PHREEQC Tutorial

August 10, 2023

1 PHREEQC Guide

The following is meant to be a guide on how phreeqc works. The idea is to give a scenario, and break down the ways one can solve the issue using phreeqc. Giving and explaining how code works as well as showing how changes in initial parameters can shift the efficiency, accuracy and quality of the results.

Specifically I will breakdown and describe how one of my models works, my model on the Aragonite <—> Calcite transition, and programs which i sampled and used to create it.

2 The Code

The following is the PHREEQC input file for the Aragonite <—> Calcite transition

```
TITLE –Aragonite Conversion 2 Electric Boogaloo
################################################################################################
#This is another model for the conversion of Aragonite and Calcite this time using the
#rate equation,
#.
#. R = k(1 - SI_C)^n
#. R = k(1 - SI_A)^n

#From the paper by James G. Acker, Robert H. Byrne, Samuel Ben-Yaakov, Richard A. Feely
#and Peter R. Betzer
################################################################################################

SOLUTION 1
temp 25
pH 7
pe 4
redox pe
units mmol/L
density 1
-water 1 # kg

EQUILIBRIUM_PHASES 1
Aragonite 1.5 0
Calcite 1.0 0
CO2(g) -3.5 10
```

```
SAVE Solution 1
END

RATES
Aragonite
-start
10 k = 10^parm(1)
20 n = parm(2)
30 R = k * (1 - SR("Aragonite"))^n
40 SAVE R * time
-end
Calcite
-start
10 k = 10^parm(1)
20 n = parm(2)
30 R = k * (1 - SR("Calcite"))^n
40 SAVE R * time
-end

KINETICS 1
Aragonite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.4 2
-tol 1e-08
Calcite
-formula CaCO3 1
-m 0
-m0 0
-parms -7.3 1
-tol 1e-08
-step 500 day in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

USE Solution 1

SELECTED_OUTPUT 1
-file AragoniteDissolution5.sel
-high_precision true
-reset false
-step false
-time true
-pH true
-saturation_indices Aragonite Calcite
-kinetic_reactants Aragonite Calcite
```

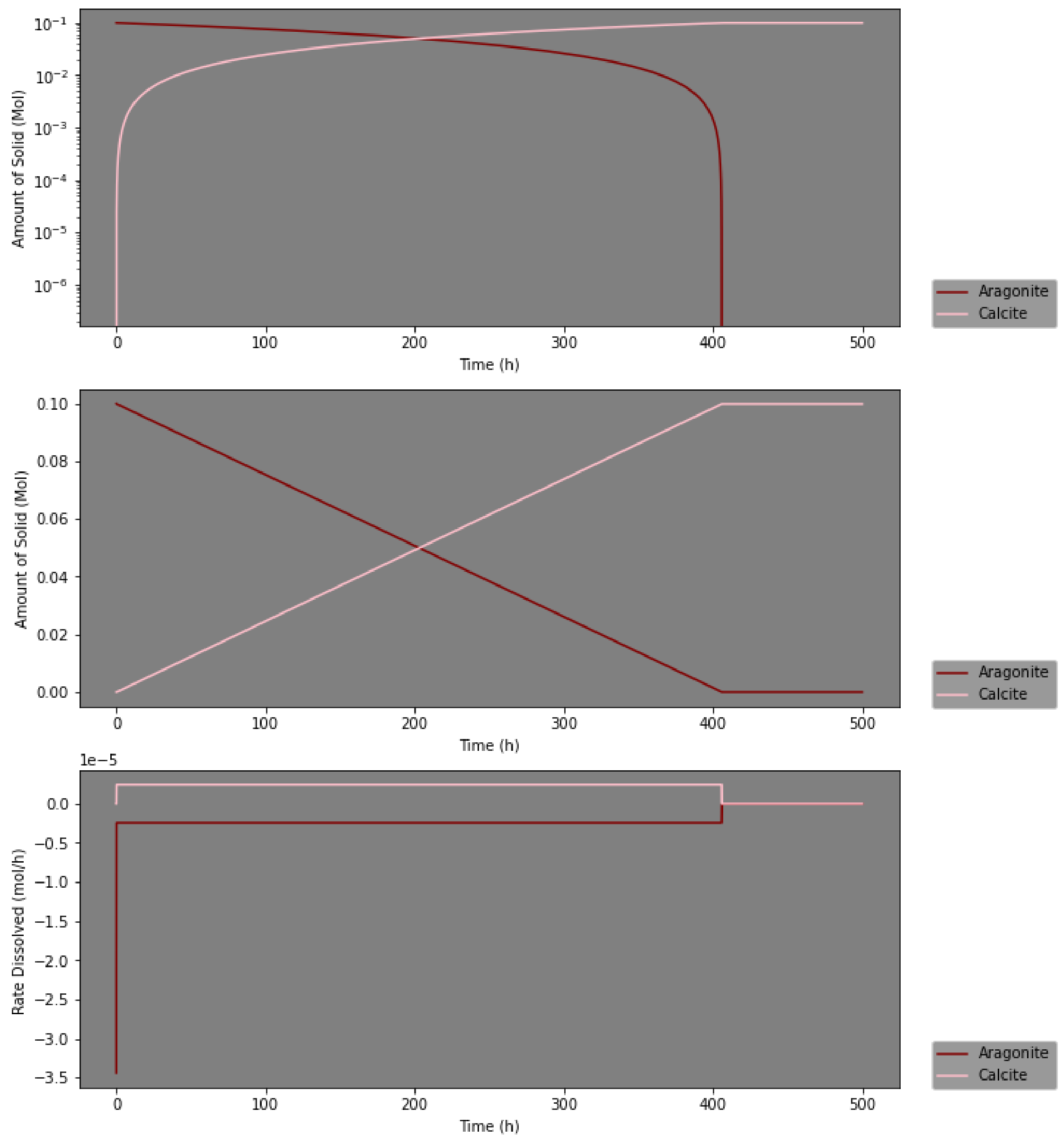
END

This code saves data for the amount of calcite and aragonite their saturations the changes per step and the time into an .sel file. We can plot this data in python.

[42]:
```
import numpy as np
import matplotlib.pyplot as plt

file_path = r'C:\Users\Jackt\OneDrive\MUN Year 3\Work Term 2023\Input␣
 ↪Files\AragoniteDissolution5.sel'

stepsize = 500 / 1000

# Read data from .sel file
data = np.loadtxt(file_path, skiprows=1)
time = data[:, 0] / 3600 / 24
pH = data[:, 1]
SI_ara = data[:, 2]
SI_cal = data[:, 3]
Ara = data[:, 4]
Ara = data[:, 5] / stepsize
Cal = data[:, 6]
Cal = data[:, 7] / stepsize


fig, ax = plt.subplots(3, 1, figsize=(10, 13.5) )
# Set the background color of the plot to light grey

ax[0].set_facecolor('grey')
#log plot
ax[0].semilogy(time, Ara, color='Maroon', label = "Aragonite")
ax[0].semilogy(time, Cal, color='Pink', label = "Calcite")
ax[0].set_xlabel("Time (h)")
ax[0].set_ylabel("Amount of Solid (Mol)")
legend = ax[0].legend(bbox_to_anchor=(1.2, 0.17), facecolor='grey')

##########################################################################

ax[1].set_facecolor('grey')
#Reg Plot
ax[1].plot(time, Ara, color='Maroon', label = "Aragonite")
ax[1].plot(time, Cal, color='Pink', label = "Calcite")
ax[1].set_xlabel("Time (h)")
ax[1].set_ylabel("Amount of Solid (Mol)")
legend = ax[1].legend(bbox_to_anchor=(1.2, 0.17), facecolor='grey')

##########################################################################
```

```
ax[2].set_facecolor('grey')
ax[2].plot(time, dAra, color='Maroon', label = "Aragonite")
ax[2].plot(time, dCal, color='Pink', label = "Calcite")
ax[2].set_xlabel("Time (h)")
ax[2].set_ylabel("Rate Dissolved (mol/h) ")
legend = ax[2].legend(bbox_to_anchor=(1.2, 0.17), facecolor='grey')

ax[0].set_xlim()
ax[0].set_ylim()
ax[1].set_xlim()
ax[1].set_ylim()
ax[2].set_xlim()
ax[2].set_ylim()
```

```
[42]: (-3.62393113756098e-05, 4.3046608450058e-06)
```



3 The Datablock Breakdown

The input file consists of datablocks. They're denoted by the words which are in all caps. Phreeqc actually is not case sensitive however I personally prefer using all caps as it catches my eye easier. Phreeqc will also auto caps it in the even where it automatically formats the datablock. The list below follows the order shown in the input files but uses sample datablocks. The code itself will be summarized later.

3.0.1 TITLE (Optional)

The title datablock simply gives the file a title its unnecessary as it only does so within the software the saved input file can be renamed at will. It does allow for a brief description of the code as well.

EX.
TITLE –Example 1 blah blah blah

3.0.2 SOLUTION 1

This datablock defines the composition and properties of the solution where the reaction takes place. This databock can appear either, after the title, if so we can use the next 4 optional datablocks or after the KINETICS datablock in which case we omit the optional datablocks. I will further explain the purpose in a different section. The following is an example solution definition.

Ex.
SOLUTION 1 Example >—-> SOLUTION is the keyword 1 is an index for the solution and Example is a title/description if the solution
temp 25 >—-> temp is the parameter for temperature, temp 25 sets the solution temperature to 25 degrees Celsius
pH 7 >—-> pH 7 sets the pH of the solution to 7
units mmol/L >—-> sets the default units to mmol/L Mg 10 >—-> defines the solution to have a concentration of 10 units of mg, in this case units is mmol/L
Cl 100 ppm >—-> defines there to be 100 ppm of Cl in solution, one can set non-default units by adding the units after the number as shown
etc..

There are other parameters which i have omitted such as density redox and pe. The software will assume some default values for these such as density = 1.000 kg/m^3 refer to the PHREEQC V3 guide for more information.

3.0.3 EQUILIBRIUM_PHASES (Optional)

This datablock allows one to define/set initial saturations of minerals in the solution.

Ex.
EQUILIBRIUM_PHASES
Calcite 2 0 >—-> the first number is target saturation the second is the moles of the mineral added
Aragonite 1 10 >—-> The code tries to set the saturation index of calcite to 2 and aragonite to 1 through addition of 10 moles of aragonite

The datablock may not be able to reach these saturations based on the chemistry of the minerals, solution and amount of moles added but it will get as close as possible.

3.0.4 SAVE (Optional)

This datablock saves the contents of solution 1 after reaching the saturations defined in the EQUILIBRIUM_PHASES datablock, the updated solution can be called again elsewhere in the input file.

Ex.
SAVE Solution 1 >—-> saves the solution 1 defined above after saturation with the defined phases

3.0.5 END (Optional)

The end datablock ends the first part of the input file. It ensures that the above is not iterated in the loops to be defined in the rates and kinetics datablock

3.0.6 RATES

The rates datablock defines the rate equations which governs the dissolution and precipitation of, in this case calcite and aragonite, however any mineral can be uses, as long as on knows its rate equation.

Ex.
RATES
Calcite >—-> defines the mineral which the rate equation will affect
-start >—-> defines the start of the BASIC statements, the code which will run the rate equation.
10 k = 10ˆparm(1)
20 n = parm(2)
30 R = k * (1 - SR("Calcite"))ˆn
40 SAVE R * time
-end

Lines 10 20 define the rate constant and the order of the reaction, line 30 calculates the rate equation using an equation found in a paper on the topic finally line 40 saves R * time which is equal to the change in the amount of calcite at that time.

3.0.7 KINETICS

The kinetics datablock defines the initial amount of mineral/compound that will react and dissolve or precipitate according to the defined rate equation.

Calcite >—-> Defines the starting amounts for calcite
-formula CaCO3 1 >—-> Defines the mole transfer coefficient for formula per mole of reaction progress i.e 1 mol of CaCO3 dissolves for each mol of reaction
-m 0 >—-> amount of compound
-m0 0 >—-> initial amount of compound… I do not know what the difference is
-parms -7.3 1 >—-> defines parm(1), parm(2)… etc

Next compound… >—-> other compounds initial amounts are defined as done above -m
-m0
etc…

-tol 1e-08 >—-> sets the tolerance for error in the numerical evaluation
-step 500 day in 1000 # seconds >—-> defines the length of time modeled and # of steps to do so other units include hours seconds and years
-step_divide 1 >—-> This parameter affects integration by the Runge-Kutta solver. If step_divide is greater than 1.0, the first time interval of each integration is set to time step / step_divide
-runge_kutta 3 >—-> defines the order of the runge kutta integration
-bad_step_max 500 >—-> sets a limit on the number of failed steps that will retry, before aborting the execution of the program

3.0.8 INCREMENTAL_REACTIONS

Speeds up batch reactions by...

Ex. INCREMENTAL_REACTIONS true >—->

3.0.9 USE (Optional)

Ex. USE Solution 1 Calls the saved solution 1 to be used as the solvent for the reaction one could put the SOLUTIONS datablock here and define a solvent for the reaction. This is used if you didn't want to set initial saturation's of minerals in solution.

3.0.10 SELECTED_OUTPUT

Defines an output file which will save and contain the data for analysis.

Ex.
SELECTED_OUTPUT 1
-file AragoniteDissolution5.sel >—-> Creates a file called AragoniteDissolution5.sel to store data
-high_precision true >—-> Data stored will be of higher precision to 12 decimal places instead of 4
-reset false >—-> Some parameters are saved by default, refer to guide for what these are, this line allows us to only pick what is needed.
-time true >—-> Saves the time each step occurred at
-pH true >—-> Saves the pH data
-saturation_indices Aragonite Calcite >—-> Saves the saturation_index of calcite and aragonite
-kinetic_reactants Aragonite Calcite >—-> Saves the amount of change in aragonite and calcite as well as the amount left of each

END

3.0.11 END (Optional)

Declares the end of the program

4 Making the Code

The following are the steps that helped me make my model for the calcite aragonite transition. There may be more efficient processes this is just what worked for me. The output along the way may not be entirely accurate either the goal is to keep iterating the code to eventually get an accurate out put of the desired result.

4.0.1 Define the problem

Before one can solve an issue we must first identify what it is we want to solve. We want to model the conversion of aragonite over time to the more stable calcite in solution. This involves dissolution or precipitation of both calcite and aragonite, the rate and direction of which depends on conditions for which need be defined. One can break this problem down into smaller easier steps.

For example we can start with just the dissolution of aragonite in acid and make incremental improvements and modifications to the input file to end up with a complete model for the conversion of aragonite to calcite.

4.0.2 Define the rates and kinetics.

The heart of dissolution modeling in PHREEQC is defined with the RATE and KINETICS datablocks. They are required to define what and how much reacts to dissolve and precipitate. There are many defined rate equations for the dissolution of calcite and aragonite in different solutions. An often used one is the equation,

$$R = k(1 - \Omega)^n$$

where k is the rate constant,

$$\Omega$$

is the saturation of aragonite or calcite, and n is the order of the reaction, it should be noted that phreeqc does not like fractional exponents so n must be an integer.

4.0.3 Define the solution which will be the solvent.

One must define the medium which the reaction occurs. This can be pure or salt water, or a solution with an combination of ions. The software may get mad if there is more solute than solvent. For our example let us dissolve Aragonite in 0.1M Hydrochloric acid.

4.0.4 Select a database

Databases are files which contain relevant data on the properties of different compounds and elements. These properties can be defined manually and often is done for uncommon molecules, refer to the PHREEQC guide for this, however for commonly used minerals and molecules the database file avoid having to do tedious work defining the rate constant, mass, formula and name of each compound that may form, which may be in the hundreds or greater. There are two ways which one may select a database either though use of the DATABASE datablock, which is useful if you want to swap databases mid way through the program. Alternatively one can select it before running the program in the popup window that appears. The database used for this program is wateq4f.dat which is a database which specializes in Aqueous solutions.

4.0.5 Plot the result

One can plot the data in real time in phreeqc. one simply needs to use the USER_GRAPH datablock.

4.1 Code

The input file for our example of aragonite dissolution like this could look something like this.

TITLE – Code Example
RATES
Aragonite

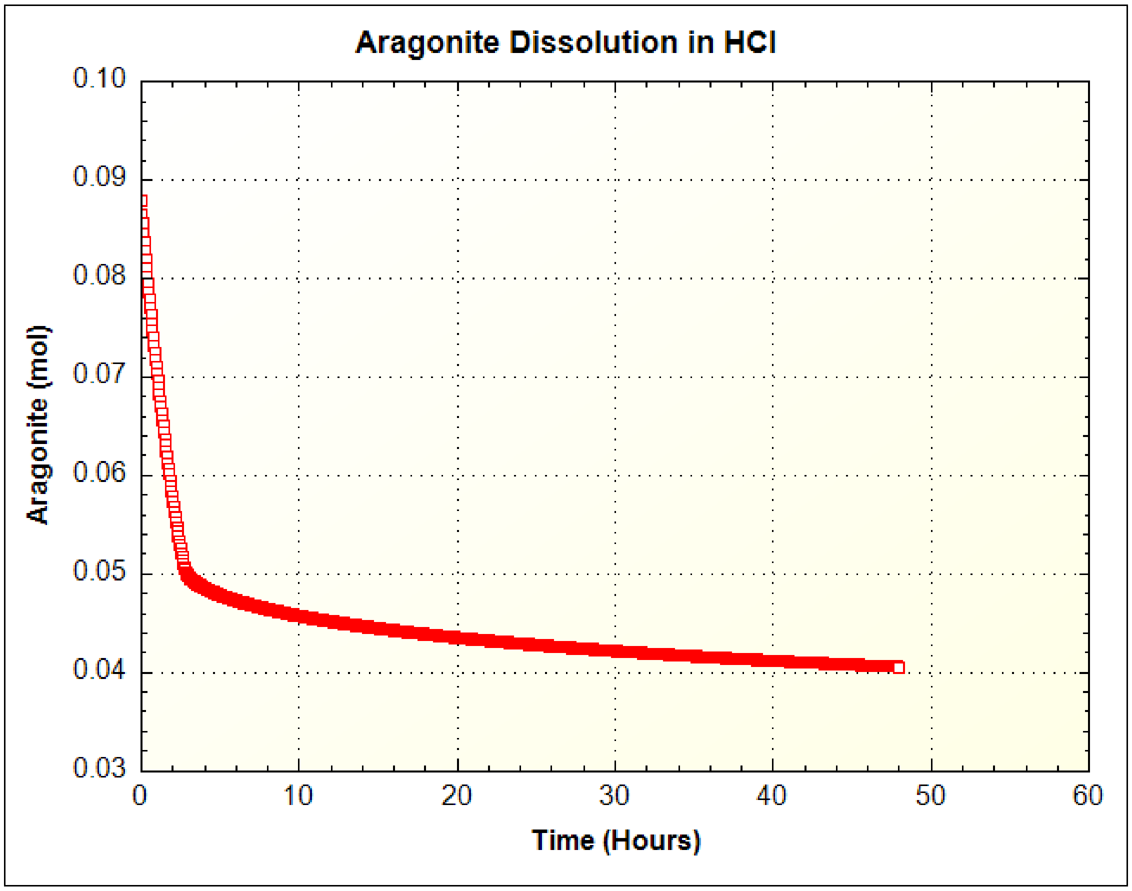
```
-start
10 k = 10^parm(1)
20 n = parm(2)
30 R = k * (1 - SI("Aragonite"))^n
40 SAVE R * time
-end

KINETICS Aragonite -formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.4 2
-tol 1e-08
-step 48 hour in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

SOLUTION 1 HCl Solution
temp 25
pH 1.000
pe 4
redox pe
units mol/kgw
density 1
Cl 0.1
H(0) 0.1
-water 1 # kg

USER_GRAPH 1
-axis_titles "Time (Hours)" "Aragonite (mol)" ""
-chart_title "Aragonite Dissolution in HCl"
-start
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN("Aragonite")
-end
-active true
END
```

4.1.1 Explanation

The code above will evaluate and plot the dissolution of Aragonite in 0.1M HCl solution all in one go. The TITLE and END datablocks are optional and may be omitted. In this case its just there for readability.

The Rate and Kinetics Datablock define the governing equation for the dissolution of aragonite. they define the rate constant to be 10^-7.8, as defined in line 10 of RATES and the -parm line in KINETICS. The KINETICS datablock also defines 0.1 mol of Aragonite to react. This dissolution is second order and the program simulates 48 hours over 1000 iterations. Note that the accuracy of this may not be 100% accurate this is more about the process than accurately describing the dissolution in acid and one can simply find better data to suit this scenario.

Assuming however that the code is accurate, the USER_GRAPH datablock will now plot the data in real time. where -axis_titles and -chart_titles, declares the labels for the axes and plot respectively, GRAPH_X total_time / 60 / 60 sets the x axis to use the time variable. Since phreeqc records all time in seconds total_time is divided by 60 twice to convert it to hours. GRAPH_Y KIN('Aragonite') plots the remaining aragonite, recorded in the KINETICS and RATES steps, against time. The resultant output should look something like this,



4.2 Modify to Include Calcite

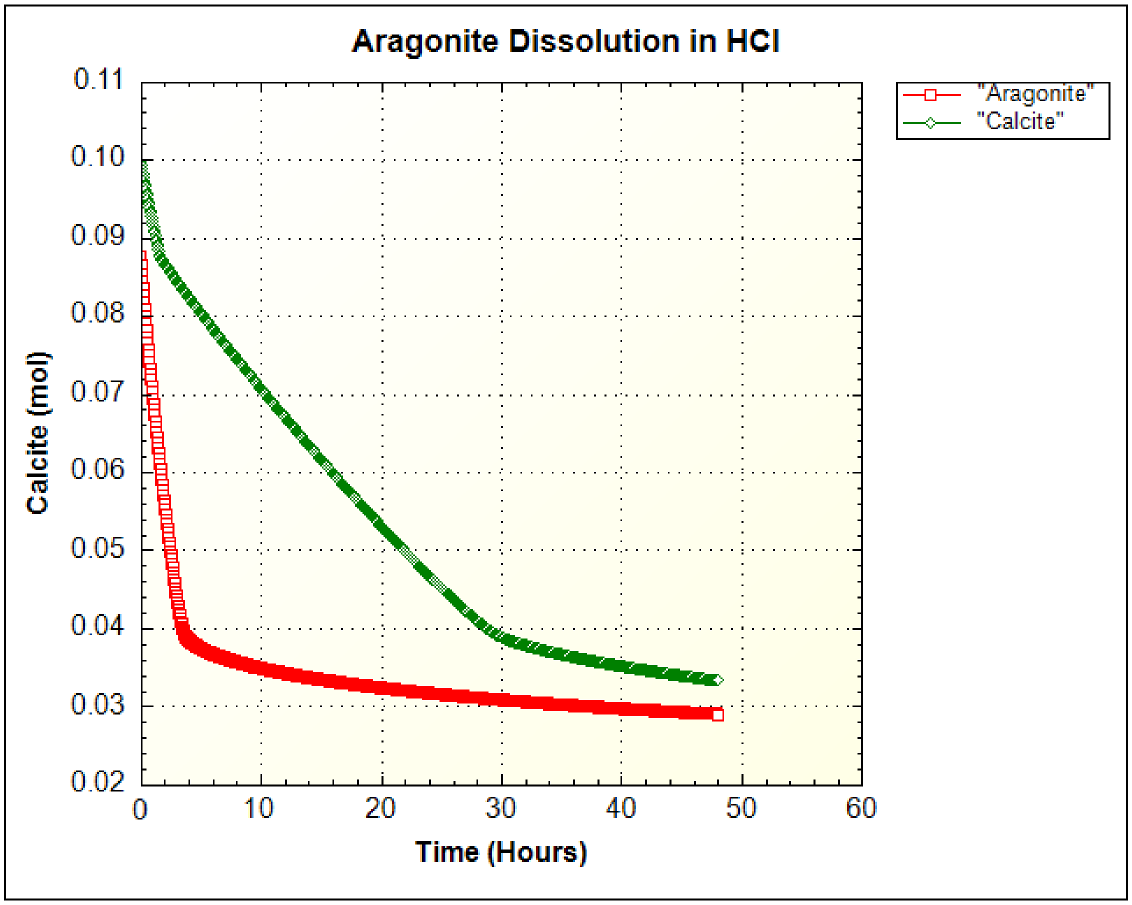
If we wanted to include calcite dissolution on the same plot so we can compare the differences in their dissolution we can add simulation. To do so we continue the code past the END datablock which will now be nesscary to split the two. We can simply copy and paste the code and modify the KINETICS and RATES to be for clacite dissolution instead, the structure however will remain the same. Doing so yeilds,

```
TITLE –Code Example
RATES
Aragonite
-start
10 k = 10^parm(1)
20 n = parm(2)
```

```
30 R = k * (1 - SI(“Aragonite”))^n
40 SAVE R * time
-end

KINETICS 1
Aragonite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.4 2
-tol 1e-08
-steps 48 hour in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

SOLUTION 1 HCl Solution
temp 25
pH 1.000
pe 4
redox pe
units mol/kgw
density 1
Cl 0.1
H(0) 0.1
-water 1 # kg

USER_GRAPH 1
-headings “Calcite” “Aragonite”
-axis_titles “Time (Hours)” “Aragonite (mol)” ""
-chart_title “Aragonite Dissolution in HCl”
-connect_simulations false
-start
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN(“Aragonite”)
-end
-active true
END

RATES
Calcite
-start
10 k = 10^parm(1)
20 n = parm(2)
30 R = k * (1 - SI(“Calcite”))^n
40 SAVE R * time
-end

KINETICS 1
```

```
Calcite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.3 1
-tol 1e-08
-steps 48 hour in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

SOLUTION 1 HCl Solution
temp 25
pH 1.000
pe 4
redox pe
units mol/kgw
density 1
Cl 0.1
H(0) 0.1
-water 1 # kg

USER_GRAPH 2 -headings "Aragonite" "Calcite"
-axis_titles "Time (Hours)" "Calcite (mol)" ""
-chart_title "Aragonite Dissolution in HCl"
-connect_simulations false -start
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN("Calcite")
-end
-active true
END
```

This time we also included the line -connect_simulations false
this is due to the fact that phreeqc will plot all the data as if it were one run otherwise. We also added a -headings line this adds a legend to the plot. For some unknown reason it would only write out the correct mineral if it went in the second position, normally the first mineral plotted would go in the first position, I.E it was expected to be "Aragonite" "Calcite". Lastly one will notice that calcite although using a similar equation, it is of order 1 and has a rate constant of 10^-7.3. Running the code should provide an output of,



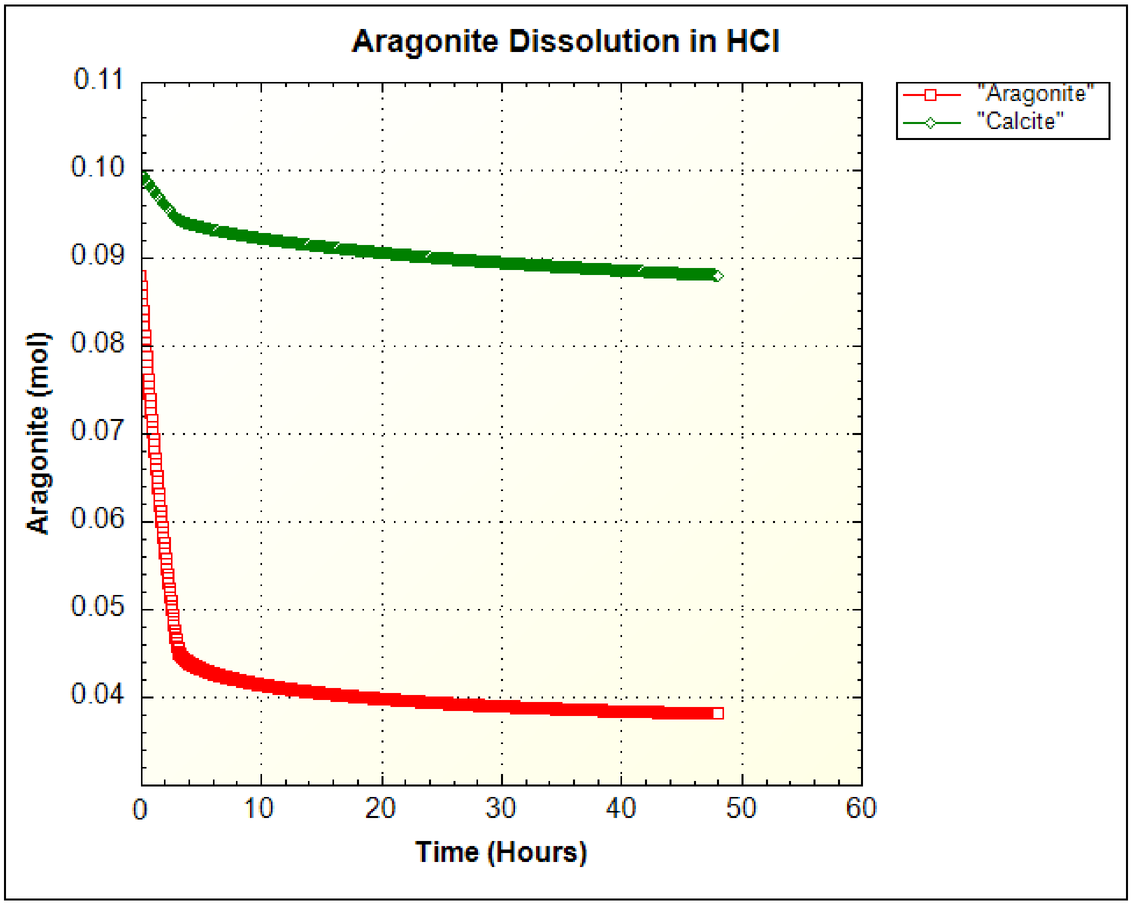
4.3 Modify so Both Minerals Dissolve at Once

If we rearrange to have both Calcite and Aragonite under one RATES and KINETICS datablock we can simulate having both dissolve at once. This will be closer to what will occur for the conversion from aragonite to calcite. To do this we will only have 1 simulation and ill move the info for calcite under the respective datablocks. Doing so we get,

```
TITLE –Code Example
RATES
Aragonite
-start
10 k = 10^parm(1)
20 n = parm(2)
30 R = k * (1 - SI("Aragonite"))^n
40 SAVE R * time
-end

Calcite
-start
10 k = 10^parm(1)
20 n = parm(2)
30 R = k * (1 - SI("Calcite"))^n
40 SAVE R * time
```

```
-end

KINETICS 1
Aragonite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.4 2
-tol 1e-08

Calcite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.3 1
-tol 1e-08

-steps 48 hour in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

SOLUTION 1 HCl Solution
temp 25
pH 1.000
pe 4
redox pe
units mol/kgw
density 1
Cl 0.1
H(0) 0.1
-water 1 # kg

USER_GRAPH 1
-headings "" “Aragonite” “Calcite” -axis_titles “Time (Hours)” “Aragonite (mol)” ""
-chart_title “Dissolution in HCl”
-connect_simulations false
-start
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN(“Aragonite”) KIN(“Calcite”)
-end
-active true
END
```

Running the code now yields,



where we can see that calcite doesn't dissolve nearly as much when aragonite is present.

4.4 Modify for the Transition Between Aragonite and Calcite

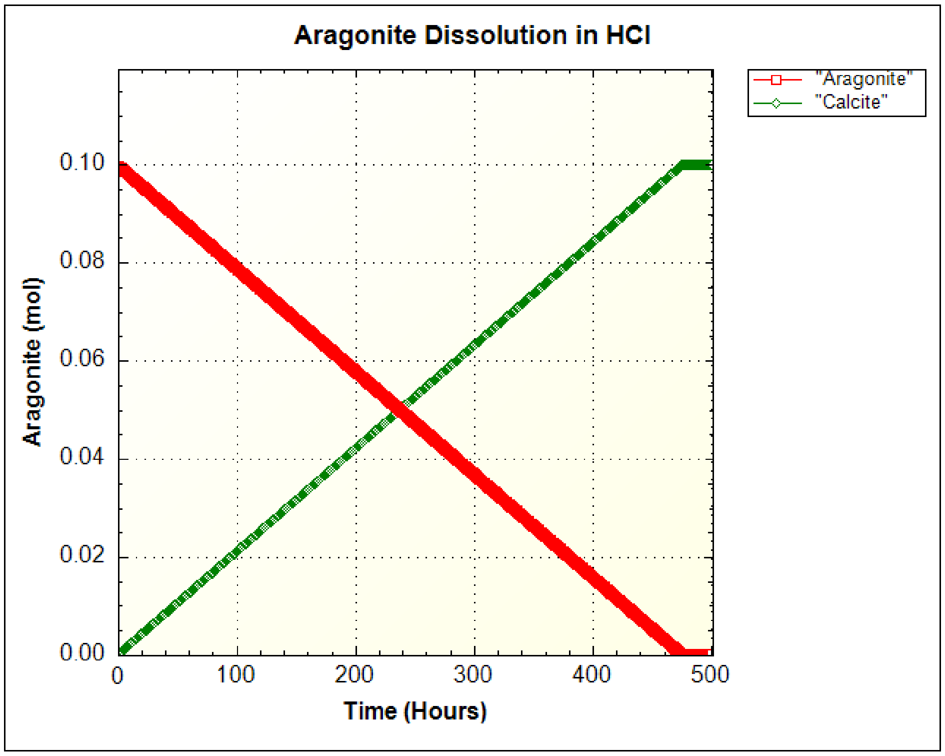
We now have a very flexible program which can be modified to model the conversion of aragonite to calcite. We can do this by essentially just changing the solution to be that of water or whatever medium one wants. I'll just use water for ease. We can also save the data to a file which will allow us to plot it in python as well as save in a spreadsheet (If that is useful to you). To do so we use the SELECTED_OUTPUT Datablock.

```
TITLE –Code Example
RATES
Aragonite
-start
10 k = 10ˆparm(1)
20 n = parm(2)
30 R = k * (1 - SI("Aragonite"))ˆn
40 SAVE R * time
-end

Calcite
-start
10 k = 10ˆparm(1)
```

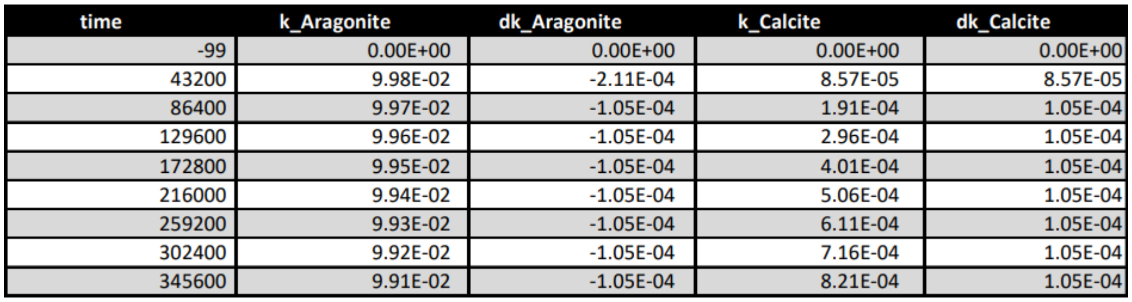
```
20 n = parm(2)
30 R = k * (1 - SI(“Calcite”))^n
40 SAVE R * time
-end

KINETICS 1
Aragonite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.4 2
-tol 1e-08

Calcite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.3 1
-tol 1e-08

-steps 500 hour in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

SOLUTION 1 HCl Solution
temp 25
pH 1.000
pe 4
redox pe
units mol/kgw
density 1
Cl 0.1
H(0) 0.1
-water 1 # kg

SELECTED_OUTPUT
-file Tutorial.sel
-high_precision true
-reset false
-step false
-time true
-kinetic_reactants Aragonite Calcite

USER_GRAPH 1
-headings "" “Aragonite” “Calcite”
-axis_titles “Time (Hours)” “Aragonite (mol)” ""
-chart_title “Dissolution in HCl”
-connect_simulations false
-start
```

```
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN("Aragonite") KIN("Calcite")
-end
-active true
END
```

The SELECTED_OUTPUT datablock saves the the specified data to a file called Tutorial.sel. Specifically it records, the time, the amount of aragonite and calcite at each step and the change in aragonite and calcite for each step. The plotted data looks something like this,



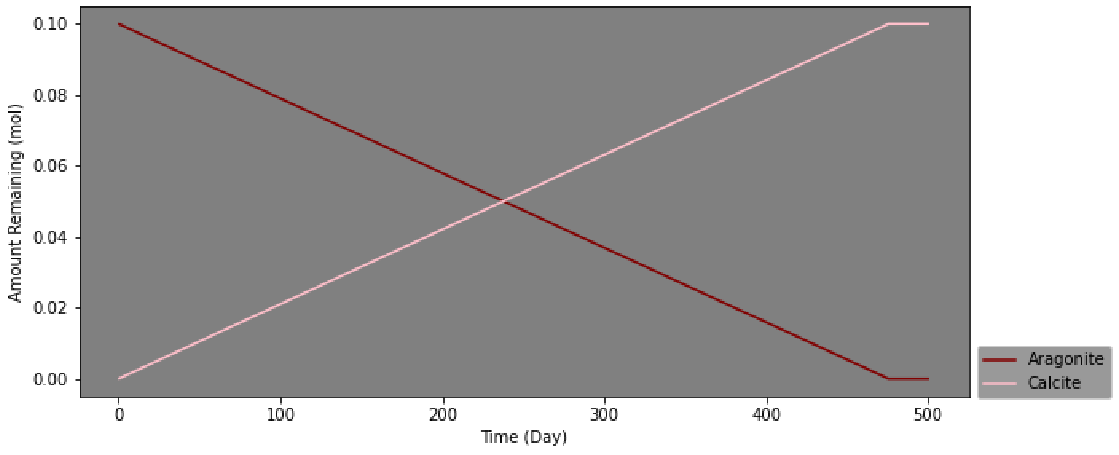
where we had to simulate for longer due to the fact that were dissolving in water. From the plot however we see that slowly over time calcite will form as aragonite dissolves in water. The same data is stored in the Tutorial.sel file the data is contained in the following table,



time	k_Aragonite	dk_Aragonite	k_Calcite	dk_Calcite
-99	0.00E+00	0.00E+00	0.00E+00	0.00E+00
43200	9.98E-02	-2.11E-04	8.57E-05	8.57E-05
86400	9.97E-02	-1.05E-04	1.91E-04	1.05E-04
129600	9.96E-02	-1.05E-04	2.96E-04	1.05E-04
172800	9.95E-02	-1.05E-04	4.01E-04	1.05E-04
216000	9.94E-02	-1.05E-04	5.06E-04	1.05E-04
259200	9.93E-02	-1.05E-04	6.11E-04	1.05E-04
302400	9.92E-02	-1.05E-04	7.16E-04	1.05E-04
345600	9.91E-02	-1.05E-04	8.21E-04	1.05E-04

This is the first few lines of data which were store. The data was copied into excel to make the table. Excel is useful as it formats the data in an easier to read format than the files do on their own, this is something you'll know when you store more and more data in one file. the file can be read in python and plotted using matplotlib. This is useful as one can plot the data with out having to rerun the simulation, which is faster. The plots also are more customizable and tend to look nicer. Going from left to right we have columns 0 to 3. You can see that the first “step” starts with time = -99. This is an initial step which we is not apart of the actual dissolution, this would be the step where the initial condidtions of the solution, would be saved if we allowed. We can plot this data in python using the following,

[51]:
```
import numpy as np
import matplotlib.pyplot as plt

file_path3 = r'Tutorial.sel'

# Read data from .sel file
data = np.loadtxt(file_path3, skiprows= 2)
time = data[:, 0] / 3600 / 24 #Converting the seconds to days
ara = data[:, 1]
cal = data[:, 3]

# Plot
fig, ax = plt.subplots(figsize=(10,4.5))

ax.set_facecolor('grey')
ax.plot(time, ara, color='Maroon', label = "Aragonite")
ax.plot(time, cal, color='Pink', label = "Calcite")
ax.set_xlabel("Time (Day)")
ax.set_ylabel("Amount Remaining (mol)")
legend = ax.legend(bbox_to_anchor=(1, .15), facecolor='grey')
```



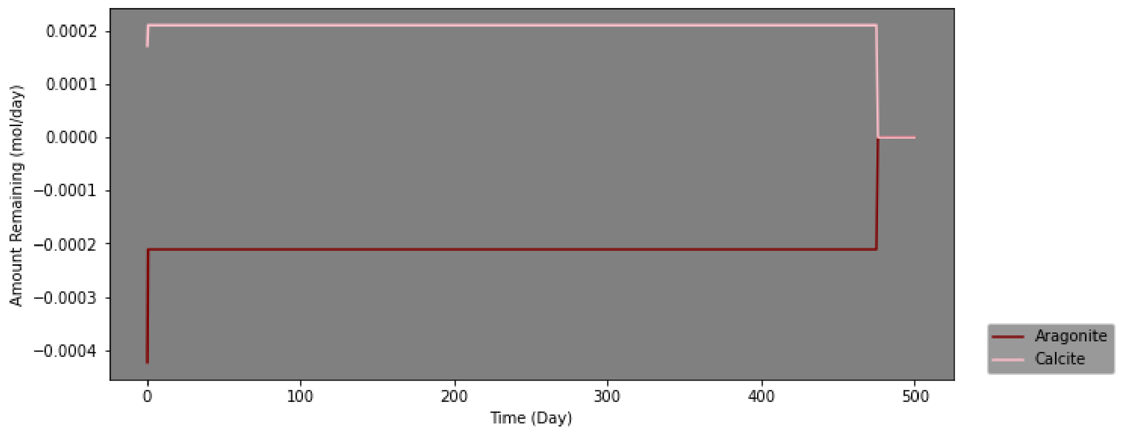
Where the we see the data is the same and the plot is miles nicer. We can also plot the rate of change in the conversion over time, we could also do this in phreeqc by writing,

USER_GRAPH 1
-headings "" “Aragonite” “Calcite”

```
-axis_titles "Time (Hours)" "Aragonite (mol)" ""
-chart_title "Dissolution in HCl"
-connect_simulations false
-start
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN_DELTA("Aragonite") KIN_DELTA("Calcite")
-end
-active true
```

Plotting in python however we use the following

```
[53]: timestep = 500/1000 #needed to get the rate from the change, r = mol/t

      dara = data[:, 2]  / timestep
      dcal = data[:, 4]  / timestep

      # Plot
      fig, ax = plt.subplots(figsize=(10,4.5))

      ax.set_facecolor('grey')
      ax.plot(time, ara, color='Maroon', label = "Aragonite")
      ax.plot(time, cal, color='Pink', label = "Calcite")
      ax.set_xlabel("Time (Day)")
      ax.set_ylabel("Amount Remaining (mol/day)")
      legend = ax.legend(bbox_to_anchor=(1.2, .17), facecolor='grey')
```



we can see a fast initial dissolution of aragonite followed by constant dissolution and precipitation until full conversion occurs and the rate goes to zero.

This is a functional model of the conversion between Aragonite and Calcite and one can now adjust initial amounts of calcite and aragonite as well as the makeup of the solution. This a fairly versatile program already however there are a few more modifications we can make to increase the amount of situations we can model.

4.5 Add Equilibrium_Phases Datablock

Through addition of this datablock we can allow the system to start with an initial saturation of calcite and aragonite as well as any other minerals. We can even edit the initial partial pressure of CO2. This may affect the progression of the conversion it may even prevent it entirely. If we want to add this datablock the code should look like this,

```
TITLE –Code Example
SOLUTION 1 HCl Solution
temp 25
pH 1.000
pe 4
redox pe
units mol/kgw
density 1
Cl 0.1
H(0) 0.1
-water 1 # kg

EQUILIBRIUM_PHASES
Calcite 0 0
Aragonite 0 0
CO2 0 0

SAVE Solution 1
END

RATES
Aragonite
-start
10 k = 10ˆparm(1)
20 n = parm(2)
30 R = k * (1 - SI("Aragonite"))ˆn
40 SAVE R * time
-end

Calcite
-start
10 k = 10ˆparm(1)
20 n = parm(2)
30 R = k * (1 - SI("Calcite"))ˆn
40 SAVE R * time
-end

KINETICS 1
Aragonite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.4 2
-tol 1e-08
```

```
Calcite
-formula CaCO3 1
-m 0.1
-m0 0.1
-parms -7.3 1
-tol 1e-08

-steps 500 hour in 1000 # seconds
-step_divide 1
-runge_kutta 3
-bad_step_max 500
INCREMENTAL_REACTIONS true

USE Solution 1

SELECTED_OUTPUT
-file Tutorial.sel
-high_precision true
-reset false
-step false
-time true
-kinetic_reactants Aragonite Calcite

USER_GRAPH 1
-headings "" “Aragonite” “Calcite”
-axis_titles “Time (Hours)” “Aragonite (mol)” ""
-chart_title “Dissolution in HCl”
-connect_simulations false
-start
10 GRAPH_X total_time / 60 / 60
20 GRAPH_Y KIN(“Aragonite”) KIN(“Calcite”)
-end
-active true
END
```

The program has to be split into two simulations and the solution after being equalized needs to be saved in the SAVE datablock. One can play around with the saturation’s and amount of Calcite Aragonite and CO2 or add more minerals if wanted. If they’re left at 0 and 0 the program will run as it did before. This is the final version of my program, with an additional USER_GRAPH block which may be omitted. This is also the current version i use.

5 PHREEQPY

Phreeqpy is a python library which allows for python to call and utilizes phreeqc software with the added benefits of the extended python libraries for plotting math etc. I also believe that it has an advantage when sharing code as more people understand using python and, because I plot in Python anyways, the code is all contained within one file rather than a phreeqc input file and a .ipynb file. The following is the a similar model for the transition of aragonite to calcite converted to python code via phreeqpy. Some constants may be different to match updated/ more accurate data.

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     from tqdm import tqdm
     from IPython.display import clear_output
     import time as tim

     """
     The following python code is a conversion of AragoniteDissolution5. A PHREEQC␣
      ↪input file whichI wish to convert to PHREEQPY code.
     """

     import sys

     # Simple Python 3 compatibility adjustment.
     if sys.version_info[0] == 2:
         range = xrange

     import os
     import timeit

     MODE = 'dll'  # 'dll' or 'com'

     if MODE == 'com':
         import phreeqpy.iphreeqc.phreeqc_com as phreeqc_mod
     elif MODE == 'dll':
         import phreeqpy.iphreeqc.phreeqc_dll as phreeqc_mod
     else:
         raise Exception('Mode "%s" is not defined use "com" or "dll".' % MODE)


     def run_phreeqc(input_string):
         """
         Run PHREEQC simulation with the given input string.
         """
         phreeqc = phreeqc_mod.IPhreeqc()
         phreeqc.load_database(r"C:\Program Files (x86)\USGS\Phreeqc Interactive 3.7.
      ↪3-15968\database\wateq4f.DAT")
         phreeqc.run_string(input_string)
         return phreeqc


     def extract_data(phreeqc):
         """
         Extract data from PHREEQC simulation results.
         """
         time = phreeqc.get_selected_output_column(0)
         SI_ara = phreeqc.get_selected_output_column(1)
```

```python
    SI_cal = phreeqc.get_selected_output_column(2)
    Ara = phreeqc.get_selected_output_column(3)
    Ara = phreeqc.get_selected_output_column(4)
    Cal = phreeqc.get_selected_output_column(5)
    Cal = phreeqc.get_selected_output_column(6)

    time.pop(0)
    SI_ara.pop(0)
    SI_cal.pop(0)
    Ara.pop(0)
    Ara.pop(0)
    Cal.pop(0)
    Cal.pop(0)

    return time, SI_ara, SI_cal, Ara, Ara, Cal, Cal


def plot_data(time, SI_ara, SI_cal, Ara, Ara, Cal, Cal):
    """
    Plot the data extracted from PHREEQC simulation.
    """
    stepsize = 150 / 1000
    t_unit = "years"
    totmol = Cal[0] + Ara[0]
    _ara = []
    _cal = []

    # Initialize the progress bar
    progress_bar = tqdm(total=len(Ara))

    for i in range(len(Ara)):

        time[i] = time[i] / 3600
        Ara[i] = Ara[i] / stepsize
        Cal[i] = Cal[i] / stepsize
        _ara.append(Ara[i] / totmol)
        _cal.append(1 - _ara[i])

    ### Uncomment the following if you want to have a progress bar, will impact␣
 ↪preformance! ###
        # Update the progress bar
        progress_bar.set_description(f"Processing data point: {i+1}/{len(Ara)}")
        progress_bar.update(1)
    # Close the progress bar
    progress_bar.close()
    # Start plotting
    fig, ax = plt.subplots(3, 1, figsize=(10, 13.5))

    # Set the background color of the plot to light grey
    ax[0].set_facecolor('grey')
    ax[0].semilogy(time, Ara, color='Maroon', label="Aragonite")
    ax[0].semilogy(time, Cal, color='Pink', label="Calcite")
    ax[0].set_xlabel("Time (" + t_unit + ")", color='white')
    ax[0].set_ylabel("Amount of Solid (Mol)", color='white')
    ax[0].tick_params(axis='x', colors='white')
    ax[0].tick_params(axis='y', colors='white')
    legend = ax[0].legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

########################################################################

    ax[1].set_facecolor('grey')
    ax[1].plot(time, Ara, color='Maroon', label="Aragonite")
    ax[1].plot(time, Cal, color='Pink', label="Calcite")
    ax[1].set_xlabel("Time (" + t_unit + ")", color='white')
    ax[1].set_ylabel("Amount of Solid (Mol)", color='white')
    ax[1].tick_params(axis='x', colors='white')
    ax[1].tick_params(axis='y', colors='white')
    legend = ax[1].legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

########################################################################

    ax[2].set_facecolor('grey')
    ax[2].plot(time, Ara, color='Maroon', label="Aragonite")
    ax[2].plot(time, Cal, color='Pink', label="Calcite")
    ax[2].set_xlabel("Time (" + t_unit + ")", color='white')
    ax[2].set_ylabel("Rate Dissolved (mol/h)", color='white')
    ax[2].tick_params(axis='x', colors='white')
    ax[2].tick_params(axis='y', colors='white')
    legend = ax[2].legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

    ax[0].set_xlim()
    ax[0].set_ylim()
    ax[1].set_xlim()
    ax[1].set_ylim()
    ax[2].set_xlim()
    ax[2].set_ylim()
```

```
############################################################################

    fig, ax = plt.subplots(figsize=(10, 4.5))

    ax.set_facecolor('grey')
    ax.plot(time, SI_ara, color='Maroon', label="Aragonite")
    ax.plot(time, SI_cal, color='Pink', label="Calcite")
    ax.set_xlabel("Time (h)", color='white')
    ax.set_ylabel("Saturation Index", color='white')
    ax.tick_params(axis='x', colors='white')
    ax.tick_params(axis='y', colors='white')
    legend = ax.legend(loc='lower right', facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

############################################################################

    fig, ax = plt.subplots(figsize=(10, 4.5))

    ax.set_facecolor('grey')
    ax.plot(time, _ara, color='Maroon', label="Aragonite")
    ax.plot(time, _cal, color='Pink', label="Calcite")
    ax.set_xlabel("Time (h)", color='white')
    ax.set_ylabel("mole fraction", color='white')
    ax.tick_params(axis='x', colors='white')
    ax.tick_params(axis='y', colors='white')
    legend = ax.legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

    # Show the plots
    plt.show()


def main():

    start_time = tim.time()

    """
    Main function to execute the PHREEQC simulation and plot the results.
    """
    input_string = """
    SOLUTION 1
        temp      20
        pH        7.8
        pe        4
        redox     pe
        units     mmol/L
        C(4)      1.0
        density   1
        -water    1 # kg

    EQUILIBRIUM_PHASES 1
        Aragonite 0.0   0
        Calcite   0.0   0
        CO2(g)    -3.5  1
    SAVE Solution 1
    END

    RATES
        Aragonite
        -start
        10 k = 10^parm(1)
        20 n = parm(2)
        30 R = k * (1 - SR("Aragonite"))^n
        40 SAVE R * time
        -end
        Calcite
        -start
        10 k = 10^parm(1)
        20 n = parm(2)
        30 R = k * (1 - SR("Calcite"))^n
        40 SAVE R * time
        -end

    KINETICS 1
    Aragonite
        -formula  CaCO3  1
        -m        0.00035
        -m0       0.00035
        -parms    -8.34 2
        -tol      1e-08
    Calcite
        -formula  CaCO3  1
        -m        0.0000
        -m0       0.0000
        -parms    -8.48 1
        -tol      1e-08
    -steps        150 hour in 1000
    -step_divide 1
    -runge_kutta 3
    -bad_step_max 500
    INCREMENTAL_REACTIONS true
```

```
    USE Solution 1

    SELECTED_OUTPUT 1
        -file                   AragoniteDissolution5(3).sel
        -high_precision         true
        -reset                  false
        -time                   true
        -saturation_indices     Aragonite  Calcite
        -kinetic_reactants      Aragonite  Calcite

    END
    """

    phreeqc = run_phreeqc(input_string)
    time, SI_ara, SI_cal, Ara, Ara, Cal, Cal = extract_data(phreeqc)
    plot_data(time, SI_ara, SI_cal, Ara, Ara, Cal, Cal)

    # Show the plots
    plt.show()
    end_time = tim.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")

if __name__ == '__main__':
    main()
```

The code includes both the execution of the phreeqc input file and the plotting of the data.

5.1 Code Breakdown

The above code has the following structure,

- Importation of needed libraries
- Definition of Functions
- Definition of main() Function

I will explain each section and how one may modify it.

5.1.1 Importing Libraries

There are a few libraries which we needed to import before execution. They include, Matplotlib and numpy, which are needed for plotting and data management.

tqdm and Ipython.display, specifically clear_output from Ipython.display and time. These are needed for debugging and the progress bar.

The next and final segment is better discussed as a full block.

[1]:

```
import sys

# Simple Python 3 compatibility adjustment.
if sys.version_info[0] == 2:
    range = xrange

import os
import timeit

MODE = 'dll'   # 'dll' or 'com'

if MODE == 'com':
    import phreeqpy.iphreeqc.phreeqc_com as phreeqc_mod
elif MODE == 'dll':
    import phreeqpy.iphreeqc.phreeqc_dll as phreeqc_mod
else:
    raise Exception('Mode "%s" is not defined use "com" or "dll".' % MODE)
```

The above is a compatibility test which will import the proper phreeqpy library depending on the compatibility with the computers OS. One can either use dll or com.

5.1.2 Defining Functions

There are multiple functions which are defined and called by the main program. The following is a list of these functions,

- run_phreeqc(input_string) >—-> Which executes the phreeqc input file
- extract_data(phreeqc) >—-> Extracts and prepares data to be plotted
- plot_data >—-> Plots the data

all of these are called and controlled by a main() function.

5.1.3 Main()

The main function calls and executes each component of the program as well as defines our input_string. The main function has the following structure,

[1]:

```
def main():

    start_time = tim.time()

    """
    Main function to execute the PHREEQC simulation and plot the results.
    """
    input_string = """
```

```
SOLUTION 1
    temp      20
    pH        7.8
    pe        4
    redox     pe
    units     mmol/L
    C(4)      1.0
    density   1
    -water    1 # kg

EQUILIBRIUM_PHASES 1
    Aragonite 0.0   0
    Calcite   0.0   0
    CO2(g)    -3.5  1
SAVE Solution 1
END

RATES
    Aragonite
    -start
    10 k = 10^parm(1)
    20 n = parm(2)
    30 R = k * (1 - SR("Aragonite"))^n
    40 SAVE R * time
    -end
    Calcite
    -start
    10 k = 10^parm(1)
    20 n = parm(2)
    30 R = k * (1 - SR("Calcite"))^n
    40 SAVE R * time
    -end

KINETICS 1
Aragonite
    -formula  CaCO3  1
    -m        0.00035
    -m0       0.00035
    -parms    -8.34 2
    -tol      1e-08
Calcite
    -formula  CaCO3  1
    -m        0.0000
    -m0       0.0000
    -parms    -8.48 1
    -tol      1e-08
-steps        150 hour in 1000
```

```
    -step_divide 1
    -runge_kutta 3
    -bad_step_max 500
    INCREMENTAL_REACTIONS true

    USE Solution 1

    SELECTED_OUTPUT 1
        -file                   AragoniteDissolution5(3).sel
        -high_precision         true
        -reset                  false
        -time                   true
        -saturation_indices     Aragonite  Calcite
        -kinetic_reactants      Aragonite  Calcite

    END
    """

    phreeqc = run_phreeqc(input_string)
    time, SI_ara, SI_cal, Ara, dAra, Cal, dCal = extract_data(phreeqc)
    plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal)

    # Show the plots
    plt.show()
    end_time = tim.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

This function has a fairly basic structure.
Firsts it defines a start_time, which is optional and begins recording the elapsed time.
The input string however is just a variable containing the entire phreeqc input file in text. This variable used as input for the run_phreeqc function which executes and saves the data in a variable called phreeqc
The phreeqc variable contains the data simulated from our phreeqc code. To get this data and store it we define variables for the, time, SI and amount of each solid and the change in each solid. We call the function extract_data function which stores this data in arrays with the same names and removes the junk data, i.e. the column headers.
Finally the plot_data function is called and it plots the data for analysis. This occurs and the elapsed time is calculated and printed to the user.

5.1.4 run_phreeqc()

The following is the run_phreeqc function,

[1]:
```
def run_phreeqc(input_string):
    """
    Run PHREEQC simulation with the given input string.
    """
    phreeqc = phreeqc_mod.IPhreeqc()
    phreeqc.load_database(r"C:\Program Files (x86)\USGS\Phreeqc Interactive 3.7.
↪3-15968\database\wateq4f.DAT")
    phreeqc.run_string(input_string)
    return phreeqc
```

There are three lines to this function, phreeqc = phreeqc_mod.IPhreeqc() <—> Creates an instance of the phreeqc class

phreeqc.load_database(...) defines and loads the database to be used. This line will have to be modified depending on the directory location of the database.

Finally,
phreeqc.run_string(input_string) takes the variable defined in main and runs the phreeqc code. I find this to usually take less time than it normally does in phreeqc software which is a nice bonus. The output from this is saved in a variable called phreeqc and returned the main function for extraction and plotting.

5.1.5 extract_data()

[1]:

```
def extract_data(phreeqc):
    """
    Extract data from PHREEQC simulation results.
    """
    time = phreeqc.get_selected_output_column(0)
    SI_ara = phreeqc.get_selected_output_column(1)
    SI_cal = phreeqc.get_selected_output_column(2)
    Ara = phreeqc.get_selected_output_column(3)
    dAra = phreeqc.get_selected_output_column(4)
    Cal = phreeqc.get_selected_output_column(5)
    dCal = phreeqc.get_selected_output_column(6)

    time.pop(0)
    SI_ara.pop(0)
    SI_cal.pop(0)
    Ara.pop(0)
    dAra.pop(0)
    Cal.pop(0)
    dCal.pop(0)

    return time, SI_ara, SI_cal, Ara, dAra, Cal, dCal
```

The command phreeqc.get_selected_output_column() takes the output from running the phreeqc code, which is stored in an array, and selects a specific column which can be saved into an array. One can also use phreeqc.get_selected_output_array() to save the entire array to a variable, this is useful to determine which column contains each output, if your unsure. After saving the data to their own variable we take each array and remove the part containing the heading to the array, i.e the line saying time, SI_ara, etc. we do this with the .pop(0) command which removes the first entry. These lists are saved and then sent to the plot_data function.

5.1.6 plot_data()

The following is the final function which has the job of plotting our calculated data.

[1]:

```
def plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal):
    """
    Plot the data extracted from PHREEQC simulation.
    """
    stepsize = 150 / 1000
    t_unit = "years"
    totmol = Cal[0] + Ara[0]
    _ara = []
    _cal = []

    # Initialize the progress bar
    progress_bar = tqdm(total=len(dAra))

    for i in range(len(dAra)):

        time[i] = time[i] / 3600
        dAra[i] = dAra[i] / stepsize
        dCal[i] = dCal[i] / stepsize
        _ara.append(Ara[i] / totmol)
        _cal.append(1 - _ara[i])

    ### Uncomment the following if you want to have a progress bar, will impact␣
 ↪preformance! ###
        # Update the progress bar
        progress_bar.set_description(f"Processing data point: {i+1}/{len(dAra)}")
        progress_bar.update(1)
    # Close the progress bar
    progress_bar.close()


    # Start plotting
    fig, ax = plt.subplots(3, 1, figsize=(10, 13.5))

    # Set the background color of the plot to light grey
    ax[0].set_facecolor('grey')
    ax[0].semilogy(time, Ara, color='Maroon', label="Aragonite")
    ax[0].semilogy(time, Cal, color='Pink', label="Calcite")
    ax[0].set_xlabel("Time (" + t_unit + ")", color='white')
    ax[0].set_ylabel("Amount of Solid (Mol)", color='white')
    ax[0].tick_params(axis='x', colors='white')
    ax[0].tick_params(axis='y', colors='white')
    legend = ax[0].legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

###########################################################################

    ax[1].set_facecolor('grey')
    ax[1].plot(time, Ara, color='Maroon', label="Aragonite")
    ax[1].plot(time, Cal, color='Pink', label="Calcite")
    ax[1].set_xlabel("Time (" + t_unit + ")", color='white')
    ax[1].set_ylabel("Amount of Solid (Mol)", color='white')
    ax[1].tick_params(axis='x', colors='white')
    ax[1].tick_params(axis='y', colors='white')
    legend = ax[1].legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

###########################################################################

    ax[2].set_facecolor('grey')
    ax[2].plot(time, dAra, color='Maroon', label="Aragonite")
    ax[2].plot(time, dCal, color='Pink', label="Calcite")
    ax[2].set_xlabel("Time (" + t_unit + ")", color='white')
    ax[2].set_ylabel("Rate Dissolved (mol/h)", color='white')
    ax[2].tick_params(axis='x', colors='white')
    ax[2].tick_params(axis='y', colors='white')
    legend = ax[2].legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

    ax[0].set_xlim()
    ax[0].set_ylim()
    ax[1].set_xlim()
    ax[1].set_ylim()
    ax[2].set_xlim()
    ax[2].set_ylim()

###########################################################################

    fig, ax = plt.subplots(figsize=(10, 4.5))
    ax.set_facecolor('grey')
    ax.plot(time, SI_ara, color='Maroon', label="Aragonite")
    ax.plot(time, SI_cal, color='Pink', label="Calcite")
    ax.set_xlabel("Time (h)", color='white')
    ax.set_ylabel("Saturation Index", color='white')
    ax.tick_params(axis='x', colors='white')
    ax.tick_params(axis='y', colors='white')
    legend = ax.legend(loc='lower right', facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

############################################################################

    fig, ax = plt.subplots(figsize=(10, 4.5))

    ax.set_facecolor('grey')
    ax.plot(time, _ara, color='Maroon', label="Aragonite")
    ax.plot(time, _cal, color='Pink', label="Calcite")
    ax.set_xlabel("Time (h)", color='white')
    ax.set_ylabel("mole fraction", color='white')
    ax.tick_params(axis='x', colors='white')
    ax.tick_params(axis='y', colors='white')
    legend = ax.legend(bbox_to_anchor=(1.25, 0.3), facecolor='grey')
    legend.get_texts()[0].set_color('white')
    legend.get_texts()[1].set_color('white')

    # Show the plots
    plt.show()
```

This function is essentially the same code that was used to plot the data before with the exception of this part

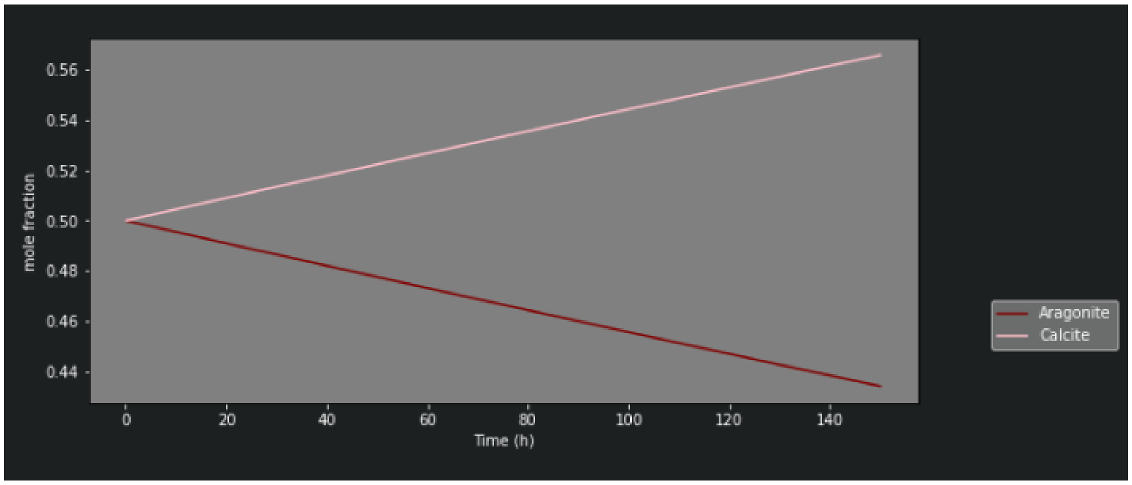
[1]:

```
def plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal):
    """
    Plot the data extracted from PHREEQC simulation.
    """
    stepsize = 150 / 1000
    t_unit = "years"
    totmol = Cal[0] + Ara[0]
    _ara = []
    _cal = []

    # Initialize the progress bar
    progress_bar = tqdm(total=len(dAra))
    for i in range(len(dAra)):

        time[i] = time[i] / 3600
        dAra[i] = dAra[i] / stepsize
        dCal[i] = dCal[i] / stepsize
        _ara.append(Ara[i] / totmol)
        _cal.append(1 - _ara[i])

    ### Uncomment the following if you want to have a progress bar, will impact␣
↪preformance! ###
        # Update the progress bar
        progress_bar.set_description(f"Processing data point: {i+1}/{len(dAra)}")
        progress_bar.update(1)
    # Close the progress bar
    progress_bar.close()
```

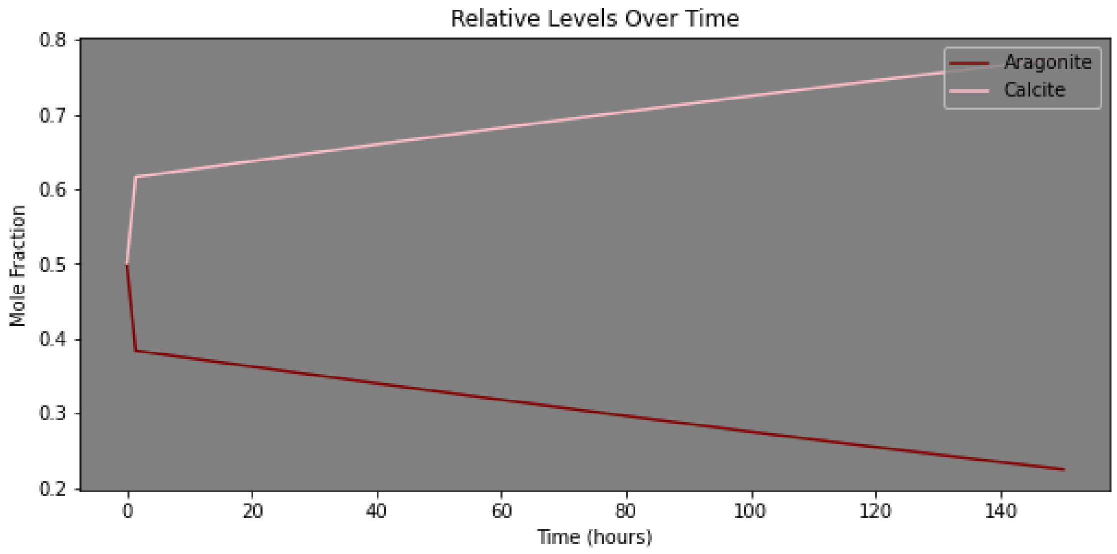
We have to scale dCal and dAra to the time-step and we do this through use of a loop, we also calculate the mole fraction of calcite and aragonite in this loop. I added a progress as well to ensure the processing doesn't get stuck. It does hinder performance so its mainly just a troubleshooting tool, especially for higher step sizes. Upon completion the data is then plotted as before yielding the same results as the phreeqc code did in less time.

5.2 Troubleshooting

There are a few issue that this model encountered when trying to match/model the real life experiments as well as when adding the surface area dependence to the rate equation. The rate would either be way too fast or way too slow and wouldn't exhibit the inital fast dissolution that it did for the test data used for initially and shown throughout this guide. The exact reasons for this are unclear however, the issue seemed to be that of scaling. By adding more water the issue resolved to where there was a about 100X as much water as solid. This of course does not match the real life conditions. The following is the bad output



The above is a plot of the mole fraction of calcite and aragonite it is the plot which bes illustrates the issue. We should expect, when we start with equal levels of calcite and aragonite, a large initial jump of about 10 - 20 percent as seen in the following,



We can see from this plot, which is from the updated code, that we once again have that inital jump which is observed in the real experiments.

5.3 New and Improved Code

I made some modifications to remedy this issue Specifically I added an if statement to my rates equation where after a set amount of time, around that observed from experiments, the rate constant shifts to to a smaller one which yields the observed results. The following is the modified RATES code,

```
RATES 1
Aragonite
-start
10 IF (TOTAL_TIME <= parm(4)) THEN k = 10^parm(1)
20 IF (TOTAL_TIME > parm(4)) THEN k = 10^parm(2)
30 n = parm(3)
40 A = 100000 * 100.09 * KIN("Aragonite")
50 R = (k *A * (1 - SR("Aragonite"))^n)
60 SAVE R * time
-end
Calcite
-start
10 IF (TOTAL_TIME <= parm(4)) THEN k = 10^parm(1)
```

```
20 IF (TOTAL_TIME > parm(4)) THEN k = 10^parm(2)
30 n = parm(3)
40 A = 100000 * 100.09 * KIN("Calcite")
50 R = (k *A * (1 - SR("Calcite"))^n)
60 SAVE R * time
-end
```

The if statements check to see which regiment of dissolution we are in, after about an hour we see in the experiments that the dissolution rate decreases and remains constant, more or less, for the remainder of the dissolution. This is achieved in the model by having two different equilibrium constants which slow or speed up the rate of dissolution to match whats expected.

This is just the modified rates blocked the following is the full modified model which has a few quality of life improvements.

[7]:
```
"""
The following python code is a conversion of AragoniteDissolution5. A PHREEQC␣
 ↪input file which
I wish to convert to PHREEQPY code.
"""

import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython.display import clear_output
import time as tim
import io
import sys
import os
import timeit

# Simple Python 3 compatibility adjustment.
if sys.version_info[0] == 2:
    range = xrange

MODE = 'dll'  # 'dll' or 'com'

if MODE == 'com':
    import phreeqpy.iphreeqc.phreeqc_com as phreeqc_mod
elif MODE == 'dll':
    import phreeqpy.iphreeqc.phreeqc_dll as phreeqc_mod
else:
    raise Exception('Mode "%s" is not defined use "com" or "dll".' % MODE)


def run_phreeqc(input_string):
    """
    Run PHREEQC simulation with the given input string.
    """
    phreeqc = phreeqc_mod.IPhreeqc()
    phreeqc.load_database(r"C:\Program Files (x86)\USGS\Phreeqc Interactive 3.7.
↪3-15968\database\wateq4f.DAT")
    phreeqc.run_string(input_string)
    return phreeqc


def extract_data(phreeqc, junkrow=1, i = 0):
    """
    Extract data from PHREEQC simulation results and save it to 'data.dat'.
    """
    # Get all selected output data as separate lists
    time = phreeqc.get_selected_output_column(0)
    pH = phreeqc.get_selected_output_column(1)
    SI_ara = phreeqc.get_selected_output_column(2)
    SI_cal = phreeqc.get_selected_output_column(3)
    Ara = phreeqc.get_selected_output_column(4)
    dAra = phreeqc.get_selected_output_column(5)
    Cal = phreeqc.get_selected_output_column(6)
    dCal = phreeqc.get_selected_output_column(7)

    while i < junkrow:
        time.pop(0)
        pH.pop(0)
        SI_ara.pop(0)
        SI_cal.pop(0)
        Ara.pop(0)
        dAra.pop(0)
        Cal.pop(0)
        dCal.pop(0)
        i += 1

    # Replace d with 'd' in the column headers
    header = "Time\tpH\tSI_ara\tSI_cal\tAra\tdAra\tCal\tdCal\n"

    # Save data to 'data.dat'
    with open("data1.dat", "w") as data_file:
        data_file.write(header)
        for i in range(len(time)):
            data_file.write(f"{float(time[i]):.16f}\t{float(pH[i]):.
↪16f}\t{float(SI_ara[i]):.16f}\t{float(SI_cal[i]):.16f}\t"
                                f"{float(Ara[i]):.16f}\t{float(dAra[i]):.
↪16f}\t{float(Cal[i]):.16f}\t{float(dCal[i]):.16f}\n")

    return time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal
```

```
def make_plots(time, data1, data2, label1, label2, x_label, y_label, Title,␣
 ↪legend_loc='lower right', fig_size=(10, 4.5), semi=False,
               compare=True, ylimit=False, xlimit=False, ylU=0, ylL=0, xlU=0,␣
 ↪xlL=0):
    """
    Create plots for the given data.

    Parameters:
        time (list): Time data for the x-axis.
        data1 (list): First dataset to plot.
        data2 (list): Second dataset to plot.
        label1 (str): Label for the first dataset.
        label2 (str): Label for the second dataset.
        x_label (str): Label for the x-axis.
        y_label (str): Label for the y-axis.
        legend_loc (str, optional): Location of the legend. Default is 'lower␣
 ↪right'.
        fig_size (tuple, optional): Size of the figure. Default is (10, 4.5).
        ylimit (Optional): if true will set limits.
        ylU and ylL (Optional): Set the upper and lower limits of the plots y␣
 ↪axis.
        xlimit (Optional): if true will set limits.
        xlU and xlL (Optional): Set the upper and lower limits of the plots x␣
 ↪axis.
    """

    fig, ax = plt.subplots(figsize=fig_size)
    ax.set_facecolor('grey')
    ax.set_xlabel(x_label, color='black')
    ax.set_ylabel(y_label, color='black')
    ax.tick_params(axis='x', colors='black')
    ax.tick_params(axis='y', colors='black')
    if compare:
        ax.set_title(Title, color='black')
        if semi:
            ax.semilogy(time, data1, color='Maroon', label=label1)
            ax.semilogy(time, data2, color='Pink', label=label2)
        else:
            ax.plot(time, data1, color='Maroon', label=label1)
            ax.plot(time, data2, color='Pink', label=label2)

        legend = ax.legend(loc=legend_loc, facecolor='grey')
        for text in legend.get_texts():
            text.set_color('black')
    else:
        ax.set_title(Title, color='black')
        if semi:
            ax.semilogy(time, data1, color='Maroon', label=label1)
        else:
            ax.plot(time, data1, color='Maroon', label=label1)

        legend = ax.legend(loc=legend_loc, facecolor='grey')
        legend.get_texts()[0].set_color('black')
    if ylimit:
        ax.set_ylim(ylL,ylU)
    if xlimit:
        ax.set_xlim(xlL,xlU)


    return fig

def plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal, pH, t_unit, TTime,␣
 ↪TSteps):
    """
    Plot the data extracted from PHREEQC simulation.

    Parameters:
        time (list): Time data.
        SI_ara (list): Saturation Index data for aragonite.
        SI_cal (list): Saturation Index data for calcite.
        Ara (list): Aragonite data.
        dAra (list): Aragonite dissolution rate data.
        Cal (list): Calcite data.
        dCal (list): Calcite dissolution rate data.
        t_unit (str): Time unit for the x-axis.
        TTime = total time
        TSteps = Total steps
    """
    Null = [0] * len(time)
    totmol = Ara[0]
    stepsize = TTime / TSteps

    _ara = [(a / totmol) for a in Ara]
    _cal = [1 -  for  in _ara]

    # Initialize the progress bar
    progress_bar = tqdm(total=len(dAra))

    for i in range(len(dAra)):
        time[i] = time[i] / 3600
        dAra[i] = dAra[i] / stepsize
        dCal[i] = dCal[i] / stepsize
        # Uncomment the following if you want to have a progress bar, will␣
↪impact performance!
        # Update the progress bar
        progress_bar.set_description(f"Processing data point: {i + 1}/
↪{len(dAra)}")
        progress_bar.update(1)

    # Close the progress bar
    progress_bar.close()

    # Create figures for each plot
    fig1 = make_plots(time, Ara, Cal, "Aragonite", "Calcite", "Time (" + t_unit␣
↪+ ")", "Amount of Solid (Mol)", "Aragonite and Calcite Levels Over Time",␣
↪semi=True)
    fig2 = make_plots(time, Ara, Cal, "Ara", "Calcite", "Time (" + t_unit + ")",␣
↪"Amount of solid Aragonite", "Amount of Aragonite and Calcite", compare=True)
    fig3 = make_plots(time, SI_ara, SI_cal, "Aragonite", "Calcite", "Time(" +␣
↪t_unit + ")", "Saturation Index", "Saturations Over Time")
    fig4 = make_plots(time, _ara, _cal, "Aragonite", "Calcite", "Time (" +␣
↪t_unit + ")", "Mole Fraction", "Relative Levels Over Time", legend_loc='upper␣
↪right')
    fig5 = make_plots(time, pH, Null, "pH", "Null", "Time (" + t_unit + ")",␣
↪"pH", "pH of Solution", compare=False)
    fig6 = make_plots(time, dAra, dCal, "Aragonite", "Calcite", "Time (" +␣
↪t_unit + ")", "Rate Dissolved (mol/h)", "Rate of Change Over Time",␣
↪legend_loc='upper right')

    # Save figures as images in memory
    buffer1 = io.BytesIO()
    buffer2 = io.BytesIO()
    buffer3 = io.BytesIO()
    buffer4 = io.BytesIO()
    buffer5 = io.BytesIO()
    buffer6 = io.BytesIO()

    fig1.savefig(buffer1, format='png')
    fig2.savefig(buffer2, format='png')
    fig3.savefig(buffer3, format='png')
    fig4.savefig(buffer4, format='png')
    fig5.savefig(buffer5, format='png')
    fig6.savefig(buffer6, format='png')

    # Rewind the buffer and read the image data
    buffer1.seek(0)
    buffer2.seek(0)
    buffer3.seek(0)
    buffer4.seek(0)
    buffer5.seek(0)
    buffer6.seek(0)

    image1 = plt.imread(buffer1)
    image2 = plt.imread(buffer2)
    image3 = plt.imread(buffer3)
    image4 = plt.imread(buffer4)
    image5 = plt.imread(buffer5)
    image6 = plt.imread(buffer6)

    # Create a 3x2 grid to display the plots side by side
    fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15, 15))
    fig.suptitle("Plotted Data", fontsize=16)

    # Adjust individual subplot sizes to make the images larger
    axes[0, 0].imshow(image1)
    axes[0, 0].axis('off')
    axes[0, 0].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[0, 1].imshow(image2)
    axes[0, 1].axis('off')
    axes[0, 1].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[1, 0].imshow(image3)
    axes[1, 0].axis('off')
    axes[1, 0].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[1, 1].imshow(image4)
    axes[1, 1].axis('off')
    axes[1, 1].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[2, 0].imshow(image5)
    axes[2, 0].axis('off')
    axes[2, 0].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[2, 1].imshow(image6)
    axes[2, 1].axis('off')
    axes[2, 1].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space
```

```
    # Adjust spacing
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])


    # Close the solo plots
    plt.close(fig1)
    plt.close(fig2)
    plt.close(fig3)
    plt.close(fig4)
    plt.close(fig5)
    plt.close(fig6)

    # Show the grid plot
    plt.show()

def Generate_Solutions(Name, temp = "20", pH = "7", units = "mmol/L", density =␣
↪"1",alkalinity = 0, mass = "1"):
    """
    This function generates a solution which will be saved and returned to main.
    Name == Name of soltuion
    Temp == temperature of solution
    pH == pH of solution
    units == Units of the concentration
    Density == Density of solvent
    mass= = mass of solvent in kg
    """

    Solution = (
        "SOLUTION " + Name + "\n"
        + "    temp        "+ temp + "\n"
        + "    pH          " + pH + "\n"
        + "    units       "+ units + "\n"
        + "    density     " + density+ "\n"
        + "    Alkalinity " + alkalinity + "\n"
        + "    -water      " + mass
        + "\n\n" )

    return Solution

def Mixer(Name, Ratio_1 = 0.5):
    """
    This function generates a mixed solution based on the added components and␣
↪their ratios to one another.
    Name == Name of newly mixed solution
    Ratio_1 and Ratio_2 are the amounts of each solution in proportion to each␣
↪other.
    S_ means that the variable is a string.
    """
    Ratio_2 = 1 - Ratio_1
    S_Ratio_1 = str(Ratio_1)
    S_Ratio_2 = str(Ratio_2)

    Mixed_sol = (
        "MIX " + Name + "\n"
        + "     1 " + S_Ratio_1 + "\n"
        + "     2 " + S_Ratio_2 + "\n"
        + "SAVE Solution 3"
        + "\n\n"
    )
    return Mixed_sol

def Equilibrate(Name, Mineral, CO2 = ["CO2", "-3.5", "10"], Target_SI = "0.0",␣
 ↪Addsolid = "0.0", CO2eq = False):
    """
    This function calls and sets the equilibrium_phases block.
    Name == The name of the final solution.
    Mineral == Name of the mineral
    Target_SI == The desired saturation of the mineral
    Addsolid == The ammount of solid added to acheive this

    """
    if CO2eq == False:
        Eq = (
            "EQUILIBRIUM_PHASES " + Name + "\n"
            + "     " + Mineral + " " + Target_SI + " " + Addsolid + "\n"
            + "SAVE Solution " + Name
            + "\n"
        )
    elif CO2eq == True:
        Eq = (
            "EQUILIBRIUM_PHASES " + Name + "\n"
            + "     " + Mineral + " " + Target_SI + " " + Addsolid + "\n"
            + "     " + CO2[0]  + " " + CO2[1] + " " + CO2[2] + "\n"
            + "SAVE Solution " + Name
            + "\n"
        )

    return Eq

def Set_Rates(Name, Amount = 2, Minerals = ["Aragonite" , "Calcite"],␣
 ↪Surface_Area = [100000, 100000], Molar_mass = [100.09, 100.09]):
    """
    The following sets up the RATES block.
    Name == The name of the rates block. I just number them.
    Amount == Number of minerals which need rates. It is assumed they follow a␣
↪similar type of equation
    Minerals == The names of the minerals
    Surface_Area == The Area per gram of Solid
    Molar_mass == The molar mass of solid

    """
    ###The following initializes the countner and sets the header for the block
    i = 0
    Rates1 = "RATES " + Name + "\n"
    Rates2 = ""

    ###Iterates to make each segment of the rates block
    while i < Amount:
        mineral = Minerals[i]
        molmass = Molar_mass[i]
        SA = Surface_Area[i]
        TempRate = ("    " + str(mineral) + "\n"
                + "    -start""" + "\n"
                + "    10 IF (TOTAL_TIME <= parm(4)) THEN k = 10^parm(1)" + "\n"
                + "    20 IF (TOTAL_TIME > parm(4)) THEN k = 10^parm(2)" + "\n"
                + "    30 n = parm(3)" + "\n"
                + "    40 A = " + str(SA)  +  " * " + str(molmass)  + ' * KIN("'␣
↪+ str(mineral) + '")' + "\n"
                + '    50 R = (k *A *  (1 - SR("' + mineral + '"))^n)' + "\n"
                + "    60 SAVE R * time" + "\n"
                + "    -end" + "\n"
        )
        Rates2 += TempRate
        i += 1
    Rates = Rates1 + Rates2 + "\n\n"
    return Rates

def Set_Kinetics(Name, Solution, Amount = 2, Minerals = ["Aragonite"␣
↪,"Calcite"], Formulas = ["CaCO3 1", "CaCO3 1"], masses = ["0", "0"],
                 Orders = ["3","5"], EqConstants1 = ["-8.34","-8.48"],␣
↪EqConstants2 = ["-10.19","-10.33"], ChngTime = "5040", Tolerance = "1e-08",
                 Time = "100", Steps = "1000", unit = "hour"):
    """
    This function sets up and initializes the kinetics block.
    Name == Name of this block, I use numbers
    Solution == The solution the kniteics occurs in (REQUIRED TO RUN MAKE SURE␣
↪TO DEFINE)
    Amount == The number of minerals which will react
    Minerals == The names of the minerals
    Formulas == The Chemical formula for the compounds
    mass = mass of solid
    Orders == the orders of each reactions
    EqConstants1 == the Constants for the first part of the reaction
    EqConstants2 == The constants for the second part of the reaction
    ChngTime == The time which the first and second parts of the reaction change.
    Tolerance == The allowed error
    Time == Timespan of the reaction
    Steps == number of steps
    unit == units of time
    """

    ###Initialize the counter and the first line of the block
    i = 0
    Kinetics1 = "KINETICS " + Name + "\n"
    Kinetics2 = ""

    ###Define the loop
    while i < Amount:
        mineral = Minerals[i]
        order = Orders[i]
        EqConstant1 = EqConstants1[i]
        EqConstant2 = EqConstants2[i]
        Formula = Formulas[i]
        mass = masses[i]

        tempkinetics = ("    " + str(mineral) + "\n"
                    + "        -formula  " + str(Formula) + "\n"
                    + "        -m0       " + str(mass) + "\n"
                    + "        -parms    " + str(EqConstant1) + " " +␣
↪str(EqConstant2) + " " + str(order) + " " + ChngTime + "\n"
                    + "        -tol      " + Tolerance + "\n"
        )
        Kinetics2 += tempkinetics
        i += 1

    Kinetics3 = ( "-steps    " + Time + " " + unit + " " + "in " + Steps + "\n"
             + "-step_divide 1" + "\n"
             + "-runge_kutta 6" + "\n"
             + "-bad_step_max 500" + "\n"
             + "INCREMENTAL_REACTIONS true" + "\n"
             + "\n"
             + "USE Solution " + Solution  + "\n"
    )

    Kinetics = Kinetics1 + Kinetics2 + Kinetics3

    return Kinetics
```

```
def Generate_Input(Solution_1 = "", Solution_2 = "", Solution_3 = "", Eq1 = "",␣
↪Eq2 = "", Rates = "", Kinetics = ""):
    input_string = (
    Solution_1 +
    Eq1 +
    Solution_2 +
    Eq2 +
    Solution_3 +
    """END \n\n"""
    + Rates
    + Kinetics +
    """

    SELECTED_OUTPUT 1
        -high_precision       true
        -reset                false
        -time                 true
        -saturation_indices   Aragonite  Calcite
        -kinetic_reactants    Aragonite  Calcite
        -pH                   true
    END

    """)

    return input_string

def main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
↪alkalinity,
         Eqconstants1, Eqconstants2, Orders, chngTime,␣
↪Surface_Areas,Molar_masses,
         Time, Steps, LUnit, Tolerance):

    start_time = tim.time()
    #Define the units of time
    t_unit = "hours"

    """
    Main function to execute the PHREEQC simulation and plot the results.
    """
    plt.style.use('default')
    ### Make and mix Solutions
    Solution_1 = Generate_Solutions(Names[0], temp = temps[0], pH = pHs[0],␣
↪units = units[0], alkalinity = alkalinity[0], mass = Wmasses[0])
    Solution_2 = Generate_Solutions(Names[1], temp = temps[1], pH = pHs[1],␣
↪units = units[1], alkalinity = alkalinity[0], mass = Wmasses[1])
    Solution_3 = Mixer(Names[0], Ratio_1 = 0.5)
    Eq1 = Equilibrate(Names[0], Minerals[0], Addsolid = Smasses[0], CO2 =␣
 ↪CO2s[0], CO2eq = CO2eq[0])
    Eq2 = Equilibrate(Names[0], Minerals[1], Addsolid = Smasses[1], CO2 =␣
 ↪CO2s[1], CO2eq = CO2eq[1])

    ### Set up the kinetics
    Rates = Set_Rates(Names[0], Amount = len(Minerals), Minerals = Minerals,␣
 ↪Surface_Area = Surface_Areas, Molar_mass = Molar_masses)
    Kinetics = Set_Kinetics(Name = Names[0], Solution = "3", Amount = 2,␣
 ↪Minerals = Minerals, masses = Smasses,
                            Orders = Orders, EqConstants1 = Eqconstants1,␣
 ↪EqConstants2 = Eqconstants2, ChngTime = chngTime,
                            Tolerance = Tolerance, Time = Time, Steps = Steps,␣
 ↪unit = LUnit)

    input_string = Generate_Input(Solution_1 = Solution_1, Solution_2 =␣
 ↪Solution_2, Solution_3 = Solution_3, Eq1 = Eq1, Eq2 = Eq2, Rates = Rates,␣
 ↪Kinetics = Kinetics)
    phreeqc = run_phreeqc(input_string)
    time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal = extract_data(phreeqc,␣
 ↪junkrow = 1)
    plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal, pH, t_unit, TTime =␣
 ↪float(Time), TSteps = float(Steps))

    # Show the plots
    plt.show()
    end_time = tim.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
    return time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string

if __name__ == '__main__':

    ###Solution information
    Minerals = ["Aragonite", "Calcite"]
    pHs = ["6.5","6.5"]
    temps = ["18", "18"]
    Names = ["1", "2", "3"]
    units = ["mmol/L", "mmol/L"]
    Wmasses = ["0.01", "0.01"] #Water masses
    Smasses = ["0.0004995", "0.0004995"] #Masses of Solids for equlibrate␣
 ↪function
    CO2s = [["CO2(g)", "-3.5", "10"], ["CO2(g)", "-3.5", "10"]]
    CO2eq = [True, True]
    alkalinity = ["5", "5"]
    ###Kinetics information
    Eqconstants1 = ["-8.34","-8.48"]
    Eqconstants2 = ["-10.19","-10.33"]
    Orders = ["3", "5"]
    chngTime = "5040" #Time when the Eq. Constants change
    Surface_Areas = [100000, 100000]
    Molar_masses = [100.09, 100.09]

    ###Loop information
    Time = "150"
    Steps = "10000"
    LUnit = "hour" #Units of time
    Tolerance = "1e-08"

    (time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string) =␣
↪main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
↪alkalinity,
                                                                        ␣
↪Eqconstants1, Eqconstants2, Orders, chngTime, Surface_Areas,
                                                                        ␣
↪Molar_masses, Time, Steps, LUnit, Tolerance)
```

The are differences to almost all parts to this code which i will explain in detail. There are also functions which i have defined which when called form the components to the input file which then get assembled and sent to the run_phreeqc command.

The functions responsible for creating the input file are as follows,

- Generate_Solutions()
- Mixer()
- Equilibrate()
- Set_Rates()
- Set_Kinetics()
- Generate_Input()

I shall attempt to explain these functions in the following,

5.3.1 Generate_Solutions()

The purpose of this function is to take in the relevant info, contained in character strings, and assemble them into a functional phreeqc Solutions datablock.

It does this by concatenation of these strings into the proper format. It then saves and returns this new block to the main program.

The following is the Generate_Solutions function,

[11]:
```
def Generate_Solutions(Name, temp = "20", pH = "7", units = "mmol/L", density =␣
 ↪"1", mass = "1"):
    """
    This function generates a solution which will be saved and returned to main.
    Name == Name of soltuion
    Temp == temperature of solution
    pH == pH of solution
    units == Units of the concentration
    Density == Density of solvent
    mass= = mass of solvent in kg
    """

    Solution = (
        "SOLUTION " + Name + "\n"
        + "    temp       "+ temp + "\n"
        + "    pH         " + pH + "\n"
        + "    units      "+ units + "\n"
        + "    density    " + density+ "\n"
        + "    -water     " + mass
        + "\n\n" )

    return Solution
```

This function takes in the data for,

- Name of solution
- The temperature of the solution
- The pH of the solution
- The units used for added elements the solution
- The density of the solution
- The mass of water, which is the solvent

5.3.2 Mixer()

This function assemble the Mix datablock. It accepts two variables, name, the name of the block, and ratio_1, which is the amount of solution 1 which will be mixed in proportion to the amount of the other solution. The default being 0.5 which is half and half.

The mixer command is as follows,

[11]:
```
def Mixer(Name, Ratio_1 = 0.5):
    """
    This function generates a mixed solution based on the added components and␣
↪their ratios to one another.
    Name == Name of newly mixed solution
    Ratio_1 and Ratio_2 are the amounts of each solution in proportion to each␣
↪other.
    S_ means that the variable is a string.

    """
    Ratio_2 = 1 - Ratio_1
    S_Ratio_1 = str(Ratio_1)
    S_Ratio_2 = str(Ratio_2)

    Mixed_sol = (
        "MIX " + Name + "\n"
        + "    1 " + S_Ratio_1 + "\n"
        + "    2 " + S_Ratio_2 + "\n"
        + "SAVE Solution 3" + "\n"
        + "\n\n"
    )
    return Mixed_sol
```

5.3.3 Equilibrate() NOT MOST UPDATED

The function sets up Equilibrium_Phases datablock which sets the desired minerals to specific saturations. This function takes in four variables,

- Name – The name of the block
- Mineral – The mineral to equalize
- Target_SI – The target SI for the mineral
- Addsolid – The amount of solid added to help reach this

5.3.4 Set_Rates()

The set rates function sets up the rates equation. It requires five variables which correspond to various parameters for the rates. They include,

- Name – The name of the block
- Amount – the amount of minerals which have rates
- Minerals – An array containing the name of each mineral
- Surface_Area – An array of the surface area in cm/g of each mineral
- Molar_mass – An array of each molar mass of the minerals

The block is assembled through a loop which assembles the Rates block until all minerals are accounted for. The following is the Set_Rates function,

```
[11]: def Set_Rates(Name, Amount = 2, Minerals = ["Aragonite" , "Calcite"],␣
      ↪Surface_Area = [100000, 100000], Molar_mass = [100.09, 100.09]):
          """
          The following sets up the RATES block.
          Name == The name of the rates block. I just number them.
          Amount == Number of minerals which need rates. It is assumed they follow a␣
      ↪similar type of equation
          Minerals == The names of the minerals
          Surface_Area == The Area per gram of Solid
          Molar_mass == The molar mass of solid

          """
          ###The following initializes the countner and sets the header for the block
          i = 0
          Rates1 = "RATES " + Name + "\n"
          Rates2 = ""

          ###Iterates to make each segment of the rates block
          while i < Amount:
              mineral = Minerals[i]
              molmass = Molar_mass[i]
              SA = Surface_Area[i]
              TempRate = ("    " + str(mineral) + "\n"
                          + "    -start""" + "\n"
                          + "    10 IF (TOTAL_TIME <= parm(4)) THEN k = 10^parm(1)" + "\n"
                          + "    20 IF (TOTAL_TIME > parm(4)) THEN k = 10^parm(2)" + "\n"
                          + "    30 n = parm(3)" + "\n"
                          + "    40 A = " + str(SA)  +  " * " + str(molmass)  + ' * KIN("'␣
      ↪+ str(mineral) + '")' + "\n"
                          + '    50 R = (k *A *  (1 - SR("' + mineral + '"))^n)' + "\n"
                          + "    60 SAVE R * time" + "\n"
                          + "    -end" + "\n"
              )
              Rates2 += TempRate
              i += 1
          Rates = Rates1 + Rates2 + "\n\n"
          return Rates
```

5.3.5 Set_Kinetics

This function takes fourteen variables,

- Name – The name of the block
- Solution – The name of the solution where the kinetics occurs
- Amount – The number of minerals

- Formulas – An array containing chemical formulas for each minerals
- masses – The amount of each compound initially contained in an array
- Orders – An array of containing the order of dissolution for each compound
- EqConstants1 – An array counting the equilibrium constants for each compound for the first phase of the dissolution
- EqConstants2 – An array containing the equilibrium constants for each compound for the second phase of the dissolution
- ChngTime – Sets the time where the dissolution swaps from phase 1 to 2
- Tolerance – The tolerance for error
- Time – The timespan the code will simulate
- Steps – The number of steps the timespan will be divided across
- Unit – The units of time

The following is the Set_Kinetics function,

[11]:
```
def Set_Kinetics(Name, Solution, Amount = 2, Minerals = ["Aragonite"␣
↪,"Calcite"], Formulas = ["CaCO3 1", "CaCO3 1"], masses = ["0", "0"],
                 Orders = ["3","5"], EqConstants1 = ["-8.34","-8.48"],␣
↪EqConstants2 = ["-10.19","-10.33"], ChngTime = "5040", Tolerance = "1e-08",
                 Time = "100", Steps = "1000", unit = "hour"):
    """
    This function sets up and initializes the kinetics block.
    Name == Name of this block, I use numbers
    Solution == The solution the kniteics occurs in (REQUIRED TO RUN MAKE SURE␣
↪TO DEFINE)
    Amount == The number of minerals which will react
    Minerals == The names of the minerals
    Formulas == The Chemical formula for the compounds
    mass = mass of solid
    Orders == the orders of each reactions
    EqConstants1 == the Constants for the first part of the reaction
    EqConstants2 == The constants for the second part of the reaction
    ChngTime == The time which the first and second parts of the reaction change.
    Tolerance == The allowed error
    Time == Timespan of the reaction
    Steps == number of steps
    unit == units of time
    """

    ###Initialize the counter and the first line of the block
    i = 0
    Kinetics1 = "KINETICS " + Name + "\n"
    Kinetics2 = ""

    ###Define the loop
    while i < Amount:
        mineral = Minerals[i]
        order = Orders[i]
        EqConstant1 = EqConstants1[i]
        EqConstant2 = EqConstants2[i]
        Formula = Formulas[i]
        mass = masses[i]

        tempkinetics = ("     " + str(mineral) + "\n"
                        + "        -formula    " + str(Formula) + "\n"
                        + "        -m0         " + str(mass) + "\n"
                        + "        -parms      " + str(EqConstant1) + " " +␣
  ↪str(EqConstant2) + " " + str(order) + " " + ChngTime + "\n"
                        + "        -tol        " + Tolerance + "\n"
        )
        Kinetics2 += tempkinetics
        i += 1

    Kinetics3 = ( "-steps     " + Time + " " + unit + " " + "in " + Steps + "\n"
             + "-step_divide 1" + "\n"
             + "-runge_kutta 6" + "\n"
             + "-bad_step_max 500" + "\n"
             + "INCREMENTAL_REACTIONS true" + "\n"
             + "\n"
             + "USE Solution " + Solution  + "\n"
    )

    Kinetics = Kinetics1 + Kinetics2 + Kinetics3

    return Kinetics
```

This function also utilizes a loop to built the consecutive parts to the block before saving it and sending it to main.

5.3.6 Generate_Input()

This function takes the outputs of the previous functions and will add them together to combine into a single input called Input_String which is then sent into the Run_Phreeqc function and executed. It is as follows,

[11]:
```
def Generate_Input(Solution_1, Solution_2, Solution_3, Eq1, Eq2, Rates,␣
 ↪Kinetics):
    input_string = (
    Solution_1 +
    Eq1 +
    Solution_2 +
    Eq2 +
    Solution_3 +
    """
    END
    """
    + Rates
    + Kinetics +
    """

    SELECTED_OUTPUT 1
        -high_precision       true
        -reset                false
        -time                 true
        -saturation_indices   Aragonite  Calcite
        -kinetic_reactants    Aragonite  Calcite
        -pH                   true
    END

    """)

    return input_string
```

The input to this function is simply the output of each function that preceded it. It takes these and fits them into the desired order and outputs it. It also adds a SELECTED_OUTPUT block which ensures the desired data is collected.

5.4 Other Changes

There were further changes which i made to improve the user friendliness of the phreeqpy code and again I shall attempt to go into what these changes are. The parts of the model which I have adjusted are, The main function and the plotting function.

5.4.1 Main() 2.0

The main function previously only executed the other functions in the correct order which i specified. Now however it does that AND it gives them the parameters which will make up the input files. The new main function is as follows,

[11]:
```
def main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
 ↪alkalinity,
         Eqconstants1, Eqconstants2, Orders, chngTime,␣
 ↪Surface_Areas,Molar_masses,
         Time, Steps, LUnit, Tolerance):
```

```
    start_time = tim.time()
    #Define the units of time
    t_unit = "hours"

    """
    Main function to execute the PHREEQC simulation and plot the results.
    """
    plt.style.use('default')
    ### Make and mix Solutions
    Solution_1 = Generate_Solutions(Names[0], temp = temps[0], pH = pHs[0],␣
↪units = units[0], alkalinity = alkalinity[0], mass = Wmasses[0])
    Solution_2 = Generate_Solutions(Names[1], temp = temps[1], pH = pHs[1],␣
↪units = units[1], alkalinity = alkalinity[0], mass = Wmasses[1])
    Solution_3 = Mixer(Names[0], Ratio_1 = 0.5)
    Eq1 = Equilibrate(Names[0], Minerals[0], Addsolid = Smasses[0], CO2 =␣
↪CO2s[0], CO2eq = CO2eq[0])
    Eq2 = Equilibrate(Names[0], Minerals[1], Addsolid = Smasses[1], CO2 =␣
↪CO2s[1], CO2eq = CO2eq[1])

    ### Set up the kinetics
    Rates = Set_Rates(Names[0], Amount = len(Minerals), Minerals = Minerals,␣
↪Surface_Area = Surface_Areas, Molar_mass = Molar_masses)
    Kinetics = Set_Kinetics(Name = Names[0], Solution = "3", Amount = 2,␣
↪Minerals = Minerals, masses = Smasses,
                            Orders = Orders, EqConstants1 = Eqconstants1,␣
↪EqConstants2 = Eqconstants2, ChngTime = chngTime,
                            Tolerance = Tolerance, Time = Time, Steps = Steps,␣
↪unit = LUnit)

    input_string = Generate_Input(Solution_1 = Solution_1, Solution_2 =␣
↪Solution_2, Solution_3 = Solution_3, Eq1 = Eq1, Eq2 = Eq2, Rates = Rates,␣
↪Kinetics = Kinetics)
    phreeqc = run_phreeqc(input_string)
    time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal = extract_data(phreeqc,␣
↪junkrow = 1)
    plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal, pH, t_unit, TTime =␣
↪float(Time), TSteps = float(Steps))

    # Show the plots
    plt.show()
    end_time = tim.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
    return time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string
```

```
if __name__ == '__main__':

    ###Solution information
    Minerals = ["Aragonite", "Calcite"]
    pHs = ["6.5","6.5"]
    temps = ["18", "18"]
    Names = ["1", "2", "3"]
    units = ["mmol/L", "mmol/L"]
    Wmasses = ["0.01", "0.01"] #Water masses
    Smasses = ["0.0004995", "0.0004995"] #Masses of Solids for equlibrate␣
 ↪function
    CO2s = [["CO2(g)", "-3.5", "10"], ["CO2(g)", "-3.5", "10"]]
    CO2eq = [True, True]
    alkalinity = ["5", "5"]


    ###Kinetics information
    Eqconstants1 = ["-8.34","-8.48"]
    Eqconstants2 = ["-10.19","-10.33"]
    Orders = ["3", "5"]
    chngTime = "5040" #Time when the Eq. Constants change
    Surface_Areas = [100000, 100000]
    Molar_masses = [100.09, 100.09]

    ###Loop information
    Time = "150"
    Steps = "10000"
    LUnit = "hour" #Units of time
    Tolerance = "1e-08"

    (time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string) =␣
 ↪main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
 ↪alkalinity,
                                                                       ␣
 ↪Eqconstants1, Eqconstants2, Orders, chngTime, Surface_Areas,
                                                                       ␣
 ↪Molar_masses, Time, Steps, LUnit, Tolerance)
```

The main function now accepts input which is specified before execution. There are three sections of parameters, which i wont restate what they are they're the inputs for all the functions anyways. There is the Solution info, The kinetics info, and the loop info. These values can be modified and will change the resulting input file. The benefit is that all the needed parameters can be modified in on spot so one need not look for the parameter to change anymore, or type too much as the input file is assembled automatically. The parameters are also placed in lists with list location corresponding to each solution an/or mineral.

5.4.2 Plotting

Plotting is handled differently now too. The plotting is now handled by a function called make plots and are now plotted in a grid system. The two plotting functions are,

```
[11]: def make_plots(time, data1, data2, label1, label2, x_label, y_label, Title,␣
      ↪legend_loc='lower right', fig_size=(10, 4.5), semi=False,
                    compare=True, ylimit=False, xlimit=False, ylU=0, ylL=0, xlU=0,␣
      ↪xlL=0):
          """
          Create plots for the given data.

          Parameters:
              time (list): Time data for the x-axis.
              data1 (list): First dataset to plot.
              data2 (list): Second dataset to plot.
              label1 (str): Label for the first dataset.
              label2 (str): Label for the second dataset.
              x_label (str): Label for the x-axis.
              y_label (str): Label for the y-axis.
              legend_loc (str, optional): Location of the legend. Default is 'lower␣
      ↪right'.
              fig_size (tuple, optional): Size of the figure. Default is (10, 4.5).
              ylimit (Optional): if true will set limits.
              ylU and ylL (Optional): Set the upper and lower limits of the plots y␣
      ↪axis.
              xlimit (Optional): if true will set limits.
              xlU and xlL (Optional): Set the upper and lower limits of the plots x␣
      ↪axis.
          """

          fig, ax = plt.subplots(figsize=fig_size)
          ax.set_facecolor('grey')
          ax.set_xlabel(x_label, color='black')
          ax.set_ylabel(y_label, color='black')
          ax.tick_params(axis='x', colors='black')
          ax.tick_params(axis='y', colors='black')
          if compare:
              ax.set_title(Title, color='black')
              if semi:
                  ax.semilogy(time, data1, color='Maroon', label=label1)
                  ax.semilogy(time, data2, color='Pink', label=label2)
              else:
                  ax.plot(time, data1, color='Maroon', label=label1)
                  ax.plot(time, data2, color='Pink', label=label2)

              legend = ax.legend(loc=legend_loc, facecolor='grey')
```

```
        for text in legend.get_texts():
            text.set_color('black')
    else:
        ax.set_title(Title, color='black')
        if semi:
            ax.semilogy(time, data1, color='Maroon', label=label1)
        else:
            ax.plot(time, data1, color='Maroon', label=label1)

        legend = ax.legend(loc=legend_loc, facecolor='grey')
        legend.get_texts()[0].set_color('black')
    if ylimit:
        ax.set_ylim(ylL,ylU)
    if xlimit:
        ax.set_xlim(xlL,xlU)


    return fig

def plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal, pH, t_unit, TTime,␣
 ↪TSteps):
    """
    Plot the data extracted from PHREEQC simulation.

    Parameters:
        time (list): Time data.
        SI_ara (list): Saturation Index data for aragonite.
        SI_cal (list): Saturation Index data for calcite.
        Ara (list): Aragonite data.
        dAra (list): Aragonite dissolution rate data.
        Cal (list): Calcite data.
        dCal (list): Calcite dissolution rate data.
        t_unit (str): Time unit for the x-axis.
        TTime = total time
        TSteps = Total steps
    """
    Null = [0] * len(time)
    totmol = Ara[0]
    stepsize = TTime / TSteps

    _ara = [(a / totmol) for a in Ara]
    _cal = [1 -  for  in _ara]

    # Initialize the progress bar
    progress_bar = tqdm(total=len(dAra))

    for i in range(len(dAra)):
        time[i] = time[i] / 3600
        dAra[i] = dAra[i] / stepsize
        dCal[i] = dCal[i] / stepsize

        # Uncomment the following if you want to have a progress bar, will␣
↪impact performance!
        # Update the progress bar
        progress_bar.set_description(f"Processing data point: {i + 1}/
↪{len(dAra)}")
        progress_bar.update(1)

    # Close the progress bar
    progress_bar.close()

    # Create figures for each plot
    fig1 = make_plots(time, Ara, Cal, "Aragonite", "Calcite", "Time (" + t_unit␣
↪+ ")", "Amount of Solid (Mol)", "Aragonite and Calcite Levels Over Time",␣
↪semi=True)
    fig2 = make_plots(time, Ara, Cal, "Ara", "Calcite", "Time (" + t_unit + ")",␣
↪"Amount of solid Aragonite", "Amount of Aragonite and Calcite", compare=True)
    fig3 = make_plots(time, SI_ara, SI_cal, "Aragonite", "Calcite", "Time(" +␣
↪t_unit + ")", "Saturation Index", "Saturations Over Time")
    fig4 = make_plots(time, _ara, _cal, "Aragonite", "Calcite", "Time (" +␣
↪t_unit + ")", "Mole Fraction", "Relative Levels Over Time", legend_loc='upper␣
↪right')
    fig5 = make_plots(time, pH, Null, "pH", "Null", "Time (" + t_unit + ")",␣
↪"pH", "pH of Solution", compare=False)
    fig6 = make_plots(time, dAra, dCal, "Aragonite", "Calcite", "Time (" +␣
↪t_unit + ")", "Rate Dissolved (mol/h)", "Rate of Change Over Time",␣
↪legend_loc='upper right')

    # Save figures as images in memory
    buffer1 = io.BytesIO()
    buffer2 = io.BytesIO()
    buffer3 = io.BytesIO()
    buffer4 = io.BytesIO()
    buffer5 = io.BytesIO()
    buffer6 = io.BytesIO()

    fig1.savefig(buffer1, format='png')
    fig2.savefig(buffer2, format='png')
    fig3.savefig(buffer3, format='png')
    fig4.savefig(buffer4, format='png')
    fig5.savefig(buffer5, format='png')
    fig6.savefig(buffer6, format='png')
    # Rewind the buffer and read the image data
    buffer1.seek(0)
    buffer2.seek(0)
    buffer3.seek(0)
    buffer4.seek(0)
    buffer5.seek(0)
    buffer6.seek(0)

    image1 = plt.imread(buffer1)
    image2 = plt.imread(buffer2)
    image3 = plt.imread(buffer3)
    image4 = plt.imread(buffer4)
    image5 = plt.imread(buffer5)
    image6 = plt.imread(buffer6)

    # Create a 3x2 grid to display the plots side by side
    fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(15, 15))
    fig.suptitle("Plotted Data", fontsize=16)

    # Adjust individual subplot sizes to make the images larger
    axes[0, 0].imshow(image1)
    axes[0, 0].axis('off')
    axes[0, 0].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[0, 1].imshow(image2)
    axes[0, 1].axis('off')
    axes[0, 1].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[1, 0].imshow(image3)
    axes[1, 0].axis('off')
    axes[1, 0].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[1, 1].imshow(image4)
    axes[1, 1].axis('off')
    axes[1, 1].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[2, 0].imshow(image5)
    axes[2, 0].axis('off')
    axes[2, 0].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    axes[2, 1].imshow(image6)
    axes[2, 1].axis('off')
    axes[2, 1].set_aspect('auto')  # Set the aspect ratio to auto to fill the␣
↪available space

    # Adjust spacing
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])


    # Close the solo plots
    plt.close(fig1)
    plt.close(fig2)
    plt.close(fig3)
    plt.close(fig4)
    plt.close(fig5)
    plt.close(fig6)

    # Show the grid plot
    plt.show()
```

The function make_plots now takes in the data and information for the title, axis labels, if data is plotted together and if the y axis is a log axis and generates and saves the plots in variables labeled fig 1-6. The plot_data function now takes the plots and saves and plots them in one figure where the plots are in a grid where one can modify the layout but i need to make it adjustable first. The data i find relevant is already outputted and is placed in 6 plots in a 2x3 grid. Limits for the plots can also be specified defined by the ylU(or L) and xlU(or L) variables.

This currently is my most updated model and further modifications will be recorded later.

5.5 Further plotting changes

I made some further modifications to plotting to ease the adding, subtraction, and modification of plotting. The new code is as follows.

[11]:
```

def make_plots(time, data1, data2, label1, label2, x_label, y_label, Title,␣
↪legend_loc='lower right', fig_size=(10, 4.5), semi=False,
               compare=True, Limits = False, ylU=0, ylL=0, xlU=0, xlL=0):
    """
    Create plots for the given data.

    Parameters:
        time (list): Time data for the x-axis.
        data1 (list): First dataset to plot.
        data2 (list): Second dataset to plot.
        label1 (str): Label for the first dataset.
        label2 (str): Label for the second dataset.
        x_label (str): Label for the x-axis.
        y_label (str): Label for the y-axis.
        legend_loc (str, optional): Location of the legend. Default is 'lower␣
↪right'.
        fig_size (tuple, optional): Size of the figure. Default is (10, 4.5).
        ylimit (Optional): if true will set limits.
        ylU and ylL (Optional): Set the upper and lower limits of the plots y␣
↪axis.
        xlimit (Optional): if true will set limits.
        xlU and xlL (Optional): Set the upper and lower limits of the plots x␣
↪axis.
    """

    fig, ax = plt.subplots(figsize=fig_size)
    ax.set_facecolor('grey')
    ax.set_xlabel(x_label, color='black')
    ax.set_ylabel(y_label, color='black')
    ax.tick_params(axis='x', colors='black')
    ax.tick_params(axis='y', colors='black')
    if compare:
        ax.set_title(Title, color='black')
        if semi:
            ax.semilogy(time, data1, color='Maroon', label=label1)
            ax.semilogy(time, data2, color='Pink', label=label2)
        else:
            ax.plot(time, data1, color='Maroon', label=label1)
            ax.plot(time, data2, color='Pink', label=label2)

        legend = ax.legend(loc=legend_loc, facecolor='grey')
        for text in legend.get_texts():
            text.set_color('black')
    else:
        ax.set_title(Title, color='black')
        if semi:
            ax.semilogy(time, data1, color='Maroon', label=label1)
        else:
            ax.plot(time, data1, color='Maroon', label=label1)

        legend = ax.legend(loc=legend_loc, facecolor='grey')
        legend.get_texts()[0].set_color('black')
    if Limits[0] == True:
        ax.set_xlim(xlL,xlU)
    if Limits[1] == True:
        ax.set_ylim(ylL,ylU)

    return fig
```

```
def plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal, pH, t_unit, xLimits,␣
 ↪yLimits, Limits, Titles, Labels, Legend, numplots, semi, compare, datas,␣
 ↪dictionary, TTime, TSteps):
    """
    Plot the data extracted from PHREEQC simulation. Also calculates the mole␣
 ↪fraction.

    Parameters:
        time (list): Time data.
        SI_ara (list): Saturation Index data for aragonite.
        SI_cal (list): Saturation Index data for calcite.
        Ara (list): Aragonite data.
        dAra (list): Aragonite dissolution rate data.
        Cal (list): Calcite data.
        dCal (list): Calcite dissolution rate data.
        t_unit (str): Time unit for the x-axis.
        TTime = total time
        TSteps = Total steps
    """

    #Calculate the mole fractions
    Null = [0] * len(time)
    totmol = Ara[0]
    stepsize = TTime / TSteps

    _ara = [(a / totmol) for a in Ara]
    # _Cal = [(c / totmol) for c in Cal]
    _cal = [1 -  for  in _ara]

    xlims = xLimits
    ylims = yLimits
    Limits = Limits
    Titles = Titles
    Labs = Labels

    lists = [Ara, Cal, dAra, dCal, _ara, _cal, SI_ara, SI_cal, pH]

    # Great globals dictionary
    for i in range(len(lists)):
        globals()[dictionary[i]] = lists[i]

    #Initialize the number of rows and columns
    nrows = int(numplots/2)
    ncols = 2

    # Initialize the progress bar
    progress_bar = tqdm(total=len(dAra))
```

```
for i in range(len(dAra)):
    time[i] = time[i] / 3600
    dAra[i] = dAra[i] / stepsize
    dCal[i] = dCal[i] / stepsize

    # Uncomment the following if you want to have a progress bar, will␣
↪impact performance!
    # Update the progress bar
    progress_bar.set_description(f"Processing data point: {i + 1}/
↪{len(dAra)}")
    progress_bar.update(1)

# Close the progress bar
progress_bar.close()

count = 0
while count < numplots:

    name = f"fig{count+1}"
    buffer = f"buffer{count+1}"
    image = f"image{count+1}"
    data = datas[count]
    temp1 = globals()[f"{data[0]}"]
    temp2 = globals()[f"{data[1]}"]

    globals()[name] = make_plots(time, temp1, temp2, Legend[count][0],␣
↪Legend[count][1], "Time (" + t_unit + ")", Labels[count], Titles[count],␣
↪semi=semi[count], compare=compare[count],
                        Limits = Limits, ylU=ylims[1], ylL=ylims[0],␣
↪xlU=xlims[1], xlL=xlims[0])

    # Save figures as images in memory
    globals()[buffer] = io.BytesIO()
    globals()[name].savefig(globals()[buffer], format="png")

    # Rewind the buffer and read the image data
    globals()[buffer].seek(0)
    globals()[image] = plt.imread(globals()[buffer])
    plt.close(globals()[name])

    count += 1

# Create a 3x2 grid to display the plots side by side
fig, axes = plt.subplots(nrows = nrows, ncols = ncols, figsize=(15 ,5 *␣
↪nrows))
fig.suptitle("Plotted Data", fontsize=16)
```

```
    # Create a 3x2 grid to display the plots side by side
    count = 0
    while count < nrows:
        count2 = 0
        while count2 < ncols:  # Changed the condition from 'count' to 'count2'
            name = f"image{count * ncols + count2 + 1}"
            axes[count, count2].imshow(globals()[name])
            axes[count, count2].axis("off")
            axes[count, count2].set_aspect('auto')  # Set the aspect ratio to␣
↪auto to fill the available space

            count2 += 1  # Increment the inner loop counter 'count2'

        count += 1  # Increment the outer loop counter 'count'

    # Adjust spacing
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])

    # Show the grid plot
    plt.show()
```

The plotting functions now take in lists of which contain the Titles and labels for the axis and legends of each plot. The number of plots one wants to make is also stored in a variable called numplots.

The output is also scaled to the number of plots automatically so one need not fiddle with the aspect ratio of the output. The plots are created saved and named dynamically using the globals() function. We also define the names of the variables in a list called dictionary so that, when using globals, we can specify what we want to plot in the plots. These parameters are defined in the main function which is as follows,

[11]:
```
def main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
↪alkalinity,
        Eqconstants1, Eqconstants2, Orders, chngTime,␣
↪Surface_Areas,Molar_masses,
        Time, Steps, LUnit, Tolerance, xLimits, yLimits, Limits, Titles,␣
↪Labels, Legend, numplots, semi, compare, datas, dictionary):

    start_time = tim.time()
    #Define the units of time
    t_unit = "hours"

    """
    Main function to execute the PHREEQC simulation and plot the results.
    """
    plt.style.use('default')
    ### Make and mix Solutions
    Solution_1 = Generate_Solutions(Names[0], temp = temps[0], pH = pHs[0],␣
 ↪units = units[0], alkalinity = alkalinity[0], mass = Wmasses[0])
    Solution_2 = Generate_Solutions(Names[1], temp = temps[1], pH = pHs[1],␣
 ↪units = units[1], alkalinity = alkalinity[0], mass = Wmasses[1])
    Solution_3 = Mixer(Names[0], Ratio_1 = 0.5)
    Eq1 = Equilibrate(Names[0], Minerals[0], Addsolid = Smasses[0], CO2 =␣
 ↪CO2s[0], CO2eq = CO2eq[0])
    Eq2 = Equilibrate(Names[0], Minerals[1], Addsolid = Smasses[1], CO2 =␣
 ↪CO2s[1], CO2eq = CO2eq[1])

    ### Set up the kinetics
    Rates = Set_Rates(Names[0], Amount = len(Minerals), Minerals = Minerals,␣
 ↪Surface_Area = Surface_Areas, Molar_mass = Molar_masses)
    Kinetics = Set_Kinetics(Name = Names[0], Solution = "3", Amount = 2,␣
 ↪Minerals = Minerals, masses = Kmasses,
                            Orders = Orders, EqConstants1 = Eqconstants1,␣
 ↪EqConstants2 = Eqconstants2, ChngTime = chngTime,
                            Tolerance = Tolerance, Time = Time, Steps = Steps,␣
 ↪unit = LUnit)

    input_string = Generate_Input(Solution_1 = Solution_1, Solution_2 =␣
 ↪Solution_2, Solution_3 = Solution_3, Eq1 = Eq1, Eq2 = Eq2, Rates = Rates,␣
 ↪Kinetics = Kinetics)
    phreeqc = run_phreeqc(input_string)
    time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal = extract_data(phreeqc,␣
 ↪junkrow = 1)

    plot_data(time, SI_ara, SI_cal, Ara, dAra, Cal, dCal, pH, t_unit, xLimits,␣
 ↪yLimits, Limits, Titles, Labels, Legend, numplots, semi, compare, datas,
              dictionary, TTime = float(Time), TSteps = float(Steps))

    end_time = tim.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
    return time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string

if __name__ == '__main__':

    ###Solution information
    Minerals = ["Aragonite", "Calcite"]
    pHs = ["6.5","6.5"]
    temps = ["18", "18"]
    Names = ["1", "2", "3"]
    units = ["mmol/L", "mmol/L"]
    Wmasses = ["0.01", "0.01"] #Water masses
    Smasses = ["0.0004995", "0.0004995"] #Masses of Solids for equlibrate␣
↪function
    CO2s = [["CO2(g)", "-3.5", "10"], ["CO2(g)", "-3.5", "10"]]
    CO2eq = [True, True]
    alkalinity = ["5", "5"]


    ###Kinetics information
    Eqconstants1 = ["-8.34","-8.48"]
    Eqconstants2 = ["-10.19","-10.33"]
    Orders = ["3", "5"]
    chngTime = "5040" #Time when the Eq. Constants change
    Surface_Areas = [100000, 100000]
    Molar_masses = [100.09, 100.09]
    Kmasses = ["0.0004995", "0.0004995"]

    ###Loop information
    Time = "150"
    Steps = "10000"
    LUnit = "hour" #Units of time
    Tolerance = "1e-08"

    ###Plotting information
    xLimits = [0, float(Time) - 140]
    yLimits = [0, 0]
    Limits = [[[False, False], [False, False], [False, False], [False, False],␣
↪[False, False], [False, False]]
    numplots = 6
    Titles = ["Aragonite and Clacite Over Time", "Aragonite and Clacite Over␣
↪Time", "Rate of Change Over Time", "Relative Levels Over Time", "Saturations␣
↪Over Time", "pH of Solution" ]
    Labels = ["Amount of Solid (mol)","Amount of Solid (mol)", "Rate of␣
↪Dissolution (mol/h)", "Mole Fraction","Saturation Index", "pH"]
    Legend = [["Aragonite", "Calcite"], ["Aragonite", "Calcite"], ["Aragonite",␣
↪"Calcite"], ["Aragonite", "Calcite"], ["Aragonite", "Calcite"], ["pH", ""]]
    semi = [True, False, False, False, False, False]
    compare = [True, True, True, True, True, False]
    datas = [["Ara", "Cal"], ["Ara", "Cal"], ["dAra", "dCal"], ["_ara", "_cal"],␣
↪["SI_ara", "SI_cal"], ["pH", "pH"]]

    #Dictionary
    dictionary = ["Ara", "Cal", "dAra", "dCal", "_ara", "_cal", "SI_ara",␣
↪"SI_cal", "pH"]
```

```
   (time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string) =␣
↪main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
↪alkalinity,
   Eqconstants1, Eqconstants2, Orders, chngTime, Surface_Areas,
   Molar_masses, Time, Steps, LUnit, Tolerance,
   xLimits, yLimits, Limits, Titles, Labels, Legend, numplots, semi, compare,␣
↪datas,
   dictionary)
```

There is now a section called plotting information which defines the following,

- numplots – Defines the number of plots one wishes to make
- xLimits – Defines the upper and lower bounds of the x limits
- yLimits – Defines the upper and lower bounds of the y Limits
- Limits – Is a list of lists containing specifying if one sets the x or y limits.
- Titles – List of the titles of each plot
- Labels – List of the x and y axis labels
- Legend – Contains the names for the legend
- semi – A list specifying weather a plot has a log y axis
- compare – Specifies if we plot one set of data or two (Optional defaults to True)
- datas – list of the data sets for each plot
- dictionary – Contains the names of all the variables for which data is stored, needed for globals function.

  Not This may or may not be easier to use or helpful thus, the code with these changes is saved separately in GitHub.

5.6 Added Solution Customization

I added the ability to change the solution makeup, i.e. add different elements to the solution. I also added some preset solutions, Salt and Pure water which are defined in the PHREEQC guide. There are now new parameters called, preset and component1 and component2, for the first and second solution. The following is the modified code sections,

[11]:
```
def Generate_Solutions(Name, temp = "20", pH = "7", units = "mmol/L", density =␣
↪"1",alkalinity = 0, mass = "1", Components = [] ,preset = "Null"):
    """
    This function generates a solution which will be saved and returned to main.
    Name == Name of soltuion
    Temp == temperature of solution
    pH == pH of solution
    units == Units of the concentration
    Density == Density of solvent
    mass= = mass of solvent in kg
    """

    if preset.lower() == 'salt':
        Solution = (
            "SOLUTION " + Name + "\n"
            + "    temp        "+ temp + "\n"
            + "    pH          8.22 \n"
            + "    density     1.023 \n"
            + "    units       ppm \n"
            + "    Ca          412.3 \n"
            + "    Mg          1291.8 \n"
            + "    K           399.1 \n"
            + "    Si          4.28 \n"
            + "    Cl          19353.0 \n"
            + "    Alkalitnity 141.682 as HCO3 \n"
            + "    S(6)        2712.0 \n"
            + "    -water      " + mass
            + "\n\n" )

    elif preset.lower() == 'pure':
        Solution = (
            "SOLUTION " + Name + "\n"
            + "    temp       "+ temp + "\n"
            + "    pH         " + "7.0" + "\n"
            + "    -water     " + mass
            + "\n\n" )

    else:
        Solution = (
            "SOLUTION " + Name + "\n"
            + "    temp       "+ temp + "\n"
            + "    pH         " + pH + "\n"
            + "    units      "+ units + "\n"
            + "    density    " + density+ "\n"
            + "    Alkalinity " + alkalinity + "\n")

        for i in range(len(Components)):
            Solution = (Solution
                        + "    " + Components[i][0] + "         " +␣
↪Components[i][1] + "\n")

        Solution =  (Solution + "    -water     " + mass
                    + "\n\n" )
    return Solution
```

The Generate_Solutions function now accepts a preset and a components variable. If preset is set too one of the two defined solutions, salt and pure water, the output is those defined solutions. If not now the function works through the components list, adding the components through use of a loop. These variables are defined in the following,

[11]:
```
if __name__ == '__main__':

    ###Solution information
    Minerals = ["Aragonite", "Calcite"]
    pHs = ["6.5","6.5"]
    temps = ["18", "18"]
    Names = ["1", "2", "3"]
    units = ["mmol/L", "mmol/L"]
    Wmasses = ["0.01", "0.01"] #Water masses
    Smasses = ["0.0004995", "0.0004995"] #Masses of Solids for equlibrate␣
 ↪function
    CO2s = [["CO2(g)", "-3.5", "10"], ["CO2(g)", "-3.5", "10"]]
    CO2eq = [True, True]
    alkalinity = ["5", "5"]
    Components1 = [] #List of minerals followed by their concentration
    Components2 = []   # Eg.  [F, 10] is flouride with a concentration of 10
    presets = ["", ""]

    ###Kinetics information
    Eqconstants1 = ["-8.34","-8.48"]
    Eqconstants2 = ["-10.19","-10.33"]
    Orders = ["3", "5"]
    chngTime = "5040" #Time when the Eq. Constants change
    Surface_Areas = [100000, 100000]
    Molar_masses = [100.09, 100.09]
    Kmasses = ["0.0004995", "0.0004995"]

    ###Loop information
    Time = "150"
    Steps = "10000"
    LUnit = "hour" #Units of time
    Tolerance = "1e-08"

    ###Plotting information
    xLimits = [0, float(Time) - 140]
    yLimits = [0, 0]
```

```
   Limits = [[False, False],[False, False],[False, False],[False,␣
↪False],[False, False],[False, False]]
   numplots = 6
   Titles = ["Aragonite and Clacite Over Time", "Rate of Change Over Time",␣
↪"Composition of Solid Mixture", "Percentage Converted", "Saturations Over␣
↪Time", "pH of Solution"]
   Labels = ["Amount of Solid (mol)", "Rate of Dissolution (mol/h)", "Mole␣
↪Fraction", "Mole Fraction", "Saturation Index", "pH"]
   Legend = [["Aragonite", "Calcite"], ["Aragonite", "Calcite"], ["Aragonite",␣
↪"Calcite"], ["Aragonite", "Calcite"], ["Aragonite", "Calcite"], ["pH", ""]]
   semi = [False, False, False, False, False, False]
   compare = [True, True, True, True, True, False]
   datas = [["Ara", "Cal"], ["dAra", "dCal"], ["_ara1", "_cal1"], ["_ara2",␣
↪"_cal2"], ["SI_ara", "SI_cal"], ["pH", "pH"]]

   #Dictionary
   dictionary = ["Ara", "Cal", "dAra", "dCal", "_ara1", "_cal1", "_ara2",␣
↪"_cal2", "SI_ara", "SI_cal", "pH"]

   (time, pH, SI_ara, SI_cal, Ara, dAra, Cal, dCal, input_string) =␣
↪main(Minerals, pHs, temps, Names, units, Wmasses, Smasses, CO2s, CO2eq,␣
↪alkalinity,
                                                                        ␣
↪Eqconstants1, Eqconstants2, Orders, chngTime, Surface_Areas, Components1,␣
↪Components2, presets,
                                                                        ␣
↪Molar_masses, Time, Steps, LUnit, Tolerance,
                                                                        ␣
↪xLimits, yLimits, Limits, Titles, Labels, Legend, numplots, semi, compare,␣
↪datas,
                                                                        ␣
↪dictionary)
```

The new variables are defined under solution information. The Presets variable is a list where one defines the desired preset for both solutions, or leaves as an empty string if one wants to define their own solution. The presets overwrite the components variables. The components variables are a list of lists, containing the desired components of solution 1 and 2. Each component of the list is to be in the following format,

[Element or molecule, concentration]

So if we wanted to add 10mmol/L of NaCl we would have component1 = [["Na", "10"], ["Cl", "10"]].