In [ ]:

```
'''>>>>> start CodeP1.2
    V.P. Carey ME249, Spring 2021'''


'''INITIALIZING PARAMETERS'''
n = []
ntemp = []
gen=[0]
n1avg = [0.0]
n2avg = [0.0]
n3avg = [0.0]
n4avg = [0.0]
n5avg = [0.0]
meanAFerr=[0.0]
aFerrmeanavgn=[0.0]

#set program parameters
NGEN = 6000      #number of generations (steps)
MFRAC = 0.5   # faction of median threshold

# here the number of data vectors equals the number of DNA strands (or organi
# they can be different if they are randomly paired to compute Ferr (survivab
for k in range(NGEN-1):
    gen.append(k+1)   # generation array stores the
    meanAFerr.append(0.0)
    aFerrmeanavgn.append(0.0)
    n1avg.append(0.0)
    n2avg.append(0.0)
    n3avg.append(0.0)
    n4avg.append(0.0)
    n5avg.append(0.0)

'''guesses for initial solution population'''
n0i =  -1.0
n1i = 0.00020
n2i = 3.4
n3i = 0.05
n4i = 1.325
n5i = 0.165

n0i =  -1.0
n1i = 0.00030
n2i = 4.4
n3i = 0.07
n4i = 1.325
n5i = 0.165


#- initialize arrays before start of evolution loop EL
#then - create array of DNA strands n[i] and ntemp[i] with dimesnion NS = 5

#i initialize array where rows are dna vectors [n0i,n1i,...n5i] with random p
```

```
n =  [[-1., n1i+0.001*random(), n2i+0.1*random(), n3i+0.0001*random(), n4i+0.
for i in range(ND):
    n.append([-1., n1i+0.0001*random(), n2i+0.001*random(), n3i+0.0001*random
#print (n) # uncomment command to print array so it can be checked

# store also in wtemp
ntemp = deepcopy(n)

#initialize Ferr values an dother loop parameters
#define arrays of Ferr (error) functions
#individual solution error and absoute error
Ferr =  [[0.0]]
#population average solution error and absoute error
Ferravgn =  [[0.0]]
aFerr =  [[0.0]]
aFerravgn =  [[0.0]]

#store zeros in ND genes
for i in range(ND-1):
    #individual solution error and absoute error
    Ferr.append([0.0])
    aFerr.append([0.0])
    #population average solution error and absoute error
    Ferravgn.append([0.0])
    aFerravgn.append([0.0])
#print (Ferr)

aFerrmeanavgnMin=1000000000.0
# these store the  n values for minimum population average error durng NGEN g
n1min = 0.0
n2min = 0.0
n3min = 0.0
n4min = 0.0
n5min = 0.0
aFerrta = 0.0
# these store the time averaged n values durng from generation 800 to NGEN ge
n1min = 0.0
n1ta = 0.0
n2ta = 0.0
n3ta = 0.0
n4ta = 0.0
n5ta = 0.0

'''START OF EVOLUTION LOOP'''
# ------------------------------------------
# k is generation number, NGEN IS TOTAL NUMBER OF GENERATIONS COMPUTED
for k in range(NGEN):

    '''In this program , the number of organisms (solutions) NS is taken to b
    number of data points ND so for each generation, each solution can be com
    data point and all the data is compared in each generation.  The order of
    that holds the solution constants is constantly changing due to mating an
    is random.'''
```

```
'''CALCULATING ERROR (FITNESS)
In this program, the absolute error in the logrithm of the physical heat
used to evaluate fitness.'''

# Here we calculate error Ferr and absolute error aFerr for each data poi
# for specified n(i), and calculate (mean aFerr) = aFerrmean
# and (median aFerr) = aFerrmedian for the data collection and specified 
# Note that the number data points ND equals the number of solutions (org
#========================================================================
for i in range(ND):

    Ferr[i] = n[i][0]*lydata[i][0] + math.log(n[i][1]) + n[i][2]*lydata[i
    Ferr[i] = Ferr[i] + n[i][3]*math.log( ydata[i][2] )

    aFerr[i] = abs(Ferr[i])/abs(lydata[i][0])  #- absolute fractional err
#-------------
aFerrmean = numpy.mean(aFerr) #mean error for population for this generat
meanAFerr[k]=aFerrmean  #store aFerrmean for this generation gen[k]=k
aFerrmedian = numpy.median(aFerr) #median error for population for this g

'''SELECTION'''
#pick survivors
#[2] calculate survival cutoff, set number kept = nkeep = 0
#===============================================
clim = MFRAC*aFerrmedian  #cut off limit is a fraction/multiplier MFRAC o
nkeep = 0

# now check each organism/solution to see if aFerr is less than cut of li
#if yes, store n for next generation population in ntemp, at end nkeep = 
#and number of new offspring = NS-nkeep
#===============================================
for j in range(NS):  # NS Ferr values, one for each solution in populatio
    if (aFerr[j] < clim):
        nkeep = nkeep + 1
        #ntemp[nkeep][0] = n[j][0] = -1 so it is unchanged;
        ntemp[nkeep-1][1] = n[j][1];
        ntemp[nkeep-1][2] = n[j][2];
        ntemp[nkeep-1][3] = n[j][3];
        ntemp[nkeep-1][4] = n[j][4];
        ntemp[nkeep-1][5] = n[j][5];
#now have survivors in leading entries in list of ntemp vectors from 1 to
#compute number to be added by mating
nnew = NS - nkeep

'''MATING'''
#[4] for nnew new organisms/solutions,
# randomly pick two survivors, randomly pick DNA (n) from pair for each o
#===============================================================
for j in range(nnew):
    # pick two survivors randomly
    nmate1 = numpy.random.randint(low=0, high=nkeep+1)
```

```
        nmate2 = numpy.random.randint(low=0, high=nkeep+1)

        #then randomly pick DNA from parents for offspring

        '''here, do not change property ntemp[nkeep+j+1][0], it's always fixe
        #if (numpy.random.rand() < 0.5)
        #    ntemp[nkeep+j+1][0] = n[nmate1][0]
        #else
        #    ntemp[nkeep+j+1][0] = n[nmate2][0]

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][1] = n[nmate1][1]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][1] = n[nmate2][1]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][2] = n[nmate1][2]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][2] = n[nmate2][2]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][3] = n[nmate1][3]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][3] = n[nmate2][3]*(1.+0.09*2.*(0.5-numpy.random.
        '''
        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][4] = n[nmate1][4]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][4] = n[nmate2][4]*(1.+0.09*2.*(0.5-numpy.random.

        if (numpy.random.rand() < 0.5):
            ntemp[nkeep+j+1][5] = n[nmate1][5]*(1.+0.09*2.*(0.5-numpy.random.
        else:
            ntemp[nkeep+j+1][5] = n[nmate2][5]*(1.+0.09*2.*(0.5-numpy.random.
        '''
    #==============================================
    n = deepcopy(ntemp)   # save ntemp as n for use in next generation (next


    '''AVERAGING OVER POPULATION AND OVER TIME, FINDING MINIMUM ERROR SET OF
    # [6] calculate n1avg[k], etc., which are average n values for population
    # at this generation k
    #==============================================
    #initialoze average n's to zero and sum contribution of each member of th
    n1avg[k] = 0.0;
    n2avg[k] = 0.0;
    n3avg[k] = 0.0;
    n4avg[k] = 0.0;
    n5avg[k] = 0.0;
    for j in range(NS):
        n1avg[k] = n1avg[k] + n[j][1]/NS;
        n2avg[k] = n2avg[k] + n[j][2]/NS;
        n3avg[k] = n3avg[k] + n[j][3]/NS;
```

```
            n4avg[k] = n4avg[k] + n[j][4]/NS;
            n5avg[k] = n5avg[k] + n[j][5]/NS;

        # Here we compute aFerravgn[i] = absolute Ferr of logrithm data point i u
        # for this solutions generation k
        # aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of o
        #
        #===============================================
        for i in range(ND):
            Ferravgn[i] = -1.*lydata[i][0] + math.log(n1avg[k]) + n2avg[k]*lydata
            Ferravgn[i] = Ferravgn[i] + n3avg[k]*math.log( ydata[i][2] )

            #aFerravgn[i] = abs(Ferr[i])/abs(lydata[i][0])
            aFerravgn[i] = abs(Ferravgn[i])/abs(lydata[i][0])
        #--------------
        aFerrmeanavgn[k] = numpy.mean(aFerravgn)


        # next, update time average of n valaues in population (n1ta[k], etc.)
        # for generations = k > 800 up to total NGEN
        #===============================================
        aFerrta = aFerrta + aFerrmeanavgn[k]/NGEN
        if (k > 800):
            n1ta = n1ta + n1avg[k]/(NGEN-800)
            n2ta = n2ta + n2avg[k]/(NGEN-800)
            n3ta = n3ta + n3avg[k]/(NGEN-800)
            n4ta = n4ta + n4avg[k]/(NGEN-800)
            n5ta = n5ta + n5avg[k]/(NGEN-800)

        # compare aFerrmeanavgn[k] to previous minimum value and save
        # it and corresponding n(i) values if the value for this generation k is
        #===============================================
        if (aFerrmeanavgn[k] < aFerrmeanavgnMin):
            aFerrmeanavgnMin = aFerrmeanavgn[k]
            n1min = n1avg[k]
            n2min = n2avg[k]
            n3min = n3avg[k]
            n4min = n4avg[k]
            n5min = n5avg[k]

        #print('avg n1-n4:', n1avg[k], n2avg[k], n3avg[k], n4avg[k], aFerrmeanavg
        #print ('kvalue =', k)
        '''end of evolution loop'''
        # ------------------------------------------------------------------
        # ------------------------------------------------------------------


    # ------------------------------------------------------------------
    #final print and plot of results
    # ------------------------------------------------------------------
    print('ENDING: pop. avg n1-n3,aFerrmean:', n1avg[k], n2avg[k], n3avg[k], aFer
    print('MINUMUM:  avg n1-n3,aFerrmeanMin:', n1min, n2min, n3min, aFerrmeanavgn
    print('TIME AVG:    avg n1-n3,aFerrmean:', n1ta, n2ta, n3ta, aFerrta)
```

```
#SETTING UP PLOTS

#========
#initialize values
qpppred =  [[0.0]]
qppdata =  [[0.0]]
for i in range(ND-1):
    qpppred.append([0.0])
    qppdata.append([0.0])
#calculate predicted and data values to plot
for i in range(ND):
    qpppred[i] = n1min*(ydata[i][1]**n2min) * ((ydata[i][2])**n3min)
    qppdata[i] = ydata[i][0]

#========

# constants evolution plots
# x axis values are generation number
# corresponding y axis values are mean absolute population error aFerrmeanavg
# plotting the points

plt.rcParams.update({'font.size': 18})

# aFerrmeanavgn[k] is the mean of the Ferravgn[i] for the population of organ
# computed using the mean n values
plt.plot(gen, aFerrmeanavgn)
plt.plot(gen, n1avg)
plt.plot(gen, n2avg)
plt.plot(gen, n3avg)
plt.legend(['aFerrmeanavgn', 'n1 avg', 'n2 avg', 'n3 avg'], loc='lower left')
#plt.plot(gen, n4avg)
#plt.plot(gen, n5avg)
#plt.legend(['aFerrmeanavgn', 'n1 avg', 'n2 avg', 'n3 avg', 'n4 avg', 'n5 avg

# naming the x axis
plt.xlabel('generation')
# naming the y axis
plt.ylabel('constants and error')
plt.loglog()
plt.yticks([0.01,0.1,1.0,10])
plt.xticks([1,10,100,1000,10000])
plt.show()

# data vs. predicted heat flux plot
plt.scatter(qpppred, qppdata)
plt.title('Genetic Algorithm')
plt.xlabel('predicted heat flux (W/cm^2), n1min - n3min constants')
plt.ylabel('measured heat flux (W/cm^2)')
plt.loglog()
plt.xlim(xmax = 1000, xmin = 10)
plt.ylim(ymax = 1000, ymin = 10)
plt.show()

'''>>>>> end CodeP1.2 '''
```

In [ ]:

In [ ]:

In [ ]: