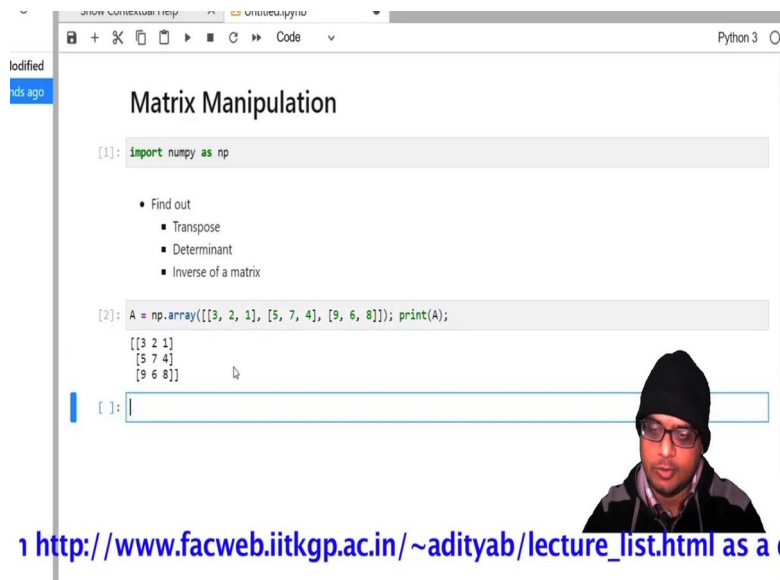


Tools in Scientific Computing
Prof. Aditya Bandopadhyay
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 04
Matrix Manipulations Mohr's circle

Hello everyone, we are in lecture 4. It is going to be a bit of a stressful lecture in the sense that we are going to study Matrix Manipulations. In particular, we will look at how rotation of a coordinate system leads to changes in the elements of the stress tensor. So, let us begin.

(Refer Slide Time: 00:50)



The screenshot shows a Jupyter Notebook interface with the following content:

```
[1]: import numpy as np
```

- Find out
 - Transpose
 - Determinant
 - Inverse of a matrix

```
[2]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A)
```

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

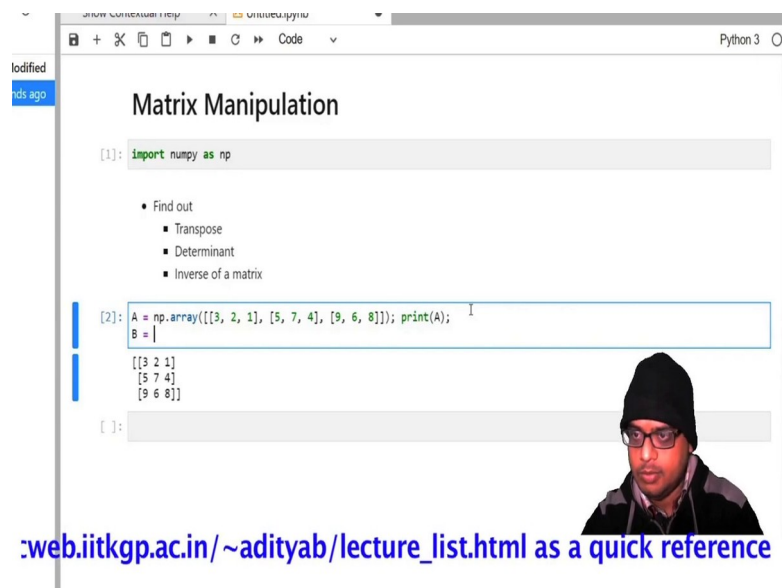
Below the code, there is a small video thumbnail of a man wearing a black beanie and glasses.

http://www.facweb.iitkgp.ac.in/~adityab/lecture_list.html as a

So, let us first import numpy. So, now, let us first look at some elementary matrix operations in particular let us find out the transpose of a matrix, let us find out determinant of a matrix. Let us find out the inverse of a matrix ok. So, let us first define a matrix to work with. So, let us say $A = np.array$ and we want a 3×3 so, there have to be three entries like this.

So, I have made three sets of bracket the first set of bracket is to show that it is an entire matrix, then we have this set of bracket, this set of bracket and this set of bracket; so, each of these three brackets stands for rows. So, let me define 3, 2, 1, 5, 7, 4 and 9, 6, 8. So, let us print what A is also ok. So, A is this 3×3 matrix.

(Refer Slide Time: 02:28)



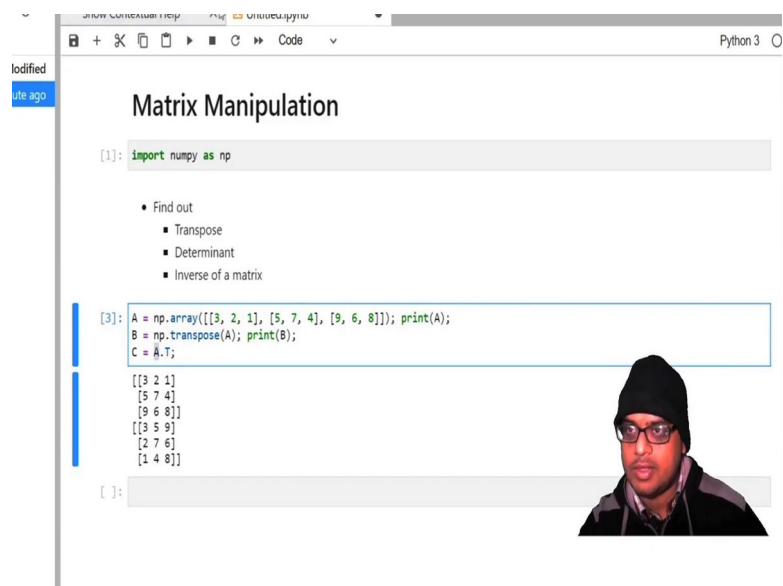
The screenshot shows a Jupyter Notebook interface with the title "Matrix Manipulation". The first code cell contains the import statement: `[1]: import numpy as np`. Below the code, there is a list of tasks to be completed: "Find out" followed by "Transpose", "Determinant", and "Inverse of a matrix". The second code cell contains: `[2]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A); B = []`. The output of this cell shows the matrix A:

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

. A small portrait of a man wearing a black beanie and glasses is visible in the bottom right corner of the notebook interface.

web.iitkgp.ac.in/~adityab/lecture_list.html as a quick reference

(Refer Slide Time: 02:34)



The screenshot shows the same Jupyter Notebook interface. The second code cell now contains: `[3]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A); B = np.transpose(A); print(B); C = A.T;`. The output shows matrix A followed by matrix B:

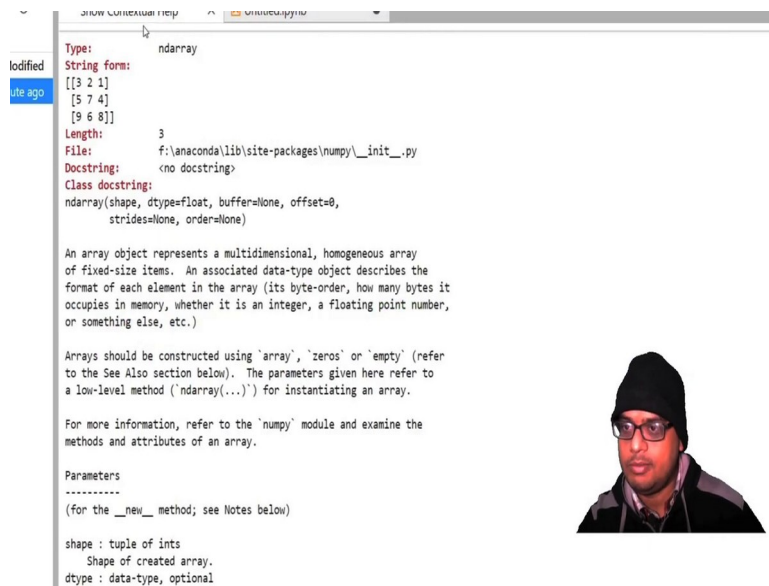
```
[[3 2 1]
 [5 7 4]
 [9 6 8]]
[[3 5 9]
 [2 7 6]
 [1 4 8]]
```

. The small portrait of the man is still present in the bottom right corner.

So, now, let us find out the transpose of A. So, let me go over here `B = np.transpose(A)`. Let me print what B is. So, if you look at the output in A, the first column was 3, 5, 9 whereas, in B the first column is the first row rather is 3, 5, 9; so, successfully taken a transpose of the matrix A.

There is another way of defining the transpose of a matrix. So, `C = A.T`. So, this is also a way of transposing the matrix A in the sense that the object A has a method transpose denoted by T ok.

(Refer Slide Time: 03:23)



```
ndarray
Type: ndarray
String form:
[[3 2 1]
 [5 7 4]
 [9 6 8]]
Length: 3
File: f:\anaconda\lib\site-packages\numpy\_init_.py
Docstring: <no docstring>
Class docstring:
ndarray(shape, dtype=float, buffer=None, offset=0,
        strides=None, order=None)

An array object represents a multidimensional, homogeneous array
of fixed-size items. An associated data-type object describes the
format of each element in the array (its byte-order, how many bytes it
occupies in memory, whether it is an integer, a floating point number,
or something else, etc.)

Arrays should be constructed using 'array', 'zeros' or 'empty' (refer
to the See Also section below). The parameters given here refer to
a low-level method ('ndarray(...)') for instantiating an array.

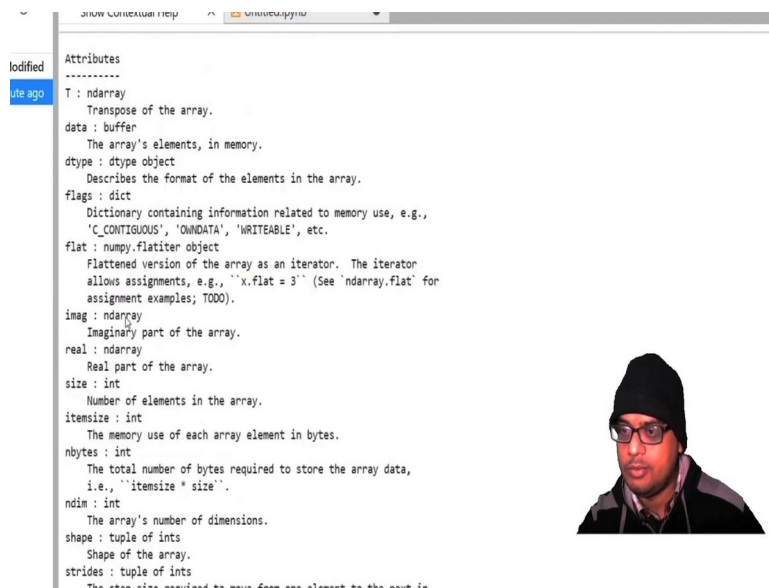
For more information, refer to the 'numpy' module and examine the
methods and attributes of an array.

Parameters
-----
(for the __new__ method; see Notes below)

shape : tuple of ints
        Shape of created array.
dtype : data-type, optional
```



(Refer Slide Time: 03:24)



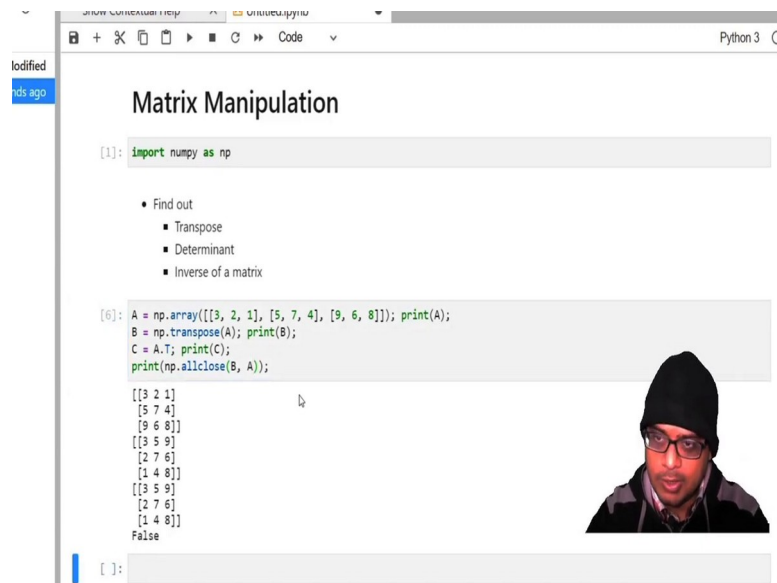
```
Attributes
-----
T : ndarray
    Transpose of the array.
data : buffer
    The array's elements, in memory.
dtype : dtype object
    Describes the format of the elements in the array.
flags : dict
    Dictionary containing information related to memory use, e.g.,
    'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', etc.
flat : numpy.flatiter object
    Flattened version of the array as an iterator. The iterator
    allows assignments, e.g., ``x.flat = 3`` (See 'ndarray.flat' for
    assignment examples; TODO).
imag : ndarray
    Imaginary part of the array.
real : ndarray
    Real part of the array.
size : int
    Number of elements in the array.
itemsize : int
    The memory use of each array element in bytes.
nbytes : int
    The total number of bytes required to store the array data,
    i.e., ``itemsize * size``.
ndim : int
    The array's number of dimensions.
shape : tuple of ints
    Shape of the array.
strides : tuple of ints
    The striding required to move from one element to the next in
```



So, if in fact, if we double click on A and go to the contextual help, it will show some of the attributes such as A.data, A .dtype.

(Refer Slide Time: 03:35)

(Refer Slide Time: 04:40)



The screenshot shows a Jupyter Notebook interface with the following content:

```
[1]: import numpy as np
```

- Find out
 - Transpose
 - Determinant
 - Inverse of a matrix

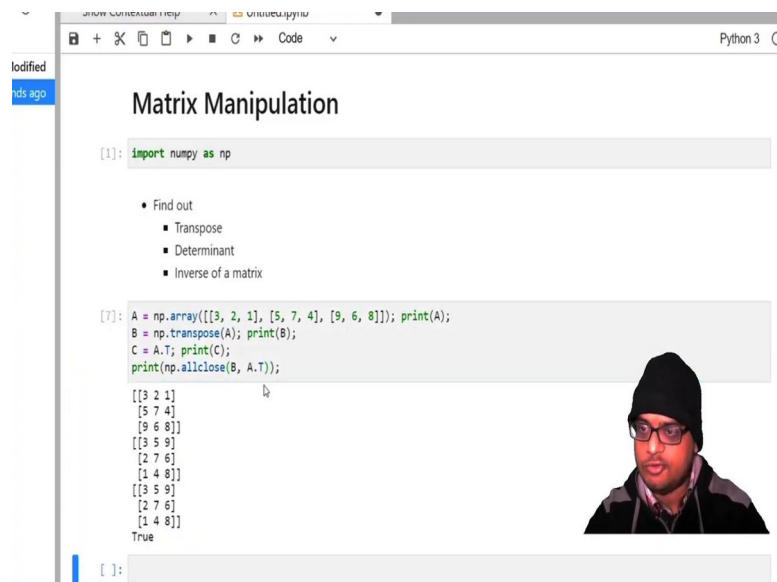
```
[6]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A);  
B = np.transpose(A); print(B);  
C = A.T; print(C);  
print(np.allclose(B, A));
```

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

A small video inset of a man wearing a black beanie and glasses is visible in the bottom right corner of the notebook window.

In fact, if I do `np.allclose(B, A)`, it says false because; obviously, B is a transpose of A. Moreover, `A.T` this should show True ok.

(Refer Slide Time: 04:46)



The screenshot shows a Jupyter Notebook interface with the following content:

```
[1]: import numpy as np
```

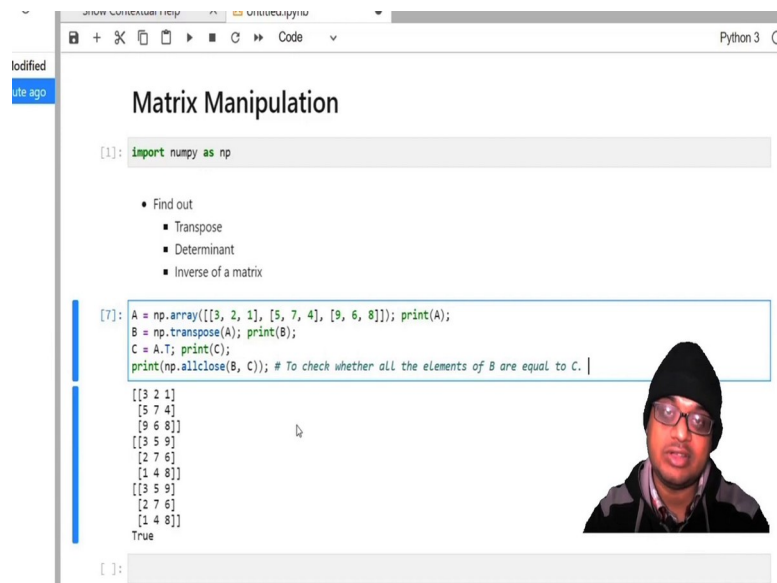
- Find out
 - Transpose
 - Determinant
 - Inverse of a matrix

```
[7]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A);  
B = np.transpose(A); print(B);  
C = A.T; print(C);  
print(np.allclose(B, A.T));
```

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

A small video inset of a man wearing a black beanie and glasses is visible in the bottom right corner of the notebook window.

(Refer Slide Time: 04:52)



The screenshot shows a Jupyter Notebook interface with the following content:

```
[1]: import numpy as np
```

- Find out
 - Transpose
 - Determinant
 - Inverse of a matrix

```
[7]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A);  
B = np.transpose(A); print(B);  
C = A.T; print(C);  
print(np.allclose(B, C)); # To check whether all the elements of B are equal to C. |
```

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

A video inset in the bottom right corner shows a person wearing a black beanie and glasses, speaking.

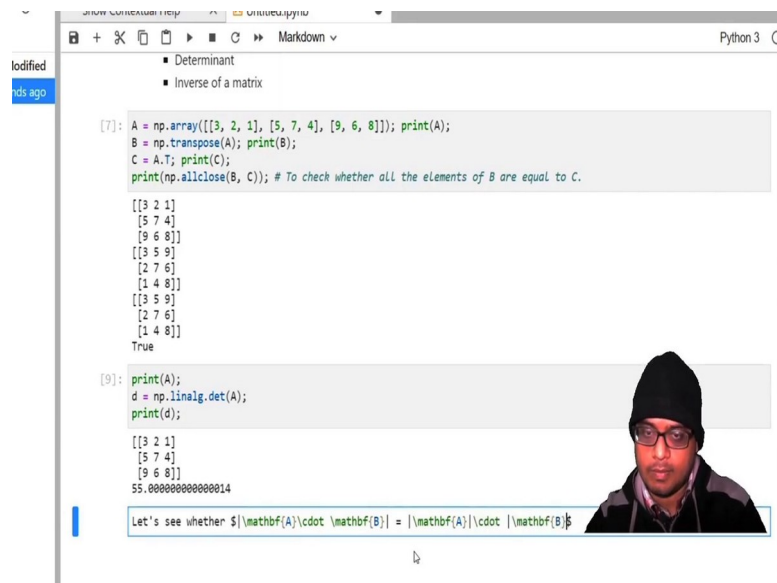
So, this is how we can make checks. So, this is to check whether all the elements of B are equal to C. So, even if one element is not equal, it throws false and the reason why it is called all close? So, in this case, we are using integers as inputs to the array.

In case, you start using floating point numbers, then precise equation, precisely equating two floating point numbers is not something which is logical to do rather the fact that a computer has a finite representation in terms of bits so, that implies that you will always be restricted by the representation of a number ok. So, often 0.0016 and 0.001599 so, they have to be interpreted as the same number that is why it is called as all close.

(Refer Slide Time: 05:56)

So, the determinant is equal to 55.000014 and you can do it by hand, you can do it by hand, and you will see that the determinant is actually 55 ok. If you take out the determinant of this, you will find out it is not 55.000014, but it is rather 55 and this is what I just spoke about. It is because of the finite accuracy and representation of a number in a computer ok, it is stored in bits and thus, there will always be these small errors ok.

(Refer Slide Time: 07:46)



The screenshot shows a Jupyter Notebook with the following code and output:

```
[7]: A = np.array([[3, 2, 1], [5, 7, 4], [9, 6, 8]]); print(A);
      B = np.transpose(A); print(B);
      C = A.T; print(C);
      print(np.allclose(B, C)); # To check whether all the elements of B are equal to C.

[[3 2 1]
 [5 7 4]
 [9 6 8]]
[[3 5 9]
 [2 7 6]
 [1 4 8]]
[[3 5 9]
 [2 7 6]
 [1 4 8]]
True

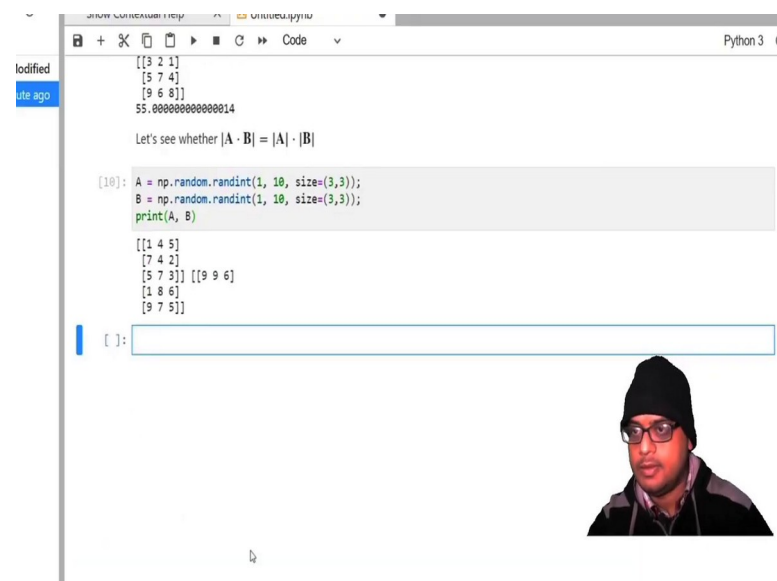
[9]: print(A);
      d = np.linalg.det(A);
      print(d);

[[3 2 1]
 [5 7 4]
 [9 6 8]]
55.00000000000014
```

Let's see whether $|\mathbf{A} \cdot \mathbf{B}| = |\mathbf{A}| \cdot |\mathbf{B}|$

So, let us look at a very important property of two matrices. So, let us see whether $|\mathbf{A} \cdot \mathbf{B}| = |\mathbf{A}| \cdot |\mathbf{B}|$

(Refer Slide Time: 08:19)



The screenshot shows a Jupyter Notebook with the following code and output:

```
[[3 2 1]
 [5 7 4]
 [9 6 8]]
55.00000000000014

Let's see whether  $|\mathbf{A} \cdot \mathbf{B}| = |\mathbf{A}| \cdot |\mathbf{B}|$ 

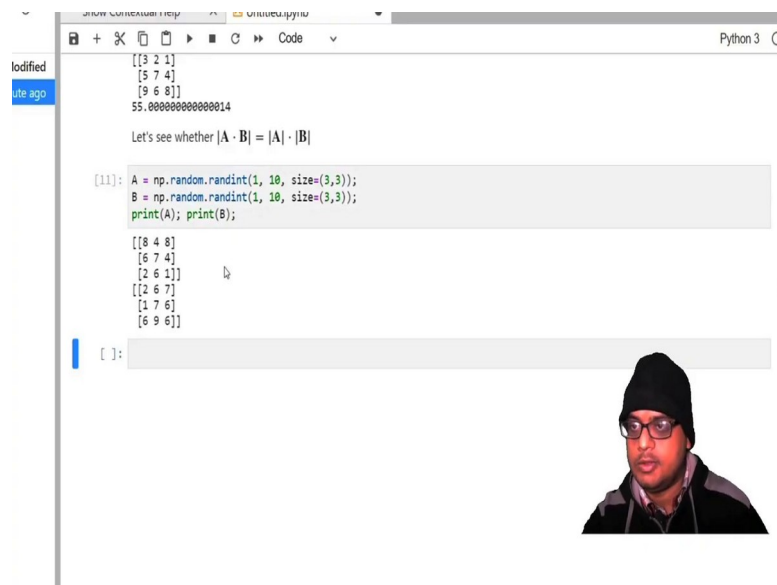
[10]: A = np.random.randint(1, 10, size=(3,3));
       B = np.random.randint(1, 10, size=(3,3));
       print(A, B)

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

[ ]:
```


So, let us look whether this holds true or not. So, let us initialize two random matrices. So, $A = np.random.randint()$ and say we want to sample randint from 1 to 10 and the size let us declare it as 3×3 . Similarly, B will be the same thing and the fact that you are calling it twice; it will generate a new random array. So in fact, let me print both A and B.

(Refer Slide Time: 09:03)



```
[[3 2 1]
 [5 7 4]
 [9 6 8]]
55.000000000000014

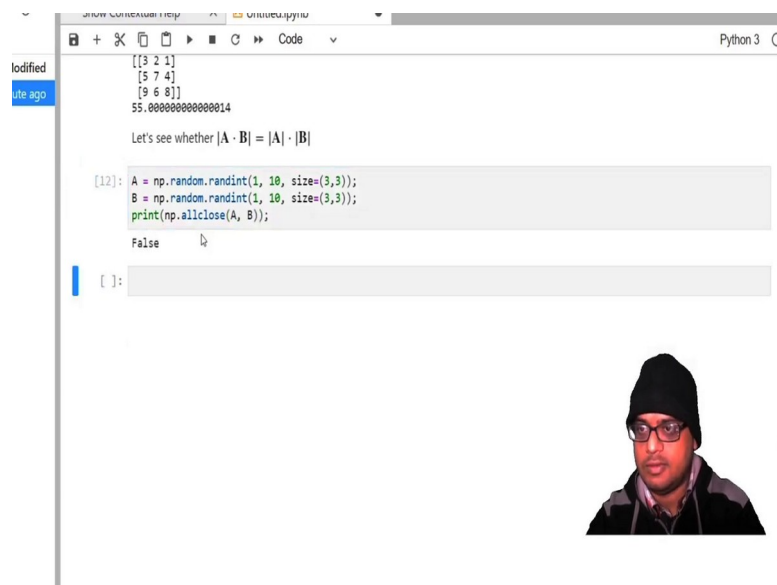
Let's see whether |A · B| = |A| · |B|

[11]: A = np.random.randint(1, 10, size=(3,3));
      B = np.random.randint(1, 10, size=(3,3));
      print(A); print(B);

[[8 4 8]
 [6 7 4]
 [2 6 1]]
[[2 6 7]
 [1 7 6]
 [6 9 6]]

[ ]:
```

(Refer Slide Time: 09:14)



```
[[3 2 1]
 [5 7 4]
 [9 6 8]]
55.000000000000014

Let's see whether |A · B| = |A| · |B|

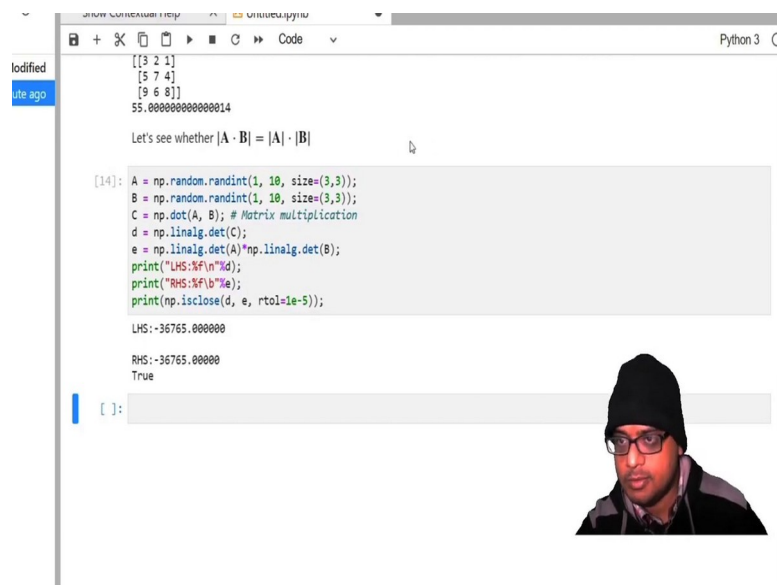
[12]: A = np.random.randint(1, 10, size=(3,3));
      B = np.random.randint(1, 10, size=(3,3));
      print(np.allclose(A, B));

False

[ ]:
```

So, A and B are obviously, different and another way, another professional way to check whether it is different is this should be false and it is false. So, it means A and B are distinct.

(Refer Slide Time: 09:26)



```
[[3 2 1]
 [5 7 4]
 [9 6 8]]
55.000000000000014

Let's see whether |A · B| = |A| · |B|

[14]: A = np.random.randint(1, 10, size=(3,3));
      B = np.random.randint(1, 10, size=(3,3));
      C = np.dot(A, B); # Matrix multiplication
      d = np.linalg.det(C);
      e = np.linalg.det(A)*np.linalg.det(B);
      print("LHS:%f\n"%d);
      print("RHS:%f\n"%e);
      print(np.isclose(d, e, rtol=1e-5));

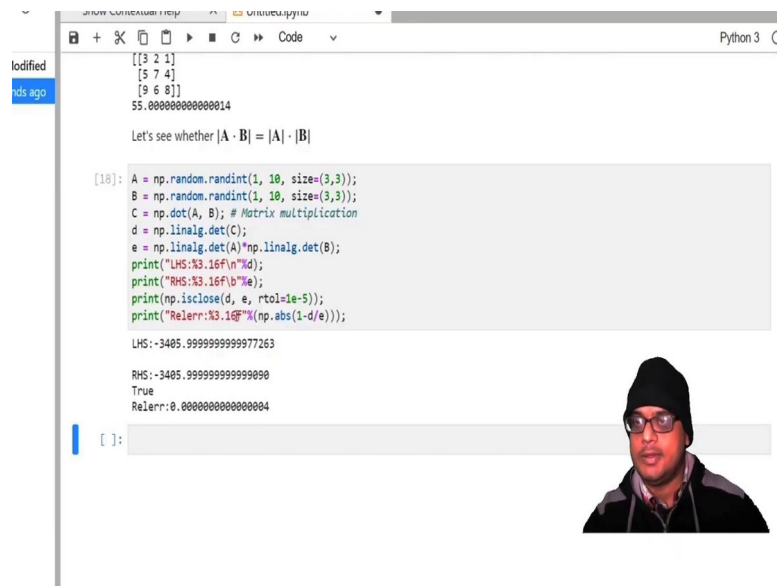
LHS:-36765.000000
RHS:-36765.000000
True

[ ]:
```

So, let us do the following. Let us define $C = \text{np.dot}(A, B)$. So, this is a way to do matrix multiplication. So, the way to do matrix multiplication is the row of A multiplies the first row of B and so on. It is the proper matrix multiplication, it is not an element wise operation, it is what we have learned in school alright. So, let me call d as the determinant of C and let me call e as the determinant of sorry A times this. Now, let me simply check np dot is close.

So, if we are trying to compare two scalar floating-point numbers instead of using all close, we can use isclose. So, isclose is typically used for comparing two scalars. So, we want to compare d and e and we give it a relative tolerance of $1e^{-5}$ for example, let me directly print this. So, it says True. So, it is close, it gives true. Let me in fact, print so, let us say LHS: %3.16f\n"%d and print RHS:%3.16f\n"%e ok. So, there you go.

(Refer Slide Time: 11:25)




```
[[3 2 1]
 [5 7 4]
 [9 6 8]]
55.0000000000000014

Let's see whether  $|A \cdot B| = |A| \cdot |B|$ 

[18]: A = np.random.randint(1, 10, size=(3,3));
      B = np.random.randint(1, 10, size=(3,3));
      C = np.dot(A, B); # Matrix multiplication
      d = np.linalg.det(C);
      e = np.linalg.det(A)*np.linalg.det(B);
      print("LHS:%3.16f"%d);
      print("RHS:%3.16f"%e);
      print(np.isclose(d, e, rtol=1e-5));
      print("Relerr:%3.16f"%(np.abs(1-d/e)));

LHS:-3485.999999999977263
RHS:-3485.99999999999090
True
Relerr:0.0000000000000004
```



These are the two in fact, let me increase the number of digits it will print ok. So, in this particular case, they are equal to a very large degree, let me run it again ok. Over here look at this, it is 1440.0000 whatever and in this case, it is 1439.99999, but the relative error between these two numbers is less than 10^{-5} .

In fact, let us print out what the relative error between them is and the relative error will be $\text{np.abs}(1-d/e)$. So, relative error is quite small in fact, its all the way and the last digit becomes 4. So, because the relative error between these two numbers are quite small, for all practical purposes in our calculation, we can consider these to be equal alright.

(Refer Slide Time: 12:48)

```
modified
ses ago

print("LHS:%3.16f"%d);
print("RHS:%3.16f"%e);
print(np.isclose(d, e, rtol=1e-5));
print("Relerr:%3.16f"%(np.abs(1-d/e)));

LHS:-3485.999999999977263

RHS:-3485.99999999999090
True
Relerr:0.000000000000004

Finding out the inverse

[20]: print(A);
[[1 1 4]
 [4 6 8]
 [3 2 3]]

[21]: a = np.linalg.det(A); print(a);
-26.000000000000004

[22]: B = np.linalg.inv(A); print(B);
[[-0.07692308 -0.19230769  0.61538462]
 [-0.46153846  0.34615385 -0.30769231]
 [ 0.38461538 -0.03846154 -0.07692308]]

[ ]:
```



So, we have so far considered the determinant, we have so far considered the transpose, let us now proceed to find out the inverse of a matrix. So, alright. So, let me define let us define a random I mean we have already defined a random array so, let me print out that array. So, this is the not the array, the matrix ok. This is the matrix A. Let us now try to find out the inverse of this matrix.

So, an important property in order to find out the inverse is that the determinant must be non-zero. If the determinant is 0, then we cannot find out the inverse. So, let us do a quick check whether the determinant is 0 or not. So, let us say $c = np.linalg$ in fact, let me call it $a = np.linalg.det(A)$, print a. So, $|A|$ is obviously, not equal to 0. So, in that case, we can find out the inverse. So, let me call it B so, $B = np.linalg.inv(A)$, let me print what B is. So, B is this particular matrix.

(Refer Slide Time: 14:07)

```

print("LHS:%3.16f\n%d");
print("RHS:%3.16f\n%e");
print(np.isclose(d, e, rtol=1e-5));
print("Relerr:%3.16f\n(np.abs(1-d/e));");

LHS:-3485.999999999977263

RHS:-3485.99999999999090
True
Relerr:0.0000000000000004

Finding out the inverse

[20]: print(A);

[[1 1 4]
 [4 6 8]
 [3 2 3]]

[21]: a = np.linalg.det(A); print(a);

-26.000000000000004

[22]: B = np.linalg.inv(A); print(B);

[[-0.07692308 -0.19230769  0.61538462]
 [-0.46153846  0.34615385 -0.30769231]
 [ 0.38461538 -0.03846154 -0.07692308]]

Quickly check whether $A \cdot B = I$

```

And let us do a Quick check whether A times B is equal to the identity matrix or not.

(Refer Slide Time: 14:18)

```

[3 2 3]]

[21]: a = np.linalg.det(A); print(a);

-26.000000000000004

[22]: B = np.linalg.inv(A); print(B);

[[-0.07692308 -0.19230769  0.61538462]
 [-0.46153846  0.34615385 -0.30769231]
 [ 0.38461538 -0.03846154 -0.07692308]]

Quickly check whether A · B = I

[24]: C = np.dot(A, B); print(C);

[[ 1.00000000e+00  2.7755756e-17 -5.55111512e-17]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  4.85722573e-17  1.00000000e+00]]

[ ]:

```

So, let me define C as $C = np.dot(A, B)$ and let me print what C is. So, if you look at this, it is $1.00e^0$, $2.77 \cdot 10^{-17}$, $-5.55 \cdot 10^{-17}$. So, these are as good as 0 alright.

(Refer Slide Time: 14:55)

```

[3 2 3]]
[21]: a = np.linalg.det(A); print(a);
-26.000000000000004


[22]: B = np.linalg.inv(A); print(B);
[[-0.07692308 -0.19230769  0.61538462]
 [-0.46153846  0.34615385 -0.30769231]
 [ 0.38461538 -0.03846154 -0.07692308]]

Quickly check whether  $A \cdot B = I$ 

[25]: C = np.dot(A, B); print(np.round(C, 1));
[[ 1.  0. -0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

 $(A^{-1})^T = (A^T)^{-1}$ 

```



So, `np.round(C,1)`. So, `np.round` means rounded to the first decimal. So, in that case, instead of having those very small numbers appearing, we round them off to one decimal place and they do come out to be 1, 1, 1, 0, 0, 0, 0. So, `C` is an identity matrix, and this helps us in verifying what we already know.

This is quite a trivial task, but nevertheless it instills a bit of confidence. Let us do this particular check as well this is a theorem in matrix algebra. So, `A` inverse transpose is equal to `A` transpose inverse. So, let us see whether this is true or not.

(Refer Slide Time: 15:54)

```

[22]: B = np.linalg.inv(A); print(B);
[[-0.07692308 -0.19230769  0.61538462]
 [-0.46153846  0.34615385 -0.30769231]
 [ 0.38461538 -0.03846154 -0.07692308]]


Quickly check whether  $A \cdot B = I$ 

[25]: C = np.dot(A, B); print(np.round(C, 1));
[[ 1.  0. -0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

 $(A^{-1})^T = (A^T)^{-1}$ 

[28]: lhs = np.transpose(np.linalg.inv(A));
      rhs = np.linalg.inv(np.transpose(A));
      print(lhs);
      print(rhs);
      print(np.allclose(lhs, rhs))
[[-0.07692308 -0.46153846  0.38461538]
 [-0.19230769  0.34615385 -0.03846154]
 [ 0.61538462 -0.30769231 -0.07692308]]
[[-0.07692308 -0.46153846  0.38461538]
 [-0.19230769  0.34615385 -0.03846154]
 [ 0.61538462 -0.30769231 -0.07692308]]
True

```

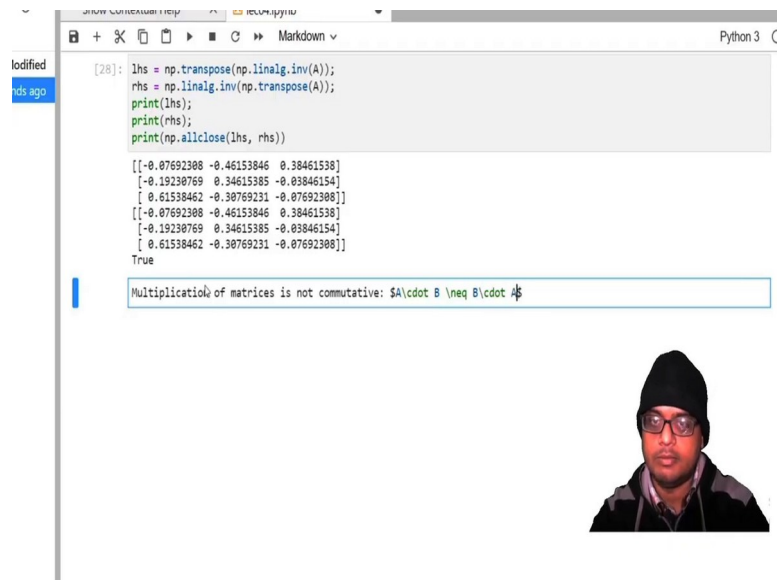


So, let us say lhs and it is equal to np.linalg.transpose sorry transpose is not inside linalg, it is simply outside, so lhs = np.transpose(np.linalg.inv(A)) so, this is lhs and rhs = np.linalg.inv(np.transpose(A)). So, then let us print whether lhs and rhs are equal.

So, here we will write np.allclose(lhs, rhs). So, it says True meaning whatever we have claimed to be true is in fact, true and these are very famous theorems in matrix algebra obviously, they are True, but again it is just to demonstrate how we can manipulate matrices.

Let us print out the lhs and rhs just for good measure alright. So, visual inspection as well allows us to see that lhs and rhs are equal. So, before proceeding, let us quickly save this file. Let me rename it to be lec 04 alright.

(Refer Slide Time: 17:38)



```
[28]: lhs = np.transpose(np.linalg.inv(A));
      rhs = np.linalg.inv(np.transpose(A));
      print(lhs);
      print(rhs);
      print(np.allclose(lhs, rhs))

[[[-0.07692308 -0.46153846  0.38461538]
  [-0.19230769  0.34615385 -0.03846154]
  [ 0.61538462 -0.30769231 -0.07692308]]
 [[[-0.07692308 -0.46153846  0.38461538]
  [-0.19230769  0.34615385 -0.03846154]
  [ 0.61538462 -0.30769231 -0.07692308]]
 True
```

Multiplication of matrices is not commutative: $A \cdot B \neq B \cdot A$



So, let us verify one more famous theorem in matrix algebra that Multiplication of two matrices is not commutative that is $A \cdot B \neq B \cdot A$.

(Refer Slide Time: 17:54)

```


Python 3
modified
ids ago
[[ -0.19230769  0.34615385 -0.03846154]
 [  0.61538462 -0.30769231 -0.07692308]
 [ -0.07692308 -0.46153846  0.38461538]
 [ -0.19230769  0.34615385 -0.03846154]
 [  0.61538462 -0.30769231 -0.07692308]]
True
Multiplication of matrices is not commutative:  $A \cdot B \neq B \cdot A$ 

[29]: print(A); print(B);

[[1 1 4]
 [4 6 8]
 [3 2 3]]
[[ -0.07692308 -0.19230769  0.61538462]
 [ -0.46153846  0.34615385 -0.30769231]
 [ 0.38461538 -0.03846154 -0.07692308]]

[ ]:

```



So, we have two matrices A and B already like this.

(Refer Slide Time: 18:14)

```

Python 3
modified
ids ago
print(np.allclose(C, D))

[[ -0.07692308 -0.46153846  0.38461538]
 [ -0.19230769  0.34615385 -0.03846154]
 [  0.61538462 -0.30769231 -0.07692308]]
[[ -0.07692308 -0.46153846  0.38461538]
 [ -0.19230769  0.34615385 -0.03846154]
 [  0.61538462 -0.30769231 -0.07692308]]
True
Multiplication of matrices is not commutative:  $A \cdot B \neq B \cdot A$ 


[32]: A = np.random.randint(1, 10, size=(3,3));
      B = np.random.randint(1, 10, size=(3,3));
      C = np.dot(A,B); D = np.dot(B, A);
      print(C); print(D);
      print(np.allclose(C, D));

[[ 96  88  90]
 [100 101 108]
 [ 52  49  53]]
[[ 67  86  60]
 [ 82 124  90]
 [ 63  82  59]]
False

A function to check whether a matrix is symmetric or not

[ ]:

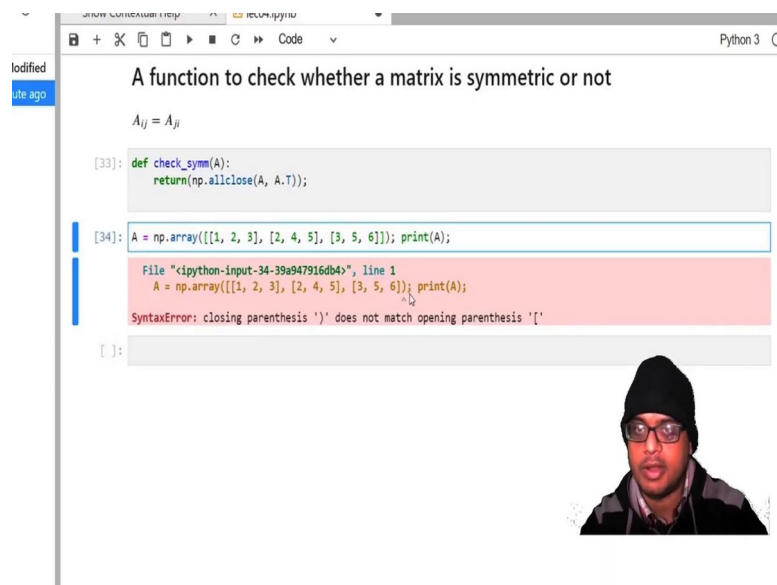
```



In fact, let me re-declare A and B to be two new matrices. Let me copy this bit of let me reuse that code alright. So, A and B are randomly initialized. So, now, we need to check whether $A \cdot B = B \cdot A$. So, let me create $C = np.dot(A,B)$.

Now, let me create $D = np.dot(B, A)$. So, now, we want to see whether $C = D$ or not. So, let us just simply print $np.allclose(C, D)$ and it says False quite obviously, because this theorem is obviously, correct. So obviously, the two products are different and hence the result alright.

(Refer Slide Time: 19:38)



```
A function to check whether a matrix is symmetric or not

 $A_{ij} = A_{ji}$ 

[33]: def check_symm(A):
      return np.allclose(A, A.T);

[34]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);

File "<ipython-input-34-39a947916db4>", line 1
A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);
SyntaxError: closing parenthesis ')' does not match opening parenthesis '['

[ ]:
```

So, with this let us prove or let us write a small function to check whether a matrix is symmetric or not. So, how can we do this? So, if a matrix is symmetric, then $A_{ij} = A_{ji}$. So, if a matrix is symmetric, then swapping the two indices makes no difference.

So, let us define a function called as `check_symm` and it will take an input as an array, as a matrix A . So, inside this we will simply check whether $A^T = A$. So, it will simply return `np.allclose(A, A.T)` that is it. So, let me run this cell. Let me declare an array, let me declare a matrix. So, let me make it 1, 2, 3, 2, 4, 5 and 3, 5, 6. Let me print what A is. We forgot bracket over here.

(Refer Slide Time: 21:00)

```
A function to check whether a matrix is symmetric or not
Aij = Aji

[33]: def check_symm(A):
      return np.allclose(A, A.T);


[35]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);

[[1 2 3]
 [2 4 5]
 [3 5 6]]

[36]: print(check_symm(A));

True

[ ]: |
```



So obviously, by visual inspection, we can see that matrix A is symmetric. I mean we have made small arrays and so, we can easily have a visual check whether its true or not. But when you are doing a computation which involves large arrays, you better make some kind of function like this and in fact, having a function like this, you can have an automatic check whether to proceed with a certain computation or not.

Suppose you are working with stresses and you know that the stress tensor has to be symmetric so, it is a good check to start off whether a stress tensor is symmetric or not alright. So, let us pass A to the checker. So, `print(check_symm(A))` and it says true. In fact, I have mentioned this in one of the earlier lectures as well. A is just a placeholder, this particular A has nothing to do with this particular A .

(Refer Slide Time: 22:08)

```

Python 3
A function to check whether a matrix is symmetric or not

 $A_{ij} = A_{ji}$ 

[37]: def check_symm(aditya):
        return(np.allclose(aditya, aditya.T));

[38]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);


[[1 2 3]
 [2 4 5]
 [3 5 6]]

[39]: print(check_symm(A));

True

[ ]:

```



Even if I declare this as something like this, it will hardly make a difference ok. So, it is just a placeholder, it is not a variable which is accessible to the other functions, it is it has a very limited local scope alright.

(Refer Slide Time: 22:35)

```

Python 3
return(np.allclose(aditya, aditya.T));

[38]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);


[[1 2 3]
 [2 4 5]
 [3 5 6]]

[39]: print(check_symm(A));

True

# Matrix-Matrix double dot product: $A:B = A_{ij}B_{ij}$

```



So, let us define the double dot product. So, it is defined as $A:B = A_{ij}B_{ij}$.

(Refer Slide Time: 22:49)

Modified 1 minute ago

```
[38]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);
```

```
[[1 2 3]
 [2 4 5]
 [3 5 6]]
```

```
[39]: print(check_symm(A));
```

```
True
```

Matrix-Matrix double dot product: $A : B = A_{ij}B_{ij}$

[]:

And this is using the Einstein summation notation where on the right-hand side, we have i which is a repeated index and j which is a repeated index.

(Refer Slide Time: 23:00)

Modified 1 minute ago

```
[38]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);
```

```
[[1 2 3]
 [2 4 5]
 [3 5 6]]
```

```
[39]: print(check_symm(A));
```

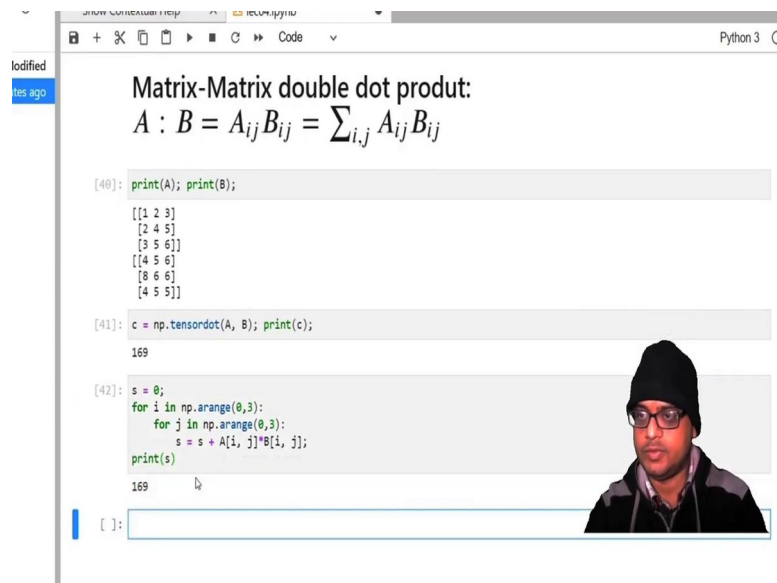
```
True
```

Matrix-Matrix double dot product: $A:B = A_{ij}B_{ij} = \sum_{i,j} A_{ij}B_{ij}$

[]:

So, essentially it implies that this is actually equal to $\sum_{i,j} A_{ij}B_{ij}$.

(Refer Slide Time: 23:14)



The screenshot shows a Jupyter notebook interface with the following content:

Matrix-Matrix double dot product:
$$A : B = A_{ij} B_{ij} = \sum_{i,j} A_{ij} B_{ij}$$

```
[40]: print(A); print(B);  
[[1 2 3]  
 [2 4 5]  
 [3 5 6]]  
[[4 5 6]  
 [8 6 6]  
 [4 5 5]]
```

```
[41]: c = np.tensordot(A, B); print(c);  
169
```

```
[42]: s = 0;  
for i in np.arange(0,3):  
    for j in np.arange(0,3):  
        s = s + A[i, j]*B[i, j];  
print(s)  
169
```

So, essentially it will be something which resembles $A_{11} B_{11}$; it will be $A_{11} B_{11} + A_{12} B_{12} + A_{13} B_{13} + A_{21} B_{21}$ and so on all the way from i equal to 1, 2, 3 to j equal to 1, 2, 3. So, let us see how to do this and this kind of product in Python is called as a tensor dot product. So, let us have, let us see what the two arrays we already have. So, these are the two arrays. So, let us define $c = \text{np.tensordot}(A, B)$ and let us print out the value of c ; so, it is 169.

So, how do we know whether this particular sum is correct or not? So, let us define sum $s = 0$, for i in $\text{np.arange}(0,3)$, for j in $\text{np.arange}(0,3)$, $s = s + A[i, j]*B[i, j]$. Well, I have written it like this, but this is also equivalent, this is also the same thing. So, at the end of the computation, let us print what s is.

So, s is equal to 169. So, we have established what a tensor dot product is. So, tensor dot product contracts two tensors into a single scalar and this kind of double dot product is typically used in finding out terms like viscous dissipation.

(Refer Slide Time: 25:35)

```

[38]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);

[[1 2 3]
 [2 4 5]
 [3 5 6]]

[39]: print(check_sym(A));

True

# Matrix-Matrix double dot product: SA:B = A_{ij}B_{ij} = \sum_{i,j} A_{ij}B_{ij}
# \tau_{ij}S_{ij}

[40]: print(A); print(B);

[[1 2 3]
 [2 4 5]
 [3 5 6]]
[[4 5 6]
 [8 6 6]
 [4 5 5]]

[41]: c = np.tensordot(A, B); print(c);

169

[42]: s = 0;
for i in np.arange(0,3):
    for j in np.arange(0,3):
        s = s + A[i, j]*B[i, j];

```



So, in fluid mechanics or convective heat and mass transfer or rather in heat transfer, you will find that if you write down the energy equation, you end up with a viscous dissipation term which resembles something like this $\tau_{ij}S_{ij}$ where τ is the stress tensor and S is the rate of deformation tensor.

(Refer Slide Time: 25:42)

```

[38]: A = np.array([[1, 2, 3], [2, 4, 5], [3, 5, 6]]); print(A);

[[1 2 3]
 [2 4 5]
 [3 5 6]]

[39]: print(check_sym(A));

True

Matrix-Matrix double dot product:
A : B = A_{ij}B_{ij} = \sum_{i,j} A_{ij}B_{ij}

\tau_{ij}S_{ij}

[40]: print(A); print(B);

[[1 2 3]
 [2 4 5]
 [3 5 6]]
[[4 5 6]
 [8 6 6]
 [4 5 5]]

[41]: c = np.tensordot(A, B); print(c);

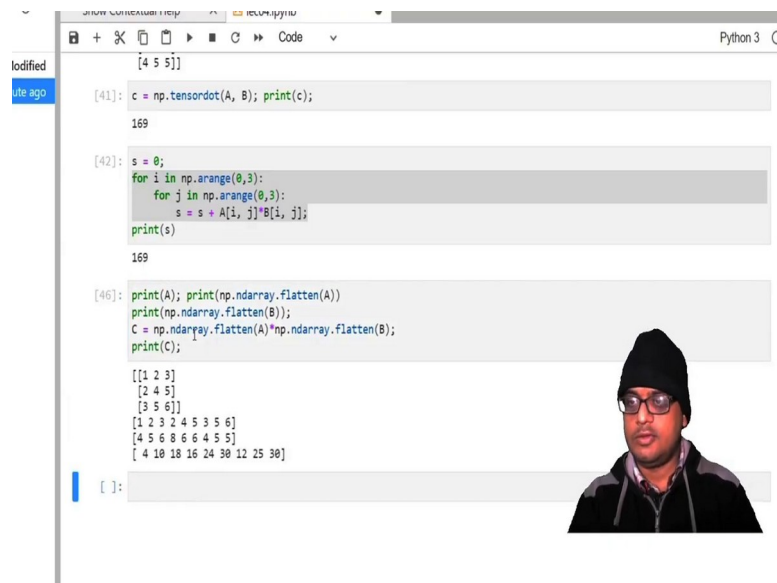
169

```



So, this is how the viscous dissipation is obtained and it has the same tensor dot structure as we have shown over here.

(Refer Slide Time: 26:35)



```
[4 5 5]
[41]: c = np.tensordot(A, B); print(c);
169
[42]: s = 0;
for i in np.arange(0,3):
    for j in np.arange(0,3):
        s = s + A[i, j]*B[i, j];
print(s)
169
[46]: print(A); print(np.ndarray.flatten(A))
print(np.ndarray.flatten(B));
C = np.ndarray.flatten(A)*np.ndarray.flatten(B);
print(C);
[[1 2 3]
 [2 4 5]
 [3 5 6]]
[1 2 3 2 4 5 3 5 6]
[4 5 6 8 6 6 4 5 5]
[ 4 10 18 16 24 30 12 25 30]
[ ]:
```

So, yet another way of quickly finding out this tensor dot product using vectorized. So, we have made use of two loops over here and obviously, in Python, you want to avoid using loops, you want to vectorize your code, you do not want to make explicit declarations of loops.

So, let me just quickly show you how to do that. So, simply we will first flatten all the elements of the matrix into a straight line. So, the way to flatten something; so, let me print out what A is. Now, let me print out what the flattened version of A is `np.ndarray.flatten(A)` ok. So, I have essentially flattened all the elements of A.

So, now you can imagine if we have a, if we have an, if you have these two arrays so, we need to take a element wise product of these two ok, we need to take an element wise product of these two and whatever we obtain using that we have to sum over all the elements. So, let me just show it. So, C equal to the product of this times this. So, let me in fact, print what C is for your benefit. So, this is what C is and now simply I need to sum over all the elements of C.

(Refer Slide Time: 27:47)

```
Python 3
[4 5 5]]
[41]: c = np.tensordot(A, B); print(c);
169

[42]: s = 0;
for i in np.arange(0,3):
    for j in np.arange(0,3):
        s = s + A[i, j]*B[i, j];
print(s)
169

[47]: print(A); print(np.ndarray.flatten(A))
print(np.ndarray.flatten(B));
C = np.ndarray.flatten(A)*np.ndarray.flatten(B);
print(np.sum(C));

[[1 2 3]
 [2 4 5]
 [3 5 6]]
[1 2 3 2 4 5 3 5 6]
[4 5 6 8 6 6 4 5 5]
169

[ ]:
```



So, the way I can do that is `np.sum(C)` and it does give 169. So, I have shown you a very efficient way of going about things.

(Refer Slide Time: 28:02)

```
Python 3
[4 5 5]]
[41]: c = np.tensordot(A, B); print(c);
169

[42]: s = 0;
for i in np.arange(0,3):
    for j in np.arange(0,3):
        s = s + A[i, j]*B[i, j];
print(s)
169

[47]: # A vectorized way of computing the tensordot product
C = np.sum(np.ndarray.flatten(A)*np.ndarray.flatten(B));

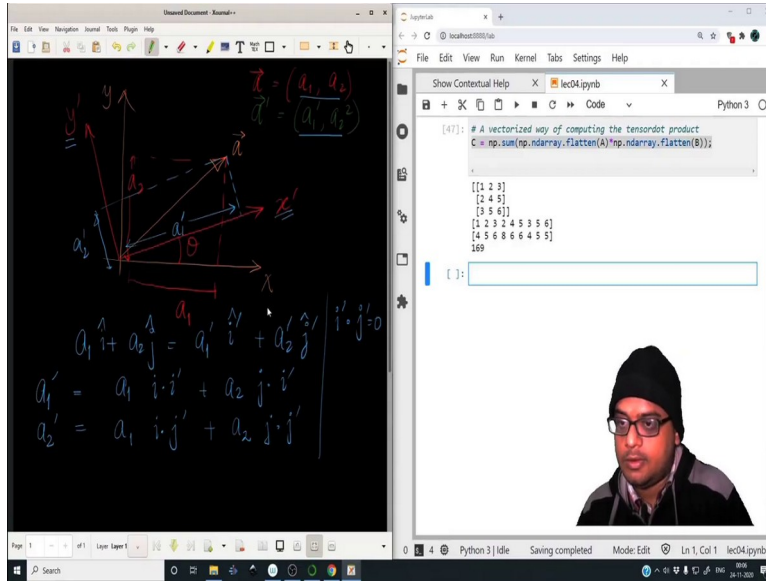
[[1 2 3]
 [2 4 5]
 [3 5 6]]
[1 2 3 2 4 5 3 5 6]
[4 5 6 8 6 6 4 5 5]
169

[ ]:
```



So, simply the way to do a tensor dot product would have been this. So, this is a vectorized way of computing the tensor dot product. So, instead of having these two loops, we have written everything in a single line and that saves time. When the matrices become really big, this is how you start saving time.

(Refer Slide Time: 28:41)



So, let us now proceed to; so, let us now proceed to find out a rotation matrix. So, what is rotation matrix? So, consider rather the transformation matrix it should be clearer. So, let us consider a coordinate system x, y and a vector say this is the vector a . Now, in a rotated coordinate system such as this let us say this is x' and this is y' so, we have rotated this coordinate system by an angle θ .

So, we want to, so, we are interested to know what are the components of a in this rotated coordinate system. So, let the components of a be a_1 and a_2 in the first coordinate system.

So, what are a' ? So, a' is a_1' and a_2' . So, geometrically this is a_1 and this is a_2 and this is a' , a_1' and this is going to be a_2' . So, we are interested to relate the components in the rotated coordinate system with the original coordinate system.

So, the fact of the matter is rotation of a coordinate system should not have any bearing on the representation of a meaning $a_1i + a_2j$ should still be equal to $a_1'i' + a_2'j'$ when where i' and j' are unit vectors in the x' and y' direction.

So, now let us take a dot product of everything with i' . So, $a_1' = a_1i \cdot i' + a_2j \cdot i'$. Similarly,

$a_2' = a_1i \cdot j' + a_2j \cdot j'$ where we have made use of the fact that $i' \cdot j' = 0$ that is i' and j' are

orthogonal alright. So, what is $i'i'$? $i'i'$ is the angle between this unit vector and this unit vector that is $\cos(\theta)$.

(Refer Slide Time: 31:35)

The slide content includes the following equations and text:

$$a_1 \hat{i} + a_2 \hat{j} = a_1' \hat{i}' + a_2' \hat{j}' \quad i' \cdot j' = 0$$

$$a_1' = a_1 i \cdot i' + a_2 j \cdot i'$$

$$a_2' = a_1 i \cdot j' + a_2 j \cdot j'$$

$$a_1' = a_1 \cos\theta + a_2 \sin\theta$$

$$a_2' = a_1 (-\sin\theta) + a_2 \cos\theta$$

$$\begin{pmatrix} a_1' \\ a_2' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

$\vec{a}' = \vec{Q} \cdot \vec{a}$ Transformation Matrix

in/~adityab/lecture_list.html as a quick reference

The Jupyter Notebook code and output are:

```
[47]: # A vectorized way of computing the tensor dot product
C = np.sum(np.ndarray.flatten(A)*np.ndarray.flatten(B));

[[1 2 3]
 [2 4 5]
 [3 5 6]]
[[1 2 3 2 4 5 3 5 6]
 [4 5 6 6 6 4 5 5]
 169]
```

(Refer Slide Time: 31:46)

The slide content is identical to the previous slide, including the equations and transformation matrix.

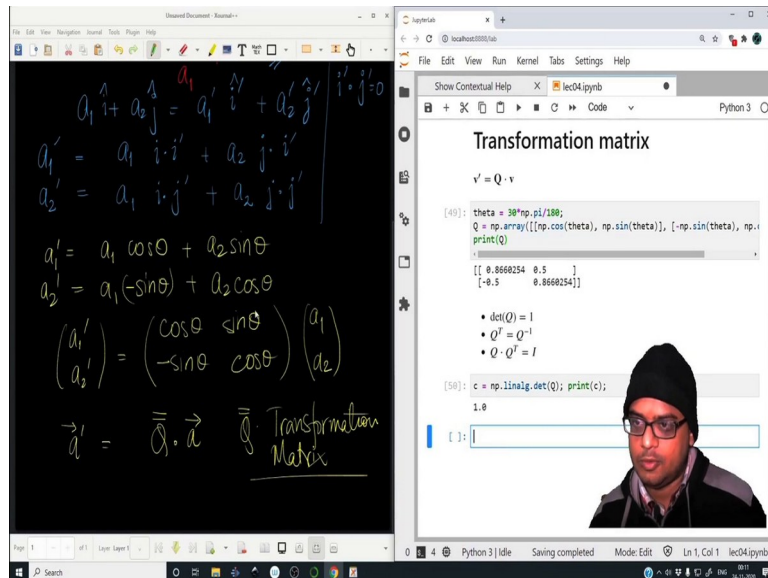
python and octave notebooks can be downloaded from <http://www.kitware.com>

The Jupyter Notebook code and output are identical to the previous slide.

So, essentially the first expression essentially boils down to $a_1' = a_1 \cos(\theta) + a_2(j \cdot i')$ so, j is this direction and i' is this direction. So, this \cos of this angle is basically $\sin(\theta)$, a_2' is

basically $a_1(i \cdot j')$ so, this is j' and this is i so, it is $\cos(90 + \theta)$ which is $-\sin(\theta)$ and a_2 this is j and this is j' so, this angle is obviously, θ so, this will be $\cos(\theta)$ because $j \cdot j' = \cos(\theta)$.

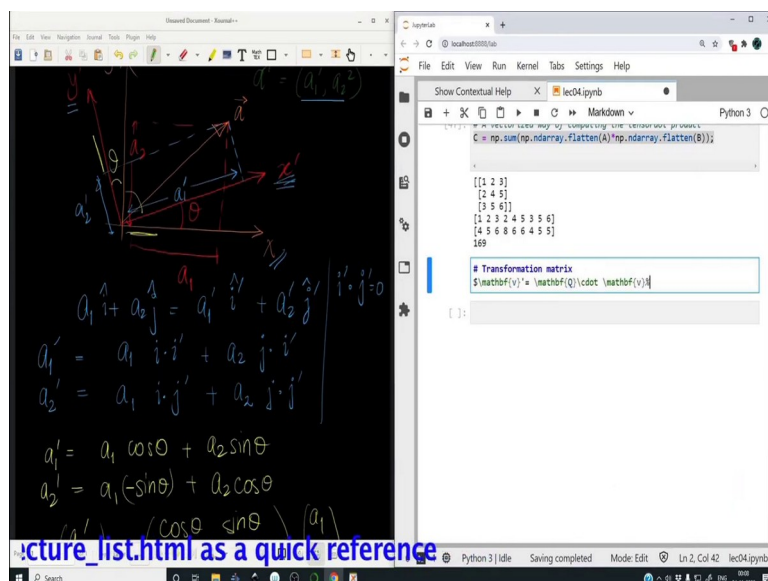
(Refer Slide Time: 32:18)



So, essentially, we end up with a representation like this. So, $\begin{pmatrix} a_1' \\ a_2' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$

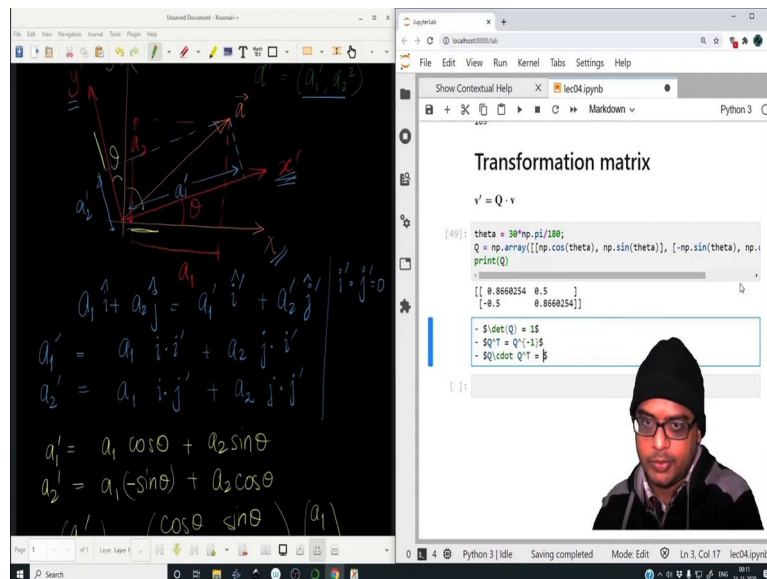
So, essentially $\vec{a}' = \vec{Q} \vec{a}$ where Q is called as the transformation matrix alright.

(Refer Slide Time: 33:06)



So, now let us encode this particular Transformation matrix in Python alright.

(Refer Slide Time: 33:08)

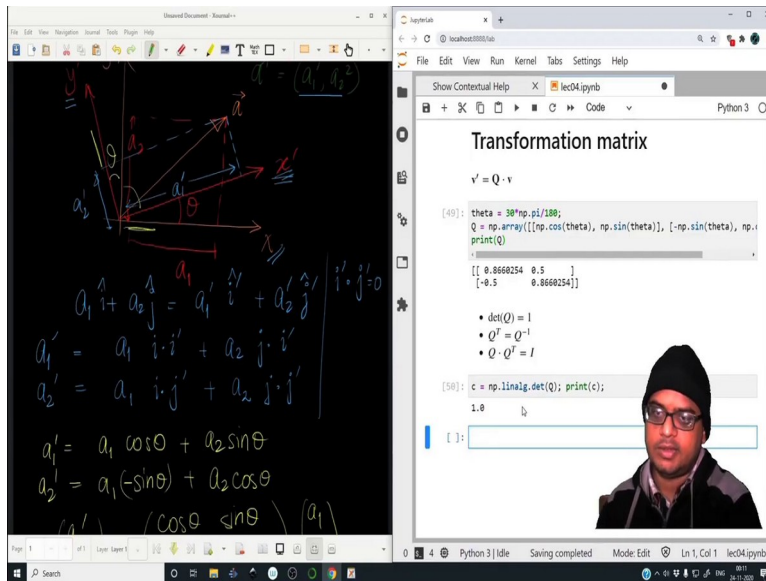


So, we want to find out v' ok. So, let us define first the θ that is the rotation of the coordinate system. So, θ is equal to so, let us declare it in terms of radian directly. So, 30 times so, we have to make it into radian because all these angles have to be in radians and not in degrees. So, times $\text{np.pi}/180$ ok. So, this is the angle.

Let us define $Q = \text{np.array}$. So, let us do a 2D transformation first. So, we need two such brackets. So, one is $\text{np.cos}(\theta)$, $\text{np.sin}(\theta)$, $[-\text{np.sin}(\theta)$, $\text{np.cos}(\theta)$. So, let us print out what the Q matrix looks like. So, Q is the transformation matrix. So, print Q this is what it looks like. So, let us now create a vector rather before delving into the transformation itself, let us quickly look at some properties of the transformation matrix.

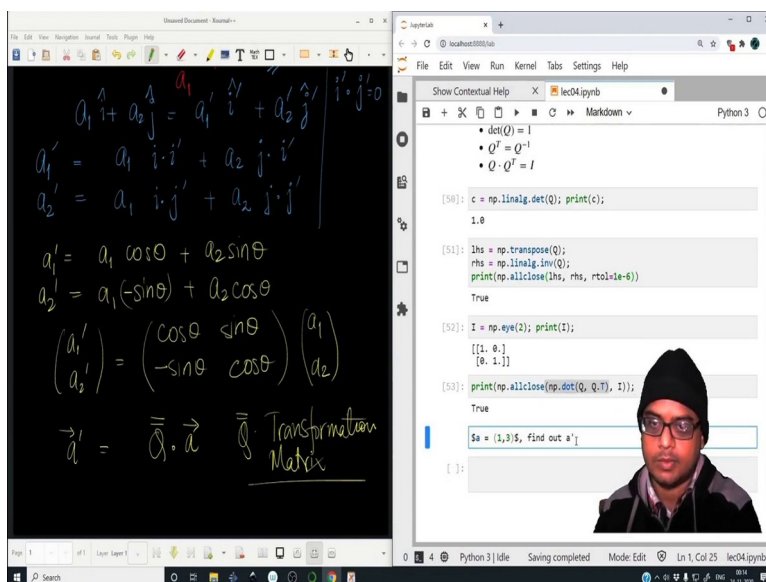
So, here are some properties which are quite well known from matrix algebra, I am just pointing them out for as a refresher nothing else. So, the first property is $|Q| = 1$. Second property is $Q^T = Q^{-1}$ and the third property is by corollary actually $Q \cdot Q^T = I$.

(Refer Slide Time: 35:32)



So, let us look at these three properties one by one. So, let us check whether the determinant of Q is 1 or not. So, let me define $c = \text{np.linalg.det}(Q)$. Let me print c and see the determinant of c is obviously, equal to 1 and that is quite obvious because the determinant of this fellow over here is $\cos^2 \theta - (-\sin^2 \theta)$ so, it is 1 so, identically equal to 1.

(Refer Slide Time: 36:19)



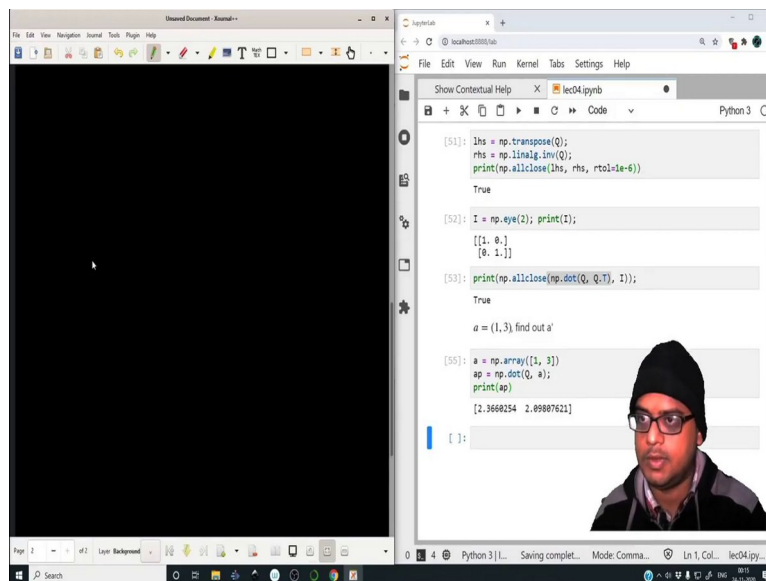
Let us check for the second identity. So, $\text{lhs} = \text{np.transpose}(Q)$, $\text{rhs} = \text{np.linalg.inv}(Q)$. So, then let us print $\text{np.allclose}(\text{lhs}, \text{rhs}, \text{rtol}=1e^{-6})$ so, it is true. In fact, in order to check whether this particular identity is true or not, we can make use of a function called as I 's or I 's not I 's, but I 's.

So, let me define it is called as I is in MATLAB, but in number it is I. So, let me define $I = \text{np.eye}(2)$. Let me print out what I is. So, I is an identity matrix. So, $\text{np.eye}(2)$ gives you a diagonal 1 matrix every other element is set to 0.

So, let me do this directly. So, $\text{print}(\text{np.allclose}(\text{np.dot}(Q, Q.T), I))$ have to check whether this is close to I or not. So, it is in fact, equal to I. So, this last identity we have done using the single line and oftentimes nesting these functions may not be in the best interest for readability of the code. So, when you write such a short code, it often ends up confusing the other reader.

So, in this particular case, it is not so complicated and so, we can do it like this, but in other case, it is expedient to define this as some other variable and then check this alright. So, now let us consider a having two components like this. So, let $a = (1, 3)$ for example. So, find out a' alright.

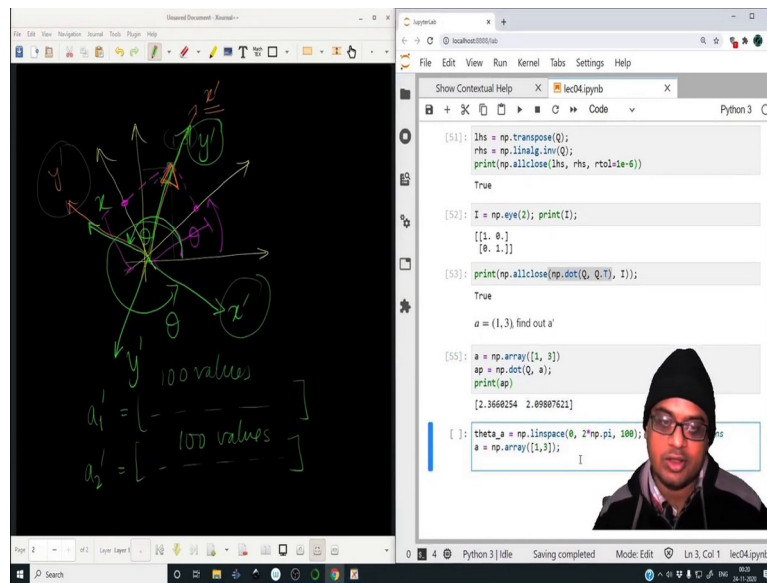
(Refer Slide Time: 39:15)



```
File Edit View Run Kernel Tabs Settings Help
Show Contextual Help X lec04.ipynb Python 3
[51]: lhs = np.transpose(Q);
      rhs = np.linalg.inv(Q);
      print(np.allclose(lhs, rhs, rtol=1e-6))
True
[52]: I = np.eye(2); print(I);
[[1.  0.]
 [0.  1.]]
[53]: print(np.allclose(np.dot(Q, Q.T), I));
True
a = (1, 3) find out a'
[55]: a = np.array([1, 3])
      ap = np.dot(Q, a);
      print(ap)
[2.3668254 2.89887621]
[ ]:
```

So, the components of a' will be equal to so, $ap = \text{np.dot}(Q, a)$, but we have not yet defined what a is. So, $a = \text{np.array}([1, 3])$ so, it has to have two components; first component is 1, well the second component is 3 ok. So, let us print out what the components of ap are. So, these are the two components of ap. So, the angle is 30 degree. So, how does this look like? Let us try to draw this I mean.

(Refer Slide Time: 40:11)



So, we have a system like this and we have rotated by 30 degree something close to this and the first it was (1, 3) alright or something like this. So, it was (1, 3) in this coordinate system and it does look to be something equal in the other coordinate system. So, over here it is something 2.366 and 2.099 I mean physically it makes sense ok.

So, let us in fact, do full rotation sweep and see how the components vary. I mean to say in a loop we can vary this θ going from 0 to a certain to all the way to 360° and we will try to find out what the components will be. So, in fact, there will be a certain angle; there will be a certain angle.

So, when the coordinate system is something like this, there will be no y component the y component is 0 because the vector is aligned completely with the x' coordinate and there will be an angle so, when the coordinate system appears to be something like this; something like this so, this will be y' and this will be x' . So, when the coordinate system would have rotated by this particular angle, we would have only a y' component which would be non-zero and x' would be 0.

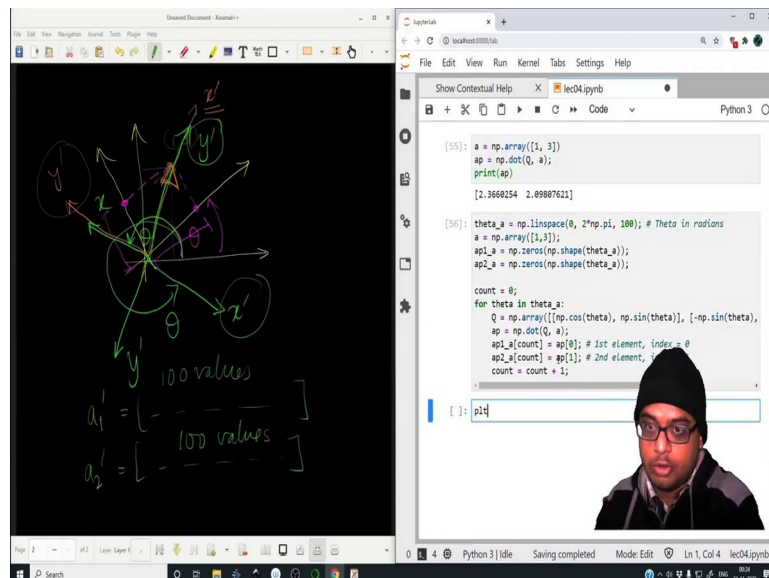
In fact, this would occur also when this will be y' and this will be x. So, when the coordinate system rotates by this angle ok. Try to draw this on your own and you will see what I mean. So, let us do a sweep of θ and find out the two components of the transformed vector a' ok.

So, let us do that. So, let us create the θ array. So, theta equal to np or let me call it theta a to signify that it is an array. So, it will be $\theta_a = \text{np.linspace}(0, 2*\text{np.pi}, 100)$. So, this is θ in radians.

So, let me create the vector v or let us use it, let us use the vector a. So, $a = \text{np.array}([1,3])$ let me redefine it just for completeness. So, now, we will have to run in a loop. So, in order to store each element of the transformed vector; so, basically, we need to have two arrays one of which will store all the components of a'_1 for all different θ and one which will store all the components of a'_2 for all the different θ .

So, there will be 100 values of θ as defined in this linspace and there will be 100 values of a'_1 as well as defined by this linspace. So, let us create two empty arrays. So, it is always a good idea to initialize; it is always a good idea to initialize whatever you are going to store instead of dynamically allocating an array, it is always a better idea to pre allocate it in terms of 0s.

(Refer Slide Time: 44:37)



So, we will declare v'_1 array is equal to $\text{np.zeros}(\text{np.shape}(\theta_a))$. It means it will create a 0 array which has the same shape as the θ array. This is important because θ array and a'_1 vp

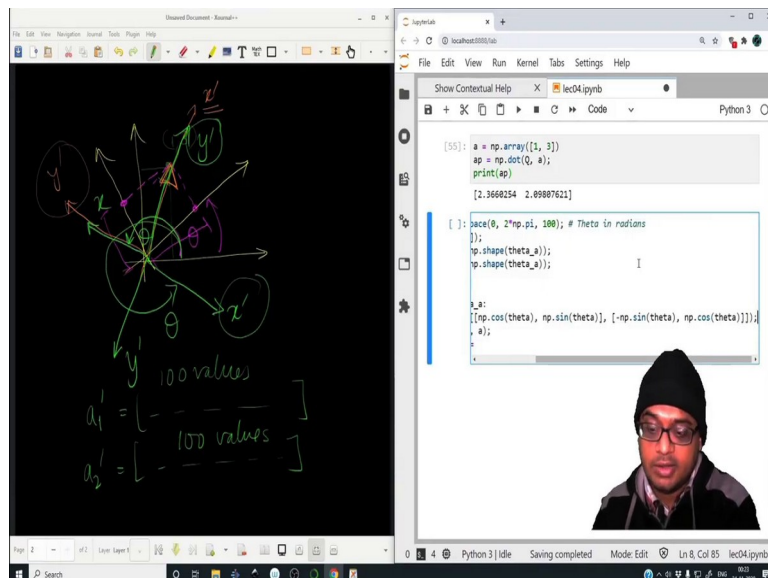
I am calling it v let me call it a . So, a_1' array and θ array must have the same length and hence, they must have the same dimension as well ok.

Similarly, `ap2_a = np.zeros(np.shape(theta_a))`. So, this ensures that if I change the number of elements in `theta_a`, it will automatically change the number of elements in `ap1_a` and `ap2_a`. So, I do not have to hard code the size of `ap1_a` and `ap2_a` ok. Instead of hard coding it, I am querying the size that these two arrays should be from the earlier declared array of θ ok.

So, now, let we can go in a loop. So, for θ in `theta_a` so, `theta` will pick up each element of `theta` in the loop alright `ap1_a` and now, we have to create a counter as well because for the first loop it has to have a value of a 0; `ap1` 0, then 2, 1, 2, 3, 4 and so on till 99 so, this will be count where I have to first define `count` outside the loop as 0.

So, `ap1` count is equal to it will be equal to in fact, let me do this `ap = np.dot(Q, a)` where `Q = np.array([[np.cos(θ), np.sin(θ)], [-np.sin(θ), np.cos(θ)]])` and we have to wrap it up with a another square bracket alright.

(Refer Slide Time: 47:26)

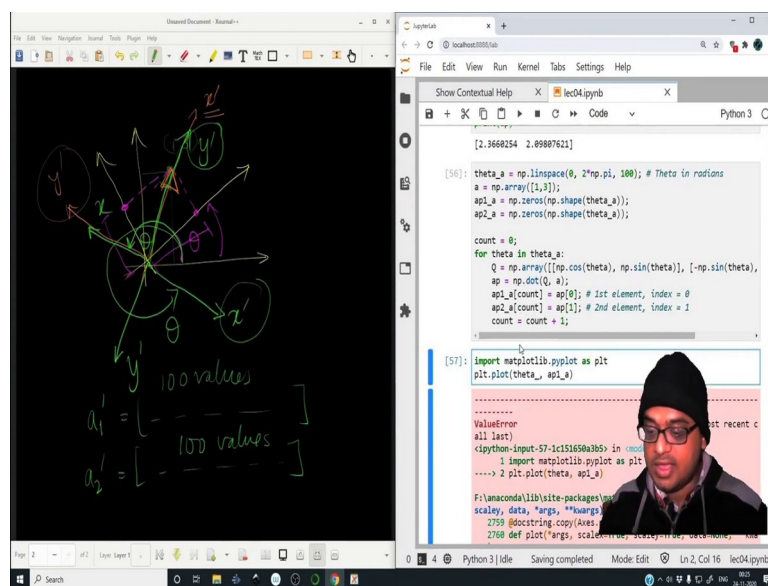


So, we have defined the transformation matrix a in Q inside the loop because θ is changing so, each time θ changes the matrix, Q has to change as well. a is defined a is not going to change. So, now, `ap = np.dot(Q, a)` and now, so, the first element that we will save it will be

this, while the second element it will be this ok. Remember that the element number and the index that differ by 1. So, this is the 2nd element, but the index is 1. Remember that this is 1st element, but the index is 0 ok.

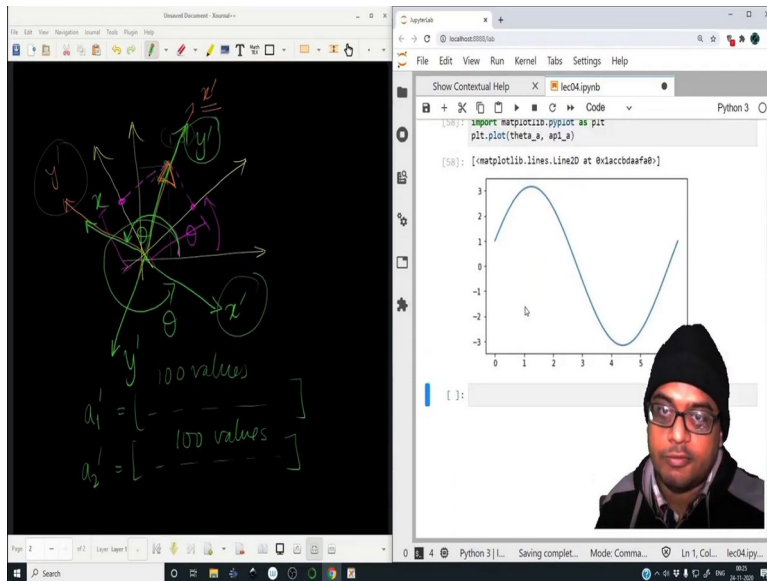
So, now, we have stored. So, when count is 0, we have stored it in ap1 underscore is 0 and ap2 underscores underscore 0. Once this loop has executed till this point, we need to increment the value of count. So, count will become count plus 1 alright. So, let us evaluate this entire cell and see what happens ok.

(Refer Slide Time: 48:58)



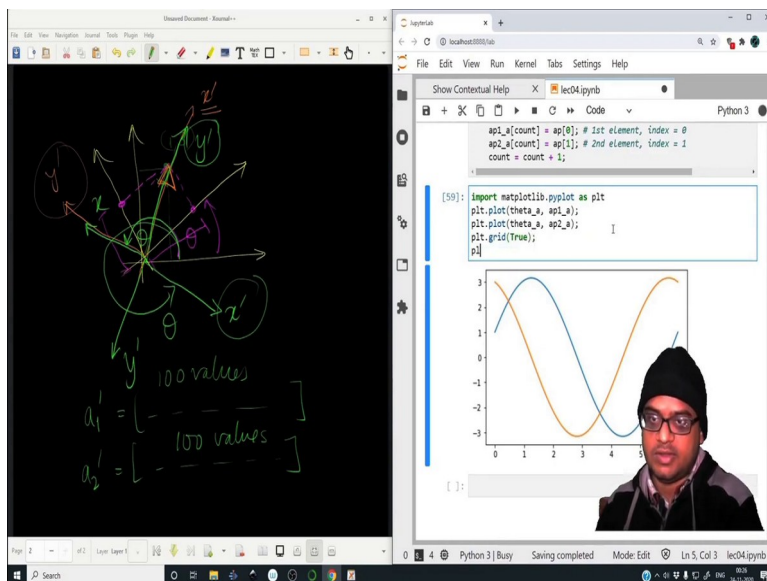
So, now let us plot. So, plt so, first we have to import the plot functions. So, import matplotlib.pyplot as plt, plt.plot(ap1_a) and on the x axis, we will have θ ok. This seems to be an error. So, it has to be theta_a because θ is just one of the many entries in theta_a where theta_a is the array alright.

(Refer Slide Time: 49:48)



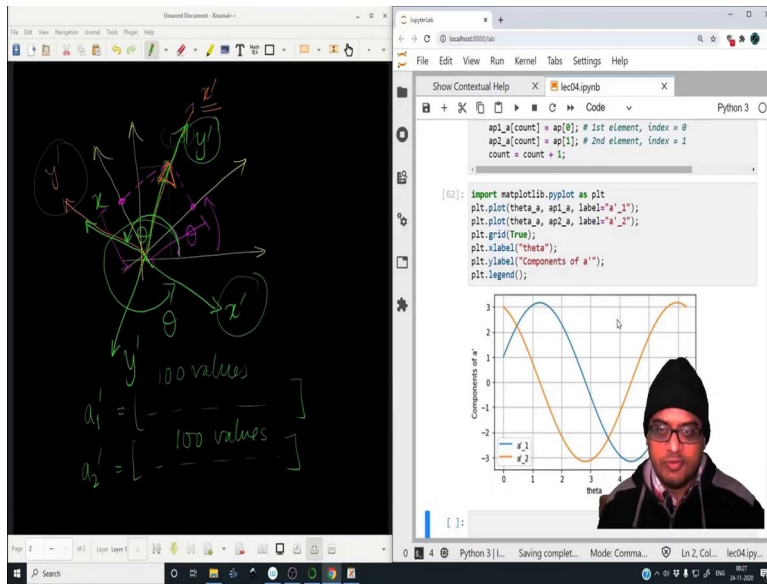
So, this is how the entry, how the plot of $ap1_a$ looks like.

(Refer Slide Time: 50:01)



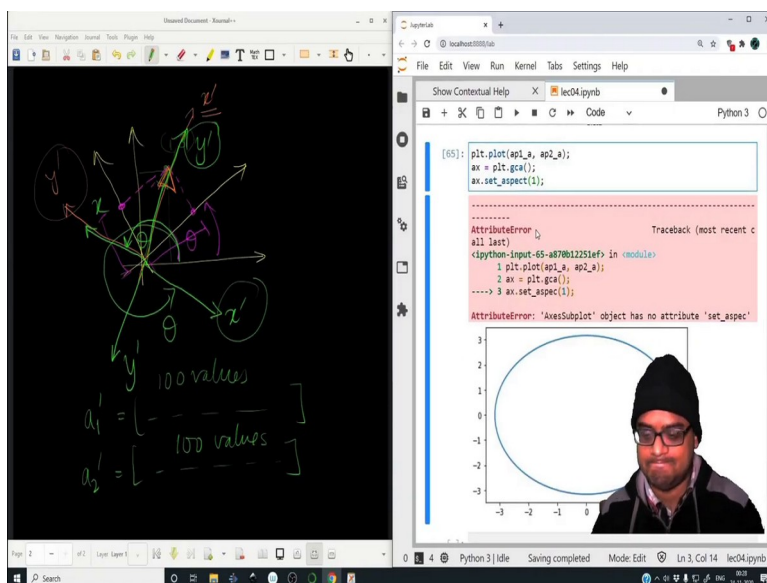
In fact, let us plot on top of this what $theta_a$, $ap2_a$ looks like alright. So, it looks something like this. Let us draw the grid let me set x levels and y levels.

(Refer Slide Time: 50:14)



So, now, let us see there does exist an angle where the let me label this two as well this is just to make the legends of the plot alright. So, there are points where the first component that is a_1' is maximum and that corresponds to a $a_2' = 0$ and that happens twice similarly, there are points where $a_1' = 0$ over here and where a_2' is negative minimum and positive maximum alright. So, the magnitude is maximum and that does correspond to these four angles that we have talked about in the working out of this problem. So, this is how you can do that.

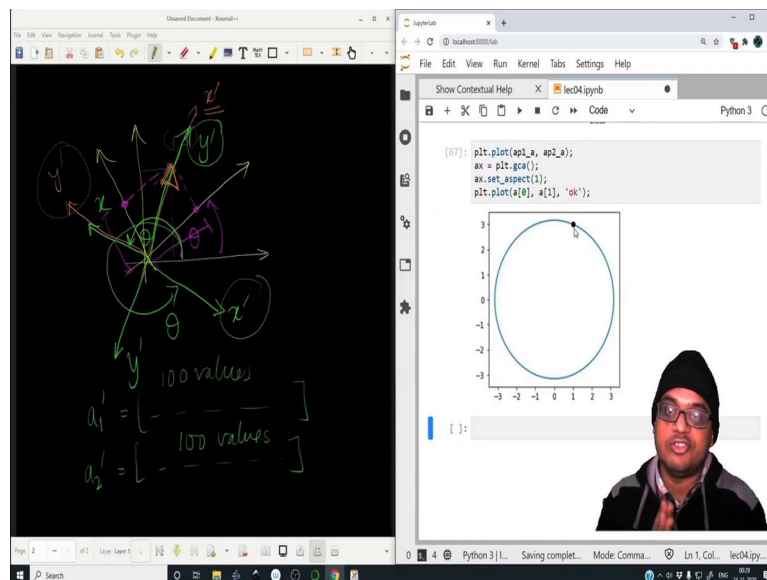
(Refer Slide Time: 51:34)



In fact, let us draw a phase plot of a' meaning plt.plot we will plot how a_1' and a_2' vary simultaneously, it is like a parametric plot. So, θ is like the parameter, but I want to plot a_2' versus a_1' . So, we will plot ap1_a and ap2_a . So, this is ap1 .

One should always be careful with the variables. So, it looks like a circle. In fact, let me make the aspect ratio of the plot better. So, we will get first the access. So, plt.gca , then we will set aspect ratio to be 1.

(Refer Slide Time: 52:34)



So, the parametric plot it does look like a circle in the sense that all the components lie on this particular circle. So, if I select this particular point alright so, it corresponds to some angle θ . In fact, let me superpose on top of this the original components. So, the original components were this ok. So, let me superpose that particular point as well.

So, plt.plot so, this will be $a[0]$, $a[1]$ and let me put a circle, black colored circle over there. So, this is the point where we have started the original vector and as you rotate it, you will change the components a_1' and a_2' in accordance to this particular circle ok. Once you rotate by a certain angle, you will reach this particular point and these the projections will be the components of a_1' and a_2' this is what it means.

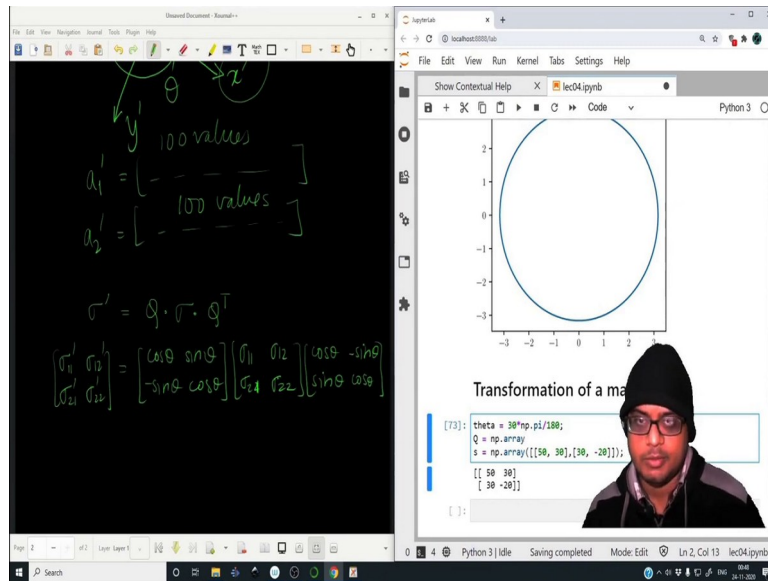
(Refer Slide Time: 54:13)

The image shows a split-screen view. On the left is a blackboard with handwritten notes in green and white. The notes include: θ (with arrows pointing to y and z axes), $a_1 = [\dots]$ and $a_2 = [\dots]$ (both with "100 values" written above them), the equation $\sigma' = Q \cdot \sigma \cdot Q^T$, and a matrix transformation:
$$\begin{bmatrix} \sigma'_{11} & \sigma'_{12} \\ \sigma'_{21} & \sigma'_{22} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$
 On the right is a JupyterLab interface. It shows a plot of a circle on a coordinate system with axes from -3 to 3. Below the plot is the text "Transformation of a matrix" and a code cell containing `s = np.array([[50, 30], [30, -20]])`. A small video inset of a man in a black beanie is visible in the bottom right corner of the JupyterLab window.

So, now once we are done with this, we are ready to move into transforming stress ok. So, now, let me just mention how a matrix transformation looks like. So, if we have a matrix σ so, the components of σ' will be $Q \cdot \sigma \cdot Q^T$. So, this is how the transformation of a matrix looks like and so, in its rawest form, this will be equal to so, it will be equal to this.

So, let us do this. Let us so, let me define a matrix s is equal to so, let me define it as $[50, 30], [30, -20]$. So, typically the tensors so, there is a difference between a tensor and a matrix, but this is not the course to discuss about that, but typically it is symmetric. So, let us print out what s is.

(Refer Slide Time: 55:43)

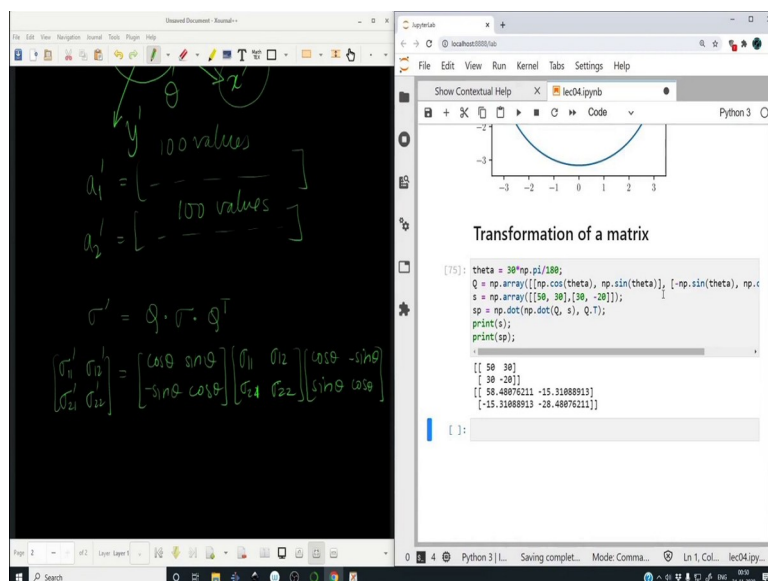


So, s is this matrix ok. So, let me remove this print, let me define what θ will be. So, θ will be say some rotation by 30 degrees. So, the coordinate system rotates by an angle 30. So,

$\frac{30\pi}{180}$

. So, the Q , the transformation matrix will be np.array in fact, let me copy it from above.

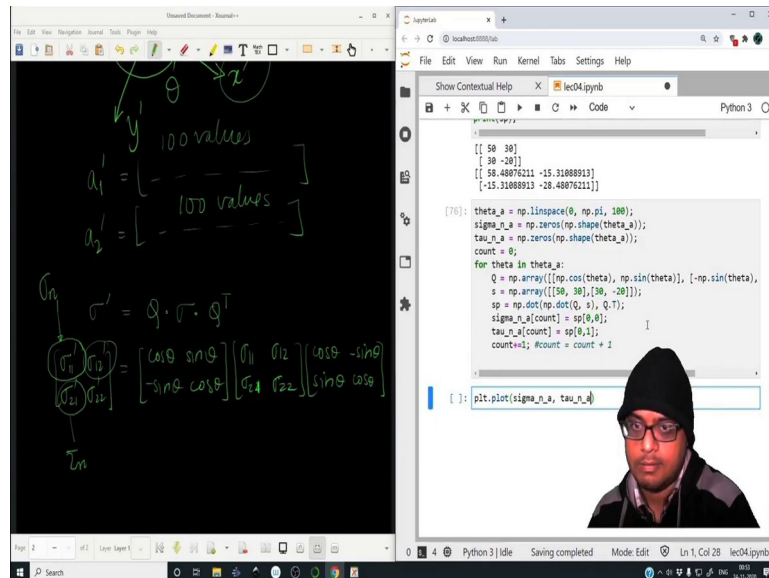
(Refer Slide Time: 56:35)



So, now with the help of this, we can find out the transformed entity. So, let the transformed matrix be sp . So, it will be $\text{np.dot}(\text{np.dot}(Q, s), Q.T)$. So, let us print out s and let

us print out sp. So, those are the transformed elements of the matrix s when the coordinate system is rotated by an angle 30 ok.

(Refer Slide Time: 57:56)



In fact, let us do the same procedure over here. Let us try to loop over all the thetas and find out how the components change alright. So, let us reuse this bit of program and let us try to convert this into a loop. So, θ first of all we have to declare as an array. Let us define it till only 180 degree i.e. $\text{theta_a} = \text{np.linspace}(0, \text{np.pi}, 100)$ and it will be clear why that is the case sorry ok.

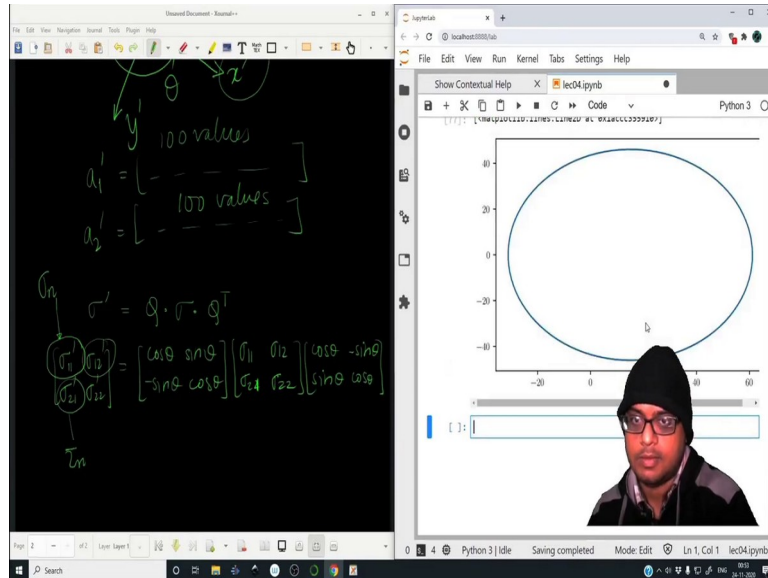
In order to save the different components of the transformed matrix, let us declare a few more arrays. So, let us define $\text{sigma_n_a} = \text{np.zeros}(\text{np.shape}(\text{theta_a}))$ and let us define $\text{tau_n_a} = \text{np.zeros}(\text{np.shape}(\text{theta_a}))$ and again we can do the same thing, we can declare everything inside a loop for θ in theta_a .

So, now we must indent these lines because these have to be executed inside a loop and we must have a loop counter as well so, count equal to 0. So, now, let us set $\text{sigma_n_a}[\text{count}] = \text{sp}[0,0]$ that is this particular element we are declaring as sigma_n , these elements we are declaring as tau_n . So, these are some of the notation's that solid mechanics that appear in solid mechanics so, alright.

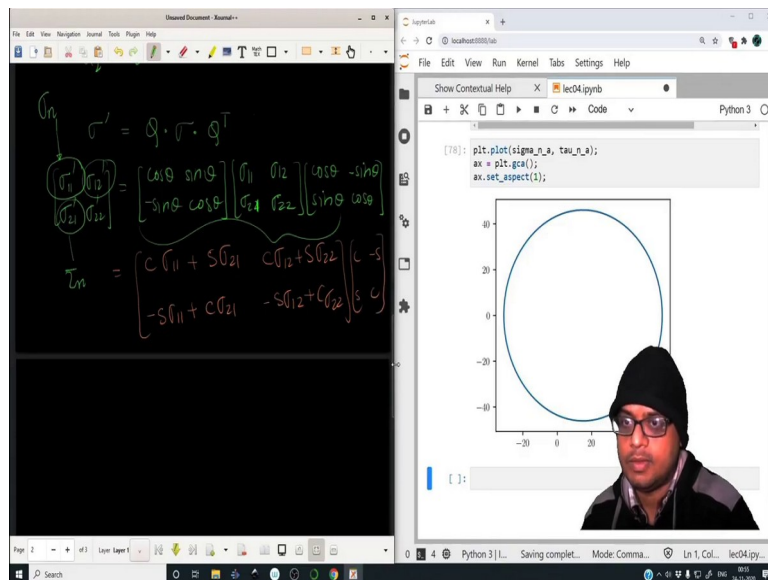
So, let us before this, we have to increment count as well. So, one way of doing this is $\text{count} += 1$. So, this is equivalent to writing $\text{count} = \text{count} + 1$ ok. So, this is an equivalent way

of writing it. So, let us execute this and see whether so, there is no error. So, great we can now plot σ_n as a function of τ_n .

(Refer Slide Time: 61:01)



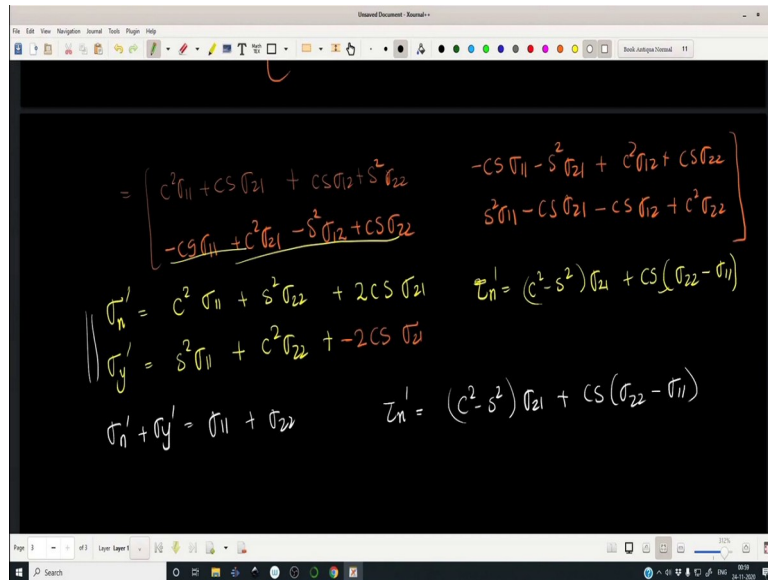
(Refer Slide Time: 61:06)



Let me make the aspect ratio. So, it does look like a circle, but can we show it, can we prove that it does actually have to be a circle and the circle is not centered about (0, 0), it is centered about some other value. So, what is that value? Ok. So, let us quickly simplify whatever we were doing over here.

So, let me simplify whatever this is. So, let us simplify this. So, instead of $\cos(\theta)$ and $\sin(\theta)$, I am just going to write C and S; so, this will be $C\sigma_{11} + S\sigma_{21}$, $C\sigma_{12} + S\sigma_{22}$, $-S\sigma_{11} + C\sigma_{21}$, $-S\sigma_{12} + C\sigma_{22}$ this times C, -S, S, C alright.

(Refer Slide Time: 62:17)



So, this will be equal to so, this multiplies this. So,

$$\begin{pmatrix} C^2\sigma_{11} + CS\sigma_{21} + CS\sigma_{12} + S^2\sigma_{22} & -CS\sigma_{11} - S^2\sigma_{21} + C^2\sigma_{12} + CS\sigma_{22} \\ -CS\sigma_{11} + C^2\sigma_{21} - S^2\sigma_{12} + CS\sigma_{22} & S^2\sigma_{11} - CS\sigma_{21} - CS\sigma_{12} + C^2\sigma_{22} \end{pmatrix}$$

So, these are the

components. And so, $\sigma'_x = C^2\sigma_{11} + S^2\sigma_{22} + 2CS\sigma_{21}$ so, this become because we have selected

the σ to be symmetric. So, this is $2CS\sigma_{21}$, τ'_x is equal to this term so, this is so, σ_{21} they are

equal alright so, this is $\tau'_x = (C^2 - S^2)\sigma_{21} + CS(\sigma_{22} - \sigma_{11})$ and if you note that this element is

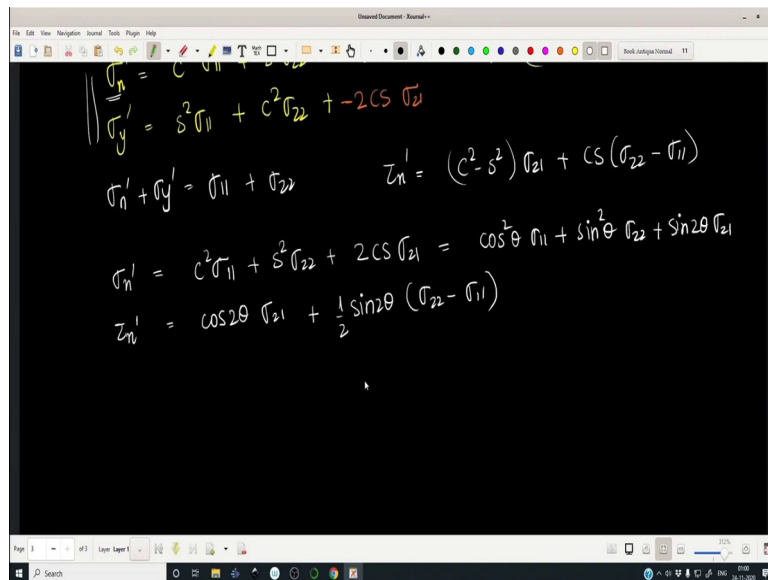
also equal to this $\sigma_{22} CS - \sigma_{11}$ so, it checks out so, its symmetric as well the transform matrix is symmetric as well.

While if I call this as σ_y' , this comes out to be $\sigma_y' = S^2\sigma_{11} + C^2\sigma_{22}$ so obviously, we have made a small mistake. So, this should have been this times this, this should have been $-CS$ ok.

So, this is $-2CS\sigma_{21}$.

So, one thing to note from this is that if you add these two, $\sigma_n' + \sigma_y'$ so, this becomes $\sigma_{11} + \sigma_{22}$ because $\cos^2(\theta) + \sin^2(\theta) = 1$, these two terms cancel out while, $\sigma_n' = \cos^2(\theta)\sigma_{11} + \sin^2(\theta)\sigma_{22} + \sin(2\theta)\sigma_{21}$ ok.

(Refer Slide Time: 66:22)



$$\sigma_y' = S^2\sigma_{11} + C^2\sigma_{22} - 2CS\sigma_{21}$$

$$\sigma_n' + \sigma_y' = \sigma_{11} + \sigma_{22} \quad \tau_n' = (C^2 - S^2)\tau_{21} + CS(\sigma_{22} - \sigma_{11})$$

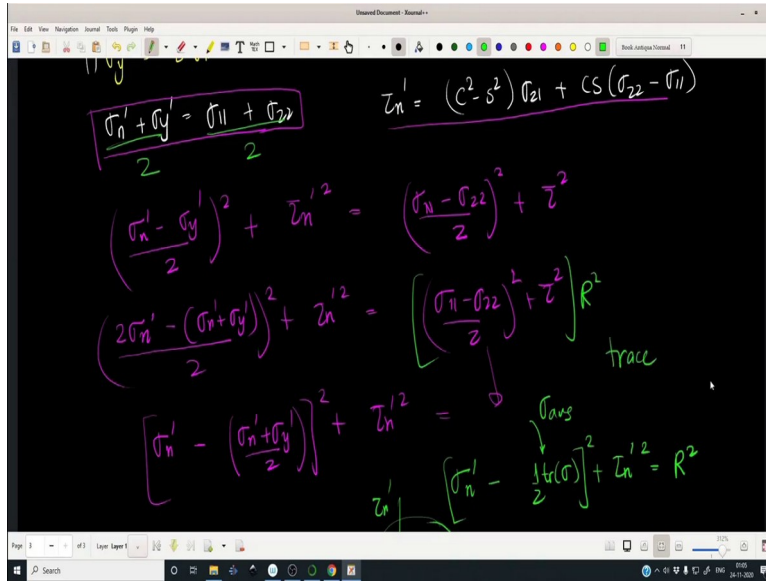
$$\sigma_n' = C^2\sigma_{11} + S^2\sigma_{22} + 2CS\tau_{21} = \cos^2\theta\sigma_{11} + \sin^2\theta\sigma_{22} + \sin 2\theta\tau_{21}$$

$$\tau_n' = \cos 2\theta\tau_{21} + \frac{1}{2}\sin 2\theta(\sigma_{22} - \sigma_{11})$$

But if we just focus on this sigma n prime so, $\sigma_n' = C^2\sigma_{11} + S^2\sigma_{22} + 2CS\sigma_{21}$; while,

$\tau_n' = \cos(2\theta)\sigma_{21} + \frac{1}{2}\sin(2\theta)(\sigma_{22} - \sigma_{11})$. While $\sigma_n' = \cos^2(\theta)\sigma_{11} + \sin^2(\theta)\sigma_{22} + \sin(2\theta)\sigma_{21}$.

(Refer Slide Time: 67:47)



So, keeping this in mind, we can further simplify everything and we can show that

$$\left(\frac{\sigma'_n - \sigma'_y}{2}\right)^2 + \tau_n'^2 = \left(\frac{\sigma_{11} - \sigma_{22}}{2}\right)^2 + \tau^2$$

. This can be shown using these expressions and with

the help of this, we can finally write down an expression like this σ'_n . So, let us add and

subtract σ'_n . So, if we add and subtract over here so, this becomes

$$\left(\frac{2\sigma'_n - (\sigma'_n + \sigma'_y)}{2}\right)^2 + \tau_n'^2 = \left(\frac{\sigma_{11} - \sigma_{22}}{2}\right)^2 + \tau^2$$

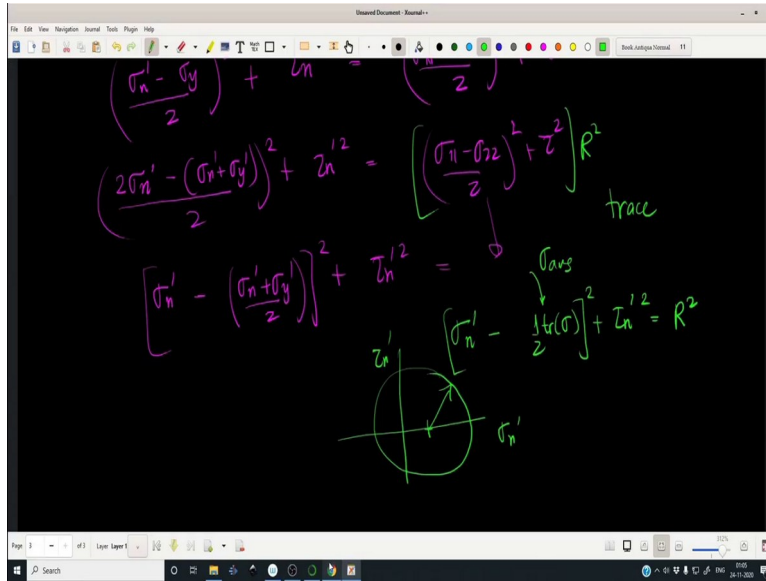
and this essentially means

$$\left(\sigma'_n - \frac{(\sigma'_n + \sigma'_y)}{2}\right)^2 + \tau_n'^2 = \left(\frac{\sigma_{11} - \sigma_{22}}{2}\right)^2 + \tau^2$$

And this implies and this particular earlier result when we combine so, this implies that

$$\left(\sigma'_n - \frac{1}{2}\text{tr}(\sigma)\right)^2 + \tau_n'^2 = R^2$$

(Refer Slide Time: 69:48)



So, it's clear that the locus or rather the curve in the σ'_n, τ'_n plane will be a circle whose

center is at this point that is the σ_{avg} , that is $\sigma_{avg} = \frac{\sigma'_n + \sigma'_y}{2}$ and I call it the average because

$$\frac{\sigma'_n + \sigma'_y}{2} = \frac{\sigma_{11} + \sigma_{22}}{2}$$

Hence, I have written it as $\frac{1}{2} \text{tr}(\sigma)$ because the trace is simply the sum of diagonals and the radius of this circle is equal to this entity ok. So, let us verify whether that is true; let us verify whether that is true.

(Refer Slide Time: 70:41)

tkgp.ac.in/~adityab/lecture_list.html as a quick reference

So, on top of this plot, let me plot the center of the circle `plt.plot` so, let me so, the x; x point will be half of the trace so, `plt.plot(np.trace(s)/2, 0, 'ok')`. So, this is indeed the center and that is simply the trace of s and we do not need the transform matrix to find out the location of the center ok.

So, on the radius of this circle, you can independently verify that it will be equal to this equation over here and this particular construction is called as the Mohr circle and it helps us in identifying which will be the coordinate system in which we will experience maximum shear stress or the maximum normal stress. So, on the x axis, we have essentially the maximum another the transform normal stress, on the y axis, we have the maximum shear stress ok.

So, let me label this. So, the x-label will be equal to sigma n prime while the we have to put a double escape character over here and `plt.ylabel("τ_n'")` ok. So, this helps us in identifying the locations of the maximum shear stress and normal stress.

And I request you to have a look at basic solid mechanics and try to figure out all this on your own. There will be n number of problems where you are required to find out the plane where the maximum stress occurs and try to resolve it using Python, I have shown you how to do it.

So, this pretty much wraps up this particular lecture and, in this lecture, we have covered a whole lot of matrix manipulations and I have shown you how it can be used to your

advantage to understand matrix transformations and vector transformations. This is different from vector rotations where you are rotating a vector. Here, we are rotating coordinate systems and in your free time, I request you to have a look into rotation of vectors and rotation of matrices not just the transformations.

With this it is goodbye from me, I will see you again next time bye.