Electronic Supplementary Material (ESI) for Soft Matter.

Supporting information

Combinatorial mixtures of organic solutes for improved liquid/liquid extraction of ions

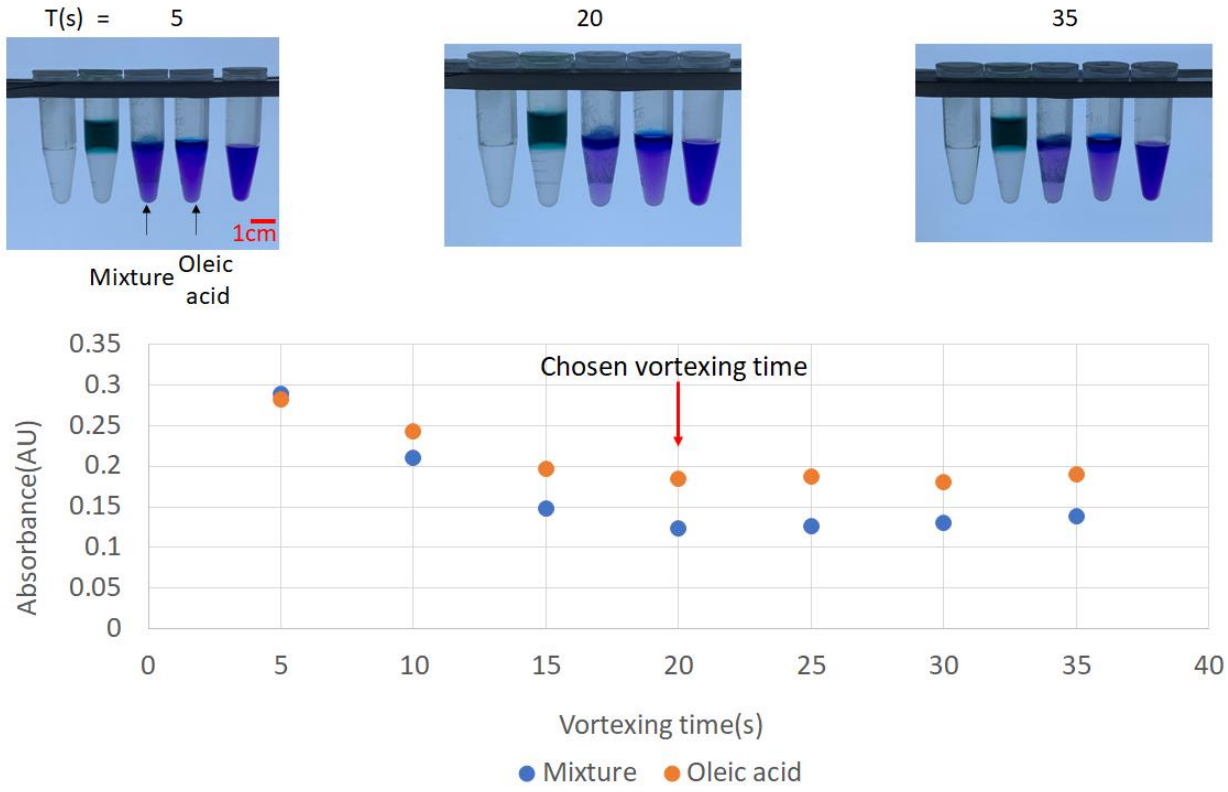

Figure S1. Evidence of extraction reaching equilibrium after vortexing 20 seconds. The absorbance value of the extraction reaches an equilibrium state after vortexing 20s. (Top) Images of tubes after extraction using pure oleic acid and organic solutes along with vortexing time, and the samples in the tubes from left to right are water, oleic acid/aqueous 1000 ppm copper ions with a volume ratio of 1: 1, mixtures in oleic acid/aqueous copper ions with ratio 1:8, oleic acid/aqueous copper ions, 1000 ppm copper ions.(Bottom) A plot of absorbance as a function of time.

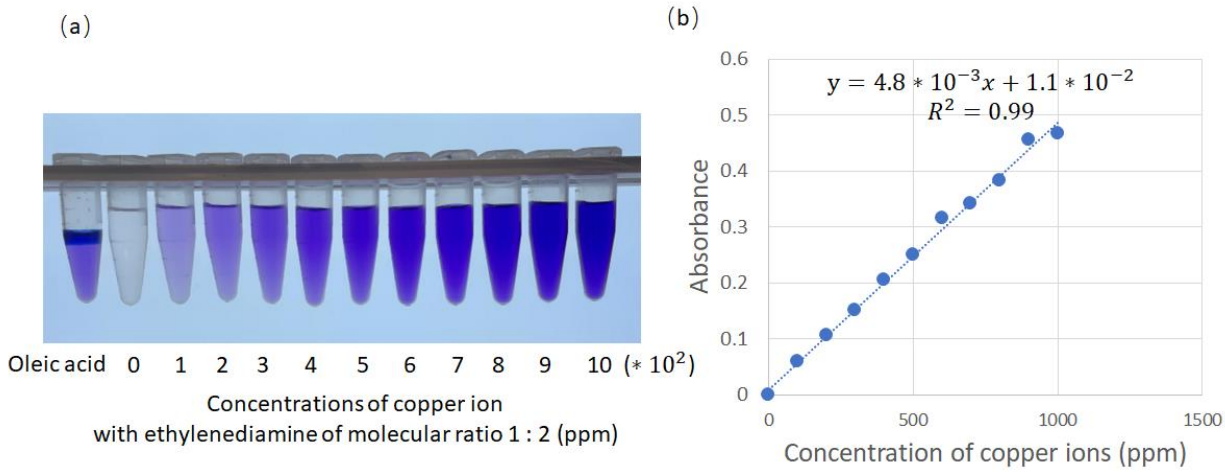

Figure S2. Calibration of the cell phone camera for centrifuge tube setup. (a) Images of tubes with different concentrations of copper ions with indicator of molecular ratio 1:2. (b) A plot of concentration of copper ions with absorbance.

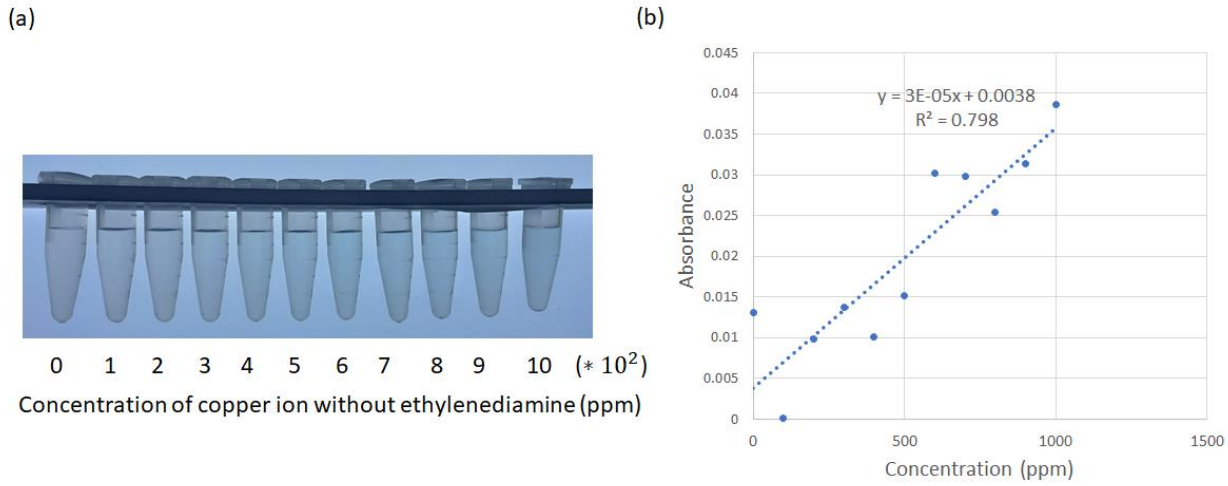

Figure S3. Calibration of the cell phone camera for copper ions. (a) Smart phone images of tubes with different concentrations of copper ions (without indicator). (b) A plot of concentration with absorbance.

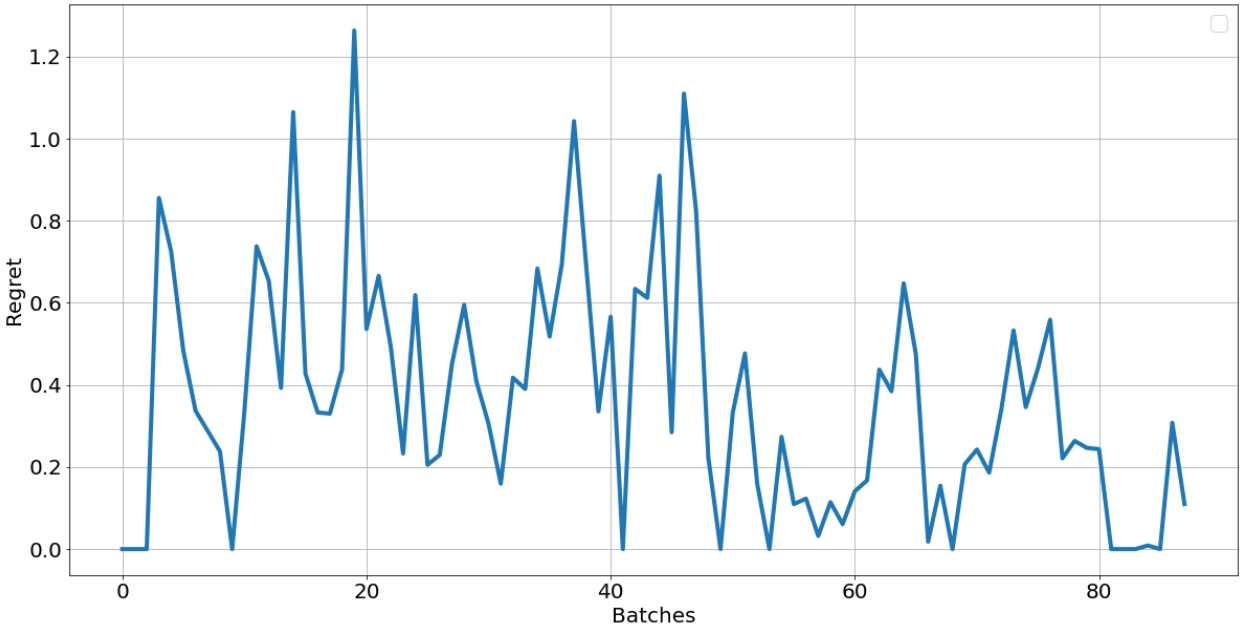

Figure S4. Demonstration of Bayesian optimization performance through a downward trend of regrets over batches (the regret is defined as the expected difference between optimal strategy and Bayesian strategy).

Sample number	Copper ions concentration in water(ppm)	Standard deviation(ppm)	Average variance
1	384.0402977	35.35801516	97.78335107
2	263.1275788	103.8762646	
3	375.1179503	3.265011627	
4	394.5549935	62.87145635	
5	364.0729028	145.451705	
6	447.7064966	34.25372547	
7	551.7766002	123.0477832	
8	450.5217577	146.0756935	
9	501.9374732	154.3800681	
10	565.8556522	149.7774449	
11	382.9314156	158.4986765	
12	405.9279595	31.52853646	
13	396.7559108	158.1762507	
14	394.9530589	241.1909338	
15	262.6675408	92.0264939	
16	229.378658	77.28199625	
17	209.6877931	73.16778584	
18	214.4402258	64.30642698	
19	154.4253635	21.93805726	
20	321.1003064	120.8499486	
21	213.9290572	56.12809836	

Figure S5. Table of 20 combinations with the concentration of copper ion, standard deviation and the average variance.

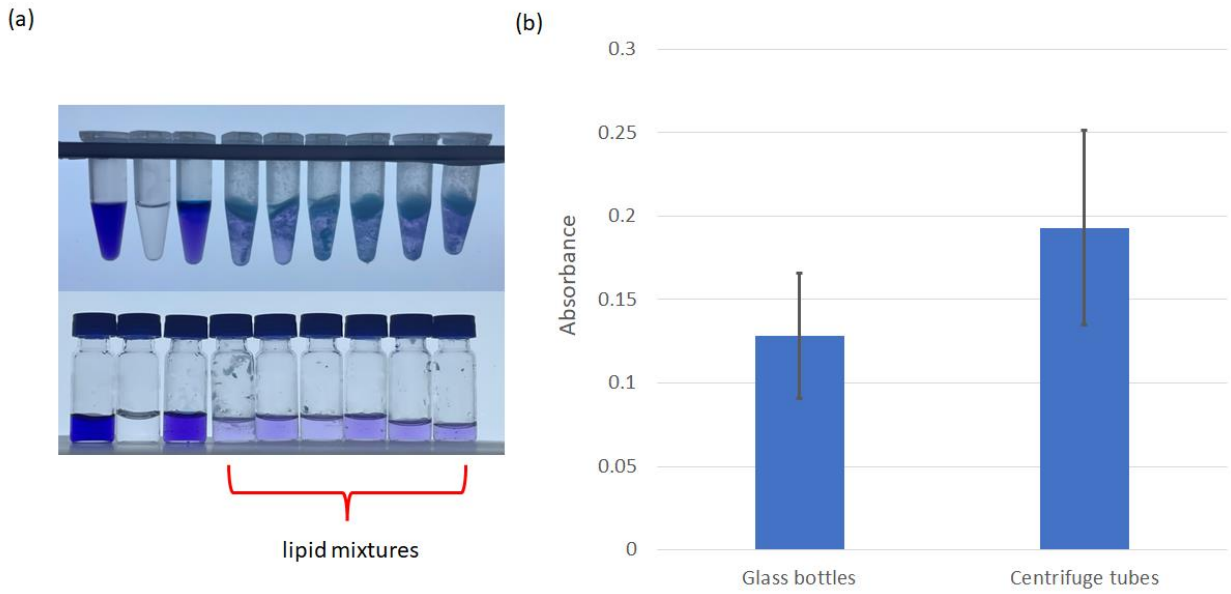

Figure S6. Measurement of concentrations by taking the aqueous solution out and centrifuging it in a glass bottle after extraction. (a) Images of extraction results in centrifuges tubes and glass bottles, and the samples in the tubes from left to right are 1000 ppm copper ions, water, oleic acid/copper ions with volume ratio 1: 8(control), six replicates of optimal mixtures in oleic acid/copper ions with volume ratio 1:8. (b) A bar plot of absorbance comparison using centrifuges tube and glass bottles with 6 replicates, shows a similar uncertainty.

A(mM)	B(mM)	C(mM)	D(mM)	E(mM)	Copper ions concentration in water(ppm)
15.000	120.000	7.500	7.500	7.500	545.196
1.875	60.000	7.500	7.500	7.500	529.127
7.500	7.500	30.000	30.000	30.000	520.508
250.000	0.122	500.000	250.000	7.813	520.508
31.250	0.122	125.000	62.500	15.625	520.508
0.244	0.244	7.813	250.000	125.000	520.508
250.000	0.122	125.000	0.122	250.000	520.508
0.244	250.000	250.000	0.061	62.500	520.508
7.813	0.977	0.977	0.488	125.000	520.508
0.977	62.500	250.000	0.122	500.000	376.483
31.250	1000.000	3.906	7.813	31.250	376.483
0.061	0.977	0.488	31.250	1000.000	376.483
1.953	0.244	500.000	1.953	500.000	376.483
31.250	0.488	0.977	0.977	0.061	376.483
15.625	3.906	1000.000	500.000	3.906	376.483
1.953	125.000	0.244	0.244	31.250	376.483
1.953	62.500	0.977	1000.000	7.813	376.483
0.061	1.953	31.250	0.122	62.500	376.483
0.122	15.625	3.906	125.000	0.061	376.483
125.000	500.000	250.000	250.000	1000.000	376.483
7.813	31.250	62.500	0.061	125.000	376.483
250.000	250.000	1.953	1000.000	62.500	376.483
250.000	0.244	0.122	250.000	125.000	376.483
7.813	1000.000	15.625	7.813	125.000	376.483
1000.000	0.244	1000.000	0.061	250.000	376.483
0.244	1000.000	0.977	0.488	62.500	376.483
125.000	125.000	3.906	3.906	250.000	376.483
0.488	0.977	7.813	0.977	3.906	376.483
62.500	15.625	0.977	0.977	0.061	376.483
15.625	500.000	7.813	1.953	3.906	376.483
7.813	7.813	0.488	125.000	7.813	376.483
500.000	1000.000	0.061	0.061	0.488	376.483
125.000	0.061	1.953	0.977	1000.000	376.483
3.906	31.250	0.977	250.000	3.906	376.483
1000.000	0.061	31.250	0.977	0.977	376.483
31.250	0.977	31.250	7.813	0.122	376.483
125.000	0.061	500.000	62.500	62.500	376.483
15.625	125.000	500.000	250.000	125.000	376.483
0.977	250.000	250.000	0.122	15.625	376.483

15.625	3.906	7.813	15.625	500.000	376.483
15.625	31.250	0.061	0.488	62.500	376.483
0.061	1000.000	15.625	125.000	3.906	366.531
0.244	15.625	500.000	1000.000	0.061	366.531
0.061	0.061	0.244	500.000	15.625	366.531
62.500	0.122	1000.000	3.906	31.250	366.531
1.953	0.977	0.488	0.244	1000.000	366.531
3.906	125.000	125.000	500.000	500.000	366.531
500.000	125.000	0.244	0.122	7.813	366.531
1000.000	1000.000	7.813	31.250	15.625	366.531
62.500	1000.000	0.488	1.953	31.250	310.178
62.500	1000.000	0.122	3.906	7.813	310.178
500.000	1000.000	31.250	1.953	1000.000	310.178
0.488	1000.000	0.244	0.244	31.250	310.178
31.250	1000.000	7.813	0.977	250.000	300.445
125.000	1000.000	0.977	0.244	15.625	300.445
0.244	1000.000	31.250	15.625	7.813	300.445
125.000	1000.000	1.953	62.500	500.000	300.445
0.061	1000.000	0.244	0.488	0.061	300.445
250.000	1000.000	3.906	0.122	0.488	300.445
0.122	1000.000	15.625	0.244	250.000	300.445
0.488	1000.000	62.500	7.813	0.122	300.445
1000.000	1000.000	1.953	0.244	125.000	300.445
125.000	1000.000	62.500	0.122	500.000	300.445
250.000	1000.000	1.953	0.244	125.000	300.445
1000.000	1000.000	0.977	0.061	0.488	300.445
1000.000	1000.000	31.250	0.061	500.000	300.445
250.000	1000.000	0.122	0.977	31.250	300.445
31.250	1000.000	0.488	31.250	0.488	300.445
250.000	1000.000	31.250	0.244	0.122	263.128
0.977	1000.000	31.250	62.500	3.906	263.128
1000.000	1000.000	1.953	62.500	500.000	263.128
62.500	1000.000	31.250	15.625	250.000	263.128
3.906	1000.000	62.500	0.061	62.500	263.128
125.000	1000.000	15.625	15.625	125.000	263.128
3.906	1000.000	1.953	3.906	1000.000	263.128
3.906	1000.000	62.500	1.953	1000.000	263.128
0.977	1000.000	3.906	31.250	7.813	263.128
62.500	1000.000	0.122	62.500	7.813	263.128
7.813	1000.000	0.977	62.500	31.250	263.128
31.250	1000.000	0.061	0.061	0.977	263.128
7.813	1000.000	62.500	31.250	7.813	263.128
0.488	1000.000	62.500	62.500	0.488	262.668

31.250	1000.000	0.977	62.500	0.488	229.379
31.250	1000.000	62.500	0.244	15.625	209.688
62.500	1000.000	3.906	31.250	500.000	209.688
125.000	1000.000	1.953	15.625	500.000	154.425
3.906	1000.000	7.813	31.250	7.813	154.425
0.061	1000.000	0.061	0.061	500.000	154.425

Figure S7. Table of combinations with the concentration of copper ion.

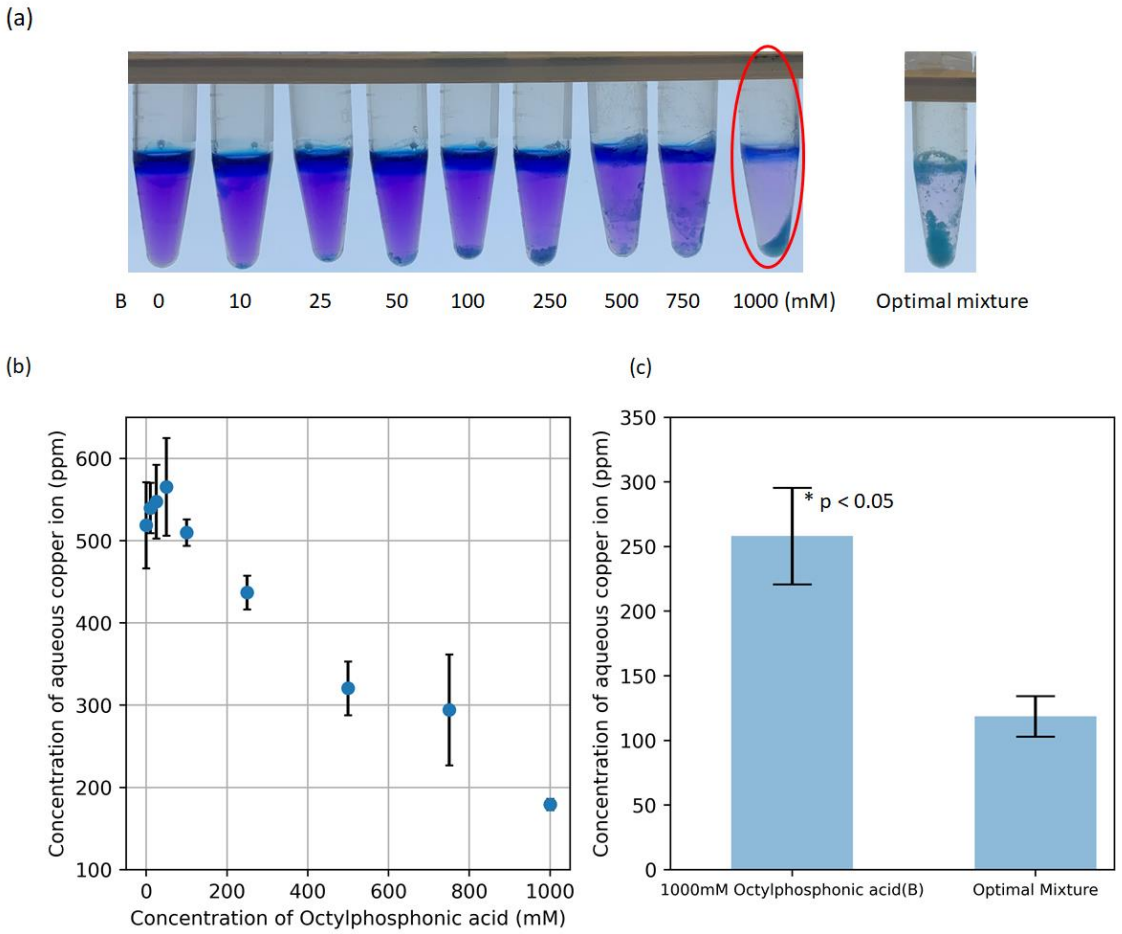

Figure S8. The combination performs better than the best individual component. Octylphosphonic acid improves extraction efficiency at a concentration above 200 mM; The extraction efficiency of the best mixture is better than pure component B.(a) Tube image of extraction using different concentrations of octylphosphonic acid in oleic acid/ 1000 ppm copper ions with a volume ratio of (1:8) and compare with the extraction performance using optimal mixtures. (b) Relationship between the different concentration of octylphosphonic acid and ions remaining in water after extraction (c) Bar plot of extraction performance between 1000 mM octylphosphonic acid and optimal mixture, t-test shows two samples are significantly different with p-value < 0.05 with a sample size of 4.

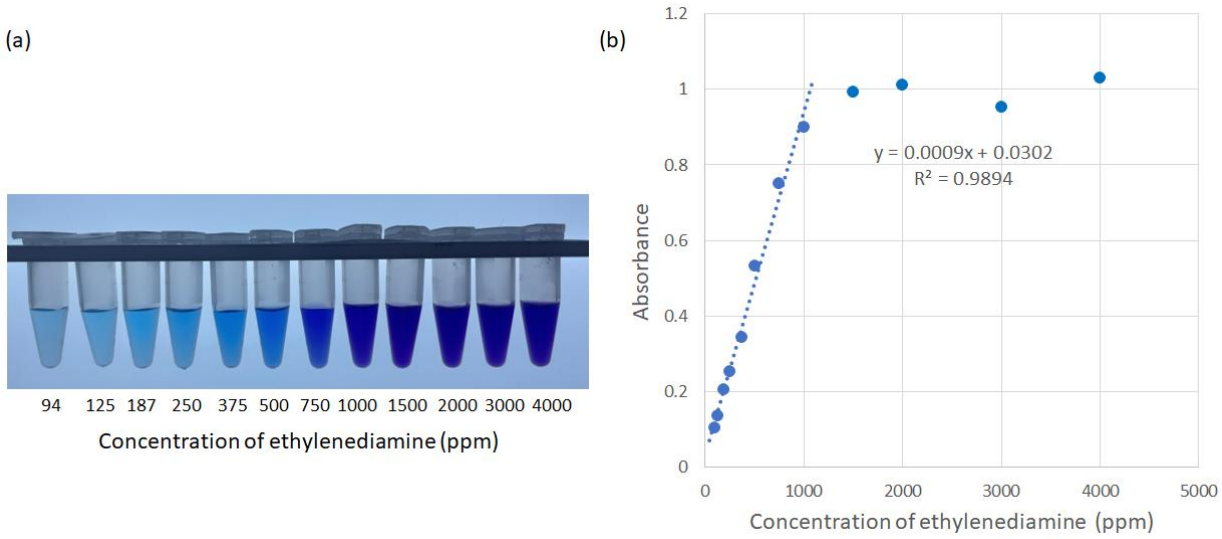

Figure S9. Calibration of the cell phone camera for ethylenediamine as an indicator of copper ions. (a) Tube images of different concentrations of ethylenediamine for 1000 ppm copper ions. (b) A plot of concentration with absorbance.

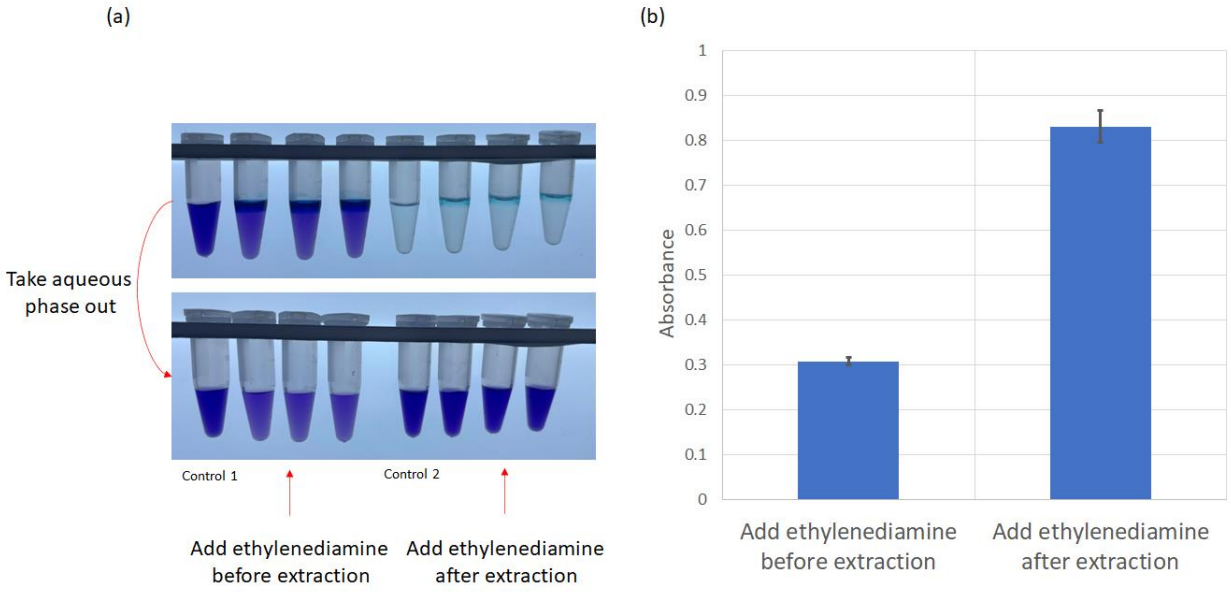

Figure S10. Extraction comparison with/without the addition of ethylenediamine in the water. (a) Images of tubes after extraction done by adding ethylenediamine before and after extraction, and the samples in the tubes from left to right are 1000 ppm copper ions and 3 replicates of oleic acid/copper ions of ratio 1:8 adding ethylenediamine before and after extraction. (b) A bar plot of extraction result of adding ethylenediamine before and after extraction.

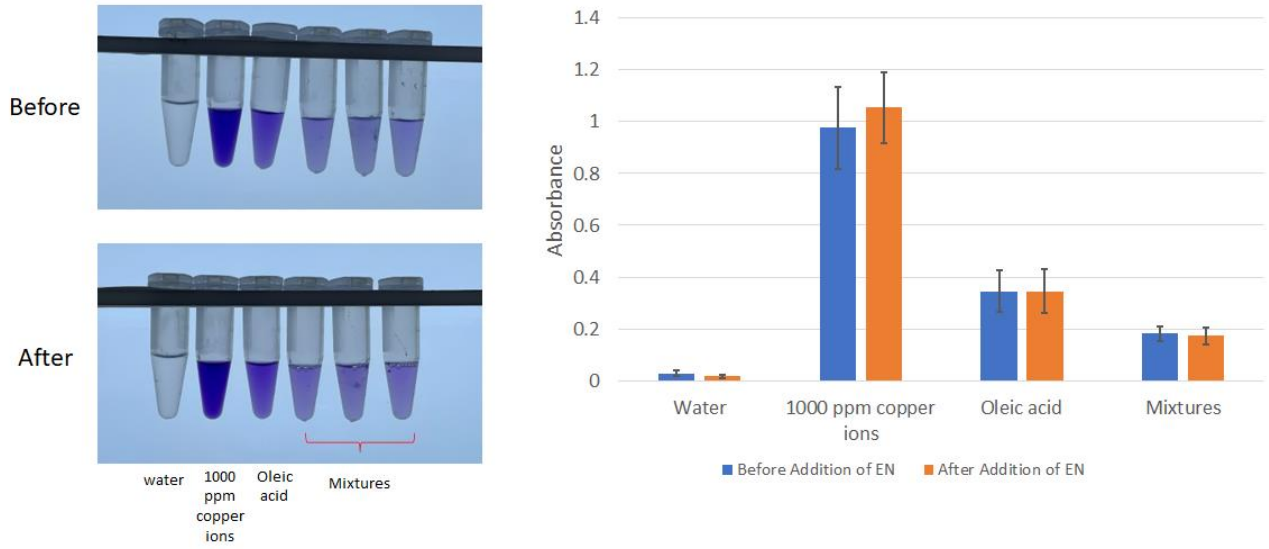

Figure S11. Extraction performance before and after the addition of ethylenediamine in the aqueous after the extraction of copper ions. The error bar in the graph represents the standard deviation of measurement and replicates.

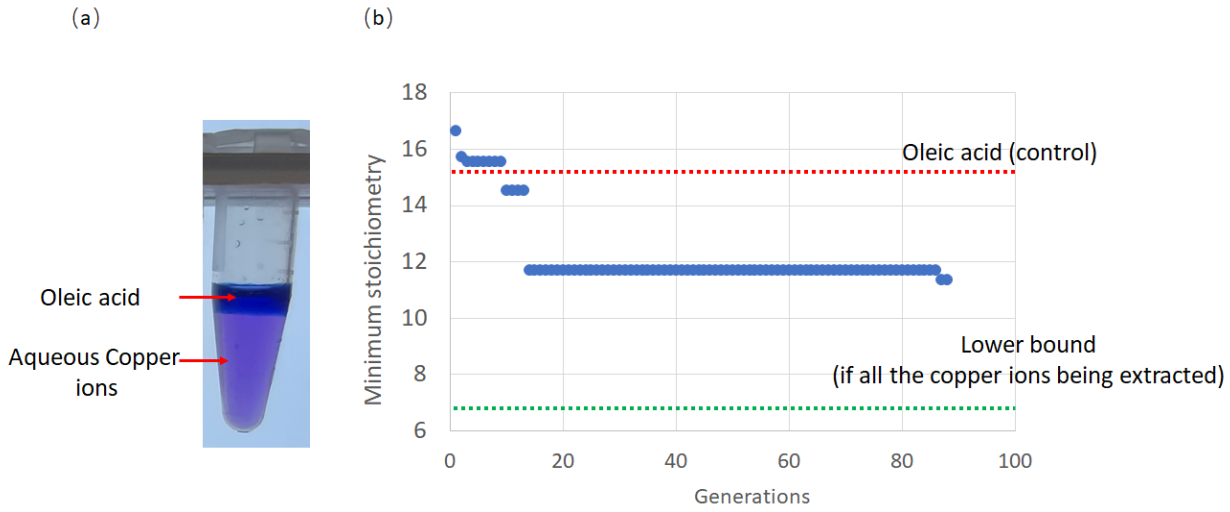

Figure S12. Stoichiometry of extraction experiment with Bayesian optimization. (a) tube image using only octylphosphonic acid and control. (b) A plot of lipid mixture with best stoichiometry value along with generations.

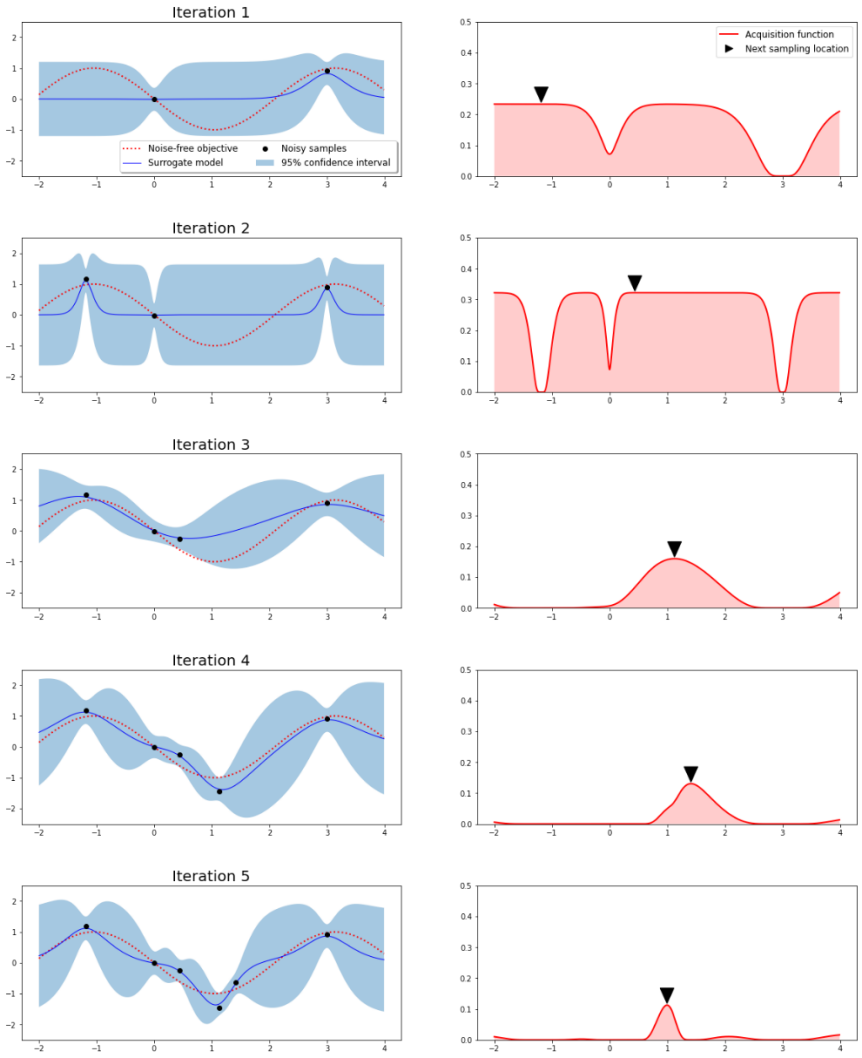

Figure S13. An illustration of the one-dimensional BO procedure over five iterations.

```
# -*- coding: utf-8 -*-
"""Supplementary_BO.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1x8SgDNpBuJWcxig2TLDyQUGDiTui039R
"""

############################################################ Attatched python code is excecuted in google colab
#########################################################################
#############################################################################################
#############################################################################################
###################################

#### Norimal inverse gamma distribution
from numpy import sum, mean, size, sqrt
from scipy.stats import norm, invgamma

def draw_mus_and_sigmas(data,m0,k0,s_sq0,v0,n_samples=10000):
    # number of samples
    N = size(data)
    # find the mean of the data
    the_mean = mean(data)
    # sum of squared differences between data and mean
    SSD = sum( (data - the_mean)**2 )

    # combining the prior with the data - page 79 of Gelman et al.
    # to make sense of this note that
    # inv-chi-sq(v,s^2) = inv-gamma(v/2,(v*s^2)/2)
    kN = float(k0 + N)
    mN = (k0/kN)*m0 + (N/kN)*the_mean
    vN = v0 + N
    vN_times_s_sqN = v0*s_sq0 + SSD + (N*k0*(m0-the_mean)**2)/kN
    s_sqN = vN_times_s_sqN/vN

    # 1) draw the variances from an inverse gamma
    # (params: alpha, beta)
    alpha = vN/2
    beta = vN_times_s_sqN/2
    # thanks to wikipedia, we know that:
    # if X ~ inv-gamma(a,1) then b*X ~ inv-gamma(a,b)
    sig_sq_samples = beta*invgamma.rvs(alpha,size=n_samples)
```

```
    # 2) draw means from a normal conditioned on the drawn sigmas
    # (params: mean_norm, var_norm)
    mean_norm = mN
    var_norm = sqrt(sig_sq_samples)/kN
    mu_samples = norm.rvs(mean_norm,scale=var_norm,size=n_samples)

    # 3) return the mu_samples and sig_sq_samples
    return kN,mN,vN,s_sqN,mu_samples, sig_sq_samples

#### A sampling test from normal inverse gamma distribution
from numpy.random import normal
from matplotlib import pyplot as plt
from statistics import median

# step 1: define prior parameters for the normal and inverse gamma
m0 = 0.
k0 = 1.
s_sq0 = 1.
v0 = 1.

# step 2: get some random data, with slightly different statistics
data1 = normal(loc=5, scale=1, size=100)
#data2 = normal(loc=-1, scale=1, size=10)
##data3 = normal(loc=15, scale=1, size=100)
#data4 = normal(loc=4, scale=1, size=10)
#data5 = normal(loc=8, scale=1, size=10)
##data6 = normal(loc=7.5, scale=1, size=10)
#data7 = normal(loc=7.8, scale=1, size=10)



# step 3: get posterior samples
k1,m1,v1,s_sq1,mus1,sig_sqs1 = draw_mus_and_sigmas(data1,m0,k0,s_sq0,v0)
plt.plot(sig_sqs1)
plt.figure()
plt.hist(sig_sqs1,bins=20)
##mus2,sig_sqs2 = draw_mus_and_sigmas(data2,m0,k0,s_sq0,v0)
#mus3,sig_sqs3 = draw_mus_and_sigmas(data3,m0,k0,s_sq0,v0)
#mus4,sig_sqs4 = draw_mus_and_sigmas(data4,m0,k0,s_sq0,v0)
#mus5,sig_sqs5 = draw_mus_and_sigmas(data5,m0,k0,s_sq0,v0)
#mus6,sig_sqs6 = draw_mus_and_sigmas(data6,m0,k0,s_sq0,v0)
#mus7,sig_sqs7 = draw_mus_and_sigmas(data7,m0,k0,s_sq0,v0)
```

Bayesian optimization

```
!pip install Gpy==1.9.8
!pip install GpyOpt==1.2.6
import numpy as np

import GPy
import GPyOpt

from GPyOpt.methods import BayesianOptimization


np.random.seed(1)
#Continous domain
#bounds = np.array([0,1000])

#Discrete domain
base = 2
max = 1000
x = np.arange(0,15)
bounds  =  max/(base**x)

##Matern kernel for GP
kernel = GPy.kern.Matern52(input_dim=5, variance=1.0, lengthscale=1.0)

##Bound for five lipid
bds = [{'name': 'Lipid', 'type': 'discrete', 'domain': bounds,'dimensionality': 5}]

#####Constraint: Manully add constraints sequentially to block the previous suggested combinations and generate a new one.  #####
con = [{'name':'constr_1','constraint':'-(x[:,1]-1000.0)'},
      {'name':'constr_2','constraint':'x[:,2]-125.0+0.000001'},
      {'name':'constr_3','constraint':'x[:,3]-125.0+0.000001'},

      ]


##### We manually add the observed X and y in the initial samples of GP for each run to sequentially optimized the combination. The first three are the initial samples randomly tried.#####
X = np.array([
[15,     120,    7.5,     7.5,     7.5],
[1.875, 60,      7.5,     7.5,     7.5],
[7.5,    7.5,    30,      30,      30],
```

[250.0,0.1220703125,500.0,250.0,7.8125],
[31.25,0.1220703125,125.0,62.5,15.625],
[0.244140625,0.244140625,7.8125,250.0,125.0],
[250.0,0.1220703125,125.0,0.1220703125,250.0],
[0.244140625,250.0,250.0,0.06103515625,62.5],
[7.8125,0.9765625,0.9765625,0.48828125,125.0],
[0.9765625,62.5,250.0,0.1220703125,500.0],
[31.25,1000.0,3.90625,7.8125,31.25],
[0.06103515625,0.9765625,0.48828125,31.25,1000.0],

[1.953125,0.244140625,500.0,1.953125,500.0],
[31.25,0.48828125,0.9765625,0.9765625,0.06103515625],
[15.625,3.90625,1000.0,500.0,3.90625],

[1.953125,125.0,0.244140625,0.244140625,31.25],
[1.953125,62.5,0.9765625,1000.0,7.8125],
[0.06103515625,1.953125,31.25,0.1220703125,62.5],

[0.1220703125,15.625,3.90625,125.0,0.06103515625],
[125.0,500.0,250.0,250.0,1000.0],
[7.8125,31.25,62.5,0.06103515625,125.0],

[250.0,250.0,1.953125,1000.0,62.5],
[250.0,0.244140625,0.1220703125,250.0,125.0],
[7.8125,1000.0,15.625,7.8125,125.0],

[1000,0.244140625,1000,0.06103515625,250],
[0.244140625,1000,0.9765625,0.48828125,62.5],
[125.0,125.0,3.90625,3.90625,250.0],

[0.48828125,0.9765625,7.8125,0.9765625,3.90625],
[62.5,15.625,0.9765625,0.9765625,0.06103515625],
[15.625,500.0,7.8125,1.953125,3.90625],

[7.8125,7.8125,0.48828125,125.0,7.8125],
[500.0,1000.0,0.06103515625,0.06103515625,0.48828125],
[125.0,0.06103515625,1.953125,0.9765625,1000.0],

[3.90625,31.25,0.9765625,250.0,3.90625],
[1000.0,0.06103515625,31.25,0.9765625,0.9765625],

[31.25,0.9765625,31.25,7.8125,0.1220703125],

[125.0, 0.06103515625, 500.0, 62.5, 62.5],
[15.625, 125.0, 500.0, 250.0, 125.0],
[0.9765625, 250.0, 250.0, 0.1220703125, 15.625],

[15.625,3.90625,7.8125,15.625,500.0],
[15.625,31.25,0.06103515625,0.48828125, 62.5],
[0.06103515625,1000.0,15.625,125.0,3.90625],

[0.244140625, 15.625, 500.0, 1000.0, 0.06103515625],
[0.06103515625, 0.06103515625, 0.244140625, 500.0, 15.625],
[62.5, 0.1220703125, 1000.0, 3.90625, 31.25],

[1.953125, 0.9765625, 0.48828125, 0.244140625, 1000.0],
[3.90625, 125.0, 125.0, 500.0, 500.0],
[500.0, 125.0, 0.244140625, 0.1220703125, 7.8125],

[1000.0,1000.0,7.8125,31.25,15.625],
[62.5,1000.0,0.48828125,1.953125,31.25],
[62.5,1000.0,0.1220703125,3.90625,7.8125],

[500.0,1000.0,31.25,1.953125,1000.0],
[0.48828125,1000.0,0.244140625,0.244140625,31.25],
[31.25,1000.0,7.8125,0.9765625,250.0],

[125.0, 1000.0, 0.9765625, 0.244140625, 15.625],
[0.244140625, 1000.0, 31.25, 15.625, 7.8125],
[125.0,1000.0,1.953125,62.5,500.0],

[0.06103515625,1000.0,0.244140625,0.48828125,0.06103515625],
[250.0,1000.0,3.90625,0.1220703125,0.48828125],
[0.1220703125,1000.0,15.625,0.244140625,250.0],

[0.48828125, 1000.0, 62.5, 7.8125, 0.1220703125],
[1000.0, 1000.0, 1.953125, 0.244140625, 125.0],
[125.0, 1000.0, 62.5, 0.1220703125, 500.0],

[250.0,1000.0,1.953125,0.244140625,125.0],

[1000.0,1000.0,0.9765625,0.06103515625,0.48828125],

[1000.0,1000.0,31.25,0.06103515625,500.0],

[250.0,1000.0,0.1220703125,0.9765625,31.25],

[31.25,1000.0,0.48828125,31.25,0.48828125],

[250.0,1000.0,31.25,0.244140625,0.1220703125],

[0.9765625,1000.0,31.25,62.5,3.90625],
[1000.0,1000.0,1.953125,62.5,500.0],
[62.5,1000.0,31.25,15.625,250.0],

[3.90625,1000.0,62.5,0.06103515625,62.5],
[125.0,1000.0,15.625,15.625,125.0],
[3.90625,1000.0,1.953125,3.90625,1000.0],

[3.90625,1000.0,62.5,1.953125,1000.0],
[0.9765625,1000.0,3.90625,31.25,7.8125],
[62.5,1000.0,0.1220703125,62.5,7.8125],

[7.8125,1000.0,0.9765625,62.5,31.25],
[31.25,1000.0,0.06103515625,0.06103515625,0.9765625],
[7.8125,1000.0,62.5,31.25,7.8125],

[0.48828125,1000.0,62.5,62.5,0.48828125],
[31.25,1000.0,0.9765625,62.5,0.48828125],
[31.25,1000.0,62.5,0.244140625,15.625],

[62.5,1000.0,3.90625,31.25,500.0],
[125.0,1000.0,1.953125,15.625,500.0],
[3.90625,1000.0,7.8125,31.25,7.8125],

[0.06103515625,1000.0,0.06103515625,0.06103515625,500.0]

])

```
y = np.array([
1.048824131,
1.019156664,
1.003246445,

1.859133429,
1.728314403,
1.486032029,

1.340374067,
1.291074878,
1.241435245,

0.737352393,
1.072208218,
1.475318643,

1.390809871,
1.129947411,
1.802342897,

1.165039202,
1.070355892,
1.067391564,

1.175026627,
2.001202681,
1.273685468,


1.403225241,
1.228475345,
0.970363522,

1.356263681,
0.942686627,
0.96697517,

1.187786093,
1.332546874,
1.146295319,
```

1.042671082,
0.896783996,
1.154858516,

1.127543065,
1.42114431,
1.255466996,

1.431245026,
1.780268116,
1.426882571,

1.072903122,
1.303509767,
0.718979632,

1.353187628,
1.331260153,
1.629187232,

1.003715271,
1.829033333,
1.542589328,

0.941977352,
0.614944568,
0.946972622,

1.091737437,
0.774839267,
0.596975675,

0.870516692,
0.706652393,
0.719849778,

0.629092735,
0.711292765,
0.657801492,

0.738030993,
0.764355007,

1.034278979,

0.981541443,
1.244570874,
1.071772202,

0.615097421,
0.751305165,
0.528081684,

0.734833139,
0.770716911,
0.714442282,

0.868842763,
1.060972185,
0.874040168,

0.968961489,
1.086964281,
0.749257998,

0.791713156,
0.774780143,
0.771451801,

0.527232383,
0.465775984,
0.429423618,

0.43819734,
0.327400671,
0.635108258,

0.437253644,

]
)

Manully input the newly-generated samples (9 samples per run: 3 combinations and each with 3 replicates)
y_update= np.array([

```
 1.157341307,
 0.936019749,
 1.053111336,
 0.96783278,
 1,
 1.089637212,
 0.925933613,
 1.101246199,
 0.982559522,


]
)

##### Manully change the 4 parameters on the updated (posterior) normal inverse gammma distribution for each run except for initial run (the reuslts will be shown in the console)  #####
##### For example: First run: we use initial parameter are used. Second run: the 4 parameters updated k,m,v,s_sqn,mus in the previous run will be used as prior
k0=k,m0=m,v0=v,s_sqn0=s_sqn.
#Initial parameter
m0 = np.mean(y)
k0 = 1.
s_sq0 = 1.
v0 = 1.

#Previous posterior as prior
#m0 = 1.0486222511221375
#k0 = 262.0
#s_sq0 = 0.15488160222723504
#v0 = 262.0

k,m,v,s_sqn,mus,sig_sqs = draw_mus_and_sigmas(y_update,m0,k0,s_sq0,v0)
print('medain of mu and sig_sqs: ',median(mus),median(sig_sqs))
print('posterior pararmeter: ',k,m,v,s_sqn)

noise=np.sqrt(median(sig_sqs))

def simple_f(X,noise=noise):
  score = 0+ np.random.normal(0, noise, 1)

  return score

optimizer = GPyOpt.methods.BayesianOptimization(f=simple_f,
                                domain=bds,
```

```
                        constraints = con,
                        model_type='GP',
                        kernel=kernel,
                        acquisition_type ='EI',
                        ##### Manually adjust the jitter for each run execpt for the initial run (The
initial jitter we set is 0.01)#####
                        acquisition_jitter = 0.0001348,

                        X=X,
                        Y=y.reshape(-1,1),
                        noise_var= median(sig_sqs),

                        exact_feval=False,
                        normalize_Y=False,
                        maximize=False)

optimizer.run_optimization(max_iter=0)

print('The combination in the history:', optimizer.X )
print('The scores for all the combinations in the history:', optimizer.Y )
print('The best combination in the history:', optimizer.X[optimizer.Y.argmin()] )
print('The best score for all the combinations in the history:',min(optimizer.Y)  )

next_combination = optimizer.suggest_next_locations()
print('The next combination would be:',next_combination)

print('A: ',next_combination[0][0] )
print('B: ',next_combination[0][1] )
print('C: ',next_combination[0][2] )
print('D: ',next_combination[0][3] )
print('E: ',next_combination[0][4] )

##### The code for dynamic aquisition jitter for next run
import itertools

bounds = list(bounds)
a = [bounds,bounds,bounds,bounds,bounds]
combination_a = list(itertools.product(*a))

combination_a = np.array(combination_a)

sum_var = 0
time = 0
```

```
for i in combination_a:
  sum_var = (sum_var + (optimizer.model.predict(i)[1])**2)
  time += 1

average_var = sum_var/time

f_star = min(optimizer.Y)

jitter =(average_var/(1/f_star))/(1.22652742*100)
print(jitter)
#print( optimizer.model.predict( np.array([[15.625, 62.5, 250.0, 7.8125, 0.244140625]]) )[1] )
#np.shape(next_combination)
#print(next_combination)
```