Basic Matrix Algebra

AUTHOR
Sujit Sandipan Chaugule

AFFILIATION
Department of Pharmaceutical Sciences and Technology, Institute of Chemical Technology, Mumbai

Introduction to Matrices

The theory of matrices was developed by the mathematician **Arthur Cayley**. Matrices are useful for expressing numerical information in compact form. They are effectively used to represent different operators. Hence, matrices are essential in fields such as Economics, Statistics, and Computer Science.

Definition

A matrix is a rectangular array of elements (numbers, symbols, or expressions) arranged in m rows and n columns. It is denoted by a capital letter such as A, and written as

$$A = [a_{ij}]_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},$$

where a_{ij} denotes the element in the i^{th} row and j^{th} column.

The pair (m, n) is called the *order* (or size) of the matrix. The collection of all such matrices with real entries is denoted by

$$\mathbb{R}^{m\times n}.$$

Creating Matrices from the Entries

Matrices in Julia are represented as two-dimensional arrays. These are constructed by giving the elements in each row separated by spaces, with the rows separated by semicolons. For example, the 3×4 matrix

$$A = \begin{bmatrix} 1 & 2.5 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

is created in Julia as

```
A = [1  2.5  3  4;
     5  6  7  8;
     9 10 11 12]
```

```
3×4 Matrix{Float64}:
 1.0   2.5   3.0   4.0
 5.0   6.0   7.0   8.0
 9.0  10.0  11.0  12.0
```

The Julia function `size(A)` returns the size of a matrix as a tuple. It can also be called as `size(A, 1)` or `size(A, 2)` to obtain only the number of rows or columns, respectively. As an example, we create a function that determines whether a matrix is *tall* (i.e., it has more rows than columns).

```
m, n = size(A)
```

```
(3, 4)
```

```
size(A, 1)  # no of rows
```

```
3
```

```
size(A, 2) # no of columns
```

```
4
```

```
# creating function
tall(X) = size(X,1)>size(X,2);
tall(A)
```

```
false
```

Indexing Entries

The (i, j) entry of a matrix A is accessed in Julia using `A[i, j]`. This returns the element in the i^{th} row and j^{th} column. We can also assign a new value to any entry of the matrix using the same indexing notation.

```
A[2, 3]   # Get 2,3 entry of A
```

```
7.0
```

```
A[1,3] = 7.5; # Set 1,3 entry of A to 7.5
A
```

```
3×4 Matrix{Float64}:
 1.0   2.5   7.5   4.0
 5.0   6.0   7.0   8.0
 9.0  10.0  11.0  12.0
```

Equality of Matrices

In Julia, the expression `A == B` determines whether two matrices A and B are equal (that is, whether all corresponding entries are the same). The expression `A .== B` performs element-wise comparison and creates a matrix of Boolean values, depending on whether the corresponding entries of A and B are equal. Finally, the expression `sum(A .== B)` gives the total number of entries in which A and B are equal.

```
A = [2 0 1;5 7 9;10 3 4]
B = copy(A)
B[2,3] = -1;
B
```

```
3×3 Matrix{Int64}:
  2  0   1
  5  7  -1
 10  3   4
```

```
A == B   # checking the whether it is equal or not
```

```
false
```

```
A .== B
```

```
3×3 BitMatrix:
 1  1  1
 1  1  0
 1  1  1
```

```
sum(A .== B)
```

```
8
```

Zero and identity matrices

Zero matrices. A zero matrix of size $m \times n$ is created using $\text{zeros}(m, n)$.

```
zeros(2, 2)
```

```
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
```

Identity matrices in Julia can be created in several ways. One simple approach is to start with a zero matrix and then set the diagonal entries to one. The `LinearAlgebra` package also provides convenient tools for creating identity matrices. In particular, `I` represents a special identity matrix object. You can use `Matrix(I, n, n)` to create an $n \times n$ identity matrix. Multiplying by `1.0` converts the Boolean entries to numerical (floating-point) values.

```julia
using LinearAlgebra
# Create a 3 × 3 identity matrix
A = Matrix(I, 3, 3)
```

```
3×3 Matrix{Bool}:
 1  0  0
 0  1  0
 0  0  1
```

Diagonal Matrices

In standard mathematical notation, $\text{diag}(1, 2, 3)$ denotes a 3×3 diagonal matrix whose diagonal entries are 1, 2, and 3, with all off-diagonal elements equal to zero. In Julia, diagonal matrices can be created using the `diagm` function from the `LinearAlgebra` package. To construct a diagonal matrix whose diagonal entries are stored in a vector `s`, we use the syntax `diagm(0 => s)`, where `s` contains the desired diagonal elements. Note that the diagonal entries must be provided as a vector.

```julia
using LinearAlgebra
# Diagonal entries
s = [1, 2, 3]
# Create diagonal matrix
D = diagm(0 => s)
```

```
3×3 Matrix{Int64}:
 1  0  0
 0  2  0
 0  0  3
```

Random Matrices

In Julia, a random $m \times n$ matrix with entries uniformly distributed between 0 and 1 can be generated using `rand(m, n)`. If instead you want the entries to follow a standard normal distribution (mean 0 and variance 1), you can use `randn(m, n)` to create the matrix.

```julia
randn(2, 2)
```

```
2×2 Matrix{Float64}:
 0.0675914  -0.65414
 1.83786    -0.799479
```

```julia
randn(5, 5)
```

```
5×5 Matrix{Float64}:
  0.581156   0.472274   0.633539   1.5139     1.46843
  0.735466  -1.77101    2.27503   -0.201555  -0.627625
 -0.479541   0.109854  -1.90121   -0.615615  -0.813699
```

```
-0.564576   0.482824  -1.19463    0.201376  -2.76755
 0.734114   1.70943    1.40719   -0.819639   0.499314
```

Transpose of a Matrix (in Julia)

The transpose of a matrix A, denoted by A^T, is obtained by interchanging its rows and columns. In Julia, the `transpose()` function is provided by the `LinearAlgebra` package. The transpose can be computed either using `transpose(A)`.

```
using LinearAlgebra

A = [1 2 3; 4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
# Transpose using transpose()
T1 = transpose(A)
```

```
3×2 transpose(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6
```

Addition, Subtraction, and Scalar Multiplication

In Julia, matrix addition, matrix subtraction, and scalar–matrix multiplication follow the standard rules of mathematical notation. That is, matrices of the same size can be added or subtracted element-wise, and a matrix can be multiplied by a scalar by multiplying each of its entries by that scalar.

```
# Example matrices
A = [1 2; 3 4]

B = [5 6; 7 8]

# Matrix addition
A + B
```

```
2×2 Matrix{Int64}:
  6   8
 10  12
```

```
# Matrix subtraction
A - B
```

```
2×2 Matrix{Int64}:
 -4  -4
 -4  -4
```

```
# Scalar multiplication
2 * A
```

```
2×2 Matrix{Int64}:
 2  4
 6  8
```

Matrix Multiplication (in Julia)

Matrix multiplication follows the standard mathematical definition. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then their product $C = AB$ is an $m \times p$ matrix.In Julia, matrix multiplication is performed using the `*` operator.

```
A = [1 2 3; 4 5 6]
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
B = [7  8; 9 10; 11 12]
```

```
3×2 Matrix{Int64}:
  7   8
  9  10
 11  12
```

```
# Matrix multiplication
C = A * B
```

```
2×2 Matrix{Int64}:
  58   64
 139  154
```

Element-wise Operations

The syntax for element-wise vector operations extends naturally to matrices. In Julia, placing a period `.` before a binary operator changes its interpretation to element-wise. For example,if A and B are matrices of the same size,then `C = A .* B` creates a matrix C of the same size whose entries are the products of the corresponding entries of A and B. Similar notation applies to other operations such as addition, subtraction, and division.

```
A = [1 2; 3 4]
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
B = [5 6; 7 8]
```

```
2×2 Matrix{Int64}:
 5  6
 7  8
```

```
# Element-wise multiplication
C = A .* B
```

```
2×2 Matrix{Int64}:
  5  12
 21  32
```

Inverse of Matrix

Inverse of a Matrix

Let A be a square matrix of order $n \times n$. The matrix A^{-1} is called the **inverse of** A if

$$AA^{-1} = A^{-1}A = I_n$$

where I_n is the identity matrix of order n.

Condition for Existence

A matrix A is said to be **invertible** (or non-singular) if and only if

$$\det(A) \neq 0$$

If

$$\det(A) = 0,$$

then A is called a **singular matrix**, and its inverse does not exist.

In Julia, the inverse of a matrix can be computed using the `inv()` function from the `LinearAlgebra` package.

```
using LinearAlgebra
```

```
A = [2 4 -24; 4 -1 3; -2 3 4]
```

```
3×3 Matrix{Int64}:
  2   4  -24
```

```
 4  -1    3
-2   3    4
```

```
inv(A)   # inverse of matrix
```

```
3×3 Matrix{Float64}:
  0.0367232  0.248588  0.0338983
  0.0621469  0.112994  0.288136
 -0.0282486  0.039548  0.0508475
```

```
A*inv(A)   # A*inv(A) = I
```

```
3×3 Matrix{Float64}:
  1.0           0.0          -5.55112e-17
 -3.46945e-18   1.0           6.93889e-18
  2.77556e-17  -5.55112e-17   1.0
```

Solving System of Linear Equations

A system of linear equations consists of multiple linear equations involving the same set of variables. A general system of m equations in n unknowns can be written as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

Matrix Representation

The above system can be written compactly in matrix form as

$$Ax = b,$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

Existence of Solution

If A is a square matrix ($n \times n$) and

$$\det(A) \neq 0,$$

then the system has a **unique solution**, given by

$$x = A^{-1}b.$$

Consider the system:

$$
\begin{aligned}
2x + y - z &= 8 \\
-3x - y + 2z &= -11 \\
-2x + y + 2z &= -3
\end{aligned}
$$

Matrix Representation

The system can be written in matrix form as

$$Ax = b,$$

where

$$A = \begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix}, \quad x = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad b = \begin{bmatrix} 8 \\ -11 \\ -3 \end{bmatrix}.$$

```
# Coefficient matrix
A = [2  1  -1;
    -3 -1   2;
    -2  1   2]
```

```
3×3 Matrix{Int64}:
  2   1  -1
 -3  -1   2
 -2   1   2
```

```
det(A)
```

```
-0.9999999999999998
```

```
# Right-hand side vector
b = [8, -11, -3];
```

```
x = inv(A)*b
```

```
3-element Vector{Float64}:
  2.0000000000000013
  3.000000000000002
 -0.9999999999999987
```

```
A*x   # you can verify
```

```
3-element Vector{Float64}:
   8.000000000000004
 -11.000000000000004
  -2.999999999999998
```

Eigen Values & Eigen-vectors

Let A be a square matrix of order $n \times n$. A non-zero vector $x \in \mathbb{R}^n$ is called an **eigenvector** of A if there exists a scalar λ such that

$$Ax = \lambda x.$$

The scalar λ is called the **eigenvalue** corresponding to the eigenvector x. The equation

$$Ax = \lambda x$$

means that multiplication of the vector x by the matrix A changes only its magnitude (scaled by λ), but not its direction.

From

$$Ax = \lambda x,$$

we write

$$Ax - \lambda x = 0.$$

Factoring,

$$(A - \lambda I)x = 0,$$

where I is the identity matrix. For a **non-trivial solution** ($x \neq 0$) to exist,

$$\det(A - \lambda I) = 0.$$

This equation is called the **characteristic equation** of A. The polynomial

$$\det(A - \lambda I)$$

is called the **characteristic polynomial**.

Finding Eigenvalues

Eigenvalues are obtained by solving

$$\det(A - \lambda I) = 0.$$

For an $n \times n$ matrix, this gives a polynomial equation of degree n, which yields n eigenvalues (including multiplicities).

Finding Eigenvectors

For each eigenvalue λ, substitute it into

$$(A - \lambda I)x = 0,$$

and solve the resulting homogeneous system to obtain the eigenvectors.

```
using LinearAlgebra

# Define a 3 × 3 matrix
```

```
A = [4  1  -2;
     1  3   0;
    -2  0   5]
```

```
3×3 Matrix{Int64}:
  4  1  -2
  1  3   0
 -2  0   5
```

```
D = eigen(A)
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 1.8548973087995777
 3.4760236029181355
 6.669079088282288
vectors:
3×3 Matrix{Float64}:
  0.679313  0.374362  -0.631179
 -0.593233  0.786436  -0.172027
  0.431981  0.491296   0.75632
```

```
tr(A)   # trace of matrix   = sum of diagonal elements
```

```
12
```

```
sum(D.values)   # trace of matrix = sum of eigen values
```

```
12.000000000000002
```