Vectors_Matrices

HDS

2024-06-21

Vectors and Matrices

A vector is a set of N values arranged in a sequence

The vector many people may be familiar with is cartesian coordinates, in which a pair of values (x,y) is used to indicate a position. This is a two dimensional vector, we use them all the time in creating graphs. The values x and y are distances, in feet or meters or some other measure of distance.

If we wanted to work with three dimensions, we would need a height z as well, giving us a 3 dimensional vector (x,y,z)

A set of coordinates such as (3,4,1) means x=3, y=4,z=1, so vectors are a compact way of writing down this information.

Latitude and longitude form another 2 dimensional vector, (Lat,Long) but these are actually angles, not distances, since they indicate locations on the surface of the earth, which is roughly spherical.

Vectors with more dimensions than 2 or 3 are possible. They are hard for us to visualize, but the mathematics of them is straightforward.

Basic ideas about vectors are covered in an undergraduate Physics 1 or Calculus 3 course. More advanced vector ideas are covered in an undergraduate course called Linear Algebra.

In Data Science, vectors are often used to represent or depict data for a given observation, event or individual.

Typically, a vector has N entries, which are all variables of the same type

The entries are in a specific order that is kept constant.

Suppose we are tracking average monthly consumer spending in several categories

Housing, Energy, Transportation, Food, Entertainment, Medical, Insurance

so there are 7 categories here, so we could represent an individual Jane's spending as a vector

We can create a vector in R to represent this

```
Jane_spending=c(1800,400,550,900,200,525,200)
Jane_spending
```

```
## [1] 1800  400  550  900  200  525  200
```

We could calculate Jane's yearly totals in each category, simply multiplying the vector by 12

```
Jane_spending*12
```

```
## [1] 21600  4800  6600 10800  2400  6300  2400
```

There are a number of simple operations we can carry out on vectors

finding the length, which is the number of entries in the vector

```
length(Jane_spending)
```

```
## [1] 7
```

Finding the min, max, median, sum, standard deviation (sd) etc

```
min(Jane_spending)
```

```
## [1] 200
```

```
median(Jane_spending)
```

```
## [1] 525
```

```
sd(Jane_spending)
```

```
## [1] 559.4693
```

We can also calculate a product of all values in the vector

This doesn’t make sense for Jane’s budget, but in other contexts it might be handy

```
prod(Jane_spending)
```

```
## [1] 7.4844e+18
```

Actions: Create cells that

a. Calculate what Jane spends per week

```
Jane_spending/7
```

```
## [1] 257.14286  57.14286  78.57143 128.57143  28.57143  75.00000  28.57143
```

b. What is Jane’s total yearly spending? Hint: use the sum() function

```
sum(Jane_spending*12)
```

```
## [1] 54900
```

c. Set up a monthly spending vector for some other person # Vectorized calculations

```
mel_spending=c(1900,300,600,500,210,400,250)
mel_spending
```

```
## [1] 1900  300  600  500  210  400  250
```

We can carry out operations on vectors that are

a.) carried out on each value in the vector, these are called element-wise operations. Our calculation of the yearly spending totals for Jane above were element-wise operations

b.) there are vector operations, which act on the value of a vector as a whole this includes operations such as calculating the magnitude or size of a vector, or things like the dot product or cross product.

If you talk about “multiplying two vectors” that can have multiple meanings

1.) element-wise multiplication, of corresponding values in the vector

```
#elementwise multiplication of vectors

x=c(1,2,3)
y=c(5,10,15)
x*y
```

```
## [1]  5 20 45
```

2.) the dot product or scalar product or inner product (synonyms)

this is the sum of the values obtained in the element-wise multiplication above

the dot product is a number of other things as well I won’t take time to talk about today

to compute a dot product, the two vectors must have the same length

```
# dot product of vectors x and y
x%*%y
```

```
##      [,1]
## [1,]   70
```

Notice that * gave us an element-wise multiplication, %*% gave use the dot product

3.) the last form of vector multiplication is called a cross product, or vector product.

I am discussing this here for two reasons:

a.) You need to be aware of the different possible types of vector multiplication, even if you never use most of them.

b.) For some areas in data science, knowing linear algebra is pretty important. It is an underpinning of a lot of statistics, and is used as part of the coding of many algorithms.

```
I want to be sure that those of you who have had a course in linear
algebra see how to do linear algebra in R.

There are many career paths in data science, you don't have to know
linear algebra to work effectively in many roles, but it is a really useful
area of knowledge.

Remember data science is a mix of mathematics, statistics, computer
science and business acumen.   People who excel in all of these
are called "Unicorns" for good reason.   Most of us are good in one
area and ideally passable in the other two.  We rely on our team members to
cover the areas we are weak in, and contribute where we are strong.

Not to discourage you from aspiring to be a Unicorn, just to say you don't
have to be one.
```

To compute a cross product, I am doing a matrix multiplication %*% of the vector x by the transpose of the vector y, written as t(y)

we will talk briefly again later about matrix multiplication

```
x%*%t(y)
```

```
##      [,1] [,2] [,3]
## [1,]    5   10   15
## [2,]   10   20   30
## [3,]   15   30   45
```

Just what is this transpose thing anyway?

```
y
```

```
## [1]  5 10 15
```

```
z=t(y)
str(y)
```

```
##  num [1:3] 5 10 15
```

```
str(z)
```

```
##  num [1, 1:3] 5 10 15
```

Notice that y is a vector, a list of 3 values. This is sometimes called a column vector, all the data is in a single column.

z is something a bit different. It has two indices, not 1.

z is a 1 row by 3 column matrix.

A row vector like y could also be called a 3 x 1 matrix, meaning it has three rows and only one column

Vectors are just a special case of a matrix, we could call them N x 1 matrices

The notation %% *really means “matrix multiplication”,* means element-wise operations

There is also a form of matrix division %/% as well as element-wise multiplication /

Vectorized Calculations

In languages such as R and Python, we want to carry out operations on the whole vector at the same time, this is called vectorization.

Vectorized calculations are actually done with C subroutines, making them much faster. When you are working on cloud-based systems or cluster systems, vector operations are much faster than any other approach

Example:

Suppose we want to graph a function, y= x^2 -3x +4 from the range -10 to 10

We can do this using vector commands in r, instead of using a loop

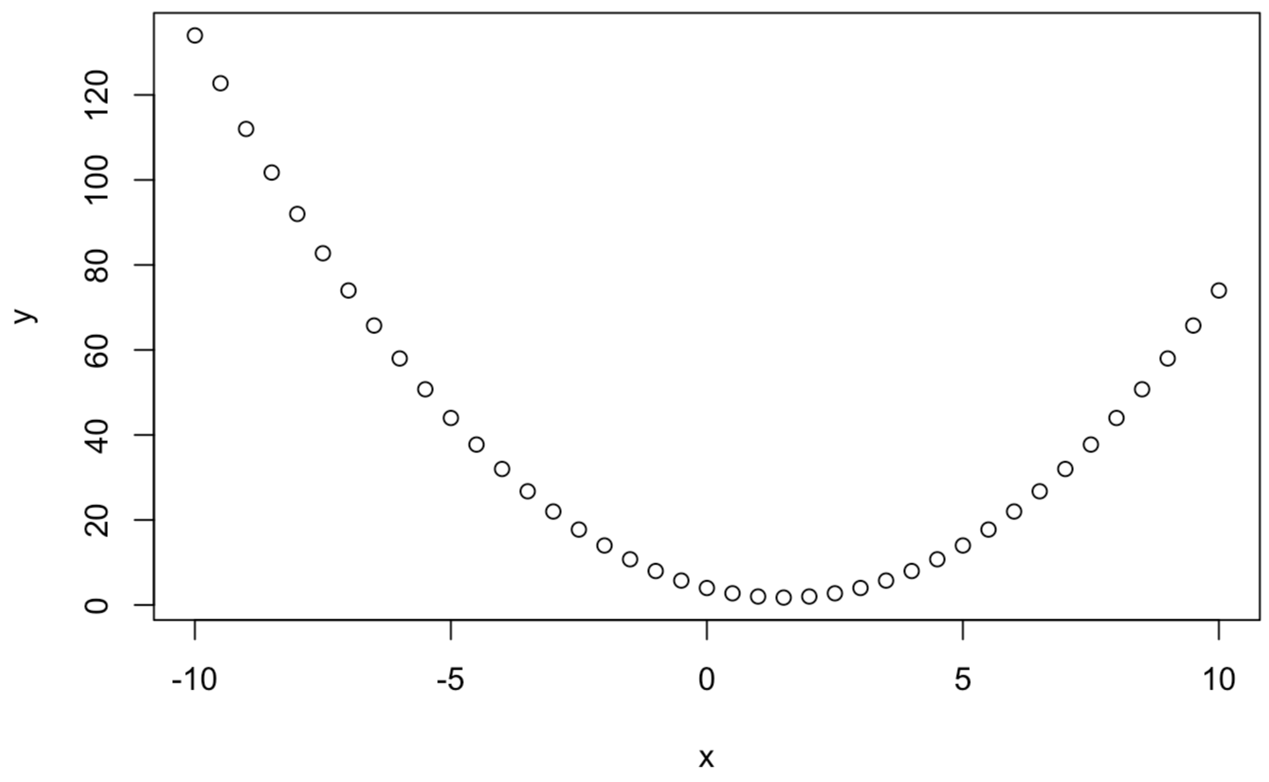
```
#set up x,  we could alter the by to get finer steps

x=seq(from=-10,to=10,by=0.5)

#use a vector calculation to get y
# all the values in y are calculated at once

y= x**2 -3*x+4

plot(x,y)
```



Actions/Questions

Use code to figure out

a.) How long is x? How long is y?

```
length(x)
```

```
## [1] 41
```

```
length(y)
```

```
## [1] 41
```

b.) What is the minimum y? what location in the y array does it occur at?

```
min(y)
```

```
## [1] 1.75
```

```
y
```

```
##  [1] 134.00 122.75 112.00 101.75  92.00  82.75  74.00  65.75  58.00  50.75
## [11]  44.00  37.75  32.00  26.75  22.00  17.75  14.00  10.75   8.00   5.75
## [21]   4.00   2.75   2.00   1.75   2.00   2.75   4.00   5.75   8.00  10.75
## [31]  14.00  17.75  22.00  26.75  32.00  37.75  44.00  50.75  58.00  65.75
## [41]  74.00
```

```
#min(y) occurs in the [24] location in the y array.
```

c.) Which x value produces the minimum y?

```
x
```

```
##  [1] -10.0  -9.5  -9.0  -8.5  -8.0  -7.5  -7.0  -6.5  -6.0  -5.5  -5.0  -4.5
## [13]  -4.0  -3.5  -3.0  -2.5  -2.0  -1.5  -1.0  -0.5   0.0   0.5   1.0   1.5
## [25]   2.0   2.5   3.0   3.5   4.0   4.5   5.0   5.5   6.0   6.5   7.0   7.5
## [37]   8.0   8.5   9.0   9.5  10.0
```

```
y
```

```
##  [1] 134.00 122.75 112.00 101.75  92.00  82.75  74.00  65.75  58.00  50.75
## [11]  44.00  37.75  32.00  26.75  22.00  17.75  14.00  10.75   8.00   5.75
## [21]   4.00   2.75   2.00   1.75   2.00   2.75   4.00   5.75   8.00  10.75
## [31]  14.00  17.75  22.00  26.75  32.00  37.75  44.00  50.75  58.00  65.75
## [41]  74.00
```

```
#the [24] of x, 1.5 produces the minimum y
```

Recycling

R has a feature not seen in other languages

If we carry out a vector operation using two vectors of unequal length, so that they don't have the same number of elements, most languages will generate an error

R doesn't, it just recycles the values in the shorter vector

Here's an example

```
x=1:10
y=c(0,5,10)
x+y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  1  7 13  4 10 16  7 13 19 10
```

R does at least give us a warning, then it completes the operation by cycling through the values in y in order

This is a feature of R I really don’t like, it is not present in most other languages.

Indexing

We can index, or read values in an array using either 1 or more integer values, which are the ordinal locations of the values in the array

```
#set up array

a=c(1,3,5,7,9,11,13)

#select one entry in a,  the fifth in this case

a[5]
```

```
## [1] 9
```

```
# Note that R starts indices at 1, so a[1] is the first item in a
#
# Python and most other languages start indexes at 0, meaning how far from the
#start of the array or vector,  in python a[1] is the second item in the list
# this is one difference between R and Python that will trip you up all the
# time

# here is a slice of a that selects the second and fourth items

a[c(2,4)]
```

```
## [1] 3 7
```

We can also slice or select using an array of logical (TRUE/FALSE) values that is the same length as the array

```
slice_values=c(FALSE, FALSE,TRUE,FALSE,FALSE, TRUE,FALSE)
a[slice_values]
```

```
## [1]  5 11
```

Writing out lists of TRUE and FALSE values seems like an awkward way to do slice.

We don’t write out these lists of logical values, we use a comparison test to do it

Comparisons are of the form of a comparison using (>,<,==,!-,>=, <=, etc)

```
 a[a>6]
```

```
## [1]  7  9 11 13
```

Set Operations

We can think of vectors are representing sets of objects

```
set_a=c(1,2,4,5,6,7,8,9)
set_b=c(1,3,5,7,9)
set_c=c(2,4,6,8)
```

Then we can look at some set operations

Union, the set of all values in either set

```
union(set_b,set_c)
```

```
## [1] 1 3 5 7 9 2 4 6 8
```

We can look at the intersection of two sets, the set of things in both

```
intersect(set_b,set_c)
```

```
## numeric(0)
```

```
intersect(set_a,set_b)
```

```
## [1] 1 5 7 9
```

We can ask if an object is in a set

```
1 %in% set_b
```

```
## [1] TRUE
```

```
1 %in% set_c
```

```
## [1] FALSE
```

This is handy for checking to see if a value is one of a set of allowed values or not

```
species_list=c('cat', 'dog','mouse')
'Cat' %in% species_list
```

```
## [1] FALSE
```

```
'cat' %in% species_list
```

```
## [1] TRUE
```

There is a way to compute differences in sets

```
setdiff(set_a,set_b)
```

```
## [1] 2 4 6 8
```

```
setdiff(set_b,set_a)
```

```
## [1] 3
```

```
?setdiff
```

Why did changing the order matter? #it looked for the rows in set_b that aren't in set_y. Look up setdiff in the help function

Matrices

Matrices are rectangular grids of data. The data always has to be the same type, either numeric, integer or complex.

Most matrices have rows and columns, so they are two dimensional structures.

By convenient, we say there are m rows and n columns.

We also refer to the row first, then the column

We will often be loading matrices from a text file, or an Excel file or some other source, but we can create them in R

We send in an array, and then tell R how to convert it to a matrix

```
a=matrix(1:12,nrow=4, byrow=FALSE)
a
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

Notice that the array filled in the matrix by columns first

We can change this in fill by the row

```
b=matrix(1:12,nrow=4, byrow=TRUE)
b
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

Both a and b are 4x3 matrices, meaning they have 4 rows and 3 columns

We always state the row first, then the column

We can use the dim() function to get the size of matric

```
dim(b)
```

```
## [1] 4 3
```

```
#number of rows
dim(b)[1]
```

```
## [1] 4
```

```
#number of columns
dim(b)[2]
```

```
## [1] 3
```

Indexing a Matrix

We have to supply both a row number and a column number

Before you run the cell below, predict what the value will be

```
b[2,2]
```

```
## [1] 5
```

We can also index all contents of a single row, or a single column

```
b[3,]
```

```
## [1] 7 8 9
```

```
b[,2]
```

```
## [1]  2  5  8 11
```

If we use negative index values it removes that particular row or column and returns everything else, handy for editing

```
b[,-2]
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    4    6
## [3,]    7    9
## [4,]   10   12
```

Notice that the indexing of a vector and of a matrix work the same way, a matrix just has 2 indices.

Special matrics

There are some special matrices that we use from time to time

This is an identity matrix

```
diag(4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    1    0    0
## [3,]    0    0    1    0
## [4,]    0    0    0    1
```

Here are matrices of all 1s or all 0s

```
matrix(0, nrow=5,ncol=5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

```
matrix(1,nrow=3,ncol=3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

We can also ask for a random matrix

```
d=matrix(runif(25),nrow=5,ncol=5)
d
```

```
##            [,1]       [,2]      [,3]      [,4]      [,5]
## [1,] 0.48276555 0.17368110 0.5124844 0.7842926 0.3467866
## [2,] 0.28366190 0.37462035 0.3813397 0.2418550 0.3945632
## [3,] 0.15024510 0.74000536 0.9393573 0.1071526 0.4430885
## [4,] 0.77900099 0.04927843 0.2411782 0.2241746 0.9813838
## [5,] 0.08242964 0.40870795 0.1569663 0.7497817 0.7457589
```

Adding row and column names

we can add row and column names, vectors with the same number of entries of rows and columns respectively

Note, we do need the assignment operator here, not the equals sign

```
# bonus if you can tell me what pop culture reference 1,2,3 Marlena's is from

rownames(d)=c("one","two","three","marlenas","four")

colnames(d) =c("a","b","c","d","e")

d
```

```
##                   a          b         c         d         e
## one      0.48276555 0.17368110 0.5124844 0.7842926 0.3467866
## two      0.28366190 0.37462035 0.3813397 0.2418550 0.3945632
## three    0.15024510 0.74000536 0.9393573 0.1071526 0.4430885
## marlenas 0.77900099 0.04927843 0.2411782 0.2241746 0.9813838
## four     0.08242964 0.40870795 0.1569663 0.7497817 0.7457589
```

indexing by name

When we have names assigned to rows and columns, we can index using the names or numbers

In many situations, we will want the row names to be identifiers or names

and the column names to be variable names

```
d['marlenas','a']
```

```
## [1] 0.779001
```

For linear algebra fans

This is just a quick look at how R can carry out some common linear algebra calculations for you-

Here is the determinant

```
det(d)
```

```
## [1] -0.05952402
```

Here is the inverse

We need the Mass library to calculate this

This is a generalized inverse calculation

```
require('MASS')
```

```
## Loading required package: MASS
```

```
d_inv=ginv(d)
d_inv
```

```
##             [,1]       [,2]       [,3]       [,4]       [,5]
## [1,]  0.05773464  5.4775939 -1.9524073 -0.2130224 -1.4845718
## [2,] -1.37054634  7.4283893 -1.8467493 -1.4720482 -0.2584809
## [3,]  1.19151279 -5.0901214  2.3296631  0.6806742 -0.1408976
## [4,]  0.98302215 -0.1738508 -0.4278578 -0.5247021  0.5795562
## [5,] -0.49437595 -3.4303715  1.1677214  1.2145564  1.0936395
```

Checking on it

d times it’s inverse should be an identity matrix

```
d%*%d_inv
```

```
##                   [,1]          [,2]          [,3]         [,4]          [,5]
## one       1.000000e+00 -3.774758e-15  6.106227e-16 1.665335e-16 -1.498801e-15
## two      -4.440892e-16  1.000000e+00  7.216450e-16 4.996004e-16  6.661338e-16
## three    -1.526557e-15 -1.554312e-15  1.000000e+00 3.330669e-16  1.276756e-15
## marlenas -3.830269e-15 -1.776357e-15  1.332268e-15 1.000000e+00  3.552714e-15
## four      3.330669e-15  8.881784e-16 -7.771561e-16 0.000000e+00  1.000000e+00
```

Notice the very small rounding error that is present

Here is an eigenvalue calculation

```
my_eigen=eigen(d)
```

```
str(my_eigen)
```

```
## List of 2
##  $ values : num [1:5] 2.163 0.811 -0.699 0.355 0.137
##  $ vectors: num [1:5, 1:5] -0.493 -0.339 -0.466 -0.485 -0.435 ...
##  - attr(*, "class")= chr "eigen"
```