1asg

Danial Rangavar

April 9, 2019

Abstract

1 1a

```
import numpy


#
    ################################################################################################
###two functions which together will return the poisson distribution at any
    give point:


#1.defining the poisson function when k = 0 (no factorial):
def zeroPoisson(lamb):
    result = numpy.float64(numpy.exp(-lamb))
    return result

#2.now that we have the poisson function value at a given lambda (with k = 0);
    one can derive a relation for p(lambda,k): p(lambda,k) = p(lambda,k-1) *
    lambda / k. this function will use this relation to find poisson's function
     value at the give k recursively.
def poisson(lamb,k):
    if k == 0:
        result = numpy.float64(zeroPoisson(lamb))
        return result
    result = numpy.float64(poisson(lamb,k-1) * lamb / k)
    return result


#
    ################################################################################################
```

poissonFunction.py

```
This is my Seed: 5761015743107

1(a):
P(1,0) = 0.367879441171442334024277442950
P(5,10) = 0.018132788707821874407688511610
P(3,21) = 0.000000000010193398241110192582
P(2.6,40) = 0.000000000000000000000000000000
P(101,200) = 0.000000000000000001269531392047
```

text/1a.txt

```
#
    ##########################################################################################################
###random number generator, which for each time calling it, will produce a new
    random number for you

```

```python

lastRandom = 5761015743107 #starts with this seed, but this is actually my storage unit for later stages
print('This is my Seed: %i'%lastRandom)
def XORshift():
    a = 21
    b = 43 #XORshift values
    c = 4
    #first inserting the global randomNumber, which is the previous number
    global lastRandom
    #now use MLCG
    lastRandom = (2685821657736338717*lastRandom)%(2**64)
    #combine it with XORshift
    lastRandom ^= (lastRandom>>a)
    lastRandom ^= (lastRandom<<b)
    lastRandom ^= (lastRandom>>c)
    #to specify a range for my random number
    return float(lastRandom%(2**64))/(2**64)


# ##########################################################################################
###producing 1000 random numbers between 0 and 1, and then plotting the scatter plot of x_i+1 vs x_i


tousandRandomNumbers = numpy.empty(1000)
for i in range(1000):
    tousandRandomNumbers[i] = XORshift()

nextRandomNumbers = numpy.empty(1000)
nextRandomNumbers[999] = nextRandomNumbers[0]
for i in range(999):
    nextRandomNumbers[i] = tousandRandomNumbers[i+1]

plot.scatter(nextRandomNumbers,tousandRandomNumbers)


# ########################################################################################
###generating one million random numbers between 0 and 1, and plotting the results in 20 bins


millionRandomNumbers = numpy.empty(1000000)
for i in range(1000000):
    millionRandomNumbers[i] = XORshift()

arange = numpy.arange(0,1.05,0.05)
plot.hist(millionRandomNumbers,arange)
```

randomNumberGenerator.py

```python
import numpy
from randomNumberGenerator import XORshift

randoms = XORshift(10,3)
print(randoms)

a = randoms[0]/(2**64)*1.4 + 1.1
b = randoms[1]/(2**64)*1.5 + 0.5
c = randoms[2]/(2**64)*2.5 + 1.5
#a = 2
#b = 1
#c = 1

print(a,b,c)

```

```python
def iWantToIntegrateThis(x):
    if x == 0: y = 0
    else:
        y = 4. * numpy.pi * (x/b)**(a-3.) * numpy.exp(-(x/b)**c) * x**2.
    return y

def Integration(f,a,b,steps):
    process = numpy.zeros((steps,steps),dtype=numpy.float64)
    process[0,0] = ( f(a)+f(b) )*(b-a)/2.

    for i in range(steps-1):
        process[i+1,0] = (b-a)*sum( f(a + (b-a)*(2.*j+1)/(2.**(i+1)) ) for j in range(2**i) )
        process[i+1,0] = process[i,0]/2. + process[i+1,0]/(2.**(i+1))
        for k in range(i+1):
            process[i+1,k+1] = process[i+1,k] + (process[i+1,k]-process[i,k])/(4**(k+1) -1.)

    return process

calc = Integration(iWantToIntegrateThis,0.,5.,6)
print(calc)

A = 1. / calc[-1,-1]
print(a,b,c,A)

def sqr(x):
    return x**2

sqrd = Integration(sqr,0.,5.,6)
print(sqrd)
```

integration.py

```python
import numpy


# ##########################################################################################################
###in the upcoming lines, I will define 4 functions, which I need for Natural Cubic Interpolation:


#(first) solve the tridiagonal linear equation (which is the result of Cubic Interpolation method), and by solving I mean: given the 3 diagonal arrays (which I will define later in the defineCubicSpline function), this function will solve the set of linear equations by matrice manipulation methods, and returns the solutions, which are y second derivatives at the given points.

#up is the upper diagonal, diag is the middle, and down in the bottom diagonl, rhs (short for rigth hand side), is the right hand side of each equation. I want to achieve a diagonal matrice with the diagonal values all equal to one (basic linear algebra), which in this case (tridiagonal matrice) is achievable, at the end the rhs values, would be the solutions:

#1.first lets make a two diagonal matrice by getting rid of the bottom diag. so I solve for each down[i] and add the upper row to the current row, to make the down[i] zero, since for each row, down[i] is below the diag[i-1] of the previous row, I need to add -row[i-1]*down[i-1]/diag[i-1] to each row which will result down[i] becoming zero, and since I know that already, Im just keeping track of the rest of the row (up,diag,and rhs).
#2.after the first loop is finished, since the last row contains only diag and rhs, its solvable for rhs, meaning dividing the row by diag value to make diag equal to one (remember, from algebra, this is the goal)
#3.now that we know the solution for the last row, we can use it to find the rest of the solutions, with the same method as above, but this time moving from bottom to top instead, and getting rid of up elements, same as we did for the down elements.
```

```
#4.at the end I'm left out with a diagonal matrice with all diag values equal to one, so the rhs of this matrice is simply the solution to the starting linear equation.

def solve(up,diag,down,rhs):
    #1
    n = len(diag)
    for i in range(n-1):
        diag[i+1] = diag[i+1] - up[i]*down[i]/diag[i]
        rhs[i+1] = rhs[i+1] - rhs[i]*down[i]/diag[i]
    #2
    rhs[-1] = rhs[-1]/diag[-1]
    diag[-1] = 1.
    #3
    for i in range(n-1):
        rhs[n-2-i] = (rhs[n-2-i] - up[n-2-i]*rhs[n-1-i])/diag[n-2-i]
        diag[n-2-i] = 1.
        up[n-2-i] = 0.
    #4
    return rhs


#(second) defining the Cubic Spline, from the given data set, simply by the given formula for the cubic spline.

#1.the loop simply goes through all the data set, and calculates the coefficients of the cubic interpolation formula, which together will define a tridiagonal matrice,solvable to find the values of y second derivatives at each point.
#2.at the end I'm returning this coefficients as the three diagonal arrays of the tridiagonal matrice, which will be solved by the above function.

def defineCubicSpline(x,y):
    n = len(x)
    up = numpy.zeros(n-1,dtype=numpy.float64)
    diag = numpy.ones(n,dtype=numpy.float64)
    down = numpy.zeros(n-1,dtype=numpy.float64)
    rhs = numpy.zeros(n,dtype=numpy.float64)
    #1
    for i in range(1,n-1):
        diag[i] = (x[i+1]-x[i-1])/3.
        rhs[i] = (y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i-1])
        if i < n-2:
            down[i] = (x[i]-x[i-1])/6.
            up[i] = (x[i+1]-x[i])/6.
    #2
    return up,diag,down,rhs


#(third) finds where does x belong in a give data set (segment).

#tries one point in the middle of the array, if x is bigger than this point, gets rid of the bottom half, and if x is smaller than that point, gets rid of the upper half. do this till left out with an array of length 2. returns that array, and the original array number of the bottom member.

def findX(array,x):
    i = 0
    seq = array
    while len(seq) != 2:
        m = int(len(seq)/2)
        if x >= seq[m]:
            del seq[0:m]
            i = i + m
        elif x < seq[m]:
            del seq[m+1:]
    return seq,i


#(fourth) now that we know where does this point of interest belongs in our
```

```
    grid (by using the function above), and finnaly, we can insert y second derivative values(solved by functions 1 and 2) into the natural cubic interpolator formula, to get back the interpolated value at the point of interest, f(point of interest).

def cubicInterpolation(pointOfInterest,segmentNumber,cubicSolutions,x,y):
    i = segmentNumber
    result = (cubicSolutions[i]/6.)*( (pointOfInterest-x[i+1])**3./(x[i]-x[i+1]) - (pointOfInterest-x[i+1])*(x[i]-x[i+1]) )
    result = result - (cubicSolutions[i+1]/6.)*( (pointOfInterest-x[i])**3./(x[i]-x[i+1])-(pointOfInterest-x[i])*(x[i]-x[i+1]) )
    result = result + ( y[i]*(pointOfInterest-x[i+1])-y[i+1]*(pointOfInterest-x[i]) )/(x[i]-x[i+1])
    return result


#putting four functions together to make the one-dimensional interpolator, which includes:
#1.defineCubicSpline
#2.solve
#3.findX
#4.cubicInterpolator
#this function gets two arrays for x(should be sorted), and f(x) values; and a point of interest, and retturns the interpolated values for f(point of interest) at the end.

def interpolate(xArray,yArray,pointOfInterest):
    #find the tridiagonal matrice from the given values for xArray, and yArray:
    up,diag,down,rhs = defineCubicSpline(xArray,yArray)
    #solve the tridiagonal matrice to find the values for y's second derivative at each one of the given x points:
    solution = solve(up,diag,down,rhs)
    #copy xArray into a new array(xCopy) to keep the xArray safe from the manipulation of the upcoming function(findX):
    xCopy = []
    for i in range(len(x)):
        xCopy.append(x[i])
    #finding the location of the point of interst in the sorted x array, which returns the closest upper and lower x to the point of interest as the segment, and the array number of the lower x belonging to that segment as lowerX(so, for example, if point of interest lies between the first and second members of the xArray, this function will return [xArray[0],xArray[1]] as the segment, and 0 as lowerX.
    segment,lowerX = findX(xCopy,pointOfInterest)
    #now we know where does this point of interest belongs in our grid, and finnaly, we can insert y second derivative values into the natural cubic interpolator formula, to get back the interpolated value at the point of interest, f(point of interest):
    valueAtPointOfInterest = cubicInterpolation(pointOfInterest,lowerX,solution,xArray,yArray)
    return valueAtPointOfInterest
#################################################################################################################
#now I want to use the above function to make a three dimensional interpolator, but for upper dimensions, the idea would remain the same:


#first, what I want to do? well, I know the values of f at a set of points (xi,yi,zi), and I want to find its value at (x,y,z). for that, I will devide the interpolation to three one-dim interpolations, and apply them as follows, and in each step I attemp to lower the dimension of the function by one, by getting rid of one of the coordinates.
#1.imagine the line f(yConstant,zConstant). for this line, I can interpolate the f(x,yConstant,zConstant) which is a 1-dim function, f_yConstant_zConstant(x)
#2.repeat the above process, but for a different value of yConstant this time. this is like doing the interpolation in the surface of f(zConstant).
#3.do the step 2, till you go through all the yi. at this point you have a set of new grid points which would look like this: f(x,yi,zConstant). which
```

```
    define a line passing through f(x,y,zConstant).
#4.interpolate on this line, and you will get the value at f(x,y,zConstant).
#5.to the above steps for all the zi, and you will end up with a line f(x,y,zi)
     passong through f(x,y,z).
#6.finally interpolate that last line to get the f(x,y,z).

def threeDInterpolate(xArray,yArray,zArray,fArray,xInterest,yInterest,zInterest
    ):
    zConstantNumber = 0
    zLine = numpy.zeros(len(zArray)) #defining the line of f(xInterest,
    yInterest,zArray)
    #loop to produce the f(xInterest,yInterest,zArray) for all the zArray
    for zConstant in zArray:
        yConstantNumber = 0
        yLine = numpy.zeros(len(yArray)) #defining the line of f(xInterest,
    yArray,zConstant)
        #loop to produce the f(xInterest,yArray,zConstant) for all the yArray
        for yConstant in yArray:
            xLine = numpy.zeros(len(xArray)) #defining the line of f(xArray,
    yConstant,zConstant)
            #loop to produce the f(xArray,yConstant,zConstant) for all xArray
            for i in range(len(xArray)):
                xLine[i] = fArray[i,yConstantNumber,zConstantNumber]
            yLine[yConstantNumber] = interpolate(xArray,xLine,xInterest)
            yConstantNumber = yConstantNumber + 1
        zLine[zConstantNumber] = interpolate(yArray,yLine,yInterest)
        zConstantNumber = zConstantNumber + 1
    result = interpolate(zArray,zLine,zInterest)
    return result


#
    ##########################################################################################

###now I want to produce that 6240 values for A(a,b,c)


#all a,b,c values in ascending order
aArray = numpy.arange(1.1,2.6,0.1)
bArray = numpy.arange(0.5,2.1,0.1)
cArray = numpy.arange(1.5,4.1,0.1)

#count = 0
Alist = numpy.zeros(((15),(16),(26)))
for i in range(15):
    for j in range(16):
        for k in range(26):
            def iWantToIntegrateThis(x):
                if x == 0: y = 0
                else:
                    y = 4. * numpy.pi * (x/bArray[j])**(aArray[i]-3.) * numpy.
    exp(-(x/bArray[j])**cArray[k]) * x**2.
                return y
            Alist[i][j][k] = findA()
            print('A(a,b,c) = %2.7f(%1.1f,%1.1f,%1.1f)'%(Alist[i][j][k],aArray[
    i],bArray[j],cArray[k]))
            #count = count + 1
            #print(count)


########################################
```

naturalCubicInterpolator.py

```
import numpy

def n(x,a,b,c,A,N):
    y = A*N*(x/b)**(a-3) * numpy.exp(-(x/b)**c)
    return y

```

```

###############################################################
###Ridder's differentiation method, returns the whole table


#1.caculate the centeral differential
#2.decreasing h and repeating 1
#3.combining pairs using Ridder's formula

def differentiator(f,x,h,d,steps,analAnswer,howManyDigits):
    #1,2
    process = numpy.zeros((steps,steps),dtype = numpy.float64)
    #calculating d(x), for n values of h, in descending oreder
    for n in range(steps):
        process[n,0] = ( f(x+h/d**n)-f(x-h/d**n) )/( 2.*(h/d**n) )
    bestError = float((process[steps-1,0]-analAnswer)**2.)  #choosing the best error
    #3
    for j in range(1,steps):
        for i in range(0,steps-j):
            process[i,j] = ( (d**(2.*j))*process[i+1,j-1]-process[i,j-1] )/((d**(2.*j))-1.)
            error = float((process[i,j]-analAnswer)**2.)
            #if the error is small enough, break it
            if error < 10.**(-(2.*howManyDigits)):
                return process[i,j],error
                break
        #if the error is getting large, break and output the table
        if float(bestError) < float(error):
            return process[i,j],error
            break
        bestError = error
    return process,bestError


#################################################################################

def function(x):
    y = numpy.exp(x)/(numpy.sin(x)-x**2.)
    return y

def cube(x):
    y = x**3
    return y
```

differentiation.py

```
import numpy
from matplotlib import pyplab as plot


######################################
###producing random values for a,b,c


a = XORshift()*1.4 + 1.1
b = XORshift()*1.5 + 0.5
c = XORshift()*2.5 + 1.5


##########################################################################
###defining the function that I want to integrate, in order to find a.


def iWantToIntegrateThis(x):
    global a
    global b
    global c
    if x == 0: y = 0
    else:
```

```python
        y = 4. * numpy.pi * (x/b)**(a-3.) * numpy.exp(-(x/b)**c) * x**2.
    return y


################################################################
###building up the numerical integrator, using Romberg algorithm


#I'll make a table, containg all the steps taken to the final estimation of the integral (process)
#1.simplest estimation, using just one trapezoid.
#2.doubling the number of trapezoids in each step, and since we already know the values at half of these points, calculating the other half. so we are making better and better estimations by increasing the number of trapizoids, but, this is getting better linearly, and very slow.
#3.we can make a better estimation, and very fast, using romberg integration. for that, solving the romberg equation, we use the previously calculated values, and make a better estimation (practically we are making our estimation better, by using the fact that we already know our error estimations).
#this return the whole table and the best estimated value which is the [-1,-1] member of the table.

def integration(f,a,b,steps):
    process = numpy.zeros((steps,steps),dtype=numpy.float64)
    #1
    process[0,0] = ( f(a)+f(b) )*(b-a)/2.
    #2
    for i in range(steps-1):
        process[i+1,0] = (b-a)*sum( f(a + (b-a)*(2.*j+1)/(2.**(i+1)) ) for j in range(2**i) )
        process[i+1,0] = process[i,0]/2. + process[i+1,0]/(2.**(i+1))
        #3
        for k in range(i+1):
            process[i+1,k+1] = process[i+1,k] + (process[i+1,k]-process[i,k])/(4**(k+1) -1.)

    return process #this return the whole table and the best estimated value which is the [-1,-1] member of the table.


# ##################################################################################################################
###now using the defined function and the numerical integrator, I can integrate that function, hence the value of A(a,b,c).


def findA():
    denom = integration(iWantToIntegrateThis,0.,5.,5)
    return 1./denom[-1,-1]


######################################
###defining the p(x) function (part d)


def p(x,A,a,b,c):
    y = A*4.*numpy.pi*(x**2.)*(x/b)**(a-3) * numpy.exp(-(x/b)**c)
    return y

#########################################################
###here I'm using p(x) function to produce 100 satellites


#I produce 1500 numbers for x, and 1500 numbers for y, of course different numbers! but, since I want to plot a log-log at the end, the idea is to produce x logarithmically, so I give a chance to the smaller x to appear in the plot.
#for each x, I'm producing a y, taking this y as the random probability for
```

```
that x, if y is smaller than f(x) it can be a member of the final 100 galaxies, else, I throw it away.

n = 0
pointsList = numpy.zeros(100)
while n< 100:
    #generate a x in the logarithmic scale, and then taking it to the linear scale
    random = XORshift()
    logX = random*4.698970004336019-4.
    x = 10.**logX
    #generating a probability for the y, f(x).
    y = XORshift()
    #if f(x) is smaller than the density function value for that x, keep the x, else, forget it.
    if float(y) <= float(p(x)):
        pointsList[n] = x
        n = n+1

print(pointsList)
#print(pointsList[-1])


# ##########################################################################################################
###making 1000 halos each with 100 galaxies, which is the same as repeating the above code but with different random numbers.


#defining bins
binSize = 4.698970004336019/20.
binStarts = numpy.zeros(21)
logBinStarts = numpy.zeros(21)
for i in range(21):
    logBinStarts[i] = -4. + binSize*float(i)
    binStarts[i] = 10.**(logBinStarts[i])
binCounts = numpy.zeros(20,dtype=int)
biggestBinCounts = numpy.zeros(1000)
biggestBinX = []

halos = numpy.zeros((1000,100))
for haloNumber in range(1000):
    n = 0
    #xRandoms = XORshift(10*(haloNumber+1),1500)
    #yRandoms = XORshift(11*(haloNumber+1),1500)
    while n<100:
        logX = XORshift()*4.698970004336019-4.
        x = 10.**logX
        y = XORshift()
        if float(y) <= float(p(x)):
            halos[haloNumber,n] = x
            #checking which bin does this x belong to
            binNumber = int((logX + 4.)/binSize)
            binCounts[binNumber] = binCounts[binNumber]+1
            n = n+1
            #if it belongs to the biggest bin, save the x for that bin to the biggest binX and
            if binNumber == 17:
                biggestBinCounts[haloNumber] = biggestBinCounts[haloNumber]+1
                biggestBinX.append(x)


##########
###plotting


xvals = numpy.arange(0.0001, 5, 0.0001)
plt.loglog(xvals,100.*p(xvals))
plt.ylabel("p(x)")
```

```python
plt.xlabel("x")
plt.xlim(.0001,5)

binCountsAverage = numpy.zeros(20)
for i in range(20):
    binCountsAverage[i] = binCounts[i]/1000.

for i in range(20):
    xvals = numpy.arange(binStarts[i],binStarts[i+1],(binStarts[i+1]-binStarts[i])/3.)
    pltFunction = numpy.zeros(len(xvals))
    for j in range(len(xvals)):
        pltFunction[j] = binCountsAverage[i]
    plt.loglog(xvals,pltFunction)

#plt.show()
plt.savefig('binning.pdf')


# ##########################################################################################################
###this function will sort a given array in ascending order, for this I am using the quich sort algorithm.


#1.get the first, end, and middle element, and sort them. choose middle element as the pivot.
#2.for i and j in the same loop (i from 0 to n, j from n-1 to 0), if a[i]>=pivot, and a[j]<=pivot, if j>i, swap a[i], a[j]. aim of this step is to make sure that all the elements at pivots right are bigger(or equal) than pivot and all the elements at its left are smaller(or equal) than pivot.
#3.pivot is at its right place, so now take the left and rigth arrays excluding pivot and perform the same thing.
#ignore all the prints, they are just for myself in order to debug the code, if necessary.

def pivotSort(a,first,last):
#sorting first,last,and middle and choosing the middle as the pivot
    middle = int((first+last+1)/2)
    pivotNumber = middle
    if a[first] > a[last]: a[first], a[last] = a[last], a[first]
    if a[first] > a[middle]: a[first], a[middle] = a[middle], a[first]
    if a[middle] > a[last]: a[last], a[middle] = a[middle], a[last]
    #print('start')
    #print('first %s,last %s'%(first,last))
    #print(a)
    if last-first == 1:return #if the length is of the array is 2, it is already sorted by performing the above process, so no need to continue.
    xP = a[middle]
    #2.looping from the segments first element to its last element, for i, j at the same time.
    i=first
    j=last
    while i<last+1 and j>=first:
        if a[i]<xP:i = i+1
        if a[j]>xP:j = j-1
        if a[i]>=xP and a[j]<=xP:
            if j<=i:break #i,j crossed or saw each other at the same element(pivot), so break.
            else:
                starti = i
                startj = j
                startPivot = pivotNumber
                a[i],a[j]=a[j],a[i]
                #print('startingPivot:%s'%startPivot)
                #print('swapping %s:%s'%(i,j))
                #print(a)
                #print('newi,j:(%s,%s)'%(i,j))
                #if the element being swaped is pivot, take care of the pivot
```

```
    number.
                if starti == startPivot:
                    pivotNumber = j
                    i = i+1
                    #print('i was pivot')
                #if the element being swaped is not pivot, continue.
                elif startj != startPivot:
                    j = j-1
                #if the element being swaped is pivot, take care of the pivot number.
                if startj == startPivot:
                    pivotNumber = i
                    j = j-1
                    #print('j was pivot')
                #if the element being swaped is not pivot, continue.
                if startj != startPivot and starti != startPivot:
                    i = i+1
                #print('newi,j:(%s,%s)'%(i,j))
                #print('pivotNumberNew: %s'%pivotNumber)
    #print('ended the loop,results')
    #print(a)
    #print('pivotNumber: %s'%pivotNumber)
    #3.make sure that we still need to continue the algorithm (we haven't reached the first or last element)
    #sort left part
    if pivotNumber-1>0 and pivotNumber-1>first:
        pivotSort(a,first,pivotNumber-1)
    #sort right part
    if pivotNumber<last-1 and pivotNumber+1<last:
        pivotSort(a,pivotNumber+1,last)

#these are just tests, ignore them.
#myArray = [1012,57,42,63,97,1234,53,41253,112,4,566,123,34,153]
#myArray = [31,42,42,42,42,42,53,41253,112,4,566,123,34,153]
#pivotSort(myArray,0,13)


#
    ##########################################################################################################################################
```

distributionAndSorting.py

```
import numpy

def function(x):
    y = x**2. - 16.
    return y


#########################
###root finding algorithm


#I'm using the bisection method, beacause it will converge for sure. this method works by linearly connecting two points of the function that have values with different sign. taking the avarage of these two points and calculating f at that point, if its bigger than zero ,then this is the new second guess for the root, and if its smaller than zero, this is the new first guess for the root. repeat this process untill you get close enough to the root.

def bisection(f,firstGuess,secondGuess,desiredError):
     f0 = f(firstGuess)
     f1 = f(secondGuess)
     newGuess = (firstGuess+secondGuess)/2.
     #print(firstGuess,secondGuess,desiredError,newGuess)
     fN = f(newGuess)
     if float(f0*fN) >= 0.:
        firstGuess = newGuess
```

```
    if float(f0*fN) < 0.:
        secondGuess = newGuess
    return newGuess if float(fN**2.) < float((desiredError)**2.) else bisection(f,firstGuess,secondGuess,desiredError)


#################################################
#firstTest = bisection(function,0.,8.,0.001)
#print(firstTest)
```

rootFinding.py

```
###sort
def pivotSort(a,first,last):
#sorting first,last,and middle and choosing the middle as the pivot
    middle = int((first+last+1)/2)
    pivotNumber = middle
    if a[first] > a[last]: a[first], a[last] = a[last], a[first]
    if a[first] > a[middle]: a[first], a[middle] = a[middle], a[first]
    if a[middle] > a[last]: a[last], a[middle] = a[middle], a[last]
    #print('start')
    #print('first %s,last %s'%(first,last))
    #print(a)
    if last-first == 1:return #if the length is of the array is 2, it is already sorted by performing the above process, so no need to continue.
    xP = a[middle]
    #2.looping from the segments first element to its last element, for i, j at the same time.
    i=first
    j=last
    while i<last+1 and j>=first:
        if a[i]<xP:i = i+1
        if a[j]>xP:j = j-1
        if a[i]>=xP and a[j]<=xP:
            if j<=i:break #i,j crossed or saw each other at the same element(pivot), so break.
            else:
                starti = i
                startj = j
                startPivot = pivotNumber
                a[i],a[j]=a[j],a[i]
                #print('startingPivot:%s'%startPivot)
                #print('swapping %s:%s'%(i,j))
                #print(a)
                #print('newi,j:(%s,%s)'%(i,j))
                #if the element being swaped is pivot, take care of the pivot number.
                if starti == startPivot:
                    pivotNumber = j
                    i = i+1
                    #print('i was pivot')
                #if the element being swaped is not pivot, continue.
                elif startj != startPivot:
                    j = j-1
                #if the element being swaped is pivot, take care of the pivot number.
                if startj == startPivot:
                    pivotNumber = i
                    j = j-1
                    #print('j was pivot')
                #if the element being swaped is not pivot, continue.
                if startj != startPivot and starti != startPivot:
                    i = i+1
                #print('newi,j:(%s,%s)'%(i,j))
                #print('pivotNumberNew: %s'%pivotNumber)
    #print('ended the loop,results')
    #print(a)
    #print('pivotNumber: %s'%pivotNumber)
    #3.make sure that we still need to continue the algorithm (we haven't reached the first or last element)
```

```
    #sort left part
    if pivotNumber-1>0 and pivotNumber-1>first:
        pivotSort(a,first ,pivotNumber-1)
    #sort right part
    if pivotNumber<last-1 and pivotNumber+1<last:
        pivotSort(a,pivotNumber+1,last)







################
###read the data, which will return a list of x

f = open('data/satgals_m14.txt','r')
lines = f.readlines()
f.close()

haloNumber = int(lines[3].split('\n')[0])
haloCount = numpy.zeros(haloNumber)
Xlist = []

haloID = 0
for i in range(4,len(lines)):
    firstRead = lines[i]
    if firstRead == '#\n' and i+1 != len(lines):
        secondRead = lines[i+1]
        if secondRead != '#\n':
            haloCount[haloID] = haloCount[haloID] + 1
            Xlist.append(float(secondRead.split(' ')[0]))
        else: haloID = haloID + 1
    else:
        if i-1 != 3:
            secondRead = lines[i-1]
            if secondRead != '#\n':
                haloCount[haloID] = haloCount[haloID] + 1
                Xlist.append(float(secondRead.split(' ')[0]))
        if i+1 != len(lines):
            nextLine = lines[i+1]
            if nextLine == '#\n':haloID = haloID + 1


####################################
###find a,b,c to maximize likelihood


#first I want to define a function which returns the log-likelihood of a given data realization. for that, I keep in mind that from eqiation (2), I have the density profile, and the likelihood of a realization set, is simply the product of the n(x) for each one of the X = (x_i), so if I take Ln() from both sides, the log likelihood would be the product of each one of the Ln(n(x_i)) for all X = (x_i).

def negativeLogLikelihoodFunction(a,b,c,A,X):
    result = 0.
    for i in range(len(X)):
        l = numpy.log(A)+(a-3.)*numpy.log(X[i]/b)-c*X[i]/b
        result = result - l
    return result


#now I want to calculate this function, for those 6240 point that already calculated the A(a,b,c) for

negLogLikeList = numpy.zeros(((15),(16),(26)))
for i in range(15):
    for j in range(16):
        for k in range(26):
```

```python
            negLogLikeList[i][j][k] = negativeLogLikelihoodFunction(aArray[i], bArray[j],cArray[k], Alist[i][j][k], Xlist)

#writing a function to minimize another function using Downhill Simplex.
fakeLogLikeList = []
for i in range(15):
    for j in range(16):
        for k in range(26):
            fakeLogLikeList.append(negLogLikeList[i][j][k])

pivotSort(fakeLogLikeList ,0, len(fakeLogLikeList)-1)
dicX = {}
for n in range(len(fakeLogLikeList)):
    for i in range(15):
        for j in range(16):
            for k in range(26):
                if (fakeLogLikeList[n]-negLogLikeList[i][j][k])**2.<=0.000000001: dicX[n] = [aArray[i],bArray[j],cArray[k], Alist[i][j][k], negLogLikeList[i][j][k]]

def downHill(fakeLogLikeList,
sumA = 0
sumB = 0
sumC = 0
for i in range(len(X)):
    sumA = sumA + dicA[X[i]]
    sumB = sum
    sumC = sum
```

maximizeLikelihood.py