Arrays, Tuples, Ranges, and Other Fundamental Types

Jesse Perla, Thomas J. Sargent and John Stachurski

December 4, 2020

1 Contents

> "Let's be clear: the work of science has nothing whatever to do with consensus. Consensus is the business of politics. Science, on the contrary, requires only one investigator who happens to be right, which means that he or she has results that are verifiable by reference to the real world. In science consensus is irrelevant. What is relevant is reproducible results." – Michael Crichton

2 Overview

In Julia, arrays and tuples are the most important data type for working with numerical data.

In this lecture we give more details on

- creating and manipulating Julia arrays
- fundamental array processing operations
- basic matrix algebra
- tuples and named tuples
- ranges
- nothing, missing, and unions

2.1 Setup

```
In [1]: using InstantiateFromURL
        # optionally add arguments to force installation: instantiate = true,
        precompile = true
        github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```
In [2]: using LinearAlgebra, Statistics
```

3 Array Basics

(See multi-dimensional arrays documentation)

Since it is one of the most important types, we will start with arrays.

Later, we will see how arrays (and all other types in Julia) are handled in a generic and extensible way.

3.1 Shape and Dimension

We've already seen some Julia arrays in action

```
In [3]: a = [10, 20, 30]
```

```
Out[3]: 3-element Array{Int64,1}:
         10
         20
         30
```

```
In [4]: a = [1.0, 2.0, 3.0]
```

```
Out[4]: 3-element Array{Float64,1}:
         1.0
         2.0
         3.0
```

The output tells us that the arrays are of types `Array{Int64,1}` and `Array{Float64,1}` respectively.

Here `Int64` and `Float64` are types for the elements inferred by the compiler.

We'll talk more about types later.

The `1` in `Array{Int64,1}` and `Array{Any,1}` indicates that the array is one dimensional (i.e., a `Vector`).

This is the default for many Julia functions that create arrays

```
In [5]: typeof(randn(100))
```

```
Out[5]: Array{Float64,1}
```

In Julia, one dimensional vectors are best interpreted as column vectors, which we will see when we take transposes.

We can check the dimensions of `a` using `size()` and `ndims()` functions

```
In [6]: ndims(a)
```

```
Out[6]: 1
```

```
In [7]: size(a)
```

```
Out[7]: (3,)
```

The syntax `(3,)` displays a tuple containing one element – the size along the one dimension that exists.

3.1.1 Array vs Vector vs Matrix

In Julia, `Vector` and `Matrix` are just aliases for one- and two-dimensional arrays respectively

```
In [8]: Array{Int64, 1} == Vector{Int64}
        Array{Int64, 2} == Matrix{Int64}
```

```
Out[8]: true
```

Vector construction with `,` is then interpreted as a column vector.

To see this, we can create a column vector and row vector more directly

```
In [9]: [1, 2, 3] == [1; 2; 3]  # both column vectors
```

```
Out[9]: true
```

```
In [10]: [1 2 3]  # a row vector is 2-dimensional
```

```
Out[10]: 1×3 Array{Int64,2}:
          1  2  3
```

As we've seen, in Julia we have both

- one-dimensional arrays (i.e., flat arrays)
- arrays of size `(1, n)` or `(n, 1)` that represent row and column vectors respectively

Why do we need both?

On one hand, dimension matters for matrix algebra.

- Multiplying by a row vector is different to multiplying by a column vector.

On the other, we use arrays in many settings that don't involve matrix algebra.

In such cases, we don't care about the distinction between row and column vectors.

This is why many Julia functions return flat arrays by default.

3.2 Creating Arrays

3.2.1 Functions that Create Arrays

We've already seen some functions for creating a vector filled with `0.0`

```
In [11]: zeros(3)
```

```
Out[11]: 3-element Array{Float64,1}:
          0.0
          0.0
          0.0
```

This generalizes to matrices and higher dimensional arrays

```
In [12]: zeros(2, 2)
```

```
Out[12]: 2×2 Array{Float64,2}:
          0.0  0.0
          0.0  0.0
```

To return an array filled with a single value, use `fill`

```
In [13]: fill(5.0, 2, 2)
```

```
Out[13]: 2×2 Array{Float64,2}:
          5.0  5.0
          5.0  5.0
```

Finally, you can create an empty array using the `Array()` constructor

```
In [14]: x = Array{Float64}(undef, 2, 2)
```

```
Out[14]: 2×2 Array{Float64,2}:
          6.91839e-310  6.91838e-310
          6.91839e-310  0.0
```

The printed values you see here are just garbage values.

(the existing contents of the allocated memory slots being interpreted as 64 bit floats)

If you need more control over the types, fill with a non-floating point

```
In [15]: fill(0, 2, 2)  # fills with 0, not 0.0
```

```
Out[15]: 2×2 Array{Int64,2}:
          0  0
          0  0
```

Or fill with a boolean type

```
In [16]: fill(false, 2, 2)  # produces a boolean matrix
```

```
Out[16]: 2×2 Array{Bool,2}:
          0  0
          0  0
```

3.2.2 Creating Arrays from Existing Arrays

For the most part, we will avoid directly specifying the types of arrays, and let the compiler deduce the optimal types on its own.

The reasons for this, discussed in more detail in this lecture, are to ensure both clarity and generality.

One place this can be inconvenient is when we need to create an array based on an existing array.

First, note that assignment in Julia binds a name to a value, but does not make a copy of that type

```
In [17]: x = [1, 2, 3]
         y = x
         y[1] = 2
         x
```

```
Out[17]: 3-element Array{Int64,1}:
          2
          2
          3
```

In the above, `y = x` simply creates a new named binding called `y` which refers to whatever `x` currently binds to.

To copy the data, you need to be more explicit

```
In [18]: x = [1, 2, 3]
         y = copy(x)
         y[1] = 2
         x
```

```
Out[18]: 3-element Array{Int64,1}:
          1
          2
          3
```

However, rather than making a copy of `x`, you may want to just have a similarly sized array

```
In [19]: x = [1, 2, 3]
         y = similar(x)
         y
```

```
Out[19]: 3-element Array{Int64,1}:
          1
          1
          0
```

We can also use `similar` to pre-allocate a vector with a different size, but the same shape

```
In [20]: x = [1, 2, 3]
         y = similar(x, 4)  # make a vector of length 4
```

```
Out[20]: 4-element Array{Int64,1}:
          140029121887504
          140029758928704
          140029448573616
          140029694705664
```

Which generalizes to higher dimensions

```
In [21]: x = [1, 2, 3]
         y = similar(x, 2, 2)  # make a 2x2 matrix
```

```
Out[21]: 2×2 Array{Int64,2}:
          140029122207120  140029122207184
          140029122207152  140029493746368
```

3.2.3 Manual Array Definitions

As we've seen, you can create one dimensional arrays from manually specified data like so

```
In [22]: a = [10, 20, 30, 40]
```

```
Out[22]: 4-element Array{Int64,1}:
          10
          20
          30
          40
```

In two dimensions we can proceed as follows

```
In [23]: a = [10 20 30 40]  # two dimensional, shape is 1 x n
```

```
Out[23]: 1×4 Array{Int64,2}:
          10  20  30  40
```

```
In [24]: ndims(a)
```

```
Out[24]: 2
```

```
In [25]: a = [10 20; 30 40]  # 2 x 2
```

```
Out[25]: 2×2 Array{Int64,2}:
          10  20
          30  40
```

You might then assume that `a = [10; 20; 30; 40]` creates a two dimensional column vector but this isn't the case.

```
In [26]: a = [10; 20; 30; 40]
```

```
Out[26]: 4-element Array{Int64,1}:
          10
          20
          30
          40
```

```
In [27]: ndims(a)
```

```
Out[27]: 1
```

Instead transpose the matrix (or adjoint if complex)

```
In [28]: a = [10 20 30 40]'
```

```
Out[28]: 4×1 Adjoint{Int64,Array{Int64,2}}:
          10
          20
          30
          40
```

```
In [29]: ndims(a)
```

```
Out[29]: 2
```

3.3 Array Indexing

We've already seen the basics of array indexing

```
In [30]: a = [10 20 30 40]
         a[end-1]
```

```
Out[30]: 30
```

```
In [31]: a[1:3]
```

```
Out[31]: 3-element Array{Int64,1}:
          10
          20
          30
```

For 2D arrays the index syntax is straightforward

```
In [32]: a = randn(2, 2)
         a[1, 1]
```

```
Out[32]: 0.15765041580654648
```

```
In [33]: a[1, :]  # first row
```

```
Out[33]: 2-element Array{Float64,1}:
          0.15765041580654648
          0.2652121417935994

In [34]: a[:, 1]  # first column

Out[34]: 2-element Array{Float64,1}:
          0.15765041580654648
          0.134644163495092
```

Booleans can be used to extract elements

```
In [35]: a = randn(2, 2)

Out[35]: 2×2 Array{Float64,2}:
          0.616704   0.374922
          1.33094   -0.670024

In [36]: b = [true false; false true]

Out[36]: 2×2 Array{Bool,2}:
          1  0
          0  1

In [37]: a[b]

Out[37]: 2-element Array{Float64,1}:
           0.6167040565642389
          -0.6700242069255393
```

This is useful for conditional extraction, as we'll see below.

An aside: some or all elements of an array can be set equal to one number using slice notation.

```
In [38]: a = zeros(4)

Out[38]: 4-element Array{Float64,1}:
          0.0
          0.0
          0.0
          0.0

In [39]: a[2:end] .= 42

Out[39]: 3-element view(::Array{Float64,1}, 2:4) with eltype Float64:
          42.0
          42.0
          42.0

In [40]: a

Out[40]: 4-element Array{Float64,1}:
           0.0
          42.0
          42.0
          42.0
```

3.4 Views and Slices

Using the `:` notation provides a slice of an array, copying the sub-array to a new array with a similar type.

```
In [41]: a = [1 2; 3 4]
         b = a[:, 2]
         @show b
         a[:, 2] = [4, 5] # modify a
         @show a
         @show b;
```

```
b = [2, 4]
a = [1 4; 3 5]
b = [2, 4]
```

A **view** on the other hand does not copy the value

```
In [42]: a = [1 2; 3 4]
         @views b = a[:, 2]
         @show b
         a[:, 2] = [4, 5]
         @show a
         @show b;
```

```
b = [2, 4]
a = [1 4; 3 5]
b = [4, 5]
```

Note that the only difference is the `@views` macro, which will replace any slices with views in the expression.

An alternative is to call the `view` function directly – though it is generally discouraged since it is a step away from the math.

```
In [43]: @views b = a[:, 2]
         view(a, :, 2) == b
```

```
Out[43]: true
```

As with most programming in Julia, it is best to avoid prematurely assuming that `@views` will have a significant impact on performance, and stress code clarity above all else.

Another important lesson about `@views` is that they **are not** normal, dense arrays.

```
In [44]: a = [1 2; 3 4]
         b_slice = a[:, 2]
         @show typeof(b_slice)
         @show typeof(a)
         @views b = a[:, 2]
         @show typeof(b);
```

```
    typeof(b_slice) = Array{Int64,1}
typeof(a) = Array{Int64,2}
typeof(b) =
SubArray{Int64,1,Array{Int64,2},Tuple{Base.Slice{Base.OneTo{Int64}},Int64},true}
```

The type of `b` is a good example of how types are not as they may seem.

Similarly

```
In [45]: a = [1 2; 3 4]
         b = a'   # transpose
         typeof(b)
```

```
Out[45]: Adjoint{Int64,Array{Int64,2}}
```

To copy into a dense array

```
In [46]: a = [1 2; 3 4]
         b = a' # transpose
         c = Matrix(b)  # convert to matrix
         d = collect(b) # also `collect` works on any iterable
         c == d
```

```
Out[46]: true
```

3.5 Special Matrices

As we saw with `transpose`, sometimes types that look like matrices are not stored as a dense array.

As an example, consider creating a diagonal matrix

```
In [47]: d = [1.0, 2.0]
         a = Diagonal(d)
```

```
Out[47]: 2×2 Diagonal{Float64,Array{Float64,1}}:
          1.0   □
           □   2.0
```

As you can see, the type is `2×2 Diagonal{Float64,Array{Float64,1}}`, which is not a 2-dimensional array.

The reasons for this are both efficiency in storage, as well as efficiency in arithmetic and matrix operations.

In every important sense, matrix types such as `Diagonal` are just as much a "matrix" as the dense matrices we have using (see the introduction to types lecture for more)

```
In [48]: @show 2a
         b = rand(2,2)
         @show b * a;
```

```
    2a = [2.0 0.0; 0.0 4.0]
b * a = [0.07902796541012158 0.36525218581573515; 0.5277291038116809
1.824094575599461]
```

Another example is in the construction of an identity matrix, where a naive implementation is

```
In [49]: b = [1.0 2.0; 3.0 4.0]
         b - Diagonal([1.0, 1.0])  # poor style, inefficient code
```

```
Out[49]: 2×2 Array{Float64,2}:
          0.0  2.0
          3.0  3.0
```

Whereas you should instead use

```
In [50]: b = [1.0 2.0; 3.0 4.0]
         b - I  # good style, and note the lack of dimensions of I
```

```
Out[50]: 2×2 Array{Float64,2}:
          0.0  2.0
          3.0  3.0
```

While the implementation of `I` is a little abstract to go into at this point, a hint is:

```
In [51]: typeof(I)
```

```
Out[51]: UniformScaling{Bool}
```

This is a `UniformScaling` type rather than an identity matrix, making it much more powerful and general.

3.6 Assignment and Passing Arrays

As discussed above, in Julia, the left hand side of an assignment is a "binding" or a label to a value.

```
In [52]: x = [1 2 3]
         y = x  # name `y` binds to whatever value `x` bound to
```

```
Out[52]: 1×3 Array{Int64,2}:
          1  2  3
```

The consequence of this, is that you can re-bind that name.

```
In [53]: x = [1 2 3]
         y = x        # name `y` binds to whatever `x` bound to
         z = [2 3 4]
         y = z        # only changes name binding, not value!
         @show (x, y, z);
```

```
(x, y, z) = ([1 2 3], [2 3 4], [2 3 4])
```

What this means is that if `a` is an array and we set `b = a` then `a` and `b` point to exactly the same data.

In the above, suppose you had meant to change the value of `x` to the values of `y`, you need to assign the values rather than the name.

```
In [54]: x = [1 2 3]
         y = x       # name `y` binds to whatever `x` bound to
         z = [2 3 4]
         y .= z      # now dispatches the assignment of each element
         @show (x, y, z);
```

```
(x, y, z) = ([2 3 4], [2 3 4], [2 3 4])
```

Alternatively, you could have used `y[:] = z`.

This applies to in-place functions as well.

First, define a simple function for a linear map

```
In [55]: function f(x)
             return [1 2; 3 4] * x  # matrix * column vector
         end

         val = [1, 2]
         f(val)
```

```
Out[55]: 2-element Array{Int64,1}:
           5
          11
```

In general, these "out-of-place" functions are preferred to "in-place" functions, which modify the arguments.

```
In [56]: function f(x)
             return [1 2; 3 4] * x # matrix * column vector
         end

         val = [1, 2]
         y = similar(val)

         function f!(out, x)
             out .= [1 2; 3 4] * x
         end

         f!(y, val)
         y
```

```
Out[56]: 2-element Array{Int64,1}:
           5
          11
```

This demonstrates a key convention in Julia: functions which modify any of the arguments have the name ending with `!` (e.g. `push!`).

We can also see a common mistake, where instead of modifying the arguments, the name binding is swapped

```
In [57]: function f(x)
             return [1 2; 3 4] * x  # matrix * column vector
         end

         val = [1, 2]
         y = similar(val)

         function f!(out, x)
             out = [1 2; 3 4] * x   # MISTAKE! Should be .= or [:]
         end
         f!(y, val)
         y
```

```
Out[57]: 2-element Array{Int64,1}:
                    5674
          140029483809008
```

The frequency of making this mistake is one of the reasons to avoid in-place functions, unless proven to be necessary by benchmarking.

3.7 In-place and Immutable Types

Note that scalars are always immutable, such that

```
In [58]: y = [1 2]
         y .-= 2    # y .= y .- 2, no problem

         x = 5
         # x .-= 2  # Fails!
         x = x - 2  # subtle difference - creates a new value and rebinds the
 ↪variable
```

```
Out[58]: 3
```

In particular, there is no way to pass any immutable into a function and have it modified

```
In [59]: x = 2

         function f(x)
             x = 3     # MISTAKE! does not modify x, creates a new value!
         end

         f(x)          # cannot modify immutables in place
         @show x;
```

```
  x = 2
```

This is also true for other immutable types such as tuples, as well as some vector types

In [60]:
```
using StaticArrays
xdynamic = [1, 2]
xstatic = @SVector [1, 2]  # turns it into a highly optimized static vector

f(x) = 2x
@show f(xdynamic)
@show f(xstatic)

# inplace version
function g(x)
    x .= 2x
    return "Success!"
end
@show xdynamic
@show g(xdynamic)
@show xdynamic;

# g(xstatic) # fails, static vectors are immutable
```

```
   f(xdynamic) = [2, 4]
f(xstatic) = [2, 4]
xdynamic = [1, 2]
g(xdynamic) = "Success!"
xdynamic = [2, 4]
```

4 Operations on Arrays

4.1 Array Methods

Julia provides standard functions for acting on arrays, some of which we've already seen

In [61]:
```
a = [-1, 0, 1]

@show length(a)
@show sum(a)
@show mean(a)
@show std(a)      # standard deviation
@show var(a)      # variance
@show maximum(a)
@show minimum(a)
@show extrema(a)  # (mimimum(a), maximum(a))
```

```
   length(a) = 3
sum(a) = 0
mean(a) = 0.0
std(a) = 1.0
var(a) = 1.0
maximum(a) = 1
minimum(a) = -1
extrema(a) = (-1, 1)
```

```
Out[61]: (-1, 1)
```

To sort an array

```
In [62]: b = sort(a, rev = true)  # returns new array, original not modified

Out[62]: 3-element Array{Int64,1}:
           1
           0
          -1

In [63]: b = sort!(a, rev = true)  # returns *modified original* array

Out[63]: 3-element Array{Int64,1}:
           1
           0
          -1

In [64]: b == a  # tests if have the same values

Out[64]: true

In [65]: b === a  # tests if arrays are identical (i.e share same memory)

Out[65]: true
```

4.2 Matrix Algebra

For two dimensional arrays, * means matrix multiplication

```
In [66]: a = ones(1, 2)

Out[66]: 1×2 Array{Float64,2}:
          1.0  1.0

In [67]: b = ones(2, 2)

Out[67]: 2×2 Array{Float64,2}:
          1.0  1.0
          1.0  1.0

In [68]: a * b

Out[68]: 1×2 Array{Float64,2}:
          2.0  2.0

In [69]: b * a'
```

```
Out[69]: 2×1 Array{Float64,2}:
          2.0
          2.0
```

To solve the linear system $AX = B$ for X use `A \ B`

```
In [70]: A = [1 2; 2 3]
```

```
Out[70]: 2×2 Array{Int64,2}:
          1  2
          2  3
```

```
In [71]: B = ones(2, 2)
```

```
Out[71]: 2×2 Array{Float64,2}:
          1.0  1.0
          1.0  1.0
```

```
In [72]: A \ B
```

```
Out[72]: 2×2 Array{Float64,2}:
          -1.0  -1.0
           1.0   1.0
```

```
In [73]: inv(A) * B
```

```
Out[73]: 2×2 Array{Float64,2}:
          -1.0  -1.0
           1.0   1.0
```

Although the last two operations give the same result, the first one is numerically more stable and should be preferred in most cases.

Multiplying two **one** dimensional vectors gives an error – which is reasonable since the meaning is ambiguous.

More precisely, the error is that there isn't an implementation of `*` for two one dimensional vectors.

The output explains this, and lists some other methods of `*` which Julia thinks are close to what we want.

```
In [74]: ones(2) * ones(2)  # does not conform, expect error
```

```
MethodError: no method matching *(::Array{Float64,1}, ::
↪Array{Float64,1})
  Closest candidates are:
    *(::Any, ::Any, !Matched::Any, !Matched::Any…) at operators.jl:
↪529
```

```
    *(!Matched::Adjoint{#s662,#s661} where #s661<:
↪Union{DenseArray{T,2}, Base.ReinterpretArray{T,2,S,A} where S
↪where A<:Union{SubArray{T,N,A,I,true} where I<:
↪Union{Tuple{Vararg{Real,N} where N},
↪Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray}, Base.
↪ReshapedArray{T,2,A,MI} where MI<:Tuple{Vararg{Base.
↪MultiplicativeInverses.SignedMultiplicativeInverse{Int64},N}
↪where N} where A<:Union{Base.ReinterpretArray{T,N,S,A} where S
↪where A<:Union{SubArray{T,N,A,I,true} where I<:
↪Union{Tuple{Vararg{Real,N} where N},
↪Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray} where N where T,
↪SubArray{T,N,A,I,true} where I<:Union{Tuple{Vararg{Real,N} where
↪N}, Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray}, SubArray{T,2,A,I,L}
↪where L where I<:Tuple{Vararg{Union{Int64, AbstractRange{Int64},
↪Base.AbstractCartesianIndex},N} where N} where A<:Union{Base.
↪ReinterpretArray{T,N,S,A} where S where A<:
↪Union{SubArray{T,N,A,I,true} where I<:Union{Tuple{Vararg{Real,N}
↪where N}, Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where
↪A<:DenseArray where N where T, DenseArray} where N where T, Base.
↪ReshapedArray{T,N,A,MI} where MI<:Tuple{Vararg{Base.
↪MultiplicativeInverses.SignedMultiplicativeInverse{Int64},N}
↪where N} where A<:Union{Base.ReinterpretArray{T,N,S,A} where S
↪where A<:Union{SubArray{T,N,A,I,true} where I<:
↪Union{Tuple{Vararg{Real,N} where N},
↪Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray} where N where T,
↪SubArray{T,N,A,I,true} where I<:Union{Tuple{Vararg{Real,N} where
↪N}, Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray} where N where T,
↪DenseArray}} where #s662, ::Union{DenseArray{S,1}, Base.
↪ReinterpretArray{S,1,S1,A} where S1 where A<:
↪Union{SubArray{T,N,A,I,true} where I<:Union{Tuple{Vararg{Real,N}
↪where N}, Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where
↪A<:DenseArray where N where T, DenseArray}, Base.
↪ReshapedArray{S,1,A,MI} where MI<:Tuple{Vararg{Base.
↪MultiplicativeInverses.SignedMultiplicativeInverse{Int64},N}
↪where N} where A<:Union{Base.ReinterpretArray{T,N,S,A} where S
↪where A<:Union{SubArray{T,N,A,I,true} where I<:
↪Union{Tuple{Vararg{Real,N} where N},
↪Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray} where N where T,
↪SubArray{T,N,A,I,true} where I<:Union{Tuple{Vararg{Real,N} where
↪N}, Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where A<:
↪DenseArray where N where T, DenseArray}, SubArray{S,1,A,I,L}
↪where L where I<:Tuple{Vararg{Union{Int64, AbstractRange{Int64},
↪Base.AbstractCartesianIndex},N} where N} where A<:Union{Base.
↪ReinterpretArray{T,N,S,A} where S where A<:
↪Union{SubArray{T,N,A,I,true} where I<:Union{Tuple{Vararg{Real,N}
↪where N}, Tuple{AbstractUnitRange,Vararg{Any,N} where N}} where
↪A<:DenseArray where N where T, DenseArray} where N where T, Base.
↪ReshapedArray{T,N,A,MI} where MI<:Tuple{Vararg{Base.
↪MultiplicativeInverses.SignedMultiplicativeInverse{Int64},N}
↪where N} where A<:Union{Base.ReinterpretArray{T,N,S,A} where S
↪where A<:Union{SubArray{T,N,A,I,true} where I<:
```

```
    *(!Matched::Adjoint{#s662,#s661} where #s661<:LinearAlgebra.
↪AbstractTriangular where #s662, ::AbstractArray{T,1} where T) at /
↪buildworker/worker/package_linux64/build/usr/share/julia/stdlib/
↪v1.4/LinearAlgebra/src/triangular.jl:1971
    …


      Stacktrace:

       [1] top-level scope at In[74]:1
```

Instead, you could take the transpose to form a row vector

```
In [75]: ones(2)' * ones(2)
```

```
Out[75]: 2.0
```

Alternatively, for inner product in this setting use `dot()` or the unicode `\cdot<TAB>`

```
In [76]: @show dot(ones(2), ones(2))
         @show ones(2) ⋅ ones(2);
```

```
   dot(ones(2), ones(2)) = 2.0
ones(2) ⋅ ones(2) = 2.0
```

Matrix multiplication using one dimensional vectors similarly follows from treating them as column vectors. Post-multiplication requires a transpose

```
In [77]: b = ones(2, 2)
         b * ones(2)
```

```
Out[77]: 2-element Array{Float64,1}:
          2.0
          2.0
```

```
In [78]: ones(2)' * b
```

```
Out[78]: 1×2 Adjoint{Float64,Array{Float64,1}}:
          2.0  2.0
```

Note that the type of the returned value in this case is not `Array{Float64,1}` but rather `Adjoint{Float64,Array{Float64,1}}`.

This is since the left multiplication by a row vector should also be a row-vector. It also hints that the types in Julia more complicated than first appears in the surface notation, as we will explore further in the introduction to types lecture.

4.3 Elementwise Operations

4.3.1 Algebraic Operations

Suppose that we wish to multiply every element of matrix `A` with the corresponding element of matrix `B`.

In that case we need to replace `*` (matrix multiplication) with `.*` (elementwise multiplication).

For example, compare

```
In [79]: ones(2, 2) * ones(2, 2)   # matrix multiplication
```

```
Out[79]: 2×2 Array{Float64,2}:
          2.0  2.0
          2.0  2.0
```

```
In [80]: ones(2, 2) .* ones(2, 2)   # element by element multiplication
```

```
Out[80]: 2×2 Array{Float64,2}:
          1.0  1.0
          1.0  1.0
```

This is a general principle: `.x` means apply operator `x` elementwise

```
In [81]: A = -ones(2, 2)
```

```
Out[81]: 2×2 Array{Float64,2}:
          -1.0  -1.0
          -1.0  -1.0
```

```
In [82]: A.^2  # square every element
```

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

However in practice some operations are mathematically valid without broadcasting, and hence the `.` can be omitted.

```
In [83]: ones(2, 2) + ones(2, 2)  # same as ones(2, 2) .+ ones(2, 2)
```

```
Out[83]: 2×2 Array{Float64,2}:
          2.0  2.0
          2.0  2.0
```

Scalar multiplication is similar

```
In [84]: A = ones(2, 2)
```

```
Out[84]: 2×2 Array{Float64,2}:
          1.0  1.0
          1.0  1.0
```

```
In [85]: 2 * A  # same as 2 .* A
```

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

In fact you can omit the `*` altogether and just write `2A`.

Unlike MATLAB and other languages, scalar addition requires the `.+` in order to correctly broadcast

```
In [86]: x = [1, 2]
         x .+ 1     # not x + 1
         x .- 1     # not x - 1
```

```
Out[86]: 2-element Array{Int64,1}:
          0
          1
```

4.3.2 Elementwise Comparisons

Elementwise comparisons also use the `.x` style notation

```
In [87]: a = [10, 20, 30]
```

```
Out[87]: 3-element Array{Int64,1}:
          10
          20
          30
```

```
In [88]: b = [-100, 0, 100]
```

```
Out[88]: 3-element Array{Int64,1}:
          -100
             0
           100
```

```
In [89]: b .> a
```

```
Out[89]: 3-element BitArray{1}:
          0
          0
          1
```

```
In [90]: a .== b
```

```
Out[90]: 3-element BitArray{1}:
          0
          0
          0
```

We can also do comparisons against scalars with parallel syntax

```
In [91]: b
```

```
Out[91]: 3-element Array{Int64,1}:
          -100
             0
           100
```

```
In [92]: b .> 1
```

```
Out[92]: 3-element BitArray{1}:
          0
          0
          1
```

This is particularly useful for *conditional extraction* – extracting the elements of an array that satisfy a condition

```
In [93]: a = randn(4)
```

```
Out[93]: 4-element Array{Float64,1}:
          -0.6434467928769482
          -0.35303489730240734
           0.9772679080446901
          -1.4015326160821104
```

```
In [94]: a .< 0
```

```
Out[94]: 4-element BitArray{1}:
          1
          1
          0
          1
```

```
In [95]: a[a .< 0]
```

```
Out[95]: 3-element Array{Float64,1}:
          -0.6434467928769482
          -0.35303489730240734
          -1.4015326160821104
```

4.3.3 Changing Dimensions

The primary function for changing the dimensions of an array is `reshape()`

```
In [96]: a = [10, 20, 30, 40]
```

```
Out[96]: 4-element Array{Int64,1}:
          10
          20
          30
          40
```

```
In [97]: b = reshape(a, 2, 2)
```

```
Out[97]: 2×2 Array{Int64,2}:
          10  30
          20  40
```

```
In [98]: b
```

```
Out[98]: 2×2 Array{Int64,2}:
          10  30
          20  40
```

Notice that this function returns a view on the existing array.

This means that changing the data in the new array will modify the data in the old one.

```
In [99]: b[1, 1] = 100  # continuing the previous example
```

```
Out[99]: 100
```

```
In [100]: b
```

```
Out[100]: 2×2 Array{Int64,2}:
           100  30
            20  40
```

```
In [101]: a
```

```
Out[101]: 4-element Array{Int64,1}:
           100
            20
            30
            40
```

To collapse an array along one dimension you can use `dropdims()`

```
In [102]: a = [1 2 3 4]  # two dimensional
```

```
Out[102]: 1×4 Array{Int64,2}:
           1  2  3  4
```

```
In [103]: dropdims(a, dims = 1)
```

```
Out[103]: 4-element Array{Int64,1}:
           1
           2
           3
           4
```

The return value is an array with the specified dimension "flattened".

4.4 Broadcasting Functions

Julia provides standard mathematical functions such as `log`, `exp`, `sin`, etc.

```
In [104]: log(1.0)
```

```
Out[104]: 0.0
```

By default, these functions act *elementwise* on arrays

```
In [105]: log.(1:4)
```

```
Out[105]: 4-element Array{Float64,1}:
           0.0
           0.6931471805599453
           1.0986122886681098
           1.3862943611198906
```

Note that we can get the same result as with a comprehension or more explicit loop

```
In [106]: [ log(x) for x in 1:4 ]
```

```
Out[106]: 4-element Array{Float64,1}:
           0.0
           0.6931471805599453
           1.0986122886681098
           1.3862943611198906
```

Nonetheless the syntax is convenient.

4.5 Linear Algebra

(See linear algebra documentation)

Julia provides some a great deal of additional functionality related to linear operations

```
In [107]: A = [1 2; 3 4]
```

```
Out[107]: 2×2 Array{Int64,2}:
           1  2
           3  4
```

```
In [108]: det(A)
```

```
Out[108]: -2.0
```

```
In [109]: tr(A)
```

```
Out[109]: 5
```

```
In [110]: eigvals(A)
```

```
Out[110]: 2-element Array{Float64,1}:
           -0.3722813232690143
            5.372281323269014
```

```
In [111]: rank(A)
```

```
Out[111]: 2
```

5 Ranges

As with many other types, a `Range` can act as a vector.

```
In [112]: a = 10:12        # a range, equivalent to 10:1:12
          @show Vector(a)  # can convert, but shouldn't

          b = Diagonal([1.0, 2.0, 3.0])
          b * a .- [1.0; 2.0; 3.0]
```

```
Vector(a) = [10, 11, 12]
```

```
Out[112]: 3-element Array{Float64,1}:
            9.0
           20.0
           33.0
```

Ranges can also be created with floating point numbers using the same notation.

```
In [113]: a = 0.0:0.1:1.0  # 0.0, 0.1, 0.2, ... 1.0
```

```
Out[113]: 0.0:0.1:1.0
```

But care should be taken if the terminal node is not a multiple of the set sizes.

```
In [114]: maxval = 1.0
          minval = 0.0
          stepsize = 0.15
          a = minval:stepsize:maxval # 0.0, 0.15, 0.3, ...
          maximum(a) == maxval
```

```
Out[114]: false
```

To evenly space points where the maximum value is important, i.e., `linspace` in other languages

```
In [115]: maxval = 1.0
          minval = 0.0
          numpoints = 10
          a = range(minval, maxval, length=numpoints)
          # or range(minval, stop=maxval, length=numpoints)

          maximum(a) == maxval
```

```
Out[115]: true
```

6 Tuples and Named Tuples

(See tuples and named tuples documentation)

We were introduced to tuples earlier, which provide high-performance immutable sets of distinct types.

```
In [116]: t = (1.0, "test")
          t[1]            # access by index
          a, b = t        # unpack
          # t[1] = 3.0    # would fail as tuples are immutable
          println("a = $a and b = $b")
```

```
a = 1.0 and b = test
```

As well as **named tuples**, which extend tuples with names for each argument.

```
In [117]: t = (val1 = 1.0, val2 = "test")
          t.val1      # access by index
          # a, b = t  # bad style, better to unpack by name with @unpack
          println("val1 = $(t.val1) and val1 = $(t.val1)") # access by name
```

```
val1 = 1.0 and val1 = 1.0
```

While immutable, it is possible to manipulate tuples and generate new ones

```
In [118]: t2 = (val3 = 4, val4 = "test!!")
          t3 = merge(t, t2)  # new tuple

Out[118]: (val1 = 1.0, val2 = "test", val3 = 4, val4 = "test!!")
```

Named tuples are a convenient and high-performance way to manage and unpack sets of parameters

```
In [119]: function f(parameters)
              α, β = parameters.α, parameters.β  # poor style, error prone if
  ↪adding parameters
              return α + β
          end

          parameters = (α = 0.1, β = 0.2)
          f(parameters)

Out[119]: 0.30000000000000004
```

This functionality is aided by the `Parameters.jl` package and the `@unpack` macro

```
In [120]: using Parameters

          function f(parameters)
              @unpack α, β = parameters  # good style, less sensitive to errors
              return α + β
          end

          parameters = (α = 0.1, β = 0.2)
          f(parameters)

Out[120]: 0.30000000000000004
```

In order to manage default values, use the `@with_kw` macro

```
In [121]: using Parameters
          paramgen = @with_kw (α = 0.1, β = 0.2)  # create named tuples with defaults

          # creates named tuples, replacing defaults
          @show paramgen()  # calling without arguments gives all defaults
          @show paramgen(α = 0.2)
          @show paramgen(α = 0.2, β = 0.5);

           paramgen() = (α = 0.1, β = 0.2)
paramgen(α = 0.2) = (α = 0.2, β = 0.2)
paramgen(α = 0.2, β = 0.5) = (α = 0.2, β = 0.5)
```

An alternative approach, defining a new type using `struct` tends to be more prone to accidental misuse, and leads to a great deal of boilerplate code.

For that, and other reasons of generality, we will use named tuples for collections of parameters where possible.

7 Nothing, Missing, and Unions

Sometimes a variable, return type from a function, or value in an array needs to represent the absence of a value rather than a particular value.

There are two distinct use cases for this

1. `nothing` ("software engineers null"): used where no value makes sense in a particular context due to a failure in the code, a function parameter not passed in, etc.
2. `missing` ("data scientists null"): used when a value would make conceptual sense, but it isn't available.

7.1 Nothing and Basic Error Handling

The value `nothing` is a single value of type `Nothing`

```
In [122]: typeof(nothing)
```

```
Out[122]: Nothing
```

An example of a reasonable use of `nothing` is if you need to have a variable defined in an outer scope, which may or may not be set in an inner one

```
In [123]: function f(y)
              x = nothing
              if y > 0.0
                  # calculations to set `x`
                  x = y
              end

              # later, can check `x`
              if isnothing(x)
                  println("x was not set")
              else
                  println("x = $x")
              end
              x
          end

          @show f(1.0)
          @show f(-1.0);
```

```
    x = 1.0
f(1.0) = 1.0
x was not set
f(-1.0) = nothing
```

While in general you want to keep a variable name bound to a single type in Julia, this is a notable exception.

Similarly, if needed, you can return a `nothing` from a function to indicate that it did not calculate as expected.

In [124]:
```
function f(x)
    if x > 0.0
        return sqrt(x)
    else
        return nothing
    end
end
x1 = 1.0
x2 = -1.0
y1 = f(x1)
y2 = f(x2)

# check results with isnothing
if isnothing(y1)
    println("f($x2) successful")
else
    println("f($x2) failed");
end
```

```
f(-1.0) failed
```

As an aside, an equivalent way to write the above function is to use the ternary operator, which gives a compact if/then/else structure

In [125]:
```
function f(x)
    x > 0.0 ? sqrt(x) : nothing  # the "a ? b : c" pattern is the ternary
end

f(1.0)
```

Out[125]:
```
1.0
```

We will sometimes use this form when it makes the code more clear (and it will occasionally make the code higher performance).

Regardless of how `f(x)` is written, the return type is an example of a union, where the result could be one of an explicit set of types.

In this particular case, the compiler would deduce that the type would be a `Union{Nothing,Float64}` – that is, it returns either a floating point or a `nothing`.

You will see this type directly if you use an array containing both types

In [126]:
```
x = [1.0, nothing]
```

Out[126]:
```
2-element Array{Union{Nothing, Float64},1}:
 1.0
  nothing
```

When considering error handling, whether you want a function to return `nothing` or simply fail depends on whether the code calling `f(x)` is carefully checking the results.

For example, if you were calling on an array of parameters where a priori you were not sure which ones will succeed, then

```
In [127]: x = [0.1, -1.0, 2.0, -2.0]
          y = f.(x)

          # presumably check `y`
```

```
Out[127]: 4-element Array{Union{Nothing, Float64},1}:
           0.31622776601683794
            nothing
           1.4142135623730951
            nothing
```

On the other hand, if the parameter passed is invalid and you would prefer not to handle a graceful failure, then using an assertion is more appropriate.

```
In [128]: function f(x)
              @assert x > 0.0
              sqrt(x)
          end

          f(1.0)
```

```
Out[128]: 1.0
```

Finally, `nothing` is a good way to indicate an optional parameter in a function

```
In [129]: function f(x; z = nothing)

              if isnothing(z)
                  println("No z given with $x")
              else
                  println("z = $z given with $x")
              end
          end

          f(1.0)
          f(1.0, z=3.0)
```

```
          No z given with 1.0
      z = 3.0 given with 1.0
```

An alternative to `nothing`, which can be useful and sometimes higher performance, is to use `NaN` to signal that a value is invalid returning from a function.

```
In [130]: function f(x)
              if x > 0.0
                  return x
              else
                  return NaN
              end
          end

          f(0.1)
```

```
f(-1.0)

@show typeof(f(-1.0))
@show f(-1.0) == NaN  # note, this fails!
@show isnan(f(-1.0))  # check with this
```

```
 typeof(f(-1.0)) = Float64
f(-1.0) == NaN = false
isnan(f(-1.0)) = true
```

```
Out[130]: true
```

Note that in this case, the return type is `Float64` regardless of the input for `Float64` input.

Keep in mind, though, that this only works if the return type of a function is `Float64`.

7.2 Exceptions

(See exceptions documentation)

While returning a `nothing` can be a good way to deal with functions which may or may not return values, a more robust error handling method is to use exceptions.

Unless you are writing a package, you will rarely want to define and throw your own exceptions, but will need to deal with them from other libraries.

The key distinction for when to use an exceptions vs. return a `nothing` is whether an error is unexpected rather than a normal path of execution.

An example of an exception is a `DomainError`, which signifies that a value passed to a function is invalid.

```
In [131]: # throws exception, turned off to prevent breaking notebook
          # sqrt(-1.0)

          # to see the error
          try sqrt(-1.0); catch err; err end  # catches the exception and prints it
```

```
Out[131]: DomainError(-1.0, "sqrt will only return a complex result if called with
  ↪a complex
          argument. Try sqrt(Complex(x)).")
```

Another example you will see is when the compiler cannot convert between types.

```
In [132]: # throws exception, turned off to prevent breaking notebook
          # convert(Int64, 3.12)

          # to see the error
          try convert(Int64, 3.12); catch err; err end  # catches the exception
  ↪and prints it.
```

```
Out[132]: InexactError(:Int64, Int64, 3.12)
```

If these exceptions are generated from unexpected cases in your code, it may be appropriate simply let them occur and ensure you can read the error.

Occasionally you will want to catch these errors and try to recover, as we did above in the `try` block.

```
In [133]: function f(x)
              try
                  sqrt(x)
              catch err                # enters if exception thrown
                  sqrt(complex(x, 0))  # convert to complex number
              end
          end

          f(0.0)
          f(-1.0)
```

```
Out[133]: 0.0 + 1.0im
```

7.3 Missing

(see "missing" documentation)

The value `missing` of type `Missing` is used to represent missing value in a statistical sense.

For example, if you loaded data from a panel, and gaps existed

```
In [134]: x = [3.0, missing, 5.0, missing, missing]
```

```
Out[134]: 5-element Array{Union{Missing, Float64},1}:
           3.0
           missing
           5.0
           missing
           missing
```

A key feature of `missing` is that it propagates through other function calls - unlike `nothing`

```
In [135]: f(x) = x^2

          @show missing + 1.0
          @show missing * 2
          @show missing * "test"
          @show f(missing);      # even user-defined functions
          @show mean(x);
```

```
missing + 1.0 = missing
missing * 2 = missing
missing * "test" = missing
f(missing) = missing
mean(x) = missing
```

The purpose of this is to ensure that failures do not silently fail and provide meaningless numerical results.

This even applies for the comparison of values, which

```
In [136]: x = missing
```

```
@show x == missing
@show x === missing  # an exception
@show ismissing(x);
```

```
 x == missing = missing
x === missing = true
ismissing(x) = true
```

Where `ismissing` is the canonical way to test the value.

In the case where you would like to calculate a value without the missing values, you can use `skipmissing`.

```
In [137]: x = [1.0, missing, 2.0, missing, missing, 5.0]
```

```
@show mean(x)
@show mean(skipmissing(x))
@show coalesce.(x, 0.0);  # replace missing with 0.0;
```

```
 mean(x) = missing
mean(skipmissing(x)) = 2.6666666666666665
coalesce.(x, 0.0) = [1.0, 0.0, 2.0, 0.0, 0.0, 5.0]
```

As `missing` is similar to R's `NA` type, we will see more of `missing` when we cover `DataFrames`.

8 Exercises

8.1 Exercise 1

This exercise uses matrix operations that arise in certain problems, including when dealing with linear stochastic difference equations.

If you aren't familiar with all the terminology don't be concerned – you can skim read the background discussion and focus purely on the matrix exercise.

With that said, consider the stochastic difference equation

$$X_{t+1} = AX_t + b + \Sigma W_{t+1} \tag{1}$$

Here

- X_t, b and X_{t+1} are $n \times 1$
- A is $n \times n$

- Σ is $n \times k$
- W_t is $k \times 1$ and $\{W_t\}$ is iid with zero mean and variance-covariance matrix equal to the identity matrix

Let S_t denote the $n \times n$ variance-covariance matrix of X_t.

Using the rules for computing variances in matrix expressions, it can be shown from (1) that $\{S_t\}$ obeys

$$S_{t+1} = AS_tA' + \Sigma\Sigma' \tag{2}$$

It can be shown that, provided all eigenvalues of A lie within the unit circle, the sequence $\{S_t\}$ converges to a unique limit S.

This is the **unconditional variance** or **asymptotic variance** of the stochastic difference equation.

As an exercise, try writing a simple function that solves for the limit S by iterating on (2) given A and Σ.

To test your solution, observe that the limit S is a solution to the matrix equation

$$S = ASA' + Q \quad \text{where} \quad Q := \Sigma\Sigma' \tag{3}$$

This kind of equation is known as a **discrete time Lyapunov equation**.

The QuantEcon package provides a function called `solve_discrete_lyapunov` that implements a fast "doubling" algorithm to solve this equation.

Test your iterative method against `solve_discrete_lyapunov` using matrices

$$A = \begin{bmatrix} 0.8 & -0.2 \\ -0.1 & 0.7 \end{bmatrix} \qquad \Sigma = \begin{bmatrix} 0.5 & 0.4 \\ 0.4 & 0.6 \end{bmatrix}$$

8.2 Exercise 2

Take a stochastic process for $\{y_t\}_{t=0}^T$

$$y_{t+1} = \gamma + \theta y_t + \sigma w_{t+1}$$

where

- w_{t+1} is distributed `Normal(0,1)`
- $\gamma = 1, \sigma = 1, y_0 = 0$
- $\theta \in \Theta \equiv \{0.8, 0.9, 0.98\}$

Given these parameters

- Simulate a single y_t series for each $\theta \in \Theta$ for $T = 150$. Feel free to experiment with different T.
- Overlay plots of the rolling mean of the process for each $\theta \in \Theta$, i.e. for each $1 \leq \tau \leq T$ plot

$$\frac{1}{\tau}\sum_{t=1}^{\tau} y_T$$

- Simulate $N = 200$ paths of the stochastic process above to the T, for each $\theta \in \Theta$, where we refer to an element of a particular simulation as y_t^n.
- Overlay plots a histogram of the stationary distribution of the final y_T^n for each $\theta \in \Theta$. Hint: pass `alpha` to a plot to make it transparent (e.g. `histogram(vals, alpha = 0.5)`) or use `stephist(vals)` to show just the step function for the histogram.
- Numerically find the mean and variance of this as an ensemble average, i.e. $\sum_{n=1}^N \frac{y_T^n}{N}$ and $\sum_{n=1}^N \frac{(y_T^n)^2}{N} - \left(\sum_{n=1}^N \frac{y_T^n}{N}\right)^2$.

Later, we will interpret some of these in this lecture.

8.3 Exercise 3

Let the data generating process for a variable be

$$y = ax_1 + bx_1^2 + cx_2 + d + \sigma w$$

where y, x_1, x_2 are scalar observables, a, b, c, d are parameters to estimate, and w are iid normal with mean 0 and variance 1.

First, let's simulate data we can use to estimate the parameters

- Draw $N = 50$ values for x_1, x_2 from iid normal distributions.

Then, simulate with different w * Draw a w vector for the `N` values and then `y` from this simulated data if the parameters were $a = 0.1, b = 0.2 c = 0.5, d = 1.0, \sigma = 0.1$. * Repeat that so you have `M = 20` different simulations of the `y` for the `N` values.

Finally, calculate order least squares manually (i.e., put the observables into matrices and vectors, and directly use the equations for OLS rather than a package).

- For each of the `M=20` simulations, calculate the OLS estimates for a, b, c, d, σ.
- Plot a histogram of these estimates for each variable.

8.4 Exercise 4

Redo Exercise 1 using the `fixedpoint` function from `NLsolve` this lecture.

Compare the number of iterations of the NLsolve's Anderson Acceleration to the handcoded iteration used in Exercise 1.

Hint: Convert the matrix to a vector to use `fixedpoint`. e.g. `A = [1 2; 3 4]` then `x = reshape(A, 4)` turns it into a vector. To reverse, `reshape(x, 2, 2)`.

9 Solutions

9.1 Exercise 1

Here's the iterative approach

```
In [138]: function compute_asymptotic_var(A, Σ;
                                          S0 = Σ * Σ',
                                          tolerance = 1e-6,
                                          maxiter = 500)
              V = Σ * Σ'
              S = S0
              err = tolerance + 1
              i = 1
              while err > tolerance && i ≤ maxiter
                  next_S = A * S * A' + V
                  err = norm(S - next_S)
                  S = next_S
                  i += 1
              end
              return S
          end
```

```
Out[138]: compute_asymptotic_var (generic function with 1 method)
```

```
In [139]: A = [0.8  -0.2;
               -0.1  0.7]

          Σ = [0.5 0.4;
               0.4 0.6]
```

```
Out[139]: 2×2 Array{Float64,2}:
           0.5  0.4
           0.4  0.6
```

Note that all eigenvalues of A lie inside the unit disc.

```
In [140]: maximum(abs, eigvals(A))
```

```
Out[140]: 0.9
```

Let's compute the asymptotic variance

```
In [141]: our_solution = compute_asymptotic_var(A, Σ)
```

```
Out[141]: 2×2 Array{Float64,2}:
           0.671228  0.633476
           0.633476  0.858874
```

Now let's do the same thing using QuantEcon's `solve_discrete_lyapunov()` function and check we get the same result.

```
In [142]: using QuantEcon
```

```
In [143]: norm(our_solution - solve_discrete_lyapunov(A, Σ * Σ'))
```

```
Out[143]: 3.883245447999784e-6
```