Table of Contents

In [3]:
```
using Colors, Interact, Plots, JLD, LinearAlgebra, Arpack
gr(
    label="",
    markerstrokewidth=0.3,
    markerstrokecolor="white"
)
```

▶ Run

Out[3]:
```
Plots.GRBackend()
```

A set of vectors is said to be *linearly dependent* if one of the vectors in the set can be expressed as a linear combination of the others. If no vector in the set can be written as a linear combination of the others, then the vectors are said to be *linearly independent*.

Concretely, suppose $a_1, \ldots, a_n$ are a set of vectors. Then they are considered to be linearly dependent if there exist scalars $\alpha_1, \ldots, \alpha_n$ (not all zero) such that

$$\alpha_1 a_1 + \ldots + \alpha_n a_n = 0. \tag{1}$$

If no such scalars exist, then the vectors are linearly independent. In other words, if the vectors are linearly independent, then the only way to satisfy the equation above is to set $\alpha_1 = \ldots, \alpha_n = 0$.

The vectors $a_1 = [1, 0]^T$ and $a_2 = [2, 0]^T$ are linearly dependent because $a_2 = 2a_1 \Rightarrow a_2 - 2a_1 = 0$.

- [x] True
- [ ] False

Submit

The vectors $a_a = [1, 0]^T$ and $a_2 = [0, 1]^T$ are linearly dependent

- [ ] True
- [x] False

Submit

The **column rank** of an $m \times n$ matrix A is the number of linearly independent columns.

The **row rank** of an $m \times n$ matrix A is the number of linearly independent rows.

A remarkable fact of linear algebra is that the number of linearly independent rows of a matrix equals the number of linearly indepenent columns:

$$r = \text{rank}(A) = \text{rank}(A^T) \tag{1}$$

Since the number of linear independent columns (or rows) cannot exceed the number of columns (or, resp., rows) in a matrix, this implies that

$$r \leq \min(m, n). \tag{2}$$

A is a 10×5 matrix. It can have rank (select **all** that apply)

- [ ] 10
- [ ] 7
- [x] 5
- [x] 2
- [x] 1

Submit

Consider the matrix

$$A = xy^T,$$

where x is an m-dimensional vector, and

$$y = \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}^T.$$

Consequently, the matrix A can be written in column partition format as

$$A = \begin{bmatrix} y_1 x & \dots & y_n x \end{bmatrix}.$$

We would like to determine the rank of A.

Assume $x, y \neq 0$, and there is an index i such that $y_i \neq 0$. The rank of $A = xy^T$ is 1 because (select **all** that apply)

- [x] $1 \cdot A[:, j] - (y_j / y_i) \cdot A[:, i] = 0$ for every $j = 1, \dots n$ so the columns of A are linearly dependent

- [x] Every column of A is a scalar multiple of its i-th column $y_i x$. Thus the i-th column is the only independent column -- the rest depend on it.
- [ ] A as written in column partition format "looks" like it has just one row

Submit

Now suppose that x_1 and x_2 are linearly independent vectors and

$$A = x_1 y_1^T + x_2 y_2^T.$$

What is the rank of A? Let us begin by examining the structure of A -- as earlier, it can be expressed as

$$A = \begin{bmatrix} y_{11}x_1 & \dots & y_{1n}x_1 \end{bmatrix} + \begin{bmatrix} y_{21}x_2 & \dots & y_{2n}x_2 \end{bmatrix},$$

or equivalently

$$A = \begin{bmatrix} y_{11}x_1 + y_{21}x_2 & \dots & y_{1n}x_1 + y_{2n}x_2 \end{bmatrix}.$$

We can now observe that every column of A is a linear combination of two linearly independent vectors. Thus the rank of A is at most two. Note now that

$$A^T = y_1 x_1^T + y_2 x_2^T,$$

and so we can see that the rank of A also depends whether y_1 and y_2 are linearly independent. If they are linearly dependent (and not equal to zero), then the rank of A^T (and hence A) will be 1!

Being able to spot low-rank matrices via the dependency structure in the rows or columns is a valuable skill in data science, machine learning and computational mathematics.

Suppose

$$x_1 = [1,0]^T, \quad x_2 = [0,1]^T, \quad y_1 = [1,0]^T, \quad y_2 = [1,1]^T.$$

Then the rank of $x_1 y_1^T + x_2 y_2^T$ is

- [x] 2, because x_1 and x_2 are linearly independent, and so are y_1 and y_2.
- [ ] 1, because y_1 and y_2 are linearly dependent.

Submit

Suppose

$$x_1 = [1,0]^T, \quad x_2 = [2,0]^T, \quad y_1 = [1,0]^T, \quad y_2 = [1,1]^T.$$

Then the rank of $x_1 y_1^T + x_2 y_2^T$ is

- [x] 1, because x_1 and x_2 are linearly dependent.
- [ ] 1, because y_1 and y_2 are linearly dependent.
- [ ] 2, because x_1 and x_2 are linearly independent, and so are y_1 and y_2.

Submit

Suppose

$$x_1 = [1,0]^T, \quad x_2 = [1,1]^T, \quad x_3 = [0,1]^T, \quad y_1 = [1,0]^T, \quad y_2 = [1,1]^T, \quad y_3 = [2^{-1/2}, 3$$

Then the rank of $x_1 y_1^T + x_2 y_2^T + x_3 y_3^T$ is (select **all** that apply)

- [x] at most 2, because it is a 2×2 matrix.
- [x] 2, because x_3 is linearly dependent on x_1 and x_2, but x_1 and x_2 are linearly independent.
- [x] 2, because y_3 is linearly dependent on y_1 and y_2, but y_1 and y_2 are linearly independent.
- [ ] 1, because x_3 is linearly dependent on x_1 and x_2.
- [ ] 1, because y_3 is linearly dependent on y_1 and y_2.

Submit

2.1 Spotting low-rank matrices

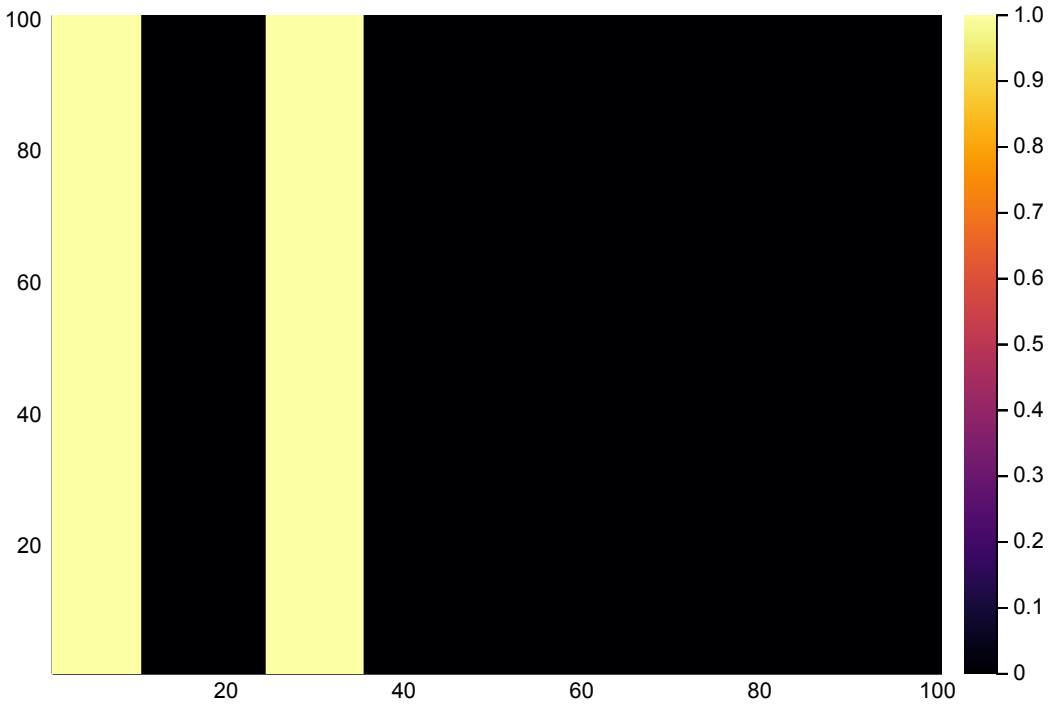
The ability to spot low-rank matrices "in the wild" is a useful skill in computational data science and applied mathematics. Let's exercise that muscle with some warm-ups :-)

Tip: For the questions that follow, express the matrix as xy^T or $x_1 y_1^T + x_2 y_2^T$, then examine the dependency structure of the x's and y's to deduce the rank of the matrix.

In [4]:
```
A = [ (1 <= j <= 10) || (25 <= j <= 35) ? 1 : 0
    for i in 1:100, j in 1:100 ]
heatmap(A)
```

Run

Out[4]:



The rank of A is

- [ ] 100, because there are 100 rows and columns in the matrix.

- [ ] 2, because there are two non-zero columns.
- [x] 1, because the same column is repeated in the other non zero columns.

Submit

`A` can be expressed as the outer product `x * y'`, where

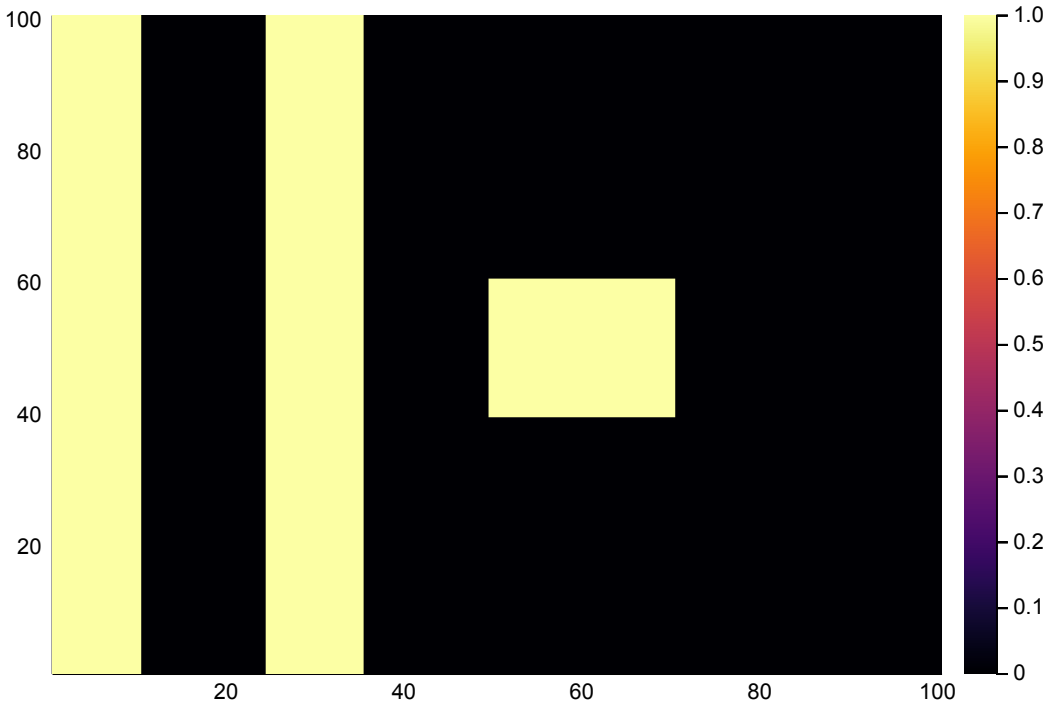
- [ ] `x = ones(100)` and `y = [ones(10); zeros(15); ones(10); zeros(65)]`.
- [x] `x = ones(100)` and `y = [ones(10); zeros(14); ones(11); zeros(65)]`.

Submit

In [5]:
```
A = [ (1 <= j <= 10) || (25 <= j <= 35) ||
    (40 <= i <= 60 && 50 <= j <= 70) ? 1 : 0
    for i in 1:100, j in 1:100 ]
heatmap(A)
```

Run

Out[5]:



`A` can be expressed as `x1 * y1' + x2 * y2'` where (select **all** that apply)

- [x] `x1 = ones(100)` and `y1 = [(25 <= j <= 35) || (1 <= j <= 10) ? 1 : 0 for j in 1:100]`.
- [ ] `x2 = ones(100)` and `y2 = [50 <= j <= 70 ? 1 : 0 for j in 1:100]`.
- [ ] `x1 = ones(100)` and `y1 = [50 <= j <= 70 ? 1 : 0 for j in 1:100]`.
- [x] `x2 = [(40 <= i <= 60) ? 1 : 0 for i in 1:100]` and `y2 = [50 <= j <= 70 ? 1 : 0 for j in 1:100 ]`.

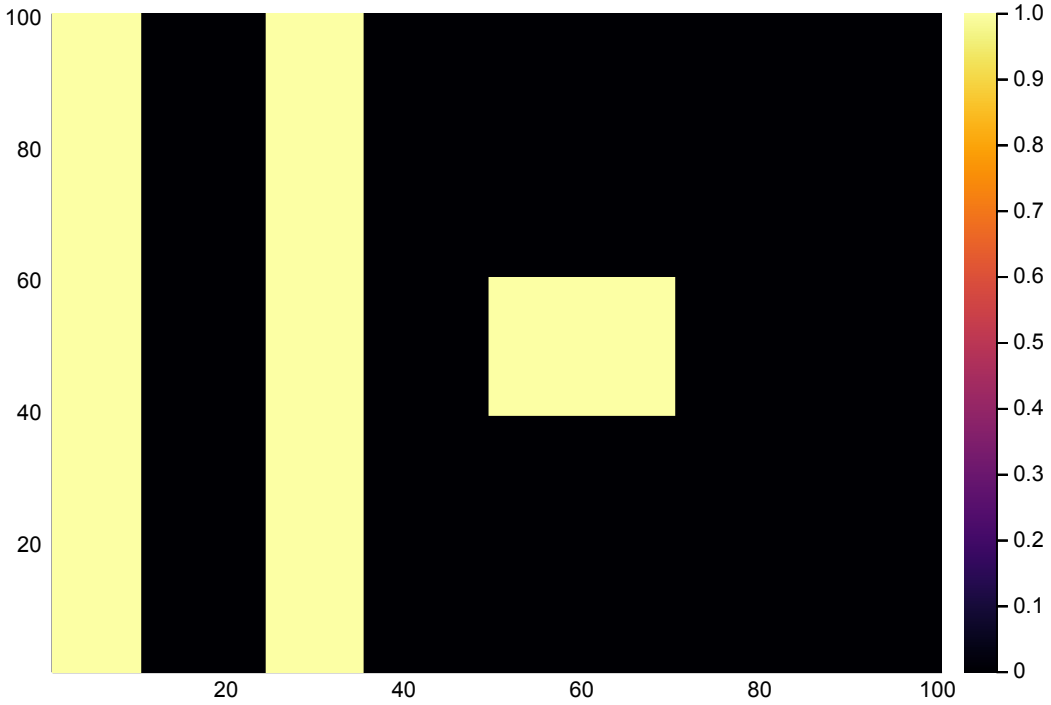
Submit

Indeed! Let us check it just to verify.

In [6]:
```
x1 = ones(100)
y1 = [(25 <= j <= 35) || (1 <= j <= 10) ? 1 : 0 for j in 1:100]
x2 = [(40 <= i <= 60) ? 1 : 0 for i in 1:100]
y2 = [50 <= j <= 70 ? 1 : 0 for j in 1:100]
Aalt = x1 * y1' + x2 * y2'
heatmap(Aalt)
```

▶ Run

Out[6]:



We can compute the Frobenius norm of the difference between the two matrices using the command `vecnorm` as in the next cell.

In [7]:
```
@show norm(A - Aalt);
```

▶ Run

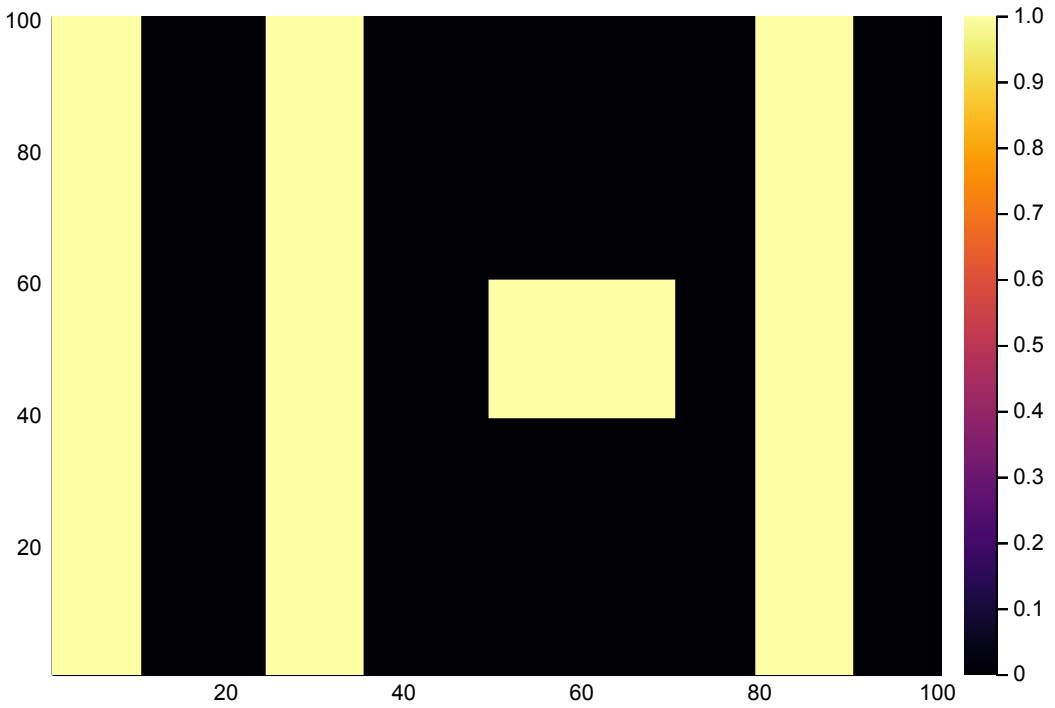
```
norm(A - Aalt) = 0.0
```

A few more exercises and we will be ready for the main point of this section -- which is to use the SVD extract low-rank matrices!

In [8]:
```
A = [(1 <= j <= 10) || (25 <= j <= 35) ||
     (40 <= i <= 60 && 50 <= j <= 70) ||
     (80 <= j <= 90) ? 1 : 0
     for i in 1:100, j in 1:100]
heatmap(A)
```

▶ Run

Out[8]:



What is the rank of A?

- [ ] 1
- [x] 2
- [ ] 3
- [ ] 4

Submit

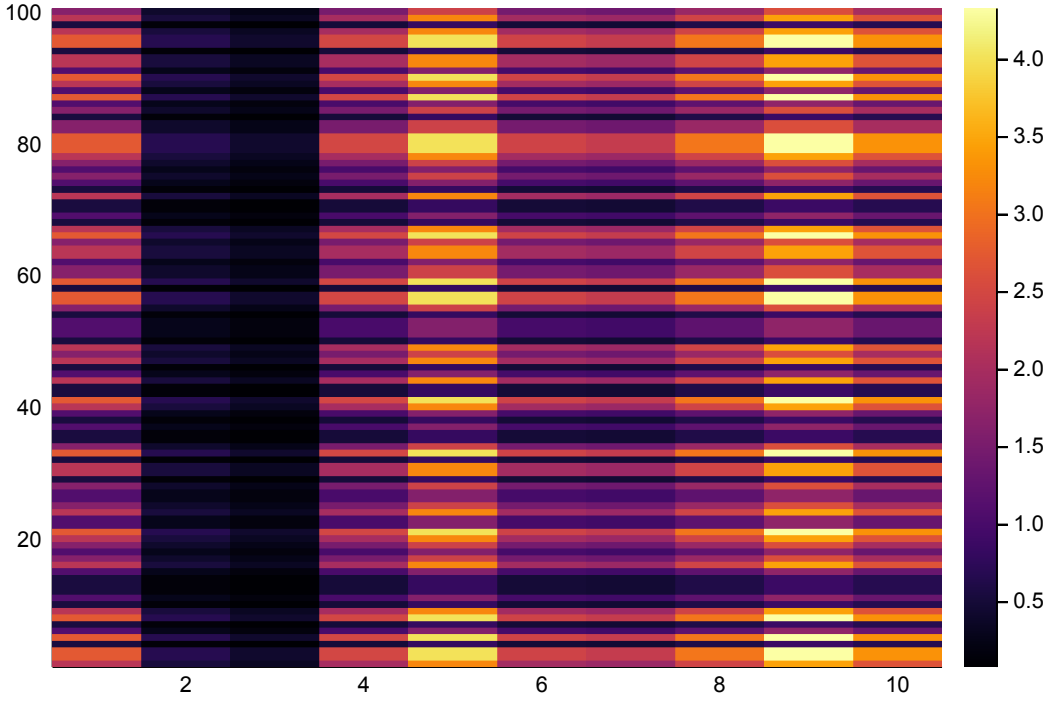
Indeed! The matrix A can be written as $x_1 y_1^T + x_2 y_2^T$. The x_1 and x_2 vectors are same as earlier. Only the y_1 entry changes -- we add extra ones in the indices of y_1 corresponding to the columns where the new "stripe" appears.

Now consider the matrix in the following cell.

In [9]:
```
x = rand(1:5, 100)
y = rand(1, 100)
A = [y[j] * x[i] for i in 1:100, j in 1:10]
heatmap(A)
```

▶ Run

Out[9]:



What is the rank of A?

- [ ] 100, because there are 100 rows.
- [ ] 10, because there are 10 columns.
- [x] 1, because the matrix can be written as xy^T, where elements of y are random numbers given by `rand`.

Submit

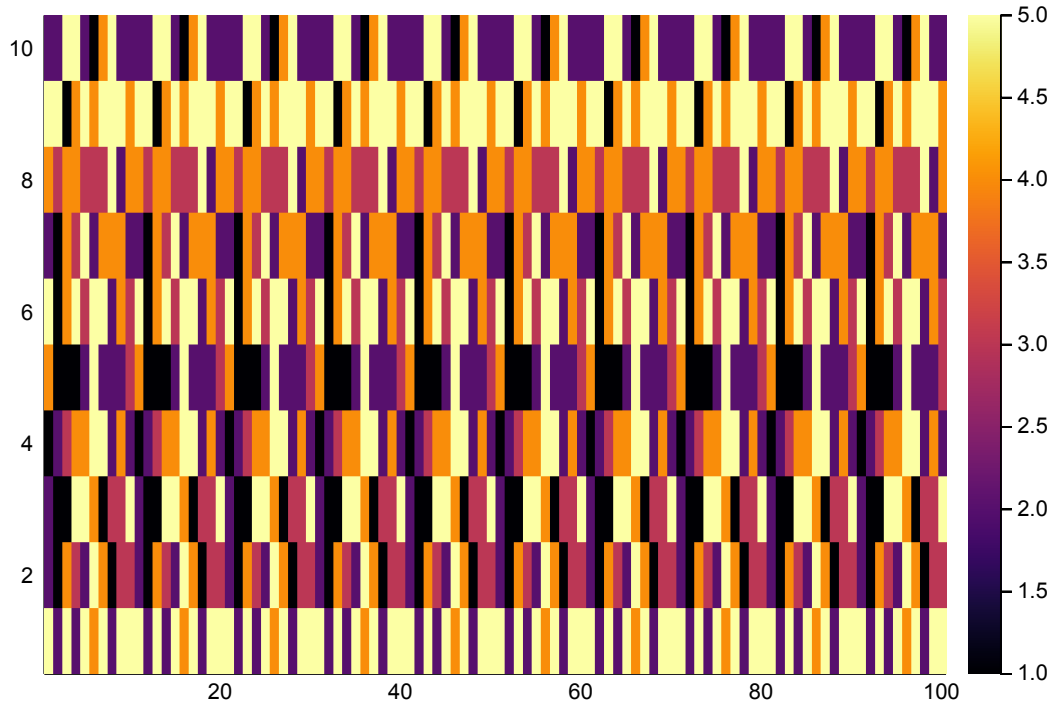
2.2 Spotting an *induced* rank-one matrix

Consider the matrix B in the following cell. It is obtained by concatenating copies of the the matrix X alongside each other.

In [10]:
```
X = rand(1:5, 10, 10)
B = repeat(X; outer=(1, 10))
heatmap(B)
```

▶ Run

Out[10]:



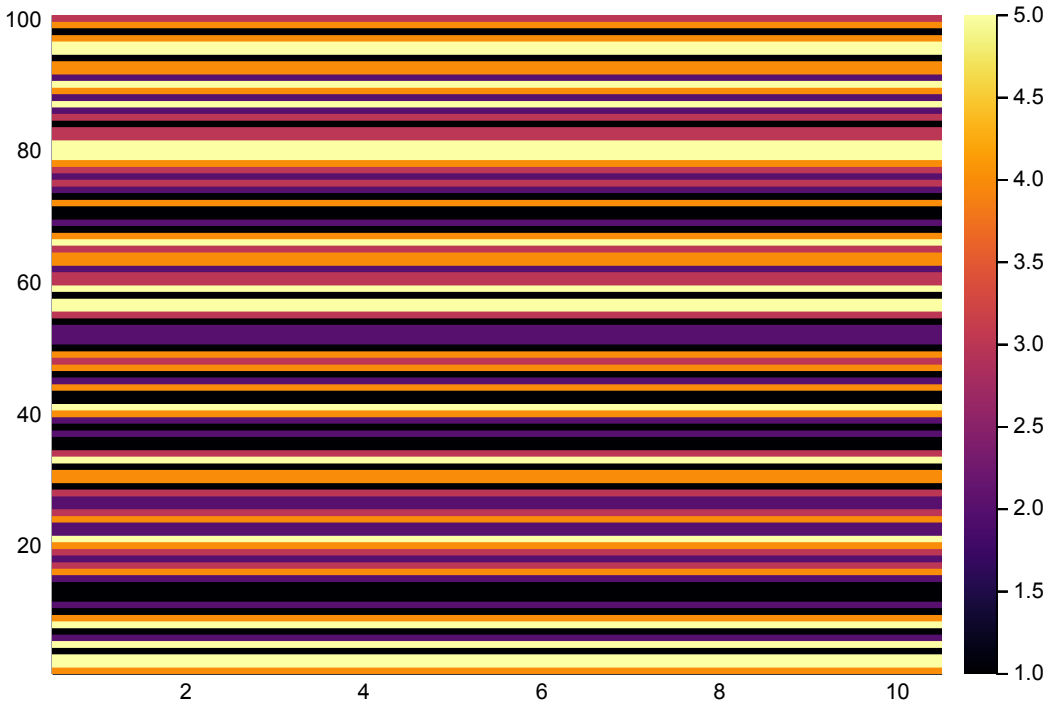
The columns of X are drawn at random, so generically (with very high probability) the rank of X will equal the number of columns. Thus, the matrix B will have as many linearly independent columns as the number of linearly independent columns of X.

Now consider the matrix A formed in the next cell.

In [11]:
```
A = repeat(x[:]; outer=(1, 10))
heatmap(A)
```

▶ Run

Out[11]:



A and B have the same elements of A, but in a different order. The rank of A is

- ☐ equal to the rank of B, regardless of the rank of B.
- ☑ 1, even when the rank of B is higher.

Submit

In this case, even though the matrix B did not have rank one, we **induced a low-rank matrix** A by reshaping the elements of B in an appropriate manner. This will be important for understanding why the SVD application we describe next works the way it does.

3 Application: Background subtraction using the SVD

SVD decomposes a matrix into a sum of rank-one outer product matrices. This powerful representation enables a seemingly magical feat when applied to stable video footage.

We will illustrate with a surveillance video, currently stored as a `.jld` file. The command `JLD.load` produces a Dictionary of the variables in the file.

In [12]:
```
1  load("lobby.jld")
```

▶ Run

Out[12]:
```
Dict{String,Any} with 1 entry:
  "movie_cube" => Float16[0.3452 0.341 … 0.2627 0.2588; 0.447 0.443 … 0.3
254 0.…
```

The command `load` is used to open a `.jld` formatted file. We can access a specific variable, `movie_cube` in this case, as follows:

In [13]:
```
1  # TODO: Fill in the ?? with the appropriate variable name from
2  # the Dictionary
3  file = load("lobby.jld")
4  varname = "movie_cube"
5
6  movie_cube = file[varname]
7
8  # data are compressed using 16-bit containers;
9  # we prefer 64-bit for computation
10 movie_cube = Float64.(movie_cube)
11
12 # alternate way to load variable:
13 # movie_cube = load("lobby.jld", varname)
14
15 @show size(movie_cube)
16 @show typeof(movie_cube);
```

▶ Run

```
size(movie_cube) = (128, 160, 650)
typeof(movie_cube) = Array{Float64,3}
```

The `movie_cube` variable is of type

- ☐ `Array{Float64,1}` -- i.e. `Vector{Float64}`
- ☐ `Array{Float64,2}` -- i.e. `Matrix{Float64}`
- ☑ `Array{Float64,3}` -- it's a "data cube"

Submit

The `movie_cube` datacube is composed of 650 frames of a video, where each frame is a 128-by-160 image. We will scale the elements of the datacube so that they lie between 0 and 1, then display the first frame of this video.

In [14]:
```
"""
    B = rescale_zero_one(A)
Return a shifted and scaled version of input array `A` where all
elements are in the range [0, 1].
"""
function rescale_zero_one(A::Array)
    B = float(A)
    B .-= minimum(B)
    B /= maximum(B)
    return B
end

movie_cube = rescale_zero_one(movie_cube)

# display first frame
Gray.(movie_cube[:, :, 1])
```

▶ Run

Out[14]:

Use the cell below to explore different frames of the video.

In [15]:
```
function plotframe(X; kw_args...)
    return heatmap(
        X;
        color=:grays,
        yflip=true,
        aspect_ratio=:equal,
        border=false,
        ticks=[],
        axis=false,
        colorbar=false,
        kw_args...,
    )
end
```

▶ Run

Out[15]: plotframe (generic function with 1 method)

In [16]:
```
nframes = size(movie_cube, 3)
@manipulate for frame = 100
    plotframe(
        movie_cube[:, :, frame],
        title="Frame $frame of $(nframes)",
        clim=(0, 1)
    )
end

# Enter values in the box below to explore the video
```

▶ Run

Out[16]:

frame 100

Frame 100 of 650

Let us now play the matrix as a movie:

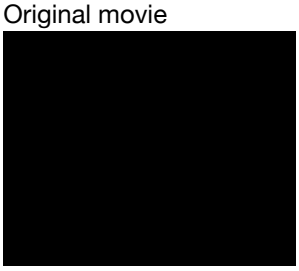
In [17]:
```
include("./webplayer.jl")
playvideo(
    [movie_cube], ["Original movie"];
    frames_per_second=120
)
# move the slider to scan through the movie
```

Run

Out[17]:

325

Original movie



What happens in the movie? (select **all** that apply)

- [x] There are people walking around.
- [x] The lights go on and off.
- [x] The camera angle is fixed.
- [x] Around frame 182, four lights are off and just one remains on.
- [x] Around frame 450, the four lights that were off come back on.
- [ ] All the lights stay on throughout the scene.
- [x] The middle light in the background stays on through the scene.

Submit

3.1 Decomposing the movie matrix with the SVD

We will now use the SVD to decompose the movie. The movie is stored as an $m \times n \times t$ 3-D array, where

- m is the number of rows of pixels in each frame,
- n is the number of columns of pixels in each frame, and
- t is the number of frames in the movie

To decompose using the SVD, we first reshape the movie array into an $mn \times t$ **matrix** A. Each column of the matrix corresponds to a frame (and contains mn pixel values).

We then compute the rank-k truncated singular value decompositon of A. In other words, if A has rank r and has SVD

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T,$$

then A_k, the rank-$k \leq r$ truncted SVD of A, is

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T,$$

and $(A - A_k)$ is the residual matrix. We will compute the residual $(A - A_k)$ directly, but it is good to observe that it has SVD

$$A - A_k = \sum_{i=1}^{r} \sigma_i u_i v_i^T - \sum_{i=1}^{k} \sigma_i u_i v_i^T = \sum_{i=k+1}^{r} \sigma_i u_i v_i^T.$$

In the cell below, write a function `svdize_cube` that takes as input the $m \times n \times t$ movie, and a rank k. The function should

1. reshape the movie into an $mn \times t$ matrix
2. compute the truncated SVD and residual
3. reshape both truncated SVD and residual into $m \times n \times t$ arrays
4. return the reshaped truncated SVD, reshaped residuals, and singular values

Hint 1: Recall that the above sum of outer products is equivalent to

$$A_k = U_k \Sigma_k V_k^T$$

where $U_k = [u_1 \cdots u_k]$ is a matrix of the first k left singular vectors, $V = [v_1 \cdots v_k]$ is a matrix of the first k right singular vectors, and $\Sigma = \text{diagm}(\sigma_1, \ldots, \sigma_k)$ is a diagonal matrix of the first k singular values.

Hint 2: The matrices U_k, Σ_k and V_k can be obtained using `svds` as

```
U, s, V = svds(A; nsv=k)[1]
```

In [18]:

```julia
using LinearAlgebra
using Arpack: svds

"""
    cube_k, cube_residual, sk = svdize_cube(cube, k)

Inputs:
* `cube` is an m x n x l array
* `k` is an integer

Outputs:
* `cube_k` is an m x n x l 3-D array
* `cube_residual` is an m x n x l 3-D array
* `sk` is a length k vector

Given a 3-D array `cube` and a rank `k`, return a (rank `k`)
truncated SVD version of the cube as `movie_cube_k`, the resiudal
as `cube_residual`, the right singular vectors as `Vk`,
and the singular values as `sk`.

*Hint: svds has a keyword argument `nsv` for the number of singular
values. To specify this argument, you use the keyword:
`svds(..., nsv=...)`*
"""
function svdize_cube(cube, k)
    # Reshape 3-D array into matrix where each column is a frame
    m, n, l = size(cube) # cube is an m x n x l 3-D array
    matrix = reshape(cube, m*n, l) # Convert to mn x l matrix

    # Compute rank k truncated SVD and residual
    UsV = svds(matrix; nsv=k)[1]
    Uk = UsV.U
    sk = UsV.S
    Vk = UsV.Vt

    matrix_k = Uk * Diagonal(sk) * Vk
    matrix_residual = matrix - matrix_k

    # Reshape back into 3-D m x n x numFrames arrays
    cube_k = reshape(matrix_k, m, n, l)
    cube_residual = reshape(matrix_residual, m, n, l)

    return cube_k, cube_residual, sk
end
```

Revert Submit

Out[18]: svdize_cube

What does the movie represented by A_1 look like? Let's find out! First we compute A_1 using the function we just defined.

In [19]:
```
# Desired rank
k = 1
# Compute truncated SVD-version of movie
movie_cube_k, movie_cube_residual, _ = svdize_cube(movie_cube, k)

movie_cube_k = rescale_zero_one(movie_cube_k)
movie_cube_residual = rescale_zero_one(movie_cube_residual);
```

▶ Run

Done! Now let's examine the frames of the rank-k approximation!

In [20]:
```
nframes = size(movie_cube, 3)
@manipulate for frame = 325
    plotframe(
        movie_cube_k[:, :, frame],
        title = "Rank $k approx: Frame $frame of $(nframes)",
        clim=(0, 1)
    )
end
# Enter values in the box below to explore the video
```

▶ Run

Out[20]:

frame 325

Rank 1 approx: Frame 325 of 650

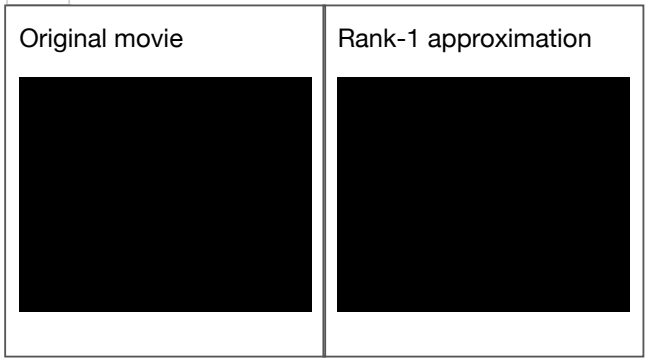
Perhaps you're seeing that something unexpected (it was to me, too!) has happened. Let's see the original movie and the movie of the rank-k approximation side-by-side.

In [21]:
```
playvideo(
    [movie_cube, movie_cube_k],
    ["Original movie","Rank-$k approximation"],
    frames_per_second=120
)
```

Run

Out[21]:

325



What all happens in the rank-1 approximation movie? (select **all** that apply)

- [ ] There are people walking around, as in the original movie .
- [x] Lights go on and off.
- [x] The middle light in the background stays on throughout the scene.
- [ ] The four lights go from on to off, then come back on and go completely dark as before.
- [x] The four lights go from on to off, then come back on, but faded (they do not go as dark as before).

Submit

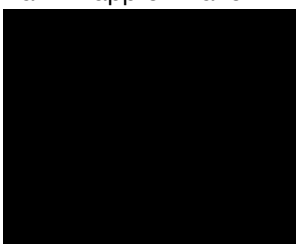
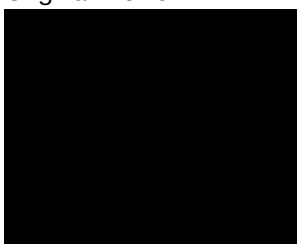
Where did the people go? Let us examine the residual - after all, `movie_cube = movie_cube_k + movie_cube_residual` !

In [22]:
```
playvideo(
    [movie_cube_k, movie_cube_residual, movie_cube],
    ["Rank-$k approximation", "Residual", "Original Movie"],
    frames_per_second=120
)
```

Run

Out[22]:

325

Rank-1 approximation	Residual	Original Movie
		

Let's take a closer look at a fixed frame in each movie.

In [23]:
```
# An interesting frame to compare
frame = 250

hbox(
    plotframe(
        movie_cube[:, :, frame],
        title = "Original movie: frame = $frame",
        clim=(0, 1)
    ),
    plotframe(
        movie_cube_k[:, :, frame],
        title="Rank-$k approx: frame = $frame",
        clim=(0, 1)
    ),
    plotframe(
        movie_cube_residual[:, :, frame],
        title="Rank-$k residual: frame = $frame",
        clim=(0, 1)
    ),
)
```

Run

Out[23]:

Original movie: frame = 250

Rank-1 approx: frame = 250

Rank-1 residual: frame = 250

In [24]:

```
frame = 20

hbox(
    plotframe(
        movie_cube[:, :, frame],
        title = "Original movie: frame = $frame",
        clim=(0, 1)
    ),
    plotframe(
        movie_cube_k[:, :, frame],
        title="Rank-$k approx: frame = $frame",
        clim=(0, 1)
    ),
)
```

▶ Run

Out[24]:

Original movie: frame = 20

Rank-1 approx: frame = 20

In [25]:

```
frame = 97

hbox(
    plotframe(
        movie_cube[:, :, frame],
        title = "Original movie: frame = $frame",
        clim=(0, 1)
    ),
    plotframe(
        movie_cube_k[:, :, frame],
        title="Rank-$k approx: frame = $frame",
        clim=(0, 1)
    ),
)
```

▶ Run

Out[25]:

Original movie: frame = 97

Rank-1 approx: frame = 97

In [26]:

```
frame = 336

hbox(
    plotframe(
        movie_cube[:, :, frame],
        title = "Original movie: frame = $frame",
        clim=(0, 1)
    ),
    plotframe(
        movie_cube_k[:, :, frame],
        title="Rank-$k approx: frame = $frame",
        clim=(0, 1)
    ),
)
```

▶ Run

Out[26]:

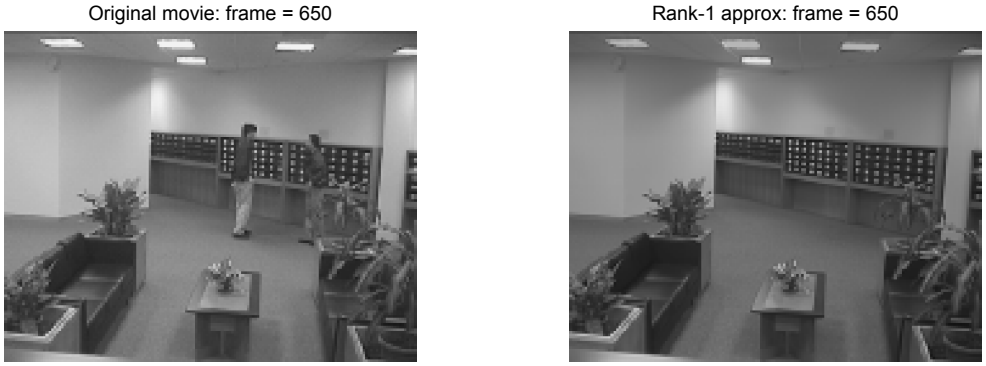
Original movie: frame = 336

Rank-1 approx: frame = 336

In [27]:
```
frame = 650

hbox(
    plotframe(
        movie_cube[:, :, frame],
        title = "Original movie: frame = $frame",
        clim=(0, 1)
    ),
    plotframe(
        movie_cube_k[:, :, frame],
        title="Rank-$k approx: frame = $frame",
        clim=(0, 1)
    ),
)
```

Run

Out[27]:



What does the rank-1 component capture?

- [x] The "background" parts of the scene not containing people .
- [ ] The people.

Submit

3.2 What information do the right singular vectors represent?

We will soon discuss why the rank-1 component captures the background. It has to do with the fact that there is a low-rank matrix hiding inside the video, if we view things just right. Recall that the SVD returns the left and right singular vectors. So we may deduce the structure of the low-rank matrix hiding inside this movie.

Let us first try to understand what the right singular vectors of the decomposed movie convey. That will give us the important clue for unlocking this puzzle; once we do, we'll start seeing (induced) low-rank matrices (almost) everywhere!

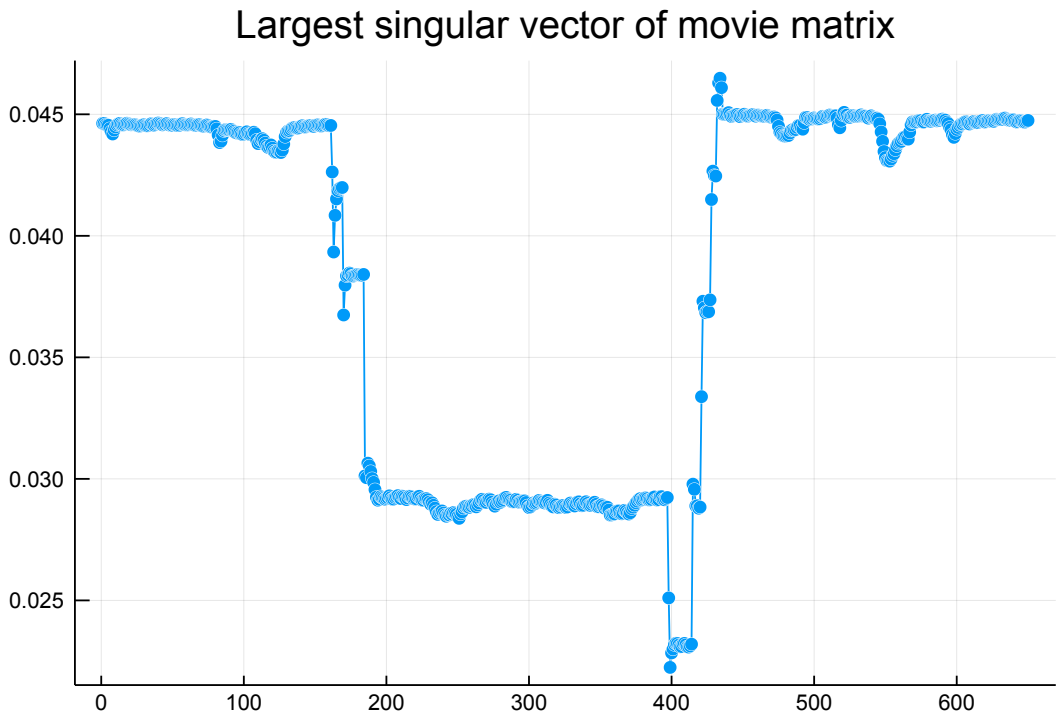
We begin by examining first right singular vector, $V[:, 1]$, more closely.

In [28]:

```
# Largest right singular vector
UsV = svds(reshape(movie_cube, :, size(movie_cube, 3)), nsv=1)[1]

# Generically, the sign of V is arbitrary, but it's easier to interpr
# our case) if the first element is positive.
positivify(V) = sign(V[1, 1]) * V
V = positivify(UsV.V)

plot(
    V[:, 1],
    linestyle=:solid,
    marker=:circle,
    title="Largest singular vector of movie matrix",
    legend=:bottomleft
)
```

▶ Run

Out[28]:



`length(V[:, 1])` equals? (select **all** that apply)

- [x] the number of frames in the movie.
- [x] 650.
- [ ] 128 x 160 -- the total number of elements in each frame of the video.

Submit

`length(U[:, 1])` equals? (select **all** that apply)

- [ ] 650.
- [x] 128 x 160.

☑ the total number of elements in each frame (once it has been reshaped into a column vector).

Submit

Do you notice something unusual in the behavior around elements 194 and 450? Recall that the A matrix has pixel indices on the rows and time on the columns. Thus, V encodes time behavior, and has dimensions equal to the number of frames in the movie. Let us scroll through the movie one frame at a time to see if we can identify the physical phenomenon that occurs in the movie when the elements of $V[:, 1]$ "jump" (up or down).

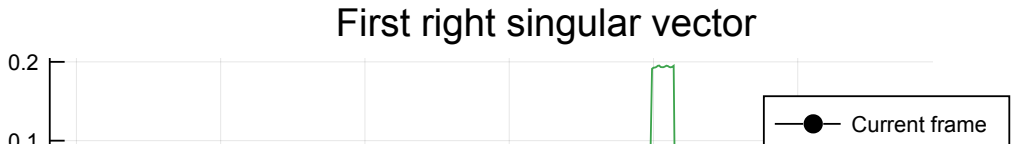
In [29]:

```
joinframes(frame::Integer, vids::Tuple) = hcat((vid[:, :, frame] for 

@manipulate for frame = (100, 194, 397, 450)
    # Plot first right singular vector
    p1 = plot(
        V,
        linestyle=:solid,
        title="First right singular vector"
    )

    # Denote current frame
    plot!(
        (frame, V[frame, 1]),
        marker=:circle, # current point
        color=:black,
        label="Current frame"
    )

    # Plot movie frames
    p2 = plotframe(
        joinframes(
            frame,
            (movie_cube,
            movie_cube_k,
            movie_cube_residual)
        ),
        title="Movie | Rank-k approx. | Residual",
        clim=(0, 1)
    )
    vbox(plot(p1, p2; layout=Plots.grid(2, 1, heights=[0.5, 0.5])))

end

# Change frame to view different frames
```

▶ Run

Out[29]:

frame 100 194 397 450



Frames 194 and 450 correspond (roughly) to the times when (select **all** that apply)

- [x] The four lights go from off to on.
- [x] The middle light in the background stays on.
- [ ] The four lights remain on.
- [ ] Nothing changes.

Submit

The right singular vector is encoding the "on-off-on" behavior of the front row of lights! Note that all elements of $V[:, 1]$ are positive, so the "off" portion corresponds to 0 in the approriate segment of the vector.

Let us now compare the low-rank decompositions obtained by setting $k = 1$ versus $k = 2$, as in the next cell. When $k = 2$, we will also plot `V[:, 2]` and try to understand what it captures.

In [30]:
```
# one-time computation

nframes = size(movie_cube, 3)

back_1, res_1, _ = svdize_cube(movie_cube, 1)
back_2, res_2, _ = svdize_cube(movie_cube, 2)

V_1 = positivify(svds(reshape(movie_cube, :, nframes); nsv=1)[1].V)
V_2 = positivify(svds(reshape(movie_cube, :, nframes); nsv=2)[1].V);
```

Run

In [31]:

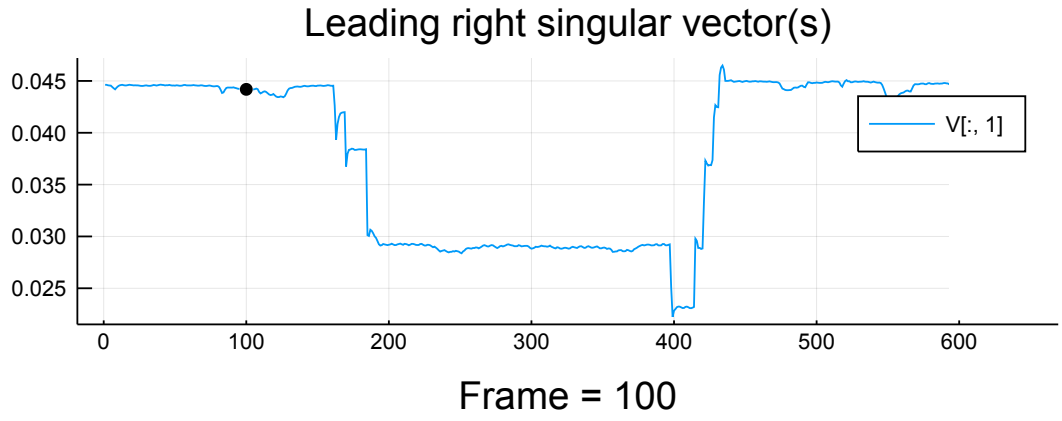
```
# interactive display
@manipulate for k = (1, 2), frame = (100, 194, 397, 450)
    if k == 1
        back_k = back_1
        res_k = res_1
        V_k = V_1
    elseif k == 2
        back_k = back_2
        res_k = res_2
        V_k = V_2
    end

    # Plot right singular vector(s)
    p1 = plot(
        V_k;
        linestyle=:solid,
        title="Leading right singular vector(s)",
        label=["V[:, 1]", "V[:, 2]"]
    )

    # Denote current frame
    for kk in 1:k
        plot!(
            (frame, V_k[frame, kk]);
            marker=:circle, # current point
            color=:black,
        )
    end

    # Plot movie frames
    p2 = plotframe(
        joinframes(
            frame,
            (movie_cube,
            back_k,
            res_k)
        ),
        title="Frame = $frame",
        clim=(0, 1)
    )

    plot(
        p1, p2;
        layout=Plots.grid(2, 1; heights=[0.5, 0.5])
    )
end

# Change k to use a different rank approximation
# Change frame to view different frames
```

Out[31]:

▶ Run

k 1 2

frame 100 194 397 450



Now let's plot the individual frames of the decomposition for $k = 2$.

In [32]:
```
# Choose rank
k = 2

# Decompose movie
movie_cube_k, movie_cube_residual, _ = svdize_cube(movie_cube, k);
UsV = svds(reshape(movie_cube, :, size(movie_cube, 3)), nsv=k)[1]
V = positivify(UsV.V);
```

▶ Run

In [33]:
```
# An interesting frame
frame = 250

hbox(
    # Original movie
    plotframe(
        movie_cube[:, :, frame],
        title="Original movie: frame $frame",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_k[:, :, frame],
        title="Rank $k approx: frame $frame",
        clim=(0, 1)
    ),
    # Residual
    plotframe(
        movie_cube_residual[:, :, frame],
        title="Rank $k residual: frame $frame"
    )
)
```

▶ Run

Out[33]:

Compared to $k = 1$, the $k = 2$ low-rank decomposition (select **all** that apply)

- ✔ better captures the on-off-on behavior of the row of four lights.
- ✔ captures more accurately the portion of the movie where only one light is on.
- ✔ has a residual (with the people moving around) with a higher image contrast.

Submit

Setting $k = 2$ does indeed better capture the background than $k = 1$. Particularly we are better able to capture the on-off-on behavior of the row of lights in the front. As before, the people have been removed in the low rank approximation and instead appear in the residual matrix. The movie in the next cell shows that this is true for all the frames.

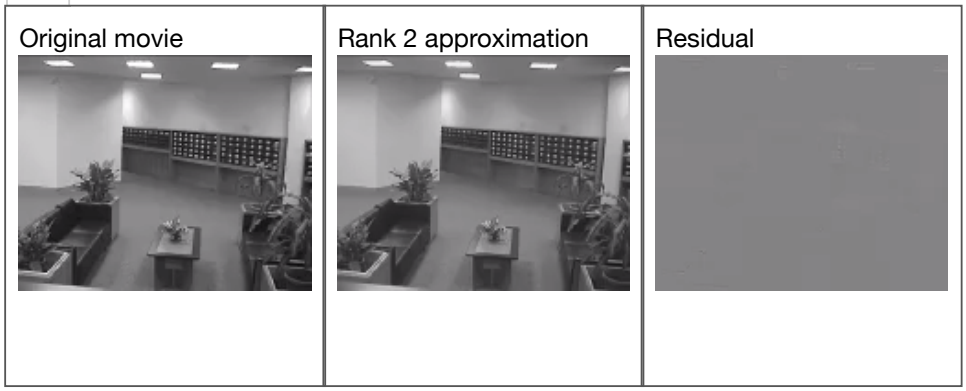
In [34]:
```
1  println("Original movie vs Rank $k approximation vs Residual")
2  playvideo(
3      [movie_cube, rescale_zero_one(movie_cube_k), rescale_zero_one(mov
4      ["Original movie", "Rank $k approximation", "Residual"], frames_p
5  )
```

▶ Run

```
Original movie vs Rank 2 approximation vs Residual
```

Out[34]:

325

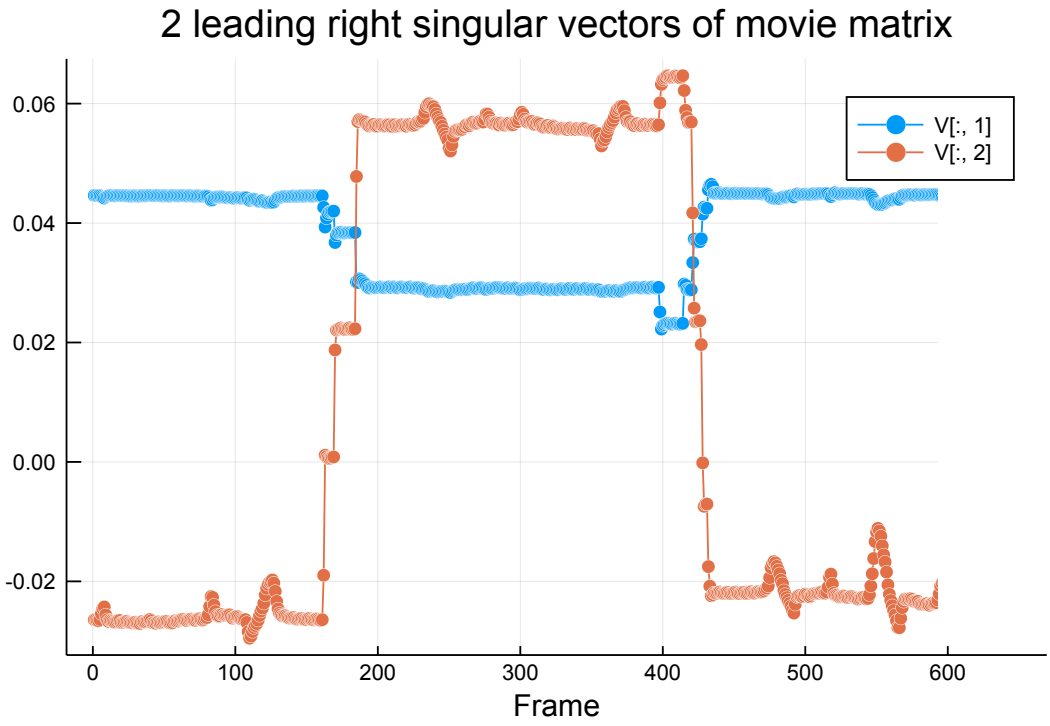


Let us look at the top two right singular vectors using the command in the next cell.

In [35]:
```
plot(
    V,
    label=["V[:, $i]" for i in collect(1:k)'],
    title="$k leading right singular vectors of movie matrix",
    xaxis="Frame",
    marker=:circle,
)
```

Run

Out[35]:



Notice the step-function-like behavior in the singular vectors! The elements of both the first and second singular vectors jump up (or down) at around index 195 and back (or up) at around index 420.

These time instances correspond to the portions of the movie where the front lights lights go from on to off and then from off to on. We verify this observation using by slicing through the video around these frames as in the next cell.

In [36]:
```
@manipulate for frame = (150, 196, 300, 420, 500)
    plotframe(
        movie_cube[:, :, frame],
        title = "Original movie: frame = $frame",
    )
end

# Change frame to view different frames
```

▶ Run

Out[36]:

frame 150 196 300 420 500

Original movie: frame = 150

What would `movie_cube_k` capture if we set `k = r`, where `r = rank(movie_cube)` ? (select **all** that apply)

- [ ] Increasing `r` decreases the error, so we will better separate the background from the foreground (people moving).

- [x] `movie_cube_k = movie_cube` by the definition of the SVD, so `movie_cube_residual = 0` , and we would not be doing any background subtraction

- [x] Increasing `r` monotonically decreases the error between `movie_cube_k` and `movie_cube` , but what we are really interested in is the error between `movie_cube_k` and the background component of the scene.

3.3 The low-rank matrix hiding inside the video

There are two main things happening in the video that can be captured via a low-rank matrix.

Let X_1 represent the 128 x 160 matrix (or image) corresponding to what the room looks like when only the middle light in the background is on, and the four lights in the foreground are off. Let x_1 denote the vector formed by reshaping X_1 into a 128 x 160 dimensional vector.

Let X_2 represent the 128 x 160 matrix corresponding to what the room looks like when only the four lights in the forgeround are on. Let x_2 denote the vector formed by reshaping X_2 into a 128 x 160 dimensional vector.

See below for what X_1 (left) and X_2 (right) look like.

X1

X2

Let B_t denote the 128 x 160 matrix corresponding to the background at time instance t. Then assuming a linear illumination model, we can express B_t as

$$B_t = y_{1t} X_1 + y_{2t} X_2, \tag{1}$$

where y_{1t} denotes the relative intensity of the middle light in the background, and y_{2t} denotes the relative intensity of the foreground row of four lights. If $y_{1t} = 0$, then only the lights in the foreground are on; if $y_{2t} = 0$, then only the light in the background is on.

Let us see what various linear combinations of X_1 and X_2 look like. We'll vary y_1 and y_2 and see what happens.

In [37]:
```
frontback = load("front_back.jld")
front = frontback["front"]
back = frontback["back"]

@manipulate for y1 in (1.0, 0.5, 0.0), y2 in (0.0, 1.0)
    plotframe(y1 * back + y2 * front)
end
```

Out[37]:

▶ Run

y1 1.0 0.5 0.0

y2 0.0 1.0

The background consists of a scene where the middle light is always on, and the four lights towards the front go on and off. Which of the following statements accurately connects matrix language to what we can see with our eyes? (select **all** apply)

- [ ] `y1 = 0, y2 = 0` means only the middle light towards the back is on.
- [x] `y1 = 1, y2 = 0` means only the middle light towards the back is on.
- [ ] `y1 = 1, y2 = 1` means only the front four lights are on.
- [x] `y1 = 0, y2 = 1` means only the front four lights are on.
- [x] `y1 = 1, y2 = 1` corresponds to all the lights being on.

Submit

Let b_t denote the vector formed by reshaping B_t into a 128 x 160 vector. Then from Eq. (???) we have that

$$b_t = y_{1t}x_1 + y_{2t}x_2 = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} y_{1t} \\ y_{2t} \end{bmatrix}.$$

Let B denote the $mn \times T$ matrix formed by stacking $b_1, \ldots, b_T$ alongside each other. Then

$$B = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} y_{11} & \cdots & y_{1T} \\ y_{21} & \cdots & y_{2T} \end{bmatrix} = x_1 y_1^T + x_2 y_2^T,$$

where y_1 and y_2 are T dimensional vectors defined as

$$y_1 = \begin{bmatrix} y_{11} & \ldots & y_{1T} \end{bmatrix}^T,$$

and

$$y_2 = \begin{bmatrix} y_{21} & \ldots & y_{2T} \end{bmatrix}^T,$$

The final step in uncovering this rank-2 background matrix hidden inside the video is assigning (relative) values to the elements of y_1 and y_2 so we can capture the dynamics of how the lights switch on or off.

Suppose x_1 and x_2 denote the components of the scene described above. Then a plausible set of values for y_1 and y_2 (that captures the dynamics but not actual intensities of lights) is

- [ ] $y_1 = [1, 1, \ldots, 1]^T, y_2 = [0, 0, \ldots, 1, \ldots, 1, 0, \ldots, 0]^T$
- [x] $y_1 = [1, 1, \ldots, 1]^T, y_2 = [1, 1, \ldots, 0, \ldots, 0, 1, \ldots, 1]^T$
- [ ] $y_1 = [1, 1, \ldots, 0, \ldots, 0, 1, \ldots, 1]^T \; y_2 = [1, 1, \ldots, 1]^T$

Submit

Indeed! If y_1 is all ones, the middle light will be on all the time, which matches what we observe. And if y_2 is a series of ones, followed by zeros, followed by ones again, then the front four lights will go from on to off and then back on.

We could vary the elements in y_1 and y_2 with time (instead of setting them to either 1 or 0) to encode variation in the intensity of the light; this would not change the rank of the resulting background matrix.

The presence of this underlying latent (or hidden) rank-2 matrix inside the video is why the rank-two approximation better captures the dynamics of the background. Note that the SVD has captured a dynamic, *time-varying* background. We could have computed a mean matrix by averaging all the images int the video. That would remove the people, but it would be static and would **not** capture the behavior of the lights!

Spotting this low-rank behavior allowed us to use the SVD to powerful effect.

What do we not yet fully understand? (select **all** that apply)

- [x] If the people are in the residual matrix, and we can think of the movie matrix as background + people, then why are the singular vectors of the movie matrix so close to the singular vectors of the people-free background matrix?

- [x] Can we determine the rank of the background matrix automatically, without going through the reasoning process we just did?
- [x] Would we have recovered the background matrix if noise was added to the movie?
- [x] Would we have recovered the background matrix if we randomly zeroed out (or deleted) pixels in the movie?
- [x] If we continually add noise and delete entries, at what point will we become unable to accurately recover the low-rank background matrix?
- [ ] Everything is illuminated -- there are no questions left to ask!
- [x] The y_1 and y_2 we modeled are linearly independent but not orthogonal, whereas the right singular vectors from the SVD *are* orthogonal. Is there a technique for expressing the movie matrix as a sum of outer products with non-orthogonal components that might model y_1 and y_2 better?
- [x] I am still confused, but at a higher level, about more important questions!

Submit

We will soon discuss insights from random matrix theory and matrix perturbation theory that address all of the above questions! We will also discuss other algorithms for decomposing (or factorizing) a matrix.

4 Summary

We saw how the SVD allows us to capture low-rank matrices. The ability to spot low rank matrices is a critical skill in computational data science. Rearrangement of the data can induce low rank structure, which the SVD can then expose.

5 Additional Exercises

5.1 Comparing the $k = 2$ vs $k = 3$ decompositions

Perform a truncated low rank decomposition of the video, as above, except with $k = 3$. Display the video of the original and residual next to each other, and compare it to the setting when $k = 2$.

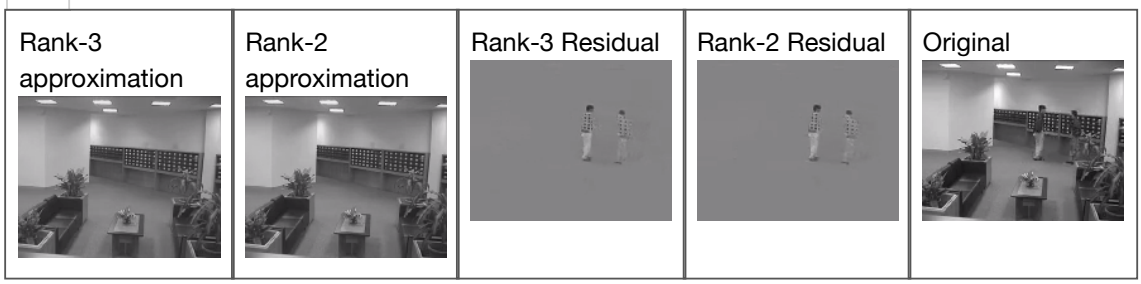
In [74]:

```
## TODO: Compare videos of the k = 2 vs k = 3 reduced-rank videos.

movie_cube_2, movie_cube_residual_2, _ = svdize_cube(movie_cube, 2)
movie_cube_2 = rescale_zero_one(movie_cube_2)
movie_cube_residual_2 = rescale_zero_one(movie_cube_residual_2);

movie_cube_3, movie_cube_residual_3, _ = svdize_cube(movie_cube, 3)
movie_cube_3 = rescale_zero_one(movie_cube_3)
movie_cube_residual_3 = rescale_zero_one(movie_cube_residual_3);

playvideo(
    [movie_cube_3, movie_cube_2, movie_cube_residual_3, movie_cube_re
    ["Rank-3 approximation", "Rank-2 approximation","Rank-3 Residual"
    frames_per_second=120
)
```

▶ Run

Out[74]:

650



In [39]:
```
## TODO: Compare representative frames of the k = 2 vs k = 3
# reduced-rank videos. Hint: use @manipulate
@manipulate for frame = (150, 196, 300, 420, 500)
hbox(
    # Original movie
    plotframe(
        movie_cube_2[:, :, frame],
        title="Rank 2 approx: frame $frame",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_3[:, :, frame],
        title="Rank 3 approx: frame $frame",
        clim=(0, 1)
    ),
)
end
```

Out[39]:

▶ Run

frame 150 196 300 420 500

Rank 2 approx: frame 150

Rank 3 approx: frame 150

Which better captures the background? Display some representative frames.

In [40]:

```
frame = 171
hbox(
    # Original movie
    plotframe(
        movie_cube_2[:, :, frame],
        title="Rank 2 approx: frame $frame",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_3[:, :, frame],
        title="Rank 3 approx: frame $frame",
        clim=(0, 1)
    ),
)

```

▶ Run

Out[40]:

Rank 2 approx: frame 171

Rank 3 approx: frame 171

In [41]:

```
frame = 187
hbox(
    # Original movie
    plotframe(
        movie_cube_2[:, :, frame],
        title="Rank 2 approx: frame $frame",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_3[:, :, frame],
        title="Rank 3 approx: frame $frame",
        clim=(0, 1)
    ),
)
```

▶ Run

Out[41]:

Rank 2 approx: frame 187

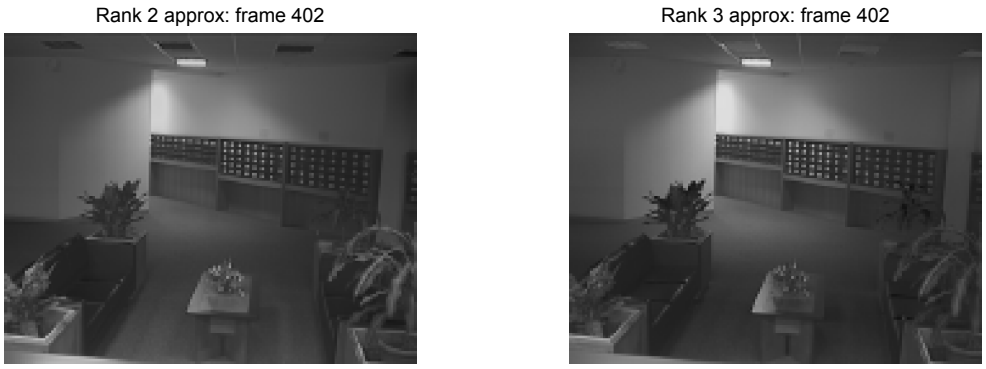
Rank 3 approx: frame 187

In [42]:

```
frame = 402
hbox(
    # Original movie
    plotframe(
        movie_cube_2[:, :, frame],
        title="Rank 2 approx: frame $frame",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_3[:, :, frame],
        title="Rank 3 approx: frame $frame",
        clim=(0, 1)
    ),
)
```

▶ Run

Out[42]:



In [43]:
```
frame = 425
hbox(
    # Original movie
    plotframe(
        movie_cube_2[:, :, frame],
        title="Rank 2 approx: frame $frame",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_3[:, :, frame],
        title="Rank 3 approx: frame $frame",
        clim=(0, 1)
    ),
)
```

▶ Run

Out[43]:

Rank 3 approx better captures the background, which is clear if you see the residuals at the above frames. Rank 2 approx deletes some of the objects in the background like plants while rank 3 captures these better.

Edit / Render

5.2 Information contained in the third singular vector

Plot the first three right singular vectors. Remark on the characteristics of the third singular vector.

- Is it flat?
- Does it have "jumps"?
- Are the jumps smooth, or sharp?
- What is happening in the video when the jumps occur?

It is not flat, the singular vector have jumps.

The jumps are sharp.

The lights in the background turn on or off when the jumps occur.

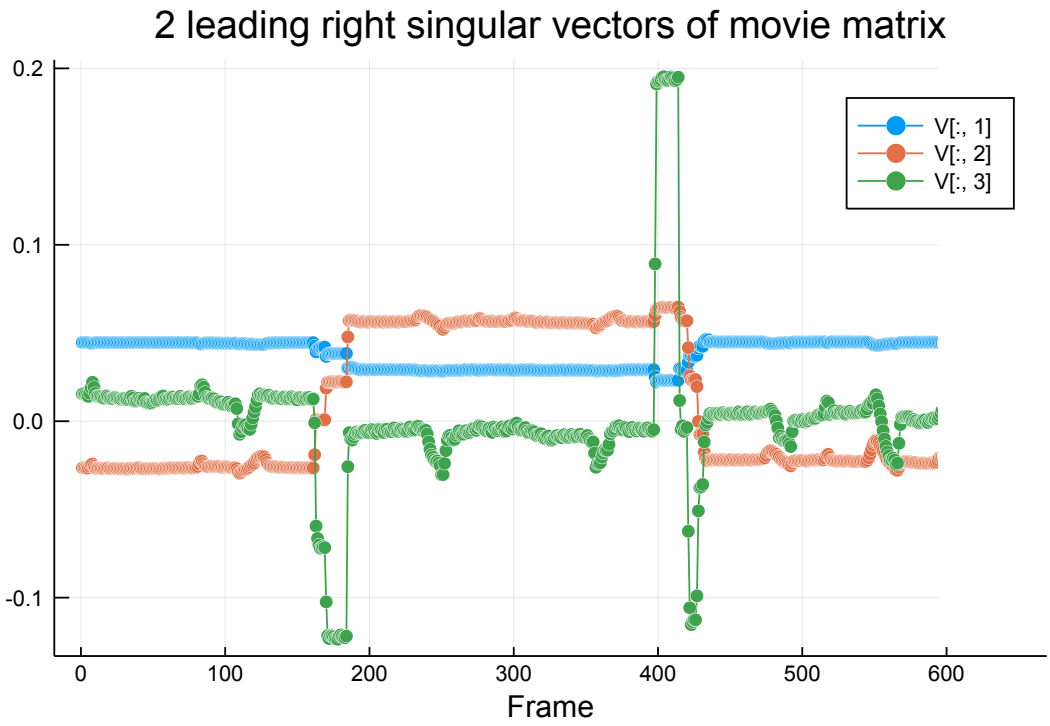
Edit / Render

In [49]:

```
##TODO: Plot the first three right singular vectors
UsV = svds(reshape(movie_cube, :, size(movie_cube, 3)), nsv=3)[1]
V = positivify(UsV.V);
plot(
    V,
    label=["V[:, $i]" for i in collect(1:3)'],
    title="$k leading right singular vectors of movie matrix",
    xaxis="Frame",
    marker=:circle,
)

```

Run

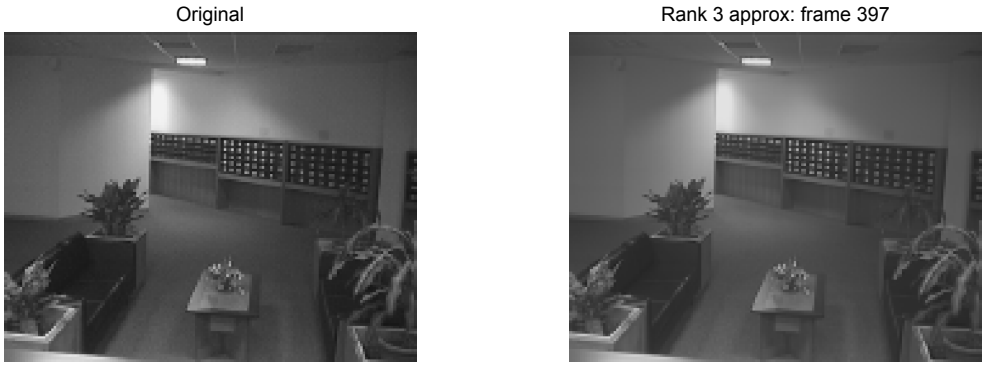
Out[49]:



In [72]:
```
## TODO: display what happens in the video at time instances
# corresponding to jumps or anything interesting happening
# in the third right singular vector.
frame = 397
hbox(
    # Original movie
    plotframe(
        movie_cube[:, :, frame],
        title="Original",
        clim=(0, 1)
    ),
    # Rank-k approx
    plotframe(
        movie_cube_3[:, :, frame],
        title="Rank 3 approx: frame $frame",
        clim=(0, 1)
    ),
)

```

▶ Run

Out[72]:



In the discussion earlier, we spoke about how the scene could be represented as $x_1 y_1^T + x_2 y_2^T$. If the scene is better described by a rank-3 matrix, that suggests that the light on/off phenomenon in the video (absent people) is actually better modeled by $x_1 y_1^T + x_2 y_2^T + x_3 y_3^T$.

What do x_3 and y_3 physically represent, if x_1, x_2 and y_1, y_2 are as described earlier?

Your answer here!

✎ Edit / Render

5.3 The singular value spectrum of the reshaped video matrix

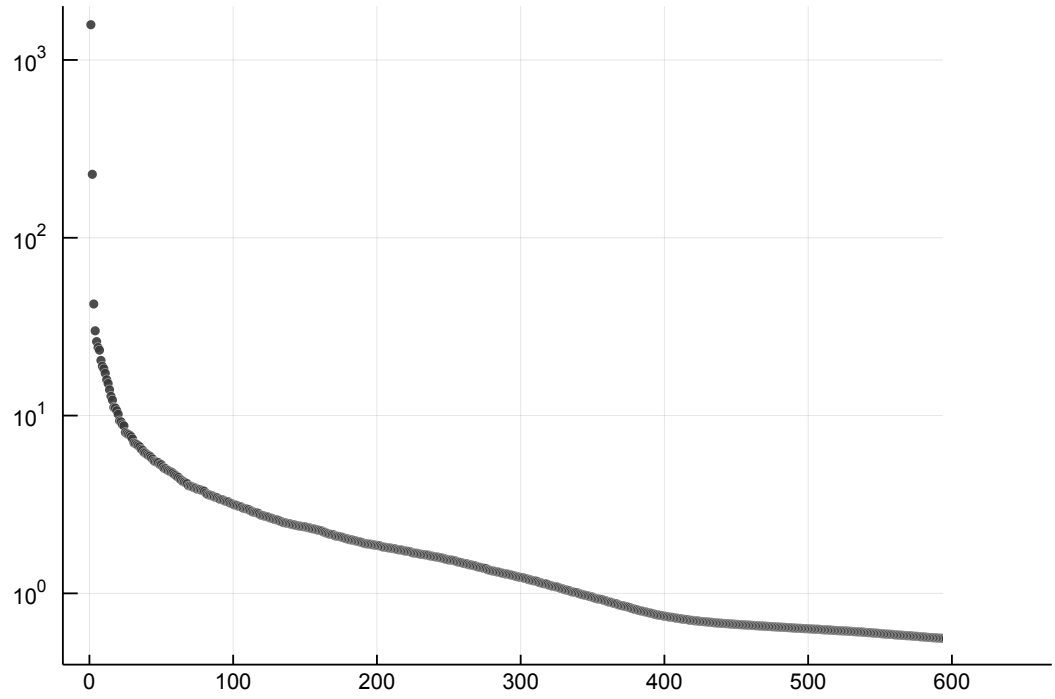
It turns out that the singular value spectrum of the video matrix contains information that could

have allowed us to determine the implicit rank ($k = 3$) of the "light on/off" matrix. Plot the singular value spectrum in the cell below. Notice that many of the values appear grouped together, and a few appear to be well-separated from this group.

In [69]:
```
##TODO: Plot the singular values of the reshaped video matrix.
# Tip: Use the `scatter` and `svdvals` commands. Consider plotting on
nframes = size(movie_cube, 3)
movie_matrix = reshape(movie_cube, :, nframes)

using LinearAlgebra: svdvals
s = svdvals(movie_matrix)

scatter(1:length(s), s; yscale=:log10, alpha=0.7, markersize=3, c=:bl
```

▶ Run

Out[69]:



It can be shown mathematically (and we will do so later) that one can estimate the rank of low-rank structure by counting the number of singular values that are relatively separate from the "continuous-looking portion" of the singular value spectrum.

Do you agree with this claim?

Yes

Edit / Render

5.4 Comparing to the "mean frame"

What if instead of using the SVD, we just took the average of all the frames? How would that differ from our low-rank approximation?

In [65]:
```
# write code to compute the "mean frame movie" and
# display it alongside the original movie and a
# rank-k approximation
m,n,l = size(movie_cube)
movie_cube_mean = zeros(m,n,l)
movie_cube_mean[:,:,1] = sum(movie_cube,dims = 3)/l
for i in 1:l
    movie_cube_mean[:,:,i] = movie_cube_mean[:,:,1]
end
playvideo(
    [movie_cube, movie_cube_2, movie_cube_mean],
    ["Original", "Rank-2 approximation","Mean Movie"],
    frames_per_second=120
)
```

▶ Run

Out[65]:

325

In [67]:
```
plotframe(
        movie_cube_mean[:, :, 1],
        title="Mean frame",
        clim=(0, 1)
    )
```

▶ Run

Out[67]: