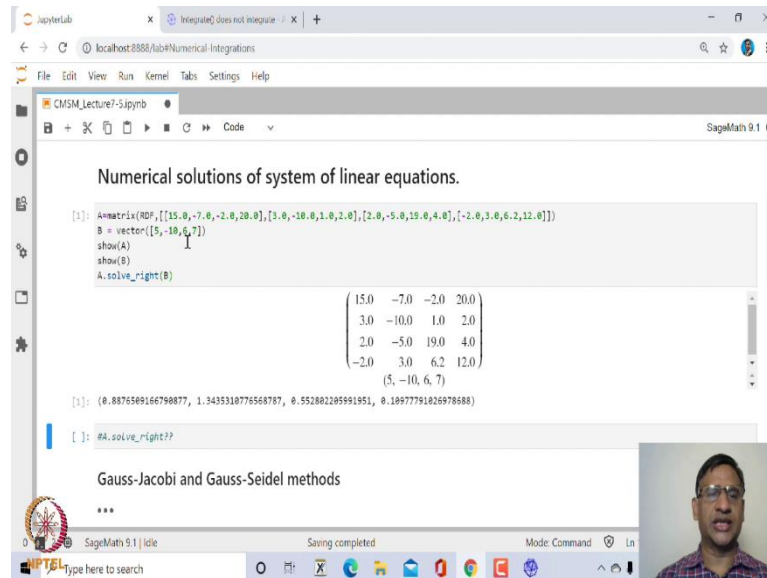


**Computational Mathematics with SageMath**  
**Prof. Ajit Kumar**  
**Department of Mathematics**  
**Institute of Chemical Technology, Mumbai**

**Lecture - 46**  
**Numerical Solutions of System of Linear Equations in SageMath**

(Refer Slide Time: 00:15)



```
NUMERICAL SOLUTIONS OF SYSTEM OF LINEAR EQUATIONS.

[1]: A=matrix(RDF,[[15.0,-7.0,-2.0,20.0],[3.0,-10.0,1.0,2.0],[2.0,-5.0,19.0,4.0],[-2.0,3.0,6.2,12.0]])
B=vector([5,-10,6,7])
show(A)
show(B)
A.solve_right(B)

      ( 15.0  -7.0  -2.0  20.0 )
      (  3.0 -10.0   1.0   2.0 )
      (  2.0  -5.0  19.0   4.0 )
      (-2.0   3.0   6.2  12.0 )
      (  5, -10, 6, 7 )

[1]: (0.8876569166750877, 1.3435310776568787, 0.552802205991951, 0.18977791026978688)

[ ]: #A.solve_right??

Gauss-Jacobi and Gauss-Seidel methods
***
```

Welcome to the 46th lecture on Computational Mathematics with SageMath. In this lecture, we will look at Numerical solutions of system of linear equations. So, we have already seen how to solve a system of linear equations in SageMath. So, for example, if you, if you have a system of linear equations  $Ax$  equal to  $B$ , we can solve this using `A dot solve underscore right`.

Here  $A$  is square system, right, or we can use solve function, in that case, we need to create each equation, and we, you will solve, and in the square bracket, we will write list of equations, and solve with respect to the given variables, right? Of course, you can look at help on solve `A dot solve underscore right` to find out how this program solve underscore right is written. So, in case you have a large system, then solving this system using `A dot solve right` may be slightly difficult. Similarly, solving manually using RREF may also be quite time consuming. So, in that situation, one can use some numerical methods to, to solve system of linear equations iteratively.

(Refer Slide Time: 01:53)

**Gauss-Jacobi and Gauss-Seidel methods**

We write the system  $AX = B$  as

$$X_{k+1} = HX_k + C$$

Here  $H$  is called an iteration matrix.

Let  $A = L + D + U$  where  $D$  is the diagonal part,  $L$  and  $U$  are strictly lower and upper triangular parts of  $A$ .

Then  $AX = B$  reduces to  $(L + D + U)X = B$  which can be written as  $DX = -(L + U)X + B$ .

If  $D$  is invertible then

$$X = -D^{-1}(L + U)X + D^{-1}B$$

Hence the iterative sequence is given by

$$X^{(k+1)} = -D^{-1}(L + U)X^{(k)} + D^{-1}B$$

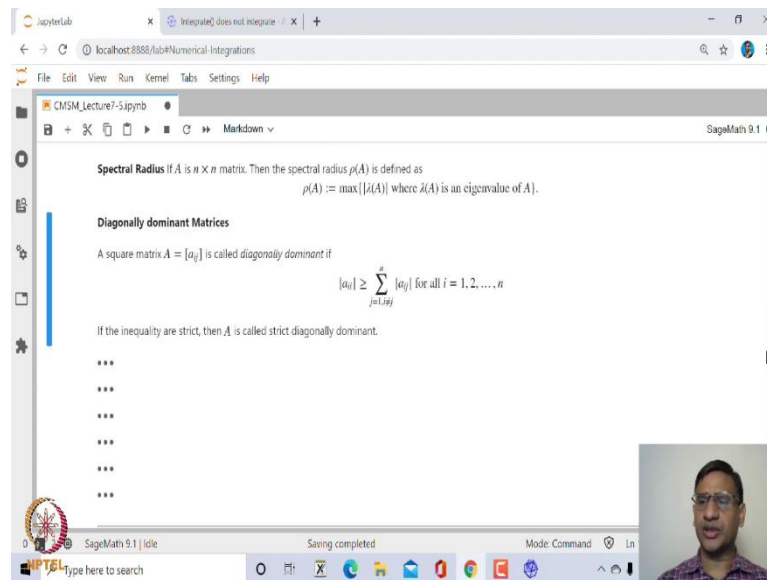
Thus the iteration matrix  $H = -D^{-1}(L + U)$  and  $C = D^{-1}B$ .

The method is convergent iff spectral radius of  $H$  is strictly less than 1.

And 2 important methods we will discuss; one is Gauss Jacobi, and Gauss Seidel method. So, let us see how does it work? So, suppose we have the system  $AX$  equal to  $B$ , then we write this system  $AX$  equal to  $B$  as an iteration scheme  $X_{k+1}$  is equal to  $H$  into  $X_k$  plus  $C$ , where  $H$  is called iteration matrix, and  $C$  is another matrix.

So, how do we write this  $H$  and  $C$ ? So, what we do? We, we split  $A$  as  $L$  plus  $D$  plus  $U$ , where  $D$  is diagonal part of  $A$ ,  $L$  and  $U$  are strictly lower and upper triangular parts of  $A$ , right? And then what do you do? You, you write  $AX$  equal to  $B$  as  $L$  plus  $D$  plus  $U$   $X$  is equal to  $B$ , and then you can separate  $DX$  from the left-hand side. So,  $DX$  will become minus  $L$  plus  $U$ , the whole thing times  $X$ , plus  $B$ . And in case  $D$  happens to be invertible matrix, that is, all diagonal entries of  $D$  are non zero, then you can write  $X$  as minus  $D$  inverse  $L$  plus  $U$  into  $X$  plus  $D$  inverse  $B$ . So, this minus  $D$  inverse  $L$  plus  $U$  is what is defined as  $H$ , the iteration matrix, and  $C$  will be  $D$  inverse  $B$ . This particular scheme is known as Gauss Jacobi iteration scheme, and in this case, one can show that this iteration scheme is convergent, no matter what is the starting guess value, starting guess solution  $X_0$ , if and only if the spectral radius of  $H$  is strictly less than 1. Now, what is spectral radius?

(Refer Slide Time: 03:34)



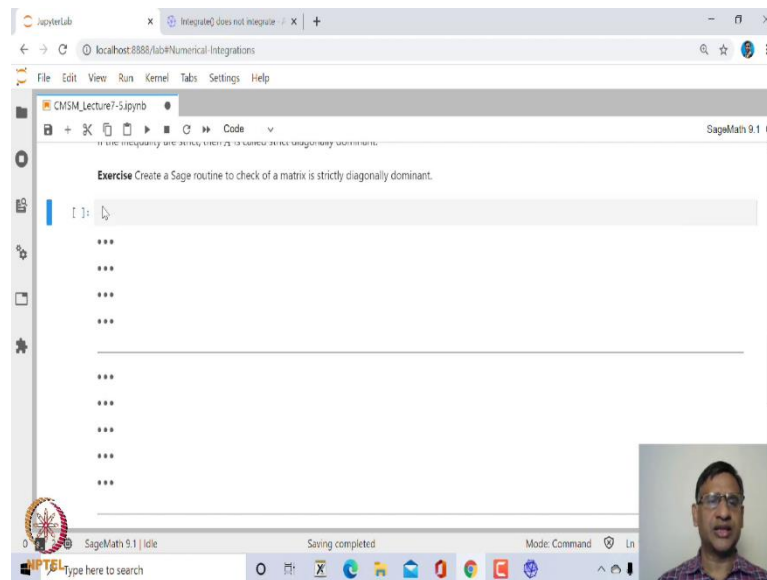
Spectral radius of a matrix is nothing but the maximum of the modulus of all the eigenvalues of the given matrix. So, one can, one can find spectral radius, this is quite because, we are taking modulus because it could also be negative and it, eigenvalue could be negative, it could be also complex eigenvalues. So, we take modulus of this, and since  $A$  is  $n$  cross  $n$ , it will have at most  $n$  real eigenvalues, it will have at most  $n$  eigenvalues, and then we take the modulus of all of these and then take the maximum.

Now, another condition on the convergence, the necessary condition on convergence is that, in case this coefficient matrix  $A$  is diagonally dominant, diagonally dominant, then this, this matrix  $D$ , the diagonal will, entries, all will be non zero, and hence you will be able to invert this  $D$ , you can find the inverse of  $D$ .

And what is diagonally dominant matrix? In case the modulus of the, the diagonal entries, let us say  $\text{mod } A[i,i]$ , is bigger than equal to summation of the modulus of the remaining entries in each row, in  $i$ th row in this case, then we say that  $A$  is diagonally dominant, right? And in case this inequality is strict, we say that  $A$  is strictly diagonally dominant.

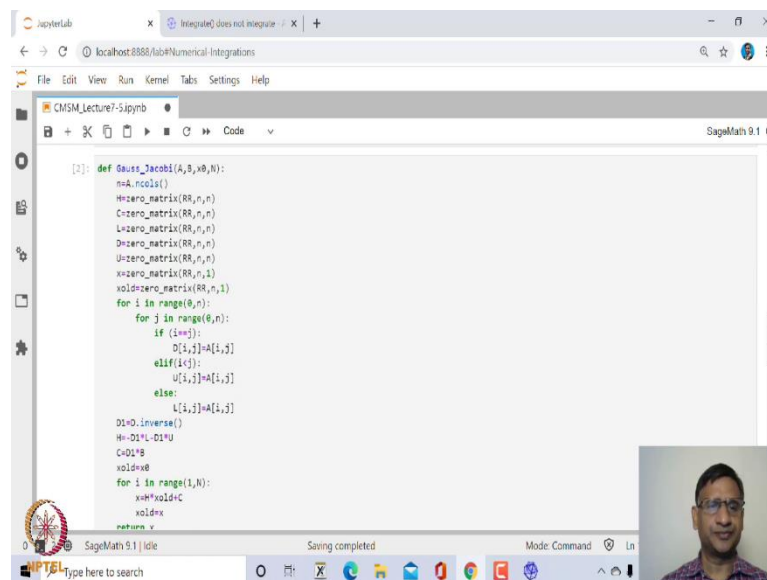
So, in case  $A$  is strictly diagonally dominant matrix, one can show that this iteration scheme will always converge no matter what is the starting guess value. And of course, this is not a sufficient condition; the sufficient condition is the spectral radius should be strictly less than 1, right?

(Refer Slide Time: 05:29)



So, I, I leave this as an exercise to, for you to write a subroutine, Sage subroutine to check, if a given matrix is strictly diagonally dominant, this would be quite easy to, to create.

(Refer Slide Time: 05:44)



Now, let us write a program in order to, to, to create this iteration scheme, Gauss Jacobi iteration scheme. So, how do we do that? I am just giving a name Gauss underscore Jacobi, you give input A, the matrix A, the right-hand side matrix B, and the X 0 is the initial guess, and then see how many iterates you want to, to, to give.

So, this is a small code which separates capital D, capital U and capital L, that is the diagonal. So, diagonal entries, you just.  $D[i,j]$ , is  $A[i,j]$ , because D was  $n$  cross  $n$ ,  $n$  cross  $n$  matrix. So, diagonal entries, you replace it by  $A[i,j]$ , and similarly, in case  $i$  is strictly less than  $j$ , then you get  $U[i,j]$  is equal to  $A[i,j]$ , which is upper, strictly upper triangular, and otherwise you take  $L[i,j]$  as  $A[i,j]$ , which is strictly lower triangular. And then let us define  $D^{-1}$  to be the inverse of D, and H is defined as  $D^{-1}$  times L minus  $D^{-1}$  times U, and C is equal to  $D^{-1}$  B. And then, then what you do? You start with, let us say, call  $X_0$ , store  $X_0$  into  $X_{old}$  and then run this iteration scheme capital N times, let us run this.

JupyterLab

Integrate does not integrate - / x +

localhost:8888/lab#Math-Integrations

File Edit View Run Kernel Tabs Settings Help

CM5M\_Lecture7-5.ipynb

SageMath 9.1

```

def L(A,x):
    L[1,1]=A[1,1]
    D1=O.inverse()
    H=D1*x-D1*u
    C=D1*b
    x0=D1*x
    for i in range(1,N):
        x=H*x0+C
        x0=x
    return x

[ ]: Annmatrix(RR, [[15.0, -7.0, -2.0], [3.0, -10.0, 2.0], [2.0, -5.0, 19.0]])
Annmatrix(RR, [[9.0], [8.0], [7.0]])
Annmatrix(RR, [[0.0], [0.0], [0.0]])
Gauss_Jacobi(4,0,x0,20)

***

***

***

***

```

0 | SageMath 9.1 | idle

Saving completed

Mode: Command

IN

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 | 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 270 | 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 | 281 | 282 | 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 | 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300 | 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 | 311 | 312 | 313 | 314 | 315 | 316 | 317 | 318 | 319 | 320 | 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 | 331 | 332 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 | 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 | 351 | 352 | 353 | 354 | 355 | 356 | 357 | 358 | 359 | 360 | 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 | 371 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 | 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 | 389 | 390 | 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 | 401 | 402 | 403 | 404 | 405 | 406 | 407 | 408 | 409 | 410 | 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 | 421 | 422 | 423 | 424 | 425 | 426 | 427 | 428 | 429 | 430 | 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 | 441 | 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 | 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459 | 460 | 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 | 471 | 472 | 473 | 474 | 475 | 476 | 477 | 478 | 479 | 480 | 481 | 482 | 483 | 484 | 485 | 486 | 487 | 488 | 489 | 490 | 491 | 492 | 493 | 494 | 495 | 496 | 497 | 498 | 499 | 500 | 501 | 502 | 503 | 504 | 505 | 506 | 507 | 508 | 509 | 510 | 511 | 512 | 513 | 514 | 515 | 516 | 517 | 518 | 519 | 520 | 521 | 522 | 523 | 524 | 525 | 526 | 527 | 528 | 529 | 530 | 531 | 532 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 | 541 | 542 | 543 | 544 | 545 | 546 | 547 | 548 | 549 | 550 | 551 | 552 | 553 | 554 | 555 | 556 | 557 | 558 | 559 | 560 | 561 | 562 | 563 | 564 | 565 | 566 | 567 | 568 | 569 | 570 | 571 | 572 | 573 | 574 | 575 | 576 | 577 | 578 | 579 | 580 | 581 | 582 | 583 | 584 | 585 | 586 | 587 | 588 | 589 | 590 | 591 | 592 | 593 | 594 | 595 | 596 | 597 | 598 | 599 | 600 | 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 | 611 | 612 | 613 | 614 | 615 | 616 | 617 | 618 | 619 | 620 | 621 | 622 | 623 | 624 | 625 | 626 | 627 | 628 | 629 | 630 | 631 | 632 | 633 | 634 | 635 | 636 | 637 | 638 | 639 | 640 | 641 | 642 | 643 | 644 | 645 | 646 | 647 | 648 | 649 | 650 | 651 | 652 | 653 | 654 | 655 | 656 | 657 | 658 | 659 | 660 | 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 | 669 | 670 | 671 | 672 | 673 | 674 | 675 | 676 | 677 | 678 | 679 | 680 | 681 | 682 | 683 | 684 | 685 | 686 | 687 | 688 | 689 | 690 | 691 | 692 | 693 | 694 | 695 | 696 | 697 | 698 | 699 | 700 | 701 | 702 | 703 | 704 | 705 | 706 | 707 | 708 | 709 | 710 | 711 | 712 | 713 | 714 | 715 | 716 | 717 | 718 | 719 | 720 | 721 | 722 | 723 | 724 | 725 | 726 | 727 | 728 | 729 | 730 | 731 | 732 | 733 | 734 | 735 | 736 | 737 | 738 |

And let us now call this function for a given matrix A and B and X 0, we are taking, let us say 0, 0, 0 you could take anything.

(Refer Slide Time: 08:00)

```
def L[i,j]=A[i,j]
D1=O.inverse()
H=-D1^-1*D1*U
C=D1*B
xold=x0
for i in range(1,K):
    x=H*xold+C
    xold=x
return x

[3]: A=matrix(RR,[[[15.0,-7.0,-2.0],[3.0,-10.0,2.0],[2.0,-5.0,19.0]]])
B=matrix(RR,[[[9.0],[8.0],[7.0]]])
x0=matrix(RR,[[[1.0],[0.0],[0.0]]])
Gauss_Jacobi(4,0,20)

[3]: [ 0.390159715811917
      -0.676784814819185
      0.159187279354267]

***

***

***
```

For example, I can take  $x_1$  to be 1 and you can notice that this is a diagonally dominant matrix, and then let us call this Gauss Jacobi method, and this is what you get after 20 iterates.

(Refer Slide Time: 08:12)

```
def L[i,j]=A[i,j]
D1=O.inverse()
H=-D1^-1*D1*U
C=D1*B
xold=x0
for i in range(1,K):
    x=H*xold+C
    xold=x
return x

[4]: A=matrix(RR,[[[15.0,-7.0,-2.0],[3.0,-10.0,2.0],[2.0,-5.0,19.0]]])
B=matrix(RR,[[[9.0],[8.0],[7.0]]])
x0=matrix(RR,[[[1.0],[0.0],[0.0]]])
Gauss_Jacobi(4,0,50)

[4]: [ 0.390159657545960
      -0.676784858833690
      0.159187259512612]

[5]: A\B

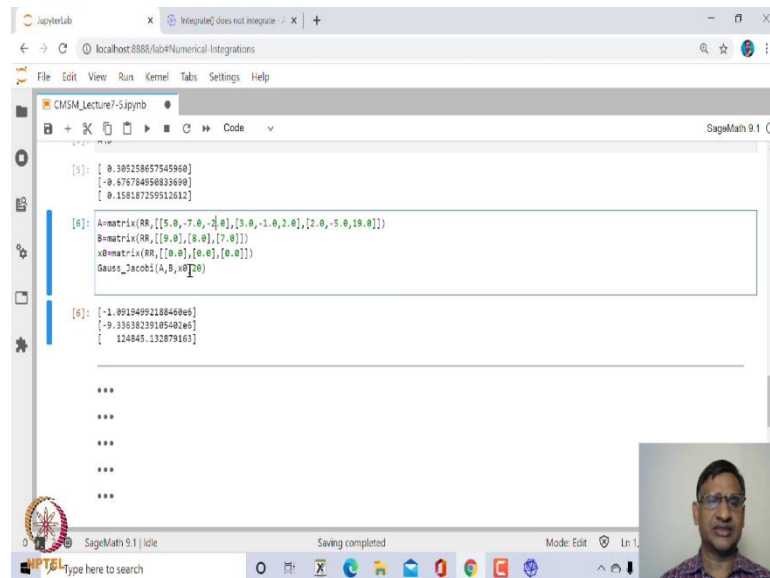
[5]: [ 0.390159657545960
      -0.676784858833690
      0.159187259512612]

***

***
```

If I take, for example, 50 iterates, this is what we get. Now, let us verify what is the solution using inbuilt function? So, you can use  $A \cdot \text{solve}$  underscore right, or  $A \backslash B$ ,  $A \backslash B$  is a command which is very similar to MATLAB command. So, this is the actual solution using inbuilt function, and what we are getting is very very close to this

actual solution. So, what we have created is, is a numeric numerical method to solve  $Ax$  equal to  $B$  using Gauss Jacobi iteration scheme. (Refer Slide Time: 08:46)



```

[5]: [ 0.305258657545960]
     [-0.676784950833890]
     [ 0.158187259512612]

[6]: A=matrix(RR, [[5.0, -7.0, -2.0], [3.0, -1.0, 2.0], [2.0, -5.0, 19.0]])
     B=matrix(RR, [[9.0], [8.0], [7.0]])
     x=matrix(RR, [[0.0], [0.0], [0.0]])
     Gauss_Jacobi(4, B, x)

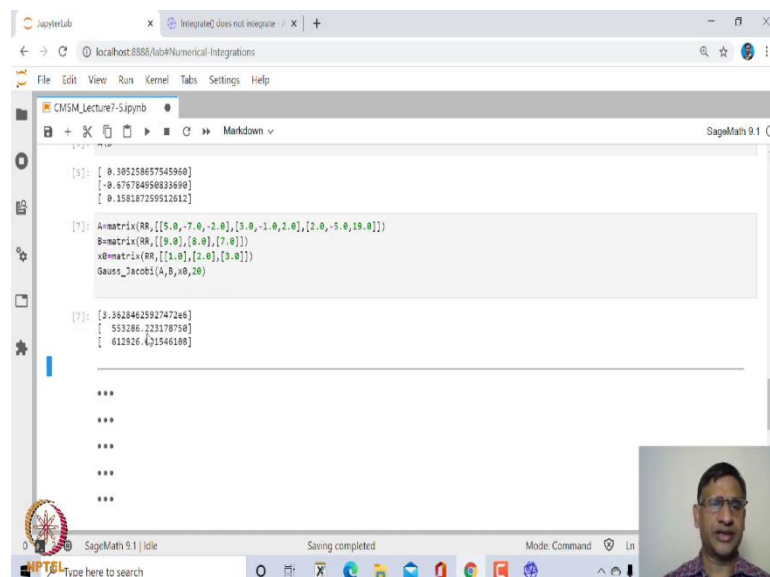
[6]: [-1.09194992186460e5]
     [-9.33638239185402e5]
     [ 124845.132879163]

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

```

Now, similarly, you can, you can try with other matrix. So, this is, let us say, if I take this matrix, and so this matrix, actually it is not diagonally dominant, you can see here, and if I take any guess value for example: 0, 0, 0, or I can take, let us say, for example, 1, 2, 3 and try with this.

(Refer Slide Time: 09:08)



```

[7]: A=matrix(RR, [[5.0, -7.0, -2.0], [3.0, -1.0, 2.0], [2.0, -5.0, 19.0]])
     B=matrix(RR, [[9.0], [8.0], [7.0]])
     x=matrix(RR, [[1.0], [2.0], [3.0]])
     Gauss_Jacobi(4, B, x)

[7]: [3.35204625927472e5]
     [ 553286.223178710]
     [ 612826.421546188]

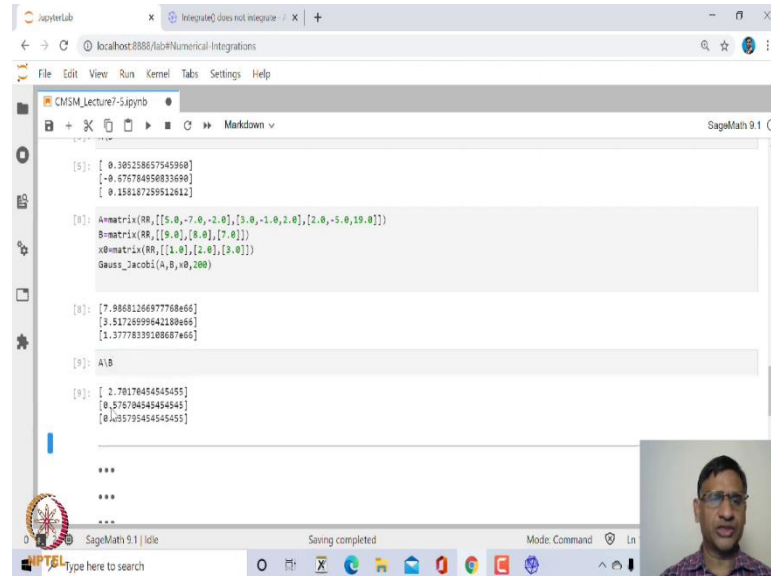
***
***
***
***

```

So, you can see here, this does not seem to be converging instead of 2, 20 iterations, if I take 200 iterations, it is becoming very very large, whereas, whereas if you try to solve

this, if you try to find what is solution of this, A backslash B, the solution is this, which is not what you are, you are getting.

(Refer Slide Time: 09:20)



```

[5]: [ 0.3052386575459160]
      [-0.6767845568336490]
      [ 0.3591872399512612]

[6]: A=matrix(RR,[[[5.0,-7.0,-2.0],[3.0,-1.0,2.0],[2.0,-5.0,19.0]])
      B=matrix(RR,[[[9.0],[8.0],[7.0]])
      x=matrix(RR,[[[1.0],[2.0],[3.0]])
      Gauss_Jacobi(4,0,100)

[7]: [ 7.98681266977768e66]
      [ 3.51726995642189e66]
      [ 1.3777839188667e65]

[8]: A\b

[9]: [ 2.70176454545455]
      [ 0.576704545454545]
      [ 0.595795454545455]

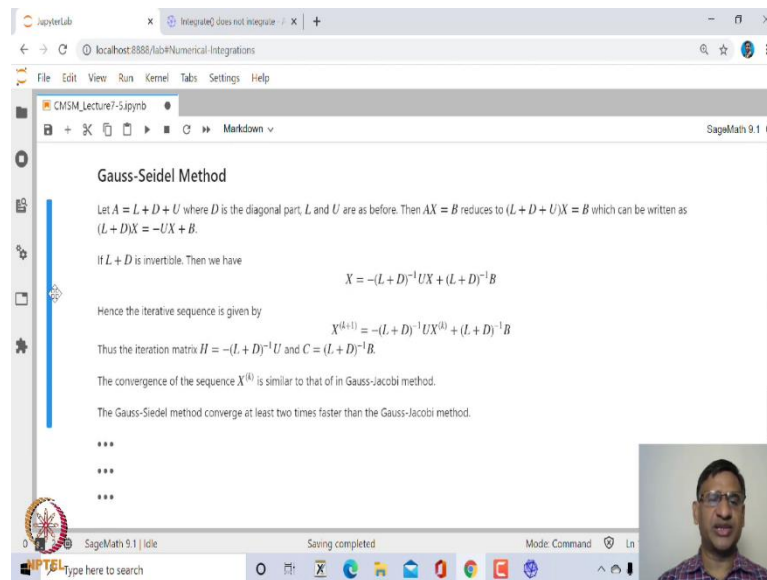
***
***
***

```

So, this, it, it also means that initial guess, in case the matrix is not diagonally dominant, you may not get solution using Gauss Jacobi method, right? So, in case, and in case the matrix is diagonally dominant, no matter what is the starting guess value, this iteration scheme will always converge. Of course, you can, you can find out what is the spectral radius of the, the iteration matrix H, and then see whether in this case spectral radius would be more than 1, right?



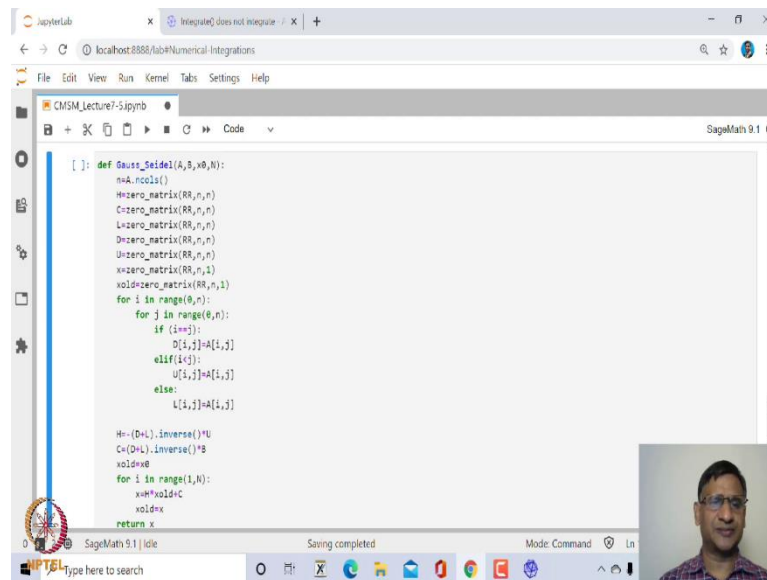
(Refer Slide Time: 10:16)



So, let us look at the next method, which is Gauss Seidel method. So, Gauss Jacobi method actually what it does is, it finds  $x_1$  from the previous iterate,  $x_2$  from the previous iterate and so on. Whereas, Gauss Seidel method  $x_1$  for example, if you are computing  $x_3$  at  $k$ th iterate, then  $x_1$ ,  $x_2$  and  $x_3$  you would have obtained in  $k$ th iterate itself. So,  $x_3$ , for computing  $x_3$ , we need the value of  $x_1$  and  $x_2$  from the  $k$ th iterate, and from, and  $x_4$  onwards, you need from the previous iterate, that is  $k-1$ th iterate. So, again, you can create this, this iteration scheme in matrix form.

So, in this case again, the splitting is exactly the same. But, however, instead of writing  $D$   $X$  on the left-hand side, now we, we write  $D$  plus  $L$  times  $X$  on the left-hand side. Therefore  $D$  plus  $L$  times  $X$  is equal to minus  $UX$  plus  $B$ .  $D$  plus  $L$  will be lower triangular matrix including the diagonal part. So, in case this  $D$  plus  $L$  is invertible, you will have  $X$  is equal to minus  $D$  plus  $L$  inverse into  $UX$  plus  $D$  plus  $L$  inverse  $B$ . So,  $H$  becomes now minus  $D$  plus  $L$  inverse times  $U$ , and  $C$  becomes  $D$  plus  $L$  inverse  $B$ . So, this is the change apart from that everything is exactly similar to Gauss Jacobi method. And in this case again, this iteration scheme converges, if and only if this spectral radius is strictly less, spectral radius of  $H$ , that is iteration matrix, is strictly less than 1, right?

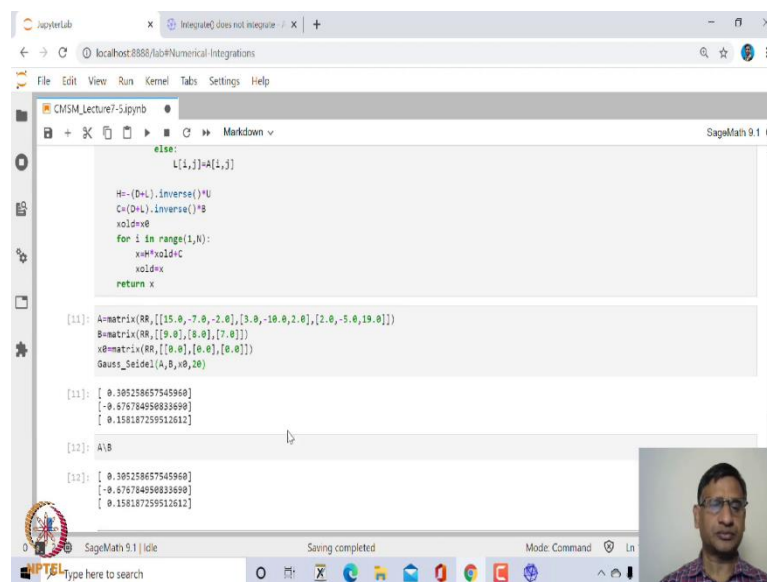
(Refer Slide Time: 12:04)



```
[ ]: def Gauss_Seidel(A,B,x0,N):  
    m=A.ncols()  
    H=zero_matrix(RR,m,m)  
    C=zero_matrix(RR,m,m)  
    L=zero_matrix(RR,m,m)  
    D=zero_matrix(RR,m,m)  
    U=zero_matrix(RR,m,m)  
    x=zero_matrix(RR,m,1)  
    xold=zero_matrix(RR,m,1)  
    for i in range(0,m):  
        for j in range(0,m):  
            if (i==j):  
                D[i,j]=A[i,j]  
            elif (i<j):  
                U[i,j]=A[i,j]  
            else:  
                L[i,j]=A[i,j]  
  
    H=-(D+L).inverse()*U  
    C=(D+L).inverse()*B  
    xold=x0  
    for i in range(1,N):  
        x=H*xold+C  
        xold=x  
    return x
```

So, let us create the program for this. So, in this case, only H and C are changing, except that everything is same. So, let me, let me just run this run this. And then show you, and then, ok.

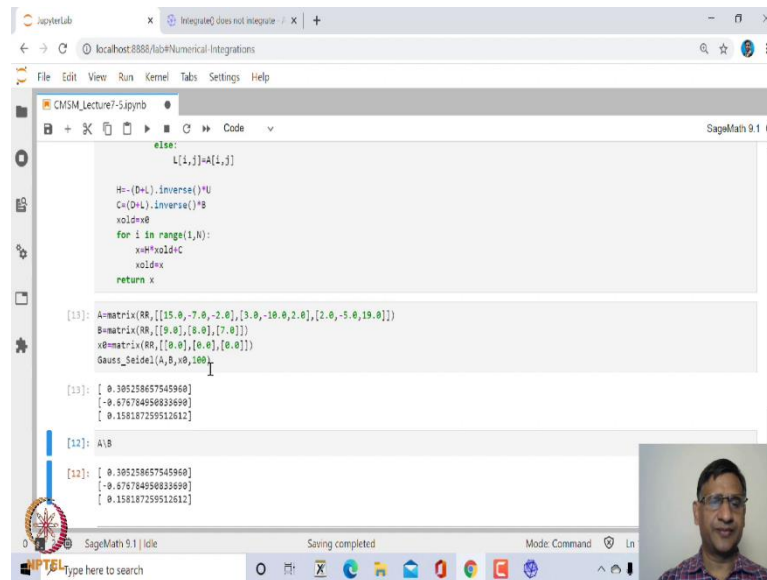
(Refer Slide Time: 12:22)



```
[ ]: A=matrix(RR,[[[15.0,-7.0,-2.0],[3.0,-10.0,2.0],[2.0,-5.0,19.0]])  
B=matrix(RR,[[[9.0],[8.0],[7.0]])  
x0=matrix(RR,[[[0.0],[0.0],[0.0]])  
Gauss_Seidel(A,B,x0,20)  
  
[11]: [ 0.305158657545960  
      [-0.676784859833690]  
      [ 0.158187259512612]  
  
[12]: A*B  
  
[12]: [ 0.305158657545960  
      [-0.676784859833690]  
      [ 0.158187259512612]
```

Now let us call this Gauss Jacobi method and one can show that the Gauss Seidel method is faster than Gauss Jacobi method. So, this is what you get, and using inbuilt function also you should get the very similar result.

(Refer Slide Time: 12:46)



```

else:
    L[1,j]=A[1,j]

H=-(D+L).inverse()*U
C=(D+L).inverse()*b
xold=x0
for i in range(1,N):
    xH=xold+C
    xold=x
return x

[13]: A=matrix(RR,[[[15.0,-7.0,-2.0],[3.0,-10.0,2.0],[2.0,-5.0,19.0]])
      b=matrix(RR,[[[9.0],[8.0],[7.0]])
      x0=matrix(RR,[[[0.0],[0.0],[0.0]])
      Gauss_Seidel(A,b,x0,100)

[13]: [ 0.305258657545960]
      [-0.676784950833690]
      [ 0.358187259512612]

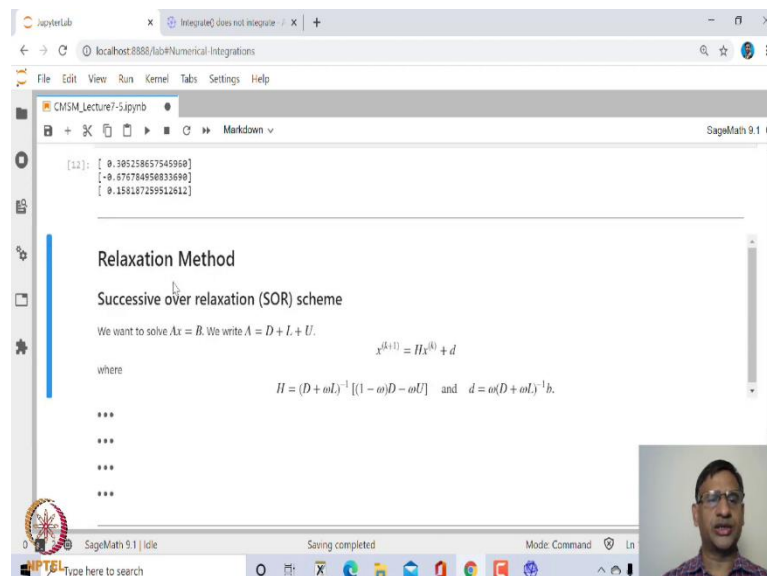
[12]: A\b

[12]: [ 0.305258657545960]
      [-0.676784950833690]
      [ 0.358187259512612]

```

So, the 2 solutions are very close to each other, and instead of, let us say 20 iterates, if I say 100 iterates, then also you will get solution which is much closer to the actual solution, right? So, these 2 are very simple methods of solving system of linear equations iteratively. There is another method which is called relaxation method.

(Refer Slide Time: 13:02)



### Relaxation Method

#### Successive over relaxation (SOR) scheme

We want to solve  $Ax = b$ . We write  $A = D + L + U$ .

$$x^{(k+1)} = Hx^{(k)} + d$$

where

$$H = (D + \omega L)^{-1} [(1 - \omega)D - \omega U] \quad \text{and} \quad d = \omega(D + \omega L)^{-1} b.$$

\*\*\*

\*\*\*

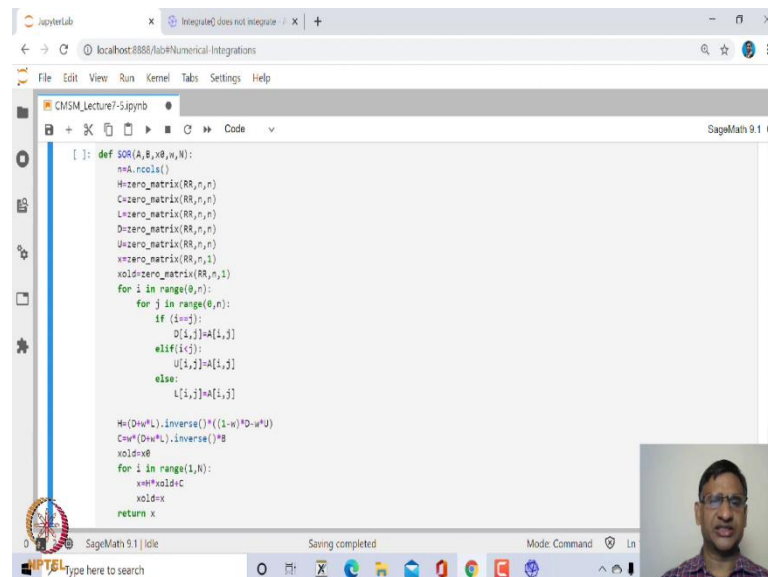
\*\*\*

\*\*\*

So, in this relaxation method the, in again this iteration scheme, the H and D, so we write this iteration scheme of AX equal to, AX plus, AX equal to B as, again the same splitting A as D plus L plus U and  $x_{k+1}$  will become  $H^{-1}(I - \omega L)^{-1}(I - \omega U)x_k + \omega D^{-1}b$ , where capital H is given by  $D + \omega L$  inverse into  $1 - \omega$ , this is omega,  $D + \omega L$  inverse into  $1 - \omega$  D plus omega U, this, that H, omega is known as relaxation factor. In this case, we are actually, what is called successive over relaxation scheme.

And D is given by this. So, and one can show that this, this method actually it converges for, under certain conditions.

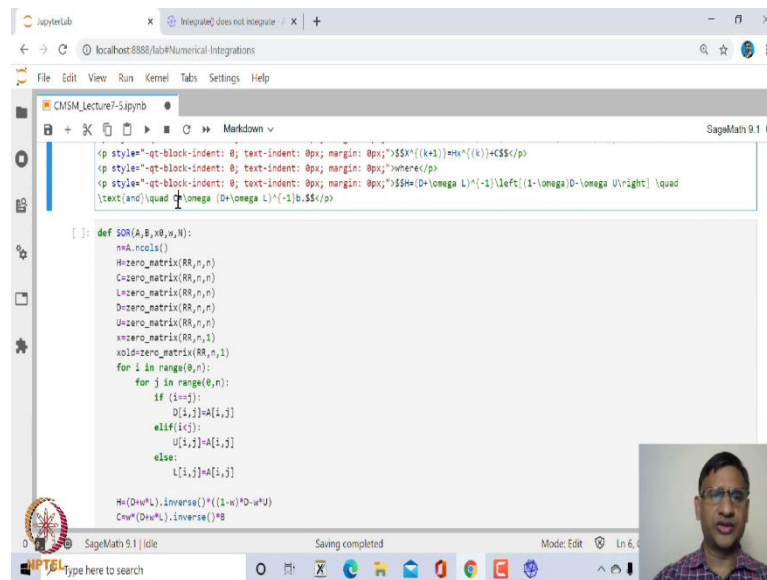
(Refer Slide Time: 13:57)



```
def sor(A,b,x0,w,N):
    m,n= A.ncols()
    H=zero_matrix(RR,n,n)
    L=zero_matrix(RR,n,n)
    U=zero_matrix(RR,n,n)
    x=zero_matrix(RR,n,1)
    xold=zero_matrix(RR,n,1)
    for i in range(0,n):
        for j in range(0,n):
            if (i==j):
                H[i,j]=A[i,j]
            elif(i<j):
                U[i,j]=A[i,j]
            else:
                L[i,j]=A[i,j]
    H=(H+U*L).inverse()*((1-w)*D+U)
    D=w*(D+L).inverse()*b
    xold=x0
    for i in range(1,N):
        x=H*xold+C
        xold=x
    return x
```

So, I will just mention that separately, so again, let us create a user-defined function for SOR, Successive Over Relaxation scheme. So, x everything is again same except that, except that H and D changes.

(Refer Slide Time: 14:23)



The screenshot shows a JupyterLab window with a SageMath 9.1 notebook. The notebook has two cells. The first cell contains LaTeX code for the SOR method, including the definition of the iteration matrix  $H$  and the iteration formula  $x^{(k+1)} = Hx^{(k)} + C$ . The second cell contains a Python function definition for the SOR method, which takes matrices  $A$ ,  $B$ ,  $x_0$ , and  $\omega$  as inputs and returns the solution vector  $x$ .

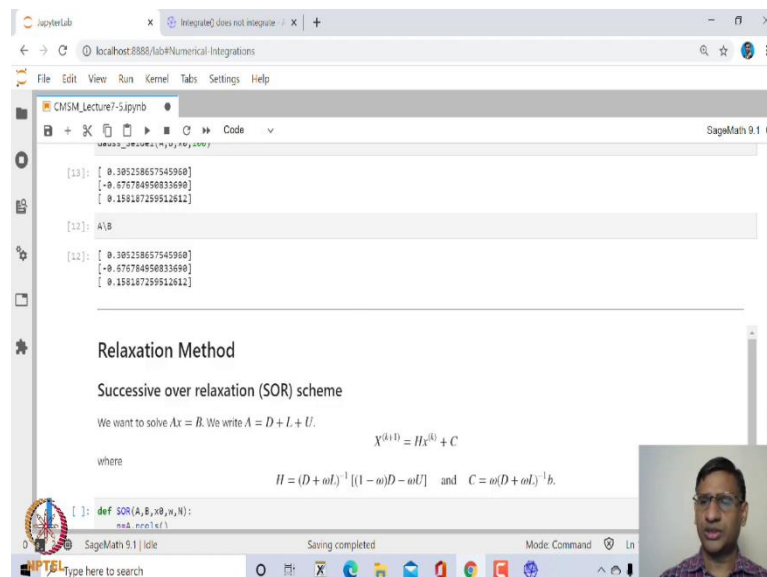
```

[ ]: def SOR(A,B,x0,w,N):
    n=A.ncols()
    H=zero_matrix(RR,n,n)
    C=zero_matrix(RR,n,n)
    L=zero_matrix(RR,n,n)
    D=zero_matrix(RR,n,n)
    U=zero_matrix(RR,n,n)
    x=zero_matrix(RR,n,1)
    xold=zero_matrix(RR,n,1)
    for i in range(0,n):
        for j in range(0,n):
            if (i==j):
                D[i,j]=A[i,j]
            elif(i<j):
                U[i,j]=A[i,j]
            else:
                L[i,j]=A[i,j]
    H=(D+omega*L).inverse()*((1-omega)*D+omega*U)
    C=(D+omega*L).inverse()*B

```

So, let us, H and C changes. So, there, just for uniformity, let me write this capital X, and this to be capital C here. So, this is, this becomes capital C. So, that it will be uniform right now. So, again in this case, we are just writing D, capital H and capital C using that formula above, rest everything are same.

(Refer Slide Time: 14:34)



The screenshot shows the JupyterLab interface with the output of the SOR method. The output displays the iteration matrix  $H$  and the iteration formula  $x^{(k+1)} = Hx^{(k)} + C$ . Below the output, there is a slide titled "Relaxation Method" which discusses the Successive over relaxation (SOR) scheme. The slide states that we want to solve  $Ax = B$  and we write  $A = D + L + U$ , where  $D$  is the diagonal part of  $A$ ,  $L$  is the lower triangular part, and  $U$  is the upper triangular part. The slide also provides the formulas for  $H$  and  $C$  in terms of  $D$ ,  $L$ , and  $U$ .

**Relaxation Method**

Successive over relaxation (SOR) scheme

We want to solve  $Ax = B$ . We write  $A = D + L + U$ .

$$x^{(k+1)} = Hx^{(k)} + C$$

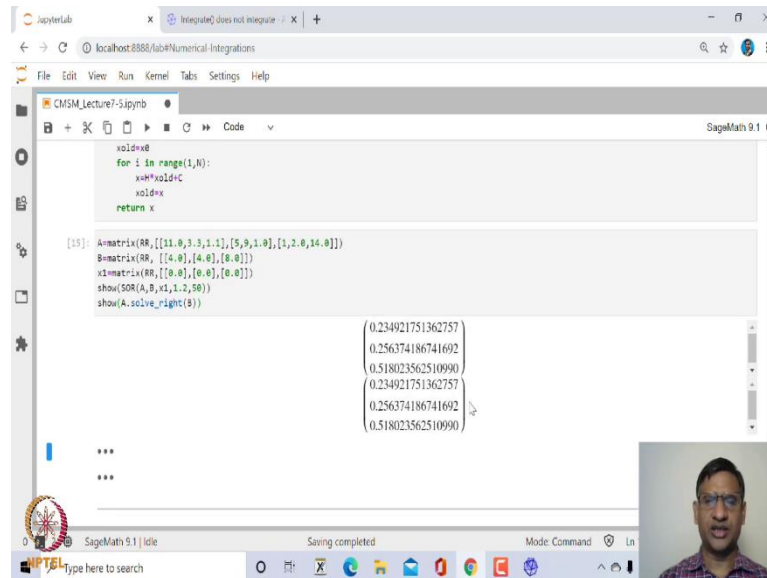
where

$$H = (D + \omega L)^{-1} [(1 - \omega)D - \omega U] \quad \text{and} \quad C = \omega(D + \omega L)^{-1} b.$$

So, how do we, how do we write? So, in this case, the input we have to give is A, B, X 0 and omega, the value of omega. That is the, the, the term which, which is the relaxation

term and then the number of iterates. So, once we, we, we let us say run this program, and then we can now call it. So, let us call this program and see, what is the solution?

(Refer Slide Time: 15:19)



```

def voidval:
    for i in range(1,N):
        x[i]=x[i]+C
    return x

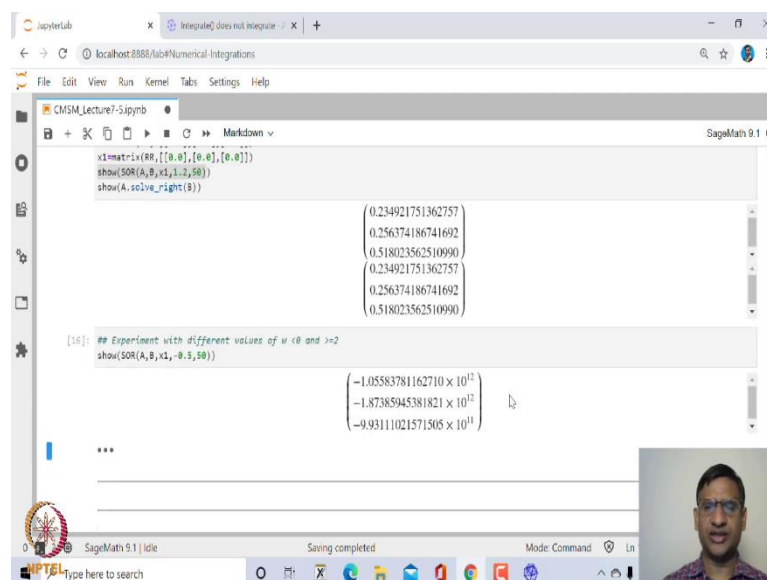
[15]: A=matrix(RR,[[[11.0,3.3,1.1],[5.9,1.0],[1.2,0.14,0]])
      B=matrix(RR,[[[4.0],[4.0],[0.0]])
      x1=matrix(RR,[[[0.0],[0.0],[0.0]])
      show(SOR(A,B,1,1.2,50))
      show(A.solve_right(B))

```

$$\begin{pmatrix} 0.234921751362757 \\ 0.256374186741692 \\ 0.518023562510990 \\ 0.234921751362757 \\ 0.256374186741692 \end{pmatrix}$$

So, in this case, you can see here the, both the solutions are also very close to each other using inbuilt function A dot solve and using SOR method, right?

(Refer Slide Time: 15:38)



```

x1=matrix(RR,[[[0.0],[0.0],[0.0]])
show(SOR(A,B,1,1.2,50))
show(A.solve_right(B))

[16]: ## Experiment with different values of w <0 and >2
      show(SOR(A,B,1,-0.5,50))

```

$$\begin{pmatrix} 0.234921751362757 \\ 0.256374186741692 \\ 0.518023562510990 \\ 0.234921751362757 \\ 0.256374186741692 \end{pmatrix}$$

$$\begin{pmatrix} -1.05583781162710 \times 10^{12} \\ -1.87385945381821 \times 10^{12} \\ -9.93111021571505 \times 10^{14} \end{pmatrix}$$

Now, suppose you, you take this and choose the various values of, let us say, omega. So, for example, let us say, choose omega to be minus 0.5, in case we choose omega to be

minus 0.5, you can see here this, this scheme does not converge, or if I choose omega to be, let us say, let us say 3. Then also this does not converge.

(Refer Slide Time: 16:01)

```

x1=matrix(RR, [[0,0],[0,0],[0,0]])
show(SOR(A,B,x1,1.2,50))
show(A.solve_right(B))

[17]: ## Experiment with different values of w < 0 and > 2
show(SOR(A,B,x1,3,50))

***

```

However, if we choose omega, let us say 0.5, then this converges.

(Refer Slide Time: 16:06)

```

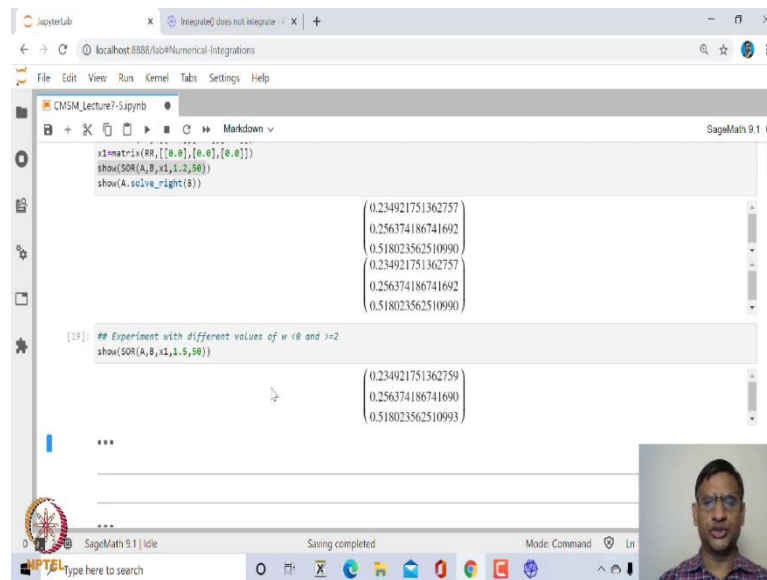
x1=matrix(RR, [[0,0],[0,0],[0,0]])
show(SOR(A,B,x1,1.2,50))
show(A.solve_right(B))

[18]: ## Experiment with different values of w < 0 and > 2
show(SOR(A,B,x1,0.5,50))

***

```

(Refer Slide Time: 16:13)



```

x1=matrix(RR,[[[0,0],[0,0],[0,0]]])
show(SOR(A,B,x1,1.2,50))
show(A.solve_right(B))

```

$$\begin{pmatrix} 0.234921751362757 \\ 0.256374186741692 \\ 0.518023562510990 \\ 0.234921751362757 \\ 0.256374186741692 \\ 0.518023562510990 \end{pmatrix}$$

```

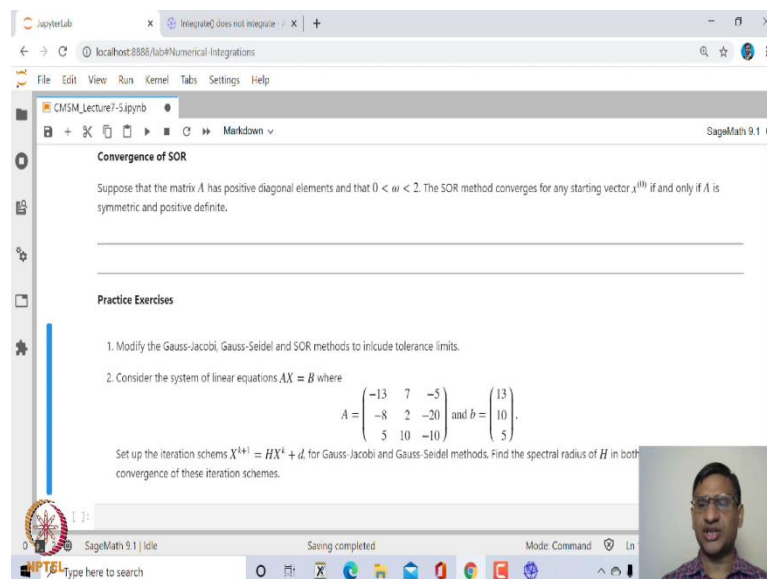
[18]: ## Experiment with different values of w < 0 and >= 2
show(SOR(A,B,x1,1.5,50))

```

$$\begin{pmatrix} 0.234921751362759 \\ 0.256374186741690 \\ 0.518023562510993 \end{pmatrix}$$

If we choose omega to be, let us say, anything between 0 and 2, this scheme will converge.

(Refer Slide Time: 16:19)



### Convergence of SOR

Suppose that the matrix  $A$  has positive diagonal elements and that  $0 < \omega < 2$ . The SOR method converges for any starting vector  $x^{(0)}$  if and only if  $A$  is symmetric and positive definite.

### Practice Exercises

1. Modify the Gauss-Jacobi, Gauss-Seidel and SOR methods to include tolerance limits.
2. Consider the system of linear equations  $AX = B$  where
 
$$A = \begin{pmatrix} -13 & 7 & -5 \\ -8 & 2 & -20 \\ 5 & 10 & -10 \end{pmatrix} \text{ and } b = \begin{pmatrix} 13 \\ 10 \\ 5 \end{pmatrix}.$$
 Set up the iteration schemes  $X^{k+1} = HX^k + d$  for Gauss-Jacobi and Gauss-Seidel methods. Find the spectral radius of  $H$  in both convergence of these iteration schemes.

So, that is a necessary condition for this SOR method. So, what does it say? If you have a matrix  $A$  which has positive diagonal elements, and you take any omega between 0 and, strictly between 0 and 2, then this SOR method converges for any starting guess value  $x_0$ , if and only if  $A$  is symmetric and positive definite.

So, in case  $A$  is symmetric and positive definite, then no matter what is your omega, which I mean, whatever omega you choose between 0 and 2, and then take any initial guess value,



this SOR method scheme converges. So, there are several other methods as well, but we will not go into all these methods.

So, these are 3 methods which I thought I will cover. So, one is Gauss Jacobi method, other one is Gauss Seidel method and one can show that Gauss Seidel method is faster than Gauss Jacobi; that means, the convergence is, I mean faster than Gauss, convergence is, Gauss Seidel is faster than Gauss Jacobi.

And then you have the relaxation method, over relaxation method SOR and all these methods are generally, it is very simple to implement. Of course, when you, when you solve this, solve  $Ax = B$  using inbuilt SageMath method that is `A.solve_right(B)`, you do not know what method it is using, but it is quite simple to create these methods.

Now, let me leave you with few exercises. 1 exercise is, I already gave you to, to create Sage subroutine in order to check, whether a given matrix is diagonally, strictly diagonally dominant. And you can also, for example, the, the method that we have created, the subroutine which we have created, it uses the number of iterations for a stopping criteria.

You could also include the tolerance limit. So, tolerance, how will you include? The difference between the 2 successive iterates, because this, the successive iterates are vectors. So, difference we have to take as norm. So, norm of the difference between 2 successive iterates, if it is less than some predefined number, then you stop.

So, that in, in the above program, you can, you can incorporate, you can use while loop and also include this tolerance limit. And the next problem is, consider the system of linear equations for this, write this iteration scheme and then find out what is spectral radius of  $H$  and then check whether this system converges. Let me stop here.

Thank you very much.