

Computational Mathematics with SageMath
Prof. Ajit Kumar
Department of Mathematics
Institute of Chemical Technology, Mumbai

Inner Product Spaces cont...
Lecture – 38
Inner Product Part 2 with SageMath

(Refer Slide Time: 00:32)

Inner Product Spaces cont...

Orthogonal sets of vectors

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space and $S = \{v_1, \dots, v_k\} \subset V$. Then S is called orthogonal set if $\langle v_i, v_j \rangle = 0$ for $i \neq j$.

- $S = \{e_1, \dots, e_n\}$, set of unit vectors in \mathbb{R}^n along the coordinate axes is an orthogonal set with respect to the standard dot product on \mathbb{R}^n .
- The set $\{f_1 = 1, f_2 = x - 1/2, f_3 = x^2 - 3/2x + 3/5, f_4 = x^2 - x + 1/6\}$ is orthogonal set in $C[0, 1]$ with respect to the inner product $\langle f, g \rangle := \int_0^1 f(x)g(x)dx$.
- The set $\{(1, -1, 1), (\frac{3}{2}, \frac{3}{2}, \frac{3}{2}), (-\frac{1}{2}, 0, \frac{1}{2})\}$ is orthogonal set in \mathbb{R}^3 with respect to the inner product $\langle x, y \rangle := y^T A x$, where $A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$

Welcome to the 38th lecture on Computational Mathematics with SageMath. Let us continue exploring some more concepts in Inner Product Spaces.

Let us define, what is meaning of an orthogonal set of vectors in an inner product space. Suppose V is an inner product space and S , a subset of V containing vector v_1, v_2, v_k , this could be infinite subset as well. Then we say that S is an orthogonal set if inner product of v_i with v_j is 0 whenever i is not equal to j . That simply means, if you take any two distinct elements in S they should be orthogonal. Such a set is called an orthogonal set.

For example, if you look at the standard unit vectors e_1, e_2, e_n with respect to the usual dot product they are orthogonal set of vectors. In fact, each of this vector has norm 1. Such a set is also known as orthonormal set.

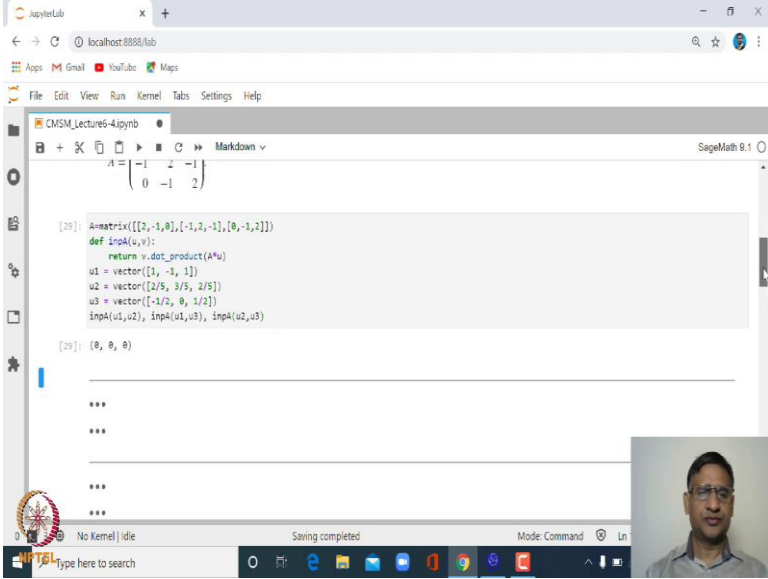
Similarly, f_1 is equal to 1, f_2 is equal to x minus half, f_3 is equal to x to the power 3 minus 3 by 2 x square plus 3 by 5 x minus 1 by 20 and f_4 is equal to x square minus x plus 1 by 6.

You can check that this is an orthogonal set with respect to the inner product defined on $C[0, 1]$, which is inner product $f g$ equals to integral of $f(x)$ into $g(x)$ from 0 to 1.

So, it is easy to check that this is an orthogonal set.

Similarly, you look at set of these three vectors 1, minus 1, 1, 2 by 5, 3 by 5, 2 by 5 minus half, 0, half and define inner product on R^3 as inner product x, y is equal to y transpose Ax , where A is this matrix. This is symmetric positive definite matrix. This example we have dealt with earlier. So, you can check that this given set of vectors are orthogonal.

(Refer Slide Time: 02:50)



```

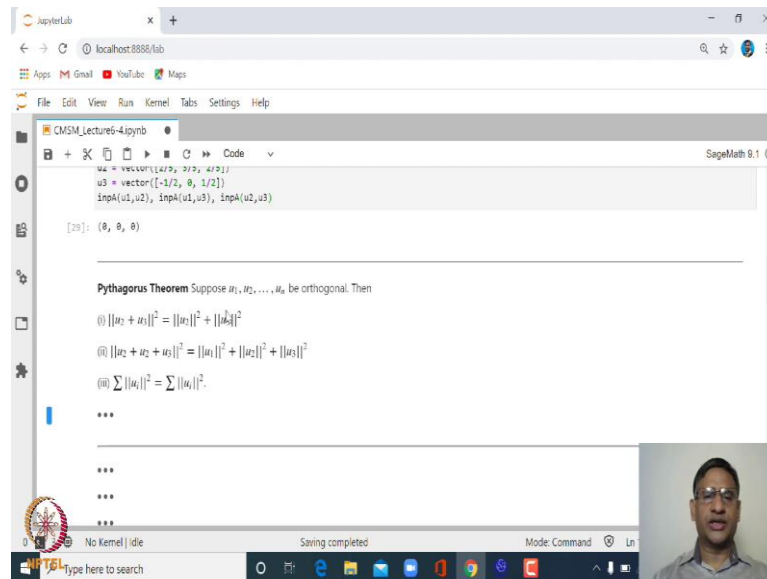
A = matrix([[2, -1, 0], [-1, 2, -1], [0, -1, 2]])
def inner(u, v):
    return v.dot_product(A*u)
u1 = vector([1, -1, 1])
u2 = vector([2/5, 3/5, 2/5])
u3 = vector([-1/2, 0, 1/2])
inner(u1, u2), inner(u1, u3), inner(u2, u3)

```

Output: (0, 0, 0)

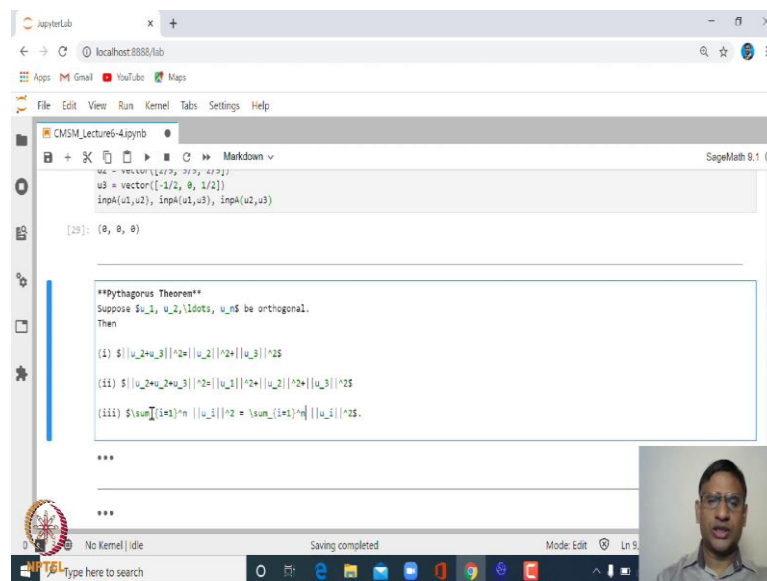
Let us check that. Define a matrix A and inner product this we have done already and two three vectors u_1, u_2, u_3 and let us check inner product a of u_1 with u_2 , u_1 with u_3 and u_2 with u_3 , all of this should give you answer 0, That is, correct. So these three vectors with respect to this inner product are orthogonal to each other and therefore, this set of vectors is an orthogonal set. Similarly, you can check the previous one. I will leave that as an exercise.

(Refer Slide Time: 03:31)



You can also verify the Pythagoras theorem.

(Refer Slide Time: 03:40)

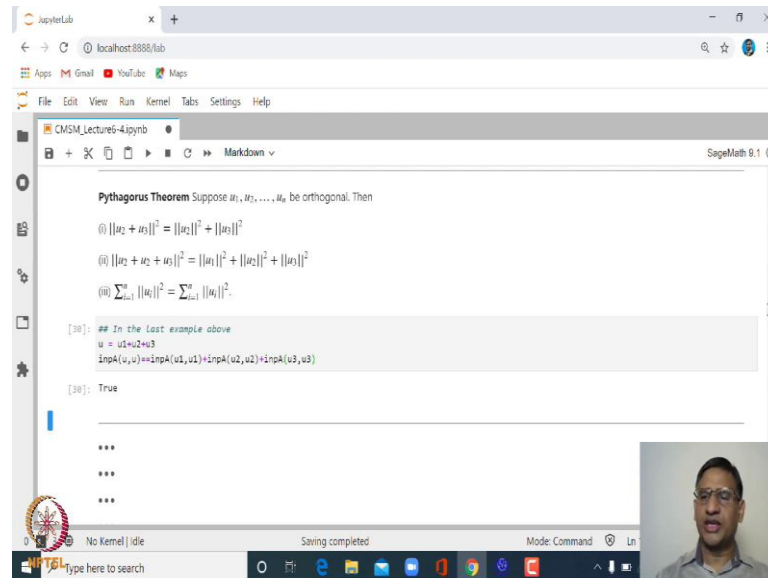


Suppose, you have vectors u_1, u_2, u_n which are orthogonal to each other and then if you look at for example, u_2 plus u_3 the norm square this is same as norm of u_1 square plus norm of u_2 square. This is, if u_1, u_2 are orthogonal, u_1, u_2 forms a right angle triangle.

This says that hypotenuse square is equal to base square plus height square and you can extend this to three vectors, u_1, u_2, u_3 norm square is equal to u_1 norm square plus u_2 norm square plus u_3 norm square. You can extend this to any set of finite vectors. This

is only for the finite set of vectors. Let me put here i is equal to 1 to n . So, also i is equal to 1 to n . This n is any natural number.

(Refer Slide Time: 04:53)



```

Pythagorus Theorem Suppose  $u_1, u_2, \dots, u_n$  be orthogonal. Then

(i)  $\|u_1 + u_2\|^2 = \|u_1\|^2 + \|u_2\|^2$ 
(ii)  $\|u_1 + u_2 + u_3\|^2 = \|u_1\|^2 + \|u_2\|^2 + \|u_3\|^2$ 
(iii)  $\sum_{i=1}^n \|u_i\|^2 = \left\| \sum_{i=1}^n u_i \right\|^2$ 

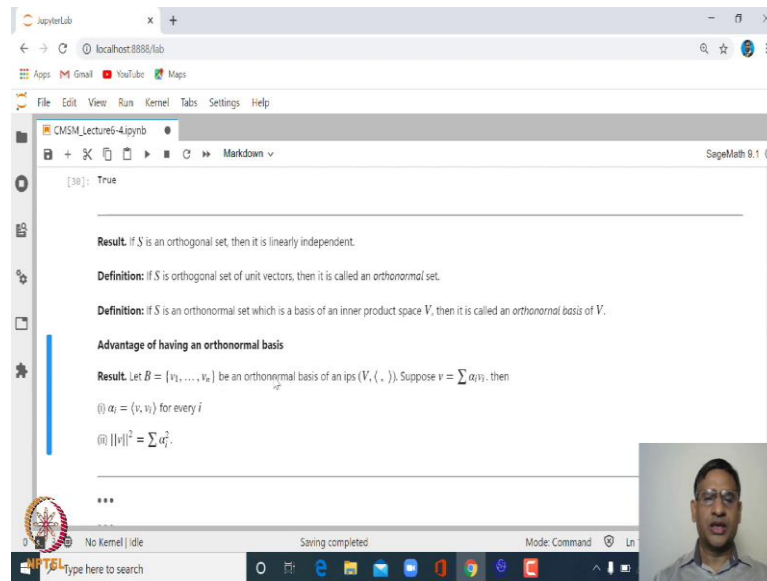
[38]: ## In the last example above
u = u1+u2+u3
inp4(u,u)==inp4(u1,u1)+inp4(u2,u2)+inp4(u3,u3)

[38]: True

```

So, let us try to verify this Pythagoras theorem for the above example where you have u_1, u_2, u_3 and in \mathbb{R}^3 and this is the inner product. Let us define u to be u_1 plus u_2 plus u_3 and define the norm of u square is it equal to norm of u_1 square plus norm of u_2 square plus norm of u_3 square, you should get answer true, that is correct. So this we have verified for three vectors, but you can generalize this to any finitely many orthogonal set of vectors.

(Refer Slide Time: 05:33)



Now, let us look at, suppose you have S which is an orthogonal set then one can show that this is a linearly independent set of vectors. This is S is linearly independent.

So, any orthogonal set is linearly independent. In case, we have orthogonal set of unit vectors, it is going to be orthogonal and unit length, we call such a set as orthonormal set.

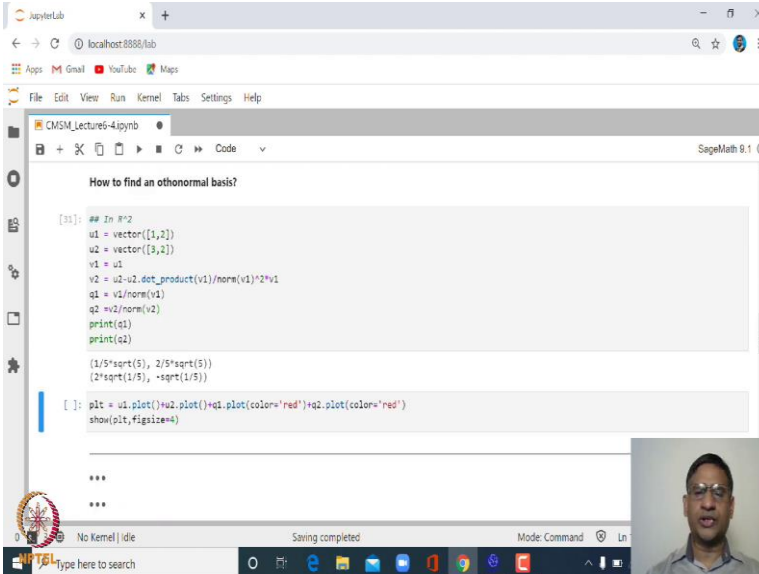
And, in case you have an orthonormal set which is also a basis of the inner product space V then such a basis we call as orthonormal basis.

Now, what is advantage of having an orthonormal basis? Let us just look at. Suppose you have a basis B containing v_1, v_2, \dots, v_n , an orthonormal basis of an inner product space V .

Then, if you take any vector v , we know that v can be written as linear combination of v_1, v_2, \dots, v_n and in that if $v = \sum_{i=1}^n \alpha_i v_i$ then α_i can be obtained as inner product of v with v_i . That is quite easy to verify because you can just take inner product with v on both sides. And, then you take inner product with v_i on both sides on the right hand side v inner product with v_j will be 0 when i is not equal to j . So, only surviving term will be α_i and α_i is v inner product with v_i and this is true for every i .

Not only that, norm of v square will be summation α_i^2 . So, that is the advantage. In case we have an orthonormal basis then finding coordinate of any vector with respect to that basis is quite easy. All you need to do is, you need to take inner product of the vector v with i th coordinate of the basis vector. So, that is the advantage. This is what exactly we had for standard basis in \mathbb{R}^n . If you have any vector x which is summation $x_i e_i$, then x_i is nothing but x inner product with e_i .

(Refer Slide Time: 08:10)



```

How to find an orthonormal basis?

[[1]]: ## In R^2
u1 = vector([1,2])
u2 = vector([3,2])
v1 = u1
v2 = u2 - u2.dot_product(v1)/norm(v1)*v1
q1 = v1/norm(v1)
q2 = v2/norm(v2)
print(q1)
print(q2)

(1/5*sqrt(5), 2/5*sqrt(5))
(2/5*sqrt(1/5), -sqrt(1/5))

[[ ]]: plt = u1.plot()+u2.plot()+q1.plot(color='red')+q2.plot(color='red')
show(plt,figsize=4)

***
***

```

So, now the question is how does one find an orthonormal basis? Having orthonormal basis helps, but can we find an orthonormal basis. We have seen how to find a basis starting with any nonzero vector we can extend that to a basis of a finite dimensional vector space.

However, how does one find an orthonormal basis? So, this is again easy. So, let us start with \mathbb{R}^2 . So, in \mathbb{R}^2 suppose you start two linearly independent vectors that is easy to find and then what we can do is we have seen that when we take orthogonal projection of a vector onto another vector that gives rise to a vector which is perpendicular to a given vector, right.

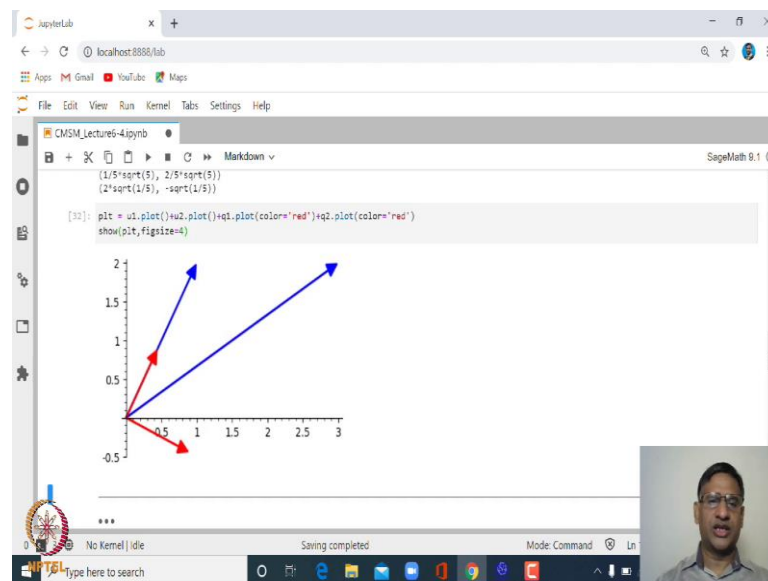
We will use that idea. So, we have two vectors, let us say, u_1 and u_2 . Then define v_1 is equal, to let us say, u_1 and for v_2 what you do? From u_2 you take out the orthogonal

projection of u_2 onto v_1 . Take out orthogonal projection of u_2 onto v_1 . And, then let us define q_1 to be v_1 upon norm v_1 and q_2 to be v_2 upon norm v_2 . You can check that these two vectors are orthogonal to each other.

This we have already checked. Only thing here we did was we made this as unit vector.

Now, let us try to plot graph of these two vectors u_1, u_2 which was given to vectors, and q_1, q_2 which we have obtained as orthonormal vectors.

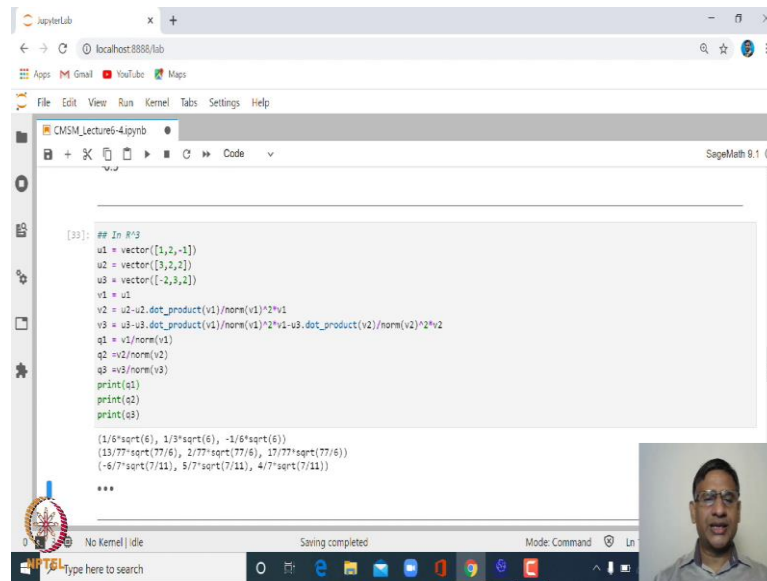
(Refer Slide Time: 09:58)



So, you can see here, this is actually u_1 , this is u_2 and v_1 we took as u_1 divided by norm u_1 . So, it is unit vector along this direction and if you take the perpendicular of u_2 on to u_1 and then take out that orthogonal component this is what you get. This is your q_2 , right.

So, this is how we can generate two or two set of vectors which are orthogonal to each other and also they are of unit length. Once you generate orthogonal set of vectors, then you just need to divide by its length to make it orthonormal.

(Refer Slide Time: 10:44)



```
[30]: ## In R^3
u1 = vector([1,2,-1])
u2 = vector([3,2,2])
u3 = vector([-2,3,2])
v1 = u1
v2 = u2 - u2.dot_product(v1)/norm(v1)^2*v1
v3 = u3 - u3.dot_product(v1)/norm(v1)^2*v1 - u3.dot_product(v2)/norm(v2)^2*v2
q1 = v1/norm(v1)
q2 = v2/norm(v2)
q3 = v3/norm(v3)
print(q1)
print(q2)
print(q3)

(1/6*sqrt(6), 1/3*sqrt(6), -1/6*sqrt(6))
(13/77*sqrt(77/6), 2/77*sqrt(77/6), 17/77*sqrt(77/6))
(-6/7*sqrt(7/11), 5/7*sqrt(7/11), 4/7*sqrt(7/11))
***
```

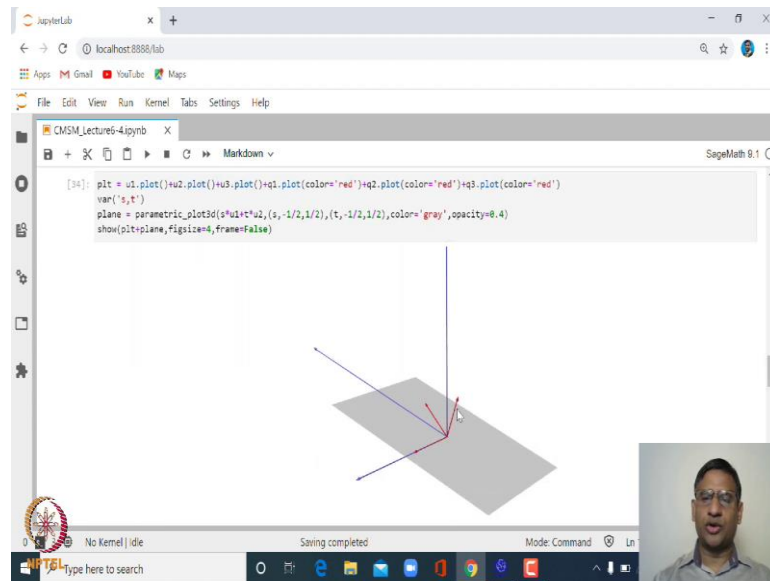
So, let us extend this in \mathbb{R}^3 . Suppose I have three vectors u_1, u_2, u_3 in \mathbb{R}^3 . Now we will do the same thing again. We start with v_1 is equal to u_1 and v_2 is a vector, how is it obtained?

Again, it is in exactly in a similar way that we obtained in the previous case. So, from u_2 , you take out the orthogonal component of orthogonal projection of u_2 onto v_1 . How do you define v_3 ?

To define v_3 , from u_3 you subtract orthogonal projection of u_3 onto v_1 and also subtract orthogonal projection of u_3 on to v_2 . And, then define q_1 is equal to norm v_1 upon v_1 upon norm v_1 q_2 to be v_2 upon norm v_2 and q_3 to be v_3 upon norm v_3 .

Let us look at, what are the these three vectors. You can check that these three vectors are orthogonal to each other and they are of unit length.

(Refer Slide Time: 11:49)

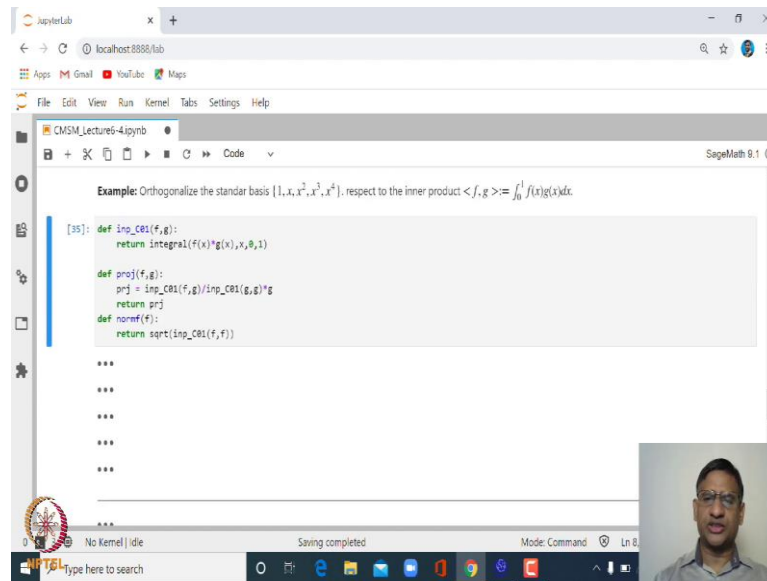


Now, again let us plot graph of these vectors, the graph of u_1 , u_2 , u_3 along with the orthonormal set of vectors that we have obtained.

So, in this case you can just check. Let me rotate little bit this. So, this may be u_1 , this may be u_2 and you have taken orthogonal projection of a u_2 upon u_1 and then you have taken out. So, this is your v_1 , this is your v_1 upon norm v_1 and this is v_2 upon norm v_2 , this is v_3 upon norm v_3 . So, this is q_1 , q_2 , q_3 .

And, so, what you can see here, this is the plane which is shown, is plane spanned by u_1 and u_2 and the vector q_3 is perpendicular to this plane. So, this is how you obtain three vectors which are orthonormal in \mathbb{R}^3 .

(Refer Slide Time: 13:06)



```

Example: Orthogonalize the stander basis [1, x, x^2, x^3, x^4], respect to the inner product <f, g> := \int_0^1 f(x)g(x)dx.

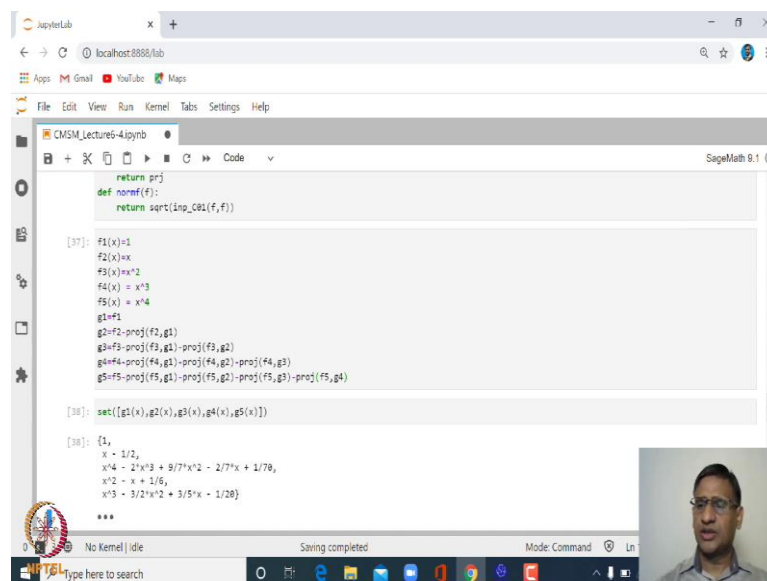
[35]: def inp_C01(f,g):
      return integral(f(x)*g(x),x,0,1)

      def proj(f,g):
      prj = inp_C01(f,g)/inp_C01(g,g)*g
      return prj
      def norm(f):
      return sqrt(inp_C01(f,f))

      ...
      ...
      ...
      ...
      ...
  
```

And, you can extend this idea to more vectors. So, for example, if I take four vectors 1, x, x square, x cube and x to the power 4 in the set of all polynomials of degree less than equal to 4 with respect to this inner product, we can extend this idea. So, first let us define inner product and the projection and the norm of vectors with respect to this inner product.

(Refer Slide Time: 13:31)



```

[37]: f1(x)=1
      f2(x)=x
      f3(x)=x^2
      f4(x) = x^3
      f5(x) = x^4
      g1=f1
      g2=f2-proj(f2,g1)
      g3=f3-proj(f3,g1)-proj(f3,g2)
      g4=f4-proj(f4,g1)-proj(f4,g2)-proj(f4,g3)
      g5=f5-proj(f5,g1)-proj(f5,g2)-proj(f5,g3)-proj(f5,g4)

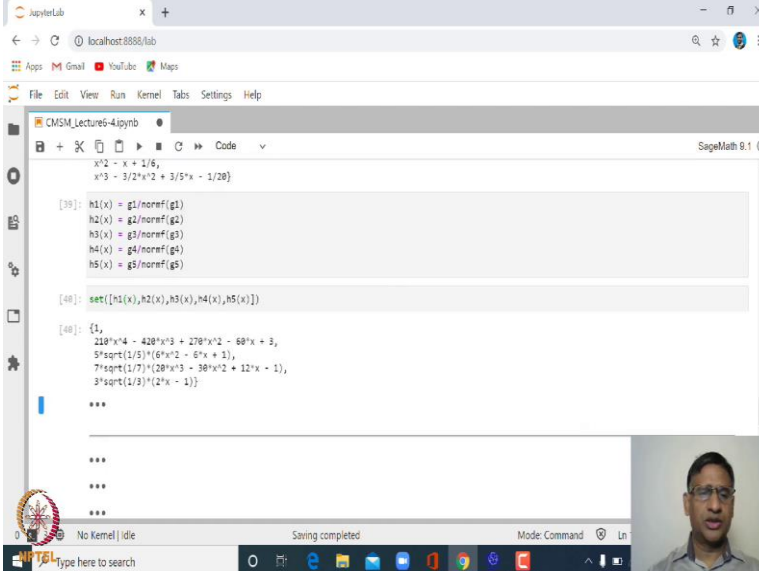
[38]: set([g1(x),g2(x),g3(x),g4(x),g5(x)])

[39]: [1,
      x - 1/2,
      x^4 - 2*x^3 + 9/7*x^2 - 2/7*x + 1/70,
      x^2 - x + 1/6,
      x^3 - 3/2*x^2 + 3/5*x - 1/20]
  
```

And, then let us define f_1, f_2, f_3, f_4, f_5 these are five set of vectors. This forms a basis of $P_4 \mathbb{R}$. And, then again the idea is similar. Define g_1 equal to f_1 , g_2 is equal to f_2 minus orthogonal projection of f_2 onto g_1 , g_3 again in the same way, g_4 and g_5 .

So, in general what is g_k ? g_k is defined as, from f_k you take out the orthogonal projection of f_k onto all the previous orthogonal set of vectors that have been obtained. So, let us execute this and let us also try to plot graph of this function. Let me first show what are these functions which are obtained 1, x minus half, x to power 4 minus x cube and so on. These are the the vectors. One can check that these are orthogonal set of vectors with respect to this inner product.

(Refer Slide Time: 14:32)



```

x^2 - x + 1/6,
x^3 - 3/2*x^2 + 3/5*x - 1/20

[39]: h1(x) = g1/norm(g1)
      h2(x) = g2/norm(g2)
      h3(x) = g3/norm(g3)
      h4(x) = g4/norm(g4)
      h5(x) = g5/norm(g5)

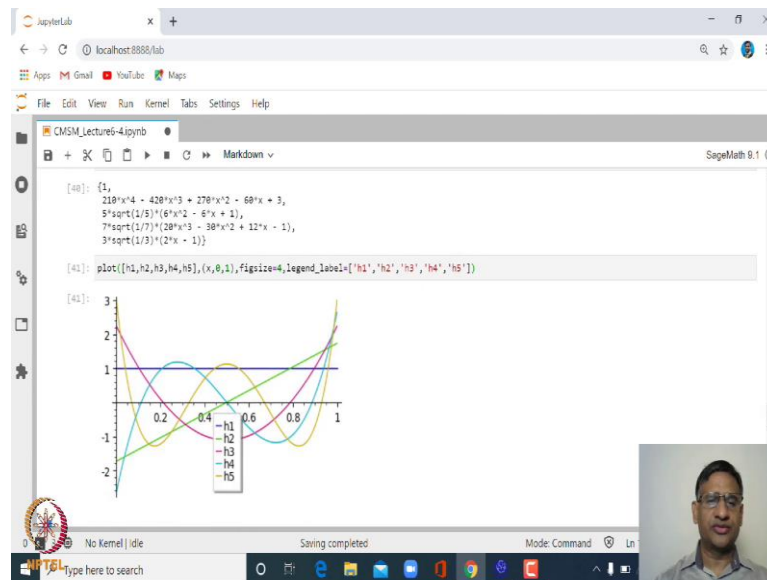
[40]: set([h1(x), h2(x), h3(x), h4(x), h5(x)])

[40]: [1,
      210*x^4 - 420*x^3 + 270*x^2 - 60*x + 3,
      3*sqrt(1/5)*(6*x^2 - 6*x + 1),
      7*sqrt(1/7)*(20*x^3 - 30*x^2 + 12*x - 1),
      3*sqrt(1/3)*(2*x - 1)]

```

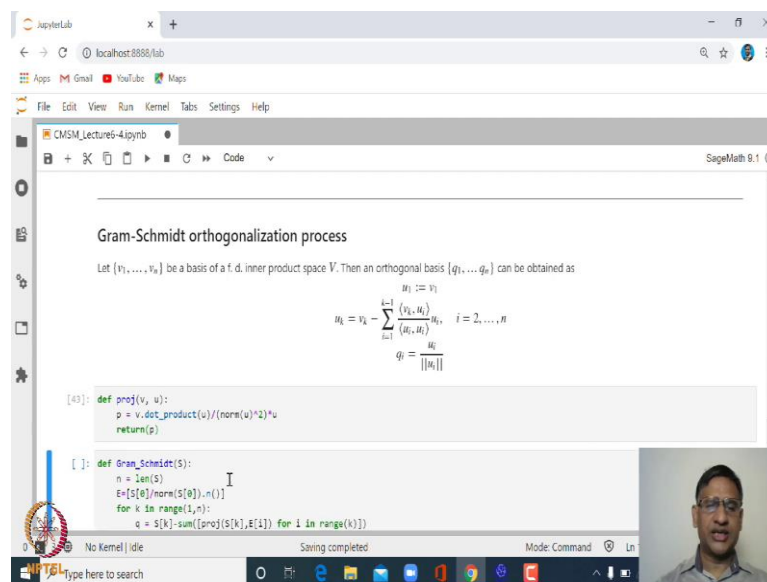
Let us define now unit vectors. So, divide each of this g_1, g_2, g_5 by its norm and then these are the vectors we have got. So, this is an orthonormal set of vectors.

(Refer Slide Time: 14:50)



Let us also plot graph of all these functions. So, is this is straight line is h1 which is a constant function 1; h2 will be some kind of a straight line and h3, h4, h5. This is how these vectors, these orthonormal set of vectors look like.

(Refer Slide Time: 15:17)



Now, you can actually extend this to any finite dimensional inner product space. this process is what is known as Gram-Schmidt process. So, what is this?

It says that, if you start with any set of vectors v_1, v_2, \dots, v_n , a linearly independent set of vectors which forms a basis. In particular take any basis v_1, v_2, \dots, v_n of the finite dimensional

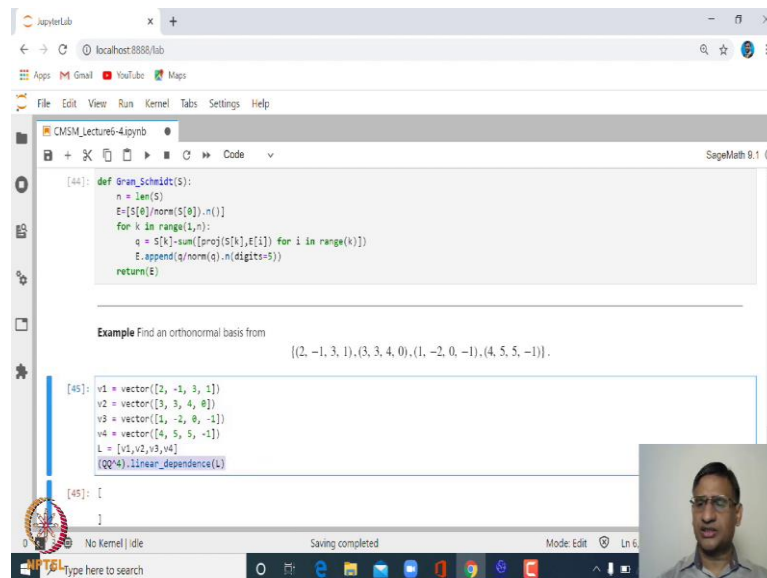
inner product space V and then you can find an orthonormal basis q_1, q_2, \dots, q_n . And, how do we obtain? The process is exactly similar. So, you start with u_1 is equal to v_1 and then define u_k to be v_k minus orthogonal projection of v_k upon u_i i going from 1 to k minus 1.

So, what we are doing? We are throwing out orthogonal projection of v_k on to all the previous orthogonal set of vectors that we have obtained and then defined q_i to be u_i upon norm u_i . This is what is called Gram-Schmidt orthogonalization process. This is quite simple. We already did it for 2 vector, 3 vectors, 4 vectors and 5 vectors.

Let us create an user defined function for this, create a Sage subroutine for this. So, how do we do that? So, let us work with standard inner product in \mathbb{R}^n .

We have the projection of a vector v onto u , which is $v \cdot u$ divided by norm of u square into u . So, that is the orthogonal projection.

(Refer Slide Time: 16:57)



```
[44]: def Gram_Schmidt(S):
      n = len(S)
      E = [S[0]/norm(S[0]).n() ]
      for k in range(1,n):
          q = S[k]-sum([proj(S[k],e[i]) for i in range(k)])
          E.append(q/norm(q).n(digits=5))
      return(E)

Example Find an orthonormal basis from
{(2, -1, 3, 1), (3, 3, 4, 0), (1, -2, 0, -1), (4, 5, 5, -1)}.

[45]: v1 = vector([2, -1, 3, 1])
      v2 = vector([3, 3, 4, 0])
      v3 = vector([1, -2, 0, -1])
      v4 = vector([4, 5, 5, -1])
      L = [v1,v2,v3,v4]
      QQ=4,11near_dependence(L)

[46]:
```

Next let us create Gram-Schmidt orthogonalization process subroutine. What we are doing? We have to pass a set of vectors which are linearly independent. Define n to be

the length of this set, that is number of vectors in S . Then initialize E ; E is going to be the set of orthonormal vectors. So, first let us define unit vector which is the first vector in S , initialize that as first vector in E . And then what we do? Run a loop for k going from at 1 to n .

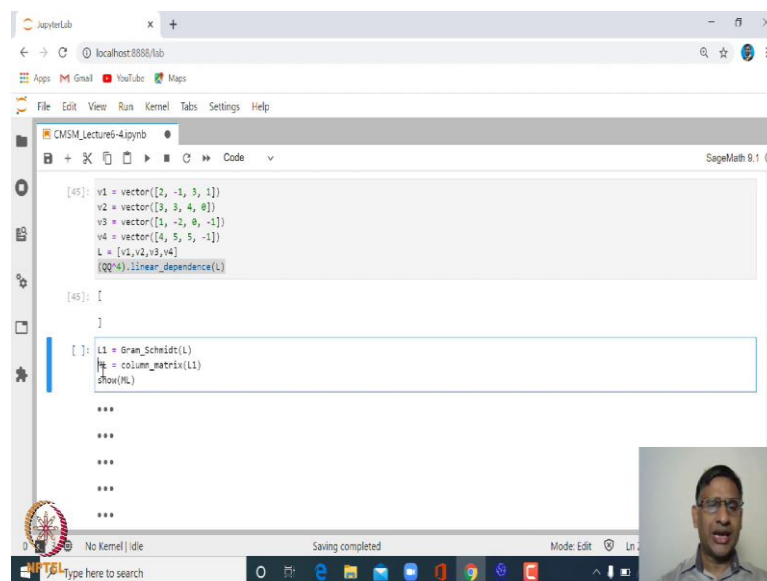
That means, it will start from actually the second vector in S . So, what is it? Define q_2 to be S_k which is k th element in S from that you take out the orthogonal projection of S_k onto E_i , where i going from 1 to range k ; that means, k minus 1. So, that is what is this loop. And, then append inside a the unit vector, that is q upon norm q . Here just I am saying that the display only 5 digits because this is going to convert into float and then return E . So, that is a very simple user defined function for Gram-Schmidt process.

And, of course, if you want to extend this to arbitrary inner product space, you may include the inner product with respect to which this orthogonal projection has to be taken.

So, that is quite easy. Now let us let us look at an example.

So, let us take these four vectors in Q_4 or in R_4 and then let us define these vectors v_1, v_2, v_3, v_4 and L is equal to list of v_1, v_2, v_3, v_4 . We can check that this set of vectors v_1, v_2, v_3, v_4 are linearly independent and hence it forms a basis of Q_4 .

(Refer Slide Time: 19:10)



```

[45]: v1 = vector([2, -1, 3, 1])
      v2 = vector([3, 3, 4, 0])
      v3 = vector([1, -2, 0, -1])
      v4 = vector([4, 5, 5, -1])
      L = [v1, v2, v3, v4]
      (QQ^4).linear_dependence(L)

[45]: [
]

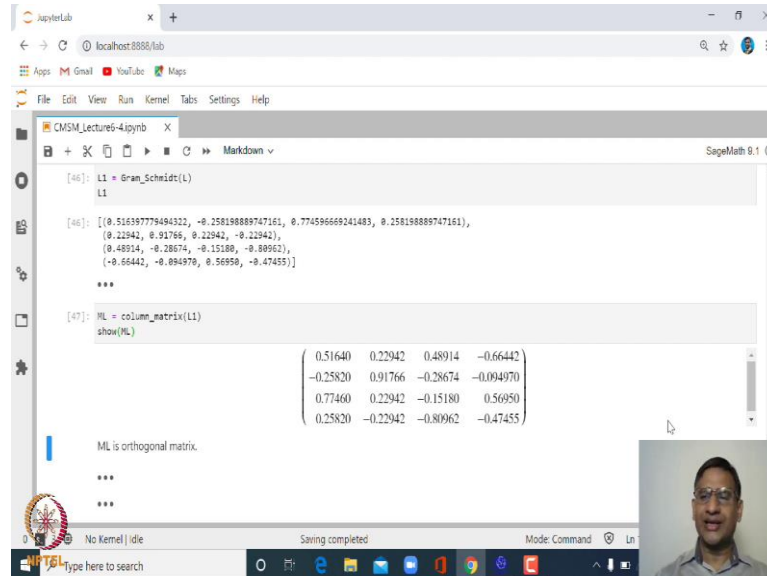
[ ]: L1 = Gram_Schmidt(L)
      M = column_matrix(L1)
      show(M)

***
***
***
***
***

```

Now, let us call this Gram-Schmidt function which we have created and apply this to set of vectors L.

(Refer Slide Time: 19:27)



```
[46]: L1 = Gram_Schmidt(L)
L1
[46]: [(0.516397779494322, -0.258198889747161, 0.774596669241483, 0.258198889747161),
(0.22942, 0.91766, 0.22942, -0.22942),
(0.48914, -0.28674, -0.15188, -0.80962),
(-0.66442, -0.094979, 0.56959, -0.47455)]
***
[47]: ML = column_matrix(L1)
show(ML)
      ( 0.51640  0.22942  0.48914  -0.66442
      -0.25820  0.91766  -0.28674  -0.094970
      0.77460  0.22942  -0.15180  0.56950
      0.25820  -0.22942  -0.80962  -0.47455 )

ML is orthogonal matrix.
***
```

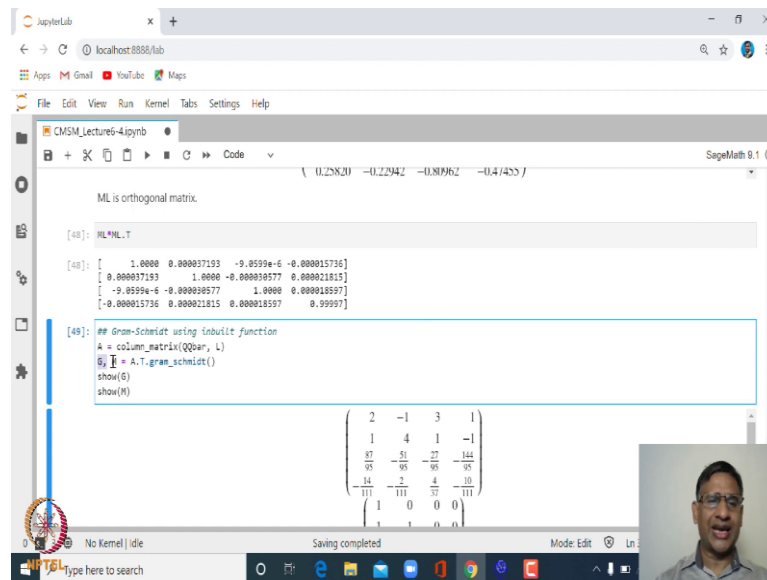
And, what we obtained is again a list of vectors L1. This is this is what we obtained. And, then let us create a matrix out of this. So, this the matrix, the first column is first element in this L1, second column is second element and so on.

And, this matrix, if you look at this matrix, this is actually a matrix whose columns are orthonormal set of vectors. Such a matrix is known as orthogonal matrix.

So, any matrix Q in which all the columns are orthonormal set of vectors, then we say that such a matrix is orthogonal matrix.

And, in case you have orthogonal matrix and if you multiply this matrix with its transpose, you will get identity matrix. So, of course, this should be square matrix here.

(Refer Slide Time: 20:23)



```

[48]: ML*ML.T
[48]: [ 1.0000  0.000037193 -9.8599e-6 -0.000015736]
      [ 0.000037193  1.0000 -0.000018577  0.000021815]
      [-9.8599e-6 -0.000018577  1.0000  0.000018597]
      [-0.000015736  0.000021815  0.000018597  0.99997]

[49]: ## Gram-Schmidt using inbuilt function
A = column_matrix(QQbar, L)
[G, I] = A.T.gram_schmidt()
show(G)
show(M)

```

$$\begin{pmatrix} 2 & -1 & 3 & 1 \\ 1 & 4 & 1 & -1 \\ 87 & -51 & -27 & -144 \\ 93 & 96 & 86 & -95 \\ -14 & -2 & 4 & -10 \\ -111 & -111 & -57 & -111 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

So, let us check that. We can very easily check here this, ML into ML transpose, ML times ML dot T , this should give identity matrix.

Of course, here these things are almost close to 0, so, off diagonal entries of the order 10^{-6} , because here there are lot of round off error.

It has some error, but you can see here, this is diagonal entries 1 1 1 this is also close to 1.

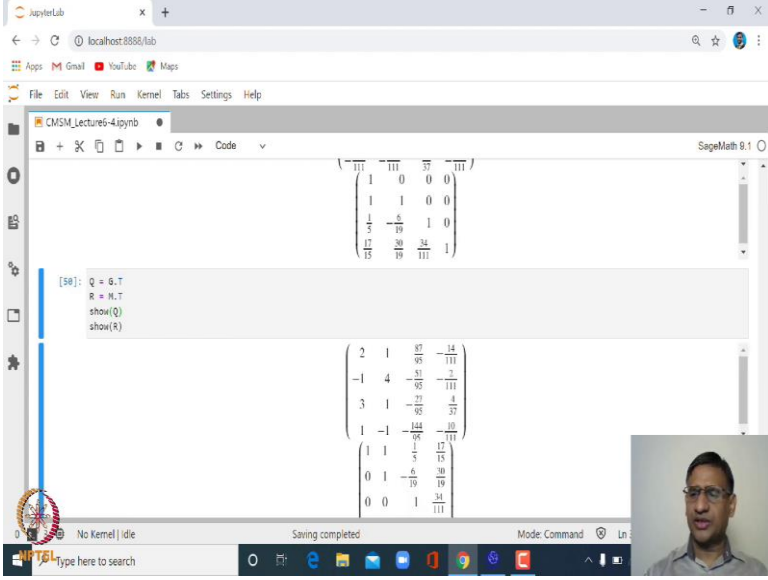
Next let us look at, there is a inbuilt function to find Gram-Schmidt orthogonalization process. Suppose you define a matrix which is a column matrix of a set of linearly independent vectors.

So, we have already defined what is L . L is set of linearly independent vectors v_1, v_2, v_3, v_4 , in this case and define A to be column matrix over $\mathbb{Q}\bar{\mathbb{Q}}$ bar.

So, it is extended rational field you can define over \mathbb{R} and \mathbb{C} . Then on A transpose, let us apply Gram-Schmidt `gram_schmidt` and this gives you actually two matrices.

This gives you two matrices, output is G and M , I have stored this in G and M .

(Refer Slide Time: 21:57)



The screenshot shows a JupyterLab window with a SageMath kernel. The code cell contains the following commands:

```
[58]: Q = G.T
      R = M.T
      show(Q)
      show(R)
```

The output displays two matrices. The first matrix, Q, is a 4x4 matrix:

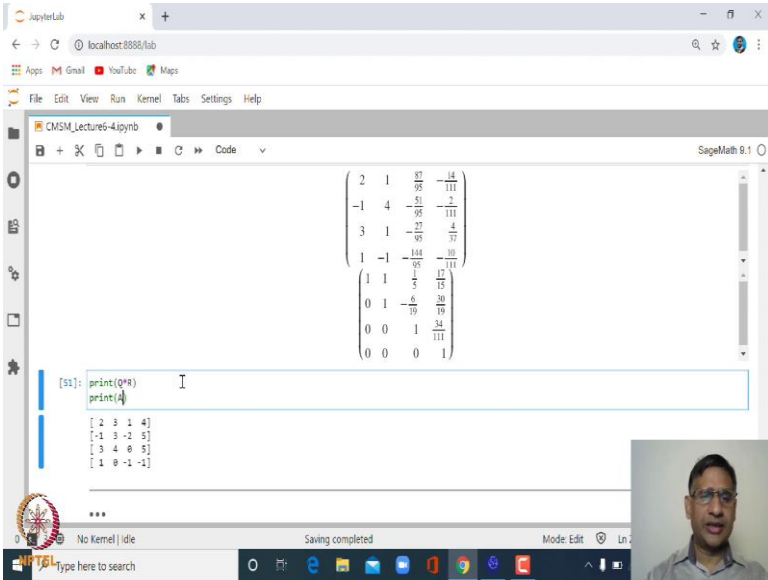
$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ \frac{1}{5} & -\frac{6}{19} & 1 & 0 \\ \frac{17}{15} & \frac{30}{19} & \frac{34}{111} & 1 \end{pmatrix}$$

The second matrix, R, is a 4x4 matrix:

$$R = \begin{pmatrix} 2 & 1 & \frac{87}{95} & -\frac{14}{111} \\ -1 & 4 & -\frac{51}{95} & -\frac{2}{111} \\ 3 & 1 & -\frac{27}{95} & \frac{4}{111} \\ 1 & -1 & -\frac{144}{95} & -\frac{10}{111} \end{pmatrix}$$

So, in this case, this is G and this is M. M is lower triangular matrix. If you look at and if I take the transpose of each one of this. Let us say transpose of this defined in Q and transpose of this defined in R and then this is what we get.

(Refer Slide Time: 22:20)



The screenshot shows the same JupyterLab window. The code cell now contains the following commands:

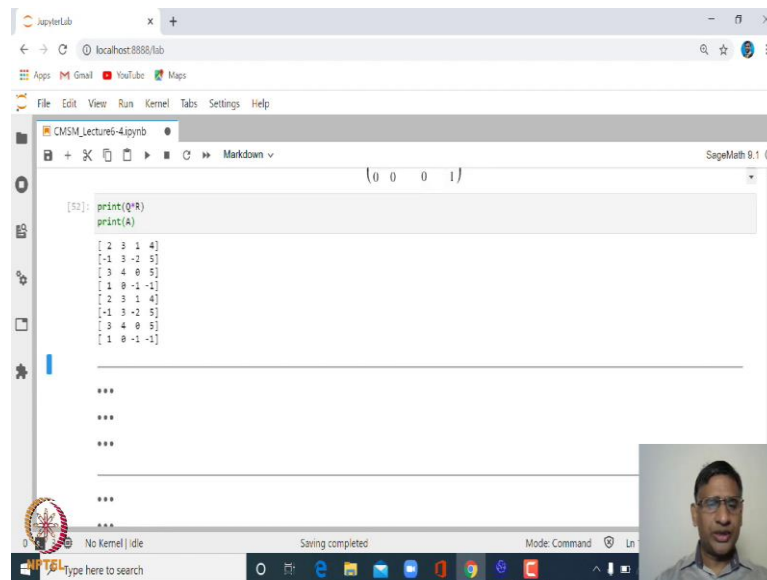
```
[52]: print(Q*R)
      print(A)
```

The output displays the result of the matrix multiplication Q*R, which is a 4x4 matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 3 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \end{bmatrix}$$

And not only that if you try to multiply these two Q and R, multiply Q and R, this is what we got using QR using Gram-Schmidt, and then when you multiply this you will get this matrix A. This is what the matrix A if you if you look. Let me just also print, what is this matrix A. So, print A.

(Refer Slide Time: 22:49)



The screenshot shows a JupyterLab window with a SageMath kernel. The code in the cell is:

```
[5]: print(Q*R)
    print(A)
```

The output displays the matrix A and the product $Q \cdot R$, which is identical to A . The matrix A is:

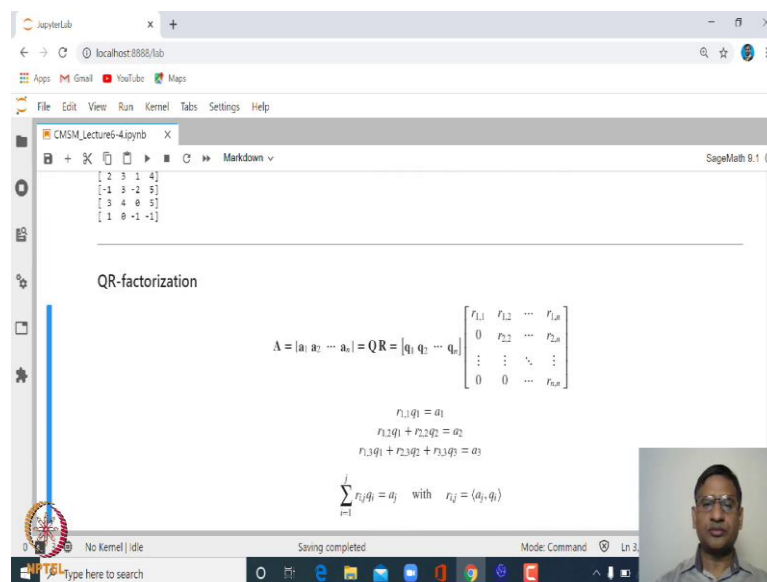
$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \\ 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \end{bmatrix}$$

The product $Q \cdot R$ is also shown as:

$$\begin{pmatrix} 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \\ 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \end{pmatrix}$$

So, that both are the same. So, Q into R is A , this also known as actually QR factorization. Here Q , though it is not an orthogonal matrix, but this is set of orthogonal vectors not orthonormal.

(Refer Slide Time: 23:08)



The screenshot shows a JupyterLab window with a SageMath kernel. The code in the cell is:

```
[5]: print(Q*R)
    print(A)
```

The output displays the matrix A and the product $Q \cdot R$, which is identical to A . The matrix A is:

$$A = \begin{bmatrix} 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \\ 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \end{bmatrix}$$

The product $Q \cdot R$ is also shown as:

$$\begin{pmatrix} 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \\ 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \end{pmatrix}$$

Below the code, the text "QR-factorization" is displayed, followed by the mathematical definition:

$$A = [a_1 \ a_2 \ \dots \ a_n] = QR = [q_1 \ q_2 \ \dots \ q_n] \begin{bmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,n} \\ 0 & r_{2,2} & \dots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_{n,n} \end{bmatrix}$$

The equations for the elements of R are given as:

$$r_{1,1}q_1 = a_1$$

$$r_{1,2}q_1 + r_{2,2}q_2 = a_2$$

$$r_{1,3}q_1 + r_{2,3}q_2 + r_{3,3}q_3 = a_3$$

The general formula for r_{ij} is:

$$\sum_{i=1}^j r_{ij}q_i = a_j \quad \text{with} \quad r_{ij} = \langle a_j, q_i \rangle$$

However, one can define in what is called QR factorization and then QR factorization can be obtained using Gram-Schmidt process. So, let us see what is it? So, it it says that if you have any matrix let us say A , it could be even square matrix any rectangular matrix or even non invertible matrix.

Let us write A as column a_1, a_2, \dots then it says that there exists an orthogonal matrix Q and an upper triangular matrix R such that A can be written as Q into R .

Now, if I say Q is given by column matrix q_1 to q_n and R is given by $r_{11}, r_{12}, \dots, r_{1n}, 0, r_{22}$ and so on. Then you can easily multiply these two matrices Q and R .

And, then equate both sides. What you will get? When you multiply this, by first column, you will get r_{11} times q_1 is equal to a_1 and if q_1 is orthonormal you will get what should be r_{11} .

r_{11} will be nothing, but norm of a_1 upon norm of a_1 actually. Similarly, multiply this by second column, what will get is r_{12} times q_1 plus r_{22} times q_2 is equal to a_2 . Since q_1, q_2 are orthogonal

if you take inner product of both the sides with q_1 you will get value of r_{21} and that will be inner product of a_2 by q_1 . Similarly, if I take inner product of this whole thing by q_2 then this will vanish, we will get r_{22} . So, and you can continue this process.

So, in general what you have is summation r_{ij} times q_i , i going from 1 to j , is nothing, but a_j . That is, j -th column of A is product of this multiplied by j -th column of this R . And, from this you can take inner product of the both the sides with q_i , you will get r_{ij} . So, you can obtain r_{ij} . Once you have obtained this q_1, q_2, \dots, q_n , which we already did in case of Gram-Schmidt process.

This r_{ij} can be obtained using this formula and this is what is known as QR factorization, which is a very useful concept.

(Refer Slide Time: 25:42)

JupyterLab

localhost:8888/lab

Apps Gmail YouTube Maps

File Edit View Run Kernel Tabs Settings Help

CMSM_Lectures6-4.ipynb

SageMath 9.1.1

$$r_{ij}q_j = a_i \quad \text{with} \quad r_{ij} = \langle a_i, q_j \rangle$$

$$A = [a_1 \ a_2 \ \dots \ a_n] = [q_1 \ q_2 \ \dots \ q_n]$$

$$\begin{bmatrix} \langle a_1, q_1 \rangle & \langle a_1, q_2 \rangle & \dots & \langle a_1, q_n \rangle \\ 0 & \langle a_2, q_2 \rangle & \dots & \langle a_2, q_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \langle a_n, q_n \rangle \end{bmatrix}$$

$$r_{1,1}q_1 = a_1$$

$$r_{1,2}q_1 + r_{1,2}q_2 = a_2$$

$$r_{1,3}q_1 + r_{1,2}q_2 + r_{1,3}q_3 = a_3$$

$$r_{ij}q_j = a_i \quad \text{with} \quad r_{ij} = \langle a_i, q_j \rangle$$

No Kernel | idle

Saving completed

Mode: Command | Ln

Let us see what is it. Once you have obtained, this is just a repetition. So, this once you have just obtained A is equal to QR , then r_{ij} is given by this. Let me just get rid of this part, this is just a repetition.

(Refer Slide Time: 26:08)

The screenshot displays a JupyterLab environment with a SageMath notebook open. The notebook's title bar indicates it is a file named 'CMSM_lecture6-4.ipynb'. The interface includes a top navigation bar with 'JupyterLab' and the URL 'localhost:8888/lab', a left sidebar with icons for Files, Edit, View, Run, Kernel, Tabs, Settings, and Help, and a bottom status bar showing 'No Kernel | idle' and 'Solving completed'.

The notebook content shows a SageMath session with the following code and output:

```
[5]: ##Example
v1 = vector([2, -1, 3, 1])
v2 = vector([3, 3, 4, 0])
v3 = vector([1, -2, 0, -1])
v4 = vector([4, 5, 5, -1])
L = [v1,v2,v3,v4]
A = column_matrix(L)
show(A)
```

$$\begin{pmatrix} 2 & 3 & 1 & 4 \\ -1 & 3 & -2 & 5 \\ 3 & 4 & 0 & 5 \\ 1 & 0 & -1 & -1 \end{pmatrix}$$

```
...

[]: ## QR-factorization using the Sage inbuilt function
q1, R1 = A.change_ring(QQbar).QR()

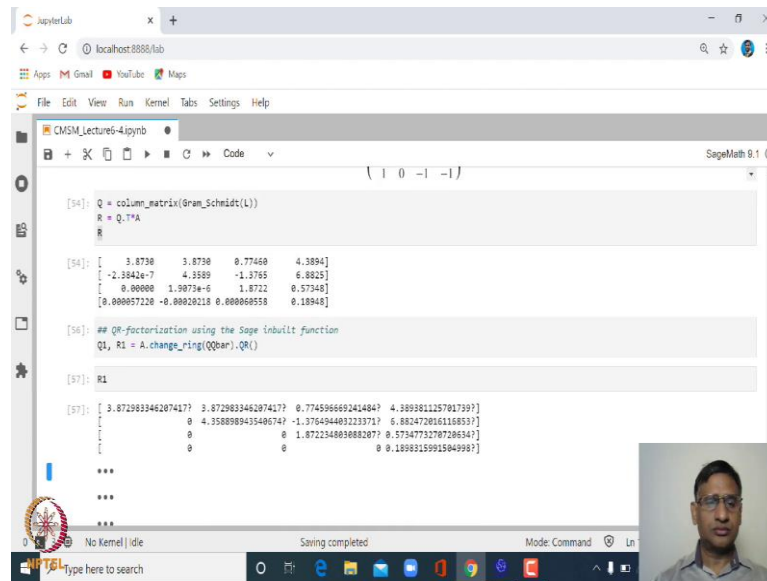
...

```

In the bottom right corner, there is a small video feed showing a man with glasses and a beard, wearing a light blue shirt, looking towards the camera.

Then next let us look at how we can obtain this QR factorization.

(Refer Slide Time: 26:22)



```

[54]: Q = column_matrix(Gram_Schmidt(L))
      R = Q.T*A
      R

[54]: [ 3.8730  3.8730  0.77460  4.3894]
      [-2.3841e-7  4.3589 -1.3765  6.8825]
      [ 0.00000  1.9073e-6  1.8722  0.57348]
      [ 0.000057220 -0.00020218  0.000069558  0.18948]

[56]: ## QR-factorization using the Sage inbuilt function
      Q1, R1 = A.change_ring(QQbar).QR()

[57]: R1

[57]: [ 3.872983346207417?  3.872983346207417?  0.774596669241404?  4.389381125701739? ]
      [ 0  4.358898943540674? -1.37649440323371?  6.882472016116853? ]
      [ 0  0  1.872234803688207?  0.5734773270720634? ]
      [ 0  0  0  0.1899515991504990? ]

***

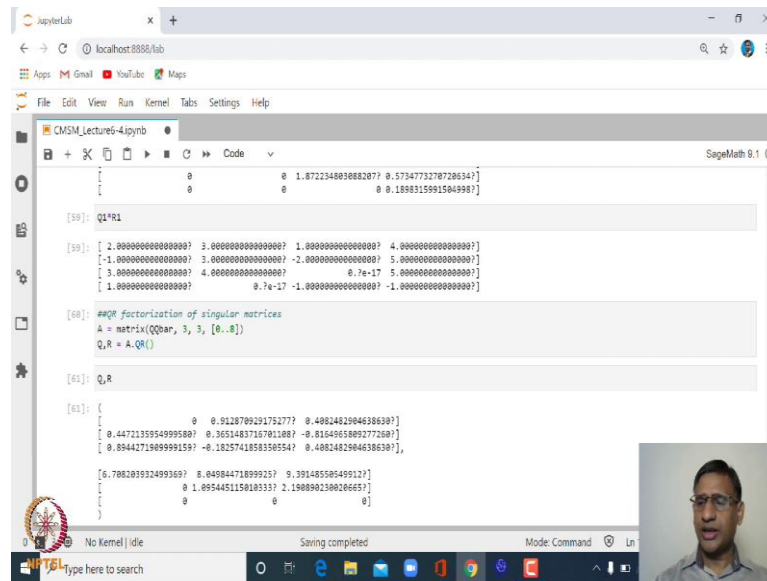
***

***
  
```

This is the again the same set of vectors and A is column matrix. Then we can obtain this Gram Schmidt process.

So, this is the column matrix, the Gram Schmidt orthogonalization process. This is what we did already earlier. And then let us define R is equal to Q transpose A. You see that Q is orthogonal matrix. So, Q into Q transpose will be identity. Therefore, Q transpose is nothing, but inverse of Q. So once we have obtained Q, we can obtain R by taking the Q inverse times the matrix A. You can also obtain this QR factorization using inbuilt function this is known as QR. So, define a matrix A which is over QQ bar and then obtained QR factorization of this and this again we will give you Q and R. So, store this in Q1 and R1 and if you multiply Q1 and R1. Let us see what is R1. This is R1 which is upper triangular matrix and diagonal entries will be non-negative in this case.

(Refer Slide Time: 27:33)



```
[58]: Q1*R1
[59]: [[ 2.000000000000000? 3.000000000000000? 1.000000000000000? 4.000000000000000?]
      [-1.000000000000000? 3.000000000000000? -2.000000000000000? 5.000000000000000?]
      [ 3.000000000000000? 4.000000000000000? 0.7e-17 5.000000000000000?]
      [ 1.000000000000000? 0.7e-17 -1.000000000000000? -1.000000000000000?]]

[60]: ##QR factorization of singular matrices
A = matrix(QQbar, 3, 3, [0..8])
Q,R = A.QR()

[61]: Q,R

[62]: [[ 0.912878929175277? 0.4082482904638630?]
      [ 0.4472135954999580? 0.3651483716701188? -0.8164965809277280?]
      [ 0.8944271969999159? -0.1825741858358054? 0.4082482904638630?]]

[63]: [[ 6.788203932499369? 8.04984471899925? 0.39148558549512?]
      [ 0.189544511501833? 2.198890238020665? 0]
      [ 0 0 0]]
```

And, then let us you can check that Q1 times R1. Let us check Q1 times R1 this would give you the original matrix. Of course, it gives you in decimal because these computations are numerical computations. So for example, question mark says that it is a kind of approximation.

So, you can find QR factorization of any matrix. For example, even you can have a singular matrix. So, here what is it? This is 3 cross 3 matrix starting from 0 to 9. So, 0 1 2 3 to 0 1 2 in first row second row is 2 3 4, third row is 5 6 7. You can find QR factorization of that as well. So, in this case you can see what are Q and R. So, let me print Q and R, in this case this is what you get, right. So, it is not necessary that the matrix A has to be singular, non-singular matrix.

(Refer Slide Time: 28:53)

```

[62]: ##QR factorization of rectangular matrices
A = matrix(CCPair, 3, 4, [0..11])
Q, R = A.QR()
print(Q)
print(R)

0 0.912878923175277? 0.4082482904638638?
0.447211595499588? 0.365148371670188? -0.8164965809277268?
0.8944271909999159? -0.182574185835054? 0.4082482904638638?
0.944271909999915? 10.28591269469994? 11.6275534819981? 12.96519426949878?
0 1.09544511501833? 2.190890238020665? 3.286335345038997?
0 0 0
***
***
***

```

Now, you can also have a matrix which is rectangular matrix. This is I should write rectangular, you can have rectangular matrix, need not be square matrix.

So, if I have rectangular matrix, let us say this is 3 cross 4 matrix. Again you can find its QR factorization using inbuilt function A dot QR.

(Refer Slide Time: 29:33)

```

[63]: A.QR??

Source:
def QR(self, full=True):
    """
    Returns a factorization of ``self`` as a unitary matrix
    and an upper-triangular matrix.

    INPUT:

    - ``full`` - default: ``True`` - if ``True`` then the
      returned matrices have dimensions as described below.
      If ``False`` the ``R`` matrix has no zero rows and the
      columns of ``Q`` are a basis for the column space of
      ``self``.

    OUTPUT:

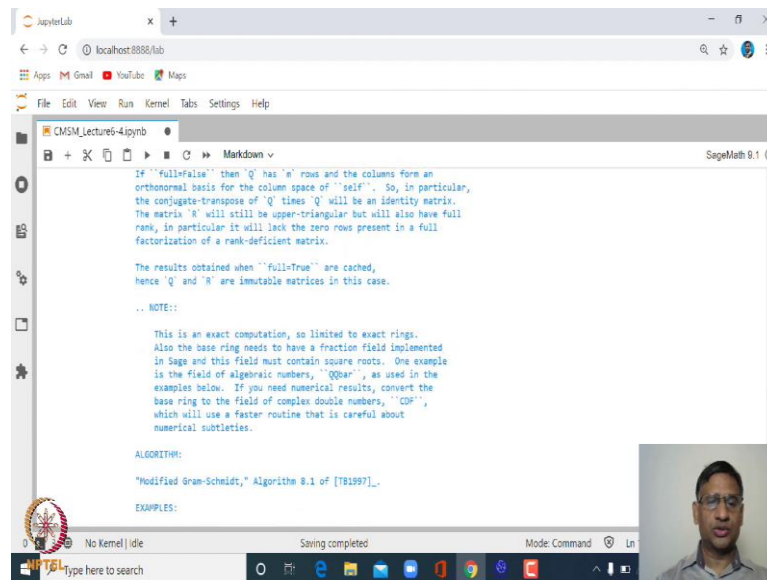
    If ``self`` is an ``m``times ``n`` matrix and ``full=True`` then this
    method returns a pair of matrices: ``Q`` is an ``m``times ``m`` unitary
    matrix (meaning its inverse is its conjugate-transpose) and ``R``

```

Next you can take help on QR factorization. So, if you have a matrix A and then say A dot QR double question mark or single question mark, you will see a help document.

And, that help document you can go through along with the examples and then see how you can make use of various options. This is the QR factorizations source code.

(Refer Slide Time: 30:02)

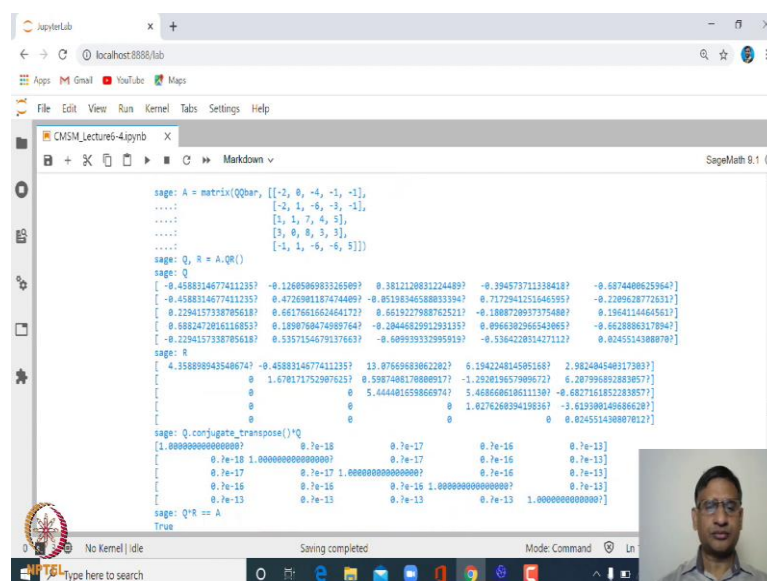


The screenshot shows a JupyterLab window with a SageMath 9.1 notebook. The notebook content includes:

- Text explaining that if `full=False`, then `Q` has `m` rows and the columns form an orthonormal basis for the column space of `self`. So, in particular, the conjugate-transpose of `Q` times `Q` will be an identity matrix. The matrix `R` will still be upper-triangular but will also have full rank, in particular it will lack the zero rows present in a full factorization of a rank-deficient matrix.
- Text stating that results obtained when `full=True` are cached, hence `Q` and `R` are immutable matrices in this case.
- A **NOTE:** section stating: "This is an exact computation, so limited to exact rings. Also the base ring needs to have a fraction field implemented in Sage and this field must contain square roots. One example is the field of algebraic numbers, `QQbar`, as used in the examples below. If you need numerical results, convert the base ring to the field of complex double numbers, `CCDF`, which will use a faster routine that is careful about numerical subtleties."
- An **ALGORITHM:** section stating: "Modified Gram-Schmidt," Algorithm 8.1 of [TB1997].
- An **EXAMPLES:** section.

The interface also shows a file explorer on the left, a terminal at the bottom, and a video feed of the presenter in the bottom right corner.

(Refer Slide Time: 30:06)



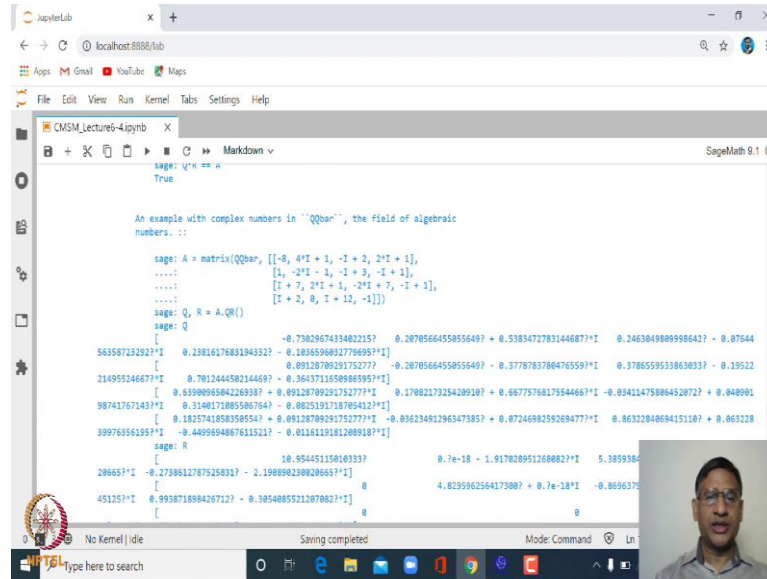
The screenshot shows the same JupyterLab window with the SageMath notebook displaying a 5x5 matrix example:

```
sage: A = matrix(QQbar, [[-2, 0, -4, -1, -1],
.....:                [-2, 1, -6, -5, -1],
.....:                [1, 1, 7, 4, 5],
.....:                [3, 0, 0, 3, 3],
.....:                [-1, 1, -6, -5, 5]])
sage: Q, R = A.QR()
sage: Q
[[-0.4588314677411235? -0.12469896983316509? 0.381212883124489? -0.39457371138418? -0.6874408625964?
[-0.4588314677411235? 0.4726981187474409? -0.8519834558803394? 0.717294125164655? -0.2289628772631?
[0.2294157338785618? 0.6617661662464172? 0.6619227988762521? -0.1888720937375488? 0.1964114464561?
[0.6882472816116853? 0.1890768474089764? -0.2844681295129313? 0.896382965424065? -0.6628886317894?
[-0.2294157338785618? 0.5357154679137663? -0.609939323995919? -0.536422851427112? 0.8245514388878?]]
sage: R
[[ 4.358898943540674? -0.4588314677411235? 13.07669693862282? 6.194224814585168? 2.982484548917383?
[ 0 1.678171752907625? 0.5987408178888917? -1.252819657989672? 6.287996812883857?
[ 0 5.444481659866974? 5.468668618611138? -0.6827161852283857?
[ 0 0 1.02762839419836? -3.619380149886628?
[ 0 0 0 0.8245514388878?]]
sage: Q.conjugate_transpose()*Q
[[1.000000000000000? 0.7e-18 0.7e-17 0.7e-16 0.7e-13]
[0.7e-18 1.000000000000000? 0.7e-17 0.7e-16 0.7e-13]
[0.7e-17 0.7e-17 1.000000000000000? 0.7e-16 0.7e-13]
[0.7e-16 0.7e-16 0.7e-16 1.000000000000000? 0.7e-13]
[0.7e-13 0.7e-13 0.7e-13 0.7e-13 1.000000000000000?]]
sage: Q*R == A
True
```

The interface also shows a file explorer on the left, a terminal at the bottom, and a video feed of the presenter in the bottom right corner.

So, it also gives you the source code and here there are several examples. You can see here this is bigger example. It is actually 5 cross 5 matrix.

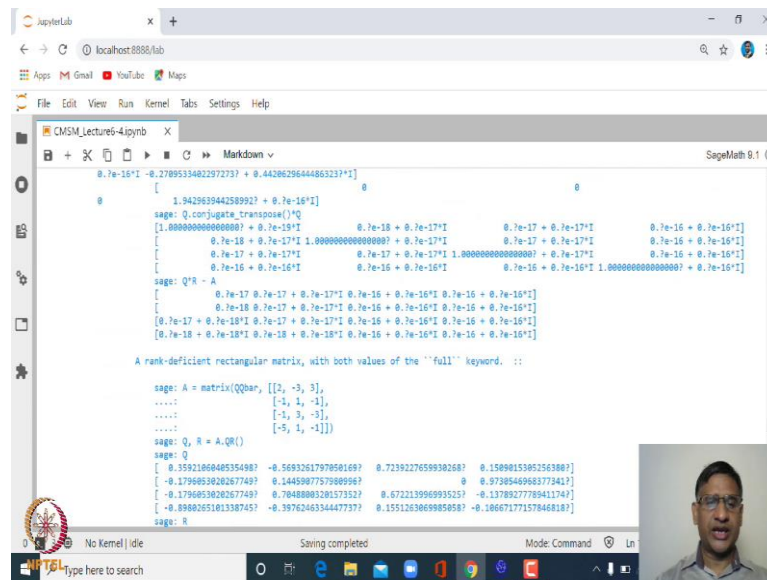
(Refer Slide Time: 30:15)



```
sage: Q, R = A.QR()
sage: Q
[
  [-0.7382967433462215? 0.2878566455855649? + 0.5383472783146687*I 0.246384989898642? - 0.07644
  56358723292*I
  [
    0.228161768313432? - 0.1836594882779695*I
    0.09128789529175277? - 0.2878566455855649? - 0.3778783780476559*I 0.378655953386383? - 0.19522
    21495524667*I
    0.781244458214469? - 0.3643711658986595*I
    0.639898958422693? + 0.8912878929175277*I 0.1788217325428918? + 0.6677576817554466*I -0.834147588645287? + 0.848901
    98741767143*I
    0.3148171885586764? - 0.0825191718705412*I
    0.1825741858358554? + 0.8512878929175277*I -0.83623491296347385? + 0.872468825269477*I 0.8632284869415118? + 0.865328
    39976356195*I
    -0.449594867611521? - 0.8116119181268918*I
  ]
  sage: R
  [
    10.85445115610333? 0.7e-18 - 1.91782895126882*I 5.3859384
    28665*I -0.273861278752583? - 1.198898238828665*I
    0 4.829596256417308? + 0.7e-18*I -0.8696376
    45125*I 0.953871898426712? - 0.3054885521287862*I
    0 0
  ]
```

And, it can also find QR factorization of complex matrices. So, that is what it says and so on.

(Refer Slide Time: 30:22)



```
sage: Q, R = A.QR()
sage: Q
[
  1.942963944258992? + 0.7e-16*I
  [
    1.000000000000000? + 0.7e-16*I
    0.7e-18 + 0.7e-17*I 1.000000000000000? + 0.7e-17*I 0.7e-17 + 0.7e-17*I 0.7e-16 + 0.7e-16*I
    0.7e-17 + 0.7e-17*I 0.7e-17 + 0.7e-17*I 1.000000000000000? + 0.7e-17*I 0.7e-16 + 0.7e-16*I
    0.7e-16 + 0.7e-16*I 0.7e-16 + 0.7e-16*I 0.7e-16 + 0.7e-16*I 1.000000000000000? + 0.7e-16*I
  ]
  sage: R - A
  [
    0.7e-17 0.7e-17 + 0.7e-17*I 0.7e-16 + 0.7e-16*I 0.7e-16 + 0.7e-16*I
    0.7e-18 0.7e-17 + 0.7e-17*I 0.7e-16 + 0.7e-16*I 0.7e-16 + 0.7e-16*I
    [0.7e-17 + 0.7e-18*I 0.7e-17 + 0.7e-17*I 0.7e-16 + 0.7e-16*I 0.7e-16 + 0.7e-16*I]
    [0.7e-18 + 0.7e-18*I 0.7e-18 + 0.7e-18*I 0.7e-16 + 0.7e-16*I 0.7e-16 + 0.7e-16*I]
  ]
  A rank-deficient rectangular matrix, with both values of the "full" keyword.
  sage: A = matrix(QQbar, [[2, -3, 3],
  ....: [-1, 1, -1],
  ....: [-1, 3, -3],
  ....: [-5, 1, -1]])
  sage: Q, R = A.QR()
  sage: Q
  [
    0.359218684853488? -0.5689361797858169? 0.7239227658938268? 0.1589815385256388?
    [-0.1796853826267749? 0.1464987757988996? 0 0.9738546968377341?
    [-0.1796853826267749? 0.784888828121732? 0.672213966993525? -0.1378827778941174?
    [-0.8986265181338745? -0.3976246354447737? 0.1551263869858585? -0.18667177357846818?
  ]
  sage: R
```

There are different examples.

(Refer Slide Time: 30:24)

[illegible]

One can one can go through this list of examples.

(Refer Slide Time: 30:27)

The screenshot shows a JupyterLab window with a SageMath session. The terminal output is as follows:

```

(5, 0)
sage: Q
[1 0 0]
[0 1 0]
[0 0 1]

TESTS:

Inexact rings are caught and "CDF" suggested. ::

sage: A = matrix(RealField(100), 2, 2, range(4))
sage: A.QR()
Traceback (most recent call last):
...
NotImplementedError: QR decomposition is implemented over exact rings, try CDF for numerical results, not Real Field with 100 b
its of precision

Without a fraction field, we cannot hope to run the algorithm. ::

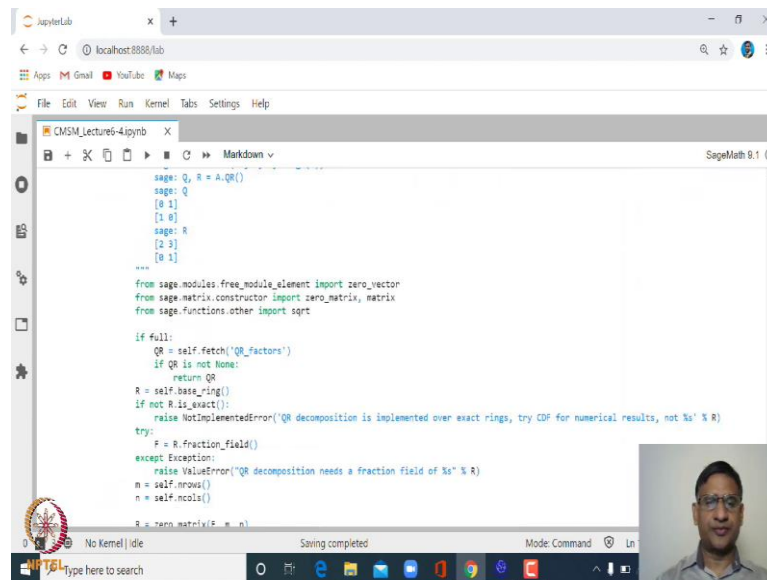
sage: A = matrix(Integers(5), 2, 2, range(4))
sage: A.QR()
Traceback (most recent call last):
...
ValueError: QR decomposition needs a fraction field of Ring of integers modulo 5

The biggest obstacle is making unit vectors, thus requiring square

```

The interface also shows a top bar with 'JupyterLab' and 'localhost:8888/lab', a left sidebar with icons for files, home, and search, and a bottom status bar showing 'No Kernel | idle' and 'Saving completed'.

(Refer Slide Time: 30:28)



The screenshot shows a JupyterLab window with a file explorer on the left displaying 'CMSM_Lecture6-4.ipynb'. The main area contains SageMath code for QR decomposition. The code defines a class with methods to compute QR factors. It includes imports for SageMath modules and functions. The code defines a class with methods to compute QR factors. It includes imports for SageMath modules and functions. The code defines a class with methods to compute QR factors. It includes imports for SageMath modules and functions.

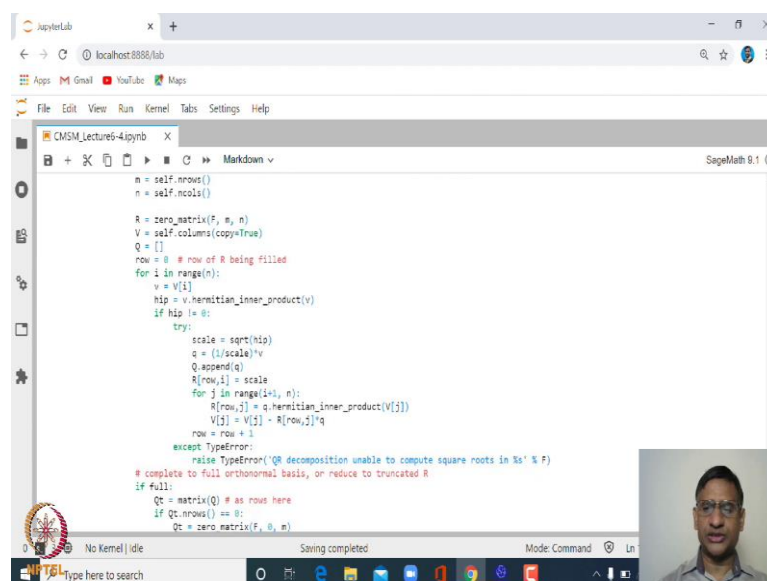
```
sage: Q, R = A.QR()
sage: Q
[0 1]
[1 0]
sage: R
[2 3]
[0 1]

...

from sage.modules.free_module_element import zero_vector
from sage.matrix.constructor import zero_matrix, matrix
from sage.functions.other import sqrt

if full:
    QR = self.fetch('QR_factors')
    if QR is not None:
        return QR
    R = self.base_ring()
    if not R.is_exact():
        raise NotImplementedError('QR decomposition is implemented over exact rings, try CDF for numerical results, not %s' % R)
    try:
        F = R.fraction_field()
    except Exception:
        raise ValueError('QR decomposition needs a fraction field of %s' % R)
    m = self.nrows()
    n = self.ncols()
    R = zero_matrix(F, m, n)
```

(Refer Slide Time: 30:29)

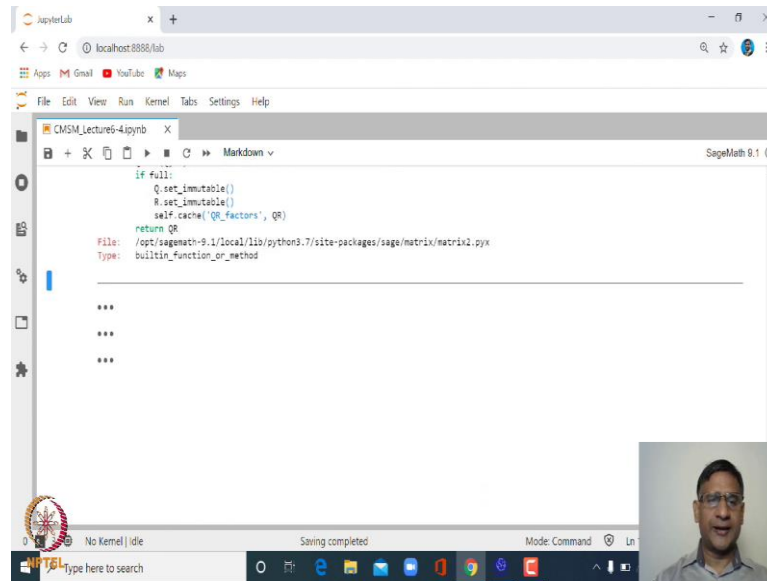


The screenshot shows a JupyterLab window with a file explorer on the left displaying 'CMSM_Lecture6-4.ipynb'. The main area contains SageMath code for QR decomposition. The code defines a class with methods to compute QR factors. It includes imports for SageMath modules and functions. The code defines a class with methods to compute QR factors. It includes imports for SageMath modules and functions. The code defines a class with methods to compute QR factors. It includes imports for SageMath modules and functions.

```
m = self.nrows()
n = self.ncols()

R = zero_matrix(F, m, n)
V = self.columns(copy=True)
Q = []
row = 0 # row of R being filled
for i in range(n):
    v = V[i]
    hip = v.hermitian_inner_product(v)
    if hip != 0:
        try:
            scale = sqrt(hip)
            q = (1/scale)*v
            Q.append(q)
            R[row,i] = scale
            for j in range(i+1, n):
                R[row,j] = q.hermitian_inner_product(V[j])
                V[j] = V[j] - R[row,j]*q
            row = row + 1
        except TypeError:
            raise TypeError('QR decomposition unable to compute square roots in %s' % F)
# complete to full orthonormal basis, or reduce to truncated R
if full:
    Qt = matrix(Q) # as rows here
    if Qt.nrows() == 0:
        Qt = zero_matrix(F, 0, m)
```

(Refer Slide Time: 30:29)



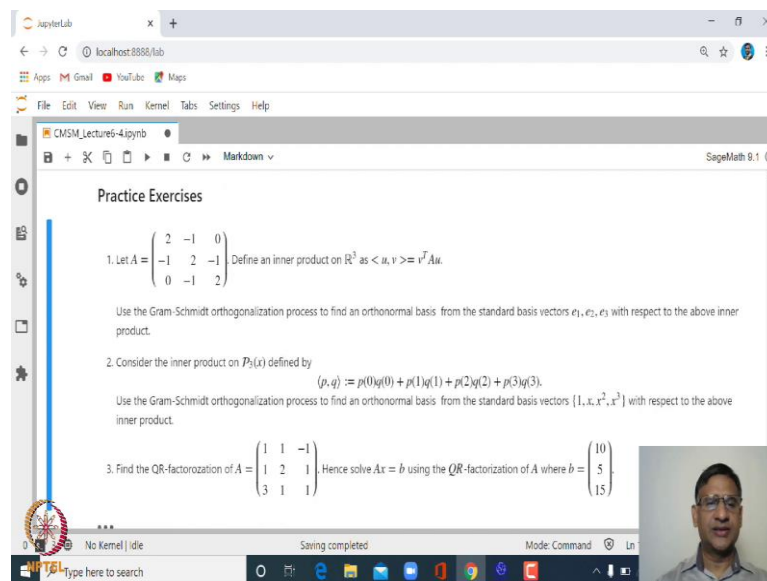
```
if full:
    Q.set_immutable()
    R.set_immutable()
    self.cache('QR_factors', QR)
    return QR
File: /opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/matrix/matrix2.pyx
Type: builtin_function_or_method

...

...

...
```

(Refer Slide Time: 30:32)



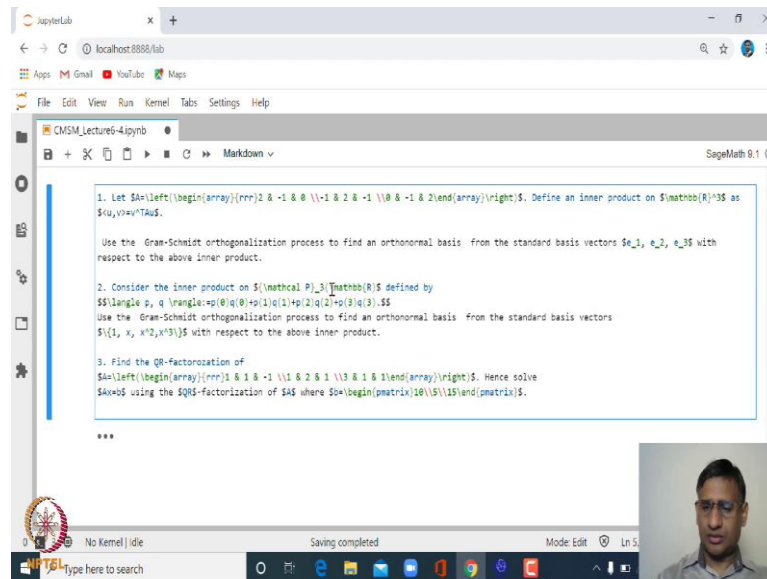
Practice Exercises

- Let $A = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$. Define an inner product on \mathbb{R}^3 as $\langle u, v \rangle = v^T A u$.
Use the Gram-Schmidt orthogonalization process to find an orthonormal basis from the standard basis vectors e_1, e_2, e_3 with respect to the above inner product.
- Consider the inner product on $\mathcal{P}_3(x)$ defined by
$$\langle p, q \rangle := p(0)q(0) + p(1)q(1) + p(2)q(2) + p(3)q(3).$$
Use the Gram-Schmidt orthogonalization process to find an orthonormal basis from the standard basis vectors $\{1, x, x^2, x^3\}$ with respect to the above inner product.
- Find the QR-factorization of $A = \begin{pmatrix} 1 & 1 & -1 \\ 1 & 2 & 1 \\ 3 & 1 & 1 \end{pmatrix}$. Hence solve $Ax = b$ using the QR-factorization of A where $b = \begin{pmatrix} 10 \\ 5 \\ 15 \end{pmatrix}$.

At the end let me leave you with few exercises. These are quite simple exercises.

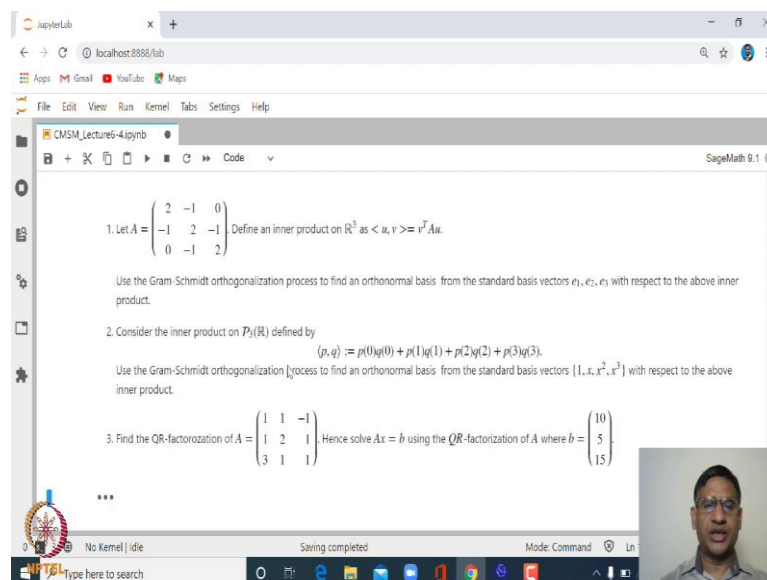
So, first take a matrix A. Define inner product with respect to this matrix and then use Gram-Schmidt orthogonalization process to find an orthonormal basis with respect to a standard basis e_1, e_2, e_3 .

(Refer Slide Time: 30:57)



Similarly, define an inner product on \mathbb{P}_3 , this I should write $\mathbb{P}_3 \times \mathbb{P}_3$. Instead of \mathbb{R}^3 let me write here. $\mathbb{P}_3 \times \mathbb{R}$ and with respect to standard basis 1 to x^3 . Again find an orthonormal set of basis and then find QR factorization of this and hence solve this Ax equal to b .

(Refer Slide Time: 31:06)



See once you have QR factorization of a matrix then this system Ax equal to b can be written as QR times x is equal to b . Now, Q is invertible. So, if you multiply both sides by Q inverse,

then we will get $Rx = Q^T b$. Then what you have Rx and R is upper triangular matrix. So, actually to solve this system of linear equations,

it has boiled down to solving upper triangular set of linear system which is quite easy to solve using back substitution.

Of course, this may be more expensive than a Gaussian elimination method. However, once you know the QR factorization of a this coefficient matrix, then it becomes easier to solve.

So, these are set of exercises. We will look at some more concepts in next lecture.