Problem 3.e) The following code is used for calculating $S(t) = exp(At)$ using the Python package **linalg**.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 18 11:26:04 2018

@author: alfred_mac
"""

import numpy as np
import scipy.linalg as sla
import matplotlib.pyplot as plt

N = 100

A = np.array([[0,1],[-1,0]])

t = np.linspace(0,7,N)

S11 = np.zeros(N)
S12 = np.zeros(N)
S21 = np.zeros(N)
S22 = np.zeros(N)

E = sla.eig(A)           # Eigen values and eigen vectors of A
R = E[1]                 # R = eigen vectors of A
L = sla.inv(R)           # L = inv(R)
e = E[0]                 # Eigen values of A


for i in range(N):
    exp_e = np.diag(np.exp(t[i]*e))      # Exponent of tA
    S = np.matmul(R,np.matmul(exp_e,L)) # Calculating S(t)

    # Taking only the real parts as the imaginary parts are insignificant
    S11[i] = np.real(S[0][0])
    S12[i] = np.real(S[0][1])
    S21[i] = np.real(S[1][0])
    S22[i] = np.real(S[1][1])


plt.plot(t,S11, 'r o')
plt.plot(t,S12)
```

```
plt.plot(t,S21)
plt.plot(t,S22, 'g--')
plt.legend(['$S_{11}(t)$','$S_{12}(t)$','$S_{21}(t)$','$S_{22}(t)$'])
plt.xlabel('t')
plt.ylabel('$S_{ij}(t)$')
plt.show()
```

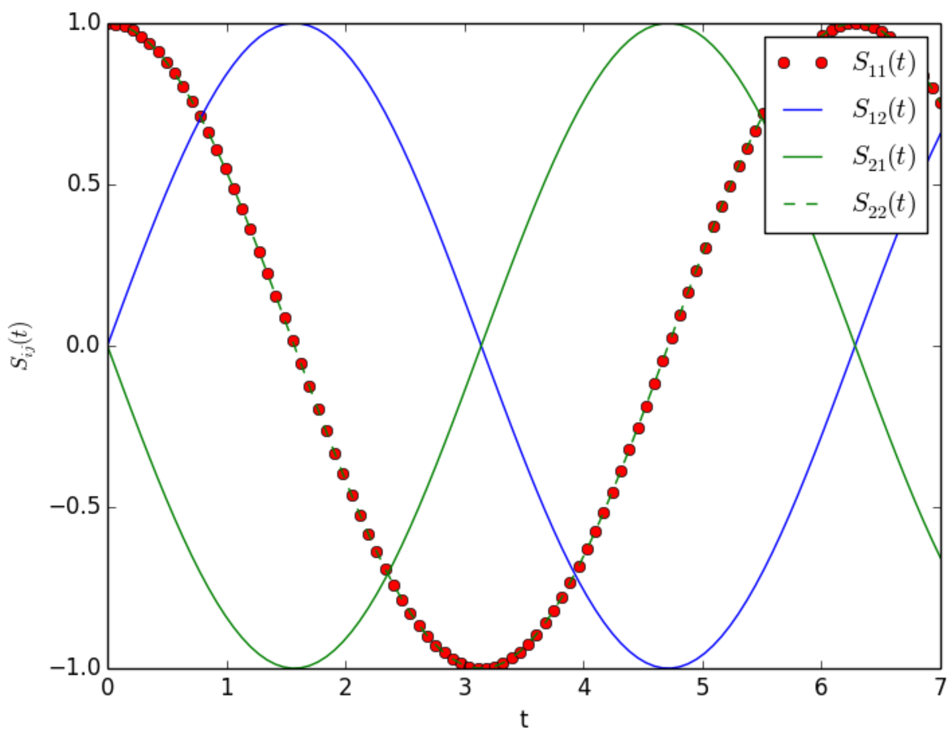

Figure 1: $S_{ij}(t)$ vs t using Python package **linalg**

Problem 4) The following code is used for calculating $S(t) = exp(At)$ by computing the Taylor series using Horner's rule.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 18 19:50:33 2018

@author: alfred_mac
"""
```

```
import numpy as np
import matplotlib.pyplot as plt

# Function definition applying Horner's rule to calculate the series
def expm_alf(t,M,k):
    S = np.eye(len(M)) + (t/k)*M
    for i in range(k-1):
        S = np.eye(len(M)) + (t/(k-1-i))*np.matmul(M,S)
    return S

# Function to convert a positive integer to a string of digits in base 2 (Source
def N_to_base2(n):
    if n == 0:
        return [0]
    digits = []
    while n:
        digits.append(int(n%2))
        n //= 2
    return digits[::-1]

# Function to calculate 2's powers of any matrix to be used for any power of a m
def Bs_for_N_to_base2(B,n):
    if n == 0:
        return np.eye(len(B))
    Blist = []
    while n:
        Blist.append(B)
        B = np.matmul(B,B)
        n //= 2
    return Blist[::-1]

# Function calculating any power of a matrix
def B_pow_n(B, digits):
    S = np.eye(2)
    for i in range(len(digits)):
        if digits[i]==1:
            S = np.matmul(S,B[i])
    return S


T = 7                           # Total simulation time
deltat = 0.1                    # delta t used for time step
N = int(T/deltat)               # Total no. of time steps

A = np.array([[0,1],[-1,0]])    # given A
```

```
S11 = np.zeros(N)
S12 = np.zeros(N)
S21 = np.zeros(N)
S22 = np.zeros(N)

k=1
S1 = np.eye(2)
S2 = expm_alf(deltat,A,k)

# Loop to find which k to use to achieve the required level of accuracy
while(np.max(abs(S1-S2))>1e-10):
    k = k+1
    S1 = S2
    S2 = expm_alf(deltat,A,k)


for i in range(N):
    i_base2 = N_to_base2(i)
    Sti = B_pow_n(Bs_for_N_to_base2(S2,i), N_to_base2(i))

    # Taking only the real parts as the imaginary parts are insignificant
    S11[i] = np.real(Sti[0][0])
    S12[i] = np.real(Sti[0][1])
    S21[i] = np.real(Sti[1][0])
    S22[i] = np.real(Sti[1][1])

t = np.arange(0,T,deltat)

plt.plot(t,S11, 'r o')
plt.plot(t,S12)
plt.plot(t,S21)
plt.plot(t,S22, 'g--')
plt.legend(['$S_{11}(t)$','$S_{12}(t)$','$S_{21}(t)$','$S_{22}(t)$'])
plt.xlabel('t')
plt.ylabel('$S_{ij}(t)$')
plt.show()
```

The following code is used for computing B^n just using $O(log(n))$ matrix multiplications.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 18 19:50:33 2018
```

```
@author: alfred_mac
"""

import numpy as np

# Function definition applying Horner's rule to calculate the series
def expm_alf(t,M,k):
    S = np.eye(len(M)) + (t/k)*M
    for i in range(k-1):
        S = np.eye(len(M)) + (t/(k-1-i))*np.matmul(M,S)
    return S

deltat = 0.2                        # delta t used for time step

A = np.array([[0,1],[-1,0]])      # given A

k=1
S1 = np.eye(2)
S2 = expm_alf(deltat,A,k)

# Loop to find which k to use to achieve the required level of accuracy
while(np.max(abs(S1-S2))>1e-10):
    print("When k is",k,'error is ',np.max(abs(S1-S2)))
    k = k+1
    S1 = S2
    S2 = expm_alf(deltat,A,k)

print("Finally when k is",k,'error is ',np.max(abs(S1-S2)))
print("S is ",S2)
```

Problem 5) The following is the code to solve the system of differential equations using the eigen value method

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 25 15:20:58 2018

@author: alfred_mac
"""

import numpy as np
import scipy.linalg as sla
import matplotlib.pyplot as plt
```

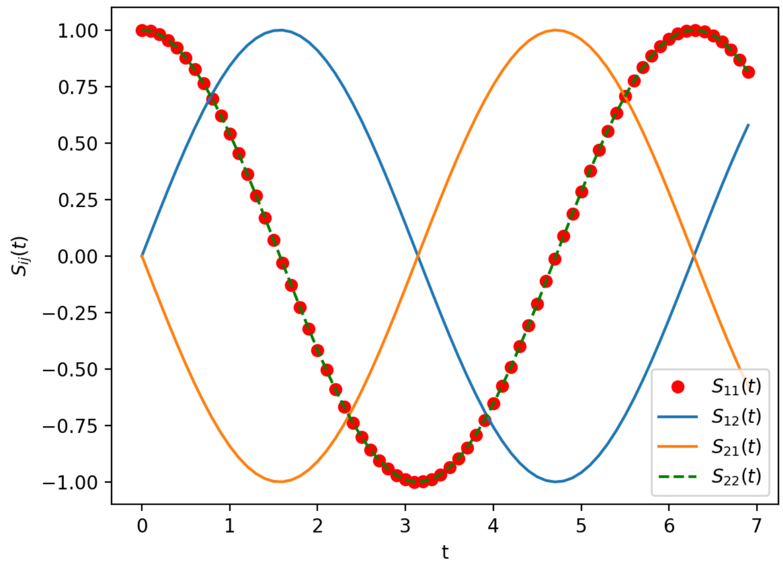

Figure 2: $S_{ij}(t)$ vs t by computing Taylor's series using Horner's rule

```
N = 100                    # Time length of the simulation
n = 50                     # Total no. of spots (or the size of X vector)

A = np.zeros([n,n])        # Matrix defining the dynamics
p = 0.7                    # Rate at which particles move left
q = 0.3                    # Rate at which particles move right


# Defining the matrix A
A[0,0] = -p
A[0,1] = p
A[n-1,n-2] = q
A[n-1,n-1] = -q

for i in range(n-2):
    A[i+1,i] = q
    A[i+1,i+1] = -(p+q)
    A[i+1,i+2] = p


t = np.linspace(0,7,N)  # Time counter
```

```
X = np.zeros([n,N])      # No. of particles at different spots at different times
X[0,0] = 1               # Initial condition for no. of particles, X

E = sla.eig(A)           # Eigen values and eigen vectors of A
R = E[1]                 # R = eigen vectors of A
L = sla.inv(R)           # L = inv(R)
e = np.real(E[0])        # Eigen values of A

AA  = np.eye(n)
for i in range(N-1):
    exp_e = np.diag(np.exp(t[i]*e))      # Exponent of tA
#    AA = np.eye(n)+np.matmul(AA,A)
    S = np.matmul(R,np.matmul(exp_e,L)) # Calculating S(t)
    X[:,i+1] = np.matmul(S,X[:,0])
#    X[:,i+1] = np.matmul(AA,X[:,0])


plt.plot(t,X[0,:], 'r o')
plt.legend(['$x_{1}(t)$'])
plt.xlabel('t')
plt.ylabel('$x_{1}(t)$')
plt.show()
```

Problem 6) The following code is to compute the matrix product AB when A and B are $n \times n$ matrices

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 25 15:42:44 2018

@author: alfred_mac
"""
import numpy as np
import random as rnd
import numpy.random as nprnd
import time as t
import matplotlib.pyplot as plt


def matprod_scalar_loop(P,Q):

    if len(P.T)==len(Q):
        R = np.zeros([len(P),len(Q.T)])
        for i in range(len(P)):
            for j in range(len(Q.T)):
```

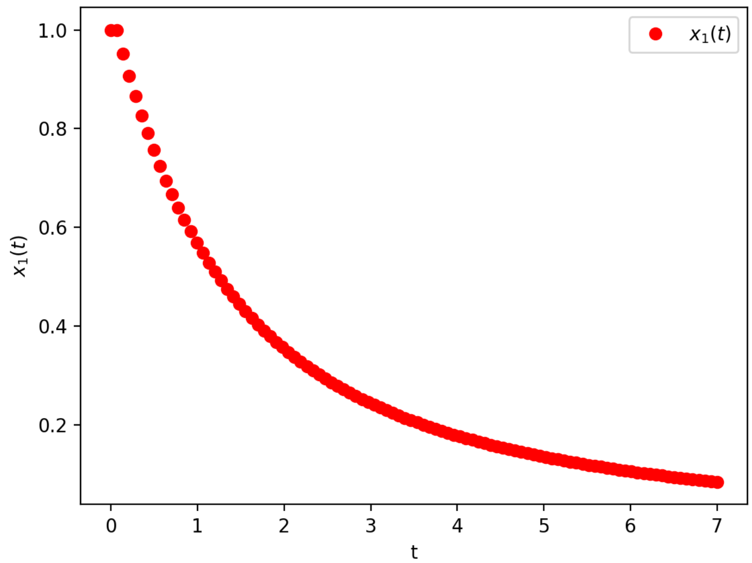

Figure 3: No. of particles at spot 1 over time, $x_1(t)$ vs t

```
                for k in range(len(Q)):
                    R[i][j] += P[i][k]*Q[k][j]

    elif len(Q.T)==len(P):
        R = np.zeros([len(Q),len(P.T)])
        for i in range(len(Q)):
            for j in range(len(P.T)):
                for k in range(len(P)):
                    R[i][j] += Q[i][k]*P[k][j]

    else:
        print("Can't multiply theses matrices :(")
        return

    return R


def matprod_vector_loop(P,Q):

    if len(P.T)==len(Q):
        R = np.zeros([len(P),len(Q.T)])
        for i in range(len(P)):
```

```
            for j in range(len(Q.T)):
                R[i][j] = np.sum(P[i,:]*Q[:,j])

        elif len(Q.T)==len(P):
            R = np.zeros([len(Q),len(P.T)])
            for i in range(len(Q)):
                for j in range(len(P.T)):
                    R[i][j] = np.sum(Q[i,:]*P[:,j])

        else:
            print("Can't multiply theses matrices :(")
            return

        return R


def matprod_numpy(P,Q):
    return np.matmul(P,Q)


nstart = 10
nend = 300
nstep = 10

N = np.arange(nstart,nend+1,nstep)
T1 = np.zeros(len(N))
T2 = np.zeros(len(N))
T3 = np.zeros(len(N))

for n in range(len(N)):
    A = nprnd.rand(N[n],N[n])
    B = nprnd.rand(N[n],N[n])
    t0 = t.time()
    C = matprod_scalar_loop(A,B)
    t1 = t.time()
    D = matprod_vector_loop(A,B)
    t2 = t.time()
    E = matprod_numpy(A,B)
    t3 = t.time()
    T1[n] = t1-t0
    T2[n] = t2-t1
    T3[n] = t3-t2


plt.plot(N,T1)
plt.plot(N,T2)
```

```
plt.plot(N,T3)
plt.legend(['T1','T2','T3'])
plt.xlabel('N')
plt.ylabel('Time taken')
plt.show()
```

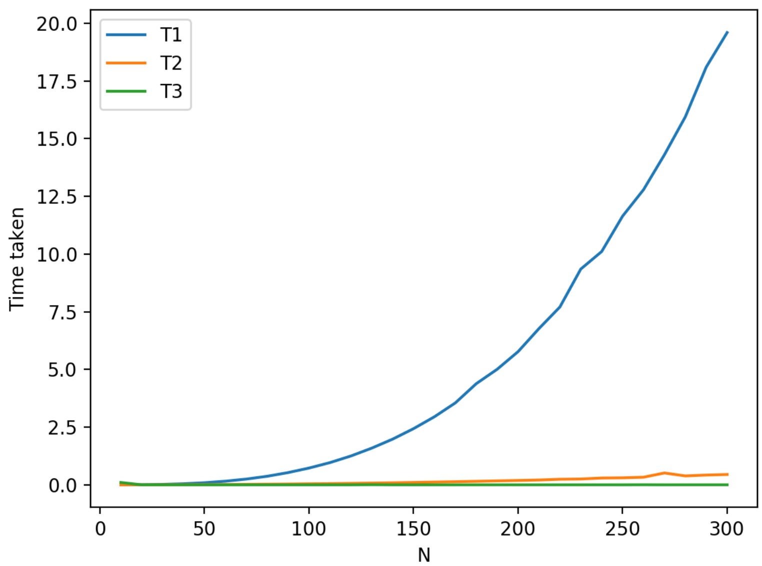

Figure 4: Time taken for computing matrix products as a function of the size of the matrix, n