[back to article]

The Matrix Cheatsheet by Sebastian Raschka is licensed under a Creative Commons Attribution 4.0 International License.

(last updated: June 22, 2018)

CREATING MATRICES

Task	MATLAB/Octave	Python NumPy	R	Julia	Task

Creating Matrices
 	
(here: 3x3 matrix)
	`M> A = [1 2 3; 4 5 6; 7 8 9]`
	
	`A =`
	`   1   2   3`
	`   4   5   6`
	`   7   8   9`
		`P> A = np.array([ [1,2,3], [4,5,6], [7,8,9] ])`
		
		`P> A`
		`array([[1, 2, 3],`
		`       [4, 5, 6],`
		`       [7, 8, 9]])`
			`R> A = matrix(c(1,2,3,4,5,6,7,8,9),nrow=3,byrow=T)`
			
			` # equivalent to`
			`# A = matrix(1:9,nrow=3,byrow=T)`
			
			`R> A`
			`[,1] [,2] [,3]`
			`[1,] 1 2 3`
			`[2,] 4 5 6`
			`[3,] 7 8 9`
				`J> A=[1 2 3; 4 5 6; 7 8 9]`
				
				`3x3 Array{Int64,2}:`
				`1 2 3`
				`4 5 6`
				`7 8 9`
					Creating Matrices
					
					(here: 3x3 matrix)

Creating an column vector (nx1 matrix)	`M> a = [1; 2; 3]`
	`a =`
	`   1`
	`   2`
	`   3`
		`P> a = np.array([1,2,3]).reshape(3,1)`
		
		`P> b.shape`
		`(3, 1)`
			`R> a = matrix(c(1,2,3), nrow=3, byrow=T)`
			
			`R> a`
			`[,1]`
			`[1,] 1`
			`[2,] 2`
			`[3,] 3`
				`J> a=[1; 2; 3]`
				`3-element Array{Int64,1}:`
				`1`
				`2`
				`3`
					Creating a column vector (nx1 matrix)

Creating an
row vector (1xn matrix)
	`M> b = [1 2 3]`
	`b =`
	`   1   2   3`
		`P> b = np.array([1,2,3]).reshape(1, 3)`
		
		`P> b`
		`array([[1],`
		`       [2],`
		`       [3]])`
		
		`# note that in numpy, 1D arrays`
		`# can be multiplied`
		`# with 2d arrays, too`
		
		`P> b.shape`
		
		`(1, 3)`
			`R> b = matrix(c(1,2,3), ncol=3)`
			
			`R> b`
			`[,1] [,2] [,3]`
			
			`[1,] 1 2 3`
				`J> b=[1 2 3]`
				`1x3 Array{Int64,2}:`
				`1 2 3`
				
				`# note that this is a 2D array.`
					Creating an
					row vector (1xn matrix)

Creating a
 	
random m x n matrix
	`M> rand(3,2)`
	
	`ans =`
	
	`   0.21977   0.10220`
	
	`   0.38959   0.69911`
		`P> np.random.rand(3,2)`
		
		`array([[ 0.29347865,  0.17920462],`
		
		`       [ 0.51615758,  0.64593471],`
		
		`       [ 0.01067605,  0.09692771]])`
			`R> matrix(runif(3*2), ncol=2)`
			
			`[,1] [,2]`
			
			`[1,] 0.5675127 0.7751204`
			
			`[2,] 0.3439412 0.5261893`
				`J> rand(3,2)`
				
				`3x2 Array{Float64,2}:`
				
				`0.36882 0.267725`
				
				`0.571856 0.601524`
					Creating a
					
					random m x n matrix

	0.15624 0.65637		[3,] 0.2273177 0.223438	0.848084 0.858935	
Creating a **zero m x n matrix**	M> zeros(3,2) ans = 0 0 0 0 0 0	P> np.zeros((3,2)) array([[0., 0.], [0., 0.], [0., 0.]])	R> mat.or.vec(3, 2) [,1] [,2] [1,] 0 0 [2,] 0 0 [3,] 0 0	J> zeros(3,2) 3x2 Array{Float64,2}: 0.0 0.0 0.0 0.0 0.0 0.0	**Creating a** **zero m x n matrix**
Creating an **m x n matrix of ones**	M> ones(3,2) ans = 1 1 1 1 1 1	P> np.ones((3,2)) array([[1., 1.], [1., 1.], [1., 1.]])	R> matrix(1L, 3, 2) [,1] [,2] [1,] 1 1 [2,] 1 1 [3,] 1 1	J> ones(3,2) 3x2 Array{Float64,2}: 1.0 1.0 1.0 1.0 1.0 1.0	**Creating an** **m x n matrix of ones**
Creating an **identity matrix**	M> eye(3) ans = Diagonal Matrix 1 0 0 0 1 0 0 0 1	P> np.eye(3) array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])	R> diag(3) [,1] [,2] [,3] [1,] 1 0 0 [2,] 0 1 0 [3,] 0 0 1	J> eye(3) 3x3 Array{Float64,2}: 1.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 1.0	**Creating an** **identity matrix**
Creating a **diagonal matrix**	M> a = [1 2 3] M> diag(a) ans = Diagonal Matrix 1 0 0 0 2 0 0 0 3	P> a = np.array([1,2,3]) P> np.diag(a) array([[1, 0, 0], [0, 2, 0], [0, 0, 3]])	R> diag(1:3) [,1] [,2] [,3] [1,] 1 0 0 [2,] 0 2 0 [3,] 0 0 3	J> a=[1, 2, 3] # added commas because julia # vectors are columnar J> diagm(a) 3x3 Array{Int64,2}: 1 0 0 0 2 0 0 0 3	**Creating a** **diagonal matrix**

ACCESSING MATRIX ELEMENTS

Getting the dimension **of a matrix** **(here: 2D, rows x cols)**	M> A = [1 2 3; 4 5 6] A = 1 2 3 4 5 6	P> A = np.array([[1,2,3], [4,5,6]]) P> A array([[1, 2, 3], [4, 5, 6]])	R> A = matrix(1:6,nrow=2,byrow=T) R> A [,1] [,2] [,3] [1,] 1 2 3	J> A=[1 2 3; 4 5 6] 2x3 Array{Int64,2}: 1 2 3 4 5 6	**Getting the dimension** **of a matrix** **(here: 2D, rows x cols)**

	M> size(A) ans = 2 3	P> A.shape (2, 3)	[2,] 4 5 6 R> dim(A) [1] 2 3	J> size(A) (2,3)	
Selecting rows	M> A = [1 2 3; 4 5 6; 7 8 9] % 1st row M> A(1,:) ans = 1 2 3 % 1st 2 rows M> A(1:2,:) ans = 1 2 3 4 5 6	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) # 1st row P> A[0,:] array([1, 2, 3]) # 1st 2 rows P> A[0:2,:] array([[1, 2, 3], [4, 5, 6]])	R> A = matrix(1:9,nrow=3,byrow=T) # 1st row R> A[1,] [1] 1 2 3 # 1st 2 rows R> A[1:2,] [,1] [,2] [,3] [1,] 1 2 3 [2,] 4 5 6	J> A=[1 2 3; 4 5 6; 7 8 9]; #semicolon suppresses output #1st row J> A[1,:] 1x3 Array{Int64,2}: 1 2 3 #1st 2 rows J> A[1:2,:] 2x3 Array{Int64,2}: 1 2 3 4 5 6	**Selecting rows**
Selecting columns	M> A = [1 2 3; 4 5 6; 7 8 9] % 1st column M> A(:,1) ans = 1 4 7 % 1st 2 columns M> A(:,1:2) ans = 1 2 4 5 7 8	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) # 1st column (as row vector) P> A[:,0] array([1, 4, 7]) # 1st column (as column vector) P> A[:,[0]] array([[1], [4], [7]]) # 1st 2 columns P> A[:,0:2] array([[1, 2], [4, 5], [7, 8]])	R> A = matrix(1:9,nrow=3,byrow=T) # 1st column as row vector R> t(A[,1]) [,1] [,2] [,3] [1,] 1 4 7 # 1st column as column vector R> A[,1] [1] 1 4 7 # 1st 2 columns R> A[,1:2] [,1] [,2] [1,] 1 2 [2,] 4 5 [3,] 7 8	J> A=[1 2 3; 4 5 6; 7 8 9]; #1st column J> A[:,1] 3-element Array{Int64,1}: 1 4 7 #1st 2 columns J> A[:,1:2] 3x2 Array{Int64,2}: 1 2 4 5 7 8	**Selecting columns**
Extracting rows and columns by criteria **(here: get rows that have value 9 in column 3)**	M> A = [1 2 3; 4 5 9; 7 8 9] A = 1 2 3 4 5 9 7 8 9 M> A(A(:,3) == 9,:) ans = 4 5 9 7 8 9	P> A = np.array([[1,2,3], [4,5,9], [7,8,9]]) P> A array([[1, 2, 3], [4, 5, 9], [7, 8, 9]]) P> A[A[:,2] == 9] array([[4, 5, 9], [7, 8, 9]])	R> A = matrix(1:9,nrow=3,byrow=T) R> A [,1] [,2] [,3] [1,] 1 2 3 [2,] 4 5 9 [3,] 7 8 9 R> A[A[,3]==9,]	J> A=[1 2 3; 4 5 9; 7 8 9] 3x3 Array{Int64,2}: 1 2 3 4 5 9 7 8 9 # use '.==' for # element-wise check J> A[A[:,3] .==9, :] 2x3 Array{Int64,2}:	**Extracting rows and columns by criteria** **(here: get rows that have value 9 in column 3)**

			[1] 7 8 9	4 5 9 7 8 9	
Accessing elements **(here: 1st element)**	M> A = [1 2 3; 4 5 6; 7 8 9] M> A(1,1) ans = 1	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) P> A[0,0] 1	R> A = matrix(c(1,2,3,4,5,9,7,8,9),nrow=3,byrow=T) R> A[1,1] [1] 1	J> A=[1 2 3; 4 5 6; 7 8 9]; J> A[1,1] 1	**Accessing elements** **(here: 1st element)**

MANIPULATING SHAPE AND DIMENSIONS

Converting **a matrix into a row vector (by column)**	M> A = [1 2 3; 4 5 6; 7 8 9] M> A(:) ans = 1 4 7 2 5 8 3 6 9	P> A = np.array([[1,2,3],[4,5,6],[7,8,9]]) P> A.flatten(1) # returns a copy array([1, 4, 7, 2, 5, 8, 3, 6, 9]) # alternatively A.ravel() # ravel() returns a view	R> A = matrix(1:9,nrow=3,byrow=T) R> as.vector(A) [1] 1 4 7 2 5 8 3 6 9	J> A=[1 2 3; 4 5 6; 7 8 9] J> vec(A) 9-element Array{Int64,1}: 1 4 7 2 5 8 3 6 9	**Converting** **a matrix into a row vector (by column)**
Converting **row to column vectors**	M> b = [1 2 3] M> b = b' b = 1 2 3	P> b = np.array([1, 2, 3]) P> b = b[np.newaxis].T # alternatively # b = b[:,np.newaxis] P> b array([[1], [2], [3]])	R> b = matrix(c(1,2,3), ncol=3) R> t(b) [,1] [1,] 1 [2,] 2 [3,] 3	J> b=vec([1 2 3]) 3-element Array{Int64,1}: 1 2 3	**Converting** **row to column vectors**
Reshaping Matrices **(here: 3x3 matrix to row vector)**	M> A = [1 2 3; 4 5 6; 7 8 9] A = 1 2 3 4 5 6 7 8 9 M> total_elements = numel(A)	P> A = np.array([[1,2,3],[4,5,9],[7,8,9]]) P> A array([[1, 2, 3], [4, 5, 9], [7, 8, 9]]) P> total_elements = np.prod(A.shape)	R> A = matrix(1:9,nrow=3,byrow=T) R> A [,1] [,2] [,3] [1,] 1 2 3 [2,] 4 5 6 [3,] 7 8 9	J> A=[1 2 3; 4 5 6; 7 8 9] 3x3 Array{Int64,2}: 1 2 3 4 5 6 7 8 9 J> total_elements=length(A) 9	**Reshaping Matrices** **(here: 3x3 matrix to row vector)**

	M> B = reshape(A,1,total_elements) % or reshape(A,1,9) B = 1 4 7 2 5 8 3 6 9	P> B = A.reshape(1, total_elements) # alternative shortcut: # A.reshape(1,-1) P> B array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])	R> total_elements = dim(A)[1] * dim(A)[2] R> B = matrix(A, ncol=total_elements) R> B [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [1,] 1 4 7 2 5 8 3 6 9	J>B=reshape(A,1,total_elements) 1x9 Array{Int64,2}: 1 4 7 2 5 8 3 6 9	
Concatenating matrices	M> A = [1 2 3; 4 5 6] M> B = [7 8 9; 10 11 12] M> C = [A; B] 1 2 3 4 5 6 7 8 9 10 11 12	P> A = np.array([[1, 2, 3], [4, 5, 6]]) P> B = np.array([[7, 8, 9],[10,11,12]]) P> C = np.concatenate((A, B), axis=0) P> C array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])	R> A = matrix(1:6,nrow=2,byrow=T) R> B = matrix(7:12,nrow=2,byrow=T) R> C = rbind(A,B) R> C [,1] [,2] [,3] [1,] 1 2 3 [2,] 4 5 6 [3,] 7 8 9 [4,] 10 11 12	J> A=[1 2 3; 4 5 6]; J> B=[7 8 9; 10 11 12]; J> C=[A; B] 4x3 Array{Int64,2}: 1 2 3 4 5 6 7 8 9 10 11 12	**Concatenating matrices**
Stacking **vectors and matrices**	M> a = [1 2 3] M> b = [4 5 6] M> c = [a' b'] c = 1 4 2 5 3 6 M> c = [a; b] c = 1 2 3 4 5 6	P> a = np.array([1,2,3]) P> b = np.array([4,5,6]) P> np.column_stack([a,b]) array([[1, 4], [2, 5], [3, 6]]) P> np.row_stack([a,b]) array([[1, 2, 3], [4, 5, 6]])	R> a = matrix(1:3, ncol=3) R> b = matrix(4:6, ncol=3) R> matrix(rbind(A, B), ncol=2) [,1] [,2] [1,] 1 5 [2,] 4 3 R> rbind(A,B) [,1] [,2] [,3] [1,] 1 2 3 [2,] 4 5 6	J> a=[1 2 3]; J> b=[4 5 6]; J> c=[a' b'] 3x2 Array{Int64,2}: 1 4 2 5 3 6 J> c=[a; b] 2x3 Array{Int64,2}: 1 2 3 4 5 6	**Stacking** **vectors and matrices**

BASIC MATRIX OPERATIONS

Matrix-scalar **operations**	M> A = [1 2 3; 4 5 6; 7 8 9] M> A * 2 ans = 2 4 6	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) P> A * 2 array([[2, 4, 6], [8, 10, 12],	R> A = matrix(1:9, nrow=3, byrow=T) R> A * 2 [,1] [,2] [,3] [1,] 2 4 6	J> A=[1 2 3; 4 5 6; 7 8 9]; # elementwise operator J> A .* 2	**Matrix-scalar** **operations**

	8 10 12 14 16 18 M> A + 2 M> A - 2 M> A / 2	[14, 16, 18]]) P> A + 2 P> A - 2 P> A / 2 # Note that NumPy was optimized for # in-place assignments # e.g., A += A instead of # A = A + A	[2,] 8 10 12 [3,] 14 16 18 R> A + 2 R> A - 2 R> A / 2	3x3 Array{Int64,2}: 2 4 6 8 10 12 14 16 18 J> A .+ 2; J> A .- 2; J> A ./ 2;	
Matrix-matrix multiplication	M> A = [1 2 3; 4 5 6; 7 8 9] M> A * A ans = 30 36 42 66 81 96 102 126 150	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) P> np.dot(A,A) # or A.dot(A) array([[30, 36, 42], [66, 81, 96], [102, 126, 150]])	R> A = matrix(1:9, nrow=3, byrow=T) R> A %*% A [,1] [,2] [,3] [1,] 30 36 42 [2,] 66 81 96 [3,] 102 126 150	J> A=[1 2 3; 4 5 6; 7 8 9]; J> A * A 3x3 Array{Int64,2}: 30 36 42 66 81 96 102 126 150	**Matrix-matrix multiplication**

Matrix-vector multiplication	M> A = [1 2 3; 4 5 6; 7 8 9]
	M> b = [1; 2; 3]
	M> A * b
	ans =
	14
	32
	50
		P> A = np.array([[1,2,3], [4,5,6], [7,8,9]])
		P> b = np.array([[1], [2], [3]])
		P> np.dot(A,b) # or A.dot(b)
		array([[14], [32], [50]])
			R> A = matrix(1:9, ncol=3)
			R> b = matrix(1:3, nrow=3)
			R> t(b %*% A)
			[,1]
			[1,] 14
			[2,] 32
			[3,] 50
				J> A=[1 2 3; 4 5 6; 7 8 9];
				J> b=[1; 2; 3];
				J> A*b
				3-element Array{Int64,1}:
				14
				32
				50
					Matrix-vector multiplication

Element-wise matrix-matrix operations	M> A = [1 2 3; 4 5 6; 7 8 9] M> A .* A ans = 1 4 9 16 25 36 49 64 81 M> A .+ A M> A .- A M> A ./ A	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) P> A * A array([[1, 4, 9], [16, 25, 36], [49, 64, 81]]) P> A + A P> A - A P> A / A # Note that NumPy was optimized for # in-place assignments # e.g., A += A instead of # A = A + A	R> A = matrix(1:9, nrow=3, byrow=T) R> A * A [,1] [,2] [,3] [1,] 1 4 9 [2,] 16 25 36 [3,] 49 64 81 R> A + A R> A - A R> A / A	J> A=[1 2 3; 4 5 6; 7 8 9]; J> A .* A 3x3 Array{Int64,2}: 1 4 9 16 25 36 49 64 81 J> A .+ A; J> A .- A; J> A ./ A;	**Element-wise matrix-matrix operations**
Matrix elements to power n	M> A = [1 2 3; 4 5 6; 7 8 9]	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]])	R> A = matrix(1:9, nrow=3, byrow=T)	J> A=[1 2 3; 4 5 6; 7 8 9];	**Matrix elements to power n**

(here: individual elements squared)	M> A.^2 ans = 1 4 9 16 25 36 49 64 81	P> np.power(A,2) array([[1, 4, 9], [16, 25, 36], [49, 64, 81]])	R> A ^ 2 [,1] [,2] [,3] [1,] 1 4 9 [2,] 16 25 36 [3,] 49 64 81	J> A .^ 2 3x3 Array{Int64,2}: 1 4 9 16 25 36 49 64 81	**(here: individual elements squared)**
Matrix to power n **(here: matrix-matrix multiplication with itself)**	M> A = [1 2 3; 4 5 6; 7 8 9] M> A ^ 2 ans = 30 36 42 66 81 96 102 126 150	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) P> np.linalg.matrix_power(A,2) array([[30, 36, 42], [66, 81, 96], [102, 126, 150]])	R> A = matrix(1:9, ncol=3) # requires the ‘expm’ package R> install.packages('expm') R> library(expm) R> A %^% 2 [,1] [,2] [,3] [1,] 30 66 102 [2,] 36 81 126 [3,] 42 96 150	J> A=[1 2 3; 4 5 6; 7 8 9]; J> A ^ 2 3x3 Array{Int64,2}: 30 36 42 66 81 96 102 126 150	**Matrix to power n** **(here: matrix-matrix multiplication with itself)**
Matrix transpose	M> A = [1 2 3; 4 5 6; 7 8 9] M> A' ans = 1 4 7 2 5 8 3 6 9	P> A = np.array([[1,2,3], [4,5,6], [7,8,9]]) P> A.T array([[1, 4, 7], [2, 5, 8], [3, 6, 9]])	R> A = matrix(1:9, nrow=3, byrow=T) R> t(A) [,1] [,2] [,3] [1,] 1 4 7 [2,] 2 5 8 [3,] 3 6 9	J> A=[1 2 3; 4 5 6; 7 8 9] 3x3 Array{Int64,2}: 1 2 3 4 5 6 7 8 9 J> A' 3x3 Array{Int64,2}: 1 4 7 2 5 8 3 6 9	**Matrix transpose**
Determinant of a matrix: **A -> |A|**	M> A = [6 1 1; 4 -2 5; 2 8 7] A = 6 1 1 4 -2 5 2 8 7 M> det(A) ans = -306	P> A = np.array([[6,1,1],[4,-2,5],[2,8,7]]) P> A array([[6, 1, 1], [4, -2, 5], [2, 8, 7]]) P> np.linalg.det(A) -306	R> A = matrix(c(6,1,1,4,-2,5,2,8,7), nrow=3, byrow=T) R> A [,1] [,2] [,3] [1,] 6 1 1 [2,] 4 -2 5 [3,] 2 8 7 R> det(A) [1] -306	J> A=[6 1 1; 4 -2 5; 2 8 7] 3x3 Array{Int64,2}: 6 1 1 4 -2 5 2 8 7 J> det(A) -306	**Determinant of a matrix:** **A -> |A|**
Inverse of a matrix	M> A = [4 7; 2 6] A = 4 7 2 6 M> A_inv = inv(A)	P> A = np.array([[4, 7], [2, 6]]) P> A array([[4, 7], [2, 6]])	R> A = matrix(c(4,7,2,6), nrow=2, byrow=T) R> A [,1] [,2] [1,] 4 7 [2,] 2 6	J> A=[4 7; 2 6] 2x2 Array{Int64,2}: 4 7 2 6 J> A_inv=inv(A)	**Inverse of a matrix**

	A_inv = 0.60000 -0.70000 -0.20000 0.40000	P> A_inverse = np.linalg.inv(A) P> A_inverse array([[0.6, -0.7], [-0.2, 0.4]])	R> solve(A) [,1] [,2] [1,] 0.6 -0.7 [2,] -0.2 0.4	2x2 Array{Float64,2}: 0.6 -0.7 -0.2 0.4	

ADVANCED MATRIX OPERATIONS

Calculating the covariance matrix of 3 random variables **(here: covariances of the means of x1, x2, and x3)**	M> x1 = [4.0000 4.2000 3.9000 4.3000 4.1000]' M> x2 = [2.0000 2.1000 2.0000 2.1000 2.2000]' M> x3 = [0.60000 0.59000 0.58000 0.62000 0.63000]' M> cov([x1,x2,x3]) ans = 2.5000e-02 7.5000e-03 1.7500e-03 7.5000e-03 7.0000e-03 1.3500e-03 1.7500e-03 1.3500e-03 4.3000e-04	P> x1 = np.array([4, 4.2, 3.9, 4.3, 4.1]) P> x2 = np.array([2, 2.1, 2, 2.1, 2.2]) P> x3 = np.array([0.6, 0.59, 0.58, 0.62, 0.63]) P> np.cov([x1, x2, x3]) Array([[0.025 , 0.0075 , 0.00175], [0.0075 , 0.007 , 0.00135], [0.00175, 0.00135, 0.00043]])	R> x1 = matrix(c(4, 4.2, 3.9, 4.3, 4.1), ncol=5) R> x2 = matrix(c(2, 2.1, 2, 2.1, 2.2), ncol=5) R> x3 = matrix(c(0.6, 0.59, 0.58, 0.62, 0.63), ncol=5) R> cov(matrix(c(x1, x2, x3), ncol=3)) [,1] [,2] [,3] [1,] 0.02500 0.00750 0.00175 [2,] 0.00750 0.00700 0.00135 [3,] 0.00175 0.00135 0.00043	J> x1=[4.0 4.2 3.9 4.3 4.1]'; J> x2=[2. 2.1 2. 2.1 2.2]'; J> x3=[0.6 .59 .58 .62 .63]'; J> cov([x1 x2 x3]) 3x3 Array{Float64,2}: 0.025 0.0075 0.00175 0.0075 0.007 0.00135 0.00175 0.00135 0.00043	**Calculating the covariance matrix of 3 random variables** **(here: covariances of the means of x1, x2, and x3)**
Calculating eigenvectors and eigenvalues	M> A = [3 1; 1 3] A = 3 1 1 3 M> [eig_vec,eig_val] = eig(A) eig_vec = -0.70711 0.70711 0.70711 0.70711 eig_val = Diagonal Matrix	P> A = np.array([[3, 1], [1, 3]]) P> A array([[3, 1], [1, 3]]) P> eig_val, eig_vec = np.linalg.eig(A) P> eig_val array([4., 2.])	R> A = matrix(c(3,1,1,3), ncol=2) R> A [,1] [,2] [1,] 3 1 [2,] 1 3 R> eigen(A) $values [1] 4 2	J> A=[3 1; 1 3] 2x2 Array{Int64,2}: 3 1 1 3 J> (eig_vec,eig_val)=eig(a) ([2.0,4.0], 2x2 Array{Float64,2}: -0.707107 0.707107 0.707107 0.707107)	**Calculating eigenvectors and eigenvalues**

	2 0 0 4	P> eig_vec Array([[0.70710678, -0.70710678], [0.70710678, 0.70710678]])	$vectors [,1] [,2] [1,] 0.7071068 -0.7071068 [2,] 0.7071068 0.7071068		
Generating a Gaussian dataset: **creating random vectors from the multivariate normal distribution given mean and covariance matrix** **(here: 5 random vectors with mean 0, covariance = 0, variance = 2)**	% requires statistics toolbox package % how to install and load it in Octave: % download the package from: % http://octave.sourceforge.net/packages.php % pkg install % ~/Desktop/io-2.0.2.tar.gz % pkg install % ~/Desktop/statistics-1.2.3.tar.gz M> pkg load statistics M> mean = [0 0] M> cov = [2 0; 0 2] cov = 2 0 0 2 M> mvnrnd(mean,cov,5) 2.480150 -0.559906 -2.933047 0.560212 0.098206 3.055316 -0.985215 -0.990936 1.122528 0.686977	P> mean = np.array([0,0]) P> cov = np.array([[2,0],[0,2]]) P> np.random.multivariate_normal(mean, cov, 5) Array([[1.55432624, -1.17972629], [-2.01185294, 1.96081908], [-2.11810813, 1.45784216], [-2.93207591, -0.07369322], [-1.37031244, -1.18408792]])	# requires the 'mass' package R> install.packages('MASS') R> library(MASS) R> mvrnorm(n=10, mean, cov) [,1] [,2] [1,] -0.8407830 -0.1882706 [2,] 0.8496822 -0.7889329 [3,] -0.1564171 0.8422177 [4,] -0.6288779 1.0618688 [5,] -0.5103879 0.1303697 [6,] 0.8413189 -0.1623758 [7,] -1.0495466 -0.4161082 [8,] -1.3236339 0.7755572 [9,] 0.2771013 1.4900494 [10,] -1.3536268 0.2338913	# requires the Distributions package from https://github.com/JuliaStats/Distributions.jl J> using Distributions J> mean=[0., 0.] 2-element Array{Float64,1}: 0 0 J> cov=[2. 0.; 0. 2.] 2x2 Array{Float64,2}: 2.0 0.0 0.0 2.0 J> rand(MvNormal(mean, cov), 5) 2x5 Array{Float64,2}: -0.527634 0.370725 -0.761928 -3.91747 1.47516 -0.448821 2.21904 2.24561 0.692063 0.390495	**Generating a Gaussian dataset:** **creating random vectors from the multivariate normal distribution given mean and covariance matrix** **(here: 5 random vectors with mean 0, covariance = 0, variance = 2)**