LQG Control Demo

Aditya Sengupta

I wasn't totally sure of the syntax and logic flow for the general LQG control problem, so this notebook is meant to help me clarify and present how it works with a toy problem!

There's a couple unintuitive results in here that might indicate unfound bugs, so consider this a draft that I might revise later; for now it's done what I need it to.

This notebook was made with Pluto.jl.

```
1000
```

```
begin
    using LinearAlgebra   , Distributions   , Plots   , Infinity   , Random
    using ControlSystems   : dare
    rms = data -> round(sqrt(sum(data .^ 2)), digits=3)
    Random.seed!(4)
    nsteps = 1000
end
```

Of the above imports, the only one that might not be standard is the `dare` function from `ControlSystems.jl`. This solves a general **d**iscrete **a**lgebraic **R**iccati **e**quation, which will give us our optimal gain values. This can be done in Python too, with `scipy.linalg.solve_discrete_are` and essentially the same syntax.

My main reference is **these** lecture slides from Stanford's EE 363.

The first step is to make a control problem, which I do by randomly choosing matrices B (input-to-state-evolution) and C (state-to-measurement), as well as slightly more carefully choosing a matrix A (state-to-state-evolution). A has to be stable so that time-evolution without anything else doesn't just blow up, so I pick some eigenvalues between -1 and +1 and apply a random change-of-basis to it to ensure none of the eigenvalues are outside the unit circle.

I also fix the same choices of noise and control cost matrices as in the slides.

Reference for what the matrices are:

- A, state-to-state-evolution
- B, input-to-state-evolution
- C, state-to-measurement
- W, process noise covariance
- V, measurement noise covariance
- Q, state penalty weighting (the cost of having a state x is $x'Qx$)
- R, input penalty weighting (the cost of having an input u is $u'Ru$)

The notes also specify the initial distribution comes from a $\mathcal{N}(0, X)$ distribution, but I'm just saying the initial state is a known random vector and not tuning X.

```
UniformScaling{Bool}
true*I
```

```
begin
    # set up the problem
    s = 5; # state size
    p = 3; # measurement size
    m = 2; # input size
    N = ∞; # control horizon; not working with this as of yet

    lambdas = rand(Uniform(-1, 1), s);
    lambdas = lambdas ./ ((1 + 1e-6) * maximum(abs.(lambdas)));
    M = rand(s, s);
    A = inv(M) * diagm(lambdas) * M;
    B = rand(s, m);
    C = rand(p, s);
    Q = I;
    R = I;
    W = 0.5I;
    V = I;
end
```

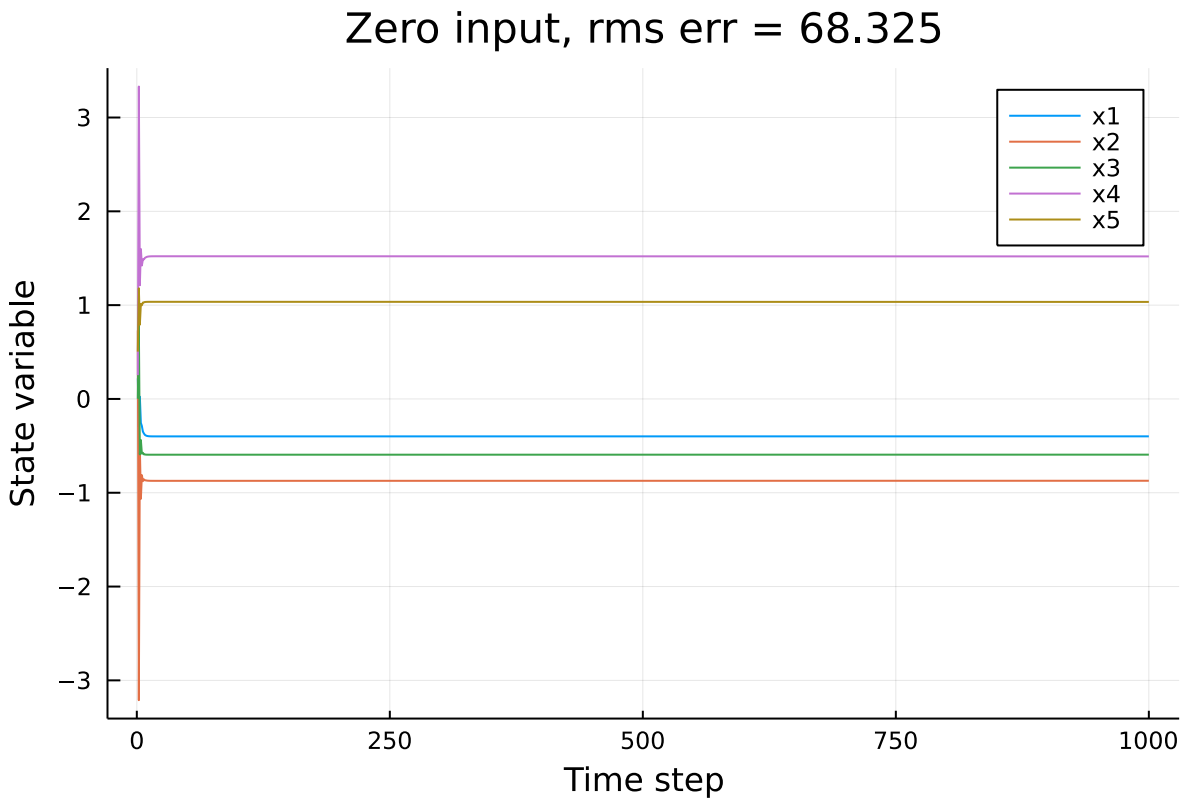
This cell solves algebraic Ricatti equations for the Kalman gain L and the regulator gain K. Essentially what this does is fix the observed state covariance (respectively, the controlled state covariance) so that it stays constant through a predict-update (respectively, a...state-update and input-update?) cycle. This is one of the main advantages of the Kalman-LQG method: these gains can be computed up front, and then controlling the system is just a matter of adding in a couple of matrix multiplications.

```
2×5 Matrix{Float64}:
  0.117277  -0.146625   -0.203931  -0.0299814   0.0978185
 -0.19511    0.0562195   0.218583  -0.0944854  -0.265731
```

```
begin
    Pk = dare(A', C', W, V) # P_Kalman
    L = Pk * C' * inv(C * Pk * C' + R) # Kalman gain
    Pr = dare(A, B, Q, R) # P_regulator
    K = -inv(R + B' * Pr * B) * B' * Pr * A # regulator gain
end
```

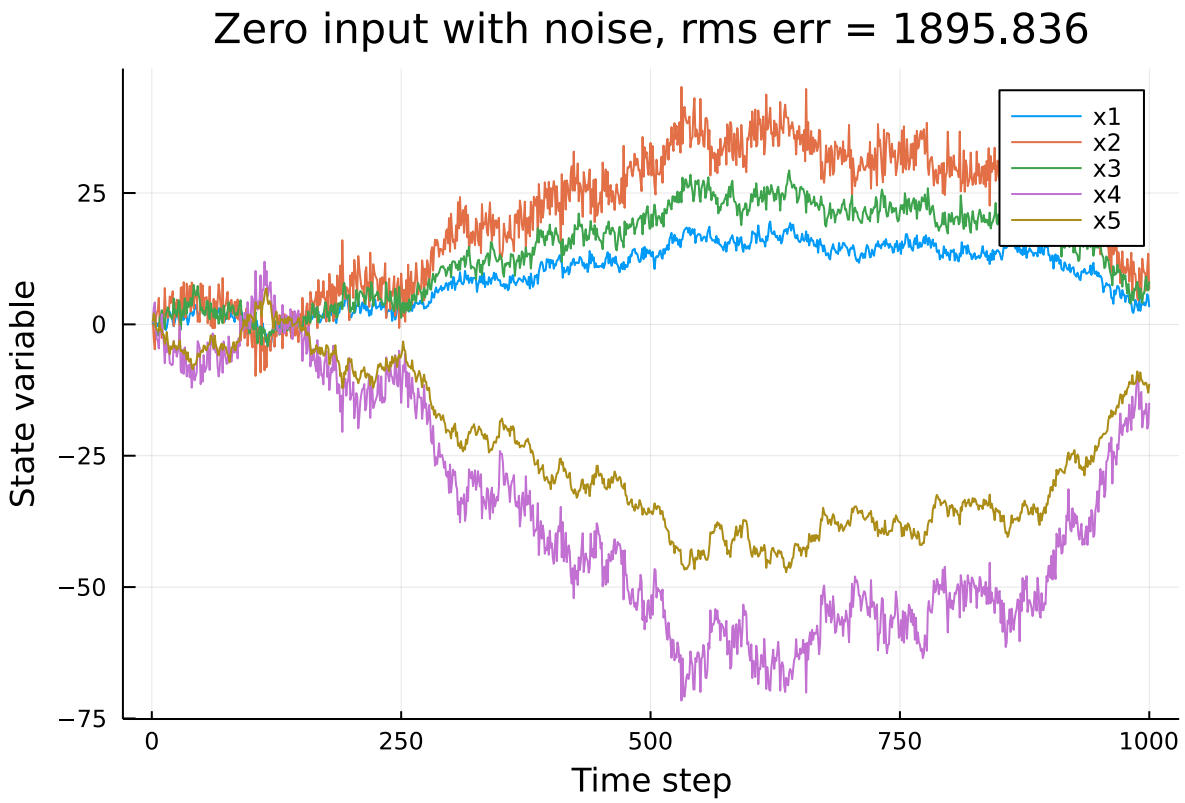
```
x = rand(s);
```

First, we just iterate $x_{t+1} = Ax_t$. Since we chose A such that all its eigenvalues were within the unit circle, this should lead to all state variables reaching a steady-state value as the transients die, and this does in fact happen...if the dominant eigenvalue is 1. Sometimes your results will be dominated by an eigenvector with eigenvalue close to -1, which leads to oscillations on short timescales. It'll still settle over a longer timescale, but it'll look spiky in the meantime. Just increase the number of iterations to see it settle more in that case!



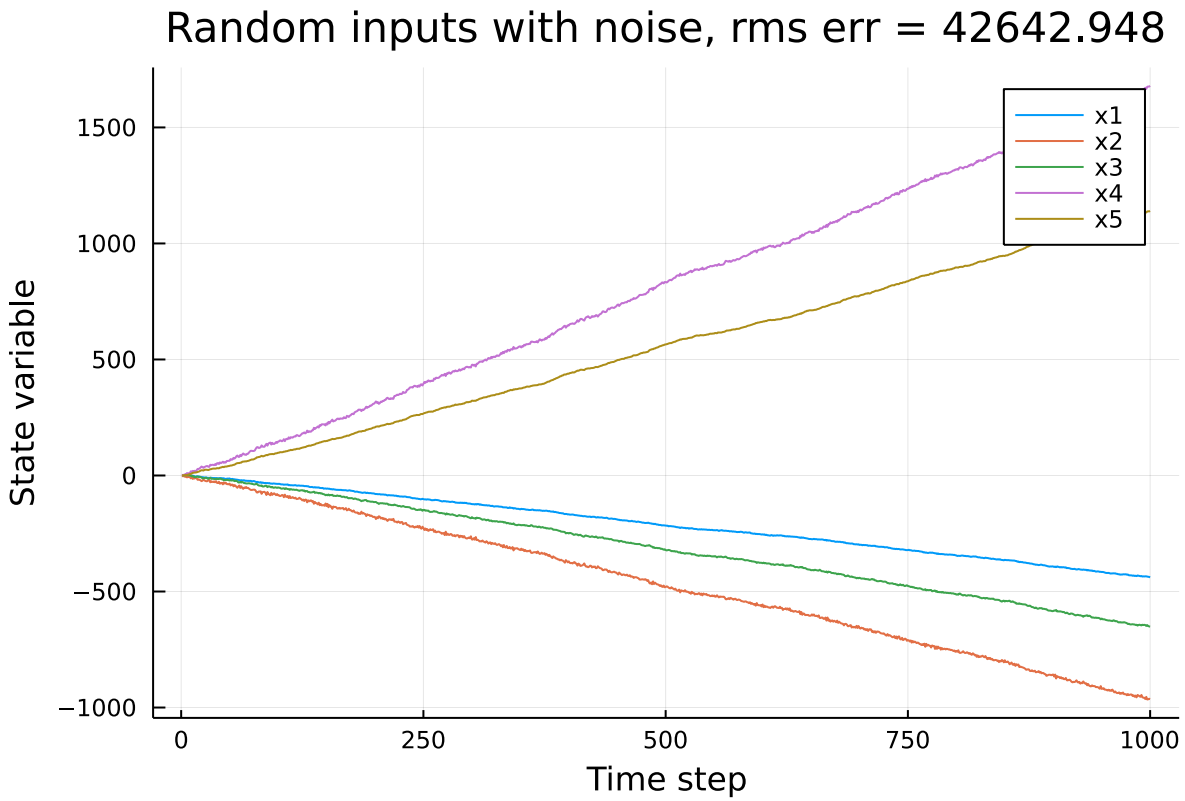

```
# uncontrolled path, zero inputs
begin
    local states = zeros(Float64, nsteps, s)
    states[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        states[i,:] = A * states[i-1,:]
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step", ylabel="State variable", title="Zero input, rms err = $(rms(states))")
end
```

Adding noise into this, we see why control is necessary; while paths seem relatively stable, they're veering far off and not staying close to 0 at all.



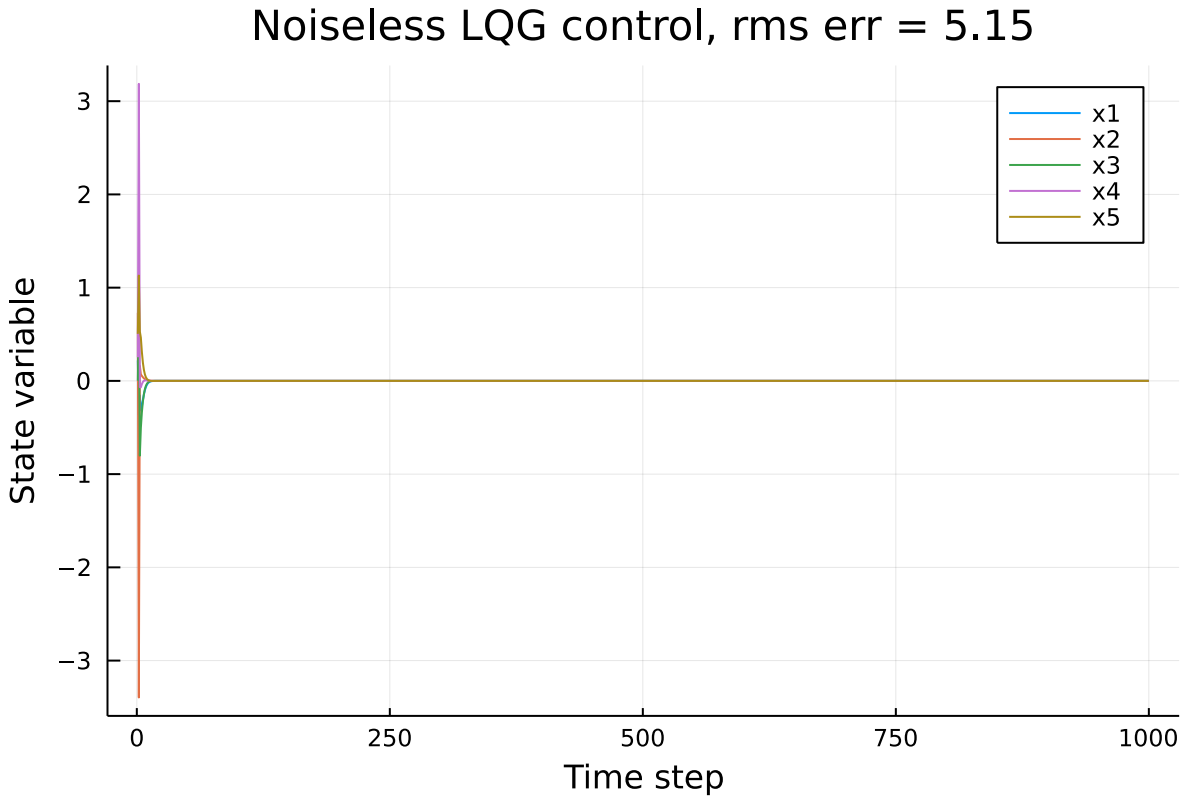

```
# uncontrolled path, zero inputs
begin
    local states = zeros(Float64, nsteps, s)
    local process_dist = MvNormal(Matrix(W, s, s))
    states[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        states[i,:] = A * states[i-1,:] + rand(process_dist)
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step", ylabel="State variable", title="Zero input with noise, rms err = $(rms(states))")
end
```

Next, we drive the system with random inputs, and this leads to it going off even further, because I didn't control the action of B the same way I did A. (I'm not really sure how to ensure the stability of a dynamic system defined by a non-square matrix, as there's no concept of an eigenvalue - this would be intellectually interesting to look at but doesn't really matter.)



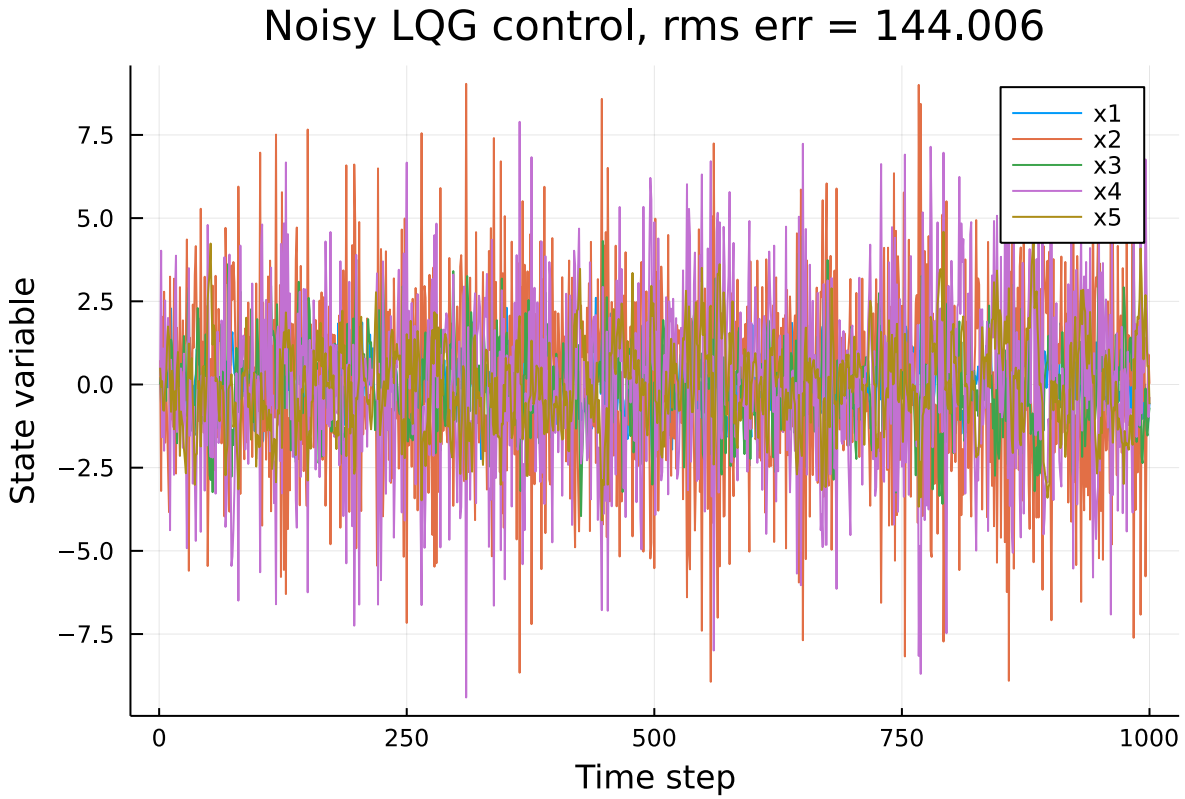

```
# uncontrolled path, random inputs
begin
    local states = zeros(Float64, nsteps, s)
    local process_dist = MvNormal(Matrix(W, s, s))
    states[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        states[i,:] = A * states[i-1,:] + B * rand(m) + rand(process_dist)
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step", ylabel="State variable", title="Random inputs with noise, rms err = $(rms(states))")
end
```

Now, we bring in the controller! We first run it without any noise. We see here that all the state variables go to 0 very quickly, as you would expect from no process or measurement noise and full-state observations.



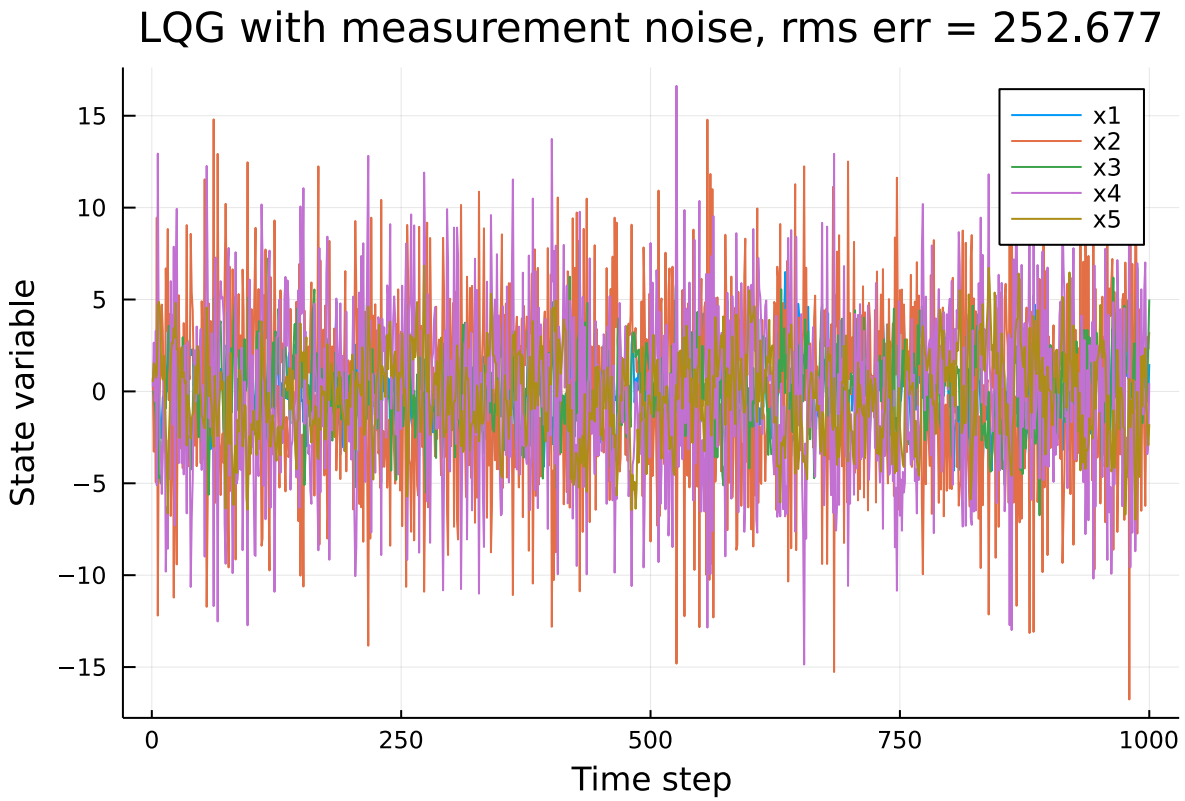

```
# full state observation, LQG control
begin
    local states = zeros(Float64, nsteps, s)
    states[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        u = K * x
        states[i,:] = A * x + B * u
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step", ylabel="State variable", title="Noiseless LQG control, rms err = $(rms(states))")
end
```

Now we add noise back in. Compare this to the case of no control - we're staying much closer to the origin now! It's still not as good as it could be, because we've also told the controller it has to compensate for measurement noise but haven't put any in, causing it to overcorrect. We'll see this solve itself in a bit, but you could rerun the AREs above with $V = 0$ to see how that affects it.




```
# full state observation plus noise, LQG control
begin
    local states = zeros(Float64, nsteps, s)
    local process_dist = MvNormal(Matrix(W, s, s))
    states[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        u = K * x
        states[i,:] = A * x + B * u + rand(process_dist)
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step", ylabel="State variable", title="Noisy LQG control, rms err = $(rms(states))")
end
```

Now, we break the final simplifying assumption, that we have noiseless full-state observations. First, let's just add in measurement noise and do nothing to correct it. With full-state observations, all this does is changes the system's process noise without telling the LQG controller that you're changing the process noise (thanks to Gaussian linearity, you're just dealing with $\mathcal{N}(0, W) + \mathcal{N}(0, V)$ instead of $\mathcal{N}(0, W)$) so in principle, this process could be optimally controlled just by re-solving the controller ARE. However, that assumption won't hold when we go to partial-state control.



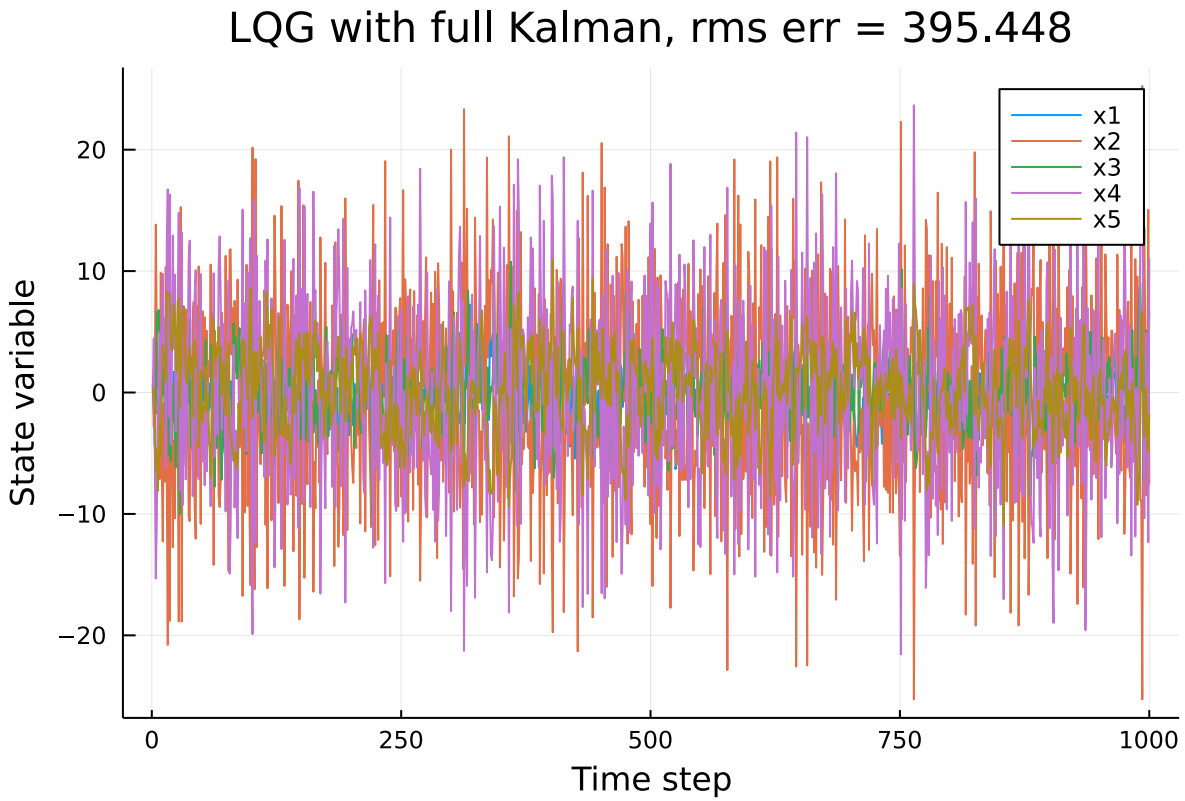

```
# full state observation plus noise and Kalman correction, LQG control
begin
    local states = zeros(Float64, nsteps, s)
    local process_dist = MvNormal(Matrix(W, s, s))
    local measure_dist = MvNormal(Matrix(V, s, s))
    states[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        u = K * x
        y = A * x + B * u + rand(measure_dist)
        states[i,:] = y + rand(process_dist)
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step",
ylabel="State variable", title="LQG with measurement noise, rms err = $(rms(states))")
end
```

Correcting *this* measurement noise is actually a different algebraic Riccati equation than the one we solved above - let's quickly compute a Kalman gain that'll help us solve this problem!

```
5×5 Matrix{Float64}:
 0.383131   0.0329179  0.0173027  0.0281387  0.0645356
 0.0329179  0.589193   0.158541   0.173919   0.156087
 0.0173027  0.158541   0.489378   0.0826488  0.0465391
 0.0281387  0.173919   0.0826488  0.465755   0.128532
 0.0645356  0.156087   0.0465391  0.128532   0.515271
```

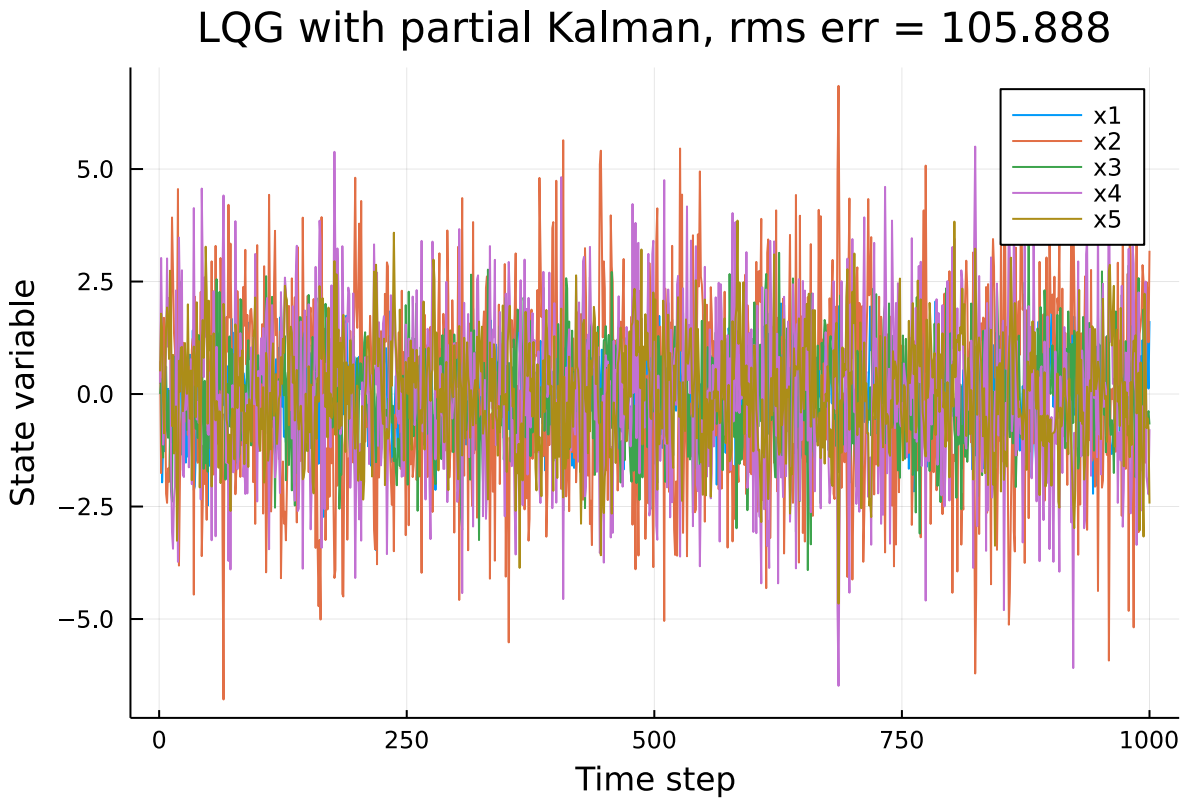
```
begin
    Pk_full = dare(A, I, W, V)
    L_full = Pk_full * inv(Pk_full + R) # Kalman gain with C = I
end
```

From here on out, we store two arrays of the states: the true values in `states`, so we can keep track of how well we're controlling them, and our best-estimate values in `states_hat`, to use as control input.




```
# full state observation plus noise and Kalman correction, LQG control
begin
    local states = zeros(Float64, nsteps, s)
    local states_hat = zeros(Float64, nsteps, s)
    local process_dist = MvNormal(Matrix(W, s, s))
    local measure_dist = MvNormal(Matrix(V, s, s))
    local C = I
    states[1,:] = x
    states_hat[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        x_hat = states_hat[i-1,:]
        u = K * x_hat
        y = C * (A * x_hat + B * u) + rand(measure_dist)
        innovation = y - C * (A * x + B * u)
        # you could just say innovation = rand(measure_dist) here, but y is separate for clarity
        states_hat[i,:] = y + L_full * innovation
        states[i,:] = states_hat[i,:] + rand(process_dist)
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step", ylabel="State variable", title="LQG with full Kalman, rms err = $(rms(states))")
end
```

Having built all of this up, let's go to the full machinery with partial-state observations!




```
# full state observation plus noise and Kalman correction, LQG control
begin
    local states = zeros(Float64, nsteps, s)
    local states_hat = zeros(Float64, nsteps, s)
    local process_dist = MvNormal(Matrix(W, s, s))
    local measure_dist = MvNormal(Matrix(V, p, p))
    states[1,:] = x
    states_hat[1,:] = x
    for i in 2:nsteps
        x = states[i-1,:]
        x_hat = states_hat[i-1,:]
        u = K * x_hat
        y = C * (A * x_hat + B * u) + rand(measure_dist)
        innovation = y - C * (A * x + B * u)
        states_hat[i,:] = A * x_hat + B * u + L * innovation
        states[i,:] = states_hat[i,:] + rand(process_dist)
    end
    plot(1:nsteps, states, labels=["x1" "x2" "x3" "x4" "x5"], xlabel="Time step",
ylabel="State variable", title="LQG with partial Kalman, rms err = $(rms(states))")
end
```