BT 5240

Assignment 3

Parameter Estimation

Parth Natekar

ED15B031

Methodology:

For estimation of parameters, two different optimization algorithms are used – a **genetic algorithm** and the **Nelder-Mead downhill simplex** algorithm.
Initially, some random values are chosen for all the 3 parameters to be estimated and these are fed to the optimization algorithm. The algorithm then attempts to find best estimates for these parameter by minimizing a cost function.
In this case, the cost function is the mean square error between the experimental data points given and the points obtained on solving the ODEs using the estimated parameter values.

While Genetic Algorithms are good global heuristic algorithms, the success of the Nelder-Mead algorithm depends largely on the initial estimates, and thus, it is a good local search method. It then makes sense to **combine these two algorithms to get the best estimates** of the parameters. First, a Genetic Algorithm is used to get estimates in the region of the global minimum, and then the Nelder Mead Algorithm is used to get even closer to the absolute minima.
To prove that a combination of both algorithms works better that each algorithm used alone, a comparison is made on the outputs of all the three cases.

Data preparation:

The experimental data has NaN values. To remove these, either the moiety conservation relations are used or the NaN value is replaced by the average concentration of that particular species.

Example code for this:

```
for i in range(len(MKKK)):
   if math.isnan(MKKK[i])==True and math.isnan(MKKK_P[i])==False:
       MKKK[i]=100 - MKKK_P[i]
   if math.isnan(MKKK_P[i])==True and math.isnan(MKKK[i])==False:
       MKKK_P[i]=100 - MKKK[i]
   elif math.isnan(MKKK[i])==True and math.isnan(MKKK_P[i])==True:
       MKKK[i] = np.around(np.nanmean(MKKK), decimals=2)
       MKKK_P[i] = np.around(np.nanmean(MKKK_P), decimals=2)
```

Solving ODEs:

Scipy's odeint is used to solve the ODEs.[1]

```
def solveODES(p1,p2,p3):
   def f(y, t, params):
       x1, x2, x3, x4, x5, x6, x7, x8 = y      # unpack current values of y
       V1, K1, p3, p1, p2, K2, K3, K4, K5, K6, K7, K8, K9, K10, KI, V9, V10
= params  # unpack parameters

       r1 = V1*x1/(((1+x8/KI)**2)*(K1+x1))
       r2 = p3*x2/(K2+x2)
       r3 = p1*x2*x3/(K3+x3)
       r4 = p1*x2*x4/(K4+x4)
       r5 = p2*x5/(K5+x5)
       r6 = p2*x4/(K6+x4)
       r7 = p1*x5*x6/(K7+x6)
       r8 = p1*x5*x7/(K8+x7)
       r9 = V9*x8/(K9+x8)
       r10 = V10*x7/(K10+x7)
```

[1] "Scipy.integrate.odeint¶." *Scipy.integrate.odeint - SciPy v1.0.0 Reference Guide*, docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

```
        derivs = [r2-r1, r1-r2, r6-r3, r3+r5-r4-r6, r4-r5, r10-r7, r7+r9-r8-r10,
r8-r9]
        return derivs

    # Parameters
    KI=18
    V1=2.5
    V9=1.25
    V10=1.25
    K1=50
    K2=40
    K3=K4=K5=K6=K7=K8=K9=K10=100
    #p1=0.04
    #p2=2
    #p3=1

    # Initial values
    x10=90
    x20=10
    x30=280
    x40=10
    x50=10
    x60=280
    x70=10
    x80=10

    # Bundle parameters for ODE solver
    params = [V1, K1, p3, p1, p2, K2, K3, K4, K5, K6, K7, K8, K9, K10, KI, V9,
V10]

    # Bundle initial conditions for ODE solver
    y0 = [x10, x20, x30, x40, x50, x60, x70, x80]

    # Make time array for solution
    #tStop = 8000.
    #tInc = 0.01
    #t = np.arange(0., tStop, tInc)
    t = MAPK_cascade[:, 0]

    # Call the ODE solver
    psoln = odeint(f, y0, t, args=(params,))
    print(p1,p2,p3)
    return psoln
```

Cost function:

The cost function is the sum of the mean squared error of each species, i.e.

$$L = ([MKKK]_{predicted} - [MKKK]_{experimental})^2 + ([MKKK\text{-}P]_{predicted} - [MKKK\text{-}P]_{experimental})^2 + \ldots\ldots\ldots\ldots + ([MAPK\text{-}PP]_{predicted} - [MAPK\text{-}PP]_{experimental})^2$$

Method 1: Genetic Algorithm

The pyeasyga library is used to create a genetic algorithm.
The code is as follows:

```
def GA():

   def create_individual(data):

       random_list=np.around(np.arange(0, 100, 0.01), decimals=2)
       random_list_2 = np.around(np.arange(0, 10, 0.05), decimals=2)
       individual = [random.choice(random_list_2),
random.choice(random_list_2), random.choice(random_list_2)]
       return individual
   seed_data=[0.00, 0.00, 0.00]
   ga = pyeasyga.GeneticAlgorithm(seed_data,
                                  population_size=5000,
                                  generations=5,
                                  crossover_probability=0.8,
                                  mutation_probability=0.05,
                                  elitism=True,
                                  maximise_fitness=False)  # initialise the GA
with data

   ga.create_individual = create_individual
   def fitness(individual, data):
       fitness_val = findlsq(individual)
       return fitness_val

   def findlsq(individual):
       individual=list(individual)

       ODEsoln = solveODES(individual[0], individual[1], individual[2])
       error_sum = mean_squared_error(MKKK, ODEsoln[:, 0]) +
mean_squared_error(MKKK_P,

ODEsoln[:, 1]) + mean_squared_error(
           MKK, ODEsoln[:, 2]) + mean_squared_error(MKK_P, ODEsoln[:, 3]) +
mean_squared_error(MKK_PP, ODEsoln[:,

4]) + mean_squared_error(
           MAPK, ODEsoln[:, 5]) + mean_squared_error(MAPK_P, ODEsoln[:, 6]) +
mean_squared_error(MAPK_PP,
```

```
ODEsoln[:, 7])
        return error_sum

    ga.fitness_function = fitness  # set the GA's fitness function
    ga.run()  # run the GA
    print(ga.best_individual())  # print the GA's best solution
    #print(len(ga.last_generation()))
    return(ga.best_individual())
```

A population of 1000 is created by randomly sampling parameter values from the range [0,10], with a resolution of 0.05. The crossover probability is 0.8 and mutation probability is 0.05. Elite individuals from the previous generation are directly allowed into the next generation unchanged.
The algorithm is allowed to run for 5 or 10 generations and the fittest individual is found.

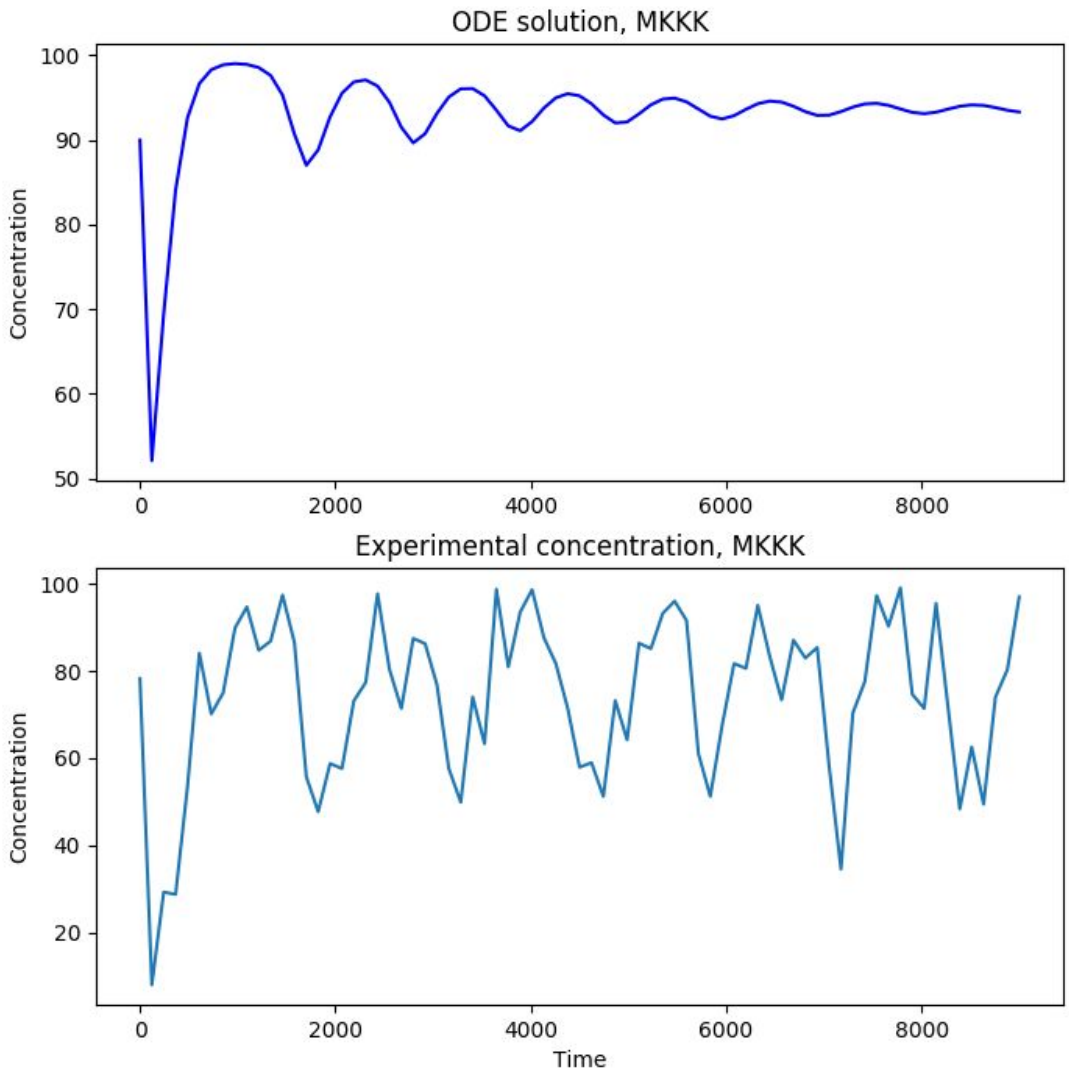
Since the output of a Genetic Algorithm is not reproducible, different solutions might be obtained on each run, but they are usually not very different from each other as long as the initial population size is large enough.
The parameters estimated by the genetic algorithm in one such run was(Dataset 2, Parameter Set 2)

p1=0.05, p2=0.6, p3=0.35

The total least square error for these values is

Least Square Error = 18504.63

The output graph of conc[MKKK] after solving the ODE using the above values of the three parameters is shown, as well as the experimental data(dataset2) is shown:



conc[MKKK] graphs for GA only (Parameter set 2, Dataset 2)

Method 2: Nelder-Mead Algorithm

The Nelder-Mead Algorithm is a direct-search method.[2] It is simplex based. A simplex in R^n is defined as a convex hull of n+1 vertices. In R^2, this is a triangle, while in R^3, this is a tetrahedron.

The algorithm works by calculation of the cost function values at the n+1 points. The points are ranked from best to worst depending on the function value. The centroid of the best side, i.e. the one opposite to the worst vertex is determined.
Finally, the new simplex is calculated, by transforming the worst vertex(reflecting it about the centroid) or by shrinking the simplex towards the best vertex.[3]

The code is as follows:

```
def findlsq(params):
   param1, param2, param3 = params
   ODEsoln = solveODES(param1, param2, param3)

   error_sum = mean_squared_error(MKKK, ODEsoln[:,0]) +
mean_squared_error(MKKK_P, ODEsoln[:,1]) + mean_squared_error(MKK,
ODEsoln[:,2]) + mean_squared_error(MKK_P, ODEsoln[:,3]) +
mean_squared_error(MKK_PP, ODEsoln[:,4]) + mean_squared_error(MAPK,
ODEsoln[:,5]) + mean_squared_error(MAPK_P, ODEsoln[:,6]) +
mean_squared_error(MAPK_PP, ODEsoln[:,7])
   return error_sum
initial2 = [9, 9.5, 10]
ret_random = optimize.minimize(findlsq, initial2, method='Nelder-Mead',
options={'maxiter':500})
```

The Nelder-Mead algorithm **might get stuck in local minima**, and thus, the initial estimates of the parameters are important. To show this, the algorithm is run with initial parameter estimates of [9, 9.5, 10].
After 500 iterations, the algorithm gives the following values for the parameters(Dataset 2, Parameter Set 2):

p1=8.72, p2=10.15, p3=9.54

The corresponding error is extremely large, and python only shows this value as ‘inf’

---

[2] Nelder, J. A., and R. Mead. "A Simplex Method for Function Minimization." *The Computer Journal* 7, no. 4 (01, 1965): 308-13. doi:10.1093/comjnl/7.4.308
[3] Nelder, John. "Nelder-Mead Algorithm." Scholarpedia. http://www.scholarpedia.org/article/Nelder-Mead_algorithm

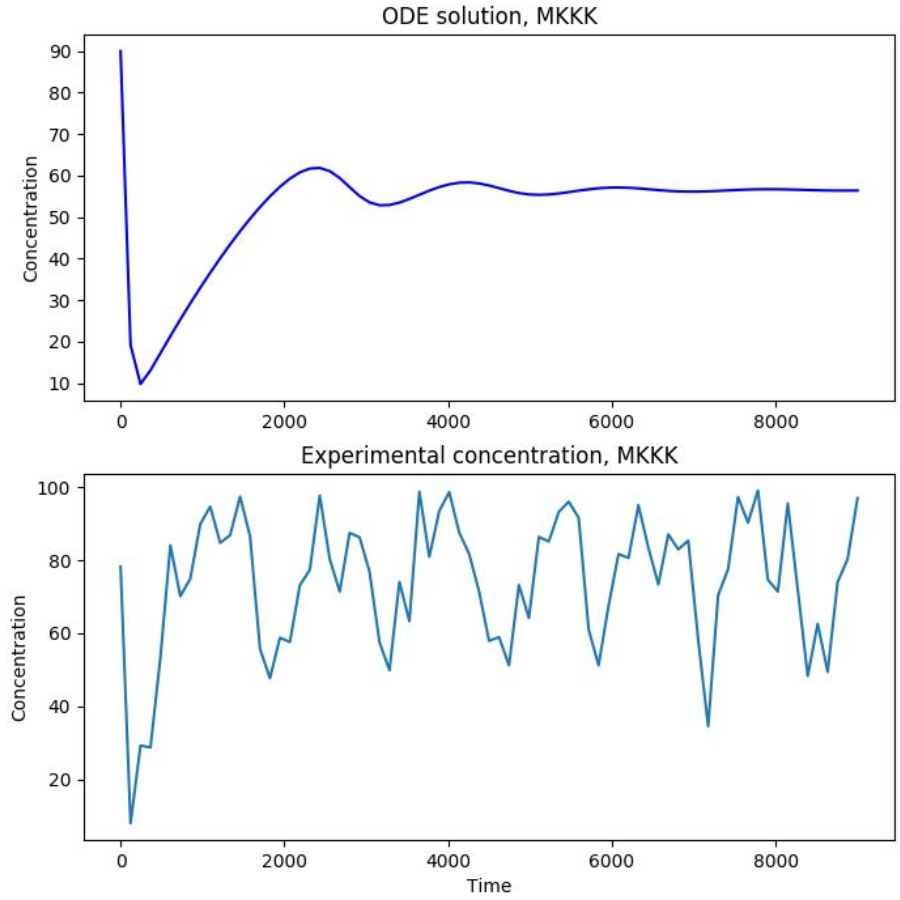
Since, the search space is to be constrained to [0,10], decent initial estimates might be values around 5. Values of **[4.5, 5, 5.5]** for the 3 parameters are chosen as standard random initializing values for all parameter sets and datasets in this method.
The corresponding estimated parameter values(Parameter set 2, Dataset 2) after 500 iterations are:

p1=0.030, p2=1.912, p3=0.055

And the corresponding error is

Least Square Error = 15188.25

This error is lesser than the Genetic Algorithm, but is still significant. As shown above, this is highly dependant on choosing good initial estimates.
The graph for conc[MKKK] for the estimated parameters is shown below:



conc[MKKK] graphs for Nelder-Mead only (Parameter set 2, Dataset 2)

Method 3: Genetic Algorithm + Nelder-Mead Algorithm:

As discussed above, a combination of the two algorithms is potentially the best solution to this problem. First, the Genetic Algorithm brings the parameter estimates in the vicinity of the global minima. Then the Nelder-Mead Algorithm, with good initial estimates in the locality around the global minima from the Genetic Algorithm, can get even better estimates.

The **hyperparameters of the two algorithms are kept exactly the same** as the above methods. This means the population size, no. of generations, crossover probability for the Genetic Algorithm, and number of iterations for Nelder Mead are not changed from Method 1 and Method 2.

The results are as follows(Dataset 2, Parameter Set 2):

Estimated Parameter Values:

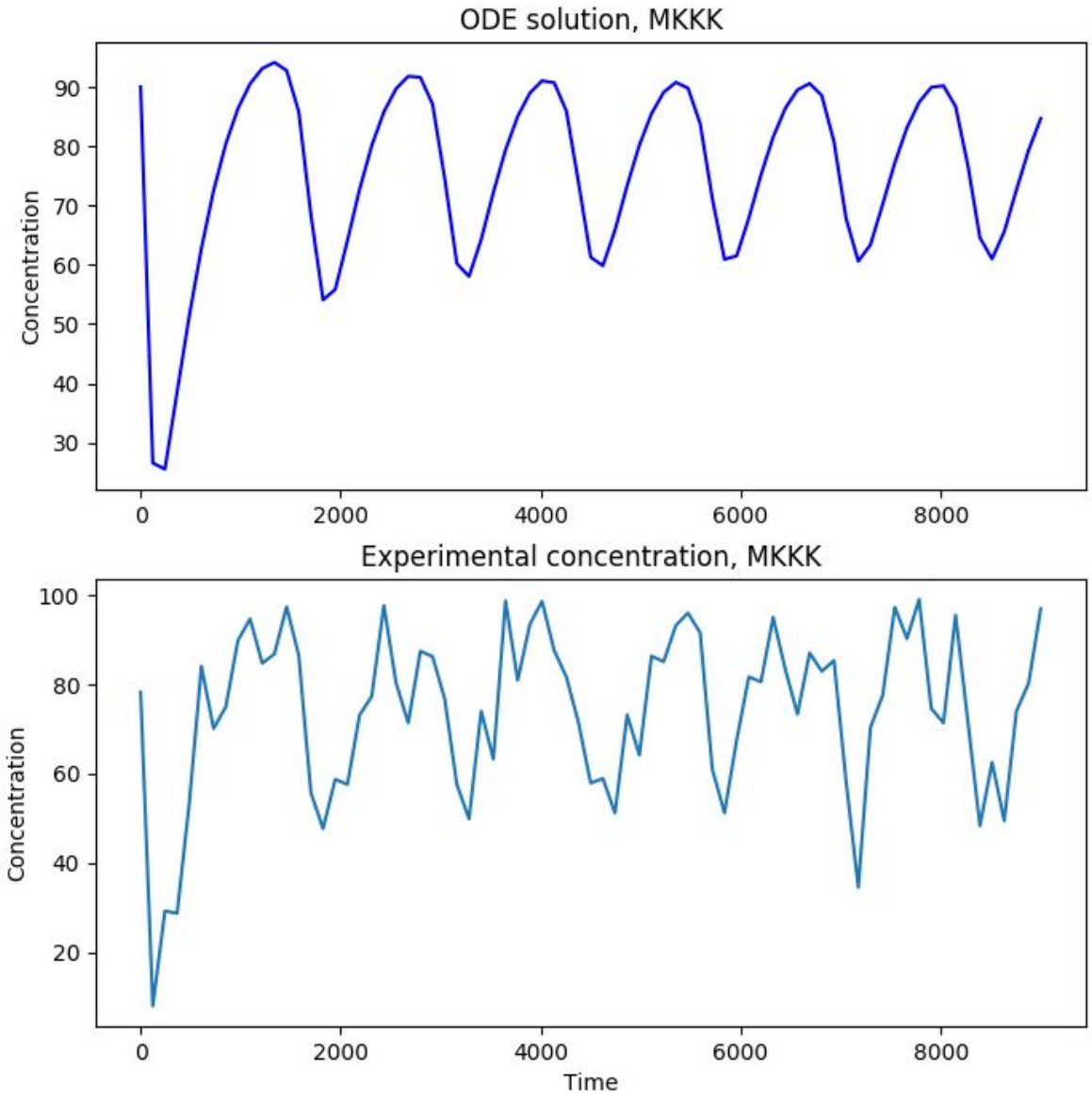
p1=0.029, p2=0.974, p3=0.192

Least Square Error:

Least Square Error = 3242.02

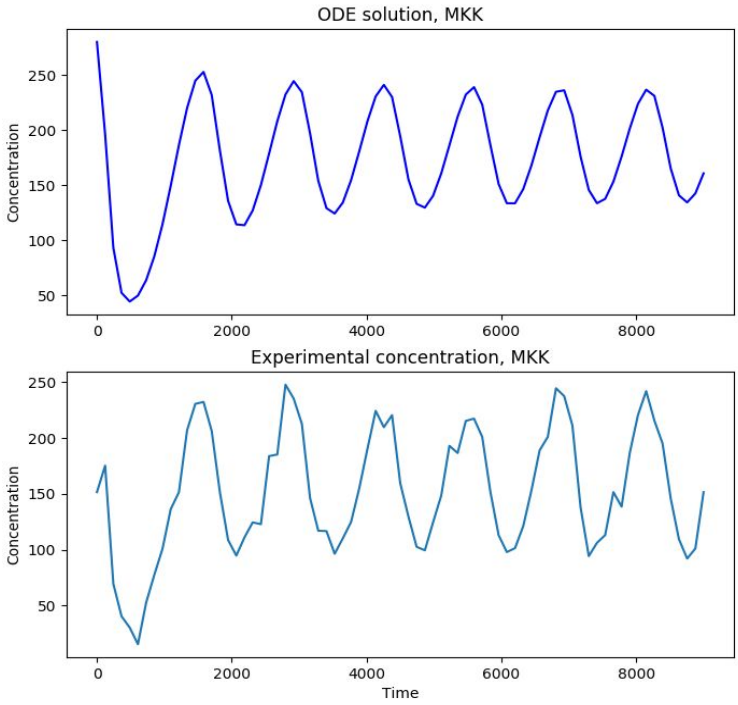
The error is **significantly lesser** than both Method 1 and Method 2. This is reflected in the predicted concentration graphs, which match the experimental graphs quite well(below). Obviously, using a combination of GA and NM is better than using each algorithm independently. **This improvement will be more and more significant as the search space increases**. For a large search space, neither algorithm independently will give a satisfactory result. For example, if the upper and lower bounds for the parameters were 0 and 1000, the Genetic Algorithm would need a very large population size to reach reasonably close to the absolute minima, and Nelder-Mead would almost surely get stuck in a local minima, but getting a vague estimate using the GA and then using Nelder-Mead would be faster and give much better results.

The graph for conc[MKKK] for the estimated parameters is shown below:

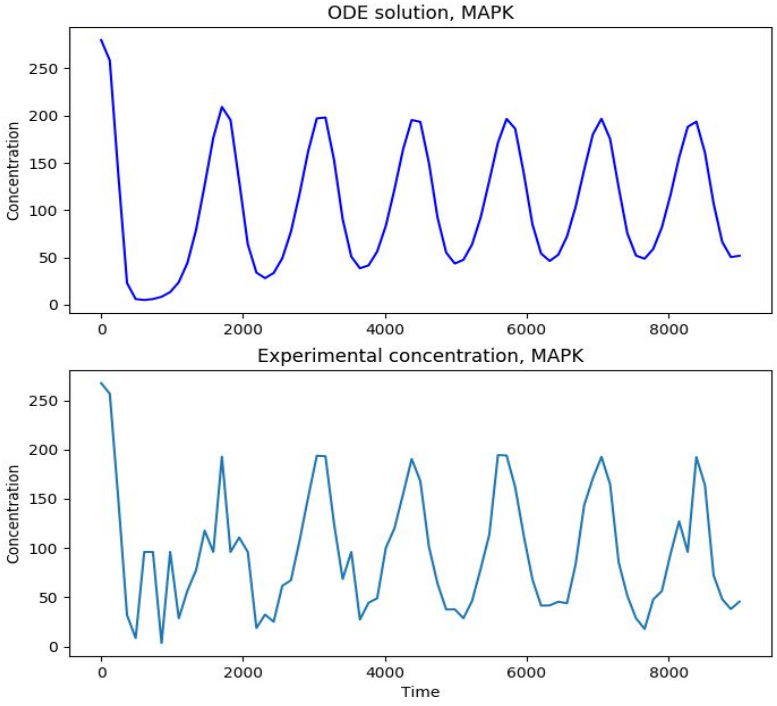


conc[MKKK] graphs for GA + Nelder Mead (Parameter set 2, Dataset 2)

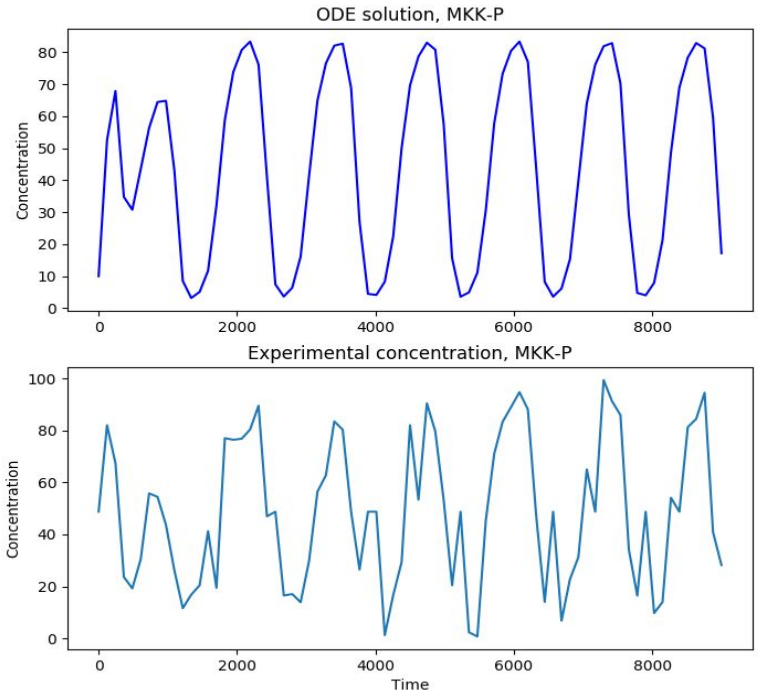
Concentration graphs of some other species:



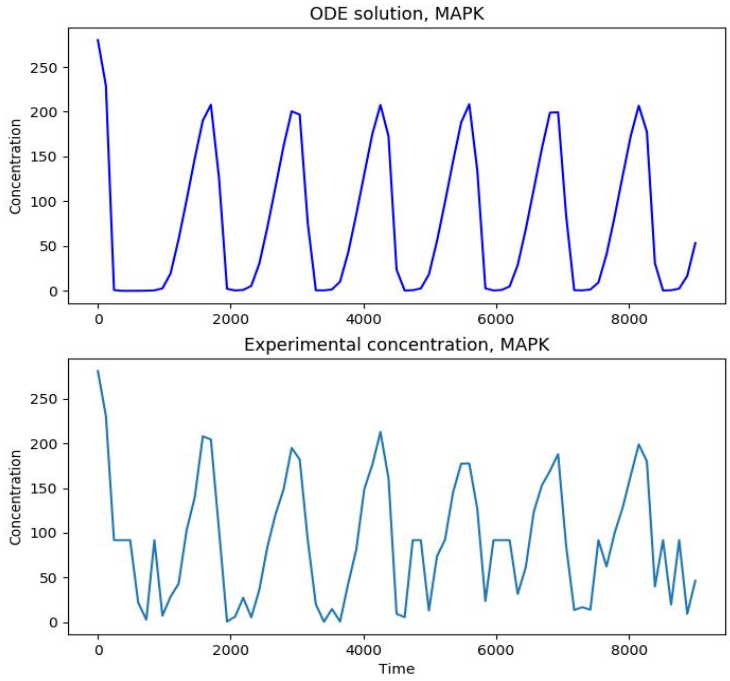
conc[MKK] graphs for GA + Nelder Mead (Parameter set 2, Dataset 2)



conc[MAPK] graphs for GA + Nelder Mead (Parameter set 2, Dataset 2)



conc[MKK-P] graphs for GA + Nelder Mead (Parameter set 1, Dataset 1)



conc[MAPK] graphs for GA + Nelder Mead (Parameter set 1, Dataset 1)

Comparison of Algorithms:

The comparison of algorithms (Parameter set 2, Dataset 2) is shown:

Algorithm	Estimated Parameters(p1, p2, p3)	Mean Square Error
GA	0.05, 0.6, 0.35	18504.63
NM	0.03, 1.912, 0.055	15188.25
GA + NM	0.029, 0.974, 0.192	**3242.02**

From the table, it is obvious that using both algorithms together gives much better results than using each algorithm independently.

Summary of Datasets and Parameter Sets:

For Dataset 1:

Algorithm	Parameter Set 1		Parameter Set 2	
	Estimated Parameters	Error	Estimated Parameters	Error
GA	0.1, 2.55, 0.15	34980.67	0.25, 0.65, 0.5	33426.47
NM	4.44, 5.00, 5.72	51389.18	0.0498, 0.042, 1.492	28460.19
GA + NM	0.0238, 0.659, 0.278	8181.63	0.035, 1.068, 0.181	15746.27

For Dataset 2:

Algorithm	Parameter Set 1		Parameter Set 2	
	Estimated Parameters	Error	Estimated Parameters	Error
GA	0.05, 1.65, 0.3	41000.20	0.05, 0.6, 0.35	18504.63
NM	4.53, 4.44, 5.42	61544.05	0.03, 1.912, 0.055	15188.25
GA + NM	0.0104, 0.232, 0.750	9921.64	0.029, 0.974, 0.192	3242.02

Conclusion:

The tables reiterate the fact that system behaviour is highly dependant on the parameter values[4]
The lowest mean square error is obtained for parameter set 2 and dataset 2. This might be helped by the fact that dataset 2 has comparatively very less NaN values as compared to dataset 1.
Parameter Set 1 seems to be a better set for dataset 1 as compared to parameter set 2, but dataset 1 is quite incomplete, which amplifies the MSE and thus it is difficult to draw conclusions.
I have used moiety conservation relations to remove the NaN values wherever possible, but there are still some NaN values remaining in the data, and I have just replaced these by the average concentration of that species. A better method to replace these values or simply removing them might reduce the error, but overall the effect of this would not be very significant.

As shown above, using a Genetic Algorithm for global search and then the Nelder-Mead algorithm (using the values from the Genetic Algorithm as initial values) for local search is the optimum strategy to save computation as well as time and get better parameter estimates. The predicted graphs seem to match the experimental graphs quite well, especially for dataset 2 and

[4] https://home.iitm.ac.in/kraman/courses/2018-BT5240/reading/papers/Sobie2011Introduction.pdf

parameter set 2, using method 3.