

```
In [2]: 1 import numpy as np
        2 import matplotlib.pyplot as plt
        3 from scipy.stats import norm
        4 import pandas as pd
        5 from scipy.integrate import odeint
        6 import time
        7 import statistics
        8 import random
        9 import scipy.stats
       10
```

Opening predator-prey dataset

```
In [3]: 1 df = pd.read_csv('predator-prey-data.csv', index_col=False)
        2 df.head()
        3
```

```
Out[3]:
```

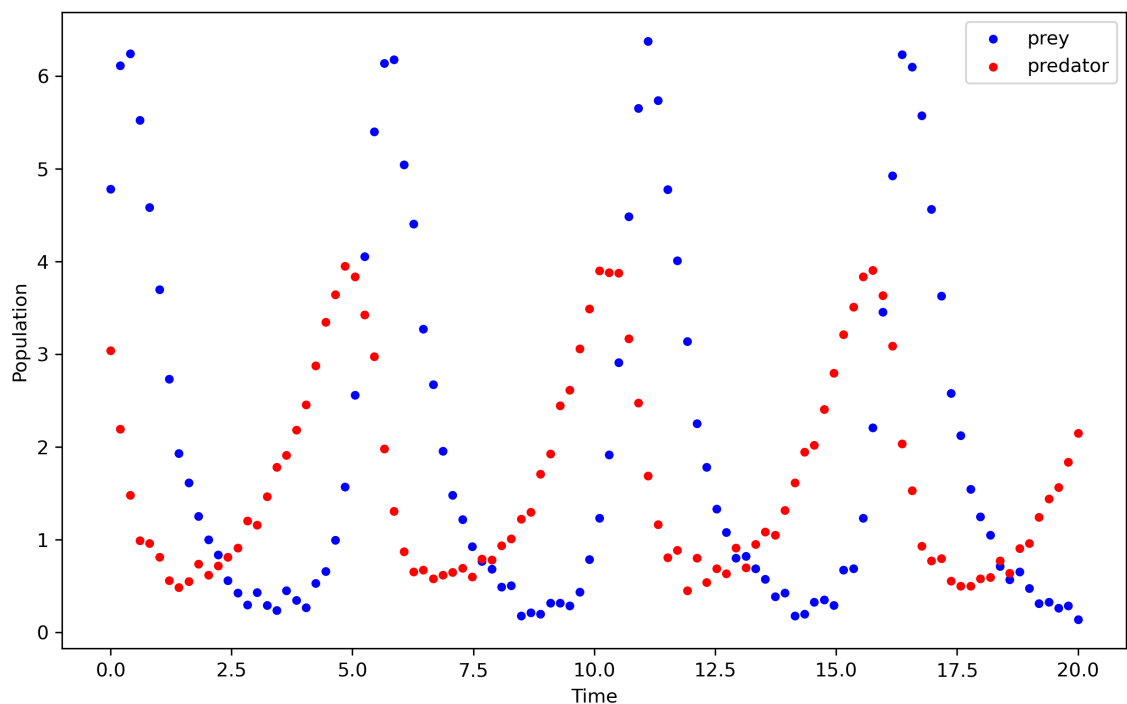
	Unnamed: 0	t	x	y
0	0	0.000000	4.781635	3.035257
1	1	0.202020	6.114005	2.189746
2	2	0.404040	6.238361	1.478907
3	3	0.606061	5.520524	0.989836
4	4	0.808081	4.582546	0.957827

```

In [4]: 1 # Loading data into read-only numpy arrays
2 data = df[['t', 'x', 'y']].values
3 # data[1], data[2] = data[2].copy(), data[1].copy()
4 data.flags.writeable = False
5
6
7 # Plotting
8 plt.figure(dpi = 300, figsize = (10, 6))
9 point_width = 13
10 # X should be prey
11 plt.scatter(data[:, 0], data[:, 1], label = 'prey', color = 'blue', s = pc
12 plt.scatter(data[:, 0], data[:, 2], label = 'predator', color = 'red', s =
13 plt.ylabel('Population')
14 plt.xlabel('Time')
15 plt.legend()
16

```

Out[4]: <matplotlib.legend.Legend at 0x14c151ad390>



Objective functions

Defining volterra equations function

```
In [5]: 1 def predator_prey_odes(initial_conditions,time ,alpha, beta, delta, gamma)
2       x = initial_conditions[0] # initial predator population
3       y = initial_conditions[1] # initial prey population
4       dxdt = (alpha * x) - (beta * x * y) # Predator ODE
5       dydt = (delta * x * y) - (gamma * y) # Predator ODE
6       return [dxdt, dydt]
7
8       #Function that will return the data for predator and prey for a given s
9       def predator_prey_integration(time,initial_conditions,parameters):
10          alpha,beta,delta,gamma = parameters
11          #odeint is now used as part of this function which returns the # of
12          results = odeint(predator_prey_odes,initial_conditions, time, args=
13          predator_values,prey_values = results[:,0], results[:,1]
14          return np.array([predator_values,prey_values]).T
15
```

```
In [ ]: 1
2
3
```

Defining objective functions

```
In [6]: 1 # modulo Linear error
2 def MSE(actual, predicted):
3     '''Mean squared error'''
4     return np.mean((actual - predicted)**2)
5
6 def MSE2(actual, predicted):
7     '''Mean squared error, handles nan values'''
8     x1, y1 = actual[:, 0], actual[:, 1]
9
10    # Getting useful indexes
11    indx_x = np.where(~np.isnan(x1))
12    indx_y = np.where(~np.isnan(y1))
13    x2, y2 = predicted[:, 0], predicted[:, 1]
14
15    err1 = (x1[indx_x] - x2[indx_x])**2
16    err2 = (y1[indx_y] - y2[indx_y])**2
17
18    # Concatenate the arrays before calculating the mean
19    errors = np.concatenate([err1, err2])
20
21    # Use np.nanmean to handle NaN values during the mean calculation
22    return np.nanmean(errors)
23
24
25 def MAE(actual, predicted):
26     '''Calculate Mean Absolute Error (MAE) for multidimensional data.'''
27     mae = np.mean(np.abs(actual - predicted))
28     return mae
29
30 def MAE2(actual, predicted):
31     '''Calculate Mean Absolute Error (MAE) for multidimensional data, h
32     x1, y1 = actual[:, 0], actual[:, 1]
33
34     # Getting useful indexes
35     indx_x = np.where(~np.isnan(x1))
36     indx_y = np.where(~np.isnan(y1))
37
38     x2, y2 = predicted[:, 0], predicted[:, 1]
39
40     err1 = np.abs(x1[indx_x] - x2[indx_x])
41     err2 = np.abs(y1[indx_y] - y2[indx_y])
42
43     mae = np.nanmean(np.concatenate([err1, err2]))
44
45     return mae
46
47
```

Algorithms & Optimisation

Defining minimization algorithms

In [7]:

```

1  def random_walk(parameters, variance = 0.5):
2      lst = [parameter + np.random.normal(0, 1) for parameter in parameters]
3      # Ensure all elements are positive
4      while any(x <= 0 for x in lst):
5          for indx in range(len(lst)):
6              if lst[indx] <= 0:
7                  while lst[indx] < 0:
8                      lst[indx] = parameters[indx] + np.random.normal(0,
9
10     return lst
11
12
13 def hill_climbing(data, time, initial_conditions, parameters, objective)
14     '''Tries to find the best solution using random walker'''
15     # Initialize starting parameter state
16     scores = []
17     x_n = parameters
18     all_scores = []
19
20     current_est = predator_prey_integration(time, initial_conditions, x
21     current_score = objective(data, current_est)
22     scores.append(current_score)
23     number_iterations = 1
24
25     for k in range(max_iterations):
26         # Generate a random walk for parameters
27         x_n_1 = random_walk(x_n, variance)
28
29         # Calculate the current and next estimations
30         current_est = predator_prey_integration(time, initial_conditions, x
31         new_estimation = predator_prey_integration(time, initial_conditions, x
32
33         new_score = objective(data, new_estimation)
34
35         # If the next estimation is better, update the parameters
36         if new_score < current_score:
37             number_iterations = k
38             current_score = new_score
39             x_n = x_n_1
40             scores.append(current_score)
41
42     return x_n, scores, number_iterations

```

```

In [8]: 1 def simulated_annealing(initial_temp,cooling_constant, data, time, init
2
3     temp = initial_temp #Scaling factor for random movement. We square
4     start = parameters #Initial starting parameters
5     x_n = start
6     scores = [] #A score is just the value of the objective function ev
7
8     current_est = predator_prey_integration(time, initial_conditions, >
9     current_score = objective(data, current_est) #The current value of
10    scores.append(current_score) #Keeping track of the values of the ob
11
12    #cur = function(x) #The function value of the current x solution
13    history = [x_n] #Stores previously searched x values
14
15    for i in range (max_iterations):
16        proposal = random_walk(x_n) #A new proposal for the parameters
17        new_est = predator_prey_integration(time, initial_conditions, p
18        new_score = objective(data, new_est) #Calculate new value of ob
19
20        delta = new_score - current_score #Difference in objective func
21
22        #if proposal < 0 or proposal > 1:
23            #proposal = x_n # Reject proposal by setting it equal to pre
24
25        acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate
26
27        #if delta < 0:
28            #x_n = proposal ##Accept proposal
29            #current_score = new_score
30
31        if np.random.rand() < acceptance_probability: #else if it is no
32            x_n = proposal #Accept proposal
33            current_score = new_score
34
35        scores.append(current_score)
36        temp = cooling_constant**i * initial_temp #Cool temperature
37        #print(temp)
38        history.append(x_n) #Add to history
39
40    return x_n, scores
41
42
43

```

Multiple run algorithms

```

In [9]: 1 def uniform_draw_g(lower_bound, upper_bound):
2         while True:
3             yield np.random.uniform(lower_bound, upper_bound)
4
5 def multiple_runs_annealing(initial_temp, cooling_constant, input_data, t,
6
7     mse_total_list = []
8     all_all_best = []
9
10    for i in range(n_runs):
11
12        x_best, scores = simulated_annealing(initial_temp, cooling_const
13        all_all_best.append(x_best)
14
15        x = predator_prey_integration(t, initial_conditions, x_best)
16        mse_prey = MSE(data[:,1], x[:,0])
17        mse_predator = MSE(data[:,2], x[:,1])
18        mse_total = mse_prey + mse_predator
19
20        mse_total_list.append(mse_total) #Add total MSE for this simul
21
22    return np.array(all_all_best) , mse_total_list
23
24
25
26 def multi_run_hill_climbing(data, objective, nruns = 50, nsamples=100,
27     initial_conditions = data[0][1:3]
28     time = data[:,0]
29
30     # Defining generators for variables
31     alpha = uniform_draw_g(0,1)
32     beta = uniform_draw_g(0,1)
33     delta = uniform_draw_g(0,1)
34     gamma = uniform_draw_g(0,1)
35
36     # Lists for storing values
37     parameter_list = []
38     best = []
39     best_score = float('inf')
40     best_param = None
41     num_iterations = []
42
43     # Running simulation for
44     for __ in range(nruns):
45
46         parameters = [next(alpha), next(beta), next(delta), next(gamma)]
47         params, score, iterations = hill_climbing(data[:,1:3], time, ir
48         parameter_list.append(params)
49         num_iterations.append(iterations)
50         scores.append(score)
51
52         #Saving best parameter combination
53         if score[-1] < best_score:
54             best_score = score[-1]
55             best_param = params
56
57     parameter_list = np.array(parameter_list)
58
59     return parameter_list, best_param, scores, best_score, num_iteratic

```


60

Plotting hill climbing

In [10]:

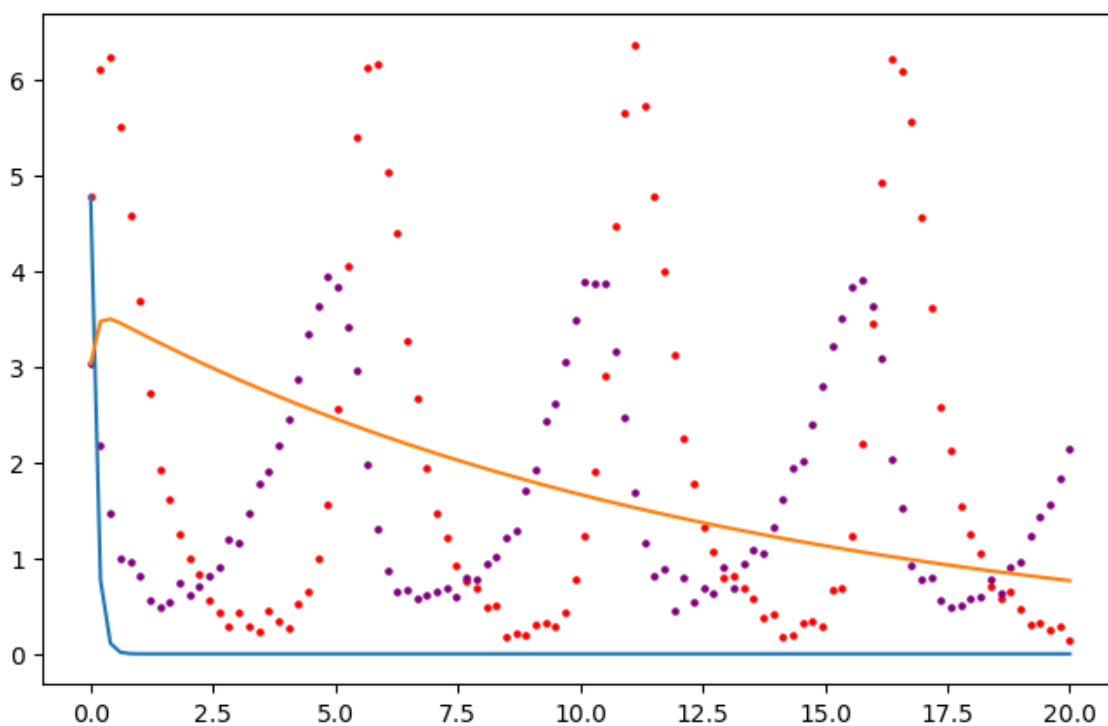
```
1 input_data = data[:,1:3]
2 # t =
3 initial_conditions = [input_data[0][0], input_data[0][1]]
4
5 alpha = np.random.uniform(0.5, 2)
6 beta = np.random.uniform(0.5, 2)
7 delta = np.random.uniform(0.5, 2)
8 gamma = np.random.uniform(0.5, 2)
9 parameters = [alpha, beta, delta, gamma]
10
11 # Using MSeX
12 x_best, scores, num_iterations = hill_climbing(input_data, data[:,0], i
13 print(x_best,scores,num_iterations)
```

```
[2.058470826384955, 3.3348226913782497, 0.33391386324627437, 0.07775331273
010042] [1.7841476511911036, 1.7630633552849264, 1.7463166480090968, 1.741
6628363579707, 1.7024628624472102, 1.6549539797197672, 1.541264871748595]
16
```

Plotting hill climbing results

```
In [11]: 1 # t, x ,y = data
2 initial_conditions = [input_data[0][0], input_data[0][1]]
3 t = data[:,0]
4
5 parameters = x_best
6 # Using MSE
7 x = predator_prey_integration(t,initial_conditions,parameters)
8
9 # Increase the figure size
10 plt.figure(figsize=(8, 5))
11
12 plt.plot(t, x[:,0])
13 plt.plot(t, x[:,1])
14
15 plt.scatter(t, data[:,1], color= 'red', s =5)
16 plt.scatter(t, data[:,2], color= 'purple', s=5)
17
18 plt.figure(figsize=(10, 8))
```

Out[11]: <Figure size 1000x800 with 0 Axes>



<Figure size 1000x800 with 0 Axes>

Running simulation for different random walker variance

```
In [ ]: 1 # parameter_list = np.array(parameter_list)
2 # # Create a figure with 3x3 subplotshttp://localhost:8888/notebooks/De
3 # fig, axes = plt.subplots(3, 3, figsize=(12, 12), sharex=True)
4
5 # # Plot histograms on each subplot using for loops with the same color
6 # color = 'blue'
7 # titles = ['variance = 0.1', 'variance=0.25', 'variance=0.5']
8 # x_titles = ['alpha', 'beta', 'delta', 'omega']
9
10 # for i in range(3):
11 #     for j in range(3):
12 #         ax = axes[i, j]
13 #         ax.hist(parameter_list[j][:,i])
14 #         ax.set_xlabel(x_titles[i])
15 #         ax.set_ylabel('Frequency')
16 #         ax.set_title(titles[j])
17
18 # # Adjust layout to prevent overlapping
19 # plt.tight_layout()
20
21 # # Show the plot
22 # plt.show()
23
```

Running multi run for hill climbing

In [12]:

```
1
2 # We save the parameter estimation we will use as ground truth for test
3 parameter_list, reference_param, scores, reference_score, num_iteration
4
5 # Integrating with best guess
6 results = predator_prey_integration(t, initial_conditions, reference_param
7
8 # Increase the figure size
9 plt.figure(figsize=(8, 5))
10
11 plt.plot(data[:,0], results[:,0])
12 plt.plot(data[:,0], results[:,1])
13
14 plt.scatter(t, data[:,1], color= 'red', s =5)
15 plt.scatter(t, data[:,2], color= 'purple', s=5)
16
17 plt.show()
18
```

C:\Users\Aleks\AppData\Local\anaconda3\Lib\site-packages\scipy\integrate\odepack_py.py:248: ODEintWarning: Excess work done on this call (perhaps wrong Dfun type). Run with full_output = 1 to get quantitative information.
warnings.warn(warning_msg, ODEintWarning)

-
KeyboardInterrupt

Traceback (most recent call last)

Cell **In[12], line 2**

```
1 # We save the parameter estimation we will use as ground truth for
testing
----> 2 parameter_list, reference_param, scores, reference_score, num_ite
rations = multi_run_hill_climbing(data, MSE, nruns = 200)
4 # Integrating with best guess
5 results = predator_prey_integration(t, initial_conditions, reference
_param)
```

Cell **In[9], line 47**, in **multi_run_hill_climbing(data, objective, nruns, nsamples, variance)**

```
44 for __ in range(nruns):
46     parameters = [next(alpha), next(beta), next(delta), next(gamma)]
---> 47     params, score, iterations = hill_climbing(data[:,1:3], time, i
nitial_conditions, parameters, objective, max_iterations=nsamples, variance
=variance)
48     parameter_list.append(params)
49     num_iterations.append(iterations)
```

Cell **In[7], line 30**, in **hill_climbing(data, time, initial_conditions, parameters, objective, max_iterations, variance)**

```
28 # Calculate the current and next estimations
29 current_est = predator_prey_integration(time, initial_conditions,
x_n)
---> 30 new_estimation = predator_prey_integration(time, initial_conditions,
s, x_n_1)
32 new_score = objective(data, new_estimation)
34 # If the next estimation is better, update the parameters
```

Cell **In[5], line 12**, in **predator_prey_integration(time, initial_conditions, parameters)**

```
10 alpha,beta,delta,gamma = parameters
11 #odeint is now used as part of this function which returns the # o
f infected in the model
---> 12 results = odeint(predator_prey_odes,initial_conditions, time, args
=(alpha,beta,delta,gamma))
13 predator_values,prey_values = results[:,0], results[:,1]
14 return np.array([predator_values,prey_values]).T
```

File **~\AppData\Local\anaconda3\Lib\site-packages\scipy\integrate\odepack_py.py:242**, in **odeint(func, y0, t, args, Dfun, col_deriv, full_output, ml, mu, rtol, atol, tcrit, h0, hmax, hmin, ixpr, mxstep, mxhnil, mxordn, mxords, printmessg, tfirst)**

```
240 t = copy(t)
241 y0 = copy(y0)
--> 242 output = _odepack.odeint(func, y0, t, args, Dfun, col_deriv, ml, m
u,
243                               full_output, rtol, atol, tcrit, h0, hmax,
hmin,
244                               ixpr, mxstep, mxhnil, mxordn, mxords,
245                               int(bool(tfirst)))
246 if output[-1] < 0:
247     warning_msg = _msgs[output[-1]] + " Run with full_output = 1 t
o get quantitative information."
```

KeyboardInterrupt:

Kaya's code section: Points removal

```
In [13]: 1 def point_removal(time, input_data, points_removed, Focus = 'both'):
2         '''removes points randomly'''
3
4         #We set the seed for removing points
5         random.seed(123)
6
7         prey = input_data.T[0].copy()
8         predator = input_data.T[1].copy()
9
10        # initialize set up for removing points randomly given the bounds
11        removal_options = np.arange(0, len(time))
12
13        # choose points to be removed randomly
14        if points_removed > len(removal_options):
15            points_removed = len(removal_options)
16            print('WARNING: Maximum number of points that can be removed has been reached')
17        removed_points_indices = random.choices(removal_options, k = points_removed)
18
19        # remove points based on choices for points to be removed
20        if Focus == 'both':
21            for i in removed_points_indices:
22                prey[i] = None
23                predator[i] = None
24
25        elif Focus == 'prey':
26            for i in removed_points_indices:
27                prey[i] = None
28
29        elif Focus == 'predator':
30            for i in removed_points_indices:
31                predator[i] = None
32
33        return np.array([time, prey, predator]).T
34
35
```



```

In [14]: 1 def extrema_removal(time, input_data, points_removed, Focus = 'both'):
2         '''Removes points in extrema'''
3         prey = input_data.T[0].copy()
4         predator = input_data.T[1].copy()
5
6         # Calculate mean and variance to set regions for data
7         mean_prey_population, mean_predator_population = np.mean(prey), np.
8         variance_prey, variance_predator = statistics.variance(prey), stati
9
10        # set upper bound and lower bound for point removals
11        ub_prey, lb_prey = mean_prey_population + 1.645*variance_prey/len(t
12        ub_predator, lb_predator = mean_predator_population + 1.645*variance
13
14        # initialize set up for removing points randomly given the bounds
15        prey_options = []
16        predator_options = []
17        # enumerate through list of stored points
18        for index, prey_count in enumerate(prey):
19            # check if they are in specified region
20            if prey_count > ub_prey or prey_count < lb_prey:
21                prey_options.append([index, prey_count, predator[index]])
22        for index, predator_count in enumerate(predator):
23            if predator_count > ub_predator or predator_count < lb_predator
24                predator_options.append([index, prey[index], predator_count
25
26        # remove points from list depending on which focus is set
27        removal_options = []
28        if Focus == 'both':
29            removal_options = removal_options + prey_options + predator_opt
30        elif Focus == 'prey':
31            removal_options = removal_options + prey_options
32        elif Focus == 'predator':
33            removal_options = removal_options + predator_options
34        else:
35            print('Error: Removal option not known. Try either both, prey,
36
37        # choose points to be removed randomly
38        if points_removed > len(removal_options):
39            points_removed = len(removal_options)
40            print('WARNING: Maximum number of points that can be removed ha
41        removed_points_indices = random.choices(np.array(removal_options).T
42
43        # turn the list into integers so we can remove them based on the in
44        integer_array = []
45        for counter in range(len(removed_points_indices)):
46            integer_array.append(int(removed_points_indices[counter]))
47
48        # update the lists based on points we wanted to remove
49        if Focus == 'both':
50            for i in integer_array:
51                prey[i] = None
52                predator[i] = None
53
54        elif Focus == 'prey':
55            for i in integer_array:
56                prey[i] = None
57
58        elif Focus == 'predator':
59            for i in integer_array:
60                predator[i] = None
61

```



```
62     return np.array(time), np.array(pre), np.array(predator)
63
64 # extrema_removal(t, input_data, 5, Focus = 'both')
```



```

In [15]: 1 def midpoint_removal(time, input_data, points_removed, Focus = 'both'):
2         '''Removes points close to the mean'''
3         prey = input_data.T[0].copy()
4         predator = input_data.T[1].copy()
5
6         # Calculate mean and variance to set regions for data
7         mean_prey_population, mean_predator_population = np.mean(prey), np.
8         variance_prey, variance_predator = statistics.variance(prey), stati
9
10        # set upper bound and lower bound for point removals
11        ub_prey, lb_prey = mean_prey_population + 1.645*variance_prey/len(t
12        ub_predator, lb_predator = mean_predator_population + 1.645*variance
13
14        # initialize set up for removing points randomly given the bounds
15        prey_options = []
16        predator_options = []
17        # enumerate through list of stored points
18        for index, prey_count in enumerate(prey):
19            # check if they are in specified region
20            if prey_count <= ub_prey or prey_count >= lb_prey:
21                prey_options.append([index, prey_count, predator[index]])
22        for index, predator_count in enumerate(predator):
23            if predator_count <= ub_predator or predator_count >= lb_predat
24                predator_options.append([index, prey[index], predator_count
25
26        # remove points from list depending on which focus is set
27        removal_options = []
28        if Focus == 'both':
29            removal_options = removal_options + prey_options + predator_opt
30        elif Focus == 'prey':
31            removal_options = removal_options + prey_options
32        elif Focus == 'predator':
33            removal_options = removal_options + predator_options
34        else:
35            print('Error: Removal option not known. Try either both, prey,
36
37        # choose points to be removed randomly
38        if points_removed > len(removal_options):
39            points_removed = len(removal_options)
40            print('WARNING: Maximum number of points that can be removed ha
41        removed_points_indices = random.choices(np.array(removal_options).T
42
43        # turn the list into integers so we can remove them based on the in
44        integer_array = []
45        for counter in range(len(removed_points_indices)):
46            integer_array.append(int(removed_points_indices[counter]))
47
48        # update the lists based on points we wanted to remove
49        if Focus == 'both':
50            for i in integer_array:
51                prey[i] = None
52                predator[i] = None
53
54        elif Focus == 'prey':
55            for i in integer_array:
56                prey[i] = None
57
58        elif Focus == 'predator':
59            for i in integer_array:
60                predator[i] = None
61

```

```
62     return np.array([time, prey, predator]).T
63
```

In []:

```
1  # 1. Run multi run for different size datasets save best parameters
2  # 2. Calculate MSE for each run for best parameters
3  # 2. Do this for 2x, one only for predator, other for prey
4  # 4. Plot error relative to best solution of y axis
5  # 5. On x axis should be relative number points
6
```

Hypothesis testing random removal points (Aleks section)

Duplicating code for the functions I use in case they are different

```

In [ ]: 1 def random_walk(parameters, variance = 0.5):
2         lst = [parameter + np.random.normal(0, 1) for parameter in parameters]
3         # Ensure all elements are positive
4         while any(x <= 0 for x in lst):
5             for indx in range(len(lst)):
6                 if lst[indx] <= 0:
7                     while lst[indx] < 0:
8                         lst[indx] = parameters[indx] + np.random.normal(0,
9                     variance)
10        return lst
11
12 def hill_climbing(data, time, initial_conditions, parameters, objective):
13     '''Tries to find the best solution using random walker'''
14     # Initialize starting parameter state
15     scores = []
16     x_n = parameters
17     all_scores = []
18
19     current_est = predator_prey_integration(time, initial_conditions,
20     parameters)
21     current_score = objective(data, current_est)
22     scores.append(current_score)
23     number_iterations = 1
24
25     for k in range(max_iterations):
26         # Generate a random walk for parameters
27         x_n_1 = random_walk(x_n, variance)
28
29         # Calculate the current and next estimations
30         current_est = predator_prey_integration(time, initial_conditions,
31         x_n_1)
32         new_estimation = predator_prey_integration(time, initial_conditions,
33         x_n_1)
34
35         new_score = objective(data, new_estimation)
36
37         # If the next estimation is better, update the parameters
38         if new_score < current_score:
39             number_iterations = k
40             current_score = new_score
41             x_n = x_n_1
42             scores.append(current_score)
43
44     return x_n, scores, number_iterations
45
46 def simulated_annealing(initial_temp, cooling_constant, data, time, initial_conditions,
47 parameters, objective):
48     temp = initial_temp #Scaling factor for random movement. We square
49     start = parameters #Initial starting parameters
50     x_n = start
51     scores = [] #A score is just the value of the objective function evaluated at x
52
53     current_est = predator_prey_integration(time, initial_conditions,
54     parameters)
55     current_score = objective(data, current_est) #The current value of the objective function
56     scores.append(current_score) #Keeping track of the values of the objective function
57
58     #cur = function(x) #The function value of the current x solution
59     history = [x_n] #Stores previously searched x values
60
61     for i in range(max_iterations):
62         proposal = random_walk(x_n) #A new proposal for the parameters
63         new_est = predator_prey_integration(time, initial_conditions,
64         proposal)
65         new_score = objective(data, new_est) #Calculate new value of the objective function

```

```

62     delta = new_score - current_score #Difference in objective function
63
64     #if proposal < 0 or proposal > 1:
65         #proposal = x_n # Reject proposal by setting it equal to previous
66
67     acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability
68
69     #if delta < 0:
70         #x_n = proposal ##Accept proposal
71         #current_score = new_score
72
73     if np.random.rand() < acceptance_probability: #else if it is not accepted
74         x_n = proposal #Accept proposal
75         current_score = new_score
76
77     scores.append(current_score)
78     temp = cooling_constant**i * initial_temp #Cool temperature
79     #print(temp)
80     history.append(x_n) #Add to history
81
82     return x_n, scores
83
84
85 def uniform_draw_g(lower_bound, upper_bound):
86     while True:
87         yield np.random.uniform(lower_bound, upper_bound)
88
89 def multiple_runs_annealing(initial_temp,cooling_constant,input_data,t_max):
90
91     mse_total_list = []
92     all_all_best = []
93
94     for i in range(n_runs):
95
96         x_best, scores = simulated_annealing(initial_temp,cooling_constant,input_data,t_max)
97         all_all_best.append(x_best)
98
99         x = predator_prey_integration(t,initial_conditions,x_best)
100         mse_prey = MSE(data[:,1],x[:,0])
101         mse_predator = MSE(data[:,2],x[:,1])
102         mse_total = mse_prey + mse_predator
103
104         mse_total_list.append(mse_total) #Add total MSE for this simulation
105
106     return np.array(all_all_best) , mse_total_list
107
108
109
110 def multi_run_hill_climbing(data, objective, nruns = 50, nsamples=100,
111     initial_conditions = data[0][1:3]
112     time = data[:,0]
113
114     # Defining generators for variables
115     alpha = uniform_draw_g(0,1)
116     beta = uniform_draw_g(0,1)
117     delta = uniform_draw_g(0,1)
118     gamma = uniform_draw_g(0,1)
119
120     # Lists for storing values
121     parameter_list = []
122     best = []

```

```

123 best_score = float('inf')
124 best_param = None
125 num_iterations = []
126
127 # Running simulation for
128 for __ in range(nruns):
129
130     parameters = [next(alpha), next(beta), next(delta), next(gamma
131     params, score, iterations = hill_climbing(data[:,1:3], time, i
132     parameter_list.append(params)
133     num_iterations.append(iterations)
134     scores.append(score)
135
136     #Saving best parameter combination
137     if score[-1] < best_score:
138         best_score = score[-1]
139         best_param = params
140
141 parameter_list = np.array(parameter_list)
142
143 return parameter_list, best_param, scores, best_score, num_iterati
144
145
146 def point_removal(time, input_data, points_removed, Focus = 'both'):
147     '''removes points randomly'''
148
149     #We set the seed for removing points
150     random.seed(123)
151
152     prey = input_data.T[0].copy()
153     predator = input_data.T[1].copy()
154
155     # initialize set up for removing points randomly given the bounds
156     removal_options = np.arange(0, len(time))
157
158     # choose points to be removed randomly
159     if points_removed > len(removal_options):
160         points_removed = len(removal_options)
161         print('WARNING: Maximum number of points that can be removed ha
162     removed_points_indices = random.choices(removal_options, k = point
163
164     # remove points based on choices for points to be removed
165     if Focus == 'both':
166         for i in removed_points_indices:
167             prey[i] = None
168             predator[i] = None
169
170     elif Focus == 'prey':
171         for i in removed_points_indices:
172             prey[i] = None
173
174     elif Focus == 'predator':
175         for i in removed_points_indices:
176             predator[i] = None
177
178     return np.array([time, prey, predator]).T

```


Getting distribution of averages of best guesses for hill climbing (reference dataset)

```
In [22]: 1 # We get the reference distribution for testing
2
3 # Timing your code
4 start_time = time.time()
5
6 # Reference distribution of averages
7 ref_average1 = []
8 for k in range(50):
9     parameter_list, best_param, scores, best_score, num_iterations = mu
10     # Appending average
11     ref_average1.append(np.mean(parameter_list, axis=0))
12
13 ref_average1 = np.array(ref_average1)
14
15 end_time = time.time()
16
17 # Calculating and printing the total time
18 total_time = end_time - start_time
19 print(f"Total time taken: {total_time} seconds")
20
21 # param_distribution, reference_param, scores, reference_score, num_ite
```

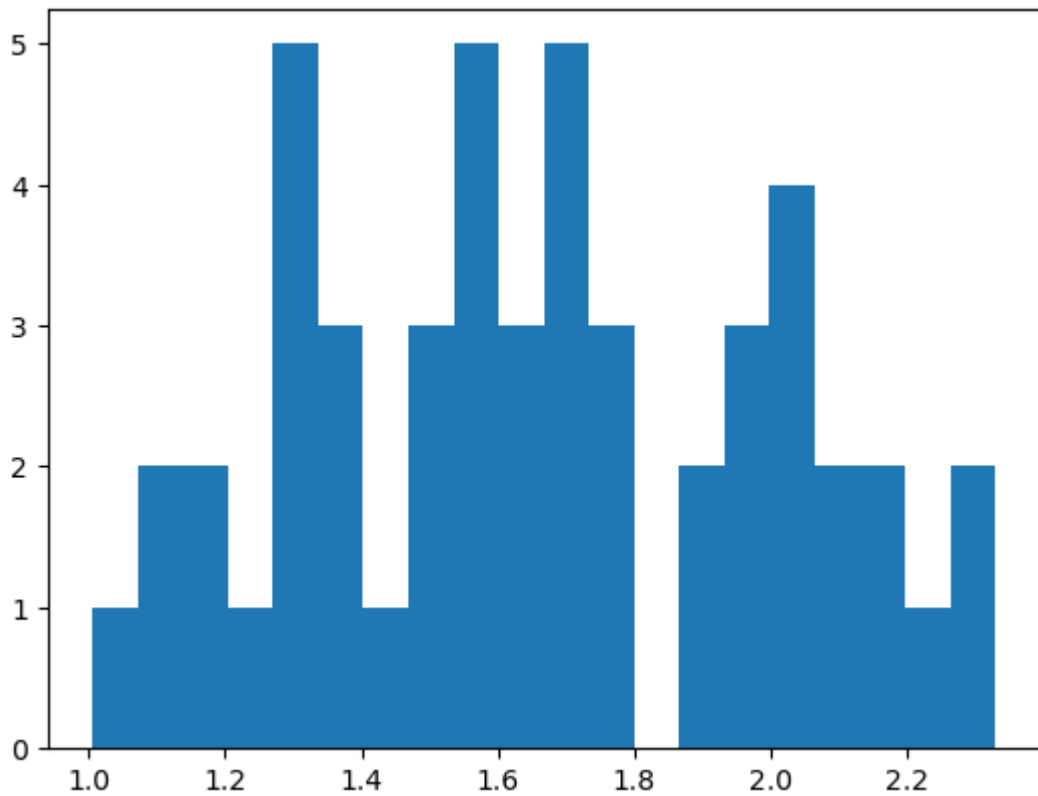
Total time taken: 439.93707609176636 seconds

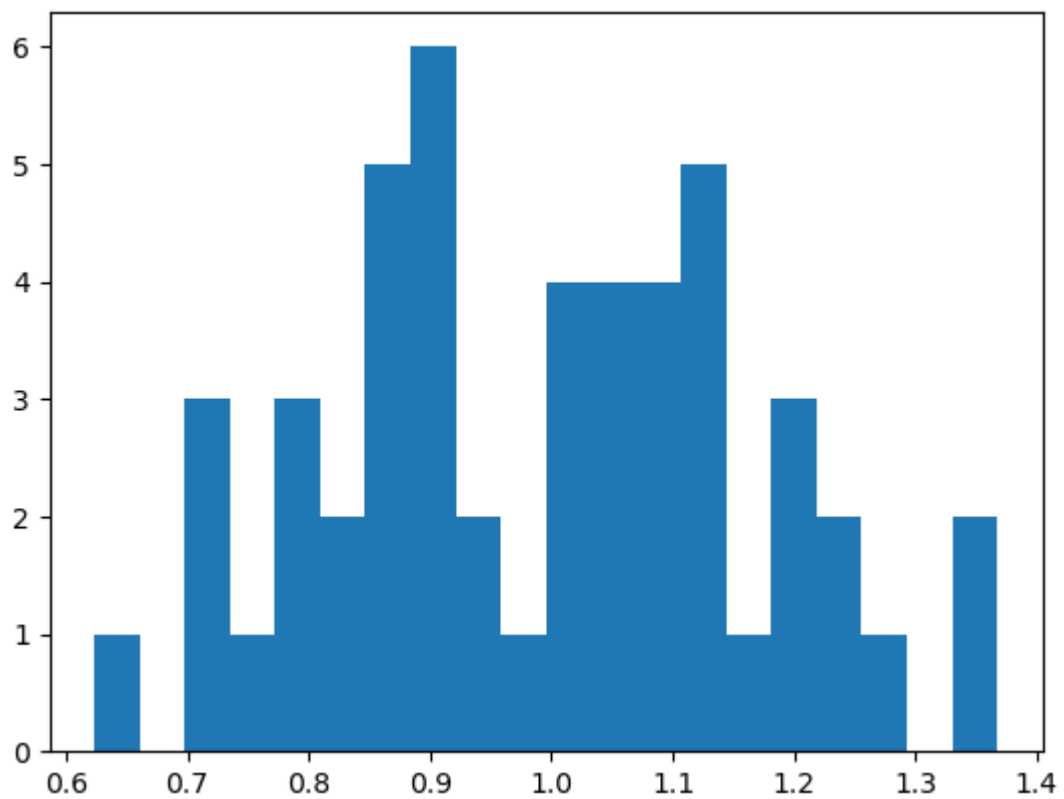
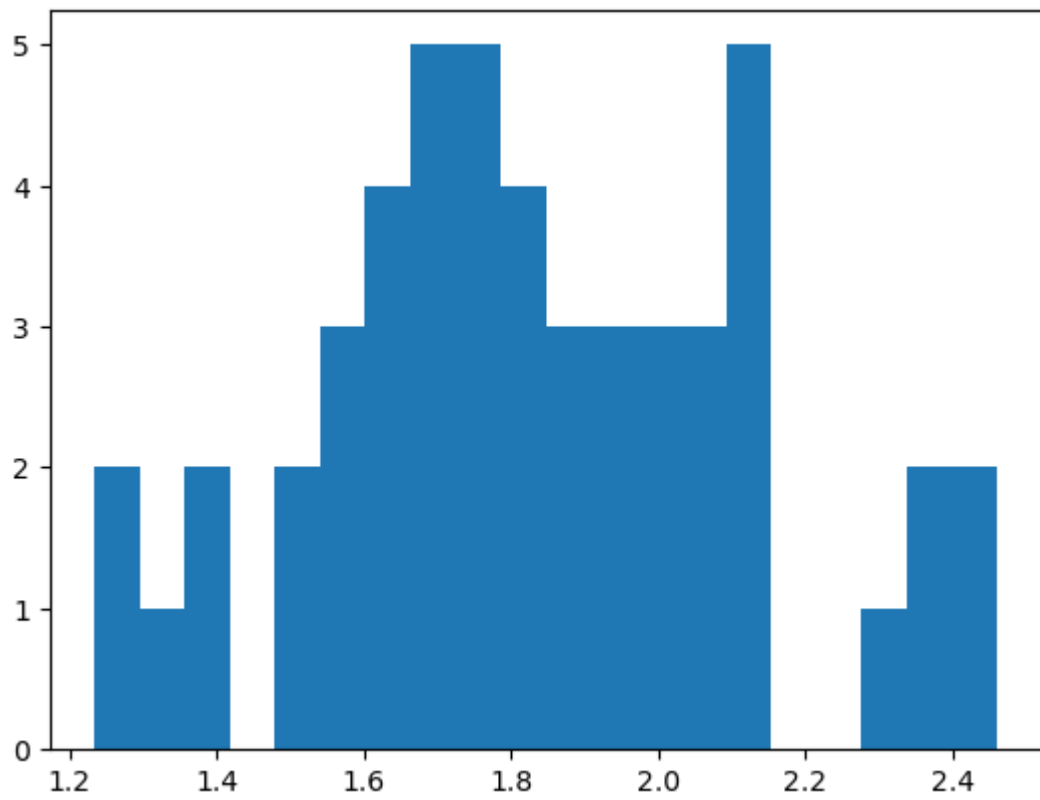
```
In [ ]: 1
```

These histograms plots are optional (I dont think im adding them to the report)

In [24]:

```
1 # num_iterations
2 # print(ref_average)
3 # for k in range(3):
4 #     plt.hist(ref_average1[:,k], bins = 20)
5 #     plt.show()
6
7 # print(np.var(ref_average1,axis=0))
```





[0.1170307 0.08449044 0.02953349 0.11417836]

```
In [25]: 1 # # print(param_distribution)
2 # plt.axvline(x=reference_param[0], color='r', linestyle='--', label='V
3 # plt.hist(param_distribution[:,0], bins =30)
```

```
-----
NameError                                Traceback (most recent call las
t)
Cell In[25], line 2
      1 # print(param_distribution)
----> 2 plt.axvline(x=reference_param[0], color='r', linestyle='--', label
      = 'Vertical Line at x=2.5')
      3 plt.hist(param_distribution[:,0], bins =30)

NameError: name 'reference_param' is not defined
```

Running welch test between reference distribution of averages and incomplete time series (hill climbing)

```
In [27]: 1 def random_point_ttests(data,ref_distribution, points_removed, focus_ch
2         '''runs welch test for multi run of hill climbing for every paramet
3         param_distribution = ref_distribution
4         # focus_choices = ['prey', 'predator', 'both']
5         # focus_choices = ['prey', 'predator', 'both']
6         scores = [],[]
7         p_values = {'prey': [], 'predator': [], 'both': []}
8
9         for indx, choice in enumerate(focus_choices):
10            print(choice)
11            for npoints in points_removed:
12                print(npoints)
13                # print(f"Points removed: {npoints}")
14                limited_data = point_removal(data[:,0], data[:,1:3], npoint
15                #Getting distribution of averages
16                average_distribution = []
17                for k in range(30):
18                    parameter_list, best_param, scores, best_score, num_ite
19                    #Appending average
20                    average_distribution.append(np.mean(parameter_list, axi
21
22                average_distribution = np.array(average_distribution)
23                t_stat1, p_value1 = scipy.stats.ttest_ind(ref_distribution[
24                t_stat2, p_value2 = scipy.stats.ttest_ind(ref_distribution[
25                t_stat3, p_value3 = scipy.stats.ttest_ind(ref_distribution[
26                t_stat3, p_value4 = scipy.stats.ttest_ind(ref_distribution[
27                p_values[choice].append([p_value1, p_value2, p_value3, p_v
28
29            return p_values
30
31
```

```
In [28]: 1 start_time = time.time()
2 p_values_hill_climbing = random_point_ttests(data, ref_average1, np.arange(
3
4 end_time = time.time()
5
6 # Calculating and printing the total time
7 total_time = end_time - start_time
8 print(f"Total time taken: {total_time} seconds")
```

prey

3
6
9
12
15

C:\Users\Aleks\AppData\Local\anaconda3\Lib\site-packages\scipy\integrate\odepack_py.py:248: ODEintWarning: Excess accuracy requested (tolerances too small). Run with full_output = 1 to get quantitative information.
warnings.warn(warning_msg, ODEintWarning)

predator

3
6
9
12
15

C:\Users\Aleks\AppData\Local\anaconda3\Lib\site-packages\scipy\integrate\odepack_py.py:248: ODEintWarning: Illegal input detected (internal error). Run with full_output = 1 to get quantitative information.
warnings.warn(warning_msg, ODEintWarning)
C:\Users\Aleks\AppData\Local\anaconda3\Lib\site-packages\scipy\integrate\odepack_py.py:248: ODEintWarning: Run terminated (internal error). Run with full_output = 1 to get quantitative information.
warnings.warn(warning_msg, ODEintWarning)

both

3
6
9
12
15

Total time taken: 1060.7844800949097 seconds

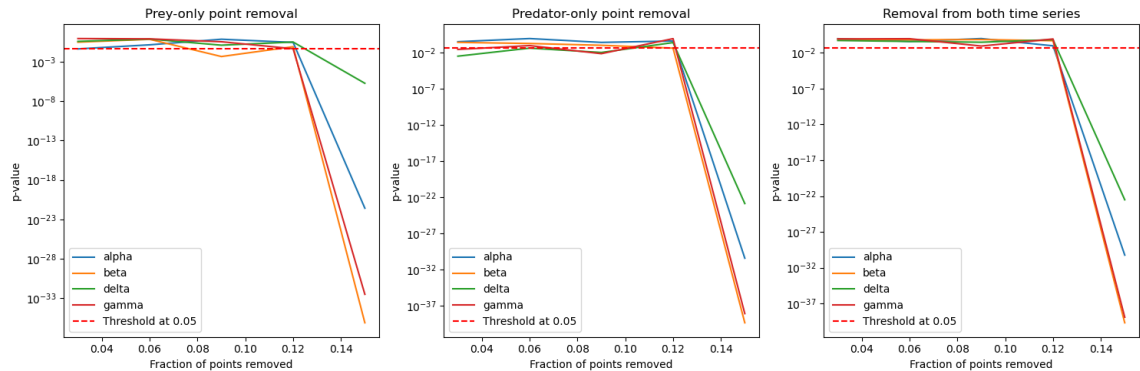
Visualizing p-values from welch test for hill climbing

In [43]:

```

1  # Fraction points removed
2  fraction_points = np.arange(3,16,3) / 100
3  p_values_pre = np.array(p_values_hill_climbing['prey'])
4  p_values_predator = np.array(p_values_hill_climbing['predator'])
5  p_values_both = np.array(p_values_hill_climbing['both'])
6
7
8  # Creating subplots
9  fig, axes = plt.subplots(1, 3, figsize=(15, 5))
10
11 # Plotting p-values when only prey points are removed
12 axes[0].plot(fraction_points, p_values_pre[:, 0], label='alpha')
13 axes[0].plot(fraction_points, p_values_pre[:, 1], label='beta')
14 axes[0].plot(fraction_points, p_values_pre[:, 2], label='delta')
15 axes[0].plot(fraction_points, p_values_pre[:, 3], label='gamma')
16 axes[0].axhline(y=0.05, color='r', linestyle='--', label='Threshold at
17 axes[0].set_ylabel('p-value')
18 axes[0].set_xlabel('Fraction of points removed')
19 axes[0].set_yscale('log')
20 axes[0].set_title('Prey-only point removal') # Add title to the first
21 axes[0].legend()
22
23 # Plotting p-values when only predator points are removed
24 axes[1].plot(fraction_points, p_values_predator[:, 0], label='alpha')
25 axes[1].plot(fraction_points, p_values_predator[:, 1], label='beta')
26 axes[1].plot(fraction_points, p_values_predator[:, 2], label='delta')
27 axes[1].plot(fraction_points, p_values_predator[:, 3], label='gamma')
28 axes[1].axhline(y=0.05, color='r', linestyle='--', label='Threshold at
29 axes[1].set_ylabel('p-value')
30 axes[1].set_xlabel('Fraction of points removed')
31 axes[1].set_yscale('log')
32 axes[1].set_title('Predator-only point removal')
33 axes[1].legend()
34
35 # Plotting p-values when both prey and predator points are removed
36 axes[2].plot(fraction_points, p_values_both[:, 0], label='alpha')
37 axes[2].plot(fraction_points, p_values_both[:, 1], label='beta')
38 axes[2].plot(fraction_points, p_values_both[:, 2], label='delta')
39 axes[2].plot(fraction_points, p_values_both[:, 3], label='gamma')
40 axes[2].axhline(y=0.05, color='r', linestyle='--', label='Threshold at
41 axes[2].set_ylabel('p-value')
42 axes[2].set_xlabel('Fraction of points removed')
43 axes[2].set_yscale('log')
44 axes[2].set_title('Removal from both time series')
45 axes[2].legend()
46
47 # Adjusting layout
48 plt.tight_layout()
49 plt.savefig('welch_tests_hill_climbing', dpi = 300)
50 plt.show()
51

```



Getting distribution of averages of best guesses for simulated annealing (reference dataset)

In [40]:

```

1  # We get the reference distribution for testing
2
3  # Timing your code
4  start_time = time.time()
5
6  initial_temp = 20
7  cooling_constant = 0.10
8
9  #Taking random draw for initial parameters (initial guess)
10 alpha = np.random.uniform(0,1)
11 beta = np.random.uniform(0,1)
12 delta = np.random.uniform(0,1)
13 gamma = np.random.uniform(0,1)
14 parameters = [alpha, beta, delta, gamma]
15 parameters = [alpha, beta, delta, gamma]
16
17 # Reference distribution of averages
18 ref_average2 = []
19 for k in range(50):
20     # parameter_list, best_param, scores, best_score, num_iterations =
21     parameter_list, scores = multiple_runs_annealing(initial_temp, cooling_constant, parameters)
22     # Appending average
23     ref_average2.append(np.mean(parameter_list, axis=0))
24
25 ref_average2 = np.array(ref_average2)
26 end_time = time.time()
27
28 # Calculating and printing the total time
29 total_time = end_time - start_time
30 print(f"Total time taken: {total_time} seconds")

```

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: overflow encountered in exp

acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: overflow encountered in scalar divide

acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

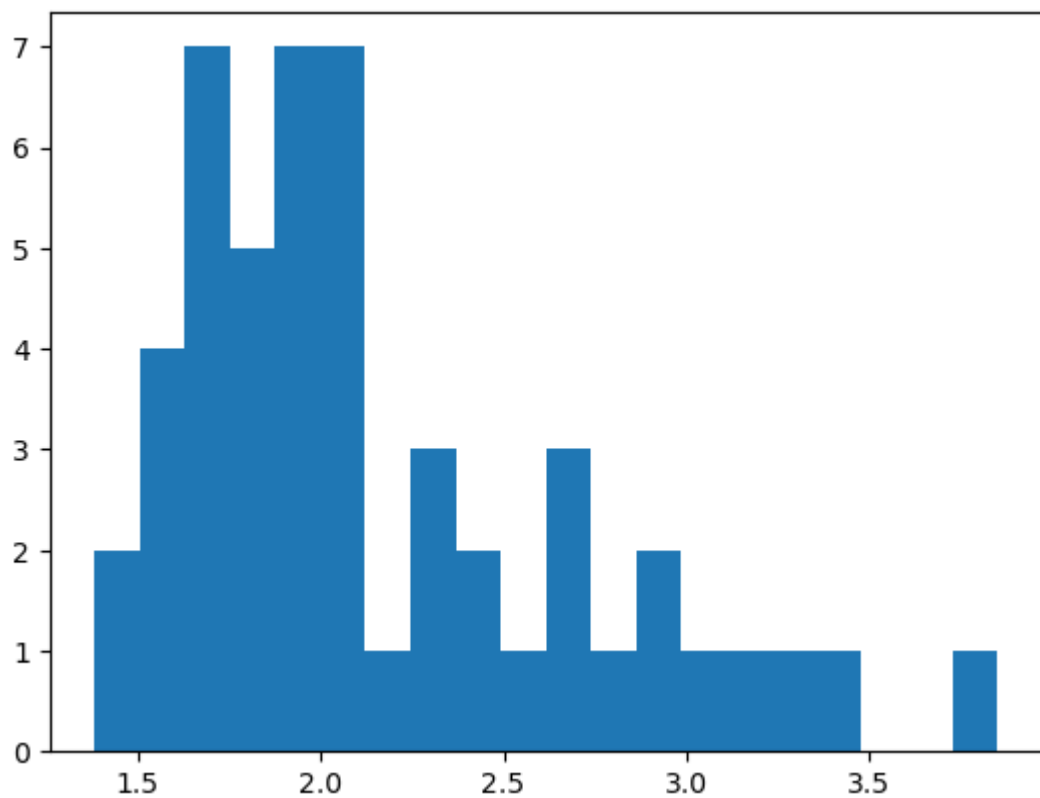
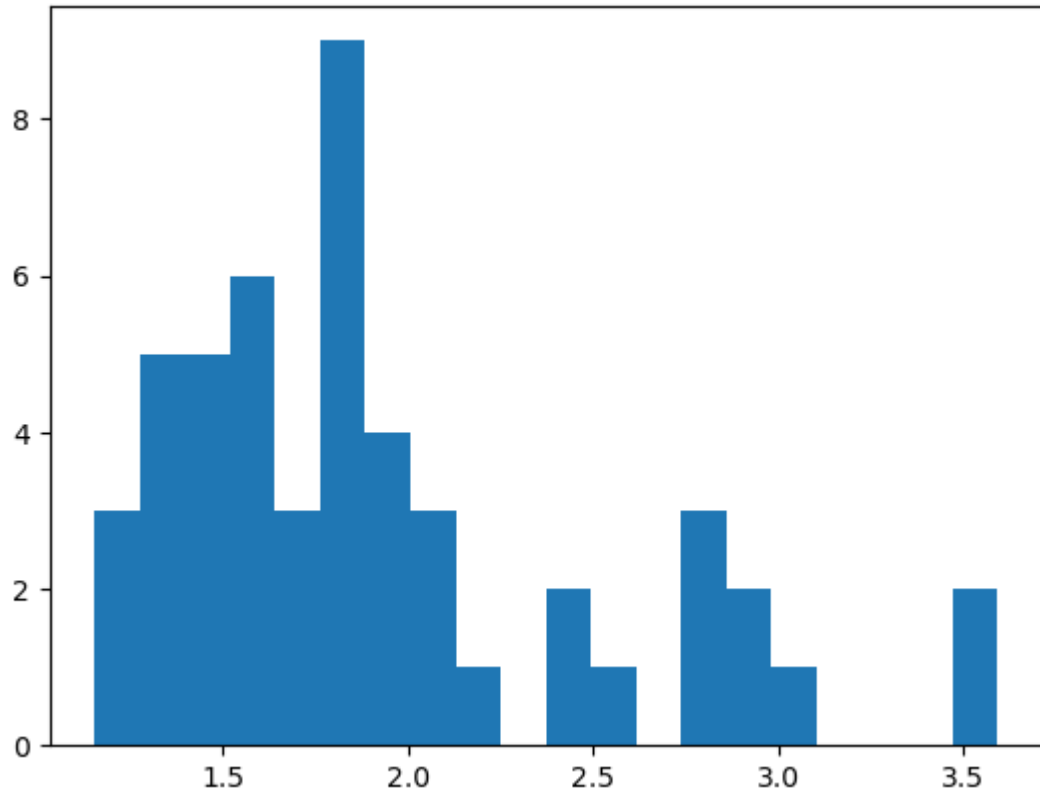
C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: divide by zero encountered in scalar divide

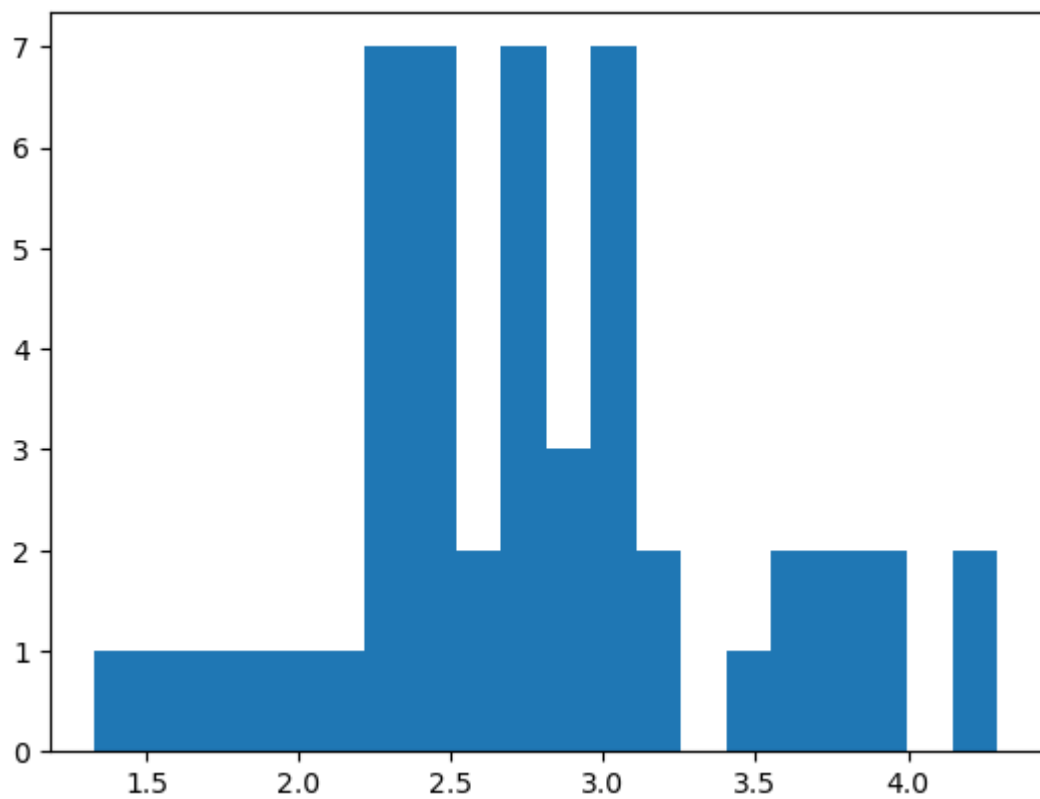
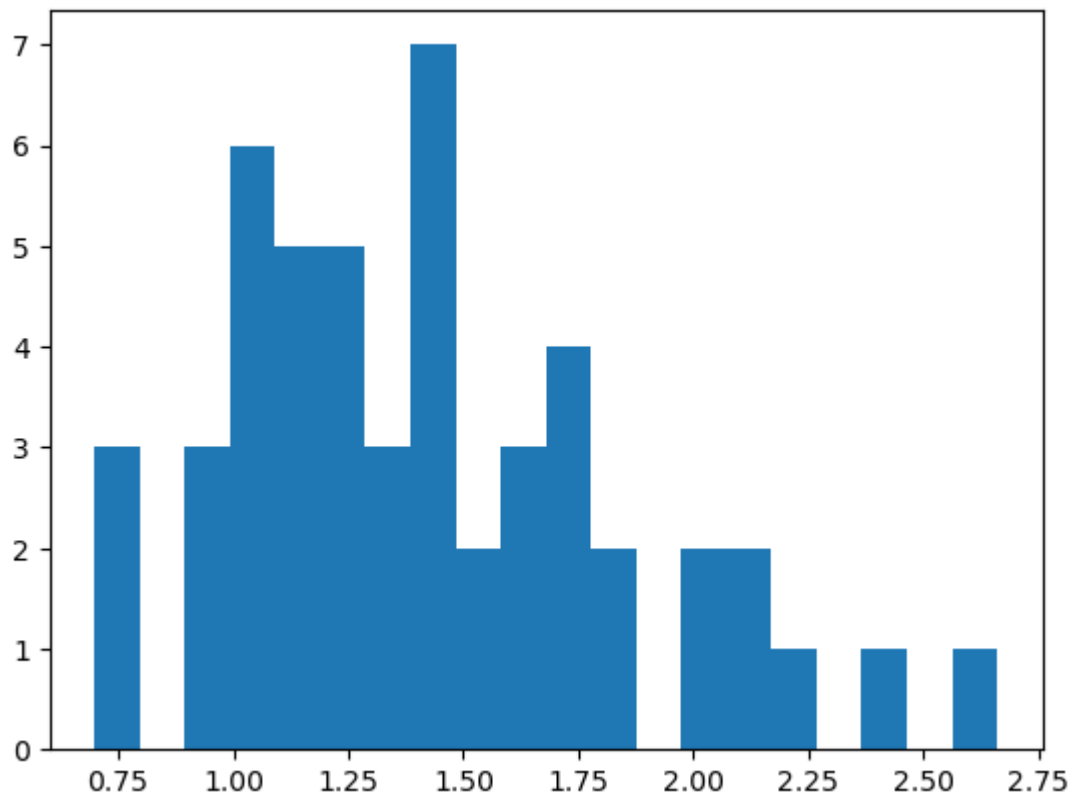
acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

Total time taken: 1028.2772996425629 seconds


```
In [41]: 1 # print(np.var(ref_average1,axis=0))
2 print(np.var(ref_average2,axis=0))
3
4 for k in range(4):
5     plt.hist(ref_average2[:,k], bins = 20)
6     plt.show()
7
```

[0.35481283 0.29947082 0.18320262 0.4135741]





```

In [20]: 1 def random_point_ttests2(data, ref_distribution, points_removed, focus_c
2         '''runs welch test for multi run of simulated annealing for every p
3         param_distribution = ref_distribution
4         # focus_choices = ['prey', 'predator', 'both']
5         # focus_choices = ['prey', 'predator', 'both']
6         scores = [[], []]
7         p_values = {'prey': [], 'predator': [], 'both': []}
8         initial_temp = 20
9         cooling_constant = 0.10
10
11        for indx, choice in enumerate(focus_choices):
12            print(choice)
13            #We Iteratively increase the amount of points we remove
14            for npoints in points_removed:
15                print(f"points: {npoints}")
16                # print(f"Points removed: {npoints}")
17                limited_data = point_removal(data[:,0], data[:,1:3], npoint
18                #Getting distribution of averages
19                average_distribution = []
20                for k in range(30):
21                    parameter_list, best_score = multiple_runs_annealing(ir
22                    #Appending average
23                    average_distribution.append(np.mean(parameter_list, axi
24
25                average_distribution = np.array(average_distribution)
26                t_stat1, p_value1 = scipy.stats.ttest_ind(ref_distribution[
27                t_stat2, p_value2 = scipy.stats.ttest_ind(ref_distribution[
28                t_stat3, p_value3 = scipy.stats.ttest_ind(ref_distribution[
29                t_stat3, p_value4 = scipy.stats.ttest_ind(ref_distribution[
30                p_values[choice].append([p_value1, p_value2, p_value3, p_va
31
32        return p_values

```

```
In [44]: 1 start_time = time.time()
2
3 # Calculating p-values for t-test
4 points_removed_annealing = np.arange(3,16,3)
5 p_values_annealing = random_point_ttests2(data,ref_average2,points_remo
6
7 end_time = time.time()
8 # Calculating and printing the total time
9 total_time = end_time - start_time
10 print(f"Total time taken: {total_time/60} min")
```

prey

points: 3

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: overflow encountered in exp

acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: overflow encountered in scalar divide

acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: divide by zero encountered in scalar divide

acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

points: 6

points: 9

points: 12

points: 15

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\951072704.py:25: RuntimeWarning: invalid value encountered in scalar divide

acceptance_probability = min(np.exp(-(delta/temp)),1)#Calculate acceptance probability

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\264107711.py:4: RuntimeWarning: overflow encountered in scalar multiply

dxdt = (alpha * x) - (beta * x * y) # Predator ODE

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\264107711.py:5: RuntimeWarning: overflow encountered in scalar multiply

dydt = (delta * x * y) - (gamma * y) # Predator ODE

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\2008888099.py:4: RuntimeWarning: overflow encountered in square

return np.mean((actual - predicted)**2)

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\2008888099.py:15: RuntimeWarning: overflow encountered in square

err1 = (x1[indx_x] - x2[indx_x])**2

predator

points: 3

points: 6

points: 9

points: 12

points: 15

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\2008888099.py:16: RuntimeWarning: overflow encountered in square

err2 = (y1[indx_y] - y2[indx_y])**2

both

points: 3

points: 6

points: 9

points: 12

points: 15

C:\Users\Aleks\AppData\Local\Temp\ipykernel_764\264107711.py:4: RuntimeWarning: invalid value encountered in scalar subtract

$dxdt = (\alpha * x) - (\beta * x * y)$ # Predator ODE

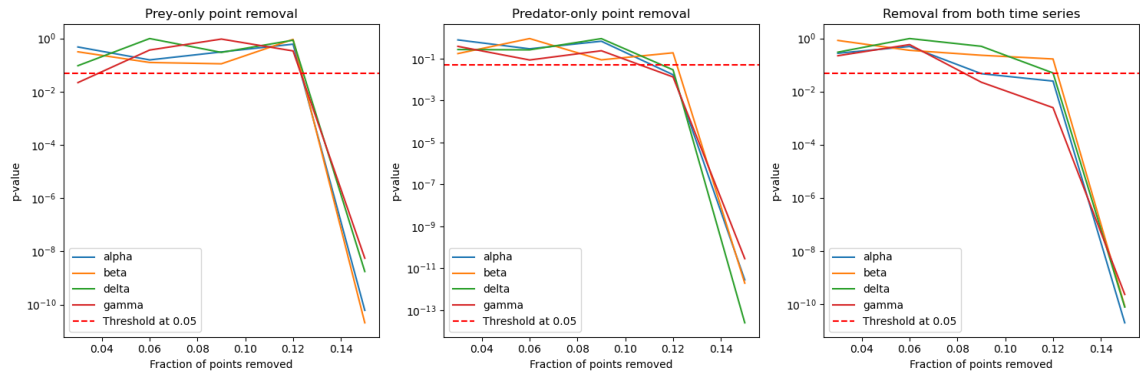
Total time taken: 63.41379015445709 min

In [46]:

```

1  # Fraction points removed
2
3  # Fraction points removed
4
5  fraction_points = np.arange(3,16,3) / 100
6  p_values_pre = np.array(p_values_annealing['prey'])
7  p_values_predator = np.array(p_values_annealing['predator'])
8  p_values_both = np.array(p_values_annealing['both'])
9
10 # Creating subplots
11 fig, axes = plt.subplots(1, 3, figsize=(15, 5))
12
13 # Plotting p-values when only prey points are removed
14 axes[0].plot(fraction_points, p_values_pre[:, 0], label='alpha')
15 axes[0].plot(fraction_points, p_values_pre[:, 1], label='beta')
16 axes[0].plot(fraction_points, p_values_pre[:, 2], label='delta')
17 axes[0].plot(fraction_points, p_values_pre[:, 3], label='gamma')
18 axes[0].axhline(y=0.05, color='r', linestyle='--', label='Threshold at
19 axes[0].set_ylabel('p-value')
20 axes[0].set_xlabel('Fraction of points removed')
21 axes[0].set_yscale('log')
22 axes[0].set_title('Prey-only point removal') # Add title to the first
23 axes[0].legend()
24
25 # Plotting p-values when only predator points are removed
26 axes[1].plot(fraction_points, p_values_predator[:, 0], label='alpha')
27 axes[1].plot(fraction_points, p_values_predator[:, 1], label='beta')
28 axes[1].plot(fraction_points, p_values_predator[:, 2], label='delta')
29 axes[1].plot(fraction_points, p_values_predator[:, 3], label='gamma')
30 axes[1].axhline(y=0.05, color='r', linestyle='--', label='Threshold at
31 axes[1].set_ylabel('p-value')
32 axes[1].set_xlabel('Fraction of points removed')
33 axes[1].set_yscale('log')
34 axes[1].set_title('Predator-only point removal')
35 axes[1].legend()
36
37 # Plotting p-values when both prey and predator points are removed
38 axes[2].plot(fraction_points, p_values_both[:, 0], label='alpha')
39 axes[2].plot(fraction_points, p_values_both[:, 1], label='beta')
40 axes[2].plot(fraction_points, p_values_both[:, 2], label='delta')
41 axes[2].plot(fraction_points, p_values_both[:, 3], label='gamma')
42 axes[2].axhline(y=0.05, color='r', linestyle='--', label='Threshold at
43 axes[2].set_ylabel('p-value')
44 axes[2].set_xlabel('Fraction of points removed')
45 axes[2].set_yscale('log')
46 axes[2].set_title('Removal from both time series')
47 axes[2].legend()
48
49 plt.savefig('welch_test_annealing', dpi=300)
50
51 # Adjusting layout
52 plt.tight_layout()
53 plt.show()

```



In []:

1