18.338 Project, Yaman Otuzbir

Introduction

Many modern data, driven models, from dynamical systems to econometric forecasting, are expressed in terms of linear relationships between variables. These relationships are often encoded in a coefficient matrix whose spectrum controls key properties of the model: stability of a dynamical system, rates of decay or growth, and how information propagates across components. In practice, however, we rarely observe clean data. Measurements are contaminated by noise, and to make matters more delicate, we frequently impose additional structure such as sparsity or low rank via regularized regression methods (e.g., LASSO, nuclear norm minimization) in order to obtain interpretable, data-efficient models.

This raises a fundamental question: how does noise in the data, together with these structural constraints, distort the learned coefficient matrix and its spectrum? Small changes in eigenvalues can change stability conclusions, alter inferred time scales, or create spurious modes that look like genuine dynamics. Understanding these effects is crucial if we want to trust the qualitative and quantitative conclusions drawn from such models. In this project, I will be examining the effects of Gaussian noise on the spectrum of coefficient matrices learned by using sparse or low rank regression techniques.

Functions

In [1]:
```
using LinearAlgebra, SparseArrays
using Plots
using GLMNet
using Random
```

In [2]:
```
using DataDrivenDiffEq
using DataDrivenDMD      # DMD / Koopman algorithms
using OrdinaryDiffEq
```

In [3]:
```
"""
    fit_A_lasso_glmnet_multi(X, Y;
                             alpha=1.0,
                             nlambda=100,
                             lambda_frac=0.1,
                             standardize=true,
                             standardize_response=true)

Fit a sparse A for
    Y ≈ X * Aᵀ
using GLMNet LASSO with a multi-response Gaussian model (MvNormal).

Arguments
---------
- X :: N×d  (rows = x_tᵀ)
- Y :: N×d  (rows = x_{t+1}ᵀ)
- alpha: 1.0 = pure LASSO, in [0,1]
```

```
- nlambda: number of λ values on the path
- lambda_frac: choose λ ≈ lambda_frac * maximum(path.lambda)
               (smaller => less regularization, denser A)
- standardize, standardize_response: passed to glmnet

Returns
-------
- A_hat :: d×d   so that x_{t+1} ≈ A_hat * x_t
- λ_chosen :: Float64
"""
function fit_A_lasso_glmnet_multi(X::AbstractMatrix, Y::AbstractMatrix;
                                  alpha::Real = 1.0,
                                  nlambda::Int = 100,
                                  lambda_frac::Real = 0.1,
                                  standardize::Bool = true,
                                  standardize_response::Bool = true)

    N, d = size(X)
    @assert size(Y) == (N, d)

    # Multi-response Gaussian family
    path = glmnet(X, Y, GLMNet.MvNormal();
                  alpha = alpha,
                  nlambda = nlambda,
                  intercept = false,              # x_{t+1} = A x_t, no bias
                  standardize = standardize,
                  standardize_response = standardize_response)

    λs = path.lambda

    # Pick a λ somewhere along the path.
    # lambda_frac = 0.1 ⇒ roughly 10% of max λ (quite regularized but not insane).
    target_λ = maximum(λs) * lambda_frac
    k = argmin(abs.(λs .- target_λ))  # index of λ closest to target_λ

    # betas has shape (nvars × nresp × nλ) for multi-response Gaussian
    B = Array(path.betas[:, :, k])    # d×d: rows = predictors, cols = responses

    # We want A such that x_{t+1} ≈ A * x_t
    # Y ≈ X * B  ⇒  rows of Y are x_tᵀ B
    # That corresponds to x_{t+1} = Bᵀ x_t, so A_hat = Bᵀ.
    A_hat = B'

    return A_hat, λs[k]
end
```

Out[3]: fit_A_lasso_glmnet_multi

In [4]:
```
"""
    sequential_threshold_dynamics(X, Y;
                                  λ=0.1,
                                  maxiter=50,
                                  tol=1e-8,
                                  verbose=false)

Sequentially thresholded least squares for the linear dynamical system

    x_{t+1} ≈ A * x_t.
```

```
Given:
    X :: N×d  with rows x_tᵀ
    Y :: N×d  with rows x_{t+1}ᵀ

Algorithm (STLSQ-style):
    1. Initial LS: B⁰ = argmin_B ‖X B − Y‖_F²  (B is d×d, columns are outputs)
    2. For k = 0,1,2,...:
         - supportᵏ = { (i,j): |Bᵏ_{ij}| ≥ λ }
         - For each column j:
               S_j = active row indices in that column
               Refit: β_Sj = argmin ‖X[:, S_j] β − Y[:, j]‖²
         - Set entries outside support to 0.
         - Stop when support and coefficients stabilize.

Returns:
    A_hat   :: d×d (so that x_{t+1} ≈ A_hat * x_t)
    support :: BitMatrix (same shape as A_hatᵀ, i.e., support of B)
"""
function sequential_threshold_dynamics(X::AbstractMatrix, Y::AbstractMatrix;
                                       λ::Real = 0.1,
                                       maxiter::Int = 50,
                                       tol::Real = 1e-8,
                                       verbose::Bool = false)
    N, d = size(X)
    @assert size(Y) == (N, d)

    # Initial full least-squares solution:
    # Solve X * B ≈ Y in Frobenius norm sense (B is d×d)
    B = X \ Y   # d×d
    support = abs.(B) .>= λ

    for k in 1:maxiter
        old_support = copy(support)
        B_new = zeros(eltype(B), d, d)

        # Refit each output dimension j on its active support
        for j in 1:d
            S = findall(support[:, j])  # active row indices for column j
            if isempty(S)
                # Entire column j is thresholded out
                continue
            end

            Xs = @view X[:, S]          # N×|S|
            yj = @view Y[:, j]          # N
            β_S = Xs \ yj               # |S|
            B_new[S, j] .= β_S
        end

        # Update support based on new coefficients
        support = abs.(B_new) .>= λ

        # Check convergence
        if all(support .== old_support) && norm(B_new - B) < tol * max(1.0, norm(B))
            verbose && println("Sequential thresholding converged at iter = $k")
            B = B_new
            break
        end
```

```
        B = B_new
    end

    # Convert to dynamics matrix A so that x_{t+1} ≈ A * x_t
    A_hat = B'

    return A_hat, support
end
```

Out[4]: sequential_threshold_dynamics

In [5]:
```
"""
    param_grid_table(plots, row_params, col_params; row_name="row", col_name="col")

Arrange an n×m matrix of plots into a labeled table layout:

    ┌──────────────── col labels ────────────────┐
    │       ·     |  col_1   col_2   ...       |
    │ row_1       |  p[1,1]  p[1,2]  ...       |
    │ row_2       |  p[2,1]  p[2,2]  ...       |
    │   ...       |   ...     ...    ...       |

- `plots` is an n×m `AbstractMatrix` of `Plots.Plot` objects.
- `row_params` is a length-n vector of row parameter values.
- `col_params` is a length-m vector of column parameter values.

Row/column labels are placed **outside** the plots in their own small cells.
"""
function param_grid_table(plots::AbstractMatrix,
                          row_params::AbstractVector,
                          col_params::AbstractVector;
                          row_name::AbstractString = "row",
                          col_name::AbstractString = "col")

    n, m = size(plots)
    @assert length(row_params) == n "row_params length must match number of rows in plot
    @assert length(col_params) == m "col_params length must match number of columns in p

    cells = Plots.Plot[]

    # helper: create a tiny "label plot" with just centered text
    function label_plot(txt; rotation = 0)
        p = plot(;
            xlim = (0, 1),
            ylim = (0, 1),
            framestyle = :none,
            ticks = nothing,
            legend = false,
        )
        annotate!(p, 0.5, 0.5, text(txt, 14, :center, rotation = rotation))
        return p
    end

    # Fill (n+1)×(m+1) grid in row-major order
    for i in 1:(n+1), j in 1:(m+1)
        if i == 1 && j == 1
            # top-left corner: optional global label or blank
            push!(cells, label_plot(""))
        elseif i == 1
```

```
            # top row → column labels (outside)
            c = col_params[j - 1]
            push!(cells, label_plot("$col_name = $c"; rotation = 0))
        elseif j == 1
            # left column → row labels (outside)
            r = row_params[i - 1]
            push!(cells, label_plot("$row_name = $r"; rotation = 90))
        else
            # actual user-provided plot
            push!(cells, plots[i - 1, j - 1])
        end
    end

    plot(cells...; layout = (n + 1, m + 1), size = (300 * m, 260 * n))
end
```

Out[5]: param_grid_table

In [6]:
```
function eigvals_by_eps(X_true, regressor, eps_vals; seed = 0)
    Random.seed!(seed)
    evals = (0 + 0im) * zeros(size(X_true, 2), size(eps_vals, 1))
    for (idx,eps) in enumerate(eps_vals)
        X_obs = X_true + eps * norm(X_true)* randn(size(X_true));
        X = copy(X_obs[1:end - 1, :])
        Y = copy(X_obs[2:end, :]);
        eps_evals, _ = regressor(X, Y)
        evals[:,idx] = eigvals(eps_evals)
    end
    return evals
end
```

Out[6]: eigvals_by_eps (generic function with 1 method)

In [16]:
```
"""
    random_orthonormal_columns(n, r; rng=Random.GLOBAL_RNG)

Return an n×r matrix U whose columns are orthonormal (U'U = I_r).
"""
function random_orthonormal_columns(n, r; rng=Random.GLOBAL_RNG)
    @assert 1 ≤ r ≤ n "r must be between 1 and n"
    A = randn(rng, n, r)
    F = qr(A)               # thin QR
    U = Matrix(F.Q[:, 1:r]) # first r orthonormal columns
    return U
end

"""
    unit_circle_block(r; rng=Random.GLOBAL_RNG, allow_real=true)

Construct an r×r real matrix T whose eigenvalues lie on the unit circle.
- Uses 2×2 rotation blocks for complex conjugate pairs e^{±iθ}.
- If r is odd and `allow_real=true`, uses a 1×1 block with eigenvalue ±1.

Returns:
  T, eigs   where eigs is a Vector{ComplexF64} of the nonzero eigenvalues.
"""
function unit_circle_block(r; rng=Random.GLOBAL_RNG, allow_real=true)
    @assert r ≥ 1
    blocks = Vector{Matrix{Float64}}()
```

```
    eigs   = ComplexF64[]

    k = div(r, 2)              # number of 2x2 rotation blocks
    for _ in 1:k
        θ = 2π * rand(rng)     # angle in [0, 2π)
        R = [cos(θ) -sin(θ);
             sin(θ)  cos(θ)]
        push!(blocks, R)
        push!(eigs, cis(θ))    # e^{iθ}
        push!(eigs, cis(-θ))   # e^{-iθ}
    end

    if isodd(r)
        if allow_real
            λ = rand(rng) < 0.5 ? 1.0 : -1.0  # ±1
            push!(blocks, reshape(λ, 1, 1))
            push!(eigs, complex(λ))
        else
            error("r is odd but allow_real=false; cannot fill spectrum with |λ|=1.")
        end
    end

    # Assemble T as block diagonal
    T = zeros(Float64, r, r)
    idx = 1
    for B in blocks
        p, q = size(B)
        @assert p == q
        T[idx:idx+p-1, idx:idx+p-1] .= B
        idx += p
    end

    return T, eigs
end

"""
    random_unit_circle_lowrank_matrix(n, r; rng=Random.GLOBAL_RNG)

Construct a real n×n matrix A with:
  - rank(A) = r
  - nonzero eigenvalues λ with |λ| = 1 (complex conjugate pairs allowed)
  - remaining eigenvalues exactly 0.

Returns:
  A, eigs   where eigs contains the nonzero eigenvalues (on the unit circle).
"""
function random_unit_circle_lowrank_matrix(n, r; rng=Random.GLOBAL_RNG)
    @assert 1 ≤ r ≤ n "r must be between 1 and n"

    # Orthonormal basis for the r-dim active subspace
    U = random_orthonormal_columns(n, r; rng=rng)

    # Dynamics restricted to that subspace (unit modulus spectrum)
    T, eigs = unit_circle_block(r; rng=rng, allow_real=true)

    # Lift to n×n real matrix
    A = U * T * U'
```

```
    return A, eigs
end
```

Out[16]: random_unit_circle_lowrank_matrix

Effect of a Small Perturbation on Eigenvalues

In this section, I illustrate how different levels of noise affect the spectrum of matrices in a controlled setting. Here, the noise is applied directly to the matrix entries, rather than arising indirectly through a regression or learning procedure. This allows us to isolate the effect of Gaussian perturbations on the eigenvalues themselves. In the special case of symmetric eigenvalue problems, classical random matrix theory gives an explicit formula for the derivative of an eigenvalue with respect to a perturbation. This provides a clean theoretical lens through which to interpret the numerical experiments in this section.

Through the following examples, we see that even without applying any additional transformations or fancy learning methods, small perturbations can already change the spectrum in significant ways. In particular, eigenvalues that are closely spaced or associated with nearly degenerate eigenspaces can be especially sensitive, leading to noticeable shifts even when the noise level is modest in norm. We also observe that the qualitative behavior depends on the underlying distribution of the original matrix entries. By varying both the noise level and the base distribution, we obtain a clearer picture of when and how Gaussian perturbations can meaningfully alter spectral properties that are often used to infer stability, dominant modes, or effective dimensionality.

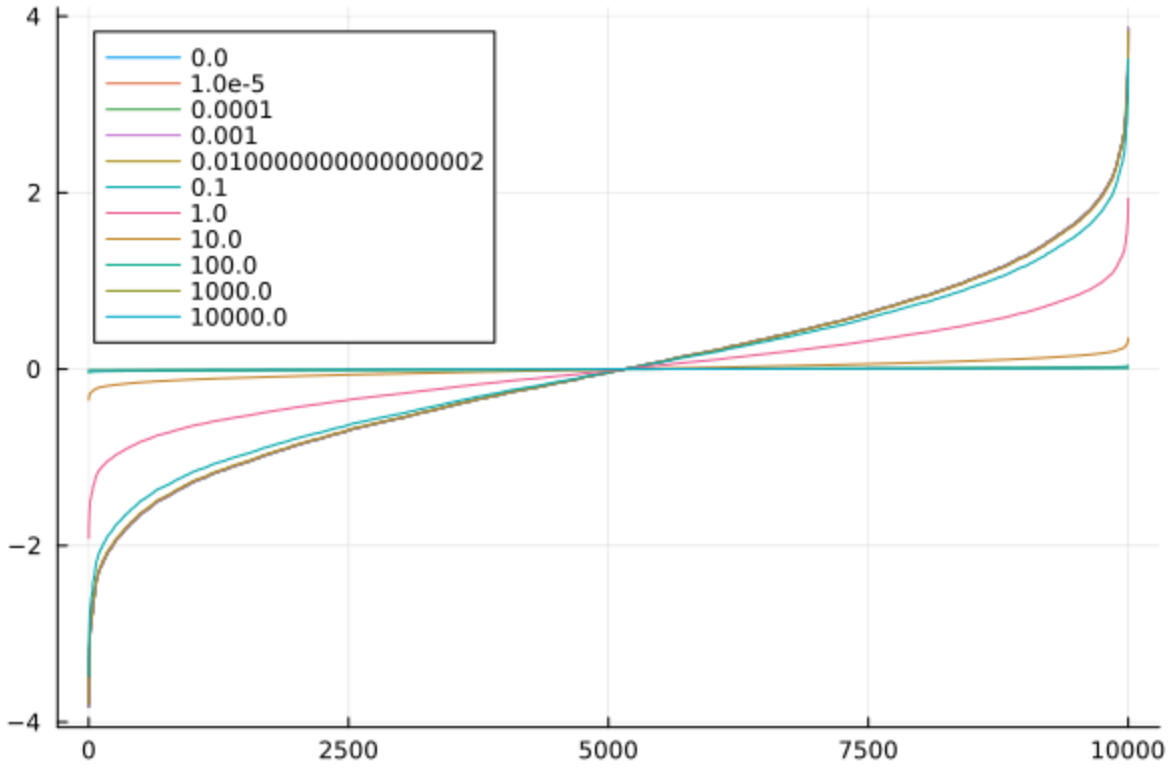
Perturbing a Matrix with $\lambda_i \sim \mathcal{N}(0,1)$

In [162…
```
n = 10000
A = diagm(randn(n))
dA = randn(n,n);
dA = (dA + dA')/(2 * (n));
```

In [163…
```
eps_vals = [0; 10.0 .^ (-5:1.0:4)]
ntrials =  length(eps_vals)
evals_by_eps = zeros(n, ntrials)

for trial = 1:ntrials
    eps = eps_vals[trial]
    S = (A + eps * dA)/(1 + eps)
    evals_by_eps[:,trial] = eigvals(S)
end
```

In [164…
```
plot(evals_by_eps,legend = true, labels = [eps for eps in eps_vals]',) #xscale = :log10,
```

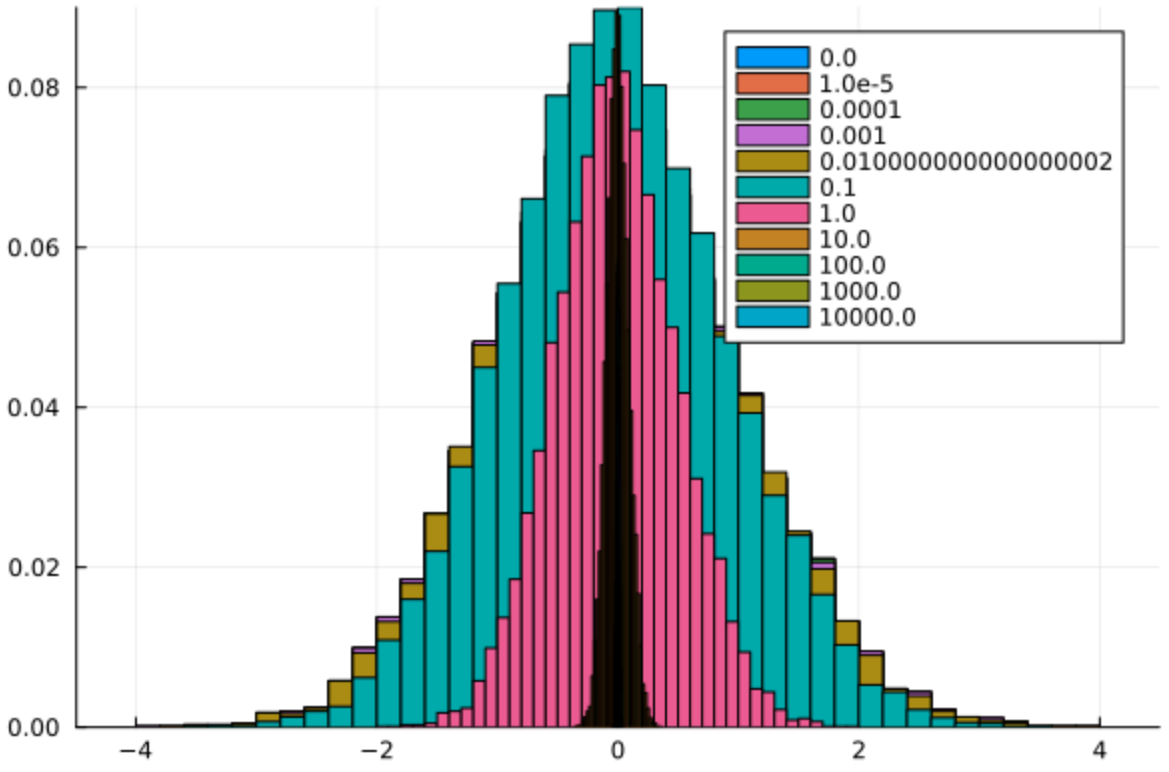
Out[164…



In [165…

```
histogram(evals_by_eps, normalize = :probability, labels = [eps for eps in eps_vals]')
```

Out[165…



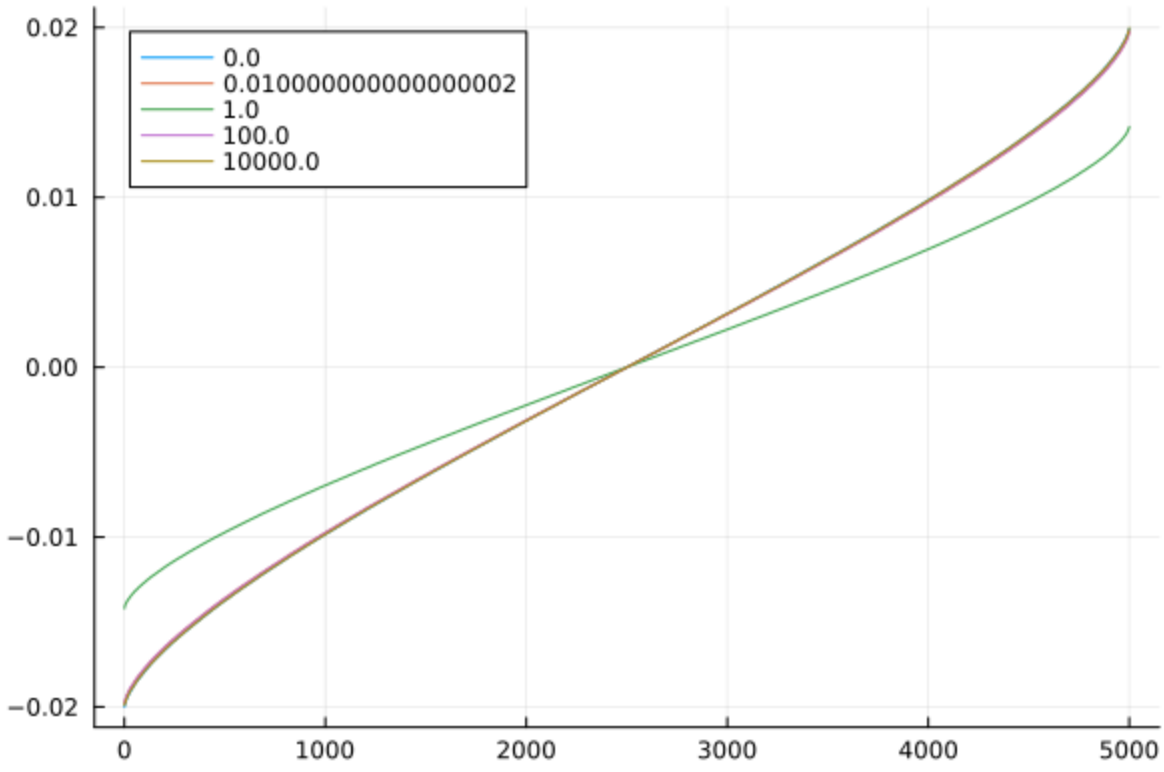
Perturbing a GOE Matrix

In [170…
```
n = 5000
A = randn(n,n)
A = (A + A')/(2 * n)
dA = randn(n,n);
dA = (dA + dA')/(2 * n);
```

In [180…
```
eps_vals = [0; 10.0 .^ (-2:2:4)]
ntrials =  length(eps_vals)
evals_by_eps = zeros(n, ntrials)

for trial = 1:ntrials
    eps = eps_vals[trial]
    S = (A + eps * dA)/(1 + eps)
    evals_by_eps[:,trial] = eigvals(S)
end
```

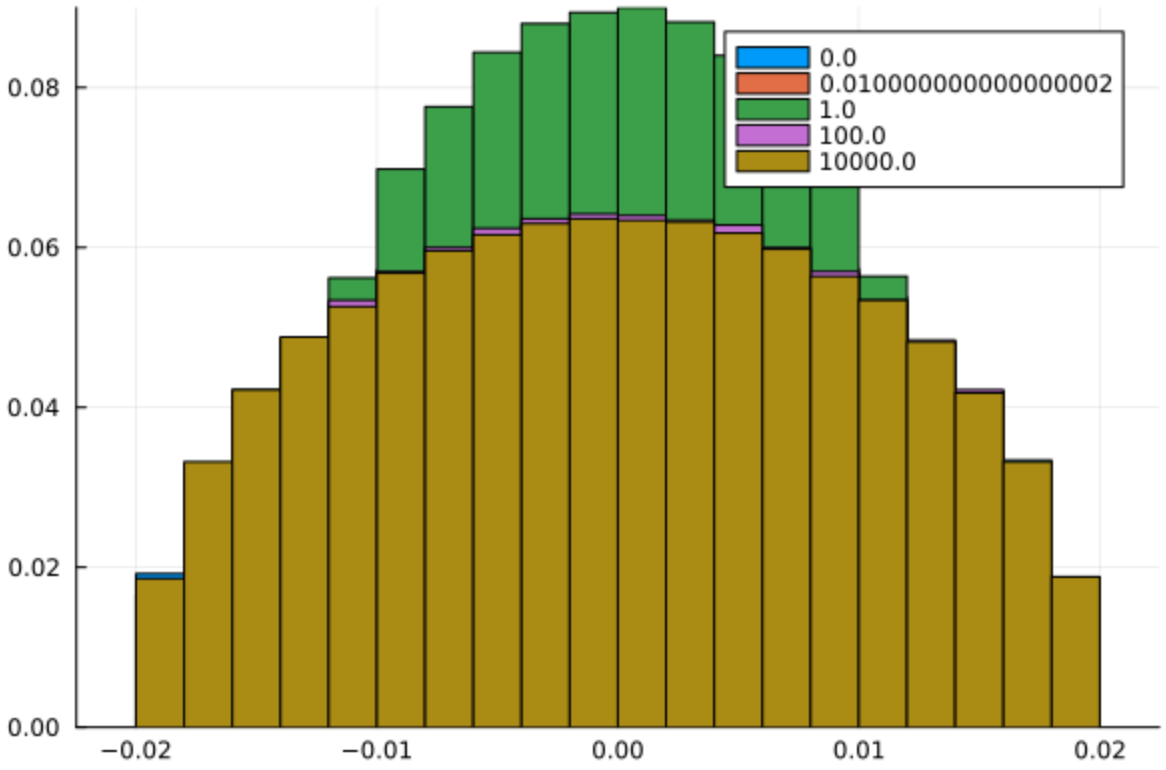
In [181…
```
plot(evals_by_eps,legend = true, labels = [eps for eps in eps_vals]',) #xscale = :log10,
```

Out[181…



In [182…
```
histogram(evals_by_eps, normalize = :probability, labels = [eps for eps in eps_vals]')
```

Out[182…



Perturbing the Identity Matrix

In [183…

```
n = 5000
A = Matrix(I, n,n)
dA = randn(n,n);
dA = (dA + dA')/(2 * n);
```

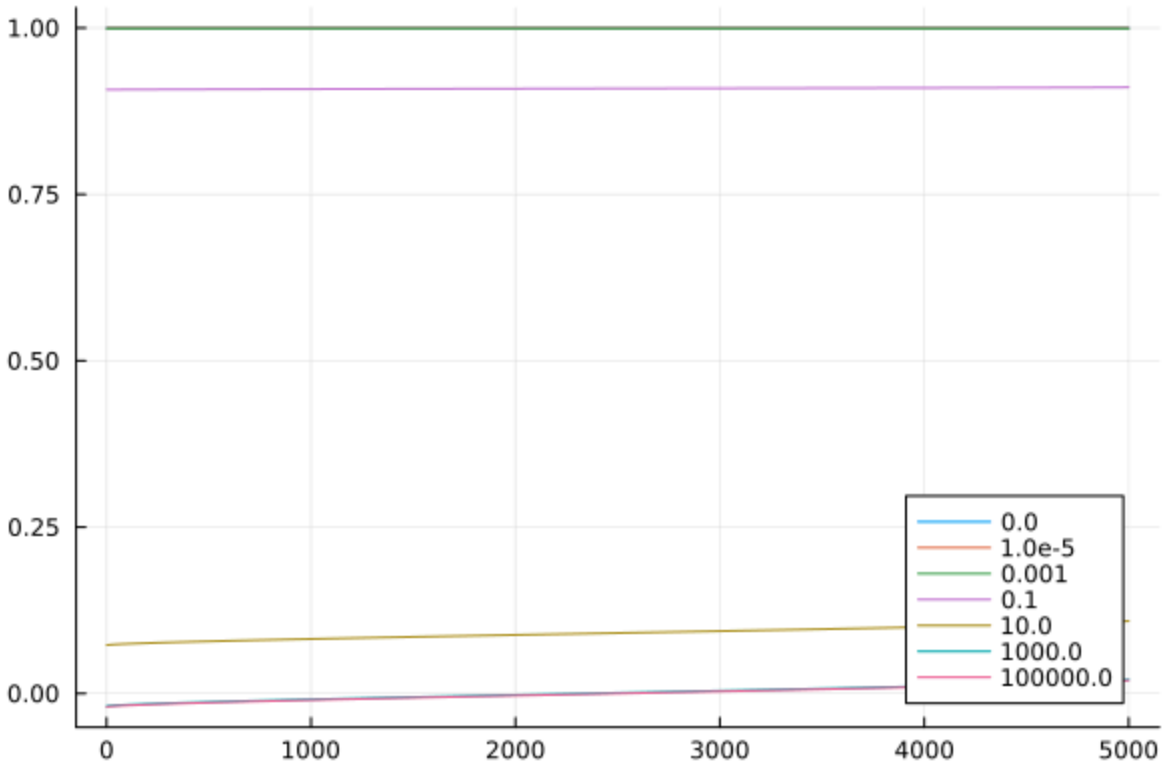
In [184…

```
eps_vals = [0; 10.0 .^ (-5:2:5)]
ntrials =  length(eps_vals)
evals_by_eps = zeros(n, ntrials)

for trial = 1:ntrials
    eps = eps_vals[trial]
    S = (A + eps * dA)/(1 + eps)
    evals_by_eps[:,trial] = eigvals(S)
end
```

In [185…

```
plot(evals_by_eps,legend = true, labels = [eps for eps in eps_vals]')
```

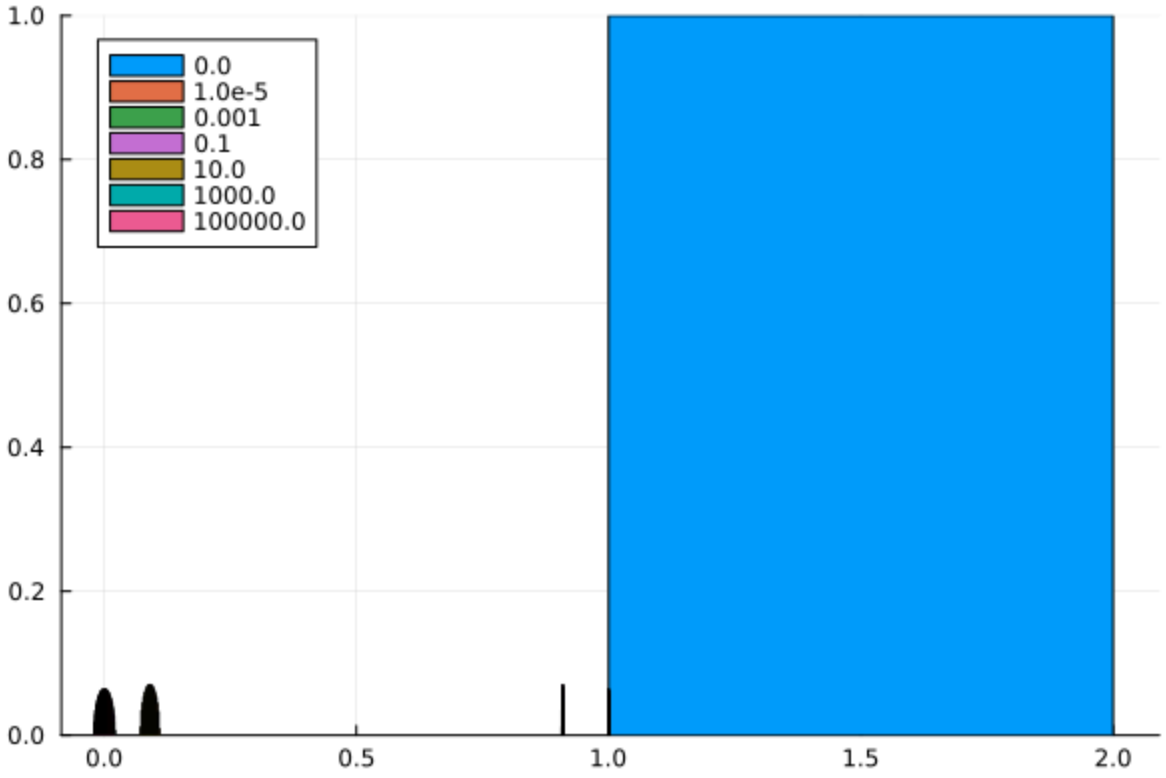
Out[185…



In [186…

```
histogram(evals_by_eps, normalize = :probability, labels = [eps for eps in eps_vals]')
```

Out[186…



Background

Discrete-time linear dynamical systems

A discrete-time dynamical system describes the evolution of a state $x_t \in \mathbb{R}^n$ at integer time steps $t = 0, 1, 2, \ldots$ according to an update rule

$$x_{t+1} = F(x_t).$$

In this project we focus on the linear case, where F is given by multiplication with a matrix $A \in \mathbb{R}^{n \times n}$:

$$x_{t+1} = Ax_t.$$

The solution after t steps is $x_t = A^t x_0$, so the long-term behavior is dictated by the eigenvalues of A. In particular, the spectral radius $\rho(A)$ controls stability: if all eigenvalues lie strictly inside the unit circle, $|\lambda_i| < 1$, then trajectories decay to zero; if some eigenvalues lie on or outside the unit circle, one may observe persistent oscillations or growth. Because of this, the spectrum of A is often interpreted in terms of time scales, dominant modes, and stability properties of the underlying system.

In data-driven settings, we typically do not know A a priori. Instead, we observe (possibly noisy) snapshot pairs (x_t, x_{t+1}) and fit a model that approximates the best linear map from x_t to x_{t+1}. The methods considered in this project, LASSO, sequential thresholding, and dynamic mode decomposition, all fit into this framework as different ways of estimating A (or its spectrum) from data under various structural assumptions.

LASSO and sparse regression

Suppose we collect snapshot data in matrices

$$X = [x_0, x_1, \ldots, x_{m-1}] \in \mathbb{R}^{n \times m}, \quad Y = [x_1, x_2, \ldots, x_m] \in \mathbb{R}^{n \times m},$$

so that, in the noise-free linear case, we would have $Y = AX$. A natural way to estimate A is to solve a least-squares problem

$$\min_A \ \frac{1}{2}\|Y - AX\|_F^2.$$

However, in many applications we believe that the true dynamics are sparse, each component of x_t depends only on a few others, and in some other applications where the ground truth may not be sparse we still enforce sparsity to keep models simple and easy to interpret. We incorporate this prior by adding an ℓ_1 penalty on the entries of A. This leads to the LASSO (least absolute shrinkage and selection operator) formulation

$$\min_A \ \frac{1}{2}\|Y - AX\|_F^2 + \lambda \|A\|_1,$$

where $\|A\|_1 = \sum_{i,j} |a_{ij}|$ and $\lambda > 0$ controls the strength of the sparsity penalty.

The ℓ_1 penalty encourages many coefficients to be exactly zero, effectively performing variable selection and producing interpretable, sparse models. At the same time, it shrinks nonzero coefficients toward zero, which can introduce a bias in the learned matrix and, consequently, in its spectrum. One of the goals of this project is to understand how this shrinkage, combined with Gaussian noise in the data, affects the eigenvalues of the learned A.

Sequential thresholding

Sequential (or iterative) thresholding is an alternative way to enforce sparsity that is conceptually simpler than LASSO. Instead of adding a penalty to the optimization problem, we alternate between least-squares regression and explicit hard-thresholding of small coefficients. A basic version proceeds as follows:

1. **Initial fit:** Solve the unregularized least-squares problem
$$A^{(0)} = \arg\min_{A} \ \frac{1}{2}\|Y - AX\|_F^2.$$
2. **Thresholding:** For a chosen threshold $\tau > 0$, set entries with small magnitude to zero:
$$a_{ij}^{(k+1)} = \begin{cases} a_{ij}^{(k)}, & |a_{ij}^{(k)}| \geq \tau, \\ 0, & |a_{ij}^{(k)}| < \tau. \end{cases}$$
3. **Refit on support (optional):** Restrict to the nonzero pattern (support) of $A^{(k+1)}$ and refit the remaining coefficients by least squares. Repeat thresholding and refitting for a few iterations.

This type of procedure is common in sparse model discovery methods such as SINDy (sparse identification of nonlinear dynamics). Unlike LASSO, once a coefficient survives the threshold it is typically kept close to its least-squares value, so sequential thresholding tends to introduce less shrinkage on the nonzero entries. As a result, when the sparsity assumption is correct and the noise is not too large, the spectrum of the learned matrix can more closely match that of the true system. On the other hand, the method can fail abruptly if the threshold is poorly chosen or if noise masks small but important coefficients.

Dynamic Mode Decomposition (DMD)

Dynamic Mode Decomposition is a widely used technique for extracting coherent spatiotemporal patterns from data generated by (possibly nonlinear) dynamical systems. At its core, DMD fits a linear operator that best advances the state from one snapshot to the next and then analyzes its eigenvalues and eigenvectors. Using the same snapshot matrices X and Y as above, the “exact” DMD algorithm can be summarized as:

1. Compute a (truncated) singular value decomposition
$$X \approx U\Sigma V^\top,$$
where the truncation rank reflects an assumed low-dimensional structure in the data.
2. Define the reduced operator
$$\tilde{A} = U^\top Y V \Sigma^{-1},$$
which is the representation of the best-fit linear map in the low-dimensional subspace spanned by the columns of U.
3. Compute the eigen-decomposition

$$\tilde{A}W = W\Lambda,$$

where Λ contains the DMD eigenvalues.

4. Recover the corresponding DMD modes in the original space as

$$\Phi = YV\Sigma^{-1}W.$$

The DMD eigenvalues approximate the eigenvalues of the underlying linear dynamics (or of a linearization/Koopman approximation in the nonlinear case), and their moduli indicate growth, decay, or oscillatory behavior of the associated modes. DMD can thus be viewed as a particular low-rank regression method that focuses directly on the spectral decomposition of the fitted operator.

In the context of this project, DMD provides a natural baseline for how spectral information is extracted from data when we impose a low-rank structure rather than sparsity. Comparing its behavior under Gaussian noise to that of LASSO and sequential thresholding helps highlight how different structural priors (sparse vs.\ low-rank) interact with noise to distort the learned spectrum of a discrete-time dynamical system.

Experiments

Sparse Ground Truth

LASSO Regression

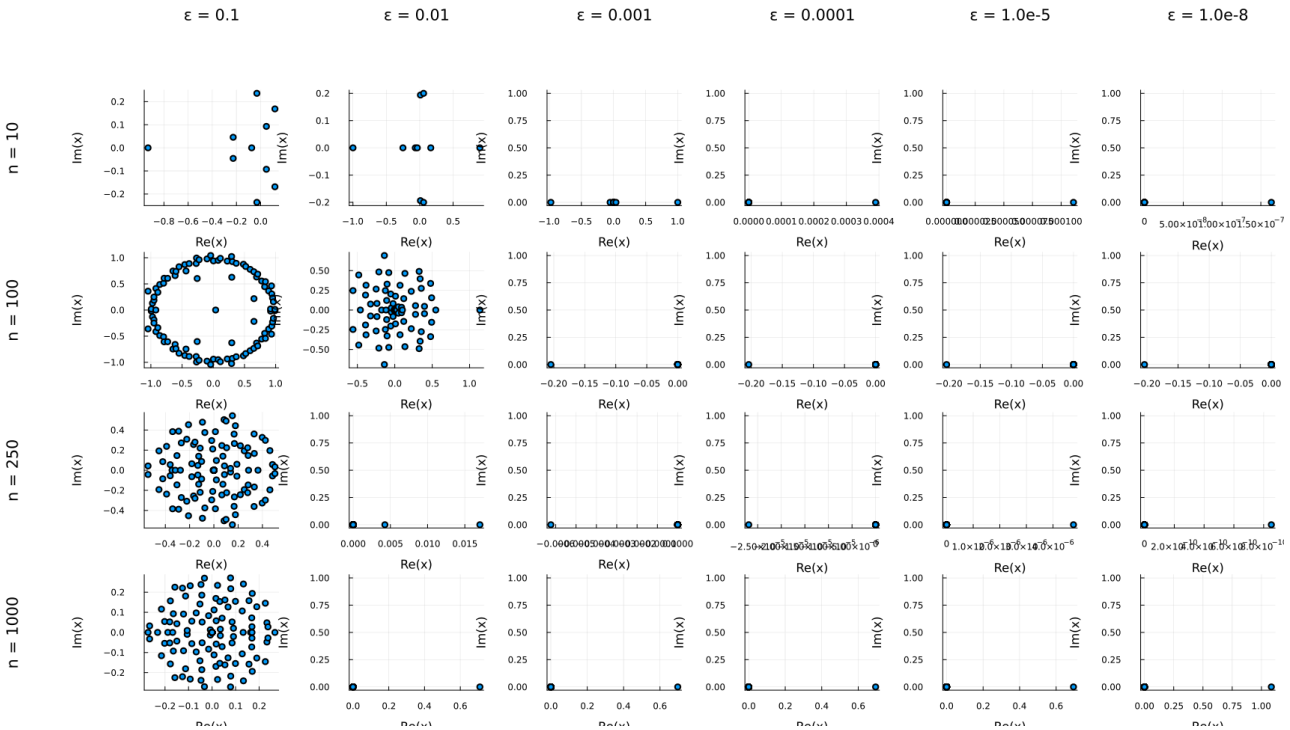
In [42]:
```
Random.seed!(0)

nvals = [10, 100, 250, 1000]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-8]
plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

for (nidx, n) in enumerate(nvals)
    A = SparseMatrixCSC(diagm(rand([-1,1], n)))
    x0 = randn(n);
    clims = (minimum(A), maximum(A))

    ground_truth(x) = A * x

    tf = 100
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        X = copy(X_obs[1:end - 1, :])
        Y = copy(X_obs[2:end, :]);
        A_lasso, λ = fit_A_lasso_glmnet_multi(X, Y;
```

```
                                        alpha = 1.0,
                                        nlambda = 150,
                                        lambda_frac = 0.00005)
            plt = scatter((1 + 0im) * eigvals(A_lasso), legend = false)
            plots[nidx, epsidx] = plt

            mod_plt = plot(sort(abs.(eigvals(A_lasso))), legend = false)
            plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
            modulus_plots[nidx, epsidx] = mod_plt

            hm = heatmap(A_lasso, colorbar = false, clim = clims)
            heatmaps[nidx, epsidx] = hm

            rh = histogram(real.((1 + 0im) * eigvals(A_lasso)), normalize = true)
            real_hists[nidx, epsidx] = rh

            ih = histogram(imag.((1 + 0im) * eigvals(A_lasso)), normalize = true)
            im_hists[nidx, epsidx] = ih
            println("n = $n, eps = $eps")
        end
        heatmaps[nidx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
        real_hists[nidx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
        im_hists[nidx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
    end
```

```
n = 10, eps = 0.1
n = 10, eps = 0.01
n = 10, eps = 0.001
n = 10, eps = 0.0001
n = 10, eps = 1.0e-5
n = 10, eps = 1.0e-8
n = 100, eps = 0.1
n = 100, eps = 0.01
n = 100, eps = 0.001
n = 100, eps = 0.0001
n = 100, eps = 1.0e-5
n = 100, eps = 1.0e-8
n = 250, eps = 0.1
n = 250, eps = 0.01
n = 250, eps = 0.001
n = 250, eps = 0.0001
n = 250, eps = 1.0e-5
n = 250, eps = 1.0e-8
n = 1000, eps = 0.1
n = 1000, eps = 0.01
n = 1000, eps = 0.001
n = 1000, eps = 0.0001
n = 1000, eps = 1.0e-5
n = 1000, eps = 1.0e-8
```

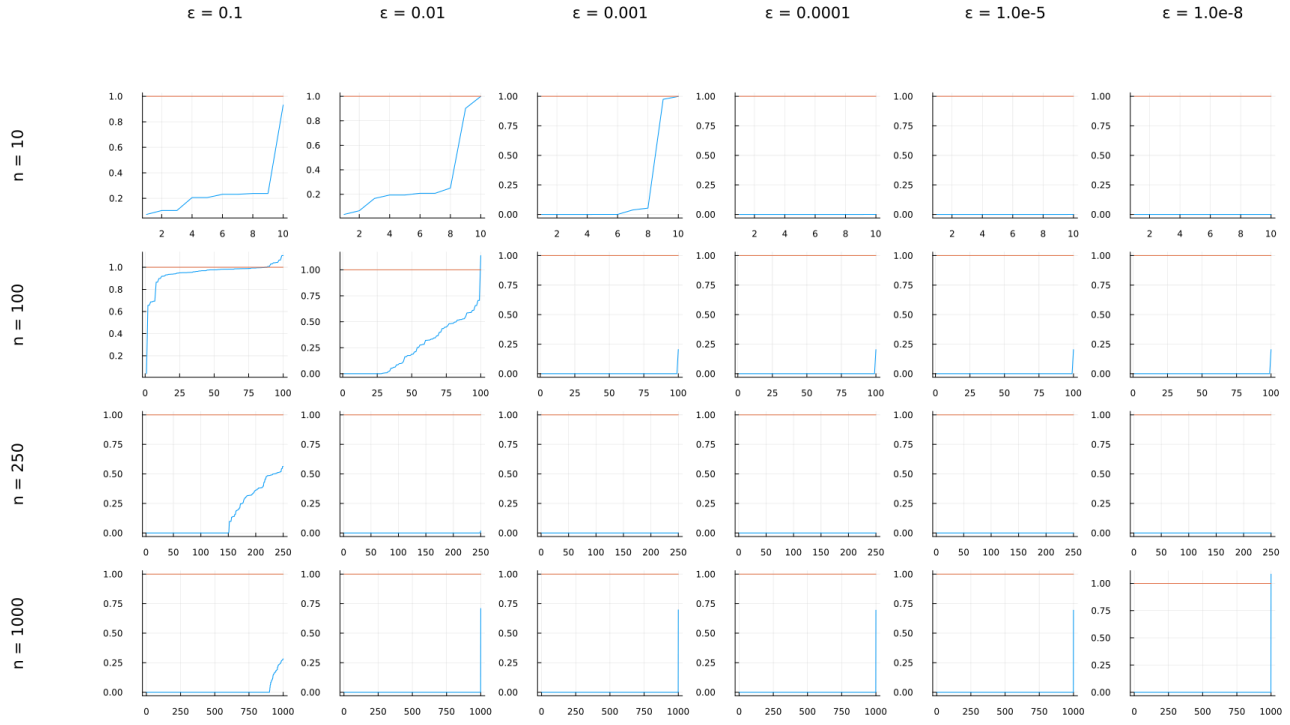
In [43]:
```
param_grid_table(plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[43]:



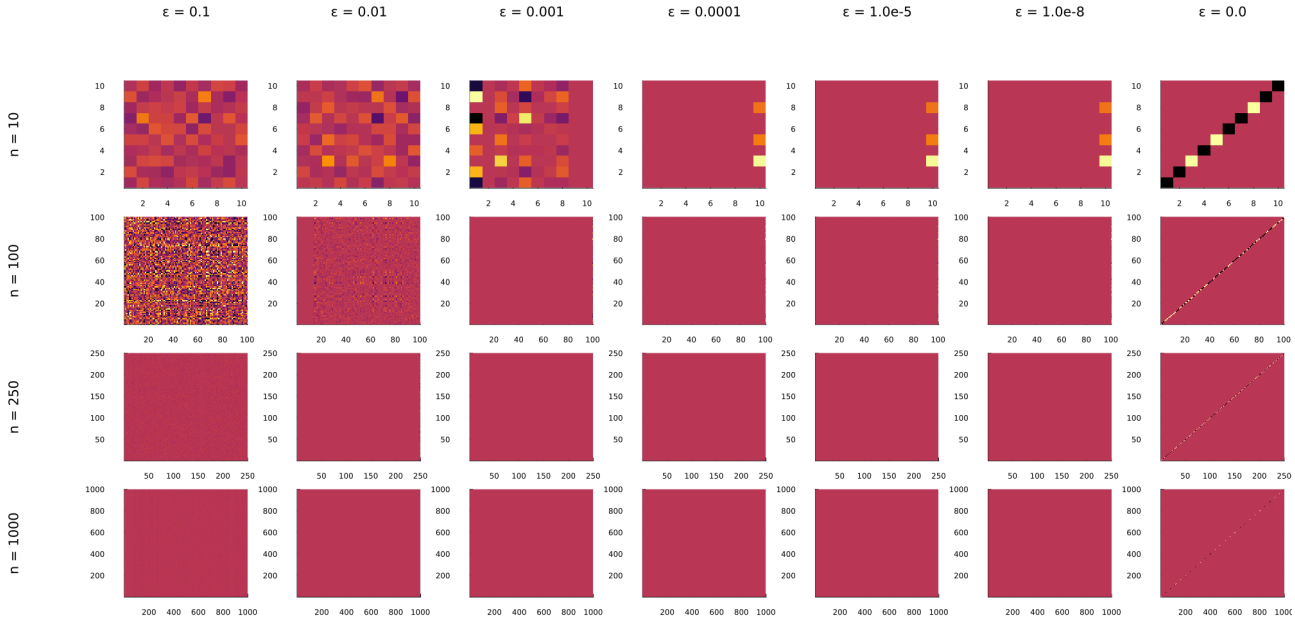
In [44]:
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[44]:



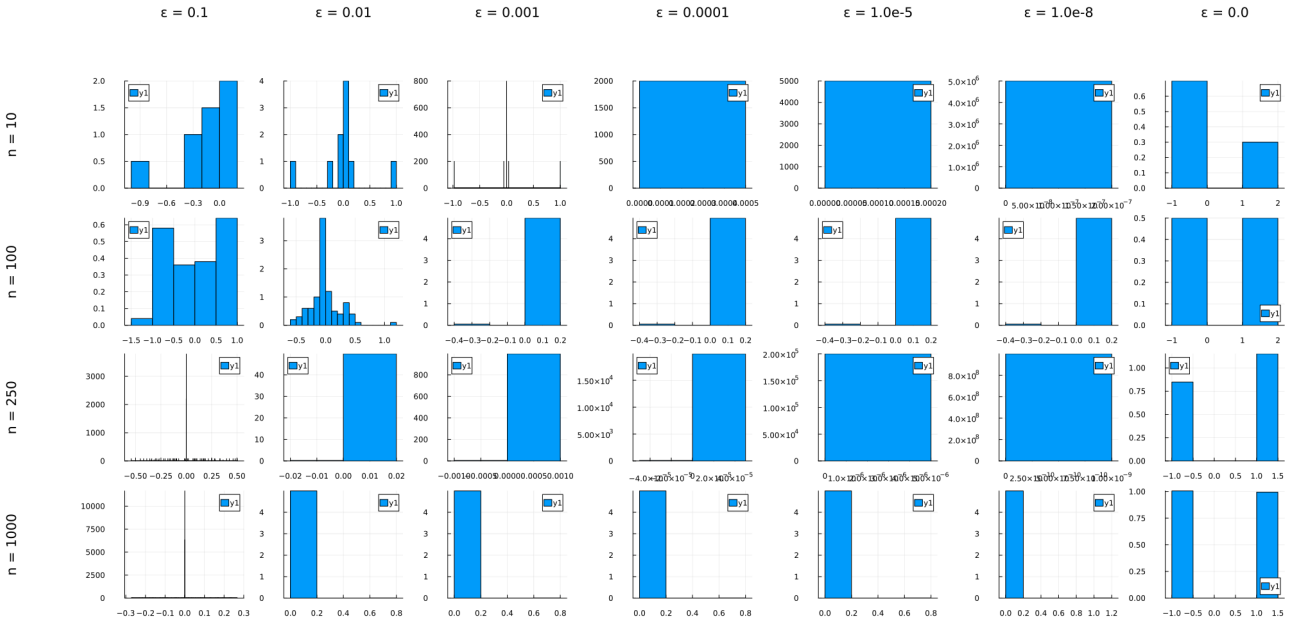
In [45]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[45]:



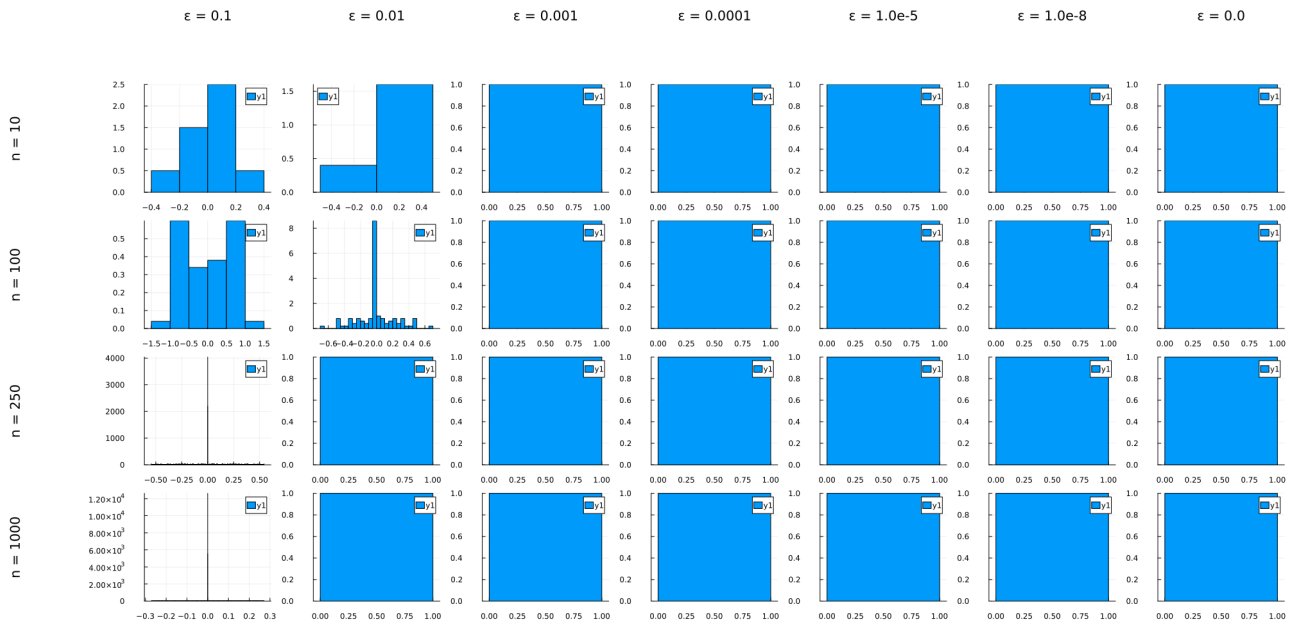
In [46]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[46]:



In [47]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[47]:



Sequantial Thresholding

In [108…

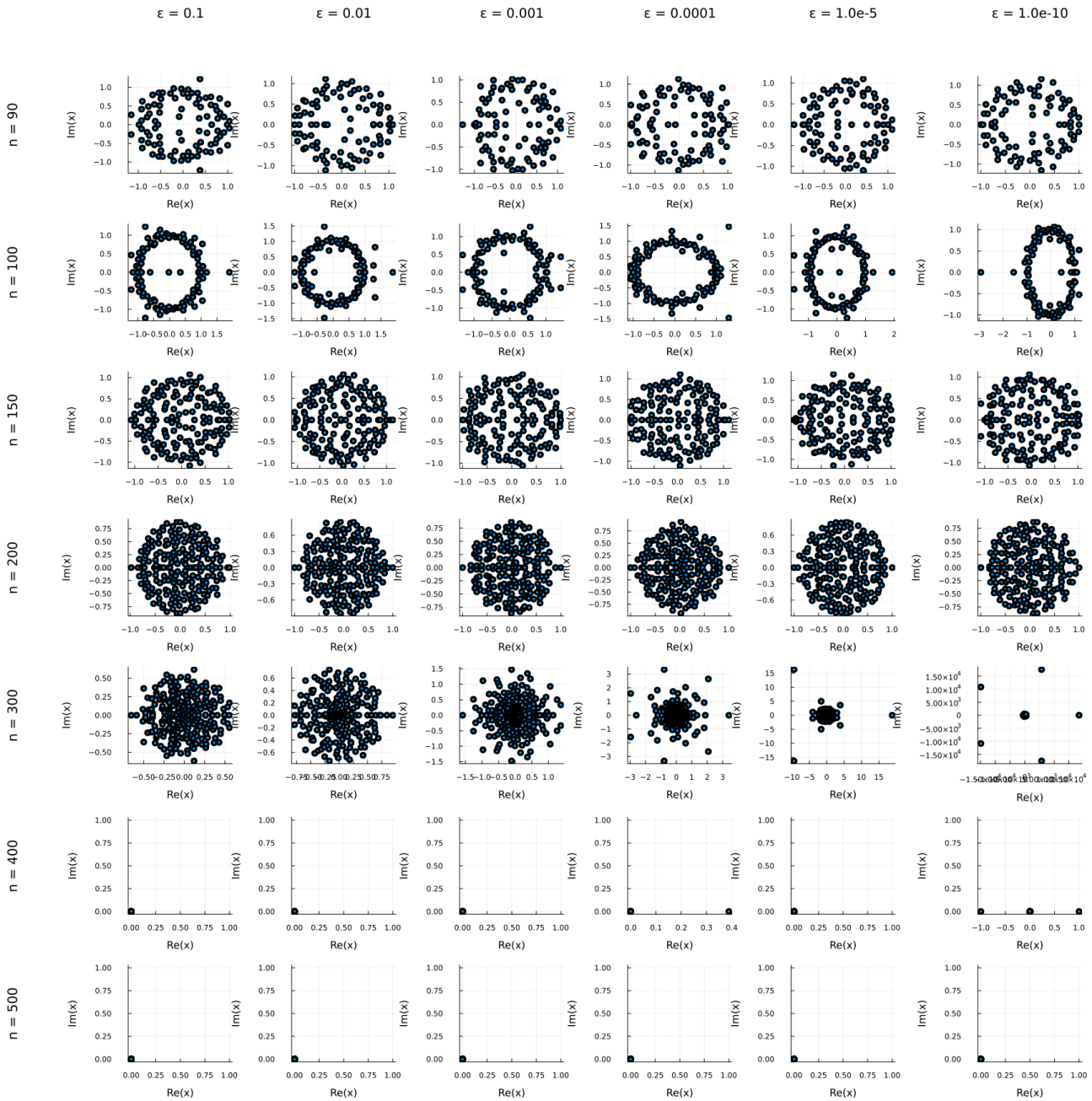
```
Random.seed!(0)

nvals = [90, 100, 150, 200, 300, 400, 500]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-10]
plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

for (nidx, n) in enumerate(nvals)
    A = SparseMatrixCSC(diagm(rand([-1,1], n)))
    x0 = randn(n);
    clims = (minimum(A), maximum(A))

    ground_truth(x) = A * x

    tf = 100
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        X = copy(X_obs[1:end - 1, :])
        Y = copy(X_obs[2:end, :]);
        A_st, _ = sequential_threshold_dynamics(X, Y, tol = 1e-6)
        plt = scatter((1+0im) * eigvals(A_st), legend = false)
        plots[nidx, epsidx] = plt

        mod_plt = plot(sort(abs.(eigvals(A_st))), legend = false)
        plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
        modulus_plots[nidx, epsidx] = mod_plt
        hm = heatmap(A_st, colorbar = false, clim = clims)
```

```
        heatmaps[nidx, epsidx] = hm

        rh = histogram(real.((1 + 0im) * eigvals(A_st)), normalize = true)
        real_hists[nidx, epsidx] = rh

        ih = histogram(imag.((1 + 0im) * eigvals(A_st)), normalize = true)
        im_hists[nidx, epsidx] = ih

    end
    heatmaps[nidx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
    real_hists[nidx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
    im_hists[nidx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
end
```

In [109…

```
param_grid_table(plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

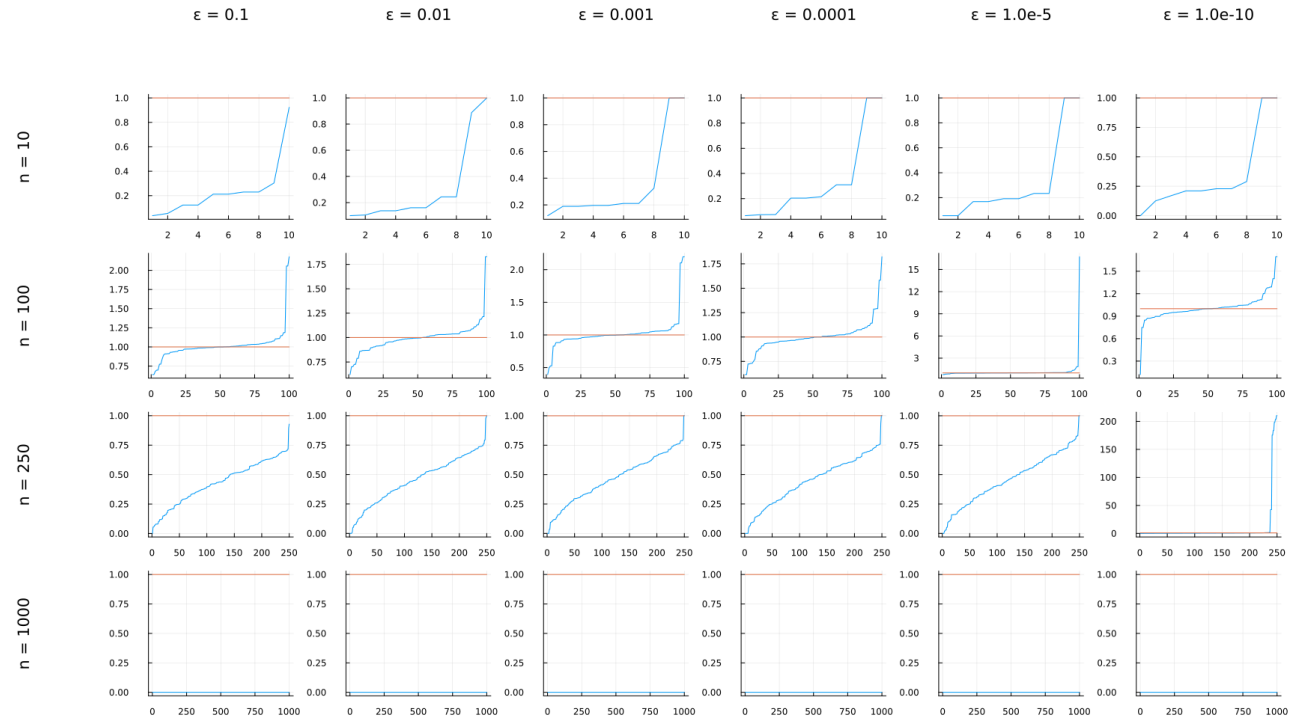
Out[109…



In [50]:

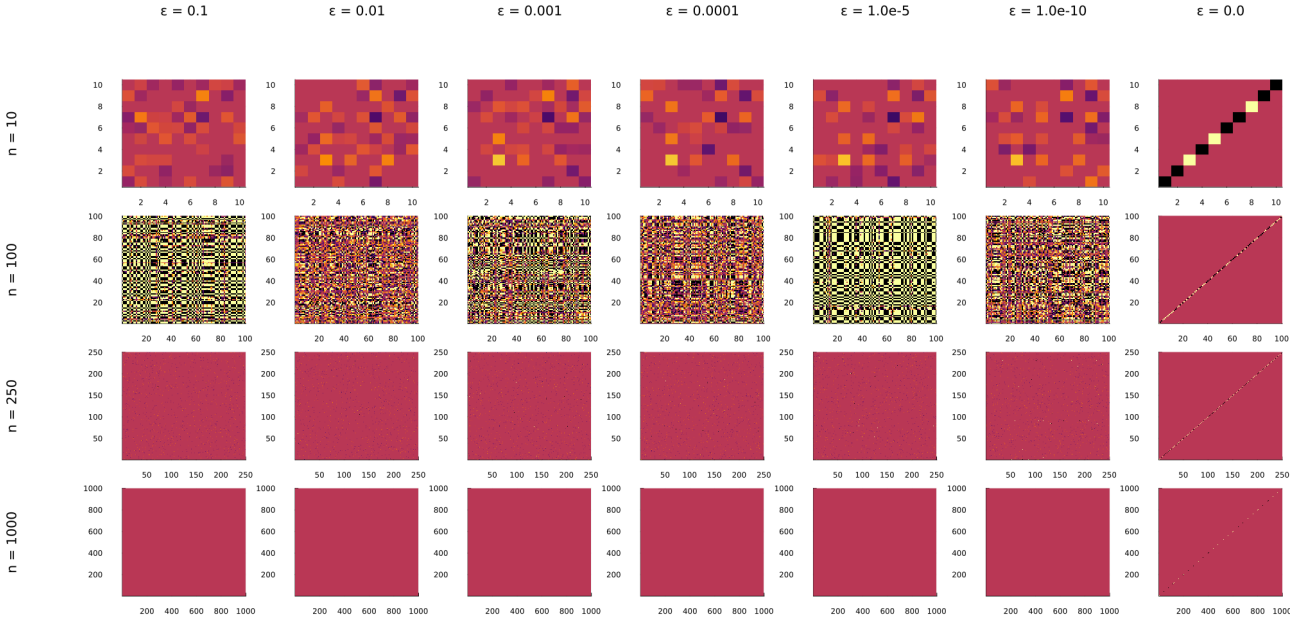
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[50]:



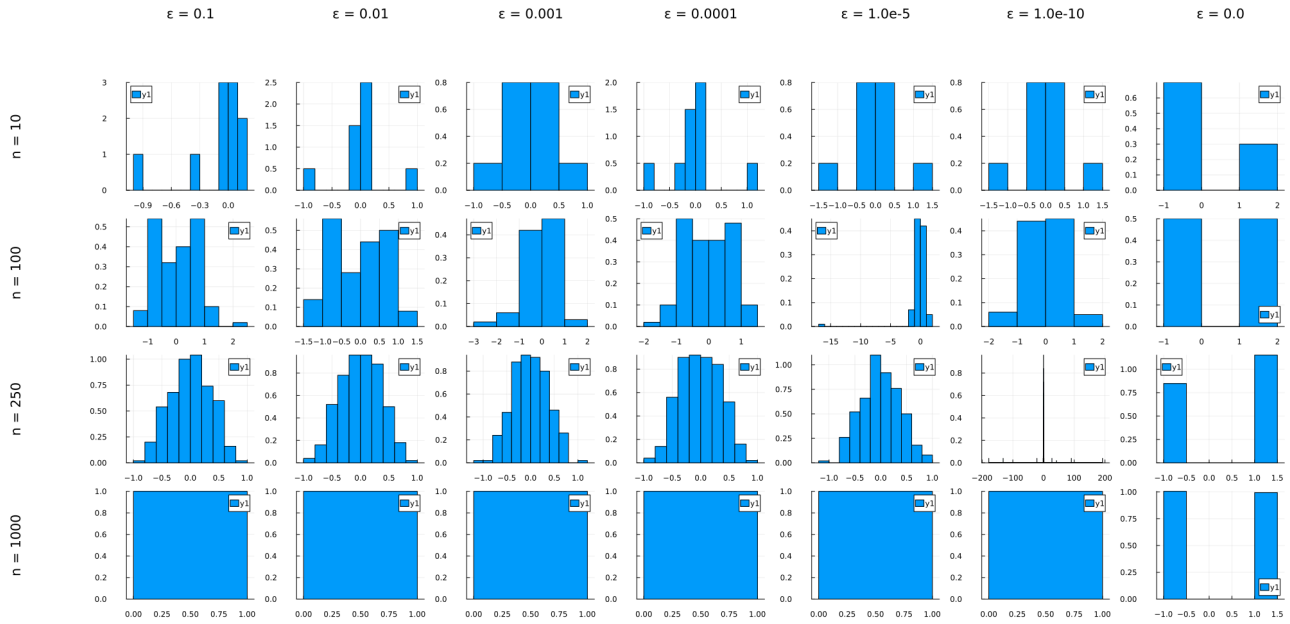
In [51]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[51]:



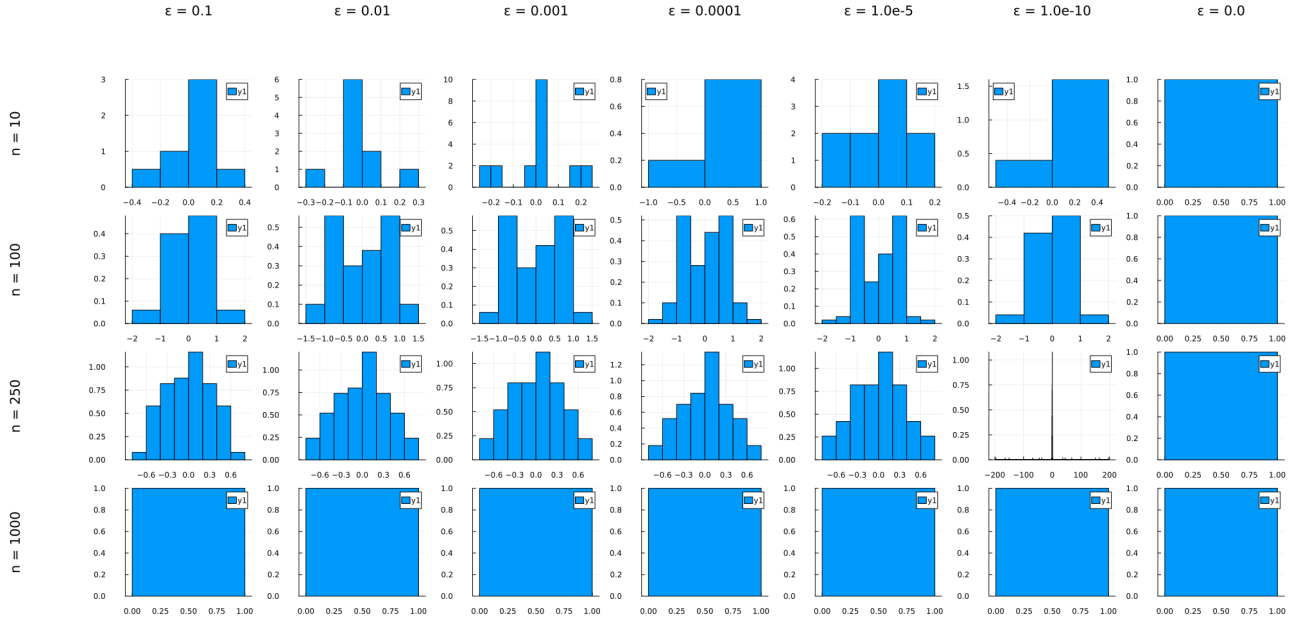
In [52]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[52]:



In [53]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[53]:



Dense Ground Truth

By taking a skew-symmetric matrix $A^T = A$, we get purely imaginary eigenvalues. Then, e^A will have eigenvalues of modulus 1. This enforces boundedness.

LASSO Regression

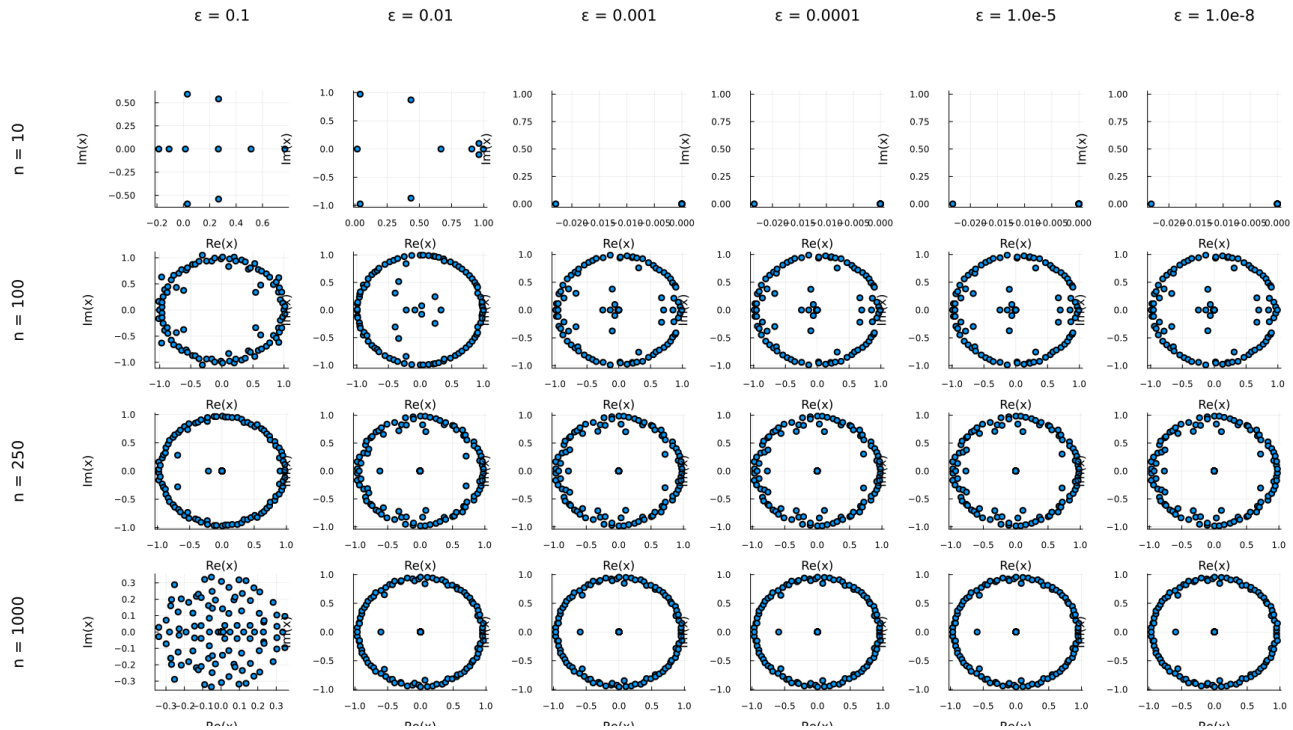
In [54]:
```
Random.seed!(0)

nvals = [10, 100, 250, 1000]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-8]
plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
```

```
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

for (nidx, n) in enumerate(nvals)
    A = sprandn(n, n, 0.1)
    A = exp(Matrix((A - A')/2))
    A = SparseMatrixCSC(A)
    x0 = randn(n);
    clims = (minimum(A), maximum(A))

    ground_truth(x) = A * x

    tf = 100
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        X = copy(X_obs[1:end - 1, :])
        Y = copy(X_obs[2:end, :]);
        A_lasso, λ = fit_A_lasso_glmnet_multi(X, Y;
                                    alpha = 1.0,
                                    nlambda = 150,
                                    lambda_frac = 0.00005)
        plt = scatter((1 + 0im) * eigvals(A_lasso), legend = false)
        plots[nidx, epsidx] = plt

        mod_plt = plot(sort(abs.(eigvals(A_lasso))), legend = false)
        plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
        modulus_plots[nidx, epsidx] = mod_plt


        hm = heatmap(A_lasso, colorbar = false, clim = clims)
        heatmaps[nidx, epsidx] = hm

        rh = histogram(real.((1 + 0im) * eigvals(A_lasso)), normalize = true)
        real_hists[nidx, epsidx] = rh

        ih = histogram(imag.((1 + 0im) * eigvals(A_lasso)), normalize = true)
        im_hists[nidx, epsidx] = ih

        println("n = $n, eps = $eps")
    end
    heatmaps[nidx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
    real_hists[nidx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
    im_hists[nidx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
end
```

```
n = 10, eps = 0.1
n = 10, eps = 0.01
n = 10, eps = 0.001
n = 10, eps = 0.0001
n = 10, eps = 1.0e-5
n = 10, eps = 1.0e-8
n = 100, eps = 0.1
n = 100, eps = 0.01
n = 100, eps = 0.001
n = 100, eps = 0.0001
n = 100, eps = 1.0e-5
n = 100, eps = 1.0e-8
n = 250, eps = 0.1
n = 250, eps = 0.01
n = 250, eps = 0.001
n = 250, eps = 0.0001
n = 250, eps = 1.0e-5
n = 250, eps = 1.0e-8
n = 1000, eps = 0.1
n = 1000, eps = 0.01
n = 1000, eps = 0.001
n = 1000, eps = 0.0001
n = 1000, eps = 1.0e-5
n = 1000, eps = 1.0e-8
```

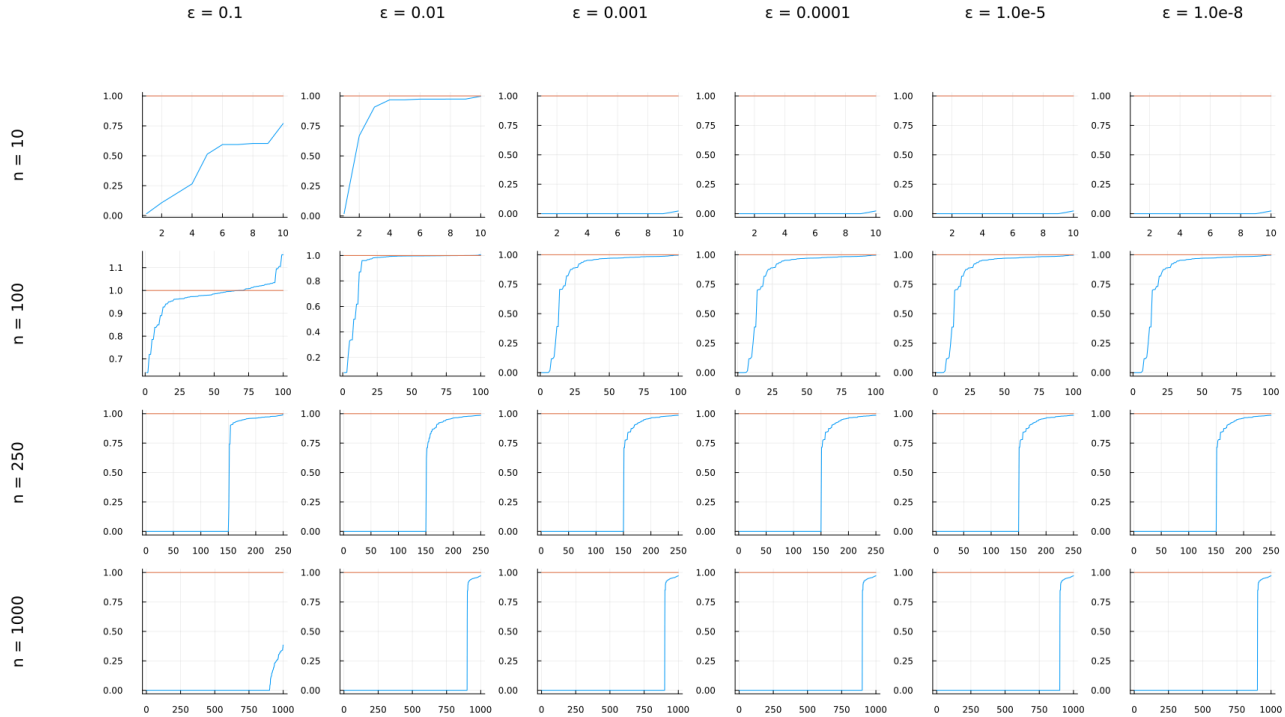
In [55]:
```
param_grid_table(plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[55]:



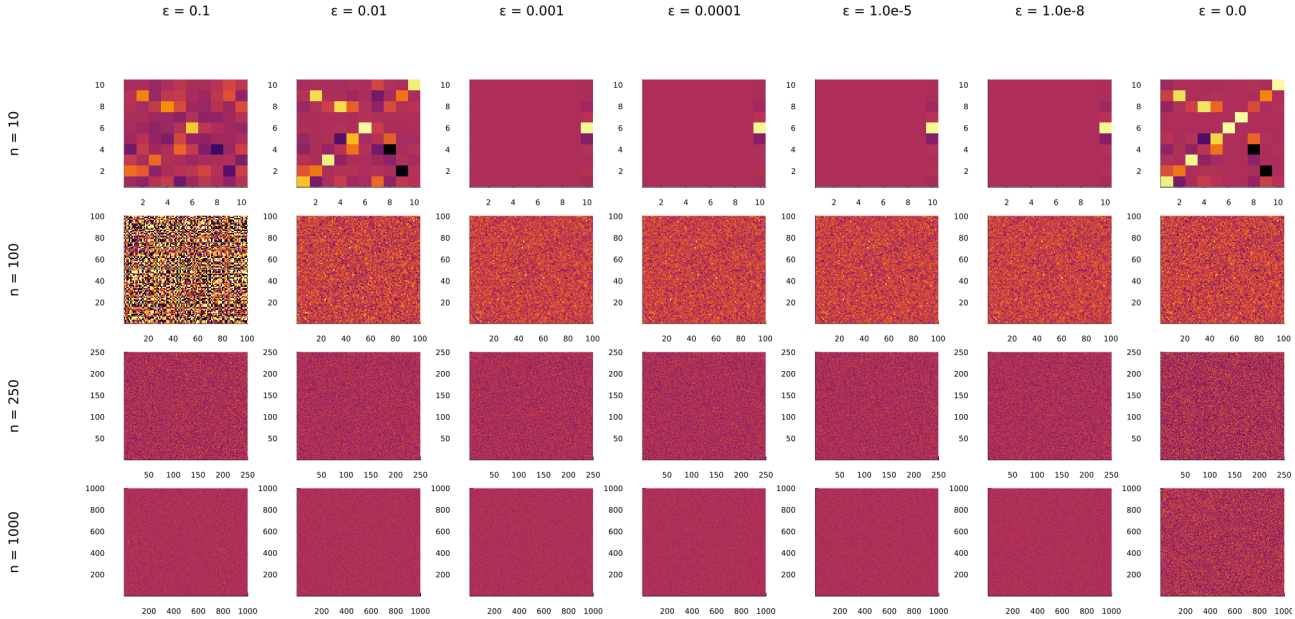
In [56]:
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[56]:



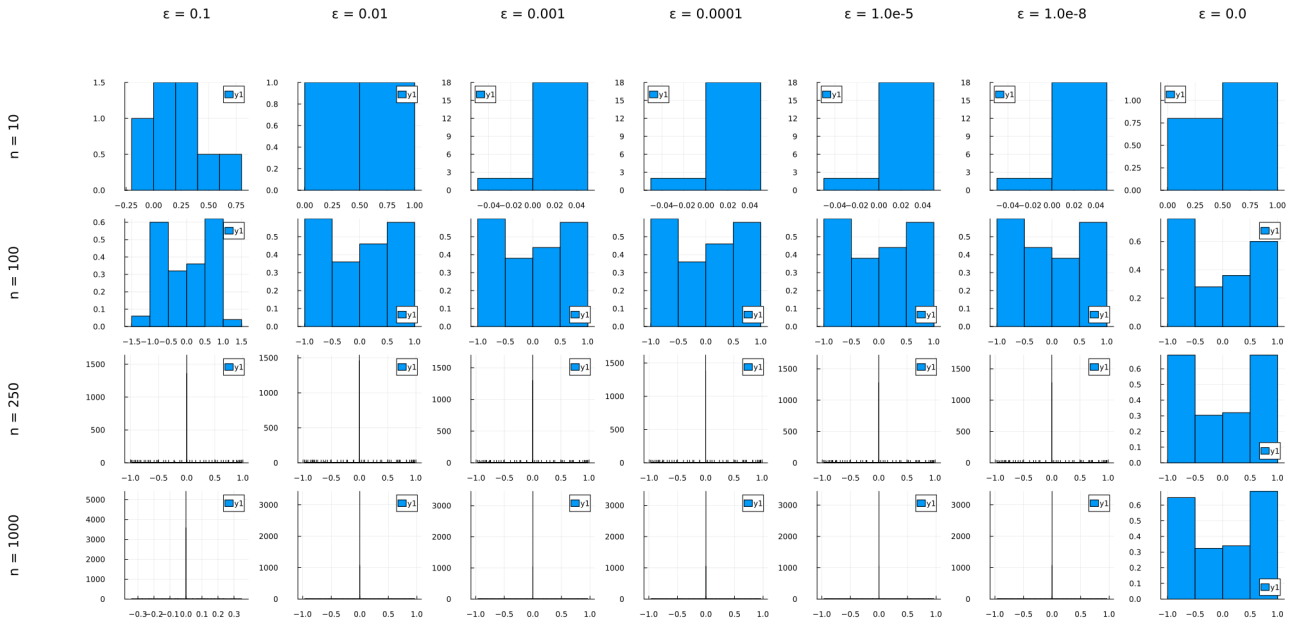
In [57]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[57]:



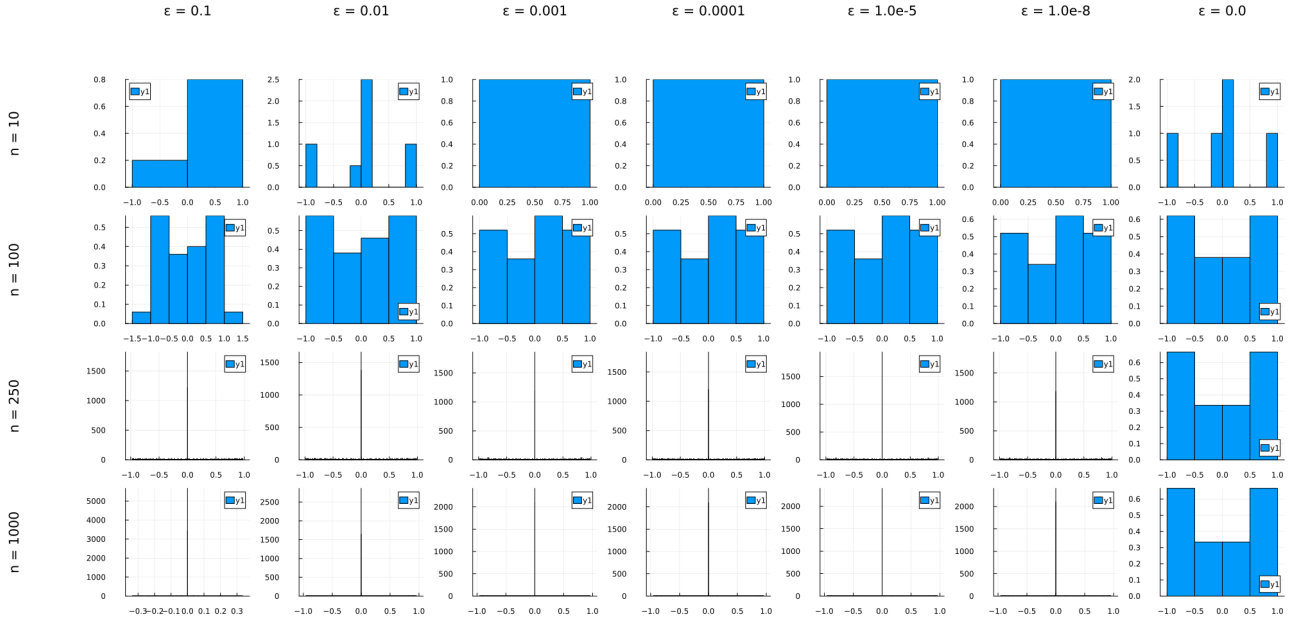
In [58]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[58]:



In [59]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[59]:



Sequential Thresholding

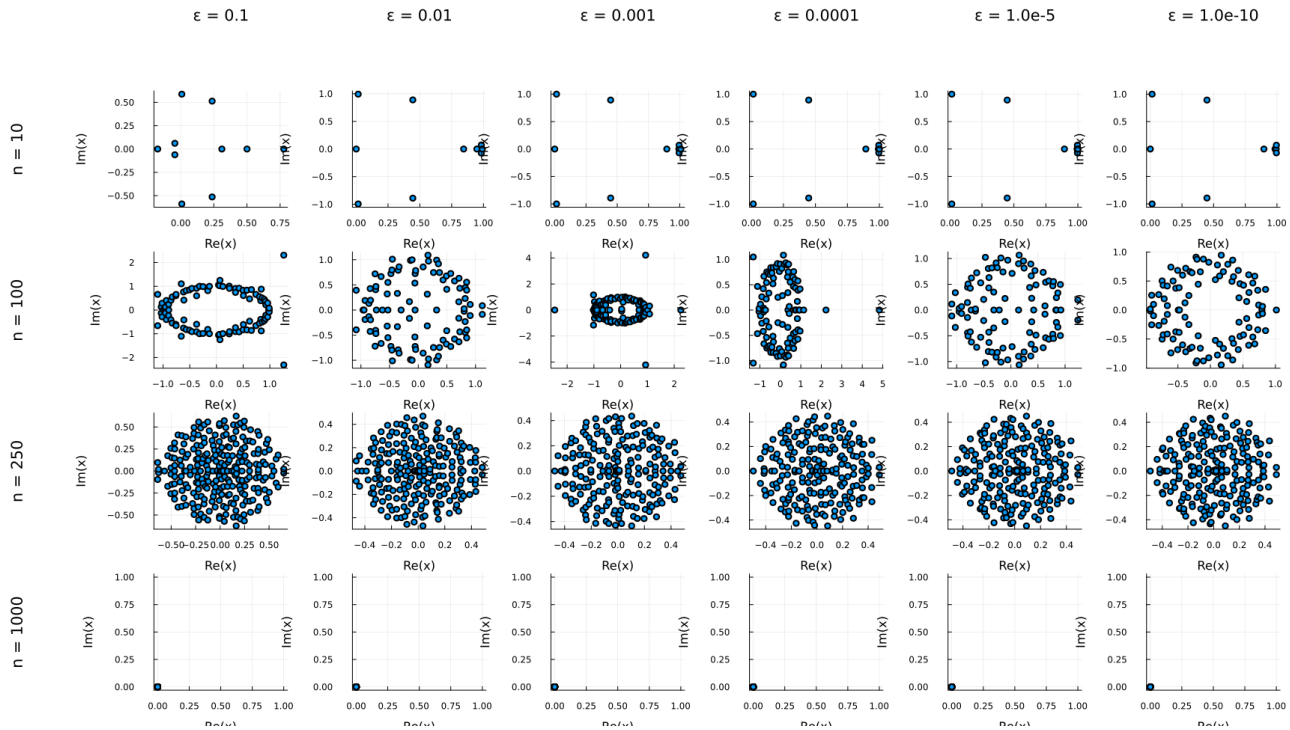
In [60]:
```
Random.seed!(0)

nvals = [10, 100, 250, 1000]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-10]
plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

for (nidx, n) in enumerate(nvals)
    A = sprandn(n, n, 0.1)
    A = exp(Matrix((A - A')/2))
```

```
    A = SparseMatrixCSC(A)
    clims = (minimum(A), maximum(A))
    x0 = randn(n);

    ground_truth(x) = A * x

    tf = 100
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        X = copy(X_obs[1:end - 1, :])
        Y = copy(X_obs[2:end, :]);
        A_st, _ = sequential_threshold_dynamics(X, Y, tol = 1e-6)
        plt = scatter((1+0im) * eigvals(A_st), legend = false)
        plots[nidx, epsidx] = plt

        mod_plt = plot(sort(abs.(eigvals(A_st))), legend = false)
        plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
        modulus_plots[nidx, epsidx] = mod_plt

        hm = heatmap(A_st, colorbar = false, clim = clims)
        heatmaps[nidx, epsidx] = hm

        rh = histogram(real.((1 + 0im) * eigvals(A_st)), normalize = true)
        real_hists[nidx, epsidx] = rh

        ih = histogram(imag.((1 + 0im) * eigvals(A_st)), normalize = true)
        im_hists[nidx, epsidx] = ih
    end
    heatmaps[nidx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
    real_hists[nidx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
    im_hists[nidx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
end
```

In [61]:

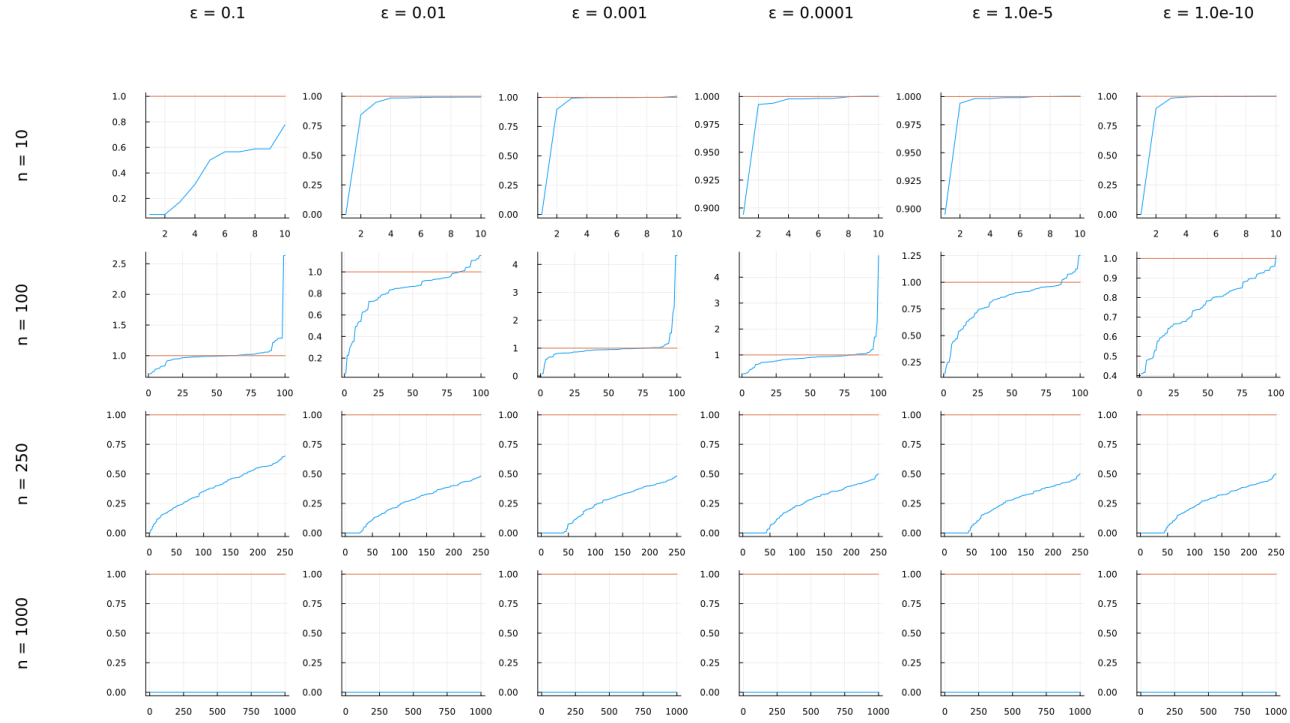
```
param_grid_table(plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[61]:



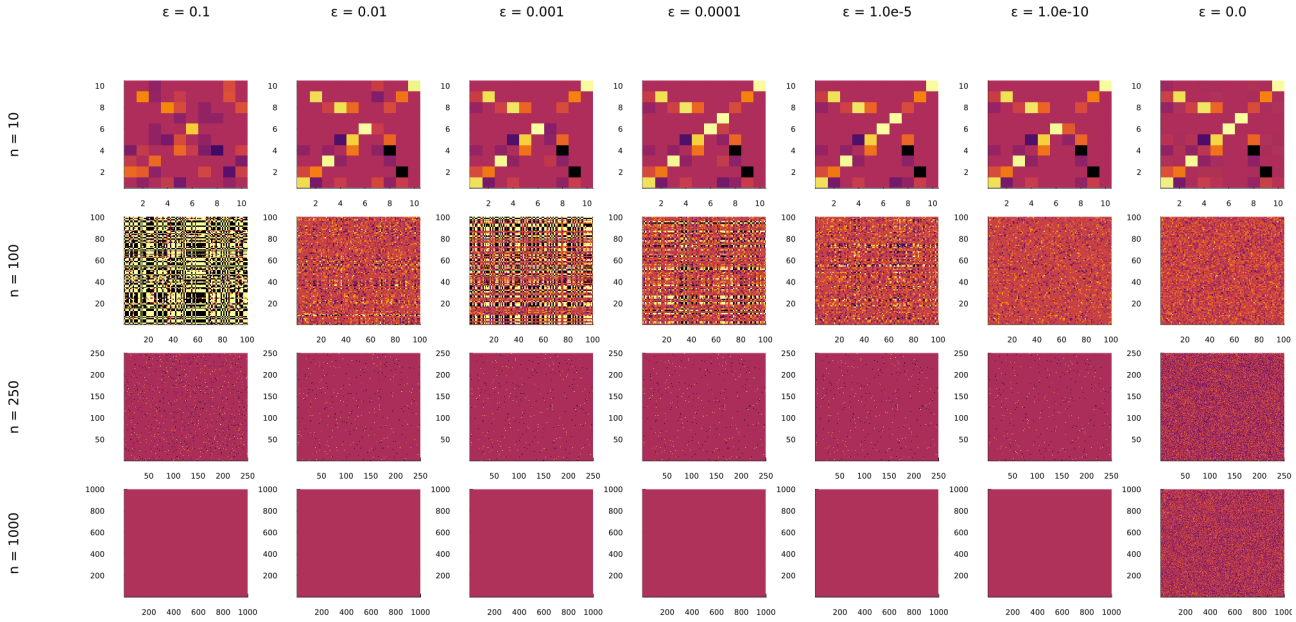
In [62]:
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[62]:



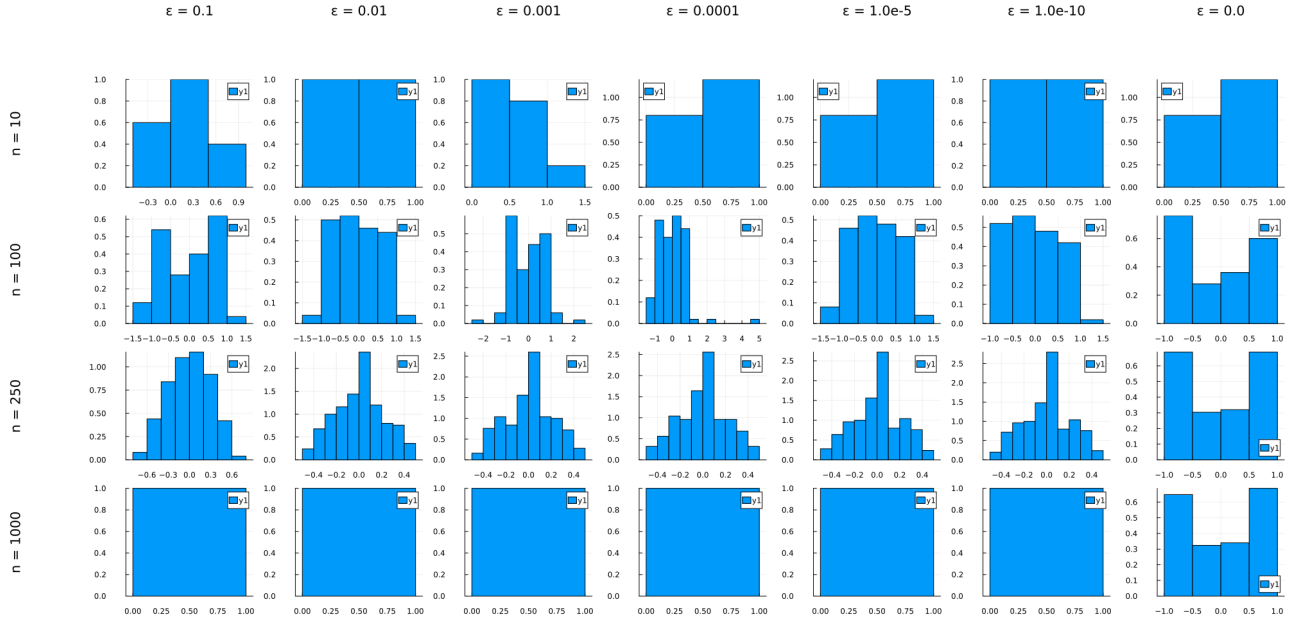
In [63]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[63]:



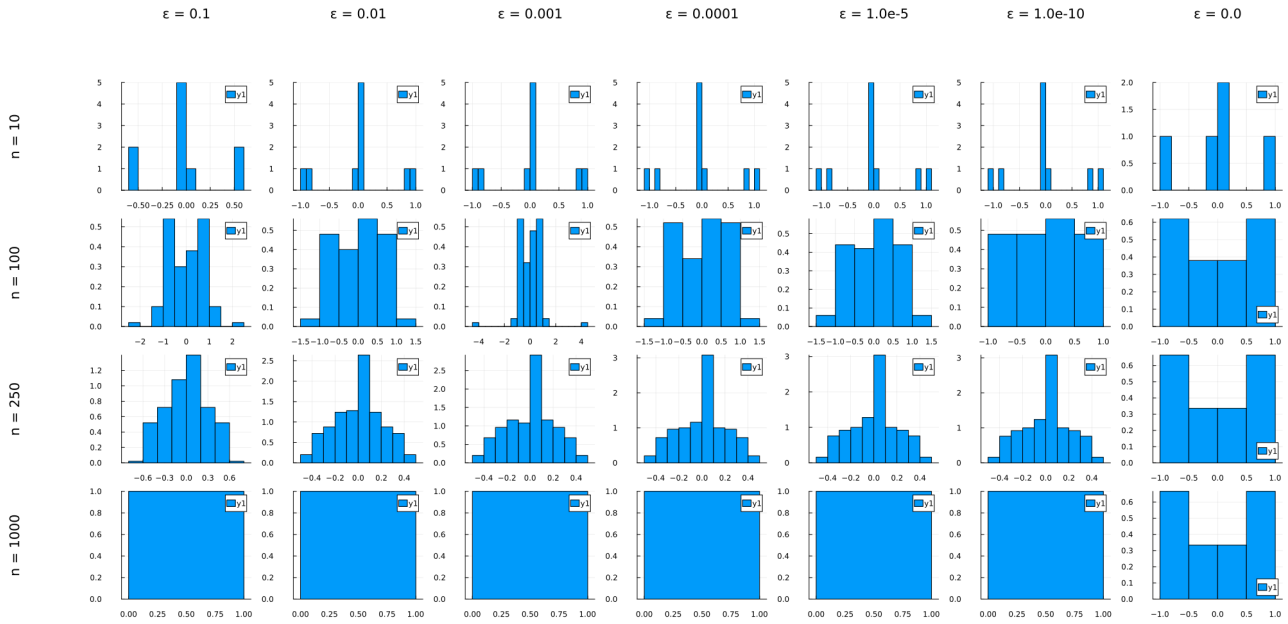
In [64]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[64]:



In [65]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[65]:



Dynamic Mode Decomposition

Sparse Ground Truth

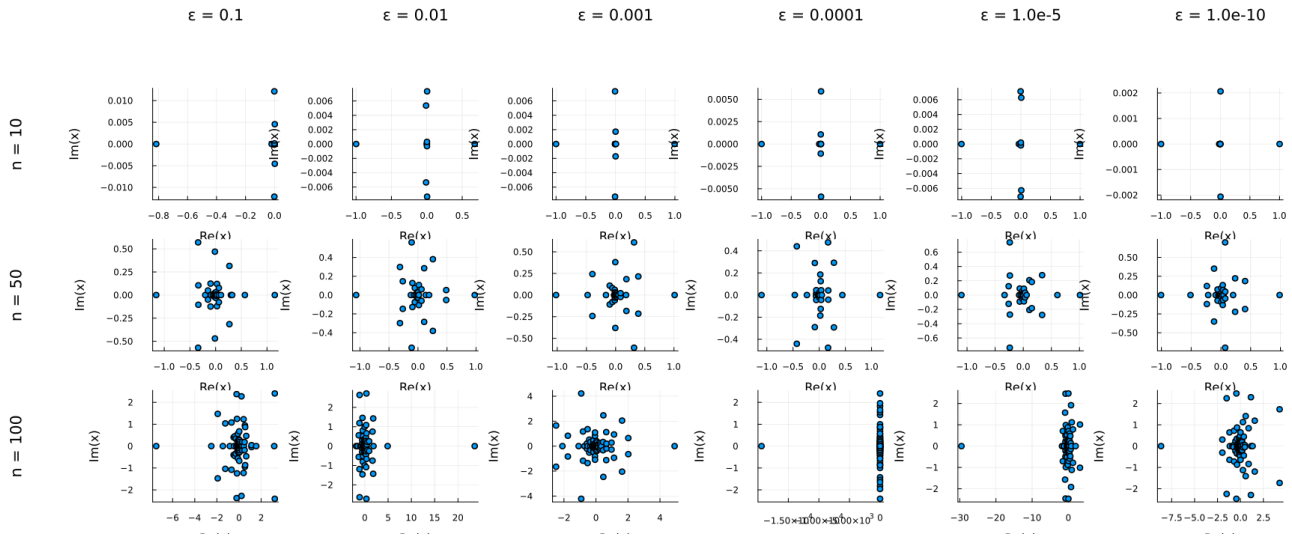
In [ ]:

```
Random.seed!(0)

nvals = [10, 50, 100, 200]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-10]
plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

MFE = Float64[]
MEE = Float64[]
MBE = Float64[]
for (nidx, n) in enumerate(nvals)
    A = diagm(rand([-1, 1], n))
    clims = (minimum(A), maximum(A))
    x0 = randn(n);

    ground_truth(x) = A * x

    tf = 100
    t0 = 0
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        prob = DataDrivenProblem(X_obs')

        # Solve with a DMD algorithm (here: SVD-based DMD)
```

```
        res = solve(prob, DMDSVD(); digits = 3)

        eigA, = get_results(res)  # this is a LinearAlgebra.Eigen factorization

        eigA = eigA.k
        A_dmd = eigA.vectors * Diagonal(eigA.values) * eigA.vectors'

        plt = scatter((1+0im) * eigvals(A_dmd), legend = false)
        plots[nidx, epsidx] = plt

        mod_plt = plot(sort(abs.(eigvals(A_dmd))), legend = false)
        plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
        modulus_plots[nidx, epsidx] = mod_plt

        hm = heatmap(real.(A_dmd), colorbar = false, clim = clims)
        heatmaps[nidx, epsidx] = hm

        rh = histogram(real.((1 + 0im) * eigvals(A_dmd)), normalize = true)
        real_hists[nidx, epsidx] = rh

        ih = histogram(imag.((1 + 0im) * eigvals(A_dmd)), normalize = true)
        im_hists[nidx, epsidx] = ih

        println("n = $n, eps = $eps")
    end
    heatmaps[nidx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
    real_hists[nidx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
    im_hists[nidx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
end
```

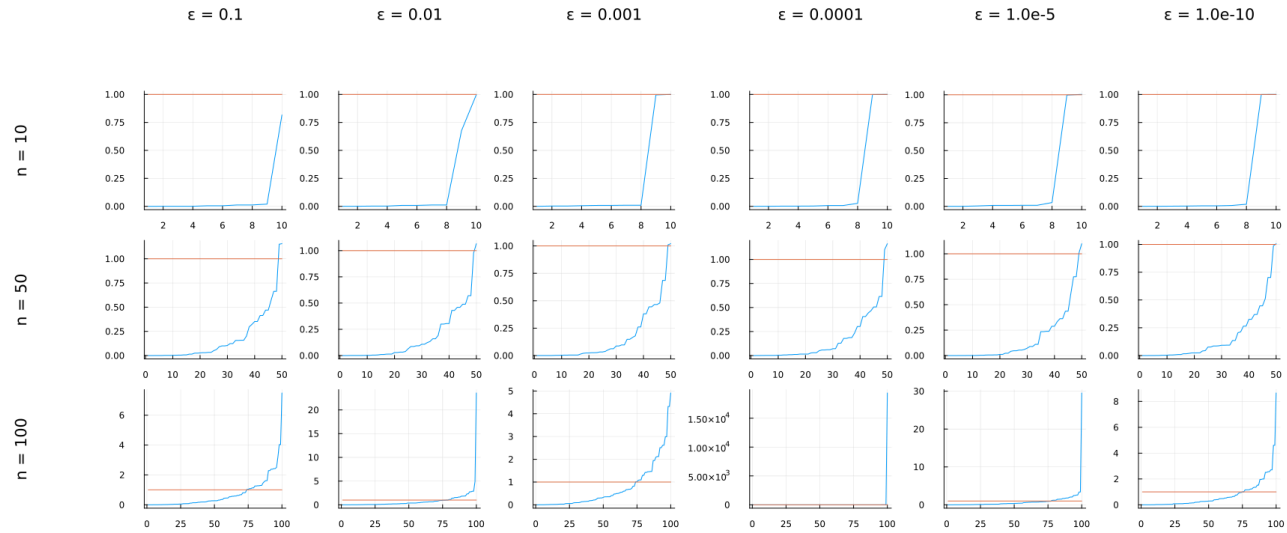
In [67]:
```
param_grid_table(plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[67]:



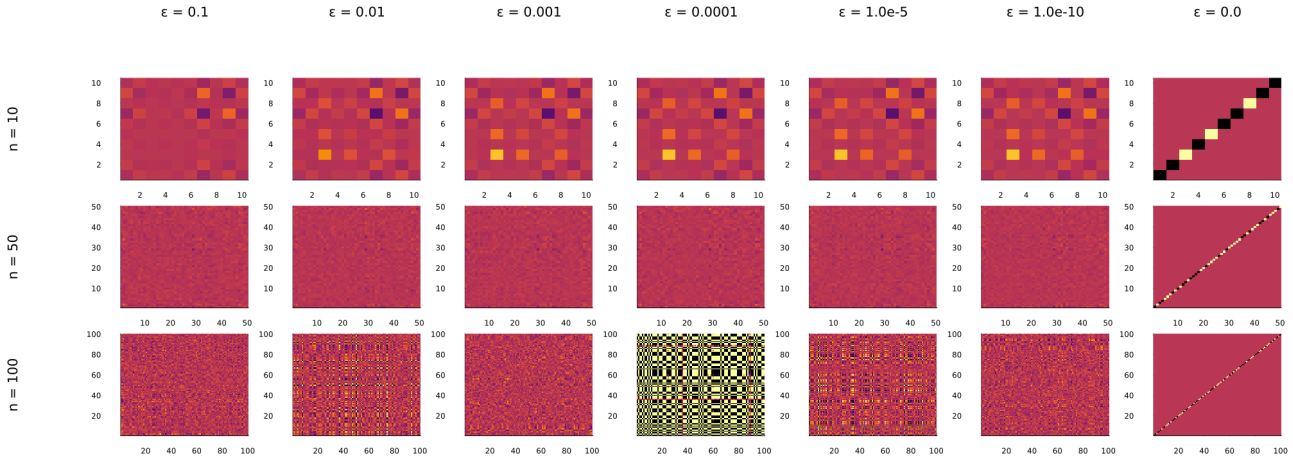
In [68]:
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[68]:



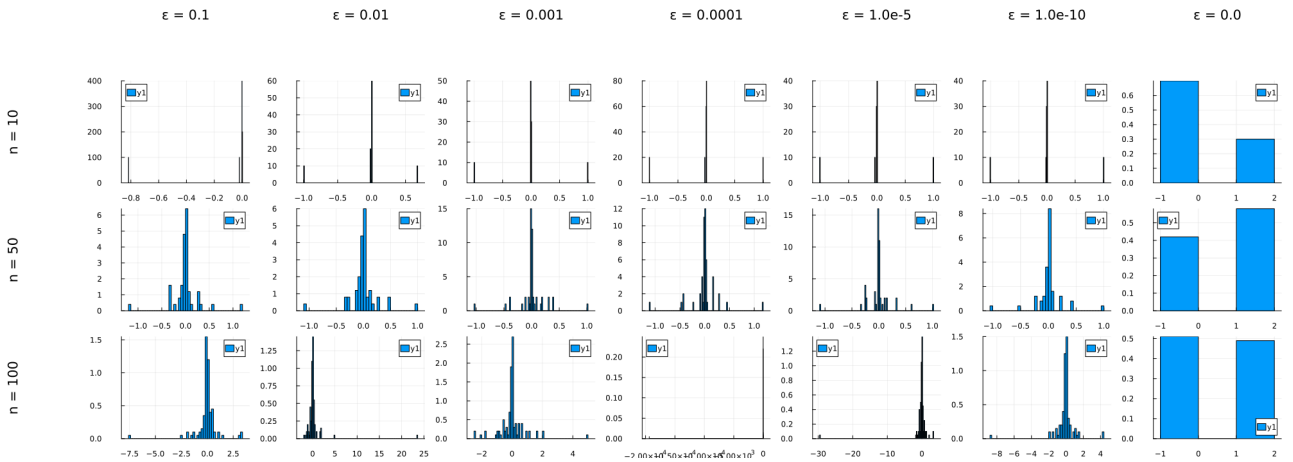
In [69]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[69]:



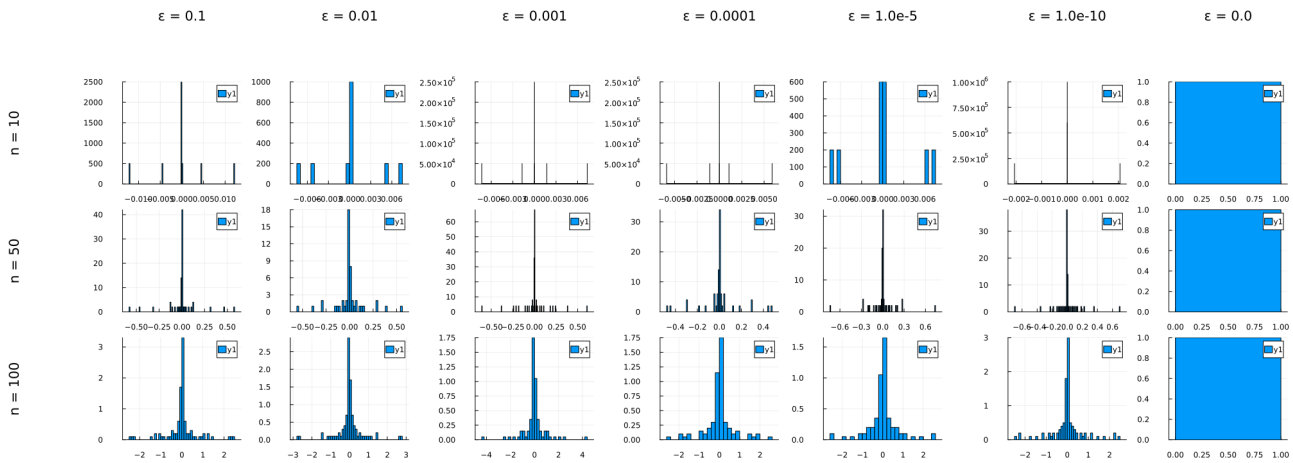
In [70]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[70]:



In [71]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[71]:



Dense Ground Truth

In [88]:

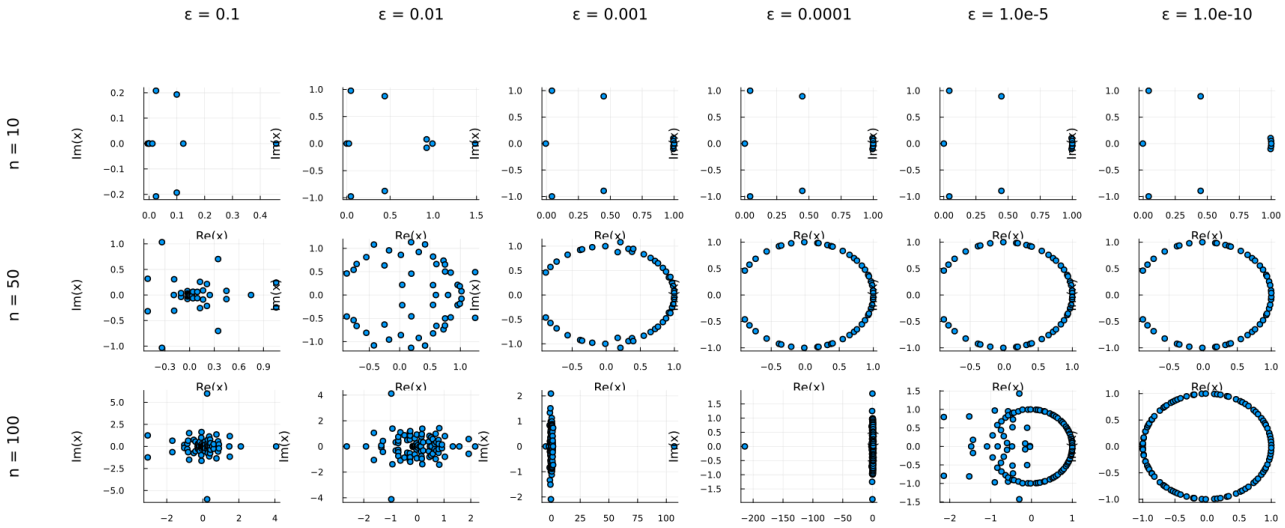
```
Random.seed!(0)

nvals = [10, 50, 100, 200]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-10]
plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

for (nidx, n) in enumerate(nvals)
    A = sprandn(n, n, 0.1)
    A = exp(Matrix((A - A')/2))
    A = SparseMatrixCSC(A)
    clims = (minimum(A), maximum(A))
    x0 = randn(n);

    ground_truth(x) = A * x

    tf = 100
    t0 = 0
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        prob = DataDrivenProblem(X_obs')

        # Solve with a DMD algorithm (here: SVD-based DMD)
        res = solve(prob, DMDSVD(); digits = 3)

        eigA, = get_results(res)  # this is a LinearAlgebra.Eigen factorization

        eigA = eigA.k
        A_dmd = eigA.vectors * Diagonal(eigA.values) * eigA.vectors'

        plt = scatter((1+0im) * eigvals(A_dmd), legend = false)
        plots[nidx, epsidx] = plt
```

```
        mod_plt = plot(sort(abs.(eigvals(A_dmd))), legend = false)
        plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
        modulus_plots[nidx, epsidx] = mod_plt

        hm = heatmap(real.(A_dmd), colorbar = false, clim = clims)
        heatmaps[nidx, epsidx] = hm

        rh = histogram(real.((1 + 0im) * eigvals(A_dmd)), normalize = true)
        real_hists[nidx, epsidx] = rh

        ih = histogram(imag.((1 + 0im) * eigvals(A_dmd)), normalize = true)
        im_hists[nidx, epsidx] = ih

        println("n = $n, eps = $eps")
    end
    heatmaps[nidx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
    real_hists[nidx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
    im_hists[nidx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
end
```

```
n = 10, eps = 0.1
n = 10, eps = 0.01
n = 10, eps = 0.001
n = 10, eps = 0.0001
n = 10, eps = 1.0e-5
n = 10, eps = 1.0e-10
n = 50, eps = 0.1
n = 50, eps = 0.01
n = 50, eps = 0.001
n = 50, eps = 0.0001
n = 50, eps = 1.0e-5
n = 50, eps = 1.0e-10
n = 100, eps = 0.1
n = 100, eps = 0.01
n = 100, eps = 0.001
n = 100, eps = 0.0001
n = 100, eps = 1.0e-5
n = 100, eps = 1.0e-10
```

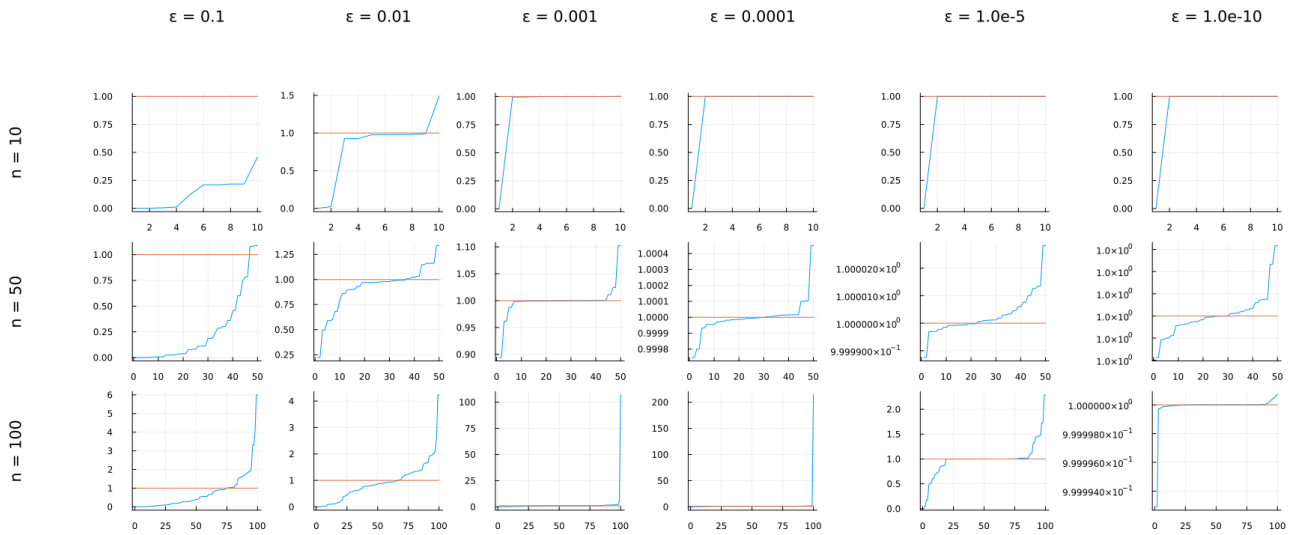
In [89]:
```
param_grid_table(plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[89]:



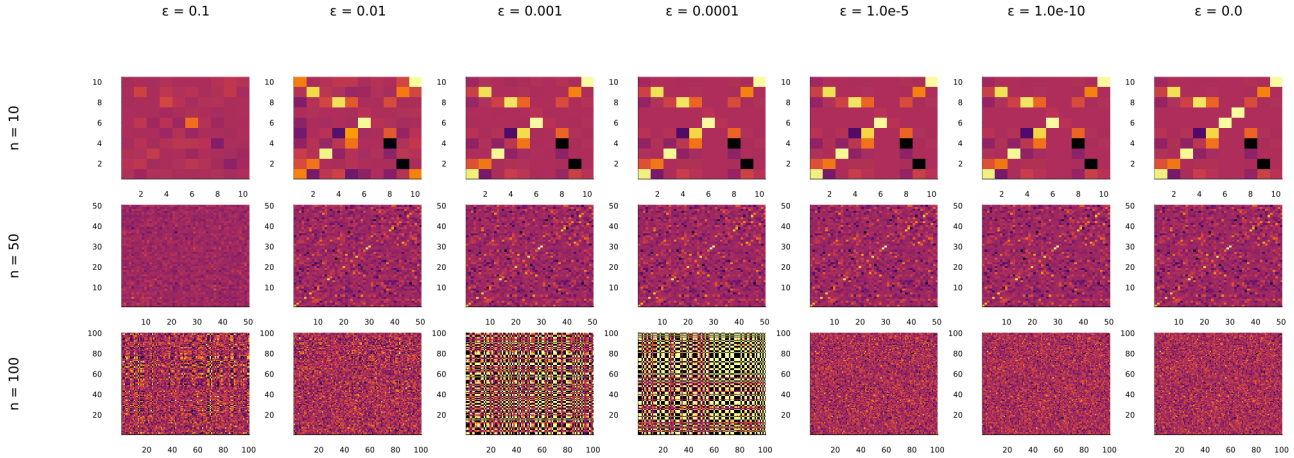
In [90]:
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "n", col_name = "ε")
```

Out[90]:



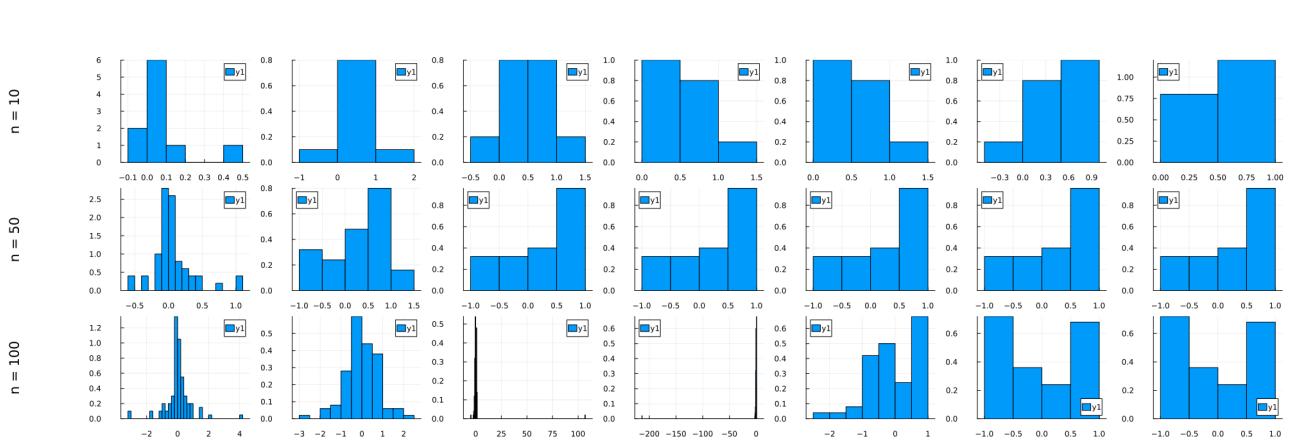
In [91]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[91]:



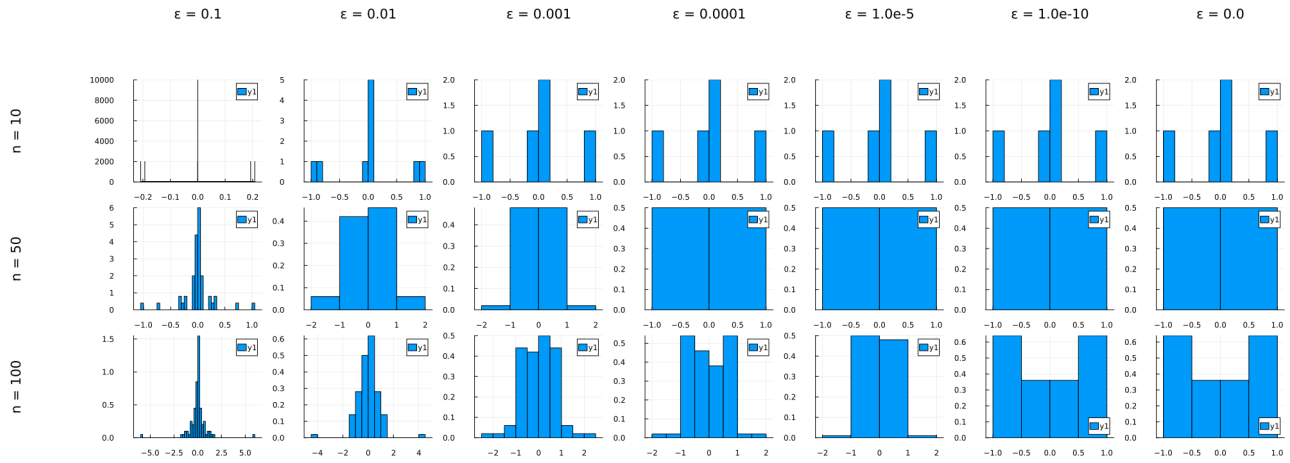
In [92]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[92]:



In [93]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[93]:



Dense Low Rank Ground Truth

In [94]:

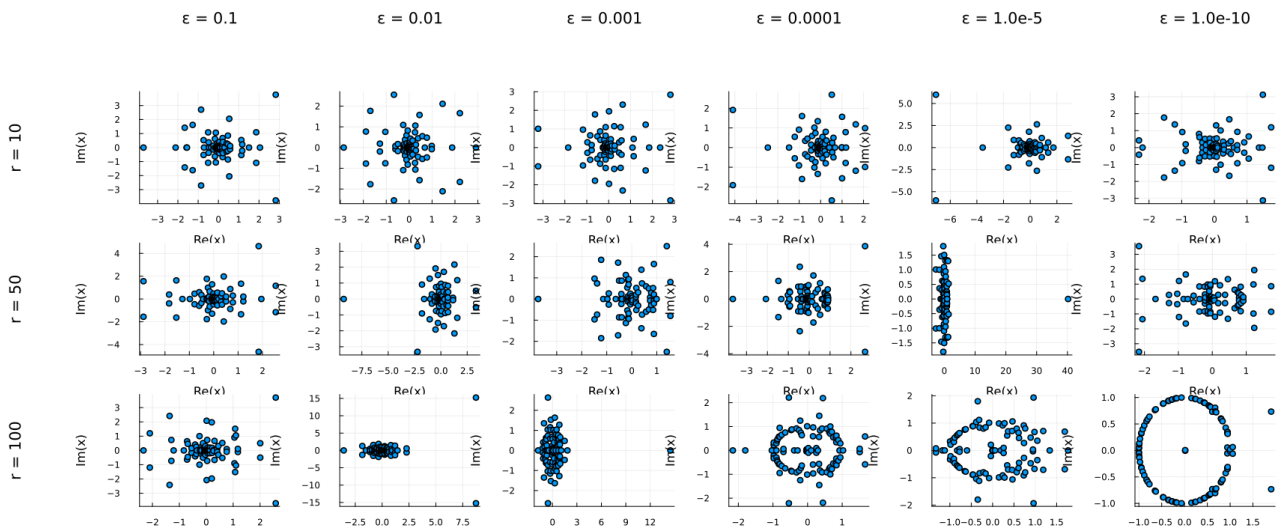
```
Random.seed!(0)

n = 100
rvals = [10, 50, 100]
epsvals = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-10]
plots = Matrix{Plots.Plot}(undef, (size(rvals,1), size(epsvals,1)))
modulus_plots = Matrix{Plots.Plot}(undef, (size(rvals,1), size(epsvals,1)))
heatmaps =  Matrix{Plots.Plot}(undef, (size(rvals,1), size(epsvals,1) + 1))
real_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))
im_hists = Matrix{Plots.Plot}(undef, (size(nvals,1), size(epsvals,1) + 1))

for (ridx, r) in enumerate(rvals)
    A,eigs = random_unit_circle_lowrank_matrix(n, r)
    clims = (minimum(A), maximum(A))
    x0 = randn(n);

    ground_truth(x) = A * x

    tf = 100
    t0 = 0
    X_true = zeros(tf-t0 + 1, n)
    X_true[1, :] .= x0
    for t = 1:tf
        X_true[t + 1,:] .= ground_truth(X_true[t, :])
    end
    for (epsidx, eps) in enumerate(epsvals)
        X_obs = X_true + 1/sqrt(n) * eps * norm(X_true)* randn(size(X_true));

        prob = DataDrivenProblem(X_obs')

        # Solve with a DMD algorithm (here: SVD-based DMD)
        res = solve(prob, DMDSVD(); digits = 3)

        eigA, = get_results(res)  # this is a LinearAlgebra.Eigen factorization

        eigA = eigA.k
        A_dmd = eigA.vectors * Diagonal(eigA.values) * eigA.vectors'

        plt = scatter((1+0im) * eigvals(A_dmd), legend = false)
        plots[ridx, epsidx] = plt
```

```
        mod_plt = plot(sort(abs.(eigvals(A_dmd))), legend = false)
        plot!(mod_plt, sort(abs.(eigvals(Matrix(A)))))
        modulus_plots[ridx, epsidx] = mod_plt

        hm = heatmap(real.(A_dmd), colorbar = false, clim = clims)
        heatmaps[ridx, epsidx] = hm

        rh = histogram(real.((1 + 0im) * eigvals(A_dmd)), normalize = true)
        real_hists[ridx, epsidx] = rh

        ih = histogram(imag.((1 + 0im) * eigvals(A_dmd)), normalize = true)
        im_hists[ridx, epsidx] = ih

        println("r = $r, eps = $eps")
    end
    heatmaps[ridx, size(epsvals,1) + 1] = heatmap(A, colorbar = false)
    real_hists[ridx, size(epsvals,1) + 1] = histogram(real.((1 + 0im) * eigvals(Matrix(A
    im_hists[ridx, size(epsvals,1) + 1] = histogram(imag.((1 + 0im) * eigvals(Matrix(A))
end
```

```
r = 10, eps = 0.1
r = 10, eps = 0.01
r = 10, eps = 0.001
r = 10, eps = 0.0001
r = 10, eps = 1.0e-5
r = 10, eps = 1.0e-10
r = 50, eps = 0.1
r = 50, eps = 0.01
r = 50, eps = 0.001
r = 50, eps = 0.0001
r = 50, eps = 1.0e-5
r = 50, eps = 1.0e-10
r = 100, eps = 0.1
r = 100, eps = 0.01
r = 100, eps = 0.001
r = 100, eps = 0.0001
r = 100, eps = 1.0e-5
r = 100, eps = 1.0e-10
```

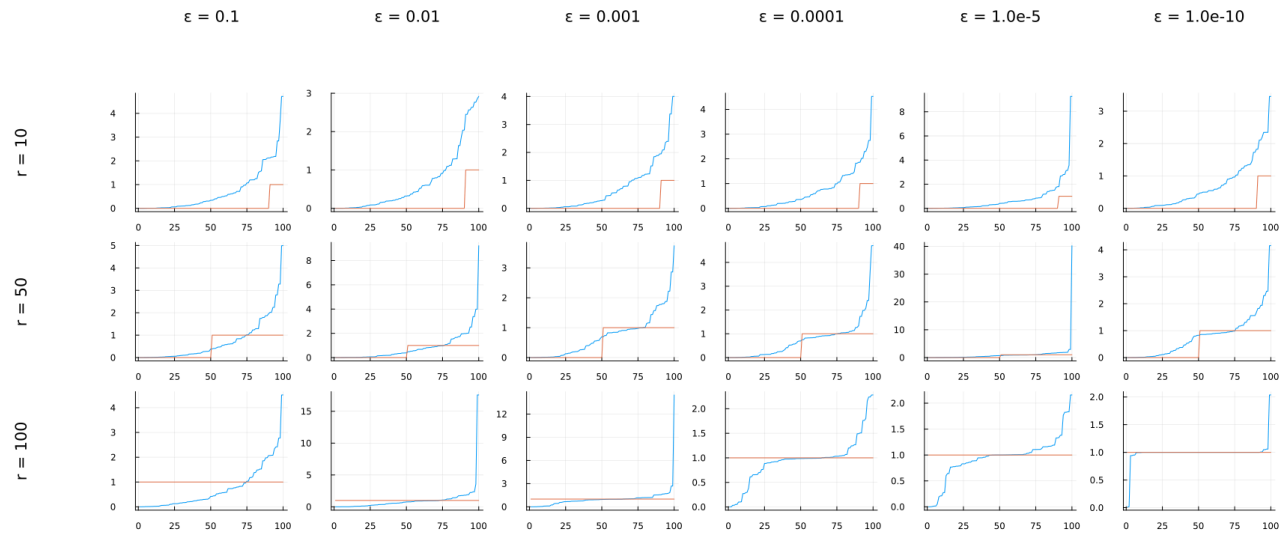
In [95]:
```
param_grid_table(plots, nvals, epsvals, row_name = "r", col_name = "ε")
```

Out[95]:



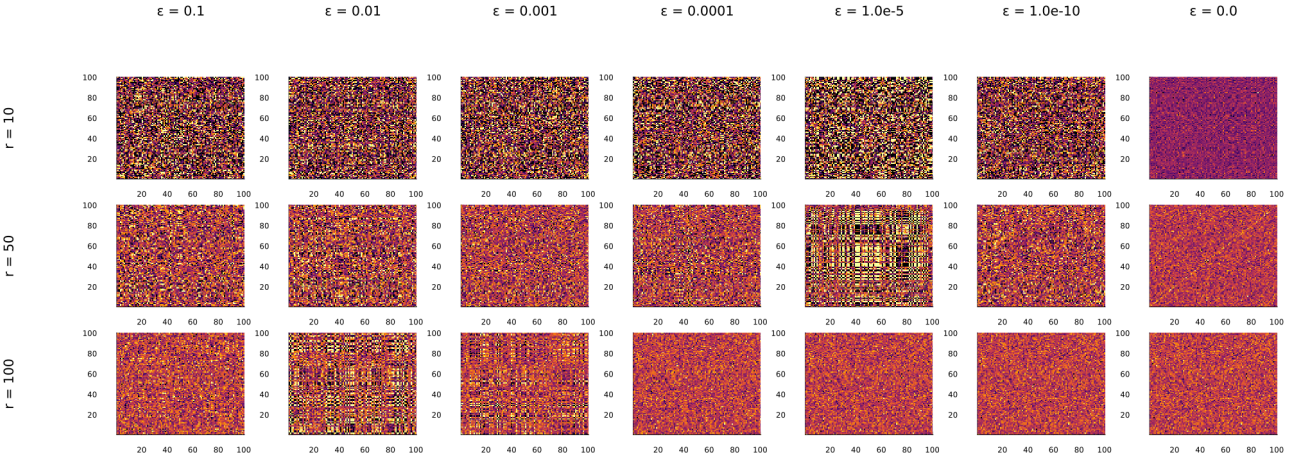
In [96]:
```
param_grid_table(modulus_plots, nvals, epsvals, row_name = "r", col_name = "ε")
```

Out[96]:



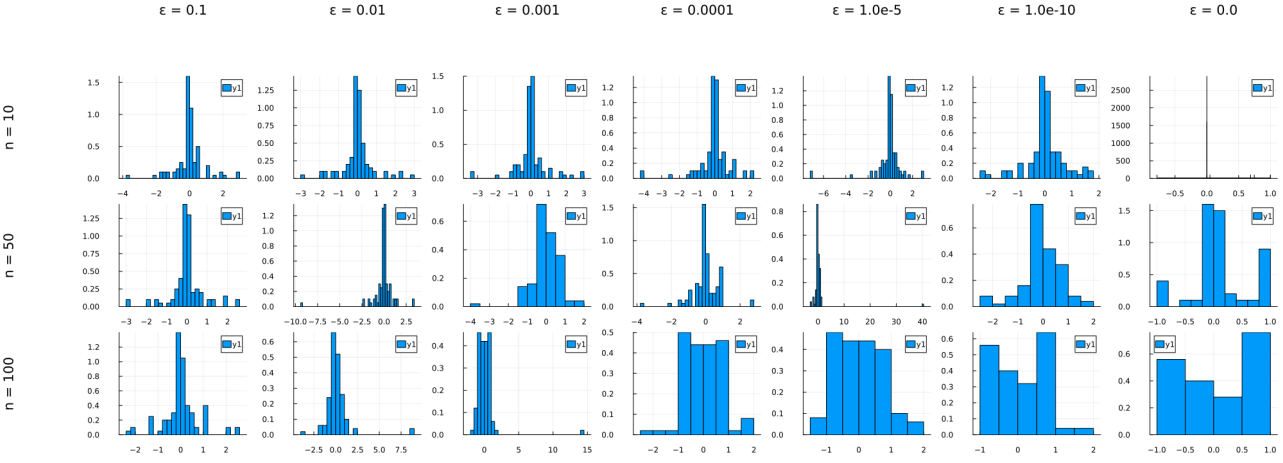
In [97]:
```
param_grid_table(heatmaps, nvals, [epsvals; 0.0], row_name = "r", col_name = "ε")
```

Out[97]:



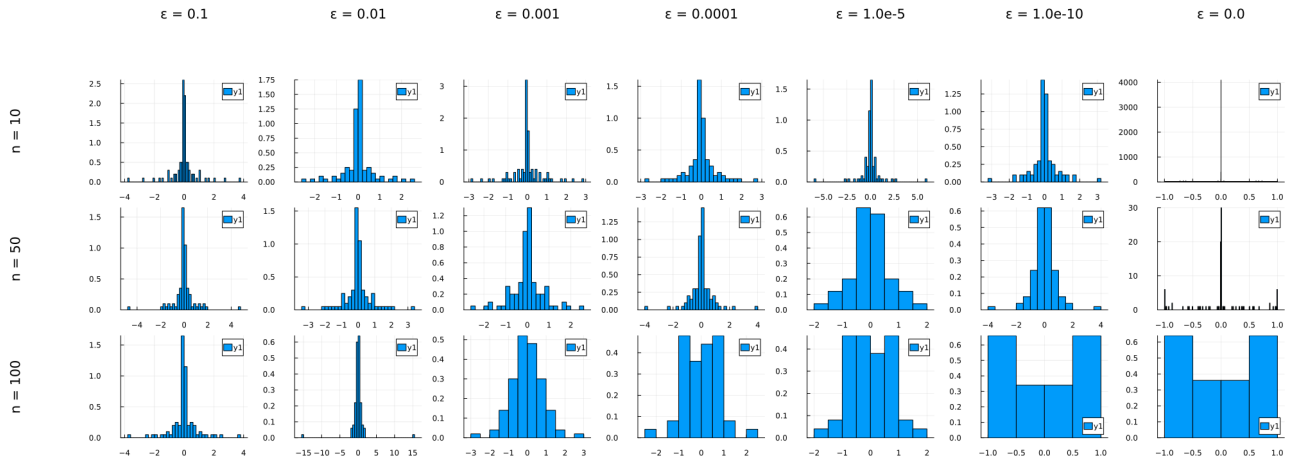
In [98]:
```
param_grid_table(real_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[98]:



In [99]:
```
param_grid_table(im_hists, nvals, [epsvals; 0.0], row_name = "n", col_name = "ε")
```

Out[99]:



Discussion

Discussion of discrete-time dynamical system experiments

In the discrete-time experiments, we study linear systems of the form

$$x_{t+1} = Ax_t$$

and ask how well different regression procedures recover both the matrix A and its spectrum from noisy trajectory data.

For each experiment, we:

- Choose a "ground truth" matrix A.
- Generate a trajectory $x_0, x_1, \ldots, x_T$ via $x_{t+1} = Ax_t$.
- Add i.i.d. Gaussian noise of level ε to the trajectory.
- Fit a model to the noisy snapshot pairs (x_t, x_{t+1}) using LASSO, sequential thresholding, or a related method.

The results are shown as grids of plots with:

- **Rows:** state dimension n,
- **Columns:** noise level ε.

For each (n, ε) we typically visualize:

- A scatter plot of the eigenvalues of the learned matrix,
- A plot of the sorted eigenvalue moduli, often overlaid with those of the true matrix,
- A heatmap of the learned matrix (with the true matrix shown in the last column),
- Histograms of the real and imaginary parts of the eigenvalues (with the true distribution shown in the last column).

This layout makes it easy to see how increasing either the noise level or the dimension affects both the structure of the learned operator and its spectrum.

Sparse diagonal ground truth

In the sparse case, the ground-truth matrix A is diagonal with random ± 1 entries. All eigenvalues are real and equal to ± 1, so:

- The dynamics are **marginally stable** (no decay or growth in magnitude),
- The system is completely **decoupled** (each coordinate evolves independently).

The heatmap of the true A shows a clean diagonal of ± 1's and zeros elsewhere, and the true modulus plot is a flat line at $|\lambda| = 1$.

LASSO on sparse ground truth

For small noise and moderate dimension, LASSO recovers this structure reasonably well:

- The eigenvalue scatter plots show points clustered near ± 1 on the real axis.
- The sorted modulus curves lie close to the flat "all ones" profile of the true spectrum.
- The heatmaps show a visible diagonal with relatively few off-diagonal entries.

As we increase the noise level ε, two systematic effects appear:

1. **Shrinkage of diagonal entries**

   LASSO penalizes the ℓ_1 norm of A, so it shrinks all coefficients, including the true nonzero diagonal entries. This shows up as:

   - Lighter colors on the diagonal in the heatmaps, compared to the ground truth.
   - Eigenvalues pulled inside the unit circle: the modulus plots now show $|\lambda_i| < 1$ for many eigenvalues.

   In other words, LASSO turns a marginally stable system into an artificially **damped** one.

2. **Spurious coupling and complex eigenvalues**

   Noise plus the ℓ_1 penalty also introduce small off-diagonal entries:

   - The learned matrices develop weak couplings between coordinates.
   - The eigenvalues are no longer strictly real: the scatter plots show eigenvalues spreading slightly into the complex plane.
   - Histograms of real/imaginary parts show broadened real parts around ± 1 and a nontrivial spread of imaginary parts around 0.

These issues become more pronounced as dimension n increases while the trajectory length stays fixed. In higher rows (larger n):

- There is less data per parameter,
- The LASSO penalty dominates more strongly,
- Many diagonal entries are driven toward zero,
- More off-diagonal entries appear.

Spectrally, the eigenvalues cluster deep inside the unit circle, so the learned system looks strongly **contracting** and more **mixed** than the true sparse, decoupled map.

Sequential thresholding on sparse ground truth

Sequential thresholding behaves quite differently. Starting from an ordinary least-squares fit, we:

1. Compute an unregularized estimate of A,
2. Hard-threshold small entries,
3. Optionally refit on the surviving support.

At low and moderate noise levels, the plots show that:

- The heatmaps almost perfectly reproduce the diagonal pattern of the true A,
- Off-diagonal entries are largely eliminated,
- The eigenvalue scatter plots show points very close to ± 1 on the real axis,
- The modulus plots remain nearly flat at $|\lambda| \approx 1$ across a wide range of (n, ε).

The key difference from LASSO is that once a coefficient survives thresholding, it is **not shrunk** toward zero. As a result:

- The nonzero entries remain close to their least-squares values,
- The eigenvalue moduli of the learned matrix stay close to the true values when the sparsity pattern is correctly recovered.

However, failure modes appear when noise is large or dimension is very high:

- Some diagonal entries are pushed below the threshold and set to zero.
  - These coordinates then behave like modes with eigenvalue near 0.
  - The modulus plots show a mix of eigenvalues near 1 and others near 0.
- Occasionally, noisy off-diagonal entries exceed the threshold and survive.
  - The heatmaps show isolated off-diagonal “pixels”.
  - The eigenvalue scatter plots gain small clusters of complex eigenvalues away from the real axis.

So, in the sparse-ground-truth setting:

- **LASSO**: smoother but biased spectra (eigenvalues consistently inside the unit circle),
- **Sequential thresholding**: nearly unbiased when it succeeds, but can fail abruptly by zeroing true entries if noise or thresholds are not well controlled.

Dense orthogonal ground truth

In the dense case, the ground-truth matrix is constructed as

$$A = \exp\left(\frac{B - B^\top}{2}\right),$$

where B is a random sparse matrix. The matrix

$$\frac{B - B^\top}{2}$$

is skew-symmetric and has purely imaginary eigenvalues, so its exponential A is **orthogonal** with all eigenvalues lying on the unit circle. Thus, the true dynamics are:

- Norm-preserving (no net growth or decay),
- Generally non-sparse and more “rotational” or oscillatory.

The heatmap of the true A shows a dense pattern of entries, and the eigenvalue scatter lies on the unit circle with a roughly uniform angular distribution. The modulus plot is again flat at $|\lambda| = 1$.

LASSO on dense ground truth

Here, the sparsity prior is **wrong**: we apply a sparsity-promoting method to a genuinely dense, orthogonal system. The plots reflect this mismatch:

- The heatmaps of the learned A show many entries driven close to zero, making the estimated matrix much sparser than the truth.
- Orthogonality is completely lost; the fitted operator no longer preserves norms.
- The eigenvalues are pulled well inside the unit circle, even for modest noise:
  - The modulus curves sit significantly below 1 and exhibit a broad spread.
  - The eigenvalue scatter plots show a cloud of points strictly inside the unit disk.

The histograms of real and imaginary parts confirm this:

- Real parts shift toward 0,
- Imaginary parts shrink in magnitude.

As the dimension n increases, this effect intensifies: more parameters are penalized, so more entries are suppressed. The learned dynamics appear more and more **dissipative**, even though the true system is energy-preserving.

From the perspective of stability analysis, this is problematic:

- A truly conservative system is misidentified as strongly stable and decaying.
- Time scales and amplitudes inferred from the learned spectrum can be severely underestimated.

Sequential thresholding on dense ground truth

Sequential thresholding also struggles in the dense case, but in a distinct way. Starting from a dense least-squares estimate and then thresholding:

- Many small entries are removed,
- A scattered pattern of relatively large entries remains,
- The heatmaps show a “patchy” matrix with no evident orthogonal structure.

Spectrally, the eigenvalues of the thresholded matrices are no longer concentrated on the unit circle:

- Some modes become overly damped ($|\lambda| < 1$) due to missing couplings and loss of energy in the model.
- Some modes can become slightly unstable ($|\lambda| > 1$), because the remaining large entries are unconstrained and no longer satisfy any orthogonality or conservation properties.

The modulus plots thus show a broader distribution around 1, often with outliers both inside and outside the unit circle. Unlike the sparse case, there is **no correct sparse support** to recover, so sequential thresholding cannot "lock onto" a true pattern; it simply produces some sparse approximation whose spectral behavior may differ qualitatively from that of the dense orthogonal system.

Summary of discrete-time experiments with sparse models

The discrete-time dynamical system experiments highlight two main themes:

1. **Noise interacts strongly with the regularization method.**
    - Gaussian noise combined with an ℓ_1 penalty (LASSO) systematically pulls eigenvalues inward and introduces spurious coupling.
    - Sequential thresholding avoids shrinkage bias on surviving entries and can preserve the spectrum well when the sparsity assumption is correct, but it is sensitive to threshold choice and can fail sharply when noise is large.
2. **Structural mismatch can be as harmful as noise.**
    - Applying sparsity-promoting methods to a truly dense, orthogonal system leads to learned matrices whose spectra are qualitatively wrong: strongly damped with LASSO, or irregular with both overly damped and slightly unstable modes under thresholding.
    - These distortions become more pronounced as dimension and noise increase.

These observations set the stage for comparing with low-rank methods (DMD), and they emphasize that interpreting stability, time scales, and dominant modes from data-driven models requires understanding both the noise level and the structural priors implicit in the chosen regression technique.

Discussion of DMD experiments

In the DMD experiments, we again consider discrete-time linear systems of the form

$$x_{t+1} = Ax_t,$$

and study how well Dynamic Mode Decomposition recovers the spectral properties of A from noisy trajectory data.

The general procedure is:

- Choose a ground-truth matrix A (for example, a diagonal matrix with entries ± 1).
- Generate a trajectory $x_0, x_1, \ldots, x_T$ via $x_{t+1} = Ax_t$.
- Add i.i.d. Gaussian noise of level ε to the observed snapshots.
- Form snapshot matrices

$$X = [x_0, x_1, \ldots, x_{T-1}], \quad Y = [x_1, x_2, \ldots, x_T],$$

and apply a standard SVD-based DMD algorithm to obtain an approximate propagator A_{DMD} and its eigenvalues.

As in the earlier sections, the results are organized in grids where:

- **Rows** correspond to the state dimension n,
- **Columns** correspond to the noise level ε,

and for each (n, ε) we visualize:

- The eigenvalues of A_{DMD} in the complex plane,
- The sorted eigenvalue moduli, compared to the true spectrum,
- A heatmap of the real part of A_{DMD},
- Histograms of the real and imaginary parts of the eigenvalues.

Behavior in the low-noise regime

In the simplest setting, the ground-truth A is diagonal with entries ± 1, so:

- All eigenvalues are exactly at $\lambda = \pm 1$ on the real axis,
- The dynamics are marginally stable and decoupled.

When ε is very small and the dimension n is moderate, the DMD results behave as expected:

- **Eigenvalue scatter plots** show tight clusters around ± 1 on the real axis.
- **Modulus plots** show that the sorted $|\lambda_i(A_{\mathrm{DMD}})|$ curve closely tracks the ideal spectrum, which is identically 1 for all eigenvalues.
- **Heatmaps of A_{DMD}** show a structure that is close to diagonal, even though DMD does not explicitly enforce sparsity.
- **Histograms of eigenvalue parts** display sharp peaks in the real part near -1 and 1, and the imaginary parts remain concentrated near 0.

In this regime, DMD is accurately capturing the dominant spectral features of the underlying linear map, despite the presence of small measurement noise and the fact that only a single trajectory is used. The method effectively recovers the neutrally stable modes at ± 1 and does not introduce significant artificial growth or decay.

Emergence of spectral distortion with increasing noise

As the noise level ε increases, the spectral picture changes in systematic ways:

1. **Radial shrinkage toward the origin**

   With higher noise, the eigenvalues of A_{DMD} tend to move inside the unit circle:

   - The eigenvalue scatter plots show points drifting inward from ± 1 toward the origin.

- The modulus plots no longer lie flat at 1: many eigenvalue moduli drop below 1, indicating artificial damping.

This can be interpreted as DMD attributing part of the variability in the snapshots to "decaying modes" that are not present in the true system. In other words, noise is being absorbed as effective dissipation in the learned linear operator.

2. **Appearance of complex eigenvalues**

   Even though the true eigenvalues are purely real (at ± 1), noisy data cause the DMD operator to develop complex-conjugate pairs:

   - The scatter plots show eigenvalues moving off the real axis into the complex plane.
   - The histograms of imaginary parts broaden from a sharp spike at 0 to a nontrivial distribution.

   This reflects the fact that the least-squares fit that underlies DMD is no longer exactly diagonalizable over the reals; small rotational components appear in 2D subspaces, and these manifest as complex eigenvalues with moduli close to (but often slightly less than) 1.

3. **Loss of diagonal-like structure**

   The heatmaps of A_{DMD} illustrate that the operator becomes more dense as ε grows:

   - Off-diagonal entries increase in magnitude and number.
   - The matrix no longer resembles a clean diagonal, even qualitatively.

   This is consistent with the idea that noise creates apparent couplings between coordinates, which DMD then fits with nonzero off-diagonal entries.

Dependence on dimension

As the dimension n increases, these noise-induced effects amplify:

- For fixed trajectory length, higher n means fewer effective samples per parameter.
- The snapshot matrices X and Y become more ill-conditioned, making the DMD operator more sensitive to noise.

In the plots, this shows up as:

- **For moderate noise levels**:
  The eigenvalue cloud is already noticeably spread inside the unit circle in higher-dimensional rows, while it remains much tighter in lower-dimensional ones.
- **For larger noise levels**:
  The spectrum often resembles a noisy disk or ring of eigenvalues, with many modes significantly damped and some modes still near the unit circle. The modulus plots flatten into a broad band rather than a sharp curve near 1.

The heatmaps similarly transition from "almost diagonal" patterns for small n to fully dense matrices for large n, indicating that DMD is using many spurious couplings to reconcile a high-dimensional trajectory with relatively little clean information.

Comparison with sparse regression methods

Contrasting the DMD experiments with the earlier LASSO and sequential-thresholding experiments highlights several important differences:

- **No explicit structural prior**
  DMD does not enforce sparsity or any particular pattern on A_{DMD}. Even when the true matrix is diagonal and sparse, the learned operator is generally dense. Spectral distortions arise primarily from noise and finite-sample effects, rather than from a bias toward sparsity.

- **Type of bias under noise**
  LASSO tends to bias eigenvalues inward due to ℓ_1 shrinkage, and sequential thresholding can produce abrupt failures when the true support is corrupted by noise. DMD, on the other hand:

  - Usually produces eigenvalues with moduli near (but often slightly below) 1 in mildly noisy settings,
  - Develops a ring- or disk-like distribution of eigenvalues for higher noise and larger n, without any "hard" sparsity pattern.
- **Qualitative stability conclusions**
  In all methods, moderate noise can change the inferred stability picture:

  - LASSO often suggests that the system is more stable (more strongly contracting) than it really is.
  - Sequential thresholding can either preserve marginal stability when it succeeds or incorrectly introduce zero/unstable modes when it fails.
  - DMD typically infers a mixture of slightly damped oscillatory modes and spurious decay modes, especially in high dimension.

Summary of DMD experiments

Overall, the DMD experiments show that:

1. **In low-noise, moderate-dimensional regimes**, DMD can accurately recover the spectrum of simple linear systems, with eigenvalues tightly clustered near the true ones and only mild structural distortions.

2. **As noise and dimension increase**, DMD's spectrum becomes increasingly smeared:

   - Eigenvalues drift inside the unit circle, indicating artificial damping,
   - Complex eigenvalue pairs appear even when the true spectrum is real,
   - The learned operator becomes dense and loses any resemblance to the original structural pattern.

These observations complement the findings from the sparse regression experiments: even when we use a method designed specifically to approximate the evolution operator from snapshot data, Gaussian noise and limited data can substantially distort the learned spectrum. Any stability or time-

scale conclusions drawn from DMD eigenvalues must therefore be interpreted in light of both the noise level and the dimensionality of the problem.

Limitations and directions for further work

While the experiments in this project already highlight several important phenomena, there are many ways in which the setup could be refined and extended. Here we outline some possible improvements and open questions.

More realistic dynamical systems and data

Most experiments focused on relatively simple ground-truth matrices (diagonal ± 1 systems, orthogonal systems from matrix exponentials). These are useful for isolating spectral effects, but real applications often involve more complex structure.

Possible extensions:

- **Non-diagonal but structured systems:**
  Use banded, block-structured, or low-rank-plus-sparse matrices A to model spatially local couplings or multi-scale dynamics. This would test whether the observed phenomena persist when the sparsity pattern is more realistic.
- **Nonlinear systems with linear surrogates:**
  Generate data from nonlinear dynamics and fit linear models (or linearizations). Compare how noise and regularization affect the *approximate* Koopman spectrum, not just the exact linear spectrum.
- **Multiple trajectories and initial conditions:**
  The current setup largely considers a single trajectory per model. Using many trajectories with different initial conditions could improve conditioning and may change how noise impacts the learned spectrum.

Open question: *How do the spectral distortions scale when we move from simple diagonal/orthogonal systems to more realistic, structured or nonlinear dynamics?*

Regularization choices and parameter selection

In the sparse experiments, the regularization strength λ (for LASSO) and the threshold τ (for sequential thresholding) are chosen arbitrarily. This leaves several open directions:

- **Data-driven tuning of λ and τ:**
  Explore cross-validation, information criteria, or stability-based criteria (e.g., choose parameters that yield spectra closest to the unit circle when marginal stability is expected).
- **Alternative penalties:**
  Compare LASSO with elastic net, group LASSO, or sparse-plus-low-rank formulations that may better match the true structure of A.

- **Adaptive thresholding schemes:**
  Use thresholds that depend on estimated noise level, column norms, or empirical distributions of coefficients, rather than a fixed global τ.

Open question: *Is there a principled way to choose λ or τ that controls the spectral error $\max_i |\hat{\lambda}_i - \lambda_i|$ in a predictable way?*

Theoretical analysis of spectral bias and variance

The project is entirely empirical. A natural next step is to connect the observed behaviors to formal perturbation results.

Possible directions:

- **Perturbation bounds with regularization:**
  Classical eigenvalue perturbation theory gives bounds for $\|A - \tilde{A}\|$ translating into eigenvalue shifts. Here, however, $\tilde{A}$ is the *output of a regularized regression*, not a simple noisy version of A. Deriving bounds of the form

  $$|\hat{\lambda}_i - \lambda_i| \leq f(\text{noise level}, \lambda, n, T)$$

  where T is trajectory length, would clarify when regularization helps or harms.

- **Bias–variance decomposition for eigenvalues:**
  Separate systematic bias due to regularization (e.g., shrinkage from LASSO) from variance due to finite-sample noise, and quantify how each term depends on problem parameters.

- **Role of eigenvalue gaps:**
  Perturbation theory suggests that closely spaced eigenvalues are more sensitive to noise. It would be useful to systematically vary eigenvalue spacing and study how this affects spectral recovery under LASSO, thresholding, and DMD.

Open question: *Can we characterize regimes where regularization actually improves eigenvalue estimation (by denoising) versus regimes where it primarily introduces bias?*

Refinements of DMD and low-rank methods

The DMD experiments consider a fairly standard SVD-based algorithm. There are many variants designed to handle noise more robustly or to exploit structure:

- **Alternative DMD variants:**
  DMD methods that explicitly account for noise in both X and Y or physics informed DMD that exploit the (expected) structure of A.

- **Low-rank-plus-sparse decompositions:**
  Instead of choosing purely sparse (LASSO/thresholding) or purely low-rank (DMD-style truncation) models, explore decompositions of the form

  $$A \approx L + S,$$

where L is low-rank and S is sparse. This might better capture systems with both coherent global modes and localized interactions.

Open question: *How do different DMD variants and rank selection rules compare in terms of spectral accuracy under Gaussian noise?*

Noise models and robustness

All experiments use additive Gaussian noise, but in many applications:

- Noise can be non-Gaussian (e.g., heavy-tailed, with outliers),
- Noise can be temporally or spatially correlated,
- There may be both **process noise** (in the dynamics) and **measurement noise** (in the observations).

Possible extensions:

- **Non-Gaussian or correlated noise:**
  Replace i.i.d. Gaussian noise with correlated noise or heavy-tailed distributions to see whether the same spectral phenomena hold.

- **Process vs measurement noise:**
  Distinguish between $x_{t+1} = Ax_t + \eta_t$ (process noise) and $x_t^{\text{obs}} = x_t + \xi_t$ (measurement noise). These two cases can affect regression and DMD in different ways.

- **Robust regression methods:**
  Investigate robust estimators (e.g., Huber loss, ℓ_1 loss for residuals) and robust DMD variants to see if they yield more stable spectra under outliers.

Open question: *How sensitive are the spectral conclusions to the assumption of Gaussian noise, and which methods remain reliable under more realistic noise structures?*

Impact on downstream tasks and interpretation

Finally, the experiments focus primarily on the **spectra** of learned operators. In applications, these operators are used for:

- Long-term prediction,
- Stability classification (e.g., “stable vs unstable modes”),
- Mode selection and model reduction.

Possible extensions:

- **Prediction error vs spectral error:**
  Relate errors in eigenvalues to errors in forecasts $A^t x_0$, and determine whether some spectral errors are “harmless” for prediction.

- **Stability classification accuracy:**
  Quantify how often methods correctly identify whether $|\lambda_i| < 1$, $|\lambda_i| = 1$, or $|\lambda_i| > 1$, and how this depends on noise and regularization.

- **Eigenvectors and invariant subspaces:**
  Extend the analysis to eigenvectors and invariant subspaces. Spectral errors might be small while eigenvectors are misaligned, or vice versa.

Open question: *Which aspects of the spectrum (exact eigenvalues, moduli, dominant eigenspaces) matter most for practical tasks, and how does noise + regularization affect each?*

Overall, this project provides a first, controlled look at how Gaussian noise interacts with structural priors (sparse vs low-rank) to distort the spectrum of learned operators. Extending the experiments along the axes above, more realistic systems, better-tuned regularization, theoretical analysis, alternative algorithms, richer noise models, and task-level metrics, would lead to a deeper and more comprehensive understanding of when we can trust spectral information extracted from noisy data-driven models.