LAB RECORD

2018 – 2019

Mathematical Models Using Python Programming

MAT 451

Name: Jeevan Koshy **Reg. No:** 1740256

TABLE OF CONTENTS

Lab 2

Topic: Inverse, Determinant & Eigen Values

Date: 15th November 2018

Aim: To find the determinant, inverse and eigen values of matrices.

Source Code:

```
import numpy as np
matrix = np.matrix([[1,4],[2,0]])
det=np.linalg.det(matrix)
print(det)

A=([[1,5,6,7],[8,9,1,0],[2,3,4,5],[4,5,2,3]])
A_det=np.linalg.det(A)
print(A)
print(A_det)
```

```
inverse=np.linalg.inv(matrix)
A_inv=np.linalg.inv(A)
print(inverse)
print(A_inv)
B=np.linalg.inv(A_inv)
print(B)
```

```
A=([[1,3,2],[2,3,1],[4,2,1]])
B=([[2,4,6],[3,2,1],[7,6,2]])
det_A=np.linalg.det(A)
det_B=np.linalg.det(B)
print(det_A)
print(det_B)
inv_A=np.linalg.inv(A)
inv_B=np.linalg.inv(B)
print(inv_A)
print(inv_B)
AB=np.dot(A,B)
AB_INV=np.linalg.inv(AB)
B_INV_A_INV=np.dot(inv_B,inv_A)
print(B_INV_A_INV)
print(AB_INV)
```

```
E=np.matrix([[1,4,7],[9,1,9],[0,0,1]])
eigvals=np.linalg.eigvals(E)
print(eigvals)
```

Output (Graphs/Tables):

```
-8.0
[[1, 5, 6, 7], [8, 9, 1, 0], [2, 3, 4, 5], [4, 5, 2, 3]]
-86.0
[[ 0.     0.5  ]
 [ 0.25  -0.125]]
[[-0.39534884  0.09302326  0.65116279 -0.1627907 ]
 [ 0.34883721 -0.02325581 -0.6627907   0.29069767]
 [ 0.02325581  0.46511628  0.75581395 -1.31395349]
 [-0.06976744 -0.39534884 -0.26744186  0.94186047]]
[[  1.00000000e+00   5.00000000e+00   6.00000000e+00   7.00000000e+00]
 [  8.00000000e+00   9.00000000e+00   1.00000000e+00   2.35922393e-16]
 [  2.00000000e+00   3.00000000e+00   4.00000000e+00   5.00000000e+00]
 [  4.00000000e+00   5.00000000e+00   2.00000000e+00   3.00000000e+00]]
-9.0
24.0
[[-0.11111111 -0.11111111  0.33333333]
 [-0.22222222  0.77777778 -0.33333333]
 [ 0.88888889 -1.11111111  0.33333333]]
[[-0.08333333  1.16666667 -0.33333333]
 [ 0.04166667 -1.58333333  0.66666667]
 [ 0.16666667  0.66666667 -0.33333333]]
[[-0.5462963   1.28703704 -0.52777778]
 [ 0.93981481 -1.97685185  0.76388889]
 [-0.46296296  0.87037037 -0.27777778]]
[[-0.5462963   1.28703704 -0.52777778]
 [ 0.93981481 -1.97685185  0.76388889]
 [-0.46296296  0.87037037 -0.27777778]]
[ 7. -5.  1.]
```

Conclusion:

From the above output, we have calculated the determinants of a matrix as well as found it’s eigen values with it’s inverse. Using these codes, we can find for any matrix – it’s determinant, inverse & eigen value.

Lab 3

Topic: Transpose & Upper/Lower Triangular Parts

Date: 17th November 2018

Aim: To find the transpose, upper & lower triangular parts of a matrix.

Source Code:

```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a)
tri_upper_diag=np.triu(a,k=0)
print(tri_upper_diag)
tri_upper_diag_no_diag=np.triu(a,k=1)
print(tri_upper_diag_no_diag)
tri_upper_diag_no_diag=np.triu(a,k=2)
print(tri_upper_diag_no_diag)
tri_upper_diag_no_diag=np.triu(a,k=3)
print(tri_upper_diag_no_diag)
```

```
# Program to transpose a matrix using nested loop

X = [[12,7],
    [4 ,5],
    [3 ,8]]

result = [[0,0,0],
         [0,0,0]]

# iterate through rows
for i in range(len(X)):
   # iterate through columns
   for j in range(len(X[0])):
       result[j][i] = X[i][j]

for r in result:
   print(r)
```

Output (Graphs/Tables):

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 2 3]
 [0 5 6]
 [0 0 9]]
[[0 2 3]
 [0 0 6]
 [0 0 0]]
[[0 0 3]
 [0 0 0]
 [0 0 0]]
[[0 0 0]
 [0 0 0]
 [0 0 0]]
[12, 4, 3]
[7, 5, 8]
```

Conclusion:

From the above codes, we have found the transpose, upper & lower triangular parts of a matrix.

Lab 4

Topic: Solving Linear Systems

Date: 22nd November 2018

Aim: To solve linear systems in Python

Source Code:

Example 01. Solve y" + y' - 2y = 0

In [14]:
```
print("Solution to the given linear differential equation is given by: ")
c1 = sympy.Symbol('c1')
c2 = sympy.Symbol('c2')
x = sympy.Symbol('x')
m = np.poly([0])
f = m**2+m-2
coeff = [1,1-2]
r = np.roots(coeff)
y = c1*sympy.exp(r[0]*x)+c2*sympy.exp(r[0]*x)
print("y = ",y)
```

Example 02. Solve y" + 6y' + 9y = 0

In [3]:
```
print("Solution to the given linear differential equation is given by: ")
c1 = sympy.Symbol('c1')
c2 = sympy.Symbol('c2')
x = sympy.Symbol('x')
m = np.poly([0])
f = m**2+6*m+9;
r=np.roots([1,6,9])
y=(c1+x*c2)*sy.exp(r[0]*x)
print(r)
print("y = ",y)
```

Linear Differential equations with constant coefficients

Q1: Solve the differential equation $\frac{d^2}{dx^2} - 5f = 0$

In [16]:
```
#to print neatly
from sympy.interactive import printing
printing.init_printing(use_latex=True)
import sympy as sp
from sympy import *
```

In [17]:
```
x=Symbol('x')
x                    #defining symbol using symbol
```

Out[17]: x

```
In [18]: from sympy import *
         f = Function('f')(x)                    #defining f as a function of x
         diffeq = Eq(f.diff(x,x)-5*f,0)          #define the differential equation
         diffeq                        #to print the differential equation
```

Out[18]: $-5f(x)+\frac{d^2}{dx^2}f(x)=0$

```
In [19]: dsolve(diffeq,f)          #solving the differential equation
```

Q2: Solve the differential equation $\frac{d^2y}{dx^2}+5\frac{dy}{dx}+6=0$

```
In [20]: x = Symbol('x')
         y = Function('y')(x)               #defining y as a function of x
         diffeq = Eq(y.diff(x,x)+5*y.diff(x)+6*y,0)      #define the #define the differential equation
         print('The differential equation is: ')
         print(diff)
         print('The solution is: ')
         dsolve(diffeq,y)
```

```
In [21]: x = Symbol('x')
         y = Function('y')(x)               #defining y as a function of x
         diffeq = Eq(y.diff(x,x,x)+y,0)     #define the differential equation
         print('The differential equation is: ')
         print(diff)
         print('The solution is: ')
         dsolve(diffeq,y)
```

Q6. Solve (D^2 + 4)y = cos3x

```
In [22]: x = Symbol('x')
         y = Function('y')(x)               #defining y as a function of x
         diffeq = Eq(y.diff(x,x)+4*y,cos(3*x))      #define the differential equation
         print('The differential equation is: ')
         print(diff)
         print('The solution is: ')
         dsolve(diffeq,y)
```

Q7: Solve (D^2 +4D + 4)y = e^(-3x)

```
In [23]: diffeq = Eq(y.diff(x,x)+4*y.diff(x)+4*y,exp(-3*x))      #define the differential equation
         print('The differential equation is: ')
         print(diff)
         print('The solution is: ')
         dsolve(diffeq,y)
```

1. Solve $\frac{dy}{dx}=x-y, y_0=1$

```
In [27]: x = sy.Symbol('x')
         f = sp.Function('f')(x)
         diffe = Eq(f.diff(x)-x+f,0)
         diffe
         dsolve(diffe,f)
```

Linear Differential equations using ODEINT

1. Model: Fuction name that returns derivative values at requested y and t values as dy_dt = model(y,t)

2. y0: Initial conditions of the differential states

3. t: Time points at which the solution should be reported. Additional internal points are often calculated to maintain accuracy of the solution but are not reported.

Solve $\frac{dy}{dx}=x-y, y_0=1$

```
In [6]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        from scipy.integrate import odeint
```

```
In [8]: #Define a function that calculates the derivative
        def dy_dx(y,x):
            return x-y
        xs = np.linspace(0,10,100)
        y0 = 1.0 #the initial condition
        ys = odeint(dy_dx,y0,xs)
```

In [9]:
```
#plot results
plt.plot(xs,ys)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
print(ys)
```

Solve $\frac{dy}{dx} = -ky(t)$ with parameter k = 0.1,0.2,0.5 and the initial condition $y_0 = 5$

In [12]:
```
#function that returns dy/dt
def model(y,t):
    k=0.2
    dydt = -k*y
    return dydt

# initial condition
y0=5

# time points
t = np.linspace(0,20,100)

#solve ODE
y = odeint(model,y0,t)

#plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
print(y)
```

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.integrate import odeint

#function that returns dy/dt
def model(y,t,k):
    dydt = -k*y
    return dydt

# initial condition
y0=5

# time points
t = np.linspace(0,20)

#solve ODE
k = 0.1
y1 = odeint(model,y0,t,args=(k,))
k = 0.2
y2 = odeint(model,y0,t,args=(k,))
k = 0.5
y3 = odeint(model,y0,t,args=(k,))


#plot results
plt.plot(t,y1,'r-',linewidth=2,label='k=0.1')
plt.plot(t,y2,'b--',linewidth=2,label='k=0.1')
plt.plot(t,y3,'g.',linewidth=2,label='k=0.5')
plt.xlabel('time')
plt.ylabel('y(t)')
plt.legend()
plt.show()
```

In [14]:
```
#function that returns dy/dt
def model(y,t):
    if t<10.0:
        u=0
    else:
        u=2
    dydt = (-y + u)/5.0
    return dydt

# initial condition
y0 = 0

# time points
t = np.linspace(0,40,1000)

#solve ODE
y = odeint(model,y0,t)

#plot results
plt.plot(t,y,'r-',label='Output (y(t))')
plt.plot([0,10,10,40],[0,0,2,2],'b-',label='Input (u(t))')
plt.xlabel('values')
plt.ylabel('time')
plt.legend(loc='best')
plt.show()
```

Output (Graphs/Tables):

```
Solution to the given linear differential equation is given by:
y = c1*exp(1.0*x) + c2*exp(1.0*x)
Solution to the given linear differential equation is given by:
[-3. +3.72529030e-08j -3. -3.72529030e-08j]
y =  (c1 + c2*x)*exp(x*(-3.0 + 3.72529029846191e-8*I))
```

```
The differential equation is:
<function diff at 0x000002719CE4E268>
The solution is:
```

Out[20]: $y(x) = (C_1 + C_2 e^{-x})\, e^{-2x}$

```
The differential equation is:
<function diff at 0x000002719CE4E268>
The solution is:
```

Out[21]: $y(x) = C_3 e^{-x} + \left(C_1 \sin\left(\frac{\sqrt{3}x}{2}\right) + C_2 \cos\left(\frac{\sqrt{3}x}{2}\right)\right)\sqrt{e^x}$

```
The differential equation is:
<function diff at 0x000002719CE4E268>
The solution is:
```

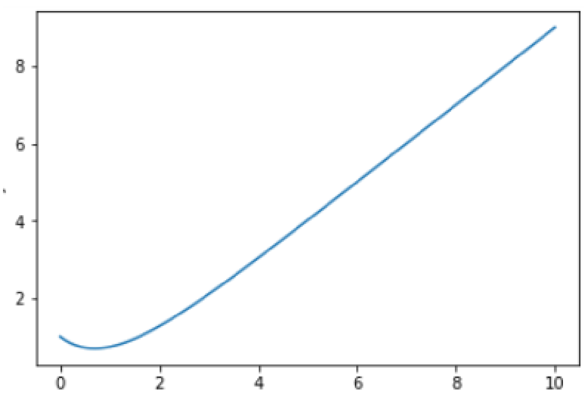
Out[22]: $y(x) = C_1 \sin(2x) + C_2 \cos(2x) - \frac{1}{5}\cos(3x)$

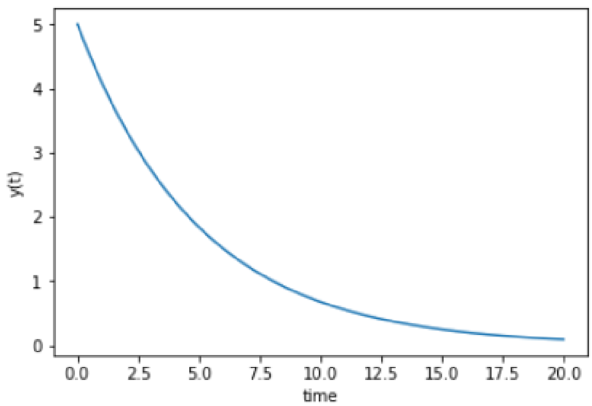
```
The differential equation is:
<function diff at 0x000002719CE4E268>
The solution is:
```

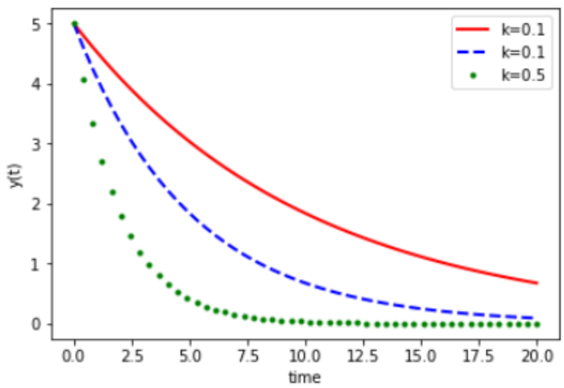
Out[23]: $y(x) = (C_1 + C_2 x + e^{-x})\, e^{-2x}$

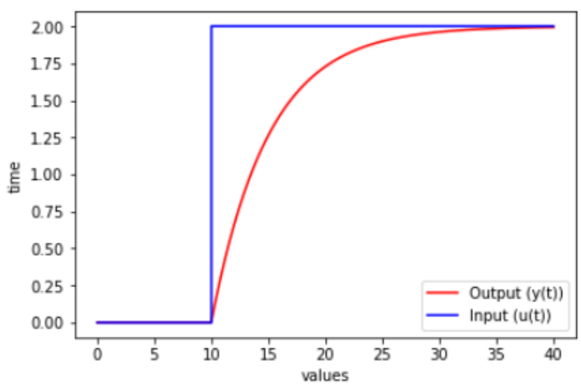
```
The differential equation is:
<function diff at 0x000002719CE4E268>
The solution is:
```

Out[26]: $y(x) = \left(C_1 \sin(x) + C_2 \cos(x) - \frac{1}{3}\cos(2x)\right) e^x$









Conclusion:

With the above codes & outputs, we have solved and found linear systems in Python. A few graphs have also been plotted for the different linear differential equations.

Lab 5

Topic: Plotting of scalar & vector fields

Date: 24th November 2018

Aim: To plot scalar & vector fields using Python and to find the cross product of vectors

Source Code:

```
In [19]: from mpl_toolkits.mplot3d import Axes3D
         import matplotlib.pyplot as plt
         import numpy as np
         from pylab import *
         from matplotlib import cm
         %matplotlib inline
         plt.style.use('seaborn-white')
```

```
In [20]: ax=Axes3D(figure())
         x=arange(-3*pi,3*pi,0.1)
         y=arange(-3*pi,3*pi,0.1)
         xx,yy=meshgrid (x,y)
         z=sin(xx)+sin(yy)
         ax.plot_surface(xx,yy,z,cmap=cm.jet,cstride=1)
         plt.show()
```

```
In [21]: #Vector Field Plots
         fig=plt.figure()
         ax=fig.gca(projection='3d')
         x,y,z=np.meshgrid(np.arange(-0.8,1,0.2),np.arange(-0.8,1,0.2),np.arange(-0.8,1,0.8))
         u=np.sin(np.pi*x)*np.cos(np.pi*y)*np.cos(np.pi*z)
         v=-np.cos(np.pi*x)*np.sin(np.pi*y)*np.cos(np.pi*z)
         w=(np.sqrt(2.0/3.0)*np.cos(np.pi*x)*np.cos(np.pi*y)*np.sin(np.pi*z))
         ax.quiver(x,y,z,u,v,w,length=0.1)
         plt.show()
```

```
In [22]: # Line Plot
         ax = Axes3D(figure())
         phi = linspace(0, 2*pi, 400)
         x = cos(phi)
         y = sin(phi)
         z = 0
         ax.plot(x, y, z, label = 'x') #circle
         z = sin(4*phi) #Modulated in Z plane
         ax.plot(x, y, z, label = 'x')
         ax.set_xlabel('X')
         ax.set_ylabel('Y')
         ax.set_zlabel('Z')
         show()
```

In [23]:
```
#Scalar Field
x = np.linspace(-3, 3, 256)
y = np.linspace(-3, 3, 256)
X, Y = np.meshgrid(x, y)
Z = np.sinc(np.sqrt(X ** 4 + Y ** 4))
fig = plt.figure()
ax = fig.gca(projection = '3d')
ax.plot_surface(X, Y, Z, cmap=cm.gray)
plt.show()
```

In [24]:
```
x = np.linspace(0,5,50)
y = np.linspace(0,5,40)
def f(x,y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
X, Y = np.meshgrid(x,y)
Z = f(X, Y)
plt.contour(X, Y, Z, colors='pink')
```

In [25]:
```
def func(x,y):
    return 3*(x-2)**2 + (y+1)**2
#setup grid
nx=200 #number of points in x direction
ny=150 #number of points in y direction
x= np.linspace(-5,5,nx) #nx points equally spaced b/w -5,5
y= np.linspace(-6,6,ny)
X,Y= np.meshgrid(x,y, indexing='ij') #2D array (matrix) of points across x and y
Z= np.zeros((nx,ny)) #initialize output of size (nx,ny)
#--evaluate across grid
for i in range(nx):
    for j in range(ny):
        Z[i,j]=func(X[i,j], Y[i,j])
#--contor plot
plt.figure() #dtart a new figure
plt.contour(X,Y,Z,50,cmap='bone') #using 50 contour lines
plt.colorbar() #add a colorbar
plt.xlabel('x') #labels for axes
plt.ylabel('y')
plt.show() #show plot
```

In [26]:
```
x= np.linspace(0,5,50)
y= np.linspace(0,5,40)
def f(x,y):
    return np.sin(x)**10+ np.cos(10+y*x)* np.cos(x)
X,Y= np.meshgrid(x,y)
Z=f(X,Y)
plt.contour(X,Y,Z, colors='pink')
plt.contour(X,Y,Z,20, cmap='ocean')
```

In [27]:
```
a = np.linspace(-3, 3, 100)
b = np.linspace(-3, 3, 100)
X , Y = np.meshgrid(a, b)
Z = (((X+Y*1j)**2-1)*(((X+Y*1j)-2-1j)**2))/((X+Y*1j)**2+2+2j)
Z
```

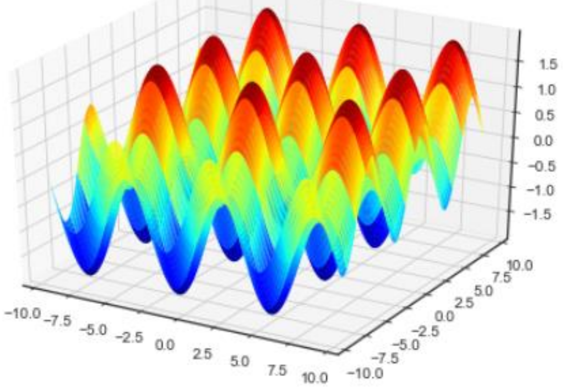
In [31]:
```
a=np.linspace(5,30,1000)
b=np.linspace(5,30,1000)
X,Y=np.meshgrid(a,b)
Z=(X+Y*1j)*(X-Y*1j)
import matplotlib.pyplot as plt
plt.contour(X,Y,Z)
```

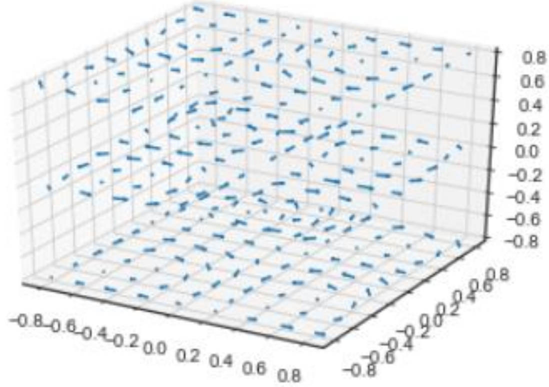
```
C:\Users\Jeevan\Anaconda3\lib\site-packages\numpy\ma\core.py:2766: ComplexWarning: Casting complex values to real discards the imaginary part
  order=order, subok=True, ndmin=ndmin)
```

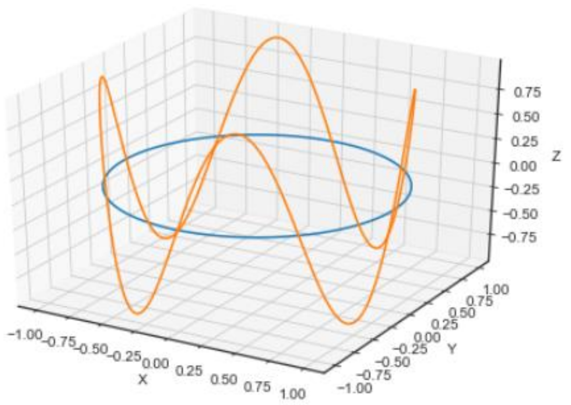
In [32]:
```
a=np.linspace(-5,5,150)
b=np.linspace(-5,5,150)
X,Y=np.meshgrid(a,b)
Z=(X+Y*1j)*(X-Y*1j)
plt.contour(X,Y,Z)
```

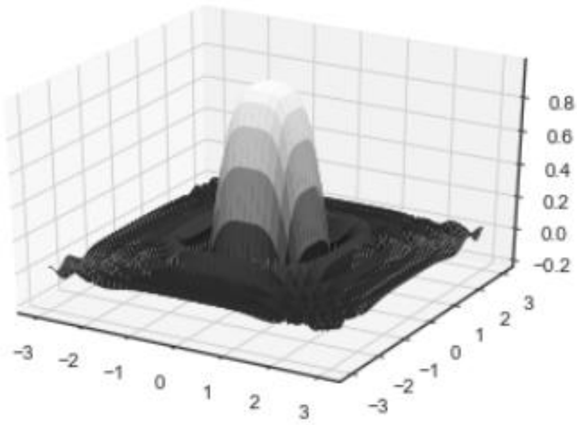
```
C:\Users\Jeevan\Anaconda3\lib\site-packages\numpy\ma\core.py:2766: ComplexWarning: Casting complex values to real discards the imaginary part
  order=order, subok=True, ndmin=ndmin)
```

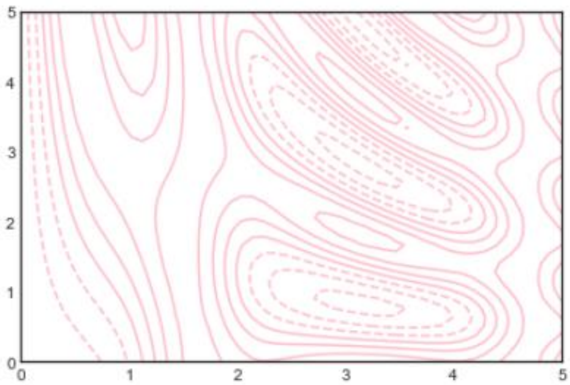
Output (Graphs/Tables):

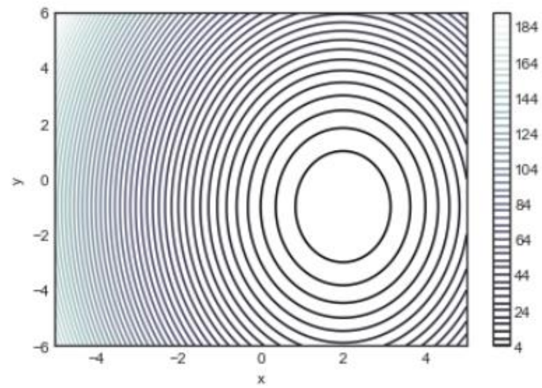


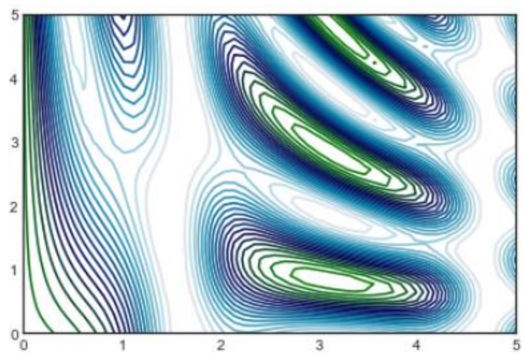






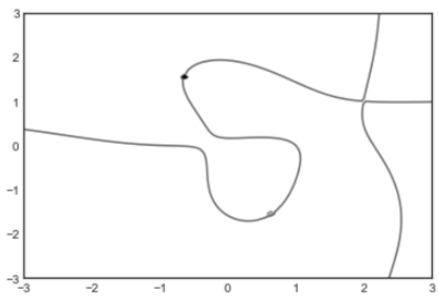


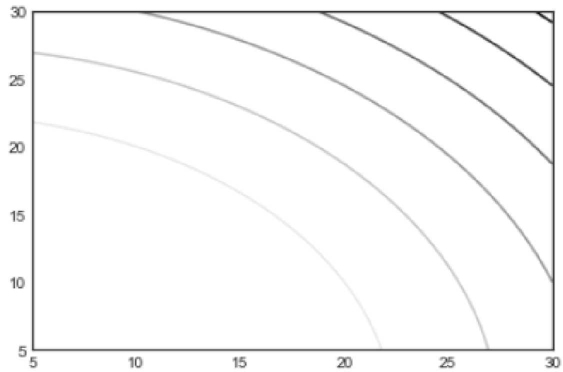




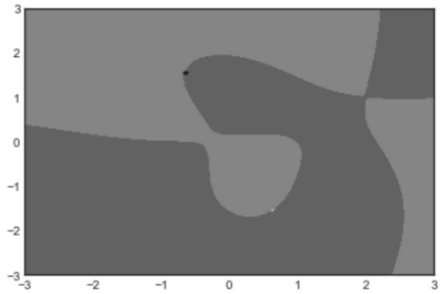
In [29]: `plt.contour(X,Y,Z.imag)`

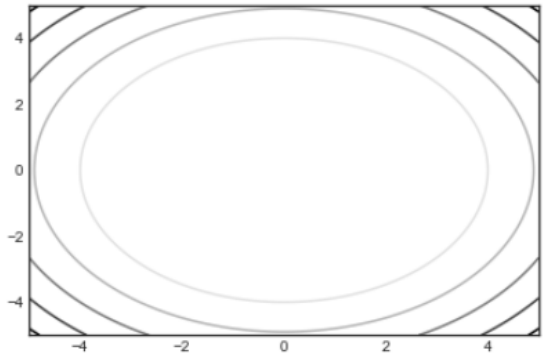
Out[29]: <matplotlib.contour.QuadContourSet at 0x229e4220898>



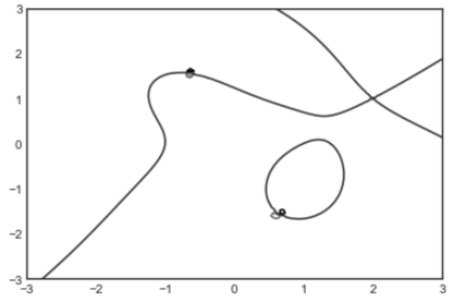



```
In [30]: plt.contourf(X,Y,Z.imag)
Out[30]: <matplotlib.contour.QuadContourSet at 0x229e4bfbfd0>
```






```
In [28]: plt.contour(X,Y,Z.real)
Out[28]: <matplotlib.contour.QuadContourSet at 0x229e41aeeb8>
```



Conclusion:

From the above graphs, we can understand how scalar and vector fields are plotted with the codes entered.

Lab 6

Topic: Mathematical model: Interest rates

Date: 10th January 2019

Aim:

Source Code:

```
In [2]: # Compute simple interest for the user inputs p,n,r
        #!/usr/bin/python

        def calculate_simple_interest(p,n,r):
            si=0.0
            si=float(p*n*r)/float(100);
            return si;

        def calc_amount(p,si):
            amt=p+si
            return amt

        if __name__=='__main__':
            p=float(input("Enter value\np:"));
            n=int(input("n: "));
            r=float(input("r:"));
            simple_interest=calculate_simple_interest(p,n,r);
            print("Simple interest value: %.2f" % simple_interest);
            amt=calc_amount(p,simple_interest)
            print("Amount: %.2f" % amt);
```

Compund interest is an interest calculated on the inital principal and also calculated on the interest of previous periods of deposit. Formulae is A=P*(1+R/100)^t

CI=A-P P-Principal A-Tota Amount CI-Compound Interest

```
In [1]: principle=float(input("Enter principle amount: "))
        time=int(input("Enter time duration: "))
        rate=float(input("Enter rate of interest: "))
        amount=(principle * (1 + (float(rate)/100))**time)
        compound_interest=amount-principle;
        print("Total amount:- ",amount)
        print("Compound interest:- %0.2f"%compound_interest)
```

Output (Graphs/Tables):

```
Enter value
p:100
n: 10
r:5.0
Simple interest value: 50.00
Amount: 150.00
Enter principle amount: 100
Enter time duration: 10
Enter rate of interest: 12
Total amount:-  310.5848208344212
Compound interest:- 210.58
```

Conclusion:

This is how interest rates are calculated in Python.

Lab 7

Topic: Mathematical model: Growth of population/Exponential model

Date: 12th January 2019

Aim: To find the bacterial growth of a population

Source Code:

BACTERIAL GROWTH

Example 01: A culture has P_0 number of bacteria. At $t = 1$ h the number of bacteria is measured to be $\frac{3}{2}P_0$. If the rate of growth is proportional to the number of bacteria Pt present at the time t, the determine the time necessary for the number of bacterial to triple.

In [21]:
```
t,k = Symbol('t'),Symbol('k')
P = Function('P')(t)
diffeq = Eq(P.diff(t)-k*P,0)
sol = dsolve(diffeq)
P0 = Symbol('P0')
constants = solve([sol.subs([(P,P0),(t,t)]),sol.subs([(P,3/2*P0),(t,t+1)])])
sol = sol.subs(k,constants[1][k])
sol
```

Out[21]: $P(t) = C_1 e^{0.405465108108164t}$

In [22]:
```
c1 = solve([sol.subs([(P,P0),(t,0)]),sol.subs([(P,3/2*P0),(t,1)])])
c1
```

Out[22]: $\{C_1 : P_0\}$

In [24]:
```
sol = sol.subs(c1)
sol
```

Out[24]: $P(t) = P_0 e^{0.405465108108164t}$

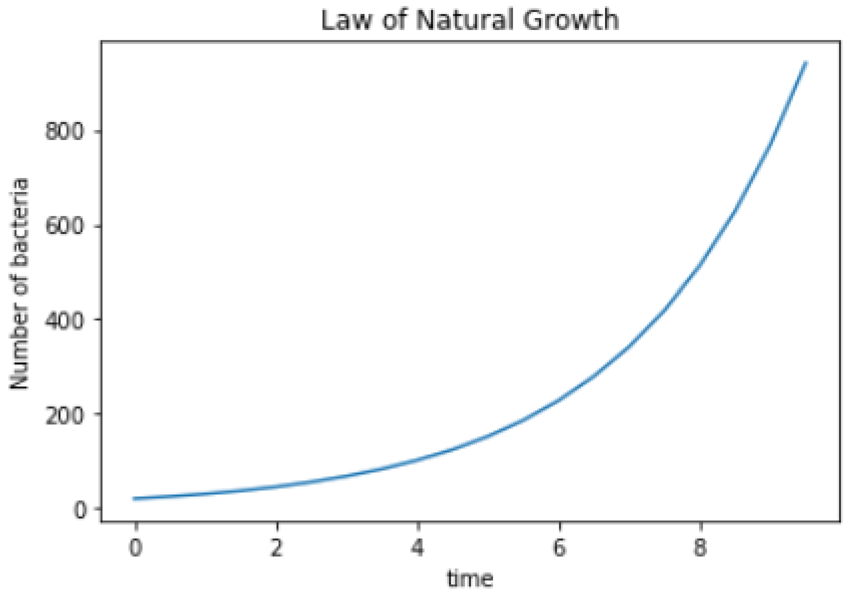
In [25]:
```
solve([sol.subs(P,3*P0)],t)
```

Out[25]: $\{t : 2.70951129135146\}$

Example 02: Plot the function $y = P_0 e^{0.405465108164}, y_0 = 20$

In [27]:
```
t = arange(0,10,0.5)
P0 = 20
y = 20*exp(0.405465108108164*t)
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('Number of bacteria')
plt.title('Law of Natural Growth')
plt.show()
```

Output (Graphs/Tables):



Conclusion:

The law of natural growth has been plotted with help of a few python codes. There is also an example or two have been used to find the time for bacteria to grow.

Lab 8

Topic: Mathematical model: Logistic Growth

Date: 24th January 2019

Aim: To build a mathematical model finding the logistic growth in a population

Source Code:

Equation for Logistic Population Growth

$$\frac{dN}{dt} = rN\left(\frac{K-N}{K}\right)$$

We can also look at logistic growth as a mathematical equation. Population growth rate is measured in number of individuals in a population (N) over time (t). The term for population growth rate is written as (dN/dt). The d just means change. K represents the carrying capacity per individual for a population. The logistic growth equation growth equation assumes that K and r do not chnage over time in a population.

Logistic Growth

In [26]:
```
import numpy as np
import math
from numpy import *
from sympy import *
from pylab import *
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from sympy.interactive import printing
printing.init_printing(use_latex=True)
```

In [3]:
```
def f(N,t,r,K):
    return r * N * (1-N/K)

r = 1
K = 50
N0 = 10
t = np.linspace(0,10,100)

N = odeint(f,N0,t,args = (r,K))
plt.plot(t,N)
plt.ylim([0,100])
plt.axhline(y=K,color='k',linewidth=0.5)
plt.show()
```

In [5]:
```
N0 = 80
t = np.linspace(0,10,100)

N = odeint(f, N0, t, args = (r,K))
plt.plot(t,N)
plt.ylim([0,100])
plt.axhline(y=K,color='k',linewidth=0.5)
plt.show()
```

In [6]:
```
N0 = 30
t = np.linspace(0,10,100)

N = odeint(f, N0, t, args = (r,K))
plt.plot(t,N)
plt.ylim([0,100])
plt.axhline(y=K,color='k',linewidth=0.5)
plt.show()
```

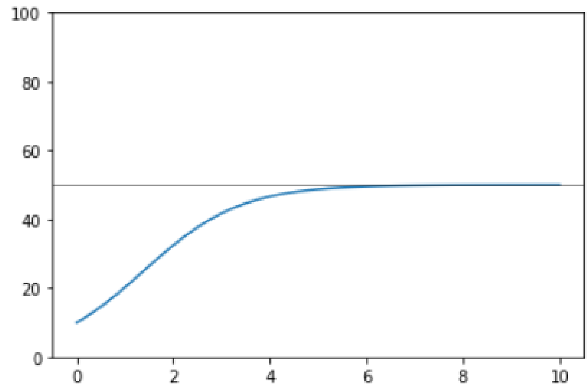
Example 02

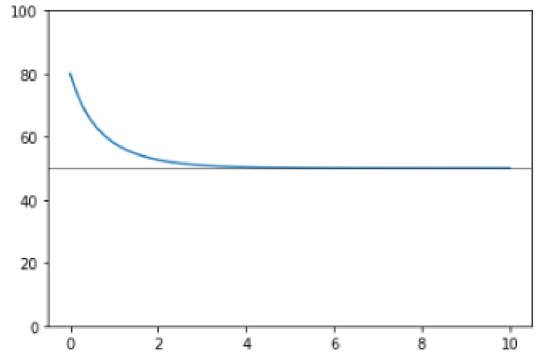
Let's start with the logistic equation, now with any parameters for growth rate and carrying capacity:

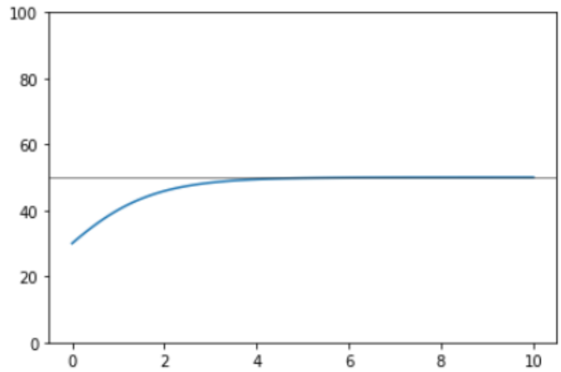
$\frac{dx}{dt} = rx(1 - \frac{x}{K})$ **with** $r = 2, K = 10$ **and** $x0 = 0.1$ ¶

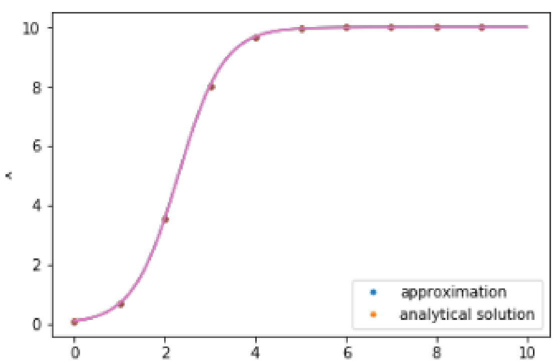
```
In [13]: t = arange(0,10,1)
# parameters
r = 2
K = 10
# initial condition
x0 = 0.1

# Let's define the right - hand side of the differential equation
# It must be a function of the dependendent variable (x) and of the time (t), even if time does not appear explicit;y.
# this is how you define a function:
def f(x,t,r,K):
    # in python, there are no curling braces '{}' to start or end a function, nor any special keyword:
    # the block is defined by leading spaces (usually 4)
    # arithmetic is done the same as in other languages: +, -, *, /
    return r*x*(1-x/K)

# call the function that performs intergration
# the order of the arguments as below: the derivative function, the initial condition, the points where you want the solution
# and the list of the parameters
x = odeint(f,x0,t,(r,K))

#plot the solution
plt.plot(t,x,'.')
plt.xlabel('t')
plt.ylabel('x')
t = arange(0,10,0.01)
# plot analytical solution
# notice that 't' is an array when you do an arithmetic operation
# with an array, it is the same as doing it for each element.
plt.plot(t,K*x0*exp(r*t)/(K+x0*(exp(r*t)-1)))
plt.legend(['approximation','analytical solution'],loc = 'best') # draw legend
plt.show()
```

<u>Output (Graphs/Tables):</u>









Conclusion:

From the above graph, the logistic growth of a population has been calculated and a few graphs have also been plotted in Python with the help of some code to demonstrate the growth.
