Everybody welcome. This is an overview of

basic matrix manipulations in R. This is a very brief demonstration,

mainly to show you how to actually construct these matrices in R.

I'll just go through each of these different ways of doing it, one at a time.

Diag is pretty straightforward, constructs a diagonal matrix.

Now, it actually has two contradictory ways to interface with it.

For example, if I type in diag 4, that's a 4 by 4 diagonal matrix.

But if for example, I type 1:5

means a vector, 1-5, so diag 1-4 that's a vector,

so it'll give me a matrix whose diagonal is 1, 2, 3, and 4.

Now, that doesn't mean that R can be a bit confused,

sometimes you confuse, for example,

sometimes you don't know from the start whether you have a single number.

Like maybe you want a diagonal matrix that's one by one.

For that purpose, what you can do is you can type,

now what that did is I gave it a vector 1:2.

I told it to create a matrix four by four.

It did so by repeating that, recycling that vector 1:2.

Now, more generally,

you can construct the matrix just by giving it the numbers.

Here, that's a four by four matrix of ones.

You can use some arbitrary number here four by three matrix.

I can also, for example, pass this vector.

Now here we need to talk a little bit about how to construct a general matrix.

We might do that by two by two matrix with values.

Let's say 2, 3, 5, 5, 7, the first row prime numbers.

What that'll do is it'll fill the columns first.

You can see that here you have two, then three, then five,

then seven in the second column.

Denote that parentheses are important.

This C thing is important here because this won't work.

Moving on, you can easily construct a column vector of ones

or a row vector of ones.

You can bind together multiple vectors or matrices. For example,

we have three vectors, 1:4, 2:5, and 3:6,

and we end up with those as columns of a new matrix.

We could also use rbind for rows like this.

Note that you can also cbind rbind matrices.

For example, here I have two matrices.

Diagonal matrix with ones on the diagonal,

that's two by two, and another matrix with 1,

2 on the diagonal and we can bind them together.

How do we operate matrices?

Well, arithmetic operations, and this can be a little bit confusing,

but arithmetic operations all operate element-wise.

When I type how about,

let's have a two by two matrix with numbers 1-4.

Then I have a matrix of ones, that's also a two by two.

Each element gets multiplied by one and each element stays the same,

or doubled in this case,

which by the way is the same as just multiplying it by two.

That's different from matrix multiplication.

For example, if we wanted to multiply it by a matrix of

ones using the matrix multiplication operation.

We'll need to use this percent star percent,

and this is what we get.

Note that, for example,

if you tried to multiply two matrices,

ordinary way, it'll give you non-convertible arrays.

Same thing if you tried to perform matrix multiplication.

Although here this will actually work because they are conformable for multiplication,

but for example, these are not.

Matrix functions they're pretty straightforward.

Dim just gives you a vector, so dim just reconstructs

the dimensions of that matrix.

Rows and columns gives you one number, t gets you transpose.

C will turn it into a dimensional vector.

Now because R internally stores these matrix elements in columns,

this is equivalent to stacking the columns.

Next item is, let's say we have again matrix 1:4.

Now, if we tried to do the obvious thing,

if you wanted inverse and we take it to the minus first power,

actually that will do element-wise inverse of the matrix.

That's not what we want. To get the matrix inverse,

we have to use solve.

If that's an inverse,

we can check it very easily by

typing and by multiplying it by the original matrix,

and getting an identity matrix.

Crossprod is the best convenience function,

a lot of the time we end up using it for a vector.

Let's say we have a vector of 1-4,

and that's 30.

Why is that 30?

Well, it's because essentially we take each element of

1-4 and multiply it by 1-4 and sum the result.

Cholesky decomposition is something that was also introduced in the notes.

We can get a Cholesky decomposition of a symmetric matrix,

so let's construct one.

Cholesky decomposition is a little bit complicated,

but basically if you have a positive symmetric positive definite matrix,

you can essentially take a square root.

Basically, let's say you have a matrix, call it x,

and we'll define it as a matrix,

which is, it's got to be symmetric and positive definite.

I'm just going to pick some arbitrary numbers here.

This is a two-by-two matrix, it's symmetric,

and that's a Cholesky decomposition an upper triangular matrix

that has this interesting property that if you take

it and multiply its transpose by itself,

you get the original matrix back.

Typo, note if you do the reverse,

you will not get the original matrix back,

you can get something else back.

The other thing we can illustrate here,

rather conveniently is the cross-product function,

which gets us to our original matrix back again.

There was another one way you take a matrix square root,

that was mentioned in the lecture.

It's done through the eigen decomposition and I'm just going to walk through that.

Let's say you have an eigen decomposition,

this is what it looks like.

You have the values and the vectors.

If then we evaluate this,

we get the original matrix back.

If we instead take the square root of the diagonal of the eigenvalues,

we get a different matrix. Here's the thing.

If I then take this matrix and multiply it by itself,

then we get the original matrix back.

Note that if we transpose it,

then we actually get the same matrix back,

but that's only because this matrix is symmetric.

Whereas if we, and perhaps I should go back to the Cholesky decomposition for a moment,

if we multiply the Cholesky decomposition by itself, it's not going to work.

I also provide some useful functions for the computation of rows and columns.

Very briefly we can compute.

The apply function is a very powerful function in

that it can compute any function on a row or on a column.

In this case, for example, if we want to compute the row sum,

we just say apply x1 sum.

Sum is the function that computes it, but it could be arbitrary.

For example, if you wanted something like sum squared, we could do something like this.

The reason for the dots is to distinguish it from the other x we're using here.

What this does is, it basically takes each row of

a matrix and compute the sum squared of its values.

Or we can do that for all columns,

although in this case the matrix is symmetric, so it's the same.

Then next we can use sweep,

which can be used to, for example,

let's say that we have our matrix x,

and let's say we want to subtract 1 from the first row and 2 from the second row.

We do sweep x, then the margin,

so that's row,

then the statistics and

then the default is actually a difference,

but I'll make it explicit.

Note the backticks for the operator.

Note this is particularly good in combination with

applied because you can compute wherever much statistically without using apply.

There's also functions colMeans, rowMeans,

colSums and rowSums which do exactly what you would expect them to.

We'll have our matrix x.

If we scale x, what this did is,

we centred them around 0 to column and then divided them so that they have the same scale.

Now, you can have further arguments which specify only one of the two operations.

For example, centre is true,

scale equals false,

for example, will do only the centring.

You can actually also explicitly specify the centre,

which is like sweep,

and so on. This concludes this particular section.