Solution to Assignment 2

MATH7502 2019, Semester 2

Assignment 2 questions (https://courses.smp.uq.edu.au/MATH7502/2019/ass2.pdf)

Solution to Question 1

We have,

$$x = \sum_{i=1}^{k} \beta_i a_i$$

with a_i orthonomral vectors.

$$||x||^2 = ||\beta_1 a_1 + \dots \beta_k a_k||^2 = (\beta_1 a_1 + \dots \beta_k a_k)^T(\beta_1 a_1 + \dots \beta_k a_k) = \beta_1^2 + \dots + \beta_k^2 = ||\beta||^2.$$

Hence $||x|| = ||\beta||$.

Solution to Question 2

We have

$$a_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \qquad a_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \qquad \dots \qquad a_n = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix},$$

(a) Running Gram-Schmidt

$$q_1 = \tilde{q}_1 = a_1 = e_1$$

$$q_2 = \tilde{q}_2 = a_2 - (q_1^T a_2) q_1 = e_2$$

$$q_3 = \tilde{q}_3 = a_3 - (q_1^T a_3) q_1 - (q_2^T a_3) q_2 = e_3$$

$$\vdots$$

$$q_n = e_n$$

(b) Yes, $a_1, \dots, a_n$ is a basis for $\mathbb{R}^n$. Because G-S didn't terminate, we found that the vectors are linearly independent and there are n of them.

(c) A G-S implementation

In [32]:
```
using LinearAlgebra

function gs(a)
    k, n = length(a), length(a[1])
    q = fill(zeros(n),k) #an array of vectors
    q[1] = a[1]/norm(a[1])
    for i in 2:k
        qT = a[i]-sum([(q[j]'*a[i])*q[j] for j in 1:i-1])
        if norm(qT) < 10^-8
            println("Dependent vectors after $(i) iterations")
            return
        end
        q[i] = qT/norm(qT)
    end
    q
end
```

Out[32]:
```
gs (generic function with 1 method)
```

Test it on some basic inputs...

In [33]:
```
q = gs([[1,1],[1,0]])
```

Out[33]:
```
2-element Array{Array{Float64,1},1}:
 [0.707107, 0.707107]
 [0.707107, -0.707107]
```

In [34]:
```
norm.(q)
```

Out[34]:
```
2-element Array{Float64,1}:
 0.9999999999999999
 1.0
```

In [35]:
```
q[1]'*q[2]
```

Out[35]:
```
2.7755575615628914e-16
```

In [44]:
```
q = gs([rand(20) for _ in 1:15]);
norm.(q)
```

Out[44]:
```
15-element Array{Float64,1}:
 0.9999999999999999
 1.0
 0.9999999999999999
 0.9999999999999999
 1.0
 1.0
 0.9999999999999999
 0.9999999999999999
 1.0000000000000002
 0.9999999999999999
 0.9999999999999999
 1.0
 0.9999999999999999
 0.9999999999999999
 1.0
```

In [45]:
```
Q = hcat(q...)
Q'*Q
```

Out[45]:
```
15×15 Array{Float64,2}:
  1.0           1.17229e-16   2.87541e-16  …   6.12258e-16   1.80411e-16
  1.17229e-16   1.0          -2.5431e-16       -2.95635e-16   5.96745e-16
  2.87541e-16  -2.5431e-16    1.0              -1.47594e-16  -6.245e-17
  7.13624e-17  -1.03405e-16  -6.06927e-17      -3.29407e-16   4.44089e-16
 -2.91614e-16  -3.53699e-16  -3.64357e-16      -4.12128e-16  -4.78784e-15
 -1.5689e-17   -1.24718e-16   4.85177e-17  …   9.88694e-16   1.72085e-15
  2.70461e-17   2.09903e-16   1.70043e-16       2.62886e-16   4.23273e-15
 -9.19821e-17   4.30837e-16  -1.07332e-16       1.43942e-15   5.02376e-15
  4.29366e-16   2.1304e-16   -1.17614e-16      -2.53108e-16   9.85323e-16
 -7.24987e-16  -3.84805e-16  -6.40421e-16       2.18725e-15  -4.996e-15
 -3.70411e-16  -5.16291e-17  -7.1818e-16   …   1.94999e-15   5.7454e-15
  1.0265e-15   -2.7893e-16   -1.1588e-16       -2.57221e-15  -2.13718e-15
  6.16813e-17  -8.80087e-17   2.64214e-16      -9.35483e-16  -9.58627e-16
  6.12258e-16  -2.95635e-16  -1.47594e-16       1.0          -1.17938e-14
  1.80411e-16   5.96745e-16  -6.245e-17        -1.17938e-14   1.0
```

In [47]:
```
q = gs([rand(20) for _ in 1:30]);
```

```
Dependent vectors after 21 iterations
```

Now for the question

In [48]:

```
a(k,n) = [ones(k) ; zeros(n-k)]

n = 10
vecs = [a(k,n) for k in 1:n]
gs(vecs)
```

Out[48]:

```
10-element Array{Array{Float64,1},1}:
 [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
```

In [49]:

```julia
n = 100
vecs = [a(k,n) for k in 1:n]
gs(vecs)
```

Out[49]:

```
100-element Array{Array{Float64,1},1}:
 [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 ⋮
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  1.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 1.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 1.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 1.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0  …  0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
```

Now shuffle the order

In [54]:
```
using Random
Random.seed!(0)
n = 10
vecs = shuffle([a(k,n) for k in 1:n])
gs(vecs)
```

Out[54]:
```
10-element Array{Array{Float64,1},1}:
 [0.353553, 0.353553, 0.353553, 0.353553, 0.353553, 0.353553, 0.353553,
0.353553, 0.0, 0.0]
 [0.612372, 0.612372, -0.204124, -0.204124, -0.204124, -0.204124, -0.2041
24, -0.204124, 0.0, 0.0]
 [0.0, 0.0, 0.408248, 0.408248, 0.408248, -0.408248, -0.408248, -0.40824
8, 0.0, 0.0]
 [0.0, 0.0, 0.408248, 0.408248, -0.816497, 1.35974e-16, 1.35974e-16, 1.35
974e-16, 0.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.408248, 0.408248, -0.816497, 0.0, 0.0]
 [0.0, 0.0, 1.11022e-16, 1.11022e-16, 2.22045e-16, -2.22045e-16, -2.22045
e-16, 1.9984e-15, 1.0, 0.0]
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.9984e-15, 1.0]
 [0.0, 0.0, 1.57009e-16, 1.57009e-16, -3.14018e-16, 0.707107, -0.707107,
7.85046e-16, -3.14018e-16, 0.0]
 [0.0, 0.0, 0.707107, -0.707107, -7.85046e-17, -1.57009e-16, 1.57009e-16,
-4.61935e-31, -1.57009e-16, 0.0]
 [0.707107, -0.707107, 3.92523e-17, 3.92523e-17, 3.92523e-17, 3.92523e-1
7, 3.92523e-17, 3.92523e-17, 0.0, 0.0]
```

In [55]:
```
vecs[1]
```

Out[55]:
```
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 0.0
 0.0
```

As appears, a different order of input yields a different set of vectors. Here for example, the first vector was e_8 and thus it was normalized... The rest of the results were very different

Solution to Question 3

A and B are matrices of the same dimension.

(a) If $Ax = Bx$ holds for all vectors x then $A = B$. This is because we can use $x = e_i$ for all i and this shows that the i'th collumn of A is the same as that of B.

(b) If $Ax = Bx$ for some non zero vector it doesn't imply that $A = B$. Take for example $x = [1 \;\; 1]^T$ with,

$$A = [1 \;\; 0], \qquad B = [0 \;\; 1].$$

Solution to Question 4

$A \in \mathbb{R}^{m \times n}$. We want to show that A has the same null-space as $A^T A$ (a matrix in $\mathbb{R}^{n \times n}$). Define, the null-spaces of A and $A^T A$ resperespectivelyctivily,

$$\mathcal{N}_1 = \{x \in \mathbb{R}^n \mid Ax = 0_m\} \qquad \mathcal{N}_2 = \{x \in \mathbb{R}^n \mid A^T Ax = 0_n\}.$$

Take $x \in \mathcal{N}_1$. Then $Ax = 0_m$. Left multiply by A^T then $A^T Ax = 0_n$. Hence $x \in \mathcal{N}_2$. Hence $\mathcal{N}_1 \subset \mathcal{N}_2$.

Take now $x \in \mathcal{N}_2$. Then $A^T Ax = 0_n$. Left multiply by x^T:

$$x^T A^T Ax = 0$$

or

$$(Ax)^T Ax = 0$$

Hence $||Ax||^2 = 0$ and $|Ax|| = 0$. Hence Ax must be the zero vector, that is $Ax = 0_m$. Hence $x \in \mathcal{N}_1$. Hence $\mathcal{N}_2 \subset \mathcal{N}_1$.

Since $\mathcal{N}_1 \subset \mathcal{N}_2$ and $\mathcal{N}_2 \subset \mathcal{N}_1$, we have that $N_1 = N_2$.

Solution to Question 5

$A^T = -A$

(a) For a 2×2 matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

We have $a = -a$, $d = -d$ and $b = -c$. This implies that a and d must be zero and there is a single number α with

$$A = \begin{bmatrix} 0 & \alpha \\ -\alpha & 0 \end{bmatrix}.$$

(b) In general we must have that $A_{ii} = -A_{ii}$ hence $A_{ii} = 0$ (diagonal elements are 0).

(c) Assume $A^T = -A$:

$$(Ax)^T x = x^T A^T x = -(x^T Ax) = -((Ax)^T x)$$

Hence the inner product must be 0 and Ax is orthogonal to x.

(d) Now assume $(Ax)^T x = 0$ for any vector x. That is $x^T A^T x = 0$. Hence,

$$x^T A^T x = 0,$$

or taking transpose of the scalar,

$$x^T A x = 0.$$

That is,

$$\sum_{k=1}^{n} \sum_{\ell=1}^{n} x_k x_\ell A_{k\ell} = 0.$$

If we set $x = e_i$ for some i we get that $A_{ii} = 0$.

Now if we set $x = e_i + e_j$ for $i \neq j$ then

$$(e_i + e_j)^T A (e_i + e_j) = e_i^T A e_i + e_i^T A e_j + e_j^T A e_i + e_j^T A e_j = 0$$

or,

$$A_{ii} + A_{ij} + A_{ji} + A_{jj} = 0.$$

Since we already showed that A_{ii} and A_{jj} are 0 we get $A_{ij} = -A_{ji}$ and hence $A = -A^T$.

(e) Take any square matrix $\tilde{A}$ and look at $A = \tilde{A} - \tilde{A}^T$. Observe that it is skew-symmetric since $A^T = -A$

In [110]:
```
function randSkewSym(n)
    A = rand(n,n)
    A-A'
end

innerProds = []

for _ in 1:10^5
    A = randSkewSym(5)
    x = rand(5)
    push!(innerProds,dot(A*x,x))
end
minimum(innerProds),maximum(innerProds)
```

Out[110]:
```
(-5.551115123125783e-16, 6.661338147750939e-16)
```

We see above that all inner products were "0"

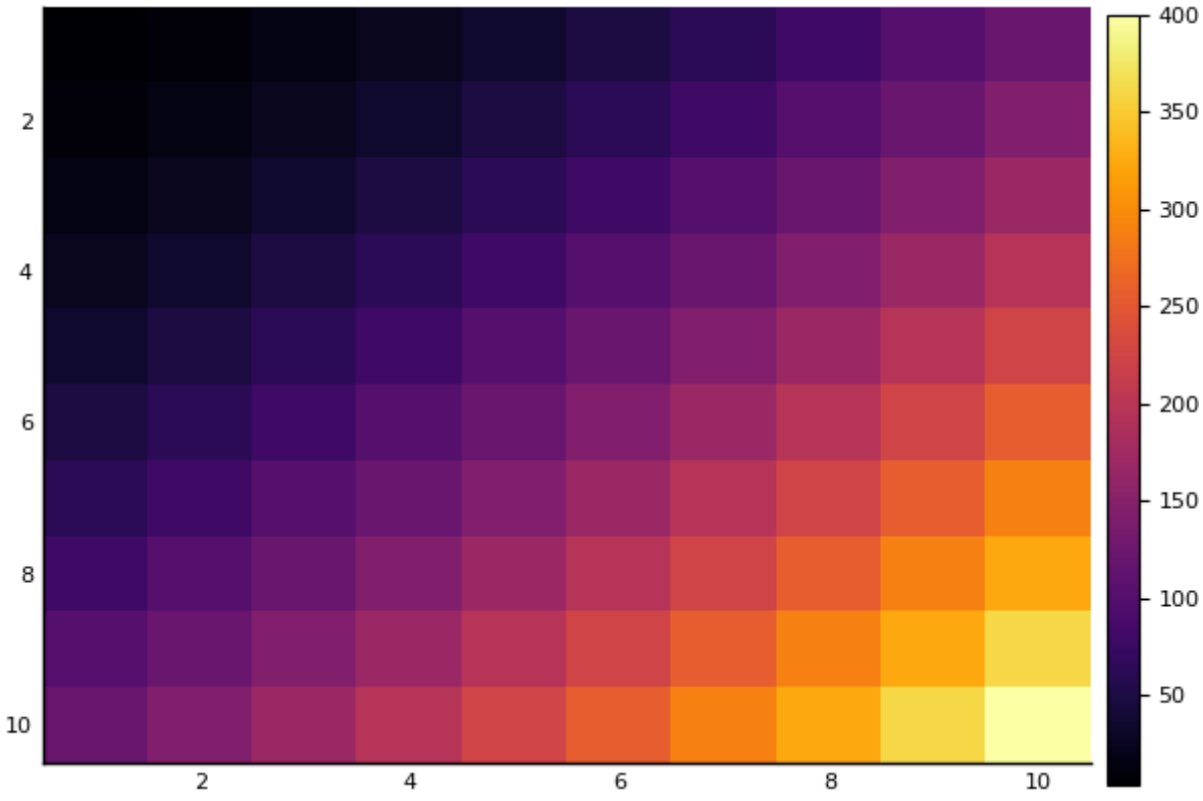
Solution to Question 6

Note: Important in this question to find the **matrix of the linear transformation**

In [173]:

```
using Plots
pyplot()
N = 10
image = [(i+j)^2 for i in 1:N, j in 1:N]
heatmap(image,yflip = true)
```

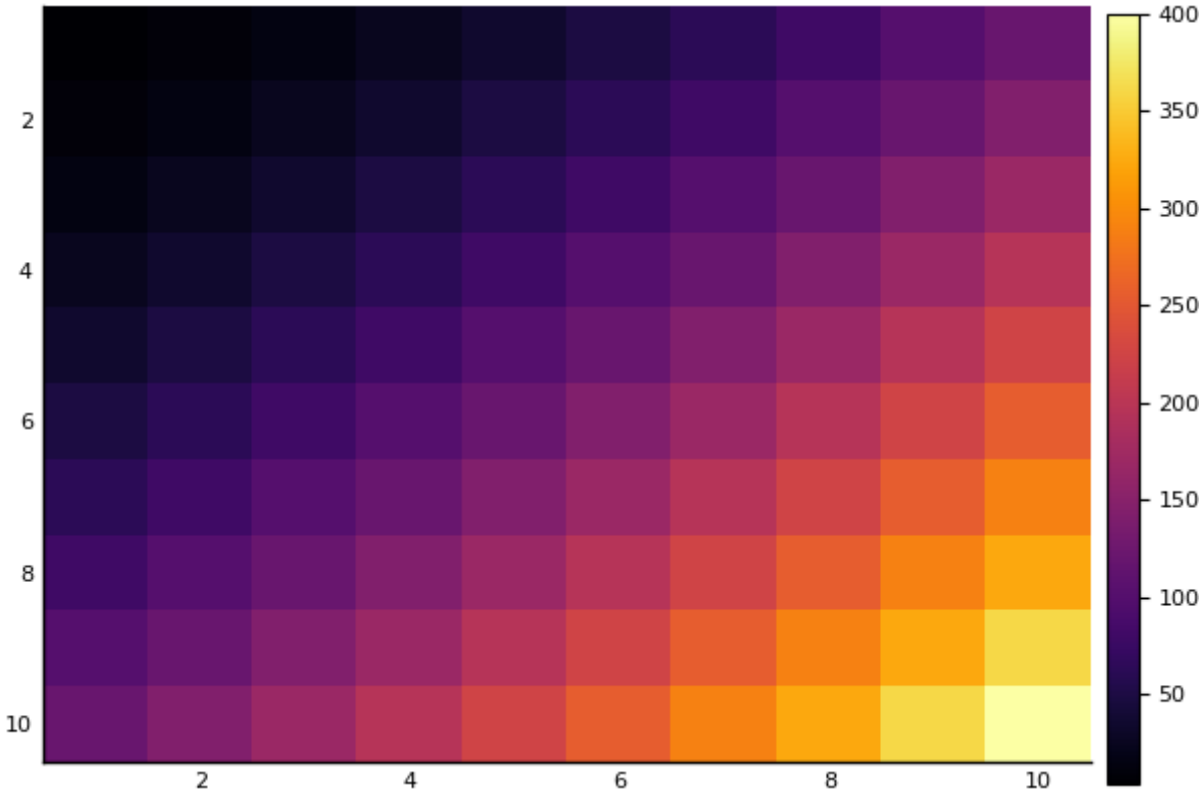
Out[173]:



Functions from changing image to vector and back

In [174]:
```
1 img2Vec(img) = vcat([img[:,j] for j in 1:N]...)
2 vec2Img(vec) = reshape(vec,N,N)
3 heatmap(vec2Img(img2Vec(image)),yflip = true) #test
```

Out[174]:



In [175]:
```
1 #The key to solving the problem is to work with identity vectors
2 ee(i) = [j ==i ? 1 : 0 for j in 1:N^2]
```

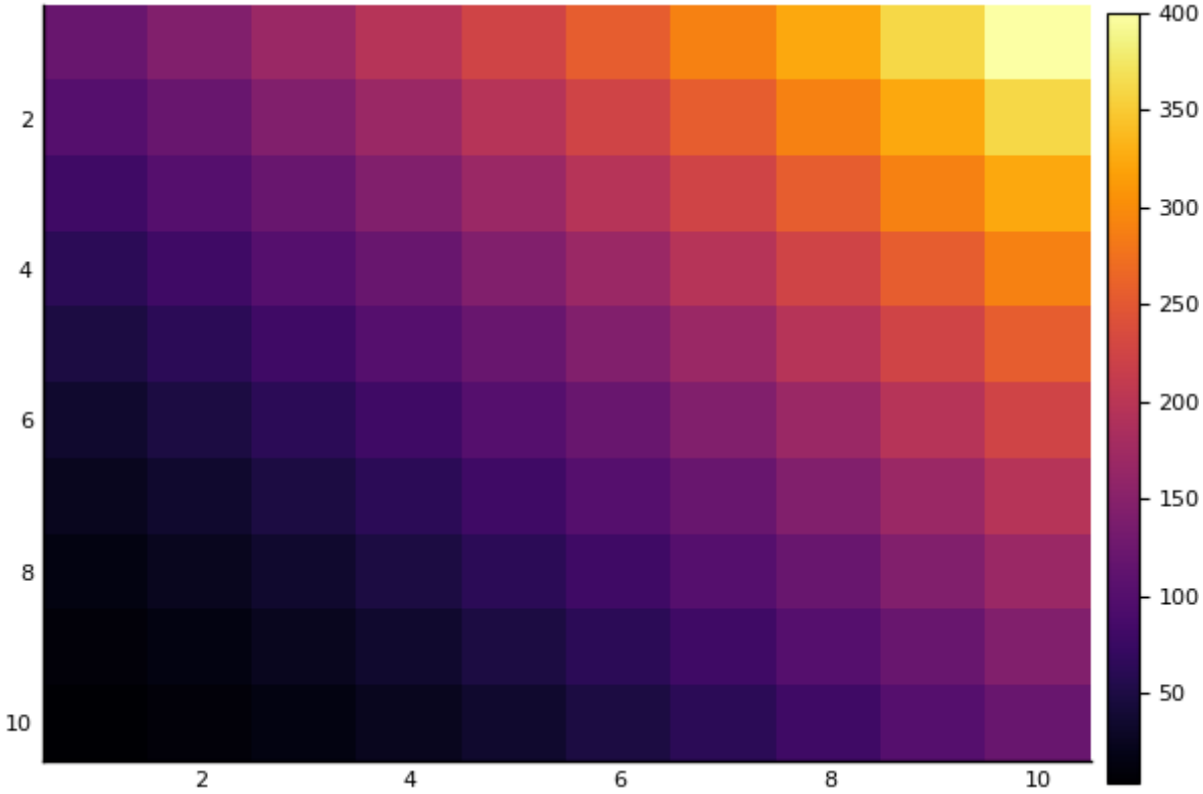
Out[175]:
```
ee (generic function with 1 method)
```

(a) Turning the orignial image upside down.

In [176]:

```
#This is the function for flipping the image (without thinking about ma
flipImageUpDown(img) = [img[i,j] for j in N:-1:1, i in 1:N]

#testIt
heatmap(
    flipImageUpDown(image),
    yflip = true)
```

Out[176]:



HOWEVER: The above isn't what we asked for. We wanted the MATRIX that does it... Here it is:

We get it by assuming the transformation is linear and applying it to the vectors $e_1, \ldots, e_{N^2}$....

In [177]:
```
1 Aflip = hcat([img2Vec(flipImageUpDown(vec2Img(ee(i)))) for i in 1:N^2]..
```

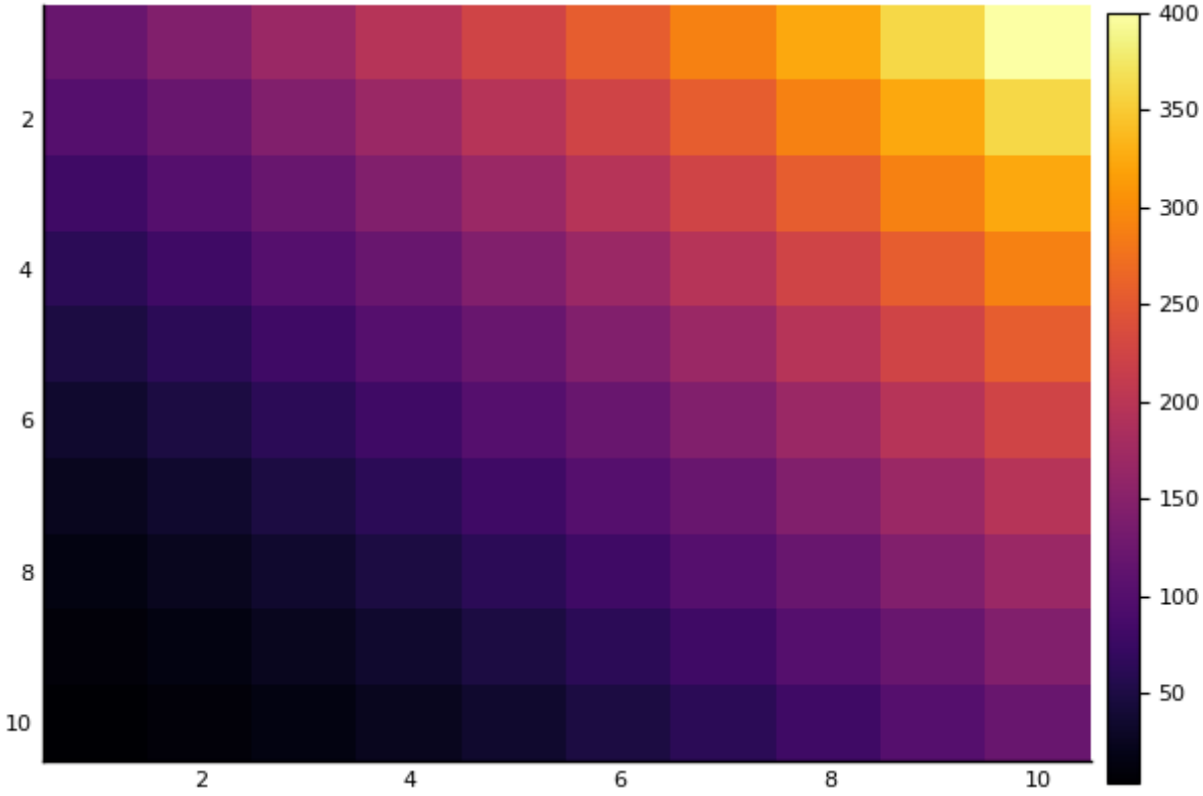
Out[177]:
```
100×100 Array{Int64,2}:
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  1  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  1  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 1  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  1  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 ⋮              ⋮              ⋮        ⋱        ⋮              ⋮
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  1  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  1
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  1  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  1  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
```

Testing it:

In [178]:

```
heatmap(
    vec2Img(
        Aflip*img2Vec(image)
        ),
    yflip = true)
```

Out[178]:

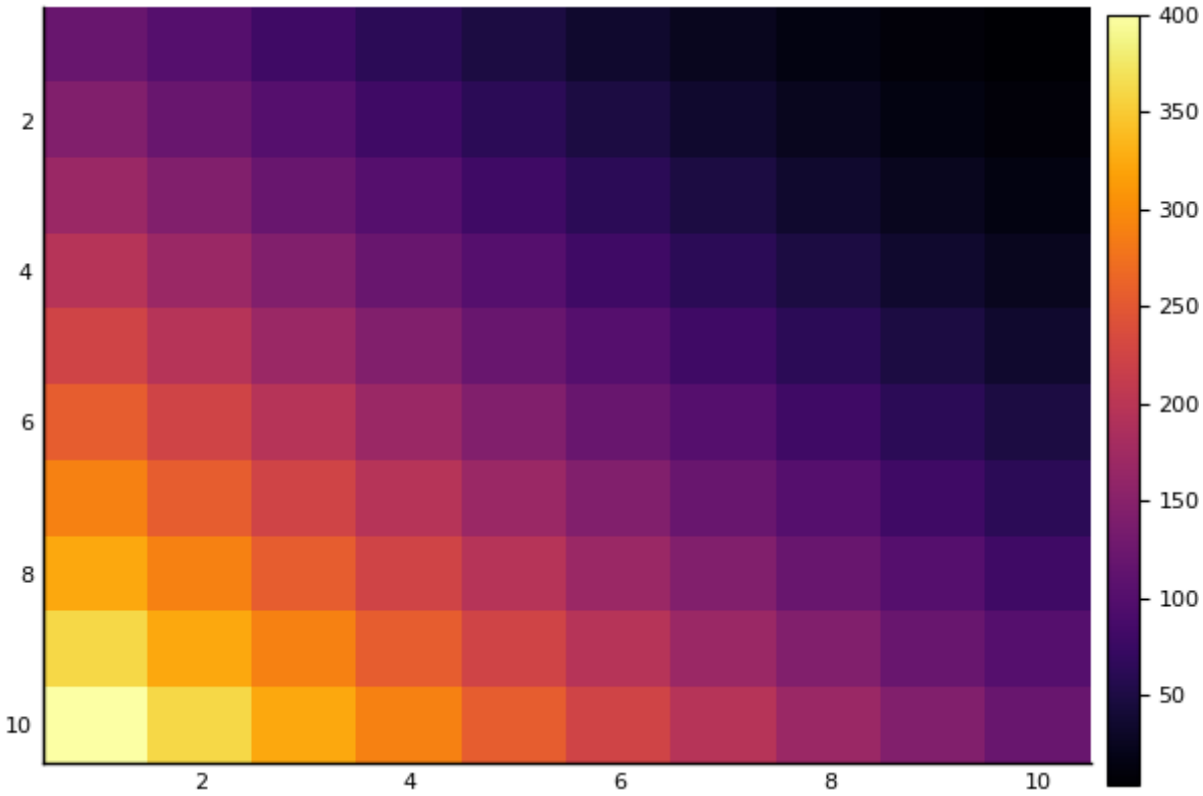


(a) Rotate the original image clockwise 90 degrees

In [179]:

```
#This is the function for it (without thinking about matrix multiplicati

rotateClockWise90(img) = [img[i,j] for j in 1:N, i in N:-1:1]

#testIt
heatmap(
    rotateClockWise90(image),
    yflip = true)
```

Out[179]:



In [180]:

```
#We can do: (this is just like before only with the rotateClockWise90 fu
#Arotate = hcat([img2Vec(rotateClockWise90(vec2Img(ee(i)))) for i in 1:1

#instead lates make a function that gives us the linear transformation o

makeA(trans) = hcat([img2Vec(trans(vec2Img(ee(i)))) for i in 1:N^2]...)

#Now
Arotate = makeA(rotateClockWise90)
```

Out[180]:

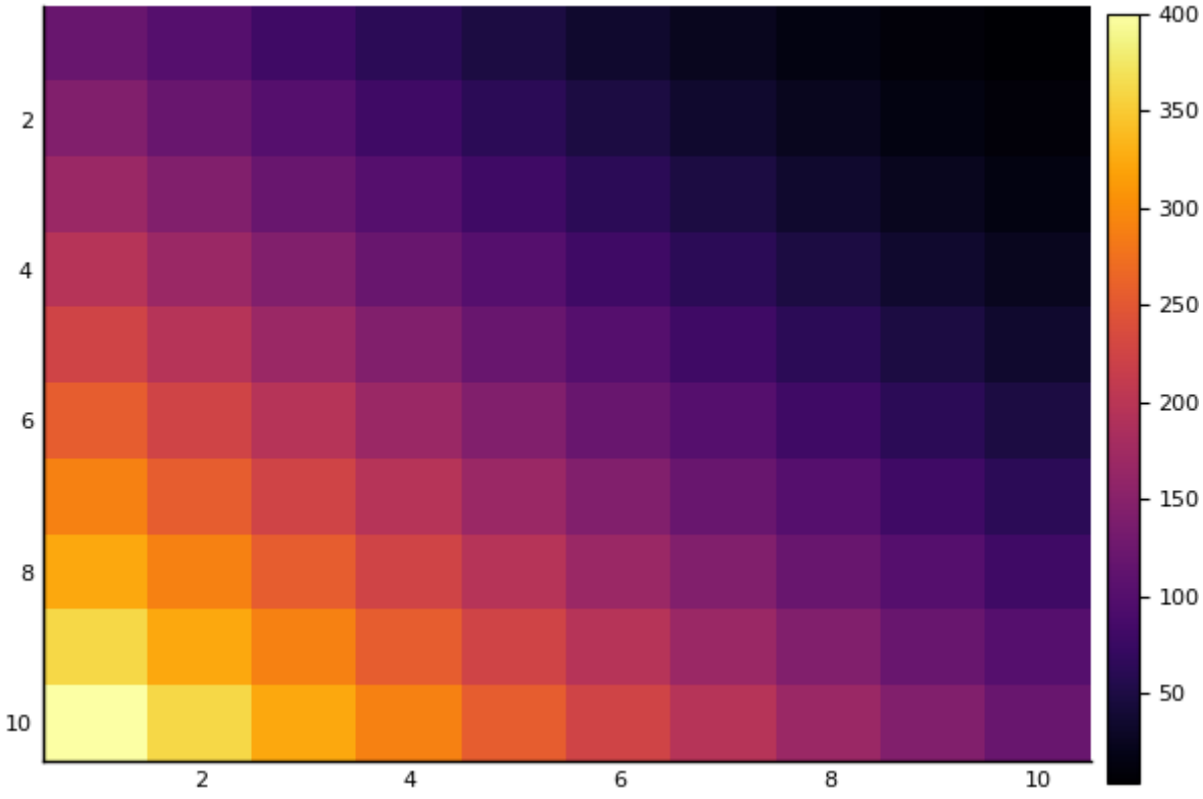
```
100×100 Array{Int64,2}:
 0  0  0  0  0  0  0  0  0  1  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  1  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  1
 0  0  0  0  0  0  0  0  1  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 ⋮                 ⋮                 ⋮          ⋱          ⋮                 ⋮
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  1  0  0  0  0  0  0
0  0
 1  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  1  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
```

```
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0
0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  1  0  0  0  0  0  0  0
0  0
```

Testing it:

In [181]:
```
heatmap(
    vec2Img(
        Arotate*img2Vec(image)
        ),
    yflip = true)
```

Out[181]:

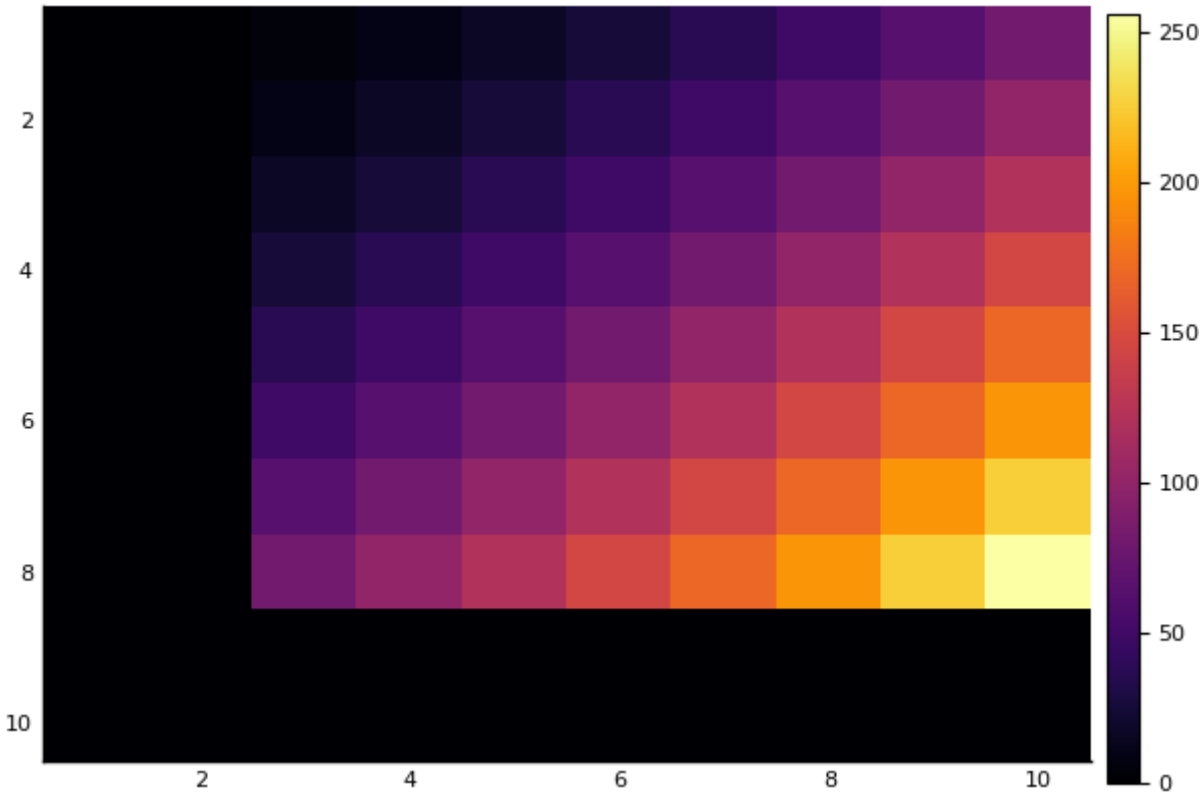


(c) Translate up and 2 right by 2 pixels...

In [198]:

```
#This is the function for it (without thinking about matrix multiplicati

function translateUpRight(img)
    outImage = zeros(N,N)
    for i in 1:N-2
        for j in 3:N
            outImage[i,j] = img[i,j-2]
        end
    end
    outImage
end

#testIt
heatmap(
    translateUpRight(image),
    yflip = true) #note the heatmap renormalizes the colours...
```

Out[198]:



In [199]:
```
1 translateUpRight(image)
```

Out[199]:
```
10×10 Array{Float64,2}:
 0.0  0.0   4.0    9.0   16.0   25.0   36.0   49.0   64.0   81.0
 0.0  0.0   9.0   16.0   25.0   36.0   49.0   64.0   81.0  100.0
 0.0  0.0  16.0   25.0   36.0   49.0   64.0   81.0  100.0  121.0
 0.0  0.0  25.0   36.0   49.0   64.0   81.0  100.0  121.0  144.0
 0.0  0.0  36.0   49.0   64.0   81.0  100.0  121.0  144.0  169.0
 0.0  0.0  49.0   64.0   81.0  100.0  121.0  144.0  169.0  196.0
 0.0  0.0  64.0   81.0  100.0  121.0  144.0  169.0  196.0  225.0
 0.0  0.0  81.0  100.0  121.0  144.0  169.0  196.0  225.0  256.0
 0.0  0.0   0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
 0.0  0.0   0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
```

In [221]:
```
#this is then the linear transformation
Atranslate = makeA(translateUpRight)
```

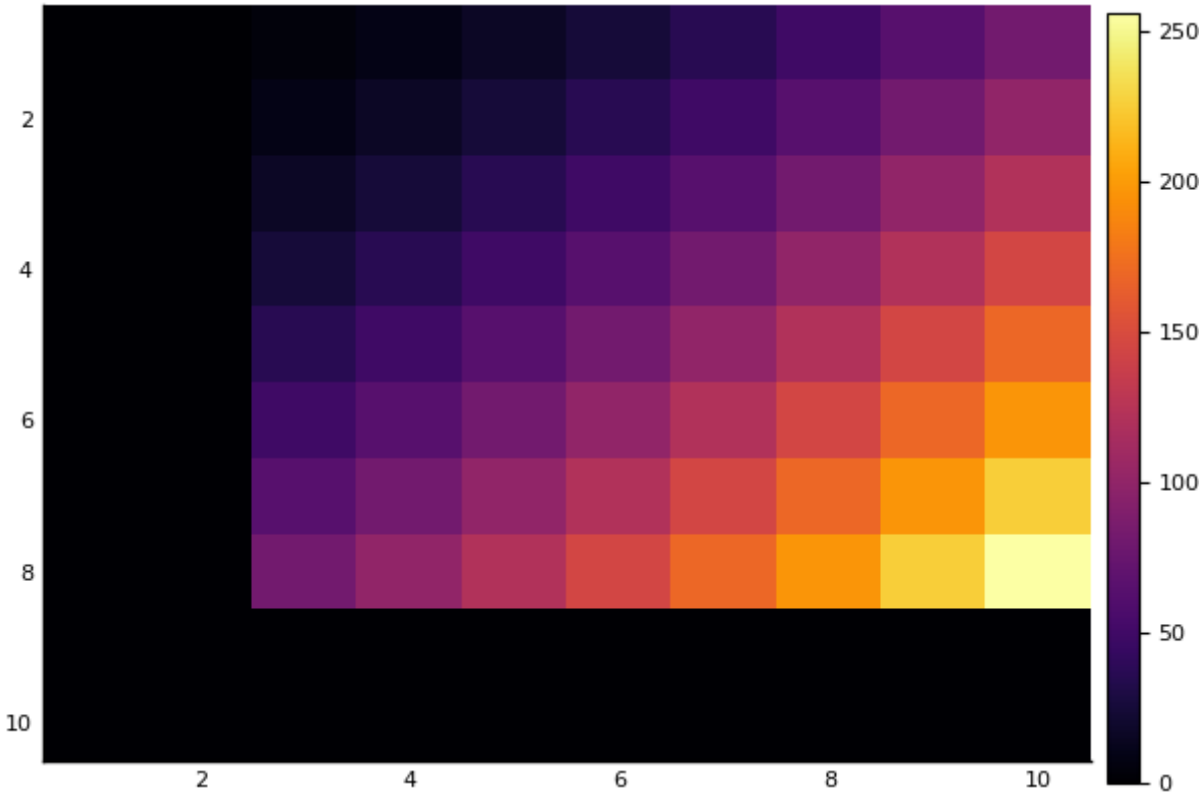
Out[221]:
```
100×100 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 ⋮                        ⋮              ⋱           ⋮
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0
0.0
```

Testing it:

In [222]:

```
heatmap(
    vec2Img(
        Atranslate*img2Vec(image)
        ),
    yflip = true)
```

Out[222]:

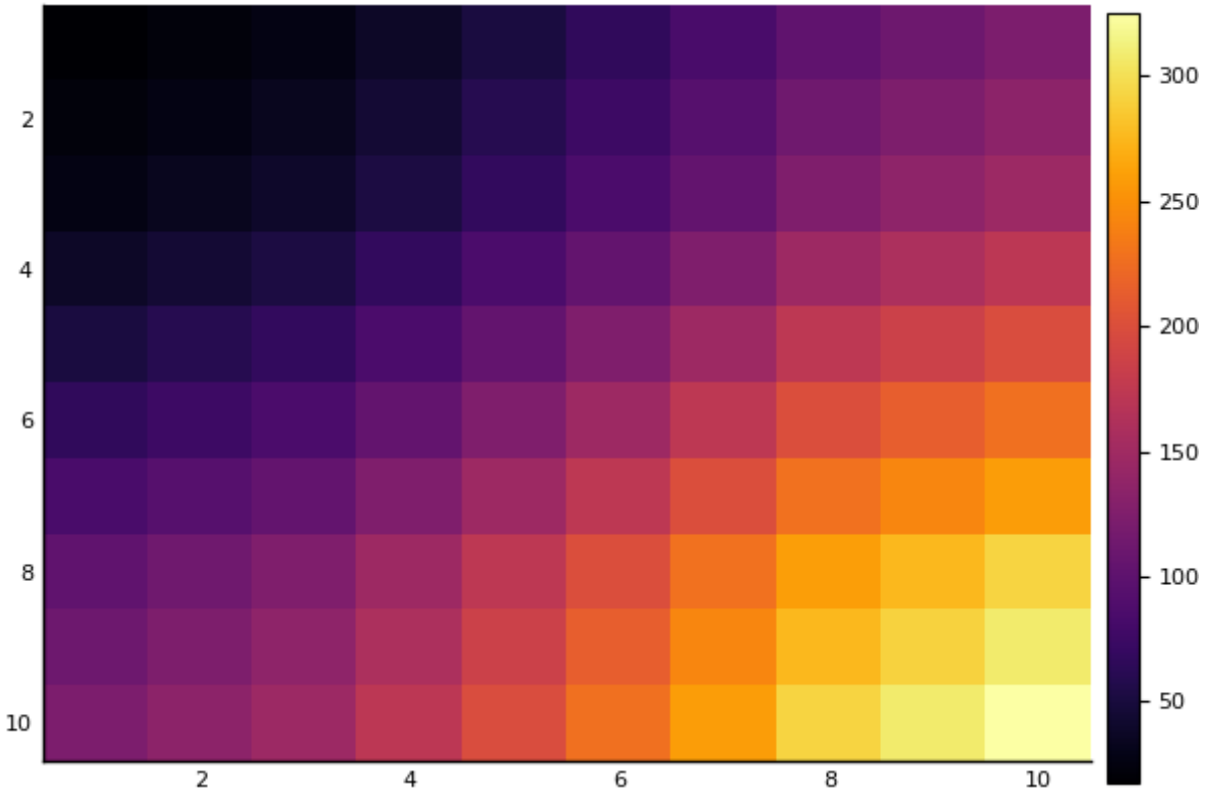


(d) Smooth (average of pixels)

In [229]:

```
using Statistics

m1(i) = max(i,1)
mN(i) = min(i,N)

d = 2

#This is the function for it (without thinking about matrix multiplicati
#It takes the average of the neighbouring d cells in each direction m1(
function smooth(img)
    outImage = zeros(N,N)
    for i in 1:N
        for j in 1:N
            outImage[i,j] = mean(img[m1(i-d):mN(i+d),m1(j-d):mN(j+d)])
        end
    end
    outImage
end

#testIt
heatmap(
    smooth(image),
    yflip = true) #note the heatmap renormalizes the colours...
```

Out[229]:



Note image looks the same, but slightly different

In [228]:
```
1 image
```

Out[228]:
```
10×10 Array{Int64,2}:
   4    9   16   25   36   49   64   81  100  121
   9   16   25   36   49   64   81  100  121  144
  16   25   36   49   64   81  100  121  144  169
  25   36   49   64   81  100  121  144  169  196
  36   49   64   81  100  121  144  169  196  225
  49   64   81  100  121  144  169  196  225  256
  64   81  100  121  144  169  196  225  256  289
  81  100  121  144  169  196  225  256  289  324
 100  121  144  169  196  225  256  289  324  361
 121  144  169  196  225  256  289  324  361  400
```

In [227]:
```
1 smooth(image)
```

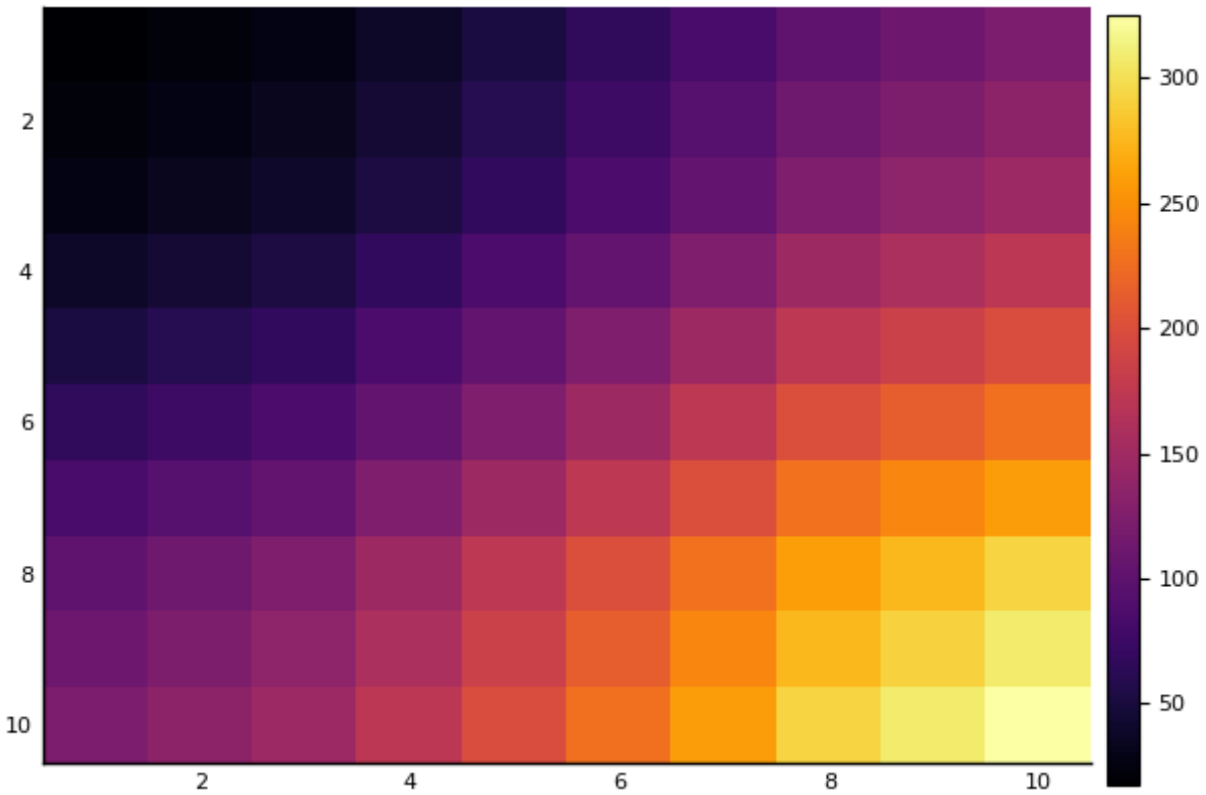
Out[227]:
```
10×10 Array{Float64,2}:
  17.3333   22.1667   27.6667  …   83.6667  102.667  112.167  122.333
  22.1667   27.5      33.5          93.5     113.5    123.5    134.167
  27.6667   33.5      40.0         104.0     125.0    135.5    146.667
  38.6667   45.5      53.0         125.0     148.0    159.5    171.667
  51.6667   59.5      68.0         148.0     173.0    185.5    198.667
  66.6667   75.5      85.0      …  173.0     200.0    213.5    227.667
  83.6667   93.5     104.0         200.0     229.0    243.5    258.667
 102.667   113.5     125.0         229.0     260.0    275.5    291.667
 112.167   123.5     135.5         243.5     275.5    291.5    308.167
 122.333   134.167   146.667       258.667   291.667  308.167  325.333
```

In [230]:
```
#this is then the linear transformation
Asmooth = makeA(smooth)
```

Out[230]:
```
100×100 Array{Float64,2}:
 0.111111   0.111111   0.111111   …  0.0        0.0        0.0
 0.0833333  0.0833333  0.0833333     0.0        0.0        0.0
 0.0666667  0.0666667  0.0666667     0.0        0.0        0.0
 0.0        0.0666667  0.0666667     0.0        0.0        0.0
 0.0        0.0        0.0666667     0.0        0.0        0.0
 0.0        0.0        0.0        …  0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0833333  0.0833333  0.0833333  …  0.0        0.0        0.0
 0.0625     0.0625     0.0625        0.0        0.0        0.0
 0.05       0.05       0.05          0.0        0.0        0.0
 ⋮                                 ⋱
 0.0        0.0        0.0           0.0625     0.0625     0.0625
 0.0        0.0        0.0           0.0833333  0.0833333  0.0833333
 0.0        0.0        0.0        …  0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0           0.0        0.0        0.0
 0.0        0.0        0.0        …  0.0666667  0.0        0.0
 0.0        0.0        0.0           0.0666667  0.0666667  0.0
 0.0        0.0        0.0           0.0666667  0.0666667  0.0666667
 0.0        0.0        0.0           0.0833333  0.0833333  0.0833333
 0.0        0.0        0.0           0.111111   0.111111   0.111111
```

In [231]:
```
heatmap(
    vec2Img(
        Asmooth*img2Vec(image)
        ),
    yflip = true)
```

Out[231]:



In [232]:
```
vec2Img(
        Asmooth*img2Vec(image)
        )
```

Out[232]:
```
10×10 Array{Float64,2}:
  17.3333   22.1667   27.6667  …   83.6667  102.667  112.167  122.333
  22.1667   27.5      33.5         93.5     113.5    123.5    134.167
  27.6667   33.5      40.0        104.0     125.0    135.5    146.667
  38.6667   45.5      53.0        125.0     148.0    159.5    171.667
  51.6667   59.5      68.0        148.0     173.0    185.5    198.667
  66.6667   75.5      85.0     …  173.0     200.0    213.5    227.667
  83.6667   93.5     104.0        200.0     229.0    243.5    258.667
 102.667   113.5     125.0        229.0     260.0    275.5    291.667
 112.167   123.5     135.5        243.5     275.5    291.5    308.167
 122.333   134.167   146.667      258.667   291.667  308.167  325.333
```

Solution to Question 7

$$f : [-1, 1] \to \mathbb{R} \qquad \alpha = \int_{-1}^{1} f(x)\, dx.$$

$$\hat{\alpha} = w_1 f(t_1) + \ldots w_n f(t_n)$$

(a) Take $f(\cdot)$ is a polynomial of degree d:

$$f(x) = \beta_0 + \beta_1 x + \ldots + \beta_d x^d$$

Hence in this case,

$$\alpha = \beta_0 x\Big|_{-1}^{1} + \beta_1 \frac{1}{2}x^2\Big|_{-1}^{1} + \beta_2 \frac{1}{3}x^3\Big|_{-1}^{1} + \ldots + \beta_d \frac{1}{d+1}x^{d+1}\Big|_{-1}^{1} = 2\left(\beta_0 + \frac{1}{3}\beta_2 + \frac{1}{5}\beta_4 + \ldots\right)$$

Now

$$\hat{\alpha} = (w_1 + \ldots + w_n)\beta_0 + (w_1 t_1 + \ldots + w_n t_n)\beta_1 + (w_1 t_1^2 + \ldots + w_n t_n^2)\beta_2 + \ldots$$

We can now equate coefficients of α and $\hat{\alpha}$ to get the system of equations $Ax = b$ where A is $(d+1) \times n$:

$$\begin{bmatrix} 1 & 1 & \ldots & 1 \\ t_1 & t_2 & \ldots & t_n \\ t_1^2 & t_2^2 & \ldots & t_n^2 \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ t_1^d & t_2^d & \ldots & t_n^d \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ \vdots \\ \vdots \\ w_n \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 0 \\ 1/3 \\ 0 \\ 1/5 \\ 0 \\ \vdots \end{bmatrix}$$

Here is a little demonstration that it works...

In [46]:
```
using Random
Random.seed!(0)
f(x) = 6 + 34x + 9x^2 - 5x^3 + 0.2x^4 + 3x^5
d = 5
δ = 0.00001
alpha = sum([f(x)*δ for x in -1:δ:1])
```

Out[46]:
```
18.08015200031334
```

In [47]:
```
n = 6;
ts = sort(2*rand(n).-1) # some arbitrary t values

A = [ts[j]^i for i in 0:d, j in 1:n]
```

Out[47]:
```
6×6 Array{Float64,2}:
  1.0        1.0        1.0         1.0        1.0        1.0
 -0.670868  -0.645342  -0.593047   -0.44224    0.647295   0.820713
  0.450064   0.416467   0.351705    0.195576   0.418991   0.67357
 -0.301934  -0.268764  -0.208577   -0.0864915  0.271211   0.552808
  0.202558   0.173445   0.123696    0.03825    0.175553   0.453696
 -0.13589   -0.111931  -0.0733576  -0.0169157  0.113635   0.372355
```

In [48]:
```
b = 2*[(i%2)/i for i in 1:d+1]
```

Out[48]:
```
6-element Array{Float64,1}:
 2.0
 0.0
 0.6666666666666666
 0.0
 0.4
 0.0
```

In [49]:
```
w = A \ b
```

Out[49]:
```
6-element Array{Float64,1}:
 -69.86735359420322
 131.08788006208087
 -71.2221609271108
  11.313909446190433
   0.4283850586230596
   0.2593399544196729
```

In [50]:
```
w'*f.(ts)
```

Out[50]:
```
18.080000000000176
```

(b) and (c): Considering different rules:

(i) Trapezoid rule $n = 2$ $t_1 = -1, t_2 = 1, w_1 = w_2 = 1/2$

$d = 1$ so A is 2×2:

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

Now $b = [2 \;\; 0]^T$ and we need to see that $w = [1/2 \;\; 1/2]^T$ satisfies $Aw = b$.

But it **doesn't** there was a mistake in the problem. It should have been $w_1 = w_2 = 1$.

In [54]:
```
A = [1 1; -1 1]; b = [2,0];
w = [1,1]
A*w - b
```

Out[54]:
```
2-element Array{Int64,1}:
 0
 0
```

In [92]:
```
f(x) = x != 0 ? sin(x)/x : 1 #can't evaluate sin(x)/x at x = 0, however
δ = 0.000001
alpha = sum([f(x)*δ for x in -1:δ:1])
```

Out[92]:
```
1.8921669822053007
```

In [74]:
```
ts = [-1,1]; w=[1,1]
alphaHat = w'*f.(ts)
```

Out[74]:
```
1.682941969615793
```

So we get $\hat{\alpha} = 1.6829$ vs. $\alpha = 1.8921$

(ii) Simpson's rule

In [75]:
```
ts = [-1,0,1]; w = [1/3, 4/3, 1/3];
d = 2; n = 3;
A = [ts[j]^i for i in 0:d, j in 1:n]
```

Out[75]:
```
3×3 Array{Int64,2}:
  1   1   1
 -1   0   1
  1   0   1
```

In [76]:
```
b = 2*[(i%2)/i for i in 1:d+1]
```

Out[76]:
```
3-element Array{Float64,1}:
 2.0
 0.0
 0.6666666666666666
```

Now see it satisfies $Aw = b$. So yes it is order $d = 2$

In [77]:
```
A*w - b
```

Out[77]:
```
3-element Array{Float64,1}:
 -2.220446049250313e-16
  0.0
  0.0
```

In [78]:
```
alphaHat = w'*f.(ts)
```

Out[78]:
```
1.8943139898719308
```

So we get $\hat{\alpha} = 1.8943$ vs. $\alpha = 1.8921$.

(ii) Simpson's 3/8 rule

In [81]:
```
ts = [-1, -1/3,1/3,1]; w = [1/4,3/4,3/4,1/4];
d = 3; n = 4;
A = [ts[j]^i for i in 0:d, j in 1:n]
```

Out[81]:
```
4×4 Array{Float64,2}:
  1.0   1.0       1.0       1.0
 -1.0  -0.333333  0.333333  1.0
  1.0   0.111111  0.111111  1.0
 -1.0  -0.037037  0.037037  1.0
```

Now see it satisfies $Aw = b$. So yes it is order $d = 3$

In [83]:
```
b = 2*[(i%2)/i for i in 1:d+1]
```

Out[83]:
```
4-element Array{Float64,1}:
 2.0
 0.0
 0.6666666666666666
 0.0
```

In [84]:
```
A*w - b
```

Out[84]:
```
4-element Array{Float64,1}:
  0.0
 -1.3877787807814457e-17
  0.0
  0.0
```

In [85]:
```
alphaHat = w'*f.(ts)
```

Out[85]:
```
1.8931116279866331
```

So we get $\hat{\alpha} = 1.8931$ vs. $\alpha = 1.8921$.

(d) Let's get even closer with an order 4 rule that we find

In [90]:
```
d = 4;
n = 5;
ts = [-1,-1/2,0,1/2,1] #just some sensible grid
A = [ts[j]^i for i in 0:d, j in 1:n]
```

Out[90]:
```
5×5 Array{Float64,2}:
  1.0    1.0      1.0  1.0      1.0
 -1.0   -0.5      0.0  0.5      1.0
  1.0    0.25     0.0  0.25     1.0
 -1.0   -0.125    0.0  0.125    1.0
  1.0    0.0625   0.0  0.0625   1.0
```

In [88]:
```
b = 2*[(i%2)/i for i in 1:d+1]
```

Out[88]:
```
5-element Array{Float64,1}:
 2.0
 0.0
 0.6666666666666666
 0.0
 0.4
```

So these are our weights:

```
In [89]:  1  w = A \ b
```

```
Out[89]: 5-element Array{Float64,1}:
          0.15555555555555545
          0.7111111111111117
          0.2666666666666662
          0.7111111111111111
          0.15555555555555553
```

```
In [91]:  1  alphaHat = w'*f.(ts)
```

```
Out[91]: 1.892156949525523
```

So we get $\hat{\alpha} = 1.8921569$ vs. $\alpha = 1.8921669$.

Solution to Question 8

We want $ab^T = ba^T$. This means the matrix $A = ab^T$, is symmetric since $A^T = (ab^T)^T = ba^T$. Hence first we must have that a and b are of the same length.

Since we want $A_{ij} = A_{ji}$ it means that

$$a_i b_j = a_j b_i.$$

If all of $a = 0$ or $b = 0$ this would hold immediatly. However assume $b \neq 0$ via $b_i \neq 0$. Now divide to get

$$\frac{a_j}{a_i} = \frac{b_j}{b_i}.$$

Hence the vectors a and b must have an angle of 0 between them.

A different (cleaner) way to arrive a this is via $ab^T = ba^T$. Left multiply by b^T and right multiply by a:

$$b^T a b^T a = b^T b a^T a$$

$$(b^T a)^2 = b^T a b^T a = \|b\|^2 \|a\|^2$$

$$\frac{a^T b}{\|b\|\|a\|} = 1$$

Hence the angle between the vectors needs to be 0.

Solution to Question 9

A and B are $m \times n$.

(a) $A^T B$ is $n \times n$ with i, j element

$$\sum_{k=1}^{m} A_{ik}^T B_{kj} = \sum_{k=1}^{m} A_{ki} B_{kj}$$

The trace is the sum over all i, i elements (for $i = 1, \ldots, n$)

$$tr(A^T B) = \sum_{i=1}^{n} \sum_{k=1}^{m} A_{ki} B_{ki}$$

which we can also write with index j instead of k if we like and change the summation order and role of variables to arrive at the desired expression.

Here is an example in Julia (even though it was not required).

In [87]:
```
Random.seed!(0)
n = 2;m=3;
A = rand([1,2],n,m)
B = rand([1,2],n,m)
A,B
```

Out[87]:
```
([1 2 1; 1 2 1], [1 2 2; 2 1 2])
```

In [92]:
```
A'*B #we see the trace is 3+6+4
```

Out[92]:
```
3×3 Array{Int64,2}:
 3  3  4
 6  6  8
 3  3  4
```

In [93]:
```
tr(A'*B)
```

Out[93]:
```
13
```

Using the formula....

In [96]:
```
sum(A.*B) #element wise multiplication
```

Out[96]:
```
13
```

(b) We already showed:

$$tr(A^T B) = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} B_{ij}$$

The same formula with A and B switched gives:

$$tr(B^T A) = \sum_{i=1}^{m} \sum_{j=1}^{n} B_{ij} A_{ij}$$

We thus see $tr(A^T B) = tr(B^T A)$.

(c) From (a): $tr(A^T A) = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} A_{ij}$

$$= \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2 = \|A\|^2 .$$

(Frobenius norm)

(d) We have

$$tr(A^T B) = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} B_{ij}$$

Using the same formula, and $A^T = (A^T)^T$, $B = (B^T)^T$:

$$tr(BA^T) = \sum_{i=1}^{m} \sum_{j=1}^{n} B_{ij}^T A_{ij}^T = \sum_{i=1}^{m} \sum_{j=1}^{n} B_{ji} A_{ji}$$

and after changing indexes i and j this equals $tr(A^T B)$.

As an example, consider the case of A and B being vectors ($m = n$ and $n = 1$). The trace of the inner product is just the trace of a scalar. The trace of the outer product is the innder product.

Solution to Question 10

It turns out that $A_i = Q_i R_i$ is the QR factorization of the first A_i because every additional column in A only gives and additional collumn in Q_i and expands the dimension of the square matrix R_i by 1 without touching the submatrix. The two equations that show this are:

(page 97 of [VMLS] in Gram-Schmidt algorithm):

$$\tilde{q}_i = a_i - (q_1^T a_i) q_1 - \ldots - (q_{i-1}^T a_i) q_{i-1}$$

(distilled page 190 in [VMLS]):

$$R_{ij} = \begin{cases} q_i^T a_j & i < j \\ \|\tilde{q}_i\| & i = j \\ 0 & i > j \end{cases}$$

Here is a demonstration

In [77]:
```
Random.seed!(0)
n = 5; k = 4;
A = rand([-1,0,1],n,k)
```

Out[77]:
```
5×4 Array{Int64,2}:
 -1   1   1  0
  1  -1   0  1
  0  -1   1  1
  0   0  -1  1
 -1   1   0  0
```

In [79]:
```
rank(A) #to make sure it is full rank (rank = 4)
```

Out[79]:
```
4
```

In [80]:
```
F = qr(A)
```

Out[80]:
```
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
5×5 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.57735  0.0  -0.516398  -0.507093  -0.377964
  0.57735  0.0  -0.258199  -0.676123   0.377964
  0.0      1.0   0.0        0.0        0.0
  0.0      0.0   0.774597  -0.507093  -0.377964
 -0.57735  0.0   0.258199  -0.169031   0.755929
R factor:
4×4 Array{Float64,2}:
 1.73205  -1.73205  -0.57735   0.57735
 0.0      -1.0       1.0       1.0
 0.0       0.0      -1.29099   0.516398
 0.0       0.0       0.0      -1.18322
```

This loop shows that all sub-QR factorizations hold... notice the norms of differences are almost 0

In [83]:
```
for i in 1:k
    subA = A[:,1:i]
    subQ = F.Q[:,1:i]
    subR = F.R[1:i,1:i]
    println( norm(subA - subQ*subR))
end
```

```
1.9229626863835638e-16
2.7194799110210365e-16
2.984744261173718e-16
3.2126737200132e-16
```