 + 1.0im
```

The eigenvalues are $e^{\frac{3\pi i}{2}} = -i$ and $e^{\frac{\pi i}{2}} = i$. This means that $\lambda^4 = 1$

```
[260]: vals, S = eigen(B)
```

```
[260]: Eigen{Complex{Float64},Complex{Float64},Array{Complex{Float64},2},Array{Complex{Float64},1}}
```

values:

2-element Array{Complex{Float64},1}:

```
2.42861286636753e-16 - 1.0000000000000004im  
2.42861286636753e-16 + 1.0000000000000004im
```

vectors:

2×2 Array{Complex{Float64},2}:

```
-0.507093+0.169031im -0.507093-0.169031im  
0.845154-0.0im      0.845154+0.0im
```

```
[261]: B^4
```

```
[261]: 2×2 Array{Int64,2}:
```

```
 1  0  
 0  1
```

```
[262]: (S * [-im 0; 0 im]^4 * S^-1) == I
```

```
[262]: true
```

```
[263]: eigvals(C)
```

```
[263]: 2-element Array{Complex{Float64},1}:
 0.5000000000000002 - 0.8660254037844388im
 0.5000000000000002 + 0.8660254037844388im
```

The eigenvalues of are $e^{\frac{5\pi i}{3}} = 0.5 - 0.866i$ and $e^{\frac{\pi i}{3}} = 0.5 + 0.866i$. This means that $\lambda^3 = -1$

```
[264]: round.((0.5000000000000002 - 0.8660254037844388im)^3)
```

```
[264]: -1.0 - 0.0im
```

```
[265]: round.((0.5000000000000002 + 0.8660254037844388im)^3)
```

```
[265]: -1.0 + 0.0im
```

```
[266]: C^3
```

```
[266]: 2x2 Array{Int64,2}:
```

```
-1 0
 0 -1
```

```
[267]: vals, S = eigen(C)
```

```
lambda = [
 0.5000000000000002 - 0.8660254037844388im 0;
 0 0.5000000000000002 + 0.8660254037844388im;
]
round.((S * lambda^3 * S^-1)) == -I
```

```
[267]: true
```

18.3 Section 5.4

18.3.1 38

First let's try out the matrices provided in the question

```
[268]: A = [
 1 1;
 0 0;
]

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

```
[268]: 2×2 Array{Int64,2}:
```

```
0  -1  
0   0
```

```
[269]: eab = exp(A) * exp(B)
```

```
[269]: 2×2 Array{Float64,2}:
```

```
2.71828  -1.0  
0.0       1.0
```

```
[270]: eba = exp(B) * exp(A)
```

```
[270]: 2×2 Array{Float64,2}:
```

```
2.71828  0.718282  
0.0       1.0
```

```
[271]: eapb = exp(A + B)
```

```
[271]: 2×2 Array{Float64,2}:
```

```
2.71828  0.0  
0.0       1.0
```

```
[272]: display(eab == eba)  
display(eab == eapb)  
display(eba == eapb)
```

```
false
```

```
false
```

```
false
```

18.3.2 36

```
[273]: A = [  
           1 1;  
           0 0;  
         ]  
  
vals, S = eigen(A)  
  
lambda = [  
           vals[1] 0;  
           0 vals[2]  
         ]  
  
A == S * lambda * S^-1
```

```
[273]: true
```

During our previous discussion on the matrix exponential, we know that computing it for a diagonal matrix is easy: exponentiate each term on the diagonal.

For $t = 1$,

[274]: `exp(lambda)`

[274]: 2×2 Array{Float64,2}:

1.0	0.0
0.0	2.71828

[275]: `exp(A) * S * exp(lambda) * S^-1`

[275]: `true`

[276]: `S * exp(lambda) * S^-1`

[276]: 2×2 Array{Float64,2}:

2.71828	1.71828
0.0	1.0

18.3.3 37

$$e^{At} = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots \quad (23)$$

$$= I + A + \frac{A}{2!} + \frac{A^2}{3!} + \dots \quad (24)$$

$$= I + A + \frac{A}{2!} + \frac{A}{3!} + \dots \quad (25)$$

$$= I + A(1 + \frac{1}{2!} + \frac{1}{3!} \dots) \quad (26)$$

$$= I + (e^t - 1)A \quad (27)$$

For $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$,

$$e^{At} = I + (e^t - 1)A \quad (28)$$

$$= I + \begin{bmatrix} e^t - 1 & e^t - 1 \\ 0 & 0 \end{bmatrix} \quad (29)$$

$$= \begin{bmatrix} e^t & e^t - 1 \\ 0 & 1 \end{bmatrix} \quad (30)$$

$$(31)$$

[277]: `# for t = 1`

[

`exp(1) (exp(1) - 1);`

```
0 1  
]
```

```
[277]: 2×2 Array{Float64,2}:  
2.71828 1.71828  
0.0 1.0
```

18.4 Sidenote: When does $e^x e^y = e^y e^x = e^{x+y}$?

I looked through the Wikipedia entry for Matrix exponential. If A and B commute, then the formula must hold!

$$e^A e^B = e^B e^A = e^{A+B}$$

Two matrices are said to commute if $AB = BA$

```
[278]: # example
```

```
A = [  
    2 0;  
    1 -1;  
]  
  
B = [  
    3 0;  
    1 0;  
]  
  
A*B == B*A
```

```
[278]: true
```

```
[279]: exp(A)*exp(B)  exp(B)*exp(A)  exp(A+B)
```

```
[279]: true
```

TODOS - [] Implementing quantum algorithms with Yao.jl - [] SVD application 2 - [] SVD application 3 - [] Chapter 2 problems - [] Resolve TODOS - [] Complete suggested exercises from the document

19 Minima, Maxima, and Saddle Points

How can we determine if a function f has a local minima, local maxima, or saddle point?

For the one-dimensional case, we first find a stationary point x (i.e., a point where $f'(x) = 0$) and then check the sign of $f''(x)$.

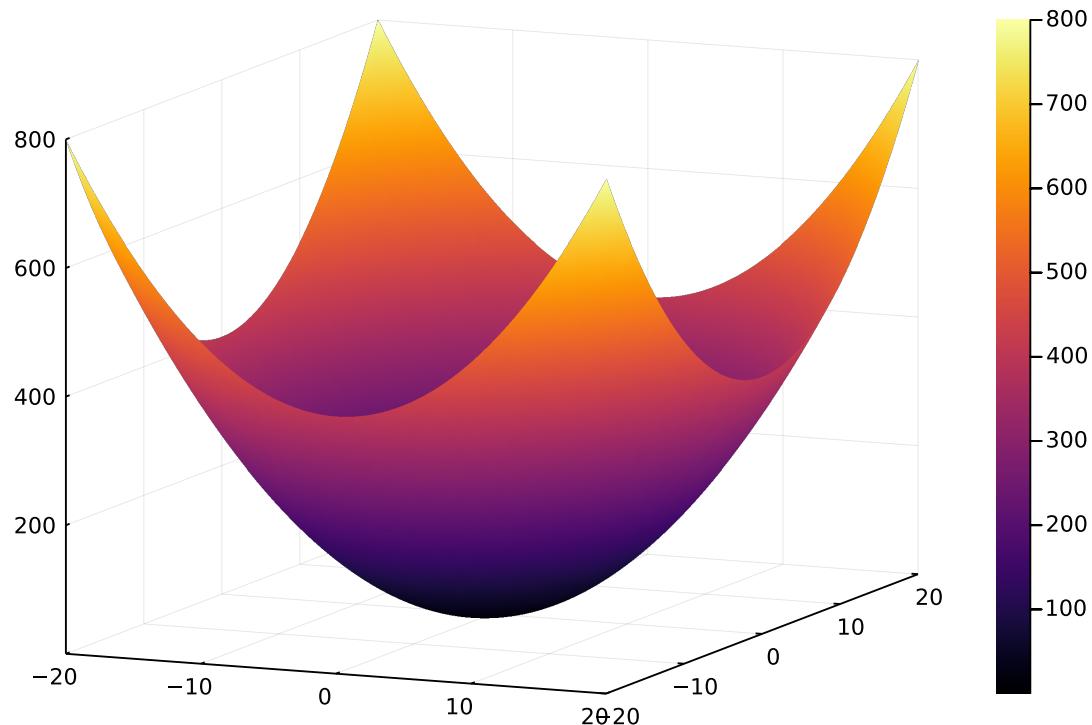
The question is how do we generalize this test to n dimensions?

```
[280]: # f(x,y) = x^2 + y^2

# Positive semi-definite matrix
# Surface shaped like a bowl

n = 100
ts = range(-20, stop = 20, length = n)
x = ts
y = ts
f(x,y) = x^2 + y^2
surface(x, y, f)
```

[280]:

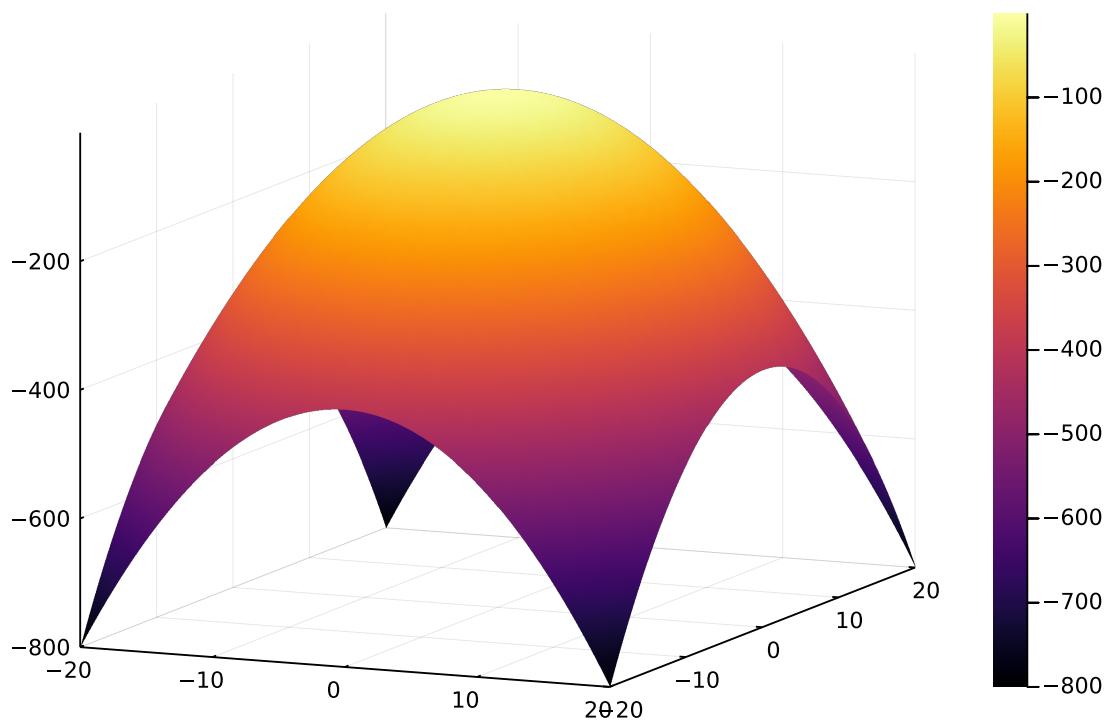


```
[281]: # f(x,y) = -(x^2 + y^2)

# Negative semi-definite matrix
# Surface shaped like an inverted bowl

n = 100
ts = range(-20, stop = 20, length = n)
x = ts
y = ts
f(x,y) = -(x^2 + y^2)
surface(x, y, f)
```

[281] :

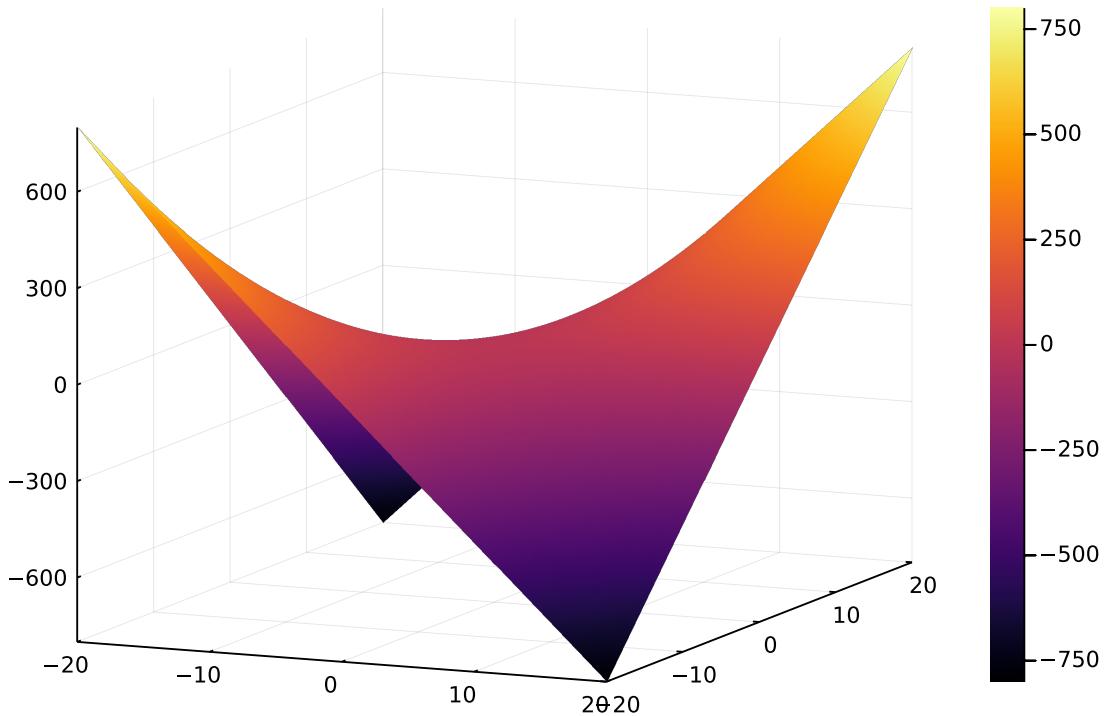


[282] : # $f(x, y) = 2xy$

```
# Indefinite matrix
# Surface shaped like a saddle

n = 100
ts = range(-20, stop = 20, length = n)
x = ts
y = ts
f(x,y) = 2*x*y
surface(x, y, f)
```

[282] :



- Notes about which order terms matter
- Introduce the quadratic
- Write it as a matrix
- Test for positive definiteness
- Ellipse
- Law of inertia
- Examples

Let F be a function of n variables.

The Taylor series is given by

$$F(x) = F(0) + F'(x) + \frac{1}{2}F''(x) + \text{higher order terms}$$

The higher order terms have no effect on the question of a local minima or maxima. The first order derivative is used to obtain the stationary point (by setting its value to zero) The second order derivative (which can be expressed in quadratic form) is used to determine definiteness.

The quadratic form can be written as a matrix A where $a_{ij} = \frac{\partial^2 F}{\partial x_i \partial x_j}$. This is the second derivative matrix (also known as the Hessian).

If x is a vector, the $x^T A x$ will give back the quadratic equation.

We have encapsulated the information we get from the second derivative in the form of a matrix A .

Now we have reduced the problem to relating properties of a matrix to definiteness.

The following tests provide a sufficient and necessary condition to determine if $x^T A x$ is positive definite:

- The quadratic form $x^T A x > 0$ for all non-zero vectors x
- All eigenvalues λ_i of A are positive
- All pivots (without row exchanges) d_k of A are positive
- All upper left submatrices A_k have positive determinants
- There is a matrix R with independent columns such that $A = R^T R$

Julia has a method `isposdef` to test for positive definiteness. Let's try it out!

```
[289]: isposdef([
    0 0;
    0 1
])
```

[289]: false

```
[290]: isposdef([
    3 3 0;
    3 10 7;
    0 7 8
])
```

[290]: true

The last test is interesting. Note that the R does not need to be unique. There are several ways to find such an R . One technique is to try to perform a Cholesky decomposition.

The Cholesky decomposition is similar to the LDU decomposition except that L and U are now transposes of each other.

If we extract the diagonal of U out into a matrix D , we get the following:

$$A = LDU = LDL^T = (\sqrt{D}L^T)^T(\sqrt{D}L^T) = R^T R$$

```
[291]: cholesky([
    3 3 0;
    3 10 7;
    0 7 8
])
```

```
[291]: Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 1.73205  1.73205  0.0
 2.64575  2.64575
           1.0
```

```
[292]: cholesky([
    0 0;
    0 1
])
```

```

PosDefException: matrix is not positive definite; Cholesky factorization failed

Stacktrace:
[1] checkpositivedefinite at /Users/julia/buildbot/worker/package_macos64/build/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 times)
[6] top-level scope at In[292]:1
[7] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

```

Cholesky decomposition is possible only for positive definite matrices. Julia raises a `PosDefException` to indicate that the matrix is not positive definite.

20 Singular Value Decomposition

For any $m \times n$ matrix A of rank r , the singular value decomposition of A is given by

$$A = U\Sigma V^T$$

where U is an $m \times m$ orthogonal matrix, V is an $n \times n$ orthogonal matrix, and Σ is an $m \times n$ matrix containing the r singular values of A along the diagonal and zero elsewhere. V contains the eigenvectors of $A^T A$ and U contains the eigenvectors of AA^T .

By convention, we arrange the singular values in descending order along the diagonal of Σ

$$\sigma_1 \geq \sigma_2 \geq \dots \sigma_r$$

The singular values σ_i are positive and non-zero. Combining this with our convention, we get $\sigma_{i+r} = 0$ for $i \geq 1$

- $AV = \Sigma U$
- For $1 \leq i \leq r$, $Av_i = \sigma_i u_i \implies u_i = \frac{Av_i}{\sigma_i}$
- U and V give the orthonormal bases for each of the four fundamental bases
 - First r columns of U give the column space of A
 - Last $m - r$ columns of U give the left nullspace of A
 - First r columns of V give the row space of A
 - Last $n - r$ columns of V give the nullspace of A

Note that each of the u_i s obtained from v_i are pairwise orthonormal.

For $i \neq j$ and $1 \leq i \leq r, 1 \leq j \leq r$,

$$u_i u_j = \left(\frac{Av_i}{\sigma_i} \right)^T \left(\frac{Av_j}{\sigma_j} \right) \quad (32)$$

$$= \left(\frac{v_i^T A^T A v_j}{\sigma_i \sigma_j} \right) \quad (33)$$

$$= \left(\frac{v_i^T \sigma_j^2 v_j}{\sigma_i \sigma_j} \right) \quad (34)$$

$$= 0 \quad (35)$$

This decomposition has a lot of interesting applications! Before we get there, let's explore the `svd` method in Julia.

[283]: # rank 1 matrix

```
A = [-1 2 2]'

F_a = svd(A)
```

[283]: SVD{Float64,Float64,Adjoint{Float64,Array{Float64,2}}}
U factor:
3×1 Adjoint{Float64,Array{Float64,2}}:
-0.3333333333333326
0.6666666666666666
0.6666666666666666
singular values:
1-element Array{Float64,1}:
3.0
Vt factor:
1×1 Adjoint{Float64,Array{Float64,2}}:
1.0

[284]: # rank 2

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

F_b = svd(B)
```

[284]: SVD{Float64,Float64,Array{Float64,2}}
U factor:
2×2 Array{Float64,2}:
-0.707107 0.707107
0.707107 0.707107
singular values:

```

2-element Array{Float64,1}:
 1.7320508075688772
 1.0
Vt factor:
2×3 Array{Float64,2}:
 0.408248 -0.816497  0.408248
 -0.707107 -2.78976e-16 0.707107

```

The first interesting thing to note from the output is the dimensions of the output matrices. In the definition, we said that U , Σ , and V^T are $m \times m$, $m \times n$, and $n \times n$ matrices respectively.

For A , the U matrix is 3×1 (instead of 3×3). Similarly, for B , the V^T matrix is 2×3 (instead of 3×3).

Let's first check if the factorizations are correct

```
[285]: A = F_a.U * diagm(F_a.S) * F_a.Vt
```

```
[285]: true
```

```
[286]: B = F_b.U * diagm(F_b.S) * F_b.Vt
```

```
[286]: true
```

The factorizations are correct! So why are the dimensions different?

It turns out that to optimize for space, implementations of SVD use a reduced form in which only the most essential information is stored.

The SVD factorization can be written as:

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_r u_r v_r^T + \sigma_{r+1} u_{r+1} v_{r+1}^T \dots$$

But, $\sigma_{i+r} = 0$, for $i \geq 1$. Therefore, we can rewrite A as

$$A = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_r u_r v_r^T$$

In other words, for the above equation, U is $m \times r$, Σ is $r \times r$, and V^T is $r \times n$. We do not need to store the rest of the columns of U and V^T because in the factorization they multiply with a zero and thus cancel out.

Now the dimensions make sense!

One disadvantage of using the reduced format is that U and V are no longer square orthogonal matrices. This means we cannot use $UU^T = U^TU = I$ and $VV^T = V^TV = I$ in calculations.

```
[287]: # Fibonacci matrix
```

```

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

```
svd(A)
```

```
[287]: SVD{Float64,Float64,Array{Float64,2}}
U factor:
2×2 Array{Float64,2}:
-0.850651 -0.525731
-0.525731  0.850651
singular values:
2-element Array{Float64,1}:
 1.618033988749895
 0.6180339887498948
Vt factor:
2×2 Array{Float64,2}:
-0.850651 -0.525731
 0.525731 -0.850651
```

```
[288]: # Zero matrix

A = zeros(3,4)

svd(A)
```

```
[288]: SVD{Float64,Float64,Array{Float64,2}}
U factor:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
singular values:
3-element Array{Float64,1}:
 0.0
 0.0
 0.0
Vt factor:
3×4 Array{Float64,2}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
```

In general, for the $m \times n$ zero matrix, the SVD will be $U = I_m$, $V = I_n$, and $\Sigma = \mathbf{0}_{m \times n}$

21 Visualizing positive definite matrices as ellipsoids

Let A be an $m \times n$ real symmetric positive definite matrix. Then, there exists the spectral decomposition such that $A = Q\Lambda Q^T$

$$x^T A x = x^T Q \Lambda Q^T x = y^T \Lambda y = \lambda_1 y_1^2 + \lambda_2 y_2^2 + \dots + \lambda_n y_n^2$$

This is the equation of an ellipsoid with axis lengths $\frac{1}{\sqrt{\lambda_1}}, \frac{1}{\sqrt{\lambda_2}}, \dots, \frac{1}{\sqrt{\lambda_n}}$

```
[496]: function ellipsoid(rx, ry, rz)
    ngrid = 100
    u = range(0, 2, length=ngrid)
    v = range(0, , length=ngrid)

    x = [rx * x * y for (x, y) in Iterators.product(cos.(u), sin.(v))]
    y = [ry * x * y for (x, y) in Iterators.product(sin.(u), sin.(v))]
    z = [rz * x * y for (x, y) in Iterators.product(ones(length(u)), cos.(v))]

    plot(vec(x), vec(y), vec(z))
end

function ellipse(rx, ry)
    n = 100
    ts = range(0, 4, length = n)

    xs = rx * cos.(ts)
    ys = ry * sin.(ts)

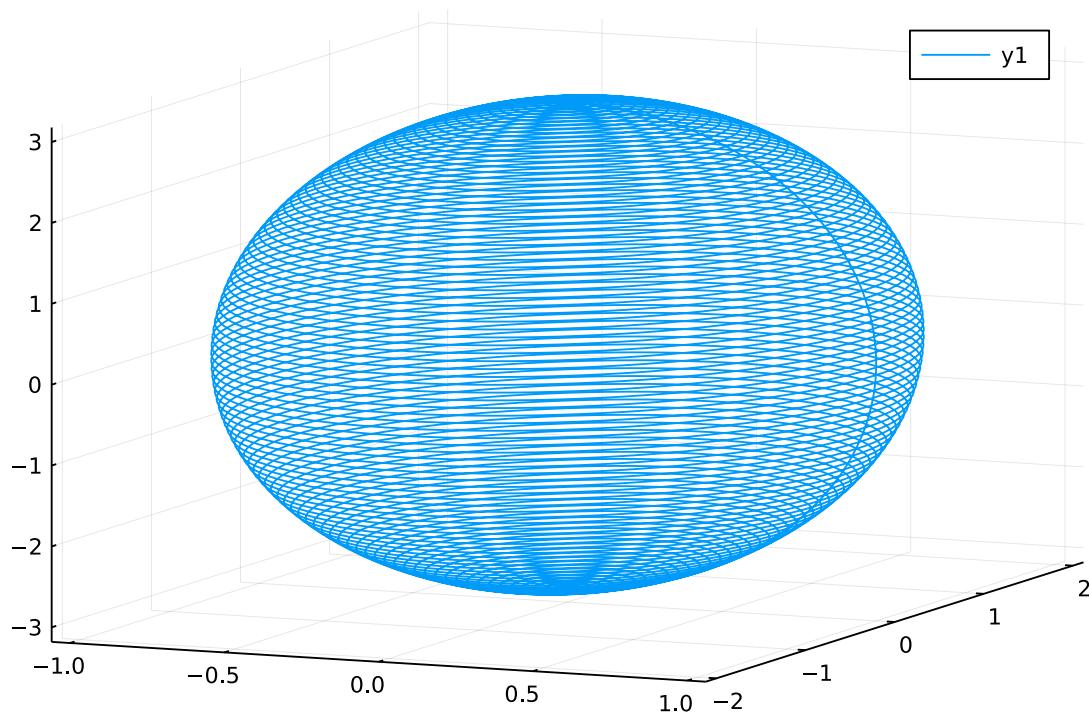
    plot(xs, ys)
end
```

[496]: ellipse (generic function with 1 method)

```
[497]: A = [
    2 -1 0;
    -1 2 0;
    0 -1 2
]

vals = eigvals(A)
ellipsoid(vals...)
```

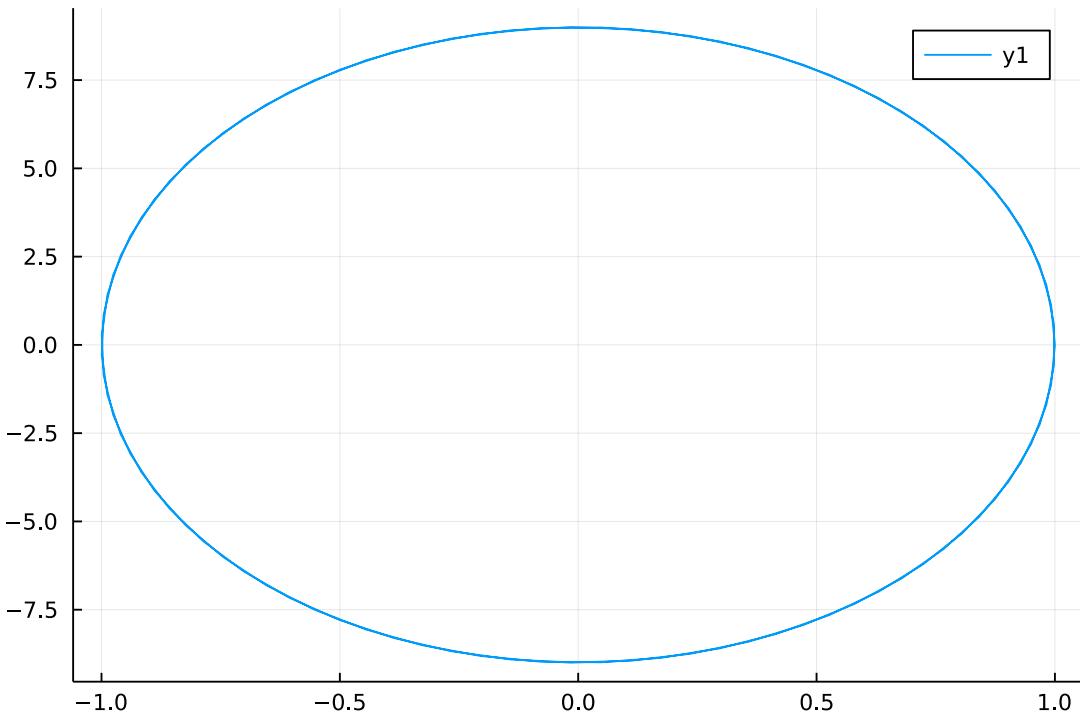
[497]:



```
[472]: A = [
    5 4;
    4 5
]

vals = eigvals(A)
ellipse(vals...)
```

[472]:



22 Applications of SVD

22.1 Image Compression

An image is nothing but an $m \times n$ matrix of real values. If we were to store the image as is, it would result in a space complexity of $O(mn)$. However, we don't need to store all of the information to maintain the fidelity of the image. This is where compression comes in to play.

One simple way to compress an image is to use the SVD and only use the first k singular values (and zero out the rest). This can save space as the compressed size $\approx m \times k + k \times n$.

[498]: `using Images`

```
Info: Precompiling Images [916415d5-f1e6-5110-898d-aaa5f9f070e0]
@ Base loading.jl:1278
```

[540]: `function compress(img, precision)
 F = svd(img)
 F.S[precision:length(F.S)] .= 0
 return F.U * diagm(F.S) * F.Vt
end`

[540]: `compress (generic function with 1 method)`

22.1.1 Grayscale image

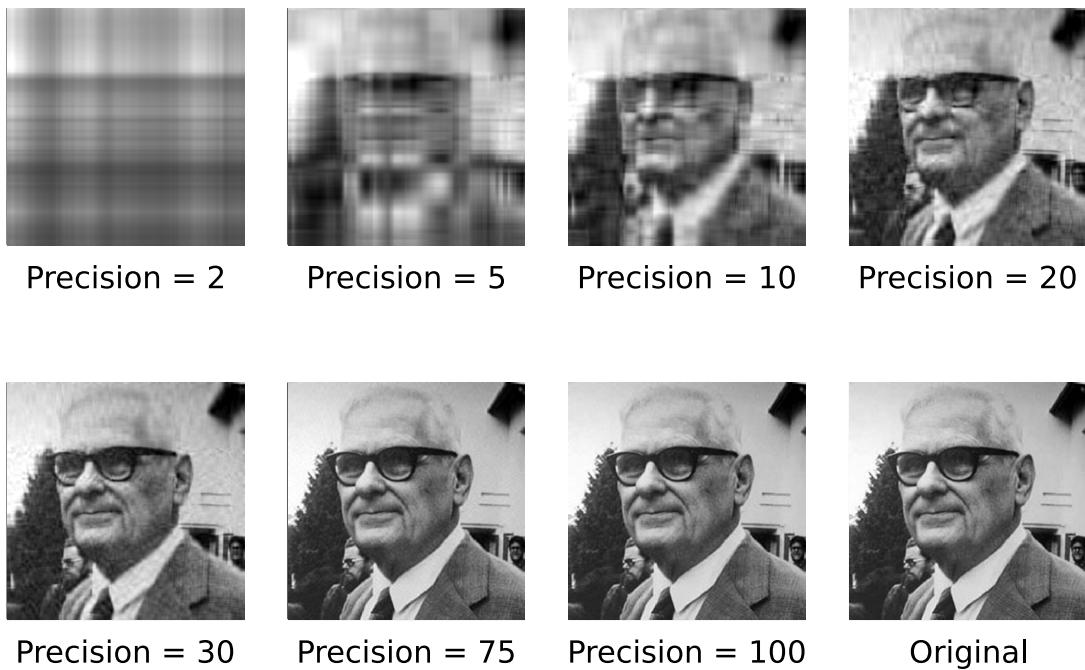
```
[549]: precision_ranges = [2, 5, 10, 20, 30, 75, 100]
```

```
[549]: 7-element Array{Int64,1}:
```

```
 2  
 5  
10  
20  
30  
75  
100
```

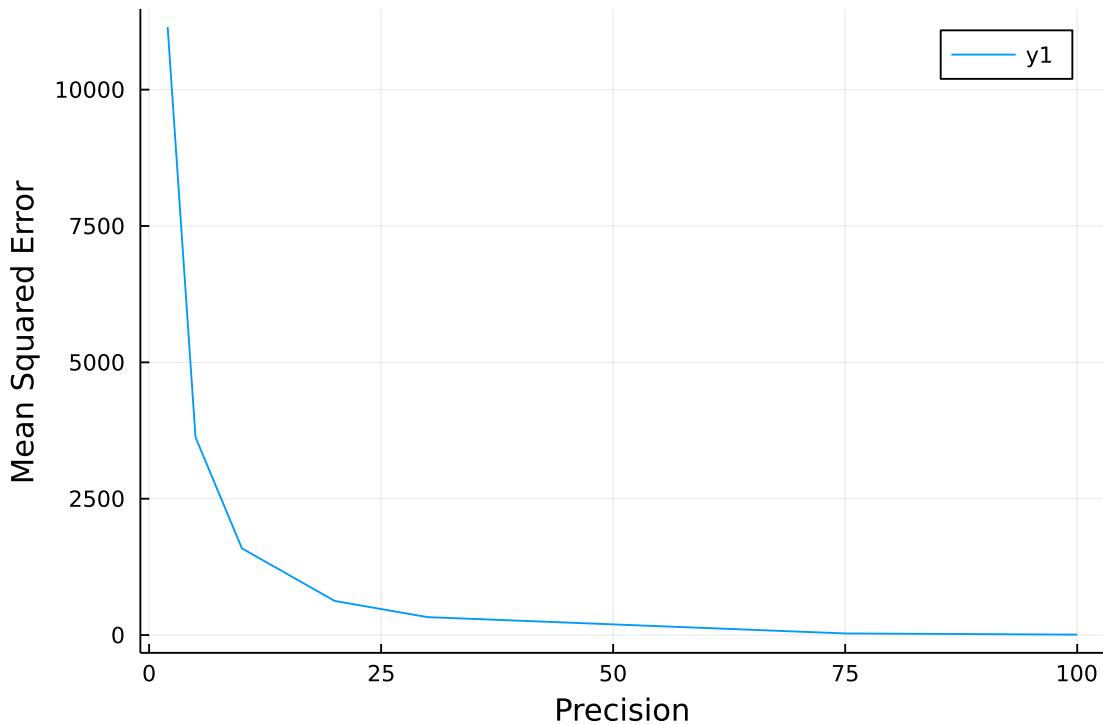
```
[565]: img = Gray.(load("assets/alonzo_church.jpeg"))  
plots = []  
errors = []  
  
for p in precision_ranges  
    compressed_image = Gray.(compress(img, p))  
    push!(errors, sum((img .- compressed_image).^2))  
    push!(plots, plot(  
        compressed_image,  
        xlabel="Precision = " * string(p),  
        xticks=false,  
        yticks=false))  
end  
  
push!(plots, plot(img, xlabel="Original", xticks=false, yticks=false))  
  
plot(plots..., layout=(2,4))
```

```
[565]:
```



```
[568]: plot(precision_ranges, Float32.(errors), xlabel="Precision", ylabel="Mean Squared Error", st=:line)
```

```
[568]:
```



22.2 Principal Component Analysis

22.3 Polar Decomposition

22.4 Least squares and pseudoinverse

22.5 Facial Recognition