REEE_ED_Takahashi_Model_Description

August 20, 2020

1 Full Description of Takahashi Model Progress (Python Implementation)

Prepared by: Water Management Section

Link to Model

1.0.1 Introduction

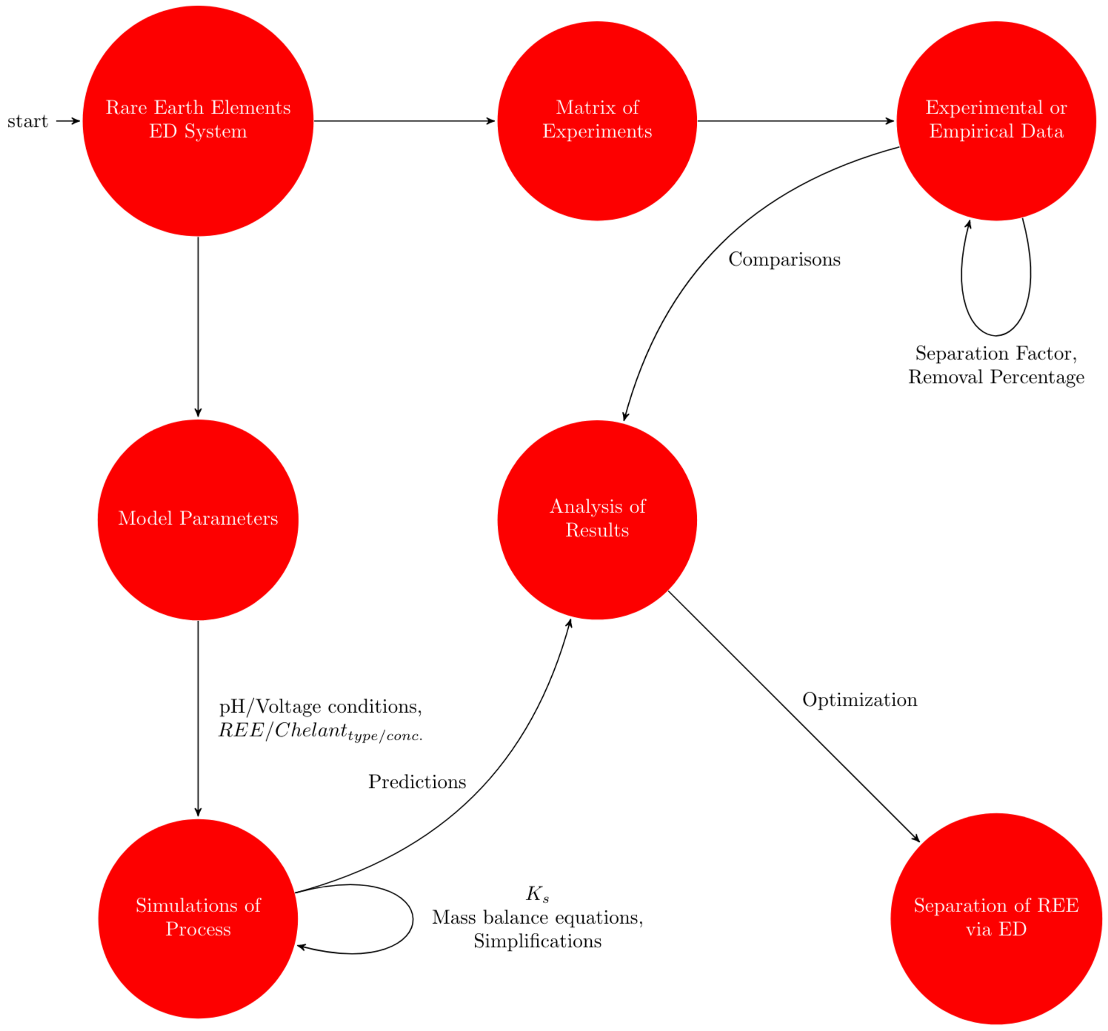
The Rare Earth Elements (REE) and Chromite R&D Program at CanmetMINING mandated an investigation into the separation of REE via electrodialysis (ED) since year 3. Motivation behind pursuing ED technology was elucidated in previous NRCan REE workshops and year-end reports.[1][2]

REE-ED's year 5 experimental plan assessed the effect of chelating agents on LREE/LREE separations. Similarly, cascade (two-phase) separations and varying SX formulations were also explored. In the latter half of FY 2019-2020, the Water Management Section head proposed an open-minded and common sense approach to optimize testing: employ a model to simulate the chelation-assisted competitive binding of REEs in ED processes.

Naturally, REE separations are difficult largely attributing to their similarities in chemical properties. The literature has previously established models for dynamic modelling of REE separation. For example, the MATLAB/SIMULINK solvent extraction model for REE separation as shown by *Lyon et al.*[3] In particular, *Takahashi et al.* have highlighted how their proposed mathematical model can be utilized to explain their findings in electrodialysis REE separation research.[4][7] Therefore, with the advent of COVID-19 resulting in the shutdown of facilities, it was an opportune time to consider this computational simulation approach.

This *Takahashi* model prototype has been implemented using **python** and seeks to demonstrate: * The merit of the *Takahashi* model. * Reproducibility of *Takahashi et al.*'s findings. * Models/Chemometrics are viable ad-hoc tools in optimizing ED REE separation testing. * python/jupyter notebooks are ideal.

1.0.2 Figure 1. Diagram of plan to optimize REE-ED project.


start
Rare Earth Elements ED System
Matrix of Experiments
Experimental or Empirical Data
Comparisons
Separation Factor, Removal Percentage
Model Parameters
Analysis of Results
pH/Voltage conditions, $REE/Chelant_{type/conc.}$
Optimization
Predictions
Simulations of Process
K_s Mass balance equations, Simplifications
Separation of REE via ED

1.0.3 Background Theory (multi-component systems and equilibrium)

According to *VanBriesen et al.*, mass balance equations can be written as:[5]

$$\boxed{M_j = \sum_{j=1}^{N_c} A_{ij}\frac{x_i}{\gamma_i}} \tag{1}$$

In short, to find the total mass of the component you simply find the quantity of the component in every relevant species and sum them up.

Within the system, according to the theoretical literature as shown by Mackey, there are two prominent competition cases we have to consider w.r.t. competing for the chelating:[6]

1. Equilibrium scenario where proton and metal ion compete.

$$\boxed{M^{+m} + H_hY^{h-y} <=> MY^{m-y} + hH^+} \tag{2}$$

2. Equilibrium scenario where two metal ions compete.

$$\boxed{M^{+m} + NY^{n-y} <=> MY^{m-y} + N^+} \tag{3}$$

Note: M is for metal, Y is for chelating agent, and N is for second metal.

From these scenarios we can determine stability constants:

$$\boxed{K_f = [ML_n]/[M][L]^n} \tag{4}$$

$$\boxed{K_2 = \frac{[MY^{m-y}][H^+]^h}{[M^{+m}][H_hY^{h-y}]} = K_{MY} * K_a} \tag{5}$$

$$\boxed{K_3 = \frac{[MY^{m-y}][N^{+n}]}{[NY^{n-y}][M^{+m}]} = \frac{[K_{MY}]}{[K_{NY}]}} \tag{6}$$

Mass Balance Equations (note: ignore hydroxide products):

$$\boxed{[L]_{total} = \sum_{m=1}^{M}\sum_{h=0}^{4}[MH_hL^{3+h-l}] + [L^{-l}]_{free}} \tag{7}$$

$$\boxed{[M]_{total} = \sum_{h=0}^{4}[MH_hL^{3+h-l}] + [M^{3+}]_{free} + \sum_{i=1}^{3}[M(OH)_i^{3-}]} \tag{8}$$

1.0.4 The Takahashi Model

From their paper, the following model equations were obtained: > > **Note: The english translated version should be available in the P: drive.**

Let it be assumed as a La^{3+} and Nd^{3+} situation:

Solving Takahashi Model Equations 2-6 Simultaneously

Takahashi Model Equations 2-6

Objective: *Solving the equations simultaneously for* $C_{La^{3+}}, C_{Nd^{3+}}, C_{EDTA^{4-}}, C_{LaEDTA^{-}}, C_{NdEDTA^{-}}$, *and* $C_{EDTA,non-complex}$ *values.*

The equations:

$$\boxed{C_{La^{3+}} = \frac{C_{LaEDTA^{-}}}{K_{ABS,La} \cdot C_{EDTA^{4-}}}} \tag{2.a}$$

$$\boxed{C_{Nd^{3+}} = \frac{C_{NdEDTA^{-}}}{K_{ABS,Nd} \cdot C_{EDTA^{4-}}}} \tag{2.b}$$

$$\boxed{C_{La,total} = C_{La^{3+}} + C_{LaEDTA^{-}}} \tag{3.a}$$

$$\boxed{C_{Nd,total} = C_{Nd^{3+}} + C_{NdEDTA^{-}}} \tag{3.b}$$

$$\boxed{C_{EDTA,total} = \sum_{i=1}^{2} (C_{LaEDTA^{-}} + C_{NdEDTA^{-}}) + C_{EDTA,nc}} \tag{4}$$

$$\boxed{C_{EDTA^{4-}} = \alpha \cdot C_{EDTA,nc}} \tag{5}$$

$$\boxed{\alpha = k_1k_2k_3k_4/(k_1k_2k_3k_4 + k_1k_2k_3C_H + k_1k_2{C_H}^2 + k_1{C_H}^3 + {C_H}^4)} \tag{6}$$

Initial values:

$$\begin{array}{lll} K_{ABS,La} = 10^{15.50}, & k_1 = 10^{-2}, & C_{La,total} = 10\ mol/m^3, \\ K_{ABS,Nd} = 10^{16.61}, & k_2 = 10^{-2.67}, & C_{Nd,total} = 10\ mol/m^3, \\ & k_3 = 10^{-6.16}, & C_{EDTA,total} = 5\ mol/m^3, \\ & k_4 = 10^{-10.26}, & C_H(pH = 3) = 10^{-3} = 0.001\ M = 1\ mol/m^3 \end{array}$$

Solving Takahashi Model Equations 7-9 Simultaneously

Takahashi Model Equations 7-9

Objective: *Solving the equations simultaneously for $\overline{C_H}$ values.*

The equations:

$$\boxed{\overline{C_{RE_i}} = K_H^{RE_i}\overline{C_H}^3\left(\frac{C_{RE_i^{3+}}}{C_H{}^3}\right)^{0.8}} \tag{7}$$

$$\boxed{\overline{C_{Na}} = K_H^{Na}\overline{C_H}\left(\frac{C_{Na}}{C_H}\right)} \tag{8}$$

$$\boxed{Q = 3\sum_{i=1}^{n}\overline{C_{RE_i}} + \overline{C_{Na}} + \overline{C_H}} \tag{9}$$

Initial values, let:

$$\begin{array}{lll} K_H^{La^{3+}} = 1.49, & K_H^{Nd^{3+}} = 1.15, & K_H^{Na} = 0.32 \\ C_{RE_i^{3+}} = 10\ mol/m^3, & C_H\ (at\ pH = 3) = 1\ mol/m^3, & C_{Na} = C_{EDTA} = 5\ mol/m^3 \end{array}$$

>>

$$\begin{aligned} \overline{C_{La^{3+}}} &= 1.49 \cdot \overline{C_H}^3 \cdot \left(\tfrac{10}{1^{-3}}\right)^{0.8} = 1.49 \cdot \overline{C_H}^3 \cdot 10^{0.8} \\ \overline{C_{Nd^{3+}}} &= 1.15 \cdot \overline{C_H}^3 \cdot \left(\tfrac{10}{1^{-3}}\right)^{0.8} = 1.15 \cdot \overline{C_H}^3 \cdot 10^{0.8} \\ \overline{C_{Na}} &= 0.32 \cdot \overline{C_H} \cdot \left(\tfrac{5}{1}\right) = 1.6 \cdot \overline{C_H} \\ Q &= 3 \cdot \left(2.64 \cdot 10^{0.8} \cdot \overline{C_H}^3\right) + 1.6\overline{C_H} + \overline{C_H} \end{aligned}$$

Let Q be equal to an arbitrary value, say 100 mol/m^3

solving for $\overline{C_H}$: $\boxed{\overline{C_H} = 1.25\ mol/m^3}$

$Therefore,$ $\boxed{\overline{C_{La^{3+}}} = 18.2\ mol/m^3}$, $\boxed{\overline{C_{Nd^{3+}}} = 14.0\ mol/m^3}$, $\boxed{\overline{C_{Na}} = 1.99\ mol/m^3}$

(Equations 1, 10-15 are all simplifications). By solving equations 2-9 simultaneously, can then proceed solve the following DE (equation 16) for the REE ions La^{3+} and Nd^{3+}. Likewise, can apply to any binary REE^{3+} and REE^{3+}. Ultimately, separation factor (SF) can be obtained from those computed concentrations.

$$-V_F \frac{dC_i}{dt} = V_s \frac{dC_{si}}{dt} = S\overline{J}_i = S \frac{z_i D_i \overline{C}_i}{\sum_{i=1}^{n+2} z_i^2 D_i \overline{C}_i} \frac{I}{F} \tag{16}$$

Important note on Takahashi references:

The second *Takahashi et al.* [7] reference talks about separation of yttrium from La-Y and neodymium from Nd-Gd-Y. They looked at how flux changes with respect to changing chelating agents (EDTA vs. lactic acid), ratio of chelating agents to REE, and pH. They calculated separation factor by simply dividing the flux of the REE with higher permeation over the lower permeation REE (e.g. it would be $SF_{J_M/J_{RE}} = \frac{J_{La}}{J_Y}$ or $SF_{J_{Nd}/J_{Gd}} = \frac{J_{Nd}}{J_{Gd}}$, $SF_{J_{Nd}/J_Y} = \frac{J_{Nd}}{J_Y}$

The first *Takahashi et al.* [4] reference (translated by NRCan) provides the model for group separation of REE. Second reference uses the exact same model.

1.0.5 Python implementation of the Takahashi Model

"Talk is cheap. Show me the code." - Linus Torvalds of Linux

Techniques and methods used to implement the model were inspired by CS 370 coursenotes and Hedengren's python tutorial courses.[9][10]

This code cell simply imports all relevant modules.

```
import numpy as np
from scipy.integrate import odeint, solve_ivp
from scipy.optimize import *
from scipy.interpolate import make_interp_spline, interp2d
from IPython.display import display
from mpl_toolkits import mplot3d
%matplotlib inline
import matplotlib.pyplot as plt
import ipywidgets as widgets
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
```

Surface area and I were quantities obtained from the paper. F is Faraday constant. Volume is from the experiments in the lab typically being around 1250 mL. Units are in meters. Elements class holds all sets of constants belonging to each respective element.

```
surface_area = 0.005
valence = 3
I = 20
F = 96485
```

```
V = 0.00125

# Container for each REE/Chelating Agent combo.

class Elements:

    def __init__(self, stab, K_hydro, diff_coeff):

        # Absolute stability constants (compiled by Sanaz)
        self.stab = stab

        # K_H and diffusion coefficients from Takahashi
        self.K_hydro = K_hydro
        self.diff_coeff = diff_coeff
```

These values were obtained from tables compiled by Sanaz. They were originally from Karraker's paper.[8]

```
La_EDTA = Elements(np.power(10, 15.50), 1.49, 4.4e13)
Pr_EDTA = Elements(np.power(10, 16.40), 1.40, 4.8e13)
Nd_EDTA = Elements(np.power(10, 16.61), 1.15, 5.2e13)
Gd_EDTA = Elements(np.power(10, 17.37), 0.91, 6.5e13)
Y_EDTA = Elements(np.power(10, 18.09), 0.66, 1.1e14)

La_DCTA = Elements(np.power(10, 16.60), 1.49, 4.4e13)
Pr_DCTA = Elements(np.power(10, 17.01), 1.40, 4.8e13)
Nd_DCTA = Elements(np.power(10, 17.31), 1.15, 5.2e13)
Gd_DCTA = Elements(np.power(10, 18.70), 0.91, 6.5e13)
Y_DCTA = Elements(np.power(10, 19.60), 0.66, 1.1e14)

La_HEDTA = Elements(np.power(10, 13.22), 1.49, 4.4e13)
Pr_HEDTA = Elements(np.power(10, 14.39), 1.40, 4.8e13)
Nd_HEDTA = Elements(np.power(10, 14.71), 1.15, 5.2e13)
Gd_HEDTA = Elements(np.power(10, 15.10), 0.91, 6.5e13)
Y_HEDTA = Elements(np.power(10, 14.49), 0.66, 1.1e14)


k_edta = [np.power(10, -2.00), np.power(10, -2.67),
          np.power(10, -6.16), np.power(10, -10.26)]

k_dcta = [np.power(10, -2.40), np.power(10, -3.55),
          np.power(10, -6.14), np.power(10, -11.70)]

k_hedta = [np.power(10, -3.23), np.power(10, -5.50),
           np.power(10, -10.02)]
```

These variables are where the sliders/dropdowns/etcetera come from. Can adjust the min/max, options, add more parameters if needed.

```
# Input parameters.
```

```
init_conds = ['La', 'Nd', 'EDTA', 10.0, 10.0, 5.0, 1.0, 30000, 30, 0.5, 9.5, 1.0, 17.0, 13.0, 
              'Changing', 'odeint', 'No']

LREE_type = widgets.Dropdown(options=['La', 'Pr', 'Nd', 'Gd', 'Y'],
                        value='La',
                        description='LREE')

HREE_type = widgets.Dropdown(options=['Pr', 'Nd', 'Gd', 'Y'],
                        value='Y',
                        description='HREE')

agent_type = widgets.Dropdown(options=['EDTA', 'HEDTA', 'DCTA'],
                         value='EDTA',
                         description='Chelant:')


light = widgets.FloatSlider(value=5.0, min=0, max=50.0, description=('$[LREE]$'))

heavy = widgets.FloatSlider(value=5.0, min=0, max=50.0, description=('$[HREE]$'))

agent = widgets.FloatSlider(value=5.0, min=0, max=100.0, description=('$[Chelant]$'))

H = widgets.FloatSlider(value=1.0, min=0, max=10, description='$[H]$')

time = widgets.FloatSlider(value=10800, min=0, max=60000, description='Time (s)')

capacity = widgets.FloatSlider(value=10.78, min=0, max=50.0, description ='$Q~(mol/m^3)$', \
                               style={'description_width': 'initial'})

guess_1a = widgets.IntSlider(value=1.0, min=0, max=10.0, description='$[LREE-Complex]$', \
                             style={'description_width': 'initial'})

guess_1b = widgets.IntSlider(value=9.5, min=0, max=10.0, description='$[HREE-Complex]$', \
                             style={'description_width': 'initial'})

guess_1c = widgets.IntSlider(value=1.0, min=0, max=10.0, description='$[No~Complex]$', \
                             style={'description_width': 'initial'})

guess_2a = widgets.IntSlider(value=17.0, min=0, max=30.0, description='$[Membrane~LREE~Ion]$',
                             style={'description_width': 'initial'})

guess_2b = widgets.IntSlider(value=13.0, min=0, max=30.0, description='$[Membrane~HREE~Ion]$',
                             style={'description_width': 'initial'})

guess_2c = widgets.IntSlider(value=2.0, min=0, max=30.0, description='$[Membrane~Chelant]$', \
                             style={'description_width': 'initial'})
```

```
guess_2d = widgets.IntSlider(value=1.25, min=0, max=30.0, description='$[Membrane~H]$', \
                             style={'description_width': 'initial'}, tooltips='hello')

delta_H = widgets.ToggleButtons(
    options=['Changing', 'Not Changing'],
    description='$[H]$',
    disabled=False,
    button_style='info',
    tooltips=['Concentration of H is changing over time', 'Concentration of H is not changing 
    icons=['check']*2)

ode_solver = widgets.ToggleButtons(
    options=['odeint', 'solve_ivp'],
    description='Type of ODE Solver: ',
    disabled=False,
    button_style='danger',
    tooltips=['Simple solver of ordinary differential equations', 'Solves ODEs as initial valu
    icons=['check']*2,
    style={'description_width': 'initial'})

interp = widgets.ToggleButtons(
    options=['No', 'Yes'],
    description='Interpolation ',
    disabled=True,
    button_style='warning',
    tooltips=['Do not display 2-D interpolation model', 'Display 2-D interpolation model'],
    icons=['check']*2,
    style={'description_width': 'initial'})

angle1 = widgets.IntSlider(value=10, min=0, max=90, description='Rotate Vertically', step=10, 
                             style={'description_width': 'initial'})

angle2 = widgets.IntSlider(value=120, min=0, max=360, description='Rotate Horizontally', step=
                             style={'description_width': 'initial'})

i_time = widgets.FloatSlider(value=10700, min=10400, max=22000, description='Time (s)')

i_capacity = widgets.FloatSlider(value=24.0, min=20, max=30.0, description ='$Q~(mol/m^3)$', \
                               style={'description_width': 'initial'})
```

The main function takes in all 19 adjustable parameters. Its main purpose is to return the computed separation factor and plot. It contains nested functions which will solve the Takahashi equations.

```
##=========================================================
## MAIN FUNCTION
##=========================================================

def conditions(LREE_type, HREE_type, agent_type, light, heavy, agent, H, time, capacity, \
                guess_1a, guess_1b, guess_1c, guess_2a, guess_2b, guess_2c, guess_2d, delta_H, 


    '''
    Conditions(LREE_type, .., capacity) produces the separation factor and plot for a binary s

    It is attempting to replicate the Takahashi model. Proceeds to solve equations 2-6 and 7-9
    Employs odeint/solve_ivp for the differential equations.
    '''

    # Set each initial condition to its input value.

    init_conds[0] = LREE_type
    init_conds[1] = HREE_type
    init_conds[2] = agent_type
    init_conds[3] = light
    init_conds[4] = heavy
    init_conds[5] = agent
    init_conds[6] = H
    init_conds[7] = time
    init_conds[8] = capacity
    init_conds[9] = guess_1a
    init_conds[10] = guess_1b
    init_conds[11] = guess_1c
    init_conds[12] = guess_2a
    init_conds[13] = guess_2b
    init_conds[14] = guess_2c
    init_conds[15] = guess_2d
    init_conds[16] = delta_H
    init_conds[17] = ode_solver
    init_conds[18] = interp
```

These lines just check for various iterations where different REEs or chelating agents are selected. Adjusts the constants used accordingly.

```
# Combinations of LREE/HREE/Chelant.

    if (init_conds[0] == 'La') and (init_conds[1] == 'Pr'):
        if (init_conds[2] == 'EDTA'):
            light = La_EDTA
            heavy = Pr_EDTA
        elif (init_conds[2] == 'DCTA'):
            light = La_DCTA
            heavy = Pr_DCTA
        elif (init_conds[2] == 'HEDTA'):
            light = La_HEDTA
            heavy = Pr_HEDTA

    elif (init_conds[0] == 'La') and (init_conds[1] == 'Nd'):
```

```
    if (init_conds[2] == 'EDTA'):
        light = La_EDTA
        heavy = Nd_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = La_DCTA
        heavy = Nd_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = La_HEDTA
        heavy = Nd_HEDTA

elif (init_conds[0] == 'La') and (init_conds[1] == 'Gd'):
    if (init_conds[2] == 'EDTA'):
        light = La_EDTA
        heavy = Gd_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = La_DCTA
        heavy = Gd_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = La_HEDTA
        heavy = Gd_HEDTA

elif (init_conds[0] == 'La') and (init_conds[1] == 'Y'):
    if (init_conds[2] == 'EDTA'):
        light = La_EDTA
        heavy = Y_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = La_DCTA
        heavy = Y_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = La_HEDTA
        heavy = Y_HEDTA

elif (init_conds[0] == 'Pr') and (init_conds[1] == 'Nd'):
    if (init_conds[2] == 'EDTA'):
        light = Pr_EDTA
        heavy = Nd_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = Pr_DCTA
        heavy = Nd_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = Pr_HEDTA
        heavy = Nd_HEDTA

elif (init_conds[0] == 'Pr') and (init_conds[1] == 'Gd'):
    if (init_conds[2] == 'EDTA'):
        light = Pr_EDTA
        heavy = Gd_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = Pr_DCTA
        heavy = Gd_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = Pr_HEDTA
        heavy = Gd_HEDTA

elif (init_conds[0] == 'Pr') and (init_conds[1] == 'Y'):
    if (init_conds[2] == 'EDTA'):
        light = Pr_EDTA
        heavy = Y_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = Pr_DCTA
        heavy = Y_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = Pr_HEDTA
        heavy = Y_HEDTA

elif (init_conds[0] == 'Nd') and (init_conds[1] == 'Gd'):
    if (init_conds[2] == 'EDTA'):
        light = Nd_EDTA
        heavy = Gd_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = Nd_DCTA
        heavy = Gd_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = Nd_HEDTA
        heavy = Gd_HEDTA

elif (init_conds[0] == 'Nd') and (init_conds[1] == 'Y'):
    if (init_conds[2] == 'EDTA'):
        light = Nd_EDTA
        heavy = Y_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = Nd_DCTA
        heavy = Y_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = Nd_HEDTA
        heavy = Y_HEDTA

elif (init_conds[0] == 'Gd') and (init_conds[1] == 'Y'):
    if (init_conds[2] == 'EDTA'):
        light = Gd_EDTA
        heavy = Y_EDTA
    elif (init_conds[2] == 'DCTA'):
        light = Gd_DCTA
        heavy = Y_DCTA
    elif (init_conds[2] == 'HEDTA'):
        light = Gd_HEDTA
        heavy = Y_HEDTA

else:
    return ['Retry']

# Alpha variable dependent on chelating agent.

if (init_conds[2] == 'EDTA') or (init_conds[2] == 'DCTA'):

    if init_conds[2] == 'EDTA':
        k1 = k_edta[0]
        k2 = k_edta[1]
        k3 = k_edta[2]
        k4 = k_edta[3]

        alpha = (k1*k2*k3*k4)/(k1*k2*k3*k4 + \
                               k1*k2*k3*init_conds[6] + \
                               k1*k2*np.power(init_conds[6], 2) + \
                               k1*np.power(init_conds[6], 3) + \
                               np.power(init_conds[6], 4))

    else:

        k1 = k_dcta[0]
        k2 = k_dcta[1]
        k3 = k_dcta[2]
        k4 = k_dcta[3]

        alpha = (k1*k2*k3*k4)/(k1*k2*k3*k4 +                     \
                               k1*k2*k3*init_conds[6] +          \
                               k1*k2*np.power(init_conds[6], 2) + \
                               k1*np.power(init_conds[6], 3) +    \
                               np.power(init_conds[6], 4))

elif (init_conds[2] == 'HEDTA'):

    alpha = (k_hedta[0]*k_hedta[1]*k_hedta[2])/           \
            (k_hedta[0]*k_hedta[1]*k_hedta[2]         +   \
             k_hedta[0]*k_hedta[1]*init_conds[6]      +   \
             k_hedta[0]*np.power(init_conds[6], 2)    +   \
             np.power(init_conds[6], 3))
```

This is the equations 2-6 solver. It takes in a set of three guess values for LREE-Complex, HREE-Complex, and No-Complex. Sets the left side of the three equations to be zero and will find solutions to values that will satisfy the three equations. After the solutions are obtained from scipy.optimize.fsolve, effective light/heavy computed values can be used for equations 7-9.

```
def eq_26(guess_26):
        '''
        eq_26 takes in an array of 3 initial guess values provided by the user,
        and returns computed concentrations of LREE-Chelant, HREE-Chelant, and Chelant-NonComp

        '''

        light_complex = guess_26[0]
        heavy_complex = guess_26[1]
        no_complex = guess_26[2]

        eq_26_sol = np.zeros(3)

        eq_26_sol[0] = init_conds[3] - (light_complex / (light.stab * alpha * no_complex) + lig
        eq_26_sol[1] = init_conds[4] - (heavy_complex / (heavy.stab * alpha * no_complex) + hea
        eq_26_sol[2] = init_conds[5] - (light_complex + heavy_complex + no_complex)

        return eq_26_sol

    guess_1 = np.array([init_conds[9], init_conds[10], init_conds[11]])

    # Solve using guess values.
    te_sol = fsolve(eq_26, guess_1)

    # Solve eqn 2 for each REE ion using solution from eq_26.
    light_computed = te_sol[0]/(light.stab*(alpha*te_sol[2]))
    heavy_computed = te_sol[1]/(heavy.stab*(alpha*te_sol[2]))
```

Similar to eq_26 function, the eq_79 function will take in a set of 4 guess values for membrane ion quantities and returns computed membrane ion quantity values the same way as eq_26 does.

```
def eq_79 (mem_guess):

        '''
        eq_79 takes in an array of 4 initial guess values provided by the user,
        and returns computed concentrations of membrane quantities for Takahashi's equations 7

        '''

        light_mem = mem_guess[0]
        heavy_mem = mem_guess[1]
        Na_mem = mem_guess[2]     # EDTA ligand equivalence sodium ion.
        H_mem = mem_guess[3]

        compute_79 = np.zeros(4)

        compute_79[0] = light_mem - (light.K_hydro * \
                                     np.power(H_mem, 3) * \
                                     (light_computed/(np.power(init_conds[6],3))) ** 0.8)
        compute_79[1] = heavy_mem - (heavy.K_hydro * \
                                     np.power(H_mem, 3) * \
                                     (heavy_computed/(np.power(init_conds[6],3))) ** 0.8)

        compute_79[2] = Na_mem - (0.32 * H_mem * (init_conds[5]/init_conds[6]))

        compute_79[3] = init_conds[8] - (3 * (light_mem+heavy_mem) + Na_mem + H_mem)

        return compute_79

    guess_2 = np.array([init_conds[12], init_conds[13], init_conds[14], init_conds[15]])

    eq_79_sol = fsolve(eq_79, guess_2)

    # Uncomment these print statements to verify output values.
    #print('Concentration of LaEDTA =', te_sol[0])
    #print('Concentration of NdEDTA ion =', te_sol[1])
    #print('Concentration of EDTA non-complex =', te_sol[2])
    #print('solution: ', te_sol, '\n', 'equations:', eq_26(te_sol))

    #print('Concentration of La in membrane=', eq_79_sol[0])
    #print('Concentration of Nd in membrane=', eq_79_sol[1])
    #print('Concentration of EDTA in membrane =', eq_79_sol[2])
    #print('Concentration of H in membrane =', eq_79_sol[3])
    #print('solution: ', eq_79_sol, '\n', 'equations:', eq_79(eq_79_sol))
```

These two functions deal with equation 16 in the Takahashi model. The first function is intended for the odeint solver, whereas the latter is for solve_ivp solver. They both take the 4 computed membrane values returned from eq_79 and use them to calculate the concentrations of the 4 ions w.r.t. time.

```
    def RKG_Takahashi(z, t):

        '''
        Solves the Takahashi differential eqn 16 simultaneously via odeint.

        z[0] = C_La
        z[1] = C_Nd
        z[2] = C_Na
        z[3] = C_H

        f1 = dC_La/dt
        f2 = dC_Nd/dt
        f3 = dC_Na/dt
        f4 = dC_H/dt

        '''

        light_mem, heavy_mem, Na_mem, H_mem = z
    f1 = surface_area * ((valence * light.diff_coeff * light_mem) / \
                         (valence**2 * light.diff_coeff * light_mem + \
                          valence**2 * heavy.diff_coeff * heavy_mem + \
                          1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/-V

    f2 = surface_area * ((valence * heavy.diff_coeff * heavy_mem) / \
                         (valence**2 * heavy.diff_coeff * heavy_mem + \
                          valence**2 * light.diff_coeff * light_mem + \
                          1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/-V

    f3 = surface_area * ((1.9e15 * Na_mem) / (valence**2 * light.diff_coeff * light_mem + \
                                              valence**2 * heavy.diff_coeff * heavy_mem + \
                                              1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/

    f4 = surface_area * ((1.1e16 * H_mem) / (valence**2 * light.diff_coeff * light_mem + \
                                             valence**2 * heavy.diff_coeff * heavy_mem + \
                                             1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/

    if (init_conds[16] == 'Not Changing'):
        f4 = 0

    return [f1, f2, f3, f4]

def RKG_Takahashi2(t, z):

    '''
    Solves the Takahashi differential eqn 16 simultaneously via solve_ivp.
    Notice that for solve_ivp timespan parameter is before z-matrix parameter.
    Same arguments can be used for both solve_ivp and odeint.

    z[0] = C_La
    z[1] = C_Nd
    z[2] = C_Na
    z[3] = C_H

    f1 = dC_La/dt
    f2 = dC_Nd/dt
    f3 = dC_Na/dt
    f4 = dC_H/dt

    '''

    light_mem, heavy_mem, Na_mem, H_mem = z

    f1 = surface_area * ((valence * light.diff_coeff * light_mem) / \
                         (valence**2 * light.diff_coeff * light_mem + \
                          valence**2 * heavy.diff_coeff * heavy_mem + \
                          1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/-V

    f2 = surface_area * ((valence * heavy.diff_coeff * heavy_mem) / \
                         (valence**2 * heavy.diff_coeff * heavy_mem + \
                          valence**2 * light.diff_coeff * light_mem + \
                          1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/-V

    f3 = surface_area * ((1.9e15 * Na_mem) / (valence**2 * light.diff_coeff * light_mem + 
                                              valence**2 * heavy.diff_coeff * heavy_mem + 
                                              1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/

    f4 = surface_area * ((1.1e16 * H_mem) / (valence**2 * light.diff_coeff * light_mem + \
                                             valence**2 * heavy.diff_coeff * heavy_mem + \
                                             1.9e15 * Na_mem + 1.1e16 * H_mem)) * I/F * 1/

    if (init_conds[16] == 'Not Changing'):
        f4 = 0

    return [f1, f2, f3, f4]
```

tspan is the timespan interval. The update_sol functions are used to call on the respective ODE solver. SF values are calculated by utilizing the terminal data values of from the ODE solver solutions.

```
# Timespan.
    tspan = np.linspace(0, init_conds[7], 100)  # 100 => 100 datapoints

    # Update functions for each solver.
    update_sol = odeint(RKG_Takahashi, ([eq_79_sol[0], eq_79_sol[1], eq_79_sol[2], eq_79_sol[3
    update_sol2 = solve_ivp(RKG_Takahashi2, [0, init_conds[7]],                       \
                           ([eq_79_sol[0], eq_79_sol[1], eq_79_sol[2], eq_79_sol[3]]), \
                            method='RK45', t_eval = np.linspace(0, init_conds[7], 100))

    # Concentration values.
    J_light = update_sol[99][0]
    J_heavy = update_sol[99][1]
    J_light2 = update_sol2.y[0][99]
    J_heavy2 = update_sol2.y[1][99]

    global SF

    # Calculate SF.
    SF = (J_light/surface_area*V)/(J_heavy/surface_area*V)
    SF2 = (J_light2/surface_area*V)/(J_heavy2/surface_area*V)
```

This segment of the conditions function creates the graph and plots the computed values onto it. The print statements prior to the return statement will output the determined SF value to the screen.

```
# Plot the computed concentrations.
```

```
    plt.rcParams['figure.figsize'] = [14, 8]

    if (init_conds[17] == 'odeint'):
        plt.plot(tspan, update_sol[:,0], 'crimson')
        plt.plot(tspan, update_sol[:,1], 'deepskyblue')
        plt.plot(tspan, update_sol[:,2], 'lightpink')
        plt.plot(tspan, update_sol[:,3], 'mediumseagreen')

    else:
        plt.plot(update_sol2.t, update_sol2.y[0], 'crimson')
        plt.plot(update_sol2.t, update_sol2.y[1], 'deepskyblue')
        plt.plot(update_sol2.t, update_sol2.y[2], 'lightpink')
        plt.plot(update_sol2.t, update_sol2.y[3], 'mediumseagreen')


    plt.xlabel('$Time (s)$')
    plt.ylabel('$Concentration~(mol/m^3)$')
    plt.legend([init_conds[0], init_conds[1], init_conds[2], 'H'], loc=1, prop={'size': 20})
    plt.title(('Concentration Profile'))

    # Print SF values to screen.
    if (init_conds[17] == 'odeint'):
        print("Separation Factor of %s" % init_conds[0] + " over %s = " % init_conds[1]  + "\03
    else:
        print("Separation Factor of %s" % init_conds[0] + " over %s = " % init_conds[1]  + "\03


    return
```

Here is where the interactive input sliders/dropdowns are linked to the conditions function. In addition, this is where you can arrange them to preference.

```
# Interactive input.
out = widgets.interactive_output(conditions, {"LREE_type":LREE_type, "HREE_type":HREE_type,
                                              "agent_type":agent_type, "light":light,
                                              "heavy":heavy, "agent":agent, "H":H, "time":time
                                              "capacity":capacity, "guess_1a":guess_1a,
                                              "guess_1b":guess_1b, "guess_1c":guess_1c,
                                              "guess_2a":guess_2a, "guess_2b":guess_2b,
                                              "guess_2c":guess_2c, "guess_2d":guess_2d,
                                              "delta_H":delta_H, "ode_solver":ode_solver, "int

# Arrangement of input sliders/dropdowns/etcetera.
column1 = widgets.VBox([LREE_type, HREE_type, agent_type])
column2 = widgets.VBox([light, heavy, agent])
column3 = widgets.VBox([H, time, capacity])
column4 = widgets.VBox([guess_1a, guess_1b, guess_1c])
column5 = widgets.VBox([guess_2a, guess_2b, guess_2c, guess_2d])
column6 = widgets.VBox([delta_H, ode_solver])
```

```
column7 = widgets.VBox([interp])
box_layout = widgets.Layout(display='flex',
                            flex_flow='row',
                            align_items='center',
                            width='100%')

tab1 = widgets.HBox([column1, column2, column3])
tab2 = widgets.HBox([column4, column5])
tab3 = widgets.HBox([column6, column7])
tabs = widgets.Tab(children=[tab1, tab2, tab3])
tabs.set_title(0, 'Parameters')
tabs.set_title(1, 'Guess')
tabs.set_title(2, 'Additional Options')

# Show to screen.
display(tabs, out)
```

The interpolant function can be used as a tool to highlight how SF is influenced by incrementing two different parameters visually. Currently it is only for time and ion exchange capacity because those two parameters are the most uncertain for the model. It can be noted here that an accuracy discrepancy function to compare SF values with the original paper would be more optimal, however, since time and Q quantities are not certain the SF values can vary dramatically as shown by the interpolant.

```
def interpolant(LREE_type, HREE_type, agent_type, light, heavy, agent, H, i_time, i_capacity, 
                guess_1a, guess_1b, guess_1c, guess_2a, guess_2b, guess_2c, guess_2d, delta_H, 
                ode_solver, interp, angle1, angle2):

    '''
    Interpolant(LREE_type, ..., angle2) uses cublic spline interpolation to calculate values
    in between computed SF values from the conditions function. Outputs the results onto the s
    '''

    # Setup the 2-D interpolation.
    ## One dimension to investigate is time (x-axis).
    ### Likewise Q (ion exchange capacity) also should be explored (y-axis).
    x2 = np.arange(i_time, 22000, 1000)
    y2 = np.arange(i_capacity, 30, 0.5)
    xx, yy = np.meshgrid(x2, y2)
    zz = np.empty_like(xx)
    for i in range(np.size(zz,0)):
        for j in range(np.size(zz,1)):
            zz[i,j] = conditions(LREE_type, HREE_type, agent_type, light, heavy, agent, H, xx[
                                 guess_1a, guess_1b, guess_1c, guess_2a, guess_2b, guess_2c, g
                                 ode_solver, 'Yes')
    f2 = interp2d(x2, y2, zz, kind='cubic')
```

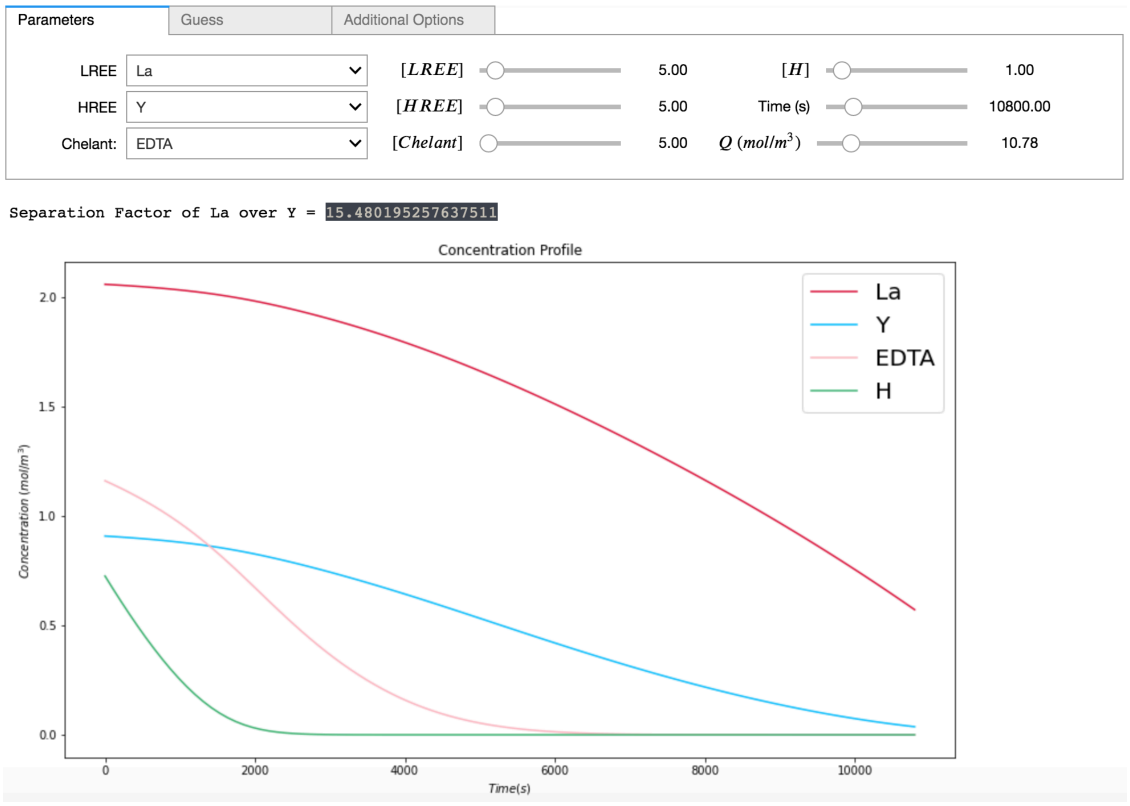
interp2d is the actual interpolation solver. The rest of the function plots and uses the interpolation solver to generate the graph. Similar to conditions. This code was adapted from Hedengren's data science course on interpolation.[9]

```
# Plot the interpolated values.
    fig = plt.figure(figsize=(14,10))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(xx,yy,zz, color='red', label='Computed SF')
    xe = np.arange(i_time, 22000, 1000)
    ye = np.arange(i_capacity, 30, 0.5)
    xxe, yye = np.meshgrid(xe, ye)
    fe = np.empty_like(xxe)
    for i in range(np.size(fe,0)):
        for j in range(np.size(fe,1)):
            fe[i,j] = f2(xxe[i,j],yye[i,j])
    ax.plot_surface(xxe,yye,fe,color='deepskyblue',alpha=0.75)

    plt.title('Comparison of Time and Q on ' + \
              "Separation Factor of %s" % LREE_type + " over %s " % HREE_type)
    ax.set_xlabel('$Time (s)$')
    ax.set_ylabel('Ion Exchange Capacity $(mol/m^3)$')
    ax.set_zlabel('SF')
    ax.view_init(elev=angle1, azim=angle2)
    plt.legend()
    plt.show()

    return
```

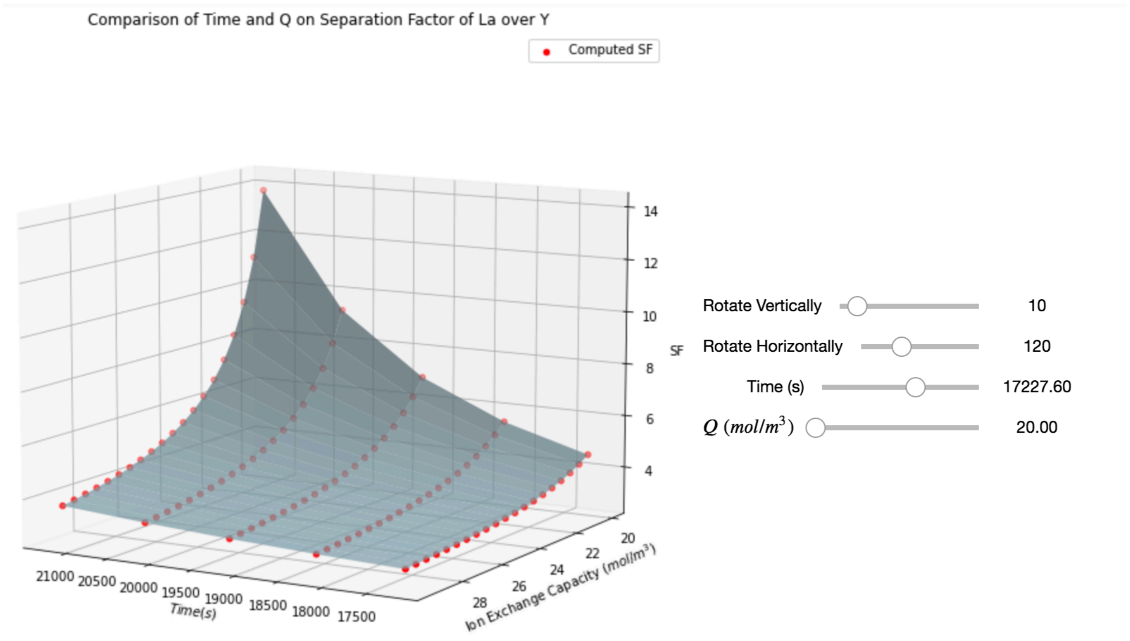
1.0.6 Interpretations



LREE La
HREE Y
Chelant: EDTA
$[LREE]$ 5.00
$[HREE]$ 5.00
$[Chelant]$ 5.00
$[H]$ 1.00
Time (s) 10800.00
$Q\ (mol/m^3)$ 10.78

```
Separation Factor of La over Y = 15.480195257637511
```

Plot 1. Binary La-Y separation, REE/EDTA = 1:1.



Rotate Vertically 10
Rotate Horizontally 120
Time (s) 17227.60
$Q\ (mol/m^3)$ 20.00

Plot 2. 2-D Interpolation of separation factor, time, and Q. The model as of the latest update provides a baseline depiction of the trends relating to stability constants and other REE-

ED parameters. Plot 1 displays the calculated separation factor result of 15.48 for a binary La-Y separation with REE/EDTA 1:1 at 3 hours timespan and Q = 10.78. This is in close agreement with *Takahashi et al.* 's reported value of 15.5 for the same separation experimentally. Separation factor calculation results can have many applications. For example, how the timespan and Q parameters affect the computation is shown by Plot 2. With this parameter variability and uncertainty in mind, an automated function to detect accuracy discrepancy with respect to *Takahashi et al.* 's original results would be a next step in advancing the model.

Ultimately, the current model is still a work in progress. It is hopeful that successor iterations of the model will be further capable of perfecting simulating REE-ED proccesses.

1.0.7 Reference List

[1] Sauber, M., Issa, S., Mosadeghsedghi, S., Mortazavi, S.; ”The Feasibility of Separation of Light Heavy REE Using Electrodialysis”. (2019)

[2] Sauber, M. ”Year 5 REE Separation Year End Report,” (2020).

[3] Lyon, K. L., Utgikar, V. P., Greenhalgh, M. R.; Ind. Eng. Chem. Res. 2017, **56** (4), 1048–1056 (2017).

[4] Takahashi, H., Miwa, K., Kikuchi, K.; J. Ion Exchange, **4** (3), 183-193 (1994).

[5] VanBriesen, J. M., Small, M., Weber, C., Wilson, J.; ”Modelling of Pollutants in Complex Environmental Systems”, Vol. 2, p. 133-149. ILM Publications, Cambridge (2010).

[6] Mackey, J. L. (1960). ”A study of the rare-earth chelate stability constants of some aminopolyacetic acids.” accessed from https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=3796&context=rtd (April 2020).

[7] Takahashi, H., Miwa, K., Kikuchi, K.; J. Int. J. of The Soc. of Mat. Eng. for Resources, **1** (1), 132-130 (1993).

[8] Karraker, R. H., ”Stability constants of some rare-earth-metal chelates,” (1961). Retrospective Theses and Dissertations. 1969. http://lib.dr.iastate.edu/rtd/1969 (April 2020).

[9] Hedengren, J., ”APMonitor Begin Python Course,” (n.d.). https://apmonitor.com/che263/index.php/Main/HomePage (June 2020).

[10] Fountoulakis, K., ”University of Waterloo CS 370 Numerical Computation Coursenotes,” (2020).