# Full Description of Takahashi Model Progress (Python)

**Prepared by: Benny Jun Chen**  
**Prepared for: Water Management Section**

[Link to Model](https://colab.research.google.com/github/bjb2chen/Takahashi-Model/blob/master/REE_ED_Takahashi_Model.ipynb)

### 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 Natural Resources Canada REE workshops and year-end reports 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, Konstantin developed a strategy to optimize testing: employ a model to simulate the chelation-assisted competitive binding of REEs in ED processes.

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:

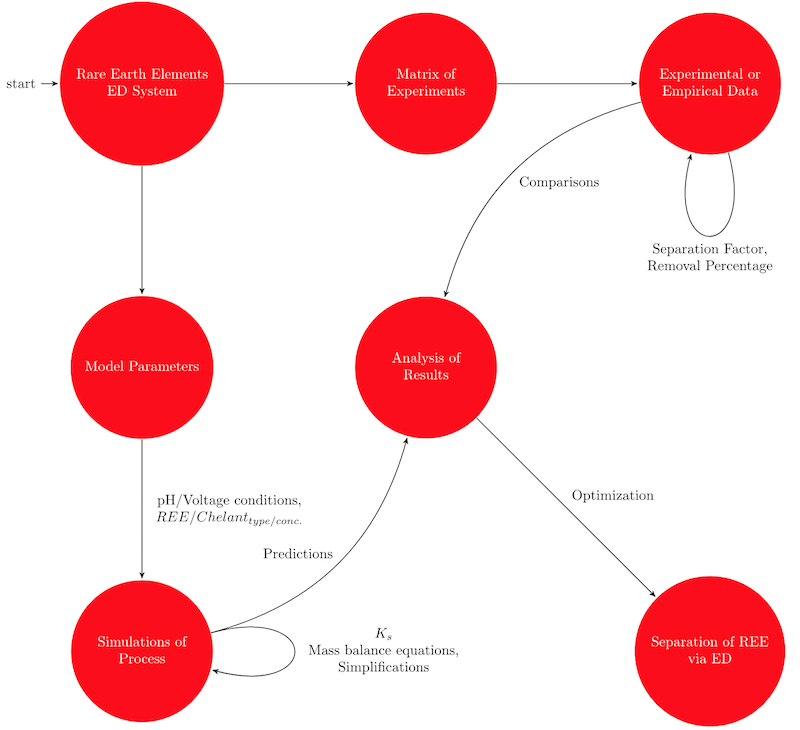
- The merit of the Takahashi model.

- Reproducibility of Takahashi et al.'s findings.

- Models/Chemometrics are viable ad-hoc tools in optimizing ED REE separation testing.

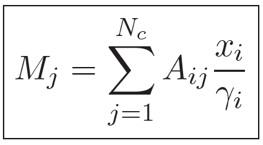
- Python/Jupyter notebooks can be selected for this application.

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



### Background Theory (multi-component systems and equilibrium)

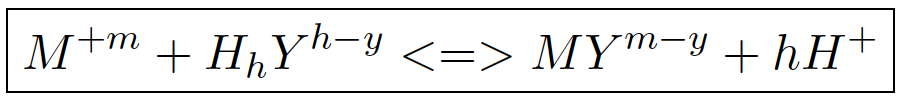
According to VanBriesen et al., mass balance equations can be written as:[5]



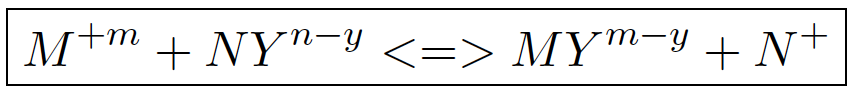
The total mass of the component is the sum of the quantity of the component in every relevant species.

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.

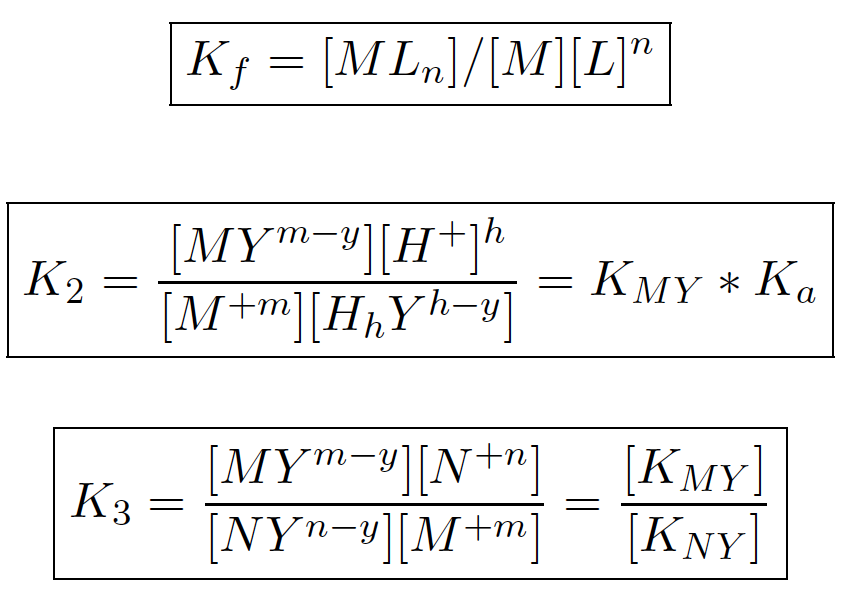


1. Equilibrium scenario where two metal ions compete.

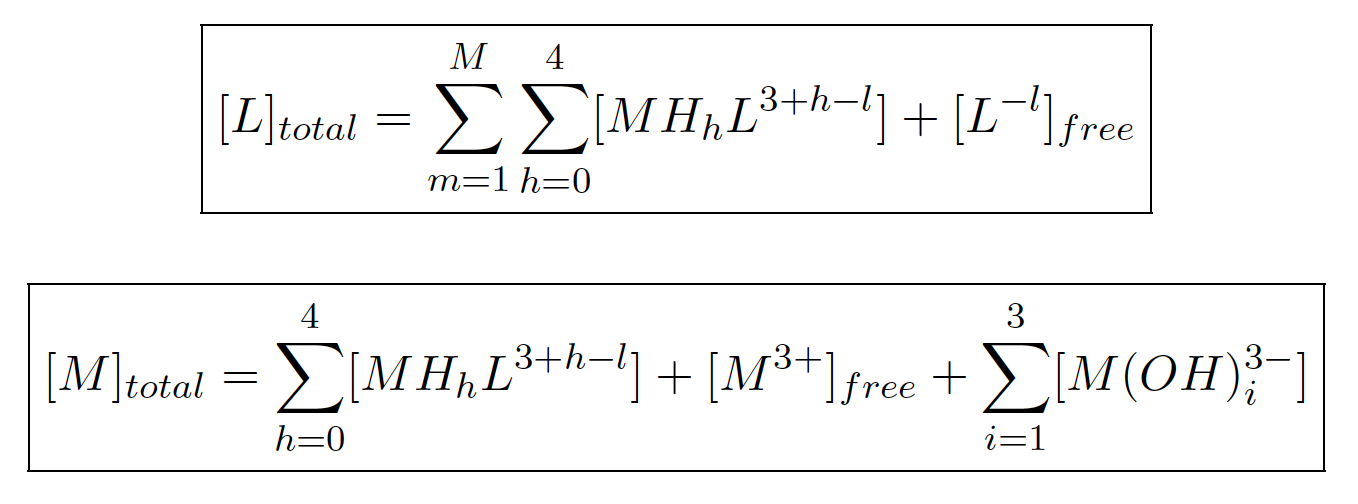


Note: M is for metal, Y or L is for chelating agent, and N is for second metal.

From these scenarios we can determine stability constants:



Mass Balance Equations (note: ignore hydroxide products):

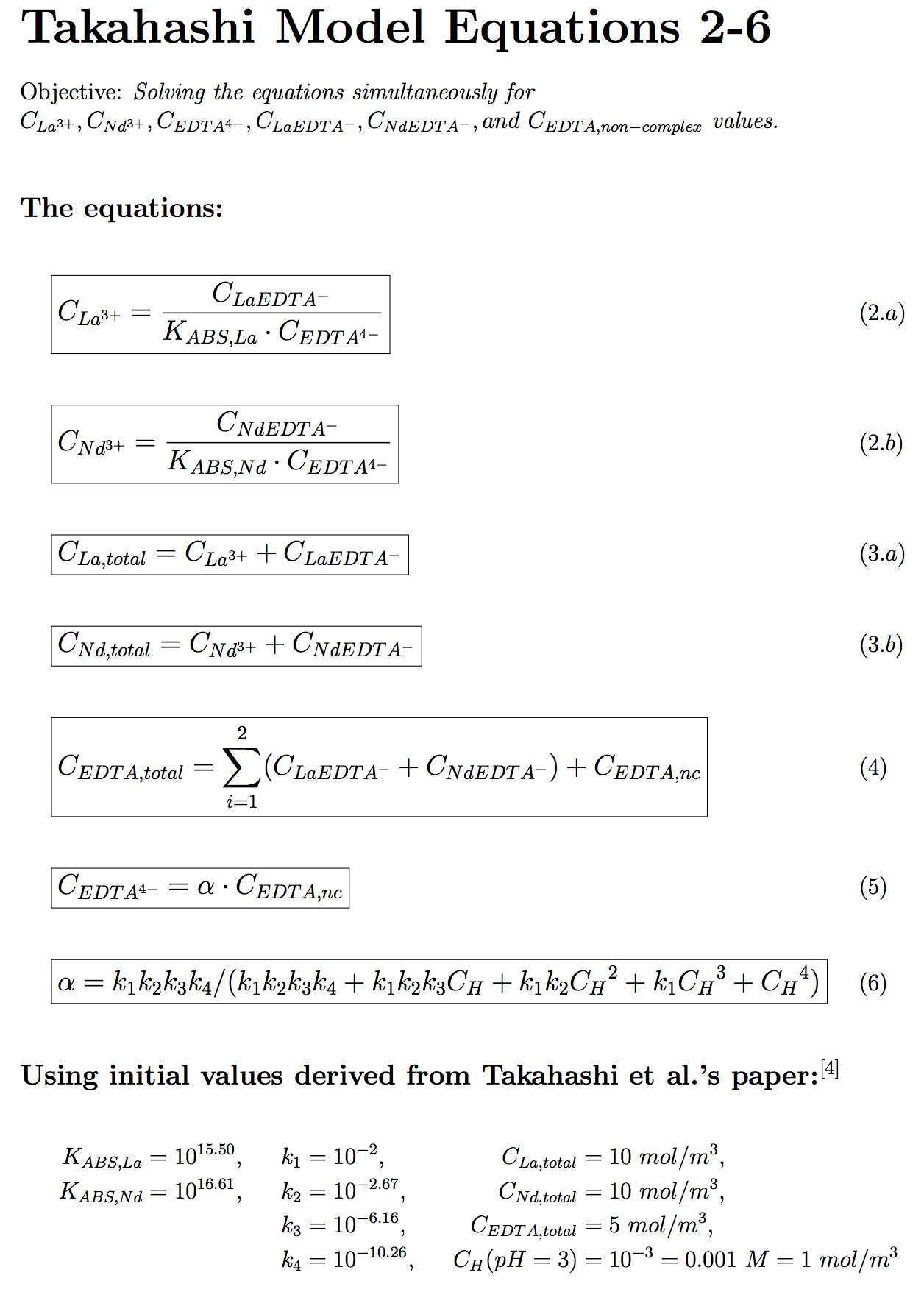


### The Takahashi Model

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

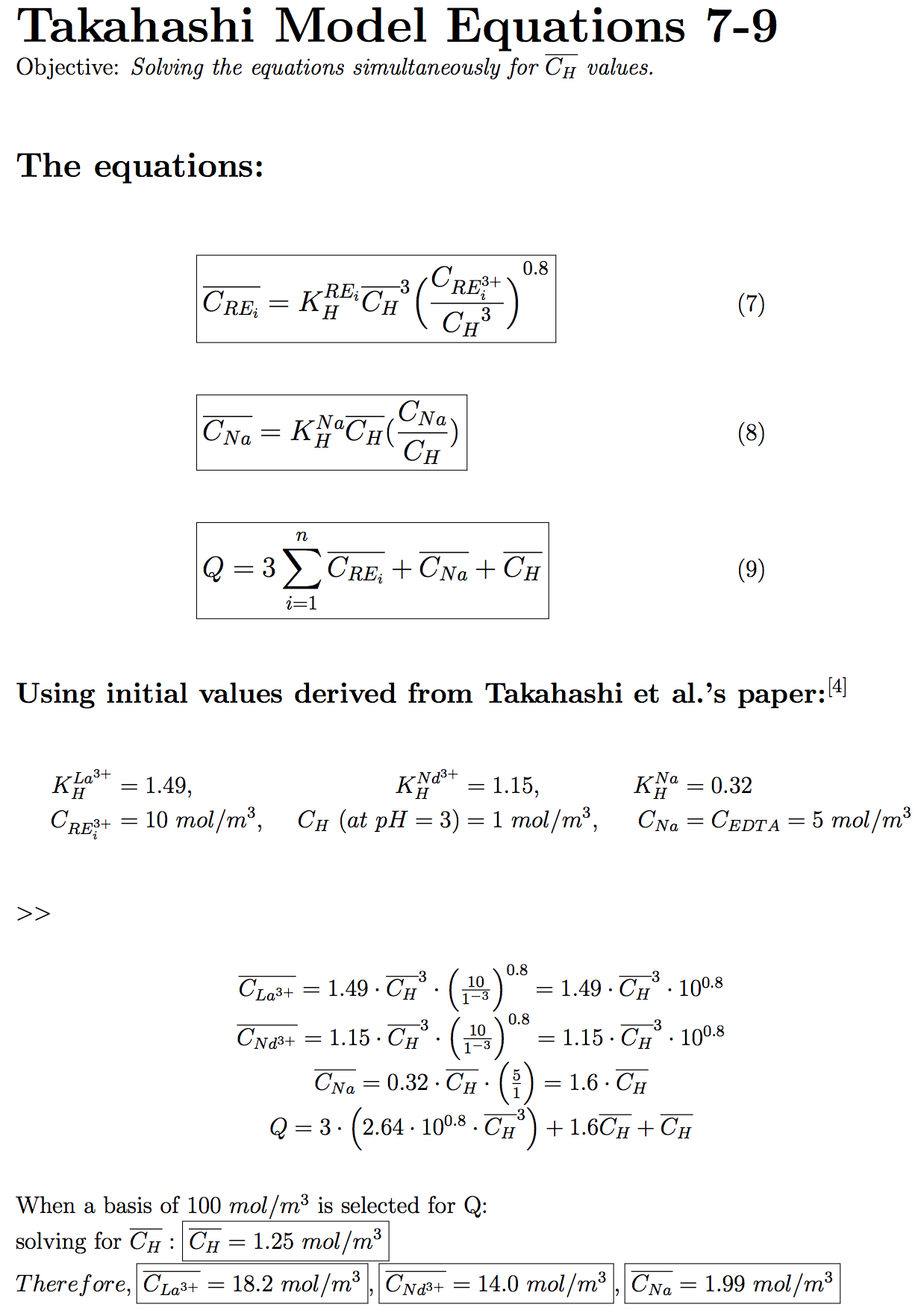
Let it be assumed as a and with situation.

The following Takahashi Model equations 2-6 are to be solved simultaneously:

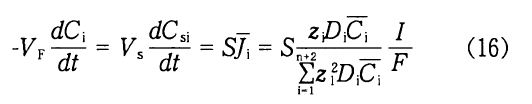


Once equations 2-6 are solved simultaneously, , , and values can be inputted into equations 7-9.

Example of Takahashi Model equations 7-9 solved simultaneously:



(Equations 1, 10-15 are all simplifications). By solving equations 2-9 simultaneously, solving of the following differential equation (Takahashi model equation 16) for the REE ions and can then proceed. Likewise, this can apply to any general binary and . Ultimately, separation factor (SF) can be obtained from those computed concentrations.



**Important note on Takahashi references**:

The second Takahashi et al. reference discusses the separation of yttrium from La-Y and neodymium from Nd-Gd-Y.[7] It investigated the changes in flux with respect to changing type of chelating agents (e.g. EDTA vs. lactic acid), ratio of chelating agents to REE, and pH. Separation factor was calculated by dividing the flux of the REE with higher permeation over the lower permeation REE (e.g. it would be or , ). The relationship between flux and concentrations is displayed in equation 16.

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

### Python and Jupyter Notebooks

Implementations of the model were originally attempted through alternative software like Excel or MATLAB. Maziar Sauber recommended Python as the ideal method to develop this model because of its very universal and powerful nature.

Jupyter Notebook is a software application that allows the exploration of python code, scientific computing, interactivty, visual output, and more--all in one elegant document. This was very appealing in the context of the model. The notebooks are being increasingly adopted throughout the Canadian technology, data science, and R&D industries.

There are multiple ways to access Jupyter Notebook:

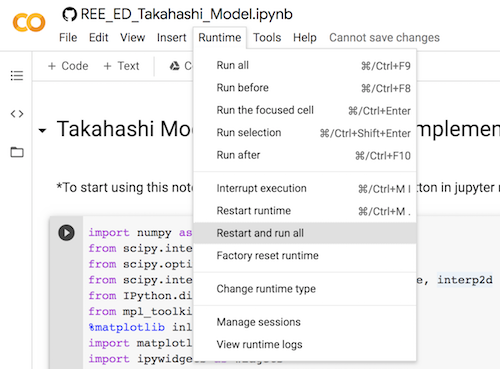
1. [Install Jupyter](https://jupyter.org/install)
2. [Google Colaboratory](https://colab.research.google.com)
3. [University Login - Syzygy](https://intro.syzygy.ca/getting-started/)

Once in the Jupyter environment, notebook files (file extension .ipynb) can be upload and downloaded easily. The model notebook file 'REE\_ED\_Takahashi\_Model.ipynb' was uploaded to Google Colab, and can be accessed through the following link:

<https://colab.research.google.com/github/bjb2chen/Takahashi-Model/blob/master/REE_ED_Takahashi_Model.ipynb>

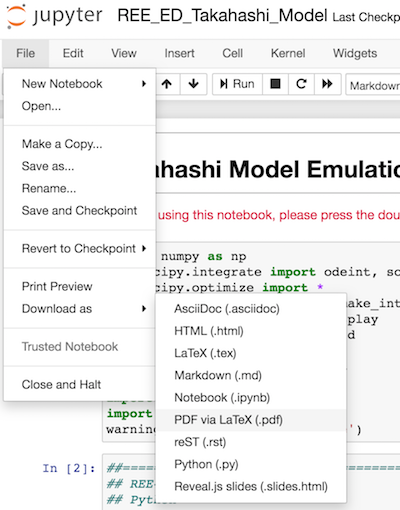
Python code can be written in code cells and executed/run either individually or all at once. To run one code cell, highlight it then press the 'Run' button. Running the entire notebook can be done by finding the 'Restart and Run All' option.

Figure 2. Running the model in Google Colaboratory.



Notebooks can be exported into a variety of files like PDF, LaTeX, Python, etcetera.

Figure 3. Options to download the notebook.



The following files are relevant to the model so far:

1. REE\_ED\_Takahashi\_Model.ipynb - Model notebook
2. REE\_ED\_Takahashi\_Model\_Description.ipynb - Model description notebook
3. REE\_ED\_Takahashi\_Model.py - Model python file
4. REE\_ED\_Takahashi\_Model\_Description.pdf - Model description in PDF
5. 10263977\_001\_TG\_CMIN13-21\_JA\_EN\_.docx - Takahashi Model original paper translated by CMIN

### 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. Matthew Hudder advised in the majority of the development of the model so far.

This code cell simply imports all relevant modules. %matplotlib inline ensures that the output plots are shown on screen as static images adjacent to the code. The 'inline' backend is for static plot images, whereas the 'notebook' backend is for interactivity in plots (however for an unknown reason using this backend resulted in blank plots and was therefore avoided). 'tk' backend can be used to output the plots into separate python windows.

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')

These are constant values from Takahashi paper. Elements class holds all sets of constants belonging to each respective element.

#==============================================  
# From Takahashi paper:  
#==============================================  
# Symbol | Meaning | Units   
#==============================================  
# (C) | Concentrations | [mol/m3]  
# (D) | Diffusion coefficient | [m2/s]  
# (F) | Faraday constant | [C/mol]  
# (I) | Current density | [A/m2]  
# (J) | Flux in membrane | [mol/(m2\*s)]  
# (Kabs) | Absolute stab. constant| [dm3/mol]  
# (Q) | Ion-exchange capacity | [mol/m3]  
# (S) | Surface area | [m2]  
# (Si) | Separation factor | [unitless]  
# (t) | Time | [s]  
# (V) | Volume of solution | [m3]  
  
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, 2.0, 1.25, \  
 '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 over time'],  
 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 value problem, RK45'],  
 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=10, \  
 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, ode\_solver, interp):  
   
   
 '''  
 Conditions(LREE\_type, .., capacity) produces the separation factor and plot for a binary separation.  
   
 It is attempting to replicate the Takahashi model. Proceeds to solve equations 2-6 and 7-9 via fsolve.  
 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-NonComplex ion.  
   
 '''  
   
 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) + light\_complex)  
 eq\_26\_sol[1] = init\_conds[4] - (heavy\_complex / (heavy.stab \* alpha \* no\_complex) + heavy\_complex)  
 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 7-9.  
   
 '''  
   
 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)

These print statements are useful in checking computed values from the eq\_26 and eq\_79 functions. Note that by setting the respective output values back as the input into eq\_26 or eq\_79, it should give solutions close to zero. Values close to zero indicate that fsolve solved correctly. Otherwise, if the values are not close to zero, then there should be an error. The for loop checks for an error situation in the guess values.

# 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)  
   
 for i in range(len(eq\_79\_sol)):  
 if eq\_79\_sol[i] == float(guess\_2[i]):  
 print('\033[93m' + 'Potential Error. Try adjusting the left column guess values')  
 continue

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/-V  
   
 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/-V  
   
 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/-V  
   
 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/-V  
   
 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]]), tspan)  
 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] + "\033[3;37;40m" + str(SF))  
 else:  
 print("Separation Factor of %s" % init\_conds[0] + " over %s = " % init\_conds[1] + "\033[3;37;40m" + str(SF2))  
   
   
 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, "interp":interp})  
  
# 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 screen.  
 '''  
   
 # 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[i,j], yy[i,j], \  
 guess\_1a, guess\_1b, guess\_1c, guess\_2a, guess\_2b, guess\_2c, guess\_2d, delta\_H,\  
 ode\_solver, 'Yes')  
 f2 = interp2d(x2, y2, zz, kind='cubic')

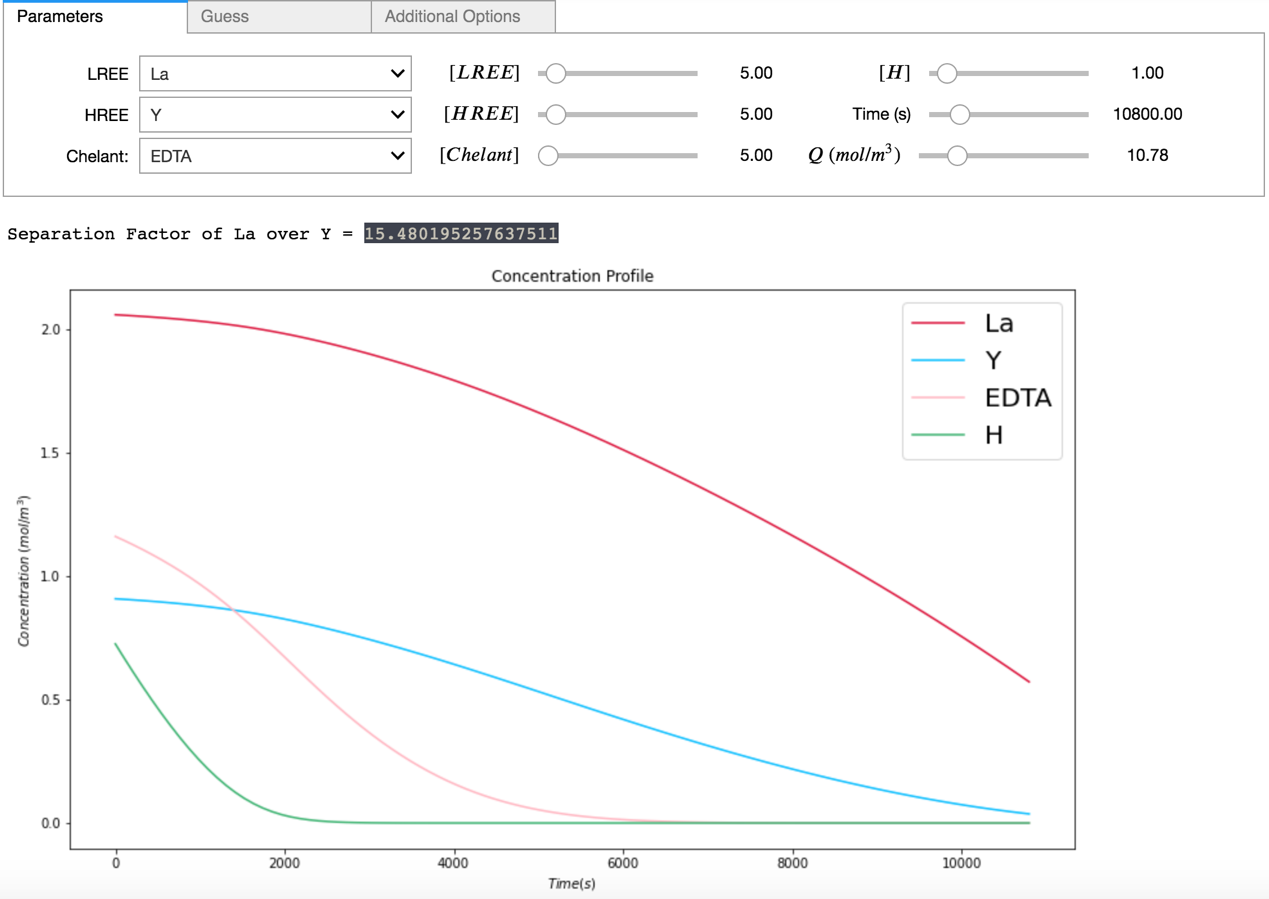
interp2d is the actual interpolation solver. The rest of the function plots and uses the interpolation solver to generate the graph. Similar to conditions. This code was adapted from Hedengren's data science course on interpolation.[9]

# 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

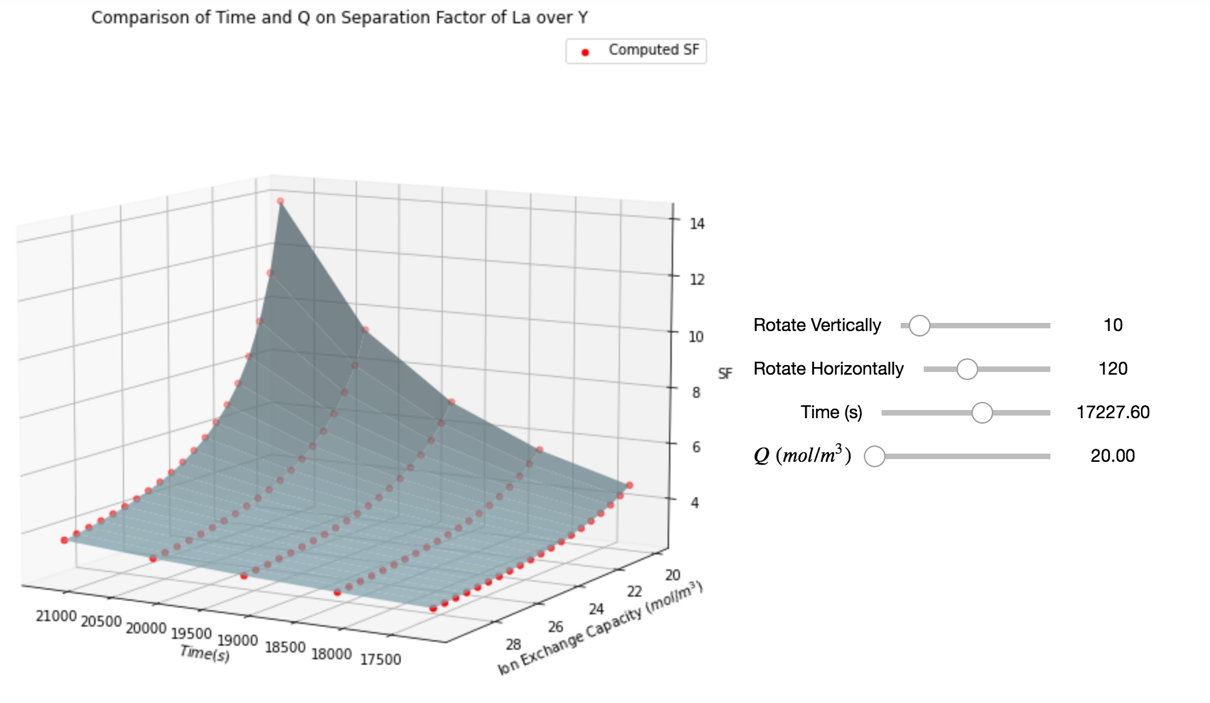
### 

### Interpretations

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



#### 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.[7] 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 processes.

### 

### 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).