```
#
# This lecture contains
#
# a brief discussion about Taylor Expansion
#
# and then
#
# we move to chapter 6 of the book
# and study SIMULTANEOUS LINEAR EQUATIONS


# ---------------------------------------------------------------------------
# ------------------
# TAYLOR EXPANSION
# ------------------

# By using a Taylor expansion we can approximate a given function f(x)
# at a certain point x=a
# by a polynomial.


# The concept of a Taylor series was formulated by the Scottish mathematician
# James Gregory and formally introduced by the English mathematician
# Brook Taylor in 1715. If the Taylor series is centered at zero,
# then that series is also called a Maclaurin series.

# The polynomial that approximates f(x) at x=a is the
# nth degree TAYLOR polynomial:
# P(x) = f(a) + f'(a)(x-a)+(f''(a))/2! (x-a)^2+ ... + (f(a)^(n) (a))/n! (x-a)^n

# Example:
# sin(x) close to x=0 is
# sin(x) ~ x - x^3/3!  +  x^5/5!  ...
# With this expansion, it becomes easy to study the limit sin(x)/x


# ------
# SYMPY
# ------
# In Python, we can use SYMPY to get Taylor expansions.
# Example: Expansion of cos(x) around x=0

import sympy as syp
x = syp.Symbol('x')

# choose n=5 if you want the expansion up to x^4
ff=syp.series(syp.cos(x),x,x0=0,n=5)
print( 'cos(x) ~', ff  )

f2 = ff.removeO()
print( 'cos(x) up to x^3: ', f2  )
```

```
print()
# -----------------------------------------------------------------
# CAREFUL!! ff and f2 are symbolic expressions
# We cannot use them as functions
# Check what happens with the lines below

# NOT SOLVED!!
print("The function is not solved!")
def func(x):
    return f2
print(func(3))


print()
# WE COULD COPY THE FUNCTION
print('Solved because we copied the function')
x=3
gg = x**4/24 - x**2/2 + 1
print('Taylor of cos up to x^4 with x=3: ', gg)


# WE CAN LAMBDIFY THE EXPRESSION
#
# CAREFUL!! Since above x became the number 3
# Say again that "x" is a SYMBOL
x = syp.Symbol('x')
# To convert a SymPy expression to an expression that can be
# numerically evaluated, use the lambdify function.
print()
print("Now it is solved, because we lambdified the function!")
g2 = syp.lambdify(x,f2)
print(g2(3))


# Now that we have g2 as a FUNCTION of x
# We can get g2 for various values of x
# -----------------------------------------------------------------
import numpy as np
x = np.arange(1, 2.1, 0.1)
g2(x)
print()
print("And with numpy we can get all values at once")
print(x)
print(g2(x))



# -----------
# Exercise 1
# -----------
```

```
# Make a plot of sin(x) and
# the first, third, fifth and seventh degree Taylor polynomials
# for x from x=-3.5 to x=3.5 in increments 0.01.
# Label your curves.




# ---------------------------------------------------------------
# After this review of Taylor expansion, we can understand why
# the central difference gives a better approximation to the
# derivative of a function than
# the forward or backward differences
# See Sec.5.10.2 and Sec.5.10.3 of the book.
# ---------------------------------------------------------------







# ----------------------------------------------------------------------------
# ----------------------------------------------------------------------------






print()
# ---------------------------------------------------------------
#                   CHAPTER 6 of the book
# ---------------------------------------------------------------
# NOTE: chapters beyond chapter 5 are NOT available online!!
# To read them, you can get the book in the library



# Suppose we want to solve the following four simultaneous equations
# for the variables w, x, y, and z
#
# 2w +  x + 4y +  z = - 4
# 3w + 4x -  y -  z =   3
#  w - 4x +  y + 5z =   9
# 2w - 2x +  y + 3z =   7

# which can be written in a matrix form as
#              A x = v
# where
import numpy as np
A = np.array([[2, 1, 4, 1],
```

```
              [3, 4, -1, -1],
              [1, -4, 1, 5],
              [2, -2, 1, 3]], float)
v = np.array([-4, 3, 9, 7], float)




print("Using 'solve' from numpy.linalg")
# -----------------
# LINALG and SOLVE
# -----------------
# This can be done with the
# module LINALG of the
# NUMPY package
# with the function SOLVE
import numpy.linalg as npa
x = npa.solve(A,v)
print('w, x, y, z = ',x)


print()
print("Using 'inv' from numpy.linalg")
# -----------------
# LINALG and INV
# -----------------
# We can also find the inverse of A
# and use A^{-1}.A.x = x = A^{-1}.v
# to find v.
# The inverse is also contained in numpy.linalg
# and is called "inv"
Ainv = npa.inv(A)
sol = np.dot(Ainv,v)
print('w, x, y, z = ',sol)
# BUT, calculating the inverse of a matrix is a
# slow process. Unless the inverse is really needed,
# it is better to avoid it.




#%%
# WHAT IS BEHIND THE FUNCTION SOLVE?
# Python uses the LU decomposition and backsubstitution.
# To understand what this is, let us start with
# Gaussian elimination and backsubstitution

# --------------------
# GAUSSIAN ELIMINATION
# --------------------

# Follow the PDF notes called "GaussianElimination"
```

```
# and write a code to find w, x, y, and z for the
# system of linear equations written above.

# The purpose of the Gaussian elimination is to write
# the matrix A as an upper triangular matrix,
# so that w, x, y, z can be obtained by backsubstitution.


import numpy as np

A = np.array([[2, 1, 4, 1],
              [3, 4, -1, -1],
              [1, -4, 1, 5],
              [2, -2, 1, 3]], float)
v = np.array([-4, 3, 9, 7], float)

Ntot = len(v)

# Gaussian Elimination
# -----------------------------------------------------------------------
for n in range(Ntot):
    # Divide the row by the diagonal element
    # to get the element 1.
    div = A[n,n]
    # NOTE: that we can do the operation on the entire row using ':' as below
    # Alternatively, we could have a loop here.
    A[n,:] = A[n,:]/div
    v[n] = v[n]/div
    # The simplified notation below do the same as above
    # A[n,:] /= div
    # v[m] /= div

    # PRINT to be sure it is doing what we want
#    print()
#    print('n=',n)
#    print('A[n,:]=', A[n,:], ' and v[n]=', v[n])

# -----------------------------------------------------------------------
    # Now we do (lower row) - (num)*(row just divided by diagonal element)
    # to get the element 0.
    for k in range(n+1,Ntot):
        mult=A[k,n]
        A[k,:] = A[k,:] - mult*A[n,:]
        v[k] = v[k] - mult*v[n]
        # The simplified notation below do the same as above
        # A[k,:] - = mult*A[n,:]
        # v[m] - = mult*v[n]
        # PRINT to be sure it is doing what we want
#        print('k=',k)
#        print('A[k,:]=', A[k,:], ' and v[k]=', v[k])
```

```
# ------------------------------------------------------------------------
# BACKSUBSTITUTION
# create an array of zeros, where the solution will be stored
x = np.zeros(Ntot,float)
# n below goes from n=Ntot-1=3 to n=0 (one before the last term -1)
for n in range(Ntot-1,-1,-1):
    x[n] = v[n]
    # k below goes from k=n+1 to k=Ntot-1 (one before the last term Ntot)
    for k in range(n+1,Ntot):
        x[n] = x[n] - A[n,k]*x[k]


print('w, x, y, z =', x)



# --------------------
# PIVOTING
# --------------------

# If an element of A is zero, which would lead to a division by zero,
# we swap the row with another one that has the farthest element from zero.
# See the PDF notes called "GaussianElimination".


# --------------------
# LU DECOMPOSITION
# --------------------

# Follow the PDF notes called "LUdecomposition"
# to understand what this decomposition is.
# The basic idea is to write the A matrix as a
# product of two matrices
#            A = L U
# where
#            L is a lower triangular matrix
# and
#            U is an upper triangular matrix
#
# In fact, U is the matrix that we obtain after the
# Gaussian elimination
#            U = L3.L2.L1.L0.A
#
# and        L = L0^{-1}.L1^{-1}.L2^{-1}.L3^{-1}



# -------------------------------
# TRIDIAGONAL and BANDED matrices
# -------------------------------
```

```
# When we have tridiagonal matrices or banded matrices,
# the code for the Gaussian elimination can (and should) be simplified!
#
# For a tridiagonal problem, each row only needs to be subtracted
# from the single row immediately below it!!
```