QuantEcon Julia Cheatsheet

1 Variables

Here are a few examples of basic kinds of variables we might be interested in creating.

Command	Description

`A = 4.1`
`B = [1, 2, 3]`
`C = [1.1 2.2 3.3]`
`D = [1 2 3]'`
`E = [1 2; 3 4]`
	How to **create a scalar, a vector, or a matrix**. Here, each example will result in a slightly different form of output. `A` is a scalar, `B` is a flat array with 3 elements, `C` is a 1 by 3 vector, `D` is a 3 by 1 vector, and `E` is a 2 by 2 matrix.

`s = "This is a string"`	A **string** variable
`x = true`	A **Boolean** variable

2 Vectors and Matrices

These are a few kinds of special vectors/matrices we can create and some things we can do with them.

Command	Description
`A = zeros(m, n)`	Creates a **matrix of all zeros** of size `m` by `n`. We can also do the following: `A = zeros(B)` which will create a matrix of all zeros with the same dimensions as matrix or vector `B`.
`A = ones(m, n)`	Creates a **matrix of all ones** of size `m` by `n`. We can also do the following: `A = ones(B)` which will create a matrix of all ones with the same dimensions as matrix or vector `B`.
`A = eye(n)`	Creates an `n` by `n` **identity matrix**. For example, `eye(3)` will return $\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$.
`A = j:k:n`	This will create a **sequence** starting at `j`, ending at `n`, with difference `k` between points. For example, `A = 2:4:10` will create the sequence `2, 6, 10`. To convert the output to an array, use `collect(A)`.

`A = linspace(j, n, m)`	This will create a **sequence** of `m` points starting at `j`, ending at `n`. For example, `A = linspace(2, 10, 3)` will create the sequence `2.0, 6.0, 10.0`. To convert the output to an array, use `collect(A)`.
`A = diagm(x)`	Creates a **diagonal matrix** using the elements in `x`. For example, if `x = [1, 2, 3]`, `diagm(x)` will return $\begin{pmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 3\end{pmatrix}$.
`A = rand(m, n)`	Creates an `m` by `n` **matrix of random numbers** drawn from a **uniform distribution** on $[0, 1]$. Alternatively, `rand` can be used to draw random elements from a set `X`. For example, if `X = [1, 2, 3]`, `rand(X)` will return either `1`, `2`, or `3`.
`A = randn(m, n)`	Creates an `m` by `n` **matrix of random numbers** drawn form a **standard normal distribution**.
`A[m, n]`	This is the general syntax for **accessing elements** of an array or matrix, where `m` and `n` are integers. The example here returns the element in the second row and third column. • We can also use ranges (like `1:3`) in place of single numbers to extract multiple rows or columns. • A colon, `:`, by itself indicates all rows or columns • The word `end` can also be used to indicate the last row or column.

`nrow, ncol = size(A)`	**Returns the number of rows and columns** in a matrix. Alternatively, we can do:
	`nrow = size(A, 1)`
	and
	`ncol = size(A, 2)`

`diag(A)`	This function returns a vector of the **diagonal elements** of `A` (i.e., `A[1, 1]`, `A[2, 2]`, etc...).
`A = hcat([1 2], [3 4])`	**Horizontally concatenates** two matrices or vectors. The example here would return $\begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix}$. An alternative syntax is: `A = [[1 2] [3 4]]` For either of these commands to work, both matrices or vectors must have the same number of rows.
`A = vcat([1 2], [3 4])`	**Vertically concatenates** two matrices or vectors. The example here would return $\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$. An alternative syntax is: `A = [[1 2]; [3 4]]` For either of these commands to work, both matrices or vectors must have the same number of columns.

`A = reshape(a, m, n)`	**Reshapes** matrix or vector `a` into a new matrix or vector, `A` with `m` rows and `n` columns. For example, `A = reshape(1:10, 5, 2)` would return $\begin{pmatrix}1 & 6\\2 & 7\\3 & 8\\4 & 9\\5 & 10\end{pmatrix}$ For this to work, the number of elements in `a` (number of rows times number of columns) must equal `m * n`.
`A[:]`	**Converts matrix A to a vector.** For example, if `A = [1 2; 3 4]`, then `A[:]` will return $\begin{pmatrix}1\\3\\2\\4\end{pmatrix}$.
`flipdim(A, d)`	Reverses the vector or matrix `A` along dimension `d`. For example, if `A = [1 2 3; 4 5 6]`, `flipdim(A, 1)`, will reverse the rows of `A` and return $\begin{pmatrix}4 & 5 & 6\\1 & 2 & 3\end{pmatrix}$. `flipdim(A, 2)` will reverse the columns of `A` and return $\begin{pmatrix}3 & 2 & 1\\6 & 5 & 4\end{pmatrix}$.
`repmat(A, m, n)`	**Repeats matrix** `A`, `m` times in the row direction and `n` in the column direction. For example, if `A = [1 2; 3 4]`, `repmat(A, 2, 3)` will return $\begin{pmatrix}1 & 2 & 1 & 2 & 1 & 2\\3 & 4 & 3 & 4 & 3 & 4\\1 & 2 & 1 & 2 & 1 & 2\\3 & 4 & 3 & 4 & 3 & 4\end{pmatrix}$.

3 Mathematical Functions

Here, we cover some useful functions for doing math.

Command	**Description**

`5 + 2`
`5 - 2`
`5 * 2`
`5 \ 2`
`5 ^ 2`
`5 % 2`
	Scalar arithmetic operations: addition, subtraction, multiplication, division, power, remainder.

`A + B`
`A - B`
`A .* B`
`A ./ B`
`A .^ B`
`A .% B`
	Element-by-element operations on matrices. This syntax applies the operation element-wise to corresponding elements of the matrices.

`A * B`	When `A` and `B` are matrices, `*` will perform **matrix multiplication**, as long as the number of columns in `A` is the same as the number of columns in `B`.

`dot(A, B)`	This function returns the **dot product/inner product** of the two vectors `A` and `B`. The two vectors need to be dimensionless or column vectors.
`A'`	This syntax returns the **transpose** of the matrix `A` (i.e., reverses the dimensions of `A`). For example if $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, then `A'` returns $\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$.

`sum(A)`
`maximum(A)`
`minumum(A)`
	These functions compute the sum, maximum, and minimum elements, respectively, in matrix or vector `A`. We can also add an additional argument for the dimension to compute the sum/maximum/minumum across. For example `sum(A, 2)` will compute the row sums of `A` and `maximum(A, 1)` will compute the maxima of each column of `A`.

`inv(A)`	This function returns the **inverse** of the matrix `A`. Alternatively, we can do: `A ^ (-1)`
`det(A)`	This function returns the **determinant** of the matrix `A`.
`val, vec = eig(A)`	Returns the **eigenvalues** (`val`) and **eigenvectors** (`vec`) of matrix `A`. In the output, `val[i]` is the eigenvalue corresponding to eigenvector `val[:, i]`.
`norm(A)`	Returns the Euclidean **norm** of matrix or vector `A`. We can also provide an argument `p`, like so: `norm(A, p)` which will compute the `p`-norm (the default `p` is 2). If `A` is a matrix, valid values of `p` are `1, 2` amd `Inf`.
`A \ b`	If `A` is square, this syntax **solves the linear system** $Ax = b$. Therefore, it returns `x` such that `A * x = b`. If `A` is rectangular, it **solves for the least-squares solution** to the problem.

4 Programming

The following are useful basics for Julia programming.

Command	**Description**

`# One line comment`
 	
`#=`
`Comment`
`block`
`=#`
	Two ways to make **comments**. Comments are useful for annotating code and explaining what it does. The first example limits your comment to one line and the second example allows the comments to span multiple lines between the `#=` and `=#`.

`for i in iterable`
`    # do something`
`end`
	A **for loop** is used to perform a sequence of commands for each element in an iterable object, such as an array. For example, the following for loop fills the vector `l` with the squares of the integers from 1 to 3:
	`N = 3`
	`l = zeros(N, 1)`
	`for i = 1:N`
	`    l[i] = i ^ 2`
	`end`

`while i <= N`
`    # do something`
`end`
	A **while loop** performs a sequence of commands as long as some condition is true. For example, the following while loop achieves the same result as the for loop above
	`N = 3`
	`l = zeros(N, 1)`
	`i = 1`
	`while i <= N`
	`    l[i] = i ^ 2`
	`    i = i + 1`
	`end`

`if i <= N`
`    # do something`
`else`
`    # do something else`
`end`
	An **if/else statement** performs commands if a condition is met. For example, the following squares `x` is `x` is 5, and cubes it otherwise:
	`if x == 5`
	`    x = x ^ 2`
	`else`
	`    x = x ^ 3`
	`end`
	We can also just have an if statement on its own, in which case it would square `x` if `x` is 5, and do nothing otherwise.
	`if x == 5`
	`    x = x ^ 2`
	`end`

`fun(x, y) = 5 * x + y`
 	
`function fun(x, y)`
`    ret = 5 * x`
`    return ret + y`
`end`
	These are two ways to define **functions**. Both examples here define equivalent functions.
	The first method is for defining a function on one line. The name of the function is `fun` and it takes two inputs, `x` and `y`, which are specified between the parentheses. The code after the equals sign tells Julia what the output of the function is.
	The second method is used to create functions of more than one line. The name of the function, `fun`, is specified right after `function`, and like the one-line version, has its arguments in parentheses. The `return` statement specifies the output of the function.

`println("Hello world")`	How to **print** to screen. We can also print the values of variables to screen: `println("The value of x is $(x).")`