Week 3 LAB

Question 1:

```
def __iter__(self):
    return iter(self.vector)
# in order to get item
def __getitem__(self, item):
    return self.vector[item]
# in order to get the length
def __len__(self):
    return(len(self.vector))
# overloading the add function
def __add__(self, other):
    for i in range(len(self.vector)):
        self.vector[i]=self.vector[i]+other.vector[i]
    return self
```

- We have defined a class **RowVectorFloat ()** and we have defined suitable methods
- getitem method gets the correct item from the RowVectorFloat object
- magic method len is defined to get the length of the RowVectorFloat object.
- Add is suitably defined to induce addition between two such objects

```
# Overloading the multiplication function
def __rmul__(self,other):
    vector=self.vector
    for i in range(len(vector)):
        self.vector[i]=other*self.vector[i]
    return self
# Seeting value for a corresponding a key
def __setitem__(self, key, value):
    self.vector[key]=value
    return self
# in order to round of the digits to 2 significant digits
def __round__(self, n=None):
    for i in range(len(self.vector)):
        self.vector[i]=round(self.vector[i],n)
    return self
```

- We have defined **rmul** to define scalar multiplication between the objects
- And also we have set up set item so that we can change any entries of the class
- We also define a round method which rounds a number upto 2 significant digits

Question 2

```
def sampleSymmetric(self):
    # Setting the non diagonal values uniformly from (0,n)
    for i in self.matrix.keys():
        s = list(self.matrix[i])
        r = random.random()
        while r == 0:
            r = random.random()
        s[i] = round(4 * r, 2)
        self.matrix[i] = RowVectorFloat(s)
    # making sure the matrix is symmetric and the
# diagonal entries are uniformly sampled from (0,1)
for i in self.matrix.keys():
    for j in range(self.rows):
        if [i, j] != [j, i]:
            r = round(random.random(), 2)
            while r == 0:
                r = random.random()
            self.matrix[i][j] = r
            self.matrix[j][i] = r

return self
```

❖ **Samplesymmetric** is defined for the **SqaureMatrixFloat** object which samples a symmetric matrix where diagonal elements are sampled from (0,n) and all other entries are sampled from (0,1)

❖
```
def isDRDominant(self):
    k = 0
    for i in self.matrix.keys():
        s = list(self.matrix[i])
        if s[i] < sum(s) - s[i]:
            return False
        else:
            k = k + 1
        s.clear()

    if k == self.rows:
        return True
```

❖ isDRDdominant is defined to check whether the matrix is diagonally row dominant or not

❖
```
for i in self.matrix.keys():
    D[i] = self.matrix[i][i]
zero_list = [0 for i in range(self.rows)]
# intiializing the lower and upper triangular matrix
l = {i: list(self.matrix[i]) for i in self.matrix.keys()}
u = {j: list(self.matrix[j]) for j in self.matrix.keys()}
t = {i: zero_list for i in range(self.rows)}
# Populating the lower and upper triangular matrix
for i in self.matrix.keys():
    for j in range(self.rows):
        if i >= j:
            u[i][j] = 0
for i in self.matrix.keys():
    for j in range(self.rows):
        if i <= j:
            l[i][j] = 0
# Forming the matrix T=-D^(-1)(L+U)
```

❖ **jSolve** method is created to produced desired iterations of the solutions of the linear system

❖ _we use the fact that co-efficient matrix is reduced to A=D+L+U where D is a diagonal matrix whereas L and U are lower triangular and upper triangular respectively.

❖ We compute the quantities T=$-D^{-1}(L+U)$ and also C=$D^{-1}b$ matrix which help us in finding the iterative solution where the solution is $x^{(k+1)} = Tx^{(k)} + C$

❖
```
while (k != self.iter):
    for i in range(self.rows):
        for j in range(self.rows):
            # multiplying T and X_sol
```

```
                s = s + t[i][j] * x_sol[j]
            b.append(s)
            s = 0
        # iterating solutions
        x_sol = C + RowVectorFloat(b)
        # calulating || Ax^(k)-b|| for each iteration
        for i in range(self.rows):
            for j in range(self.rows):
                add = add + self.matrix[i][j] * x_sol[j]
            error.append((add - self.b[i]) ** 2)
            add = 0
        # getting toatl error
        total_err.append(sum(error) ** (1 / 2))
        k = k + 1
    return total_err, list(x_sol)
```

❖ We also calculate the error quantity as each iteration shown in the above code
❖ The method returns two quantities one is the error term at each iteration and the other is the solution after n iterations.

Question 3:

```
#Evaluating the polynomial at a point and using getitem to hget the
value
def __getitem__(self, item):
    return sum([(item ** i) * self.list_co[i] for i in
range(len(self.list_co))])
# Defining addition of Polynom,ials properly
def __add__(self, other):
        size=max(len(self.list_co),len(other.list_co))
        sum=[0 for i in range(size)]
        for i in range(0,len(self.list_co),1):
             sum[i]=self.list_co[i]
        for i in range(len(other.list_co)):
            sum[i]+=other.list_co[i]
        self.list_co=sum
        return self
# defining Subtraction properly
def __sub__(self, other):
        size=max(len(self.list_co),len(other.list_co))
        sum=[0 for i in range(size)]
        for i in range(0,len(self.list_co),1):
             sum[i]=self.list_co[i]
        for i in range(len(other.list_co)):
            sum[i]-=other.list_co[i]
        self.list_co=sum
        return self
# Defining constant multiplication
```

❖ We have defined a class called **Polynomial** which uses the attribute getitem to evaluate it any point.
❖ Add and sub are suitably defined to counter if no entries in the objects are same or not

```
def __rmul__(self, other):
    vector = self.list_co
    for i in range(len(vector)):
        self.list_co[i] = other * self.list_co[i]
    return self
# defining multiplication of two ploynomials
def __mul__(self, other):
    r1=len(self.list_co)
    r2=len(other.list_co)
    prod=[0]*(r1+r2-1)
    for i in range(r1):
        for j in range(r2):
            prod[i+j]+=self.list_co[i]*other.list_co[j]
    self.list_co=prod
    return self
```

- ❖ We have defined rmul and mul to define scalar multiplication of the polynomial and also usual multiplication between two polynomials

```
def show(self,a,b):
    self.a=a
    self.b=b
    r=(self.b-self.a)/5
    x=np.arange(self.a,self.b+r,r)
    # setting up a polynomial object to valuate a value at points and
    # using it in a plot
    p=Polynomial(self.list_co)
    y=[p[i]for i in x]
    fig,ax=plt.subplots()
    ax.plot(x,y)
    s=[p[self.a],p[self.b]]
    s.sort()
    ax.set_ylim(s)
    ax.set_xlabel("x")
    ax.set_ylabel("p(x)")

    ax.set_title("Plot of the Polynomial")
    ax.set_xlim(self.a,self.b)
    plt.grid(True,which="both")
    plt.show()
```

- ❖ We have defined show to plot the polynomial between any interval .

```
self.plot_val=plot_val
x_val=[]
y_val=[]
s=[]
# setting X values and y values separately
for i in range (len(self.plot_val)):
    x_val.append(self.plot_val[i][0])
    y_val.append(self.plot_val[i][1])
# evaluating the co-efecient matrix
for i in range(len(self.plot_val)):
     s.append([x_val[i]**j for j in range(len(self.plot_val))])
A=np.array(s)
b=np.array(y_val)
# using linalg module to solve the linear equation
x=np.linalg.solve(A,b)
```

```
p=Polynomial(x)
fig,ax=plt.subplots()
r=(max(x_val)-min(x_val))/5
x=np.arange(min(x_val),max(x_val)+0.5,0.5)
ax.plot(x_val,y_val,"ro")
```

- ❖ In the above code we have used pythons' numpy module to solve a linear equation and then plot the polynomial along with the points provided
- ❖ We separate the x values and y values and then evaluated the polynomials at x values and use them to plot the polynomial.

```
def fitViaLagrangePoly(self,plo_val):
    self.plot_val=plo_val
    x_val = []
    y_val = []
    #A plynomial to store the numerartor of the lagrange method
    s = {i : Polynomial([1]) for i in range(len(self.plot_val)) }
    a=Polynomial([1])
    # list to store the denominator of the lagrange method
    denom=[]
    b=1
    for i in range(len(self.plot_val)):
        x_val.append(self.plot_val[i][0])
        y_val.append(self.plot_val[i][1])
    # creating the denominator and numerator for each x value
    for i in  range(len(self.plot_val)):
        for j in range(len(x_val)):
            if j!=i:
                p=Polynomial([-x_val[j],1])
                # getting the polynomial multiplication in numerator
                b=b*(x_val[i]-x_val[j])
                a=a*p
        s[i]=s[i]*a # storing the polynomials in a dictionary
        denom.append(b)
        a=Polynomial([1])
        b=1

    l=Polynomial([0]) # to hold the numerator of the lagrange method
    for i in range(len(y_val)):
        l=l+y_val[i]*((1/denom[i])*s[i]) # Estimating the polynomial
```

- ❖ Lastly **<u>fitViaLagrangePoly</u>** is defined to use Lagrange interpolation to interpolate the polynomials for given set of data points
- ❖ We use the polynomials class to multiply the polynomials in the numerator and as shown in red also we store the denominators and finally we multiply with the data values and add it to get the estimated polynomial