COMPUTATIONAL MODELS OF THE EVOLUTION OF
PURE OXYGEN ICE GRAINS

by

Tyler Whitaker

A senior thesis submitted to the faculty of
Brigham Young University - Idaho
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics
Brigham Young University - Idaho
April 2018
Copyright © 2018 Tyler Whitaker

All Rights Reserved
DEPARTMENT APPROVAL

of a senior thesis submitted by

Tyler Whitaker

This thesis has been reviewed by the research advisor, research coordinator, and department chair and has been found to be satisfactory.

Lance Nelson, Advisor

Date

Jon Paul Johnson, Committee Member

Date

Brian Tonks, Committee Member

Date

Stephen McNeil, Department Chair

Date

R. Todd Lines, Senior Thesis Coordinator

Date
ABSTRACT

COMPUTATIONAL MODELS OF THE EVOLUTION OF

PURE OXYGEN ICE GRAINS

Tyler Whitaker

Department of Physics

Bachelor of Science

Near solar ice grains are a potential source of the complex organic molecules that could have started life on Earth. I developed two computational models of UV photolysis of pure oxygen ice grains to test both the feasibility of modeling this system and to see if the results of this method are reproduced in laboratory tests. The first model tracks the number of each type of molecule, allowing the reactions to happen using random numbers and probability rates. The second tracks the positions of the particles. The two codes showed slightly different compositions results, and had significantly different runtimes. Composition seems to match laboratory results, although better reactions rates should be used. This success implies that accurate computational models are feasible, though better algorithms should be developed to improve runtime.
ACKNOWLEDGMENTS

I would like to thank first and foremost the physics department of the University of Alabama at Birmingham, who were willing to provide the resources for me to come to their campus, use their facilities, and offer the experience needed to write this thesis. Specifically I would like to thank Dr. Robert Mohr, who mentored me through the Research Experience for Undergraduates (REU) program, and Dr. Yogesh Vohra, who organized everything for the REU. I express a deep gratitude to the National Aeronautics and Space Administration (NASA) and the Alabama Space Grant Consortium for providing the funding for this research. In addition, I would like to thank Dr. Lance Nelson, Dr. Todd Lines, Dr. Jon Paul Johnson, and Dr. Brian Tonks for all of their input and revisions during the writing of this thesis. Last and most importantly, I would like to thank my patient and supportive wife Camila, who was willing to suffer ten weeks of being alone while I worked on this research half way across the country.
Contents
List of Figures
Chapter 1

Introduction

One of the main purposes of research in astronomy is to better understand our place in the universe and answer some of the questions about how we ended up here. According to current scientific theories, the solar system wasn’t built in a day and the same holds true for all life on Earth. How life started on our planet is an area of active research in the scientific community. This thesis explains a computational study of part of one theory explaining how life began on our planet through ice grains under ultraviolet photolysis.

1.1 Early Universe

The most widely accepted theory on how the universe started is called the Big Bang theory. This theory suggests that all matter, energy, and space started off as a single point. In fact, space and time had no meaning before the Big Bang. Suddenly, everything exploded outward, expanding rapidly, creating what we now understand as the universe [? ].

After millions of years of expansion and cooling, galaxies formed as extremely
large stars were created out of collapsing hydrogen clouds. These early stars created heavier elements by forcing the nuclei of these hydrogen atoms together to create larger elements, a process called nuclear fusion \textsuperscript{[? ]}. These early super-massive stars could fuse elements up to iron in their cores before they started to die. Once a star reaches this point, it can no longer produce the energy necessary to resist the gravitational pressure being exerted by its own mass. At this point, the star explodes in a brilliant supernova, sending all of the materials it created during its life out into space, forming the beautiful nebulae found in pictures taken by astronomers. Figure ?? shows one of these nebulae. These explosions are so violent that many different types of elements are formed that are heavier than iron. The creation of all of these elements was necessary for the Earth to be built.

1.2 Accretion Theory

The Pockets of gases and dust inside the nebula created by supernovae will slowly collapse under its own gravitational attraction. New stars and planets can begin to form out of these remains. In the center of one of these collapsing clouds, the gas became dense and hot enough to begin fusion and become a our sun. This star lives a similar life cycle to the stars described above although with significantly less mass. Around our Sun, a disk of gas, dust, rocks, and metals formed from the remaining collapsing matter \textsuperscript{[? ]}. This disk, called the accretion or protoplanetary disk, contained all the material that eventually became our planets. Over the course of several millions years, the all of this matter condensed into the rocky and gaseous planets that form our solar system.

During this process, ice grains could have formed throughout the solar system which tends to be around 4 K. These ice grains could be composed of water, methane,
Figure 1.1 The Crab Nebula shown here is the remnant of a supernova explosion. The composite image shows the intensities of nearly all wavelengths of electromagnetic radiation. The matter ejected from this explosion will likely condense into many planetary systems after the gases have cooled down. Image courtesy of NASA.
alcohols, and diatomic gases, such as oxygen, and nitrogen \[?\]. These grains are of particular interest as they tend to have the building blocks of life so we understand it. These ices grains are being bombarded by radiation from the Sun. When the right frequencies of radiation interact with the ice grain, chemical reactions begin to take place. This light-driven reaction typically occurs in the ultraviolet (UV) range, and is referred to as photolysis. Through this process, the chemical composition of the ice grain will change over time. Potentially, UV photolysis could create the complex organic molecules that would be necessary to kick-start life on a planet such as our own.

1.3 Project & Goals

This project attempts to shed some light on one of the possible ways life could have begun on an earth-like planet. This is done by creating computational model of and simulating photolysis in an ice grain to see how the composition changes over time. My code focuses on pure oxygen (O$_2$) ice grains as a starting point before including other types of molecules and compounds. The reason for this approach is twofold: 1.) This is a first generation code, being built from the ground up. So, for simplicity, only one element is included. 2.) Gerakines, Schuttes, and Ehrenfreund \[?\] performed an experimental version of what I am doing computationally. One of the samples they used was pure O$_2$, and so using this as a starting point, I can compare my results to theirs to better analyze the accuracy of my code.

In computational modeling, the time required to run the simulation is nearly as important as the accuracy of the model. For this reason, two different models were created to explore the trade-off between accuracy and speed. The first model does not keep track of the positions of the molecules in the ice grain, while the second does.
Running both, we can obtain a better idea if the added realistic nature of position tracking improves accuracy enough to be worth the added computation time.
Chapter 2

Procedure

This chapter discusses the assumptions made, and the code used. Only a brief overview of the code is given here, and the reader is directed to the appendix to see the code for this project.

2.1 Ice Grains

As mentioned in the introduction, this project explored the chemical changes of ice grains under UV photolysis in the early solar system. Before continuing much further, some assumptions should be addressed. One of the biggest assumptions is that the grains are composed of only oxygen atoms. With this assertion in place, we can restrict the types of reactions that will occur, and the types of molecules that can be formed. This simplifies the code significantly as only 3 types of compounds can exist, specifically O, O$_2$, and O$_3$. Restricting the types of compounds in this way requires less debugging and can be expanded to more complicated models in later versions of the code.

Figure ?? contains a representation of what the ice grain would look like. In it, two
Figure 2.1 A representation of an ice grain used in this model. The only compounds in this model are those composed entirely of oxygen. The figure represents two of the possible reactions allowed in this model through UV Photolysis. The incoming photon interactions with a molecule or atom and will react with other molecules combining to make larger one, or will break apart forming two molecules. Note that the code will only process one reaction at a time.
of the possible photolysis reactions are depicted. However, only one reaction will occur in the computational model at any given time. We will assume the grains are spherical in shape, have a radius of 5 microns, and are optically thin. The optical thinness of the grain allows for all molecules to have an equally likely chance of absorbing the incoming photon. The Sun is assumed to produce a blackbody spectrum, and therefore the radiation interacting with the grain includes wide range of frequencies. Many of the photons will pass through because the grain is optically thin, or because they do not contain the right energy to trigger the reactions. These photons will not be counted in this model. The allowed reactions are as follows:

\[ 2O \rightarrow O_2 \]
\[ O + O_2 \rightarrow O_3 \]
\[ O_2 \rightarrow 2O \]
\[ O_3 \rightarrow O + O_2 \] (2.1)

Note that these reactions will likely not share the same activation energy, which is one of the reasons we assume a broad spectrum of light.

Some more uncategorized assumptions include the following. The region of space between the sun and ice grain is not filled with dust, thus allowing the light to reach the ice grain. The grain is in the habitable zone of the sun. The average temperature is 4 K. The total atom count is not changed through proton deposition.

2.2 Code

The code was developed over the course of a 10 week internship through a NASA funded Research Experience for Undergraduate (REU) program at the University of Alabama at Birmingham. The two different models shared a lot of similarities. First and foremost, they both simulated the changing composition through UV photoly-
sis reactions. Both are event-based simulations, modeling a single chemical reaction triggered by a single absorbed photon. Both begin by creating an ice gain containing 100,000 O$_2$ molecules. A pseudorandom number generator was used heavily in selecting what types of reaction would occur. During each of these simulations, a log was recorded after each reaction. The log contains information about runtime, reaction that occurred, and most importantly, particle counts of O, O$_2$, and O$_3$. These logs were used to plot particle counts as a function of time, and debugging. An brief overview and flow chart of both models is included in Figure ??.

2.2.1 First Model: No Position Tracking

The first model was a very rudimentary model of the ice grains. The most important feature of this code is that it did not track the position of the individual molecule in the ice grain. The grain was initialized by assigning an initial composition. All of my simulations started with 100,000 O$_2$. However, using this same code, one could initialize the grain with any combination of the allowed molecules. After the ice grain was created, the model simulates photons striking the grain by selecting at random a type of molecule. The probabilities of hitting any molecule type was calculated by finding the number ratio of one molecule type to the total number of molecules. For example, if there were twice as many O$_2$ as O$_3$, then O$_2$ was twice as likely to be hit by the incoming photon.

Once a molecule type was selected, the type of reaction had to be chosen. The reaction equations listed above (eq. ??) show the allowed changes for this system. In addition to these reactions, the model can allow the molecule to absorb a photon but not react with any other molecule (referred to here as a static reaction). Such an event simply raises the overall thermal energy of the ice grain. Each of the allowed reactions (depending on the molecule selected) were assigned arbitrary probabilities.
2.2 Code

Figure 2.2 A flow chart describing an extremely generalized summary of
the process of both models created.
of occurring. Once the reaction was selected (again, through random numbers), the number of each molecule type was changed accordingly. This process repeats over for a total of 600,000 photons absorbed by the ice grain.

This model has a very quick runtime, and can run many iterations of this code within minutes. This allows us to test various sets of initial conditions and probability settings. This helpful feature allows us to test the sensitivity to such settings. I include 6 different probability settings in Chapter ??.

2.2.2 Second Model: Position Tracking

The second model took a different approach by assigning position vectors, and a number of oxygen atoms to each molecule. These molecules were stored in a list which could grow and shrink as needed. The initial composition is assigned by the user. The ice grain is generated by randomly placing molecules within a 5 micron radius sphere.

It is at this point that the algorithm differs significantly from the first model. Once initialized, molecules are selected at random. The code first determines what type of molecule has been hit by the photon to know what reactions are allowed. If the molecule hit is a monatomic oxygen, it can only combine with other molecules, and the code to break apart molecules is ignored. A similar logic follows for O$_3$ where it cannot be used to create larger molecules. Diatomic oxygen, however, is allowed to undergo either reaction, so we must allow the model access to those reactions.

If the molecule splits, then a designated function lowers the oxygen count of the selected molecule, and generates a new monatomic oxygen atom. The first original molecule is moved a slight amount in a random direction and the new O atom will
be placed an equal amount in the opposite direction. This is expressed as,

\[ \vec{r}_{af} = \vec{r}_{ai} + \Delta \vec{r} \]
\[ \vec{r}_{\beta} = \vec{r}_{ai} - \Delta \vec{r} \]

(2.2)

where \( \vec{r}_{ai} \) and \( \vec{r}_{af} \) are respectively the initial and final position of the original molecule, \( \vec{r}_{\beta} \) is the position of new O that was created, and \( \Delta \vec{r} \) is the small change in position of the molecule \( \alpha \). As these molecules are being created and modified, a running counter is kept to minimize computation time, and allow for easier logging and plotting.

Combination reactions are more complicated. The first model only needed to know if there are enough molecules for the reaction. The second model also requires they be any close enough for a reaction. In the real world, a diatomic oxygen will never combine with a monatomic oxygen that is on the opposite end of the ice grain and neither do they in this model.

In order to ensure that this happens, the program searches through all of the molecules for ones that are within a specific distance. This is set from the beginning of the simulation. After finding all of the molecules that 1.) are within the appropriate distance and 2.) are molecules that are allowed to react with the selected molecule, one of these potential molecules is picked at random. If none meet the criteria above, then the program treats this as a static event, where no chemical reactions occur. Otherwise, the two molecules are combined. The first molecule is destroyed by removing it from the list, and the oxygen atom count of the first is added to the second. These changes are also represented in the number counts similar to described above.
Chapter 3

Results

This chapter presents the results found in this project. It discusses the different final compositions achieved through varying the reaction rates of the first version of the code, and how they compare to the final composition of the second version. We also discuss the run-time of each model to compare the usefulness of each.

3.1 Composition of Ice Grain

As stated in the previous chapter, all runs of the simulation were initialized with an ice grain containing 100,000 O$_2$ molecules. All variations of the first model reached an equilibrium point where the number of molecules no longer changes significantly. This tends to occurs well before 600,000 photons as can be seen in the graphs included in Figure ???. The second model did not quite reach a stable equilibrium, as seen from Figure ???. However, it is clear that the system is approaching one.

In these stable equilibria, every iteration of both models indicate that at least two particle types will have the same number of particles. However, the exact molecules that share this number ratio differs depending on the initial settings. More re-
Chapter 3 Results

(a) Every reaction has even probabilities of occurring.

(b) The probability of a static reaction was set to zero.

(c) The probability of breaking was assigned to 75%, and combining was $O_2$, but still allows for nothing to happen.

(d) The chances are higher to break assigned to 75%, but still allows for nothing to happen.

(e) The ratio is exactly reversed when compared to figure ??.

(f) The 75% chance of combining $O_2$ is shifted to 90%.

Figure 3.1 Plots of the composition of the grain vs. the number of photons that hit the grain. These graphs were created using the first model, changing the probability rates to see what would happen. These scenarios and probabilities were chosen arbitrarily, and the reader should remember that a more accurate simulation would use experimentally discovered numbers.
3.1 Composition of Ice Grain

Figure 3.2 Plot of the composition of the grain vs. the number of photons that hit the grain using the second model.
search would have to be done to understand the significance of this result. With the molecules that differ, the average number ratio in the more evenly distributed probabilities ranged from 1:1 to 2:1. The most extreme ratio was that of the 90% chance of combining \( \text{O}_2 \) (Figure ??), which had a ratio of \( \text{O}_2 \) and \( \text{O}_3 \) to \( \text{O}_1 \) of about 4:1.

3.2 Computation

The two models varied significantly in run-time, as expected. The first model took an average of 3 seconds to run, and the second model took an average of 2 days. For this reason, I kept the probability rates the same in order to compare computation times as I adjusted the algorithm. The first version of the second model ran for 2 days and 9 hours, and after several modifications to the algorithm, I was able to decrease the time to 1 day and 21 hours. It should be noted that more optimizations could be made to improve the computational speed. In addition, the code ran a personal computer, using python. The model will inevitably run slower because it is built using an interpreted language instead of a compiled language.
Chapter 4

Analysis

This chapter discusses the implications of the results found. We discuss the realistic nature of the compositions, what things are affecting the computation time, and how the probabilities choices affect the outcome of the ice grain and other potential issues of the model.

4.1 Composition

The first question that one should ask about this model is whether the outcome is realistic. Gerakins, Schutte and Ehrenfund \cite{Gerakins2022} created sheets of ice in a vacuum chamber at 10 K. This ice was then irradiated with ultraviolet (UV) radiation to trigger the photolysis reactions described in this work. Only their O\textsubscript{2} sample is of interest here.

They report that 36\% of their O\textsubscript{2} sample had been changed into O\textsubscript{3} after one hour of irradiation. This seems to be consistent with the simulations I ran. It must be noted that the outcomes are dependent on the probabilities assigned at the beginning of the simulation. For example, the simulations where a heavier emphasis is
placed on combining O\textsubscript{2} into O\textsubscript{3} will produce outcomes that match those found in the laboratory. Meanwhile, combining reaction probability of 90% seems unrealistic. In order to adequately compare the results of the two studies, a better understanding of spectroscopy and the spectrograph of oxygen would be beneficial as Gerakins’ data was reported through spectrograph.

### 4.2 Computation Time

The first model had minimal issues with computation time. Most of the time used was in the main loop with 600,000 iterations. All of the rest of the code is simple manipulation of some count variables. It would be hard to optimize this much further than it already is. One possible way to decrease the computation time of the first version would be have two photons strike the grain at the same time. This could be done with certain methods of parallelization, allowing two different threads to run the same loop of code on different cores of a processor. Thus, the 600,000 iterations of the main loop would be reduced to 300,000 iterations per core.

As mentioned earlier, taking on average 3 seconds is an extremely convenient way of running many different conditions. One could automate the code to run multiple iterations, changing the reaction probabilities with each one. Something like ranging from the combination of O\textsubscript{2} from 10% to 90% and incrementing in steps of 10, the whole code could be run in under a minute.

On the other hand, the second model needs more optimization to make it more useful. Most of the computation time is spent looking at each molecule in the ice grain. In order to know if a molecule will have the ability to combine with something else, it needs to look through the entire grain to find the neighboring molecules that could work for a combination reaction (refer back to equations ??). This becomes
4.3 Potential issues

problematic as the simulation starts off with 100,000 molecules, and grows to around 122,000 molecules. These nested loops use much of the time but there are a few potential ways to speed this up. One way is through parallelization, where the list of molecules could be broken in pieces, allowing different threads to search through the list at the same time. Other methods could also be used to organize the molecules differently, so that they would be easier to find. One such method could be to use an array and only let the molecules being hit to interact with the other molecules in the same box. There are a wide variety of things that could be done in an effort to better the computational time of the project.

4.3 Potential issues

No computational model is going to represent exactly what happens. Both models have a few things that should be addressed in order to make sure that it produces a realistic model.

The first thing that should be fixed is to find proper reaction rates. This would probably best be done through further studies of photolysis reactions in laboratory environments similar to those used by Gerakins et. al. [?]. If a set of reliable reaction rates were know, they could be used in this model to produce better results. Another detail that could use more research is the reaction distance. For this project, an arbitrary number was selected as a maximum distance between two molecules to have a successful reaction. A better option would be to actually include the a known distance between the atoms in the crystal, or some other empirically discovered length. One final issue is that neither model creates a crystal structure. In the second model, molecules were placed at random in the ice grain and were only restricted by the spherical shape of the grain. The first model does not have the capability of creating
a crystalline structure. A better model might place the atoms and molecules in a crystal structure, if one is known to exist.
Chapter 5

Future Work

This project is far from being an extensive study on the subject. If more time had been alloted to this project, there are some things that could have been improved on, expanded, and made more elaborate. The following are some suggestions of what could be improved.

One of the biggest issues is the elapse time of the position tracking model. The long run-time makes for quick changes to the code almost unreasonable. Running multiple variations of the parameters would take weeks to months, not counting any debugging needed. One way of mitigating this is to recode this model in a compiled language, such as C++. Creating a new data type for the molecules with a position variable and oxygen count would be a very elegant way to handle the information. A type of array in C++ called a Vector would have much of the same functionality as that of a python list, allowing the number of molecules to grow and shrink as the reactions took place. In addition, any of the changes suggested in Section ?? would lead to plenty of work in improving this model.

In a realistic situation, no ice grain is going to be composed of only one type of atom or molecule. There are going to be a wide variety of atoms and composition
One of the biggest additions to either of the codes discussed here would be to include other elements. Carbon, hydrogen, and nitrogen would likely be a good place to start, as these are the base elements for the complex organic molecules discussed in Chapter ???. The first model would only need to add functions similar to the ones that currently exist to allow for the added reaction. Each molecule would have a designated function that determines certain reactions, and once that reaction has been picked, adjust the numbers accordingly. A good starting place here would be the work of F. Stuhl and H. Niki [? ]. The second version of the code would be much more difficult to implement however. The types of reactions would be determined by the types of molecules that are close enough. However, one would need to insure that the type of reactions are allowed, not only that the oxygen counts match up nicely.

One final addition to the project as a whole would be to calculate the time scale over which this process would occur. I attempted this using Plank’s equation for black body curves. I used a distance of of the habitable zone calculated by James F. Kasting and Daniel P. Whitmire and Ray T. Reynolds [? ]. However, my calculations suggested that around $1.8 \times 10^9$ to $6.3 \times 10^9$ UV photons travel through a $1 \mu m \times 1 \mu m$ square in one second. This suggests that the entire simulation would happen within one second. However, going back to our assumptions in Chapter ??, we assumed that there was no dust in between the Sun and our ice grain. This calculation does not account for any possible attenuation that would occur in a realistic accretion period of the solar system.

Overall, this project provides a good starting place for someone interested in this topic. With more work, this model has potential to predict some interesting results.
Bibliography


Appendix A

First Model

The following is the code used for the first model (no position tracking) the functions are labeled and commented. Some important things to note are the constructor `Oxygen.__int__(self)` where the user can set the initial composition, and the for loop in `Oxygen.moveTime(self)`, where the probabilities for each reaction are assigned.

```
#!/usr/bin/python3

############################################################
# Author: Tyler Whitaker
#
# Oxygen Photolysis: This program has the assumptions that
# that particles will have even chances of both being hit
# by a photon, as well as equal chances of all possible
# reactions.
#
#*****************************************************************************
```
Chapter A  First Model

import numpy as np
import matplotlib.pyplot as plt
from random import random

class Oxygen():
    def __init__(self):
        self.numO3 = 0  # Number of O3 molecules
        self.numO2 = 100000  # Number of O2 molecules
        self.numO1 = 0  # Number of O1 "molecules"

        # Total sum of all Oxygen atoms. Used as backup to ensure
        # that we aren't losing or gaining random atoms.
        self.numOAtoms = 3*self.numO3 + 2*self.numO2 + 1*self.numO1

        # Track number of each particle with respect to time for plotting
        self.trackO3 = []
        self.trackO2 = []
        self.trackO1 = []

        # are we goin to plot this time?
        self.show = False
        self.name = 'Even Probabilities'

    def changeO3():
        # Decides what type of reaction an O3 molecule will go
        # through and performs the appropriate math in order to
# conserve matter. Also returns a string for the log.

# changeO3(self):

def changeO3(self):
    reactionType = random()
    logType = '' # log the type of reaction that occured.
    # 03 => O2 + O1
    if reactionType <= 1/2:
        self.numO3 -= 1
        self.numO2 += 1
        self.numO1 += 1
        logType = "03 => O2 + O1"
    # (03 => 03)
    else:
        # do nothing.
        logType = "03 => 03"
        pass

    return logType

# changeO2():
# Decides what type of reaction an O2 molecule will go through and performs the appropriate math in order to conserve matter. Also returns a string for the log.

def changeO2(self):
reactionSuccess = False  # ensure that a reaction happens
logType = ''  # log the type of reaction that occurred.

while not reactionSuccess:
    reactionType = random()

    # 02 + 01 => 03
    if reactionType <= 1/3:
        # if not enough 01 atoms exist, dont perform this reaction.
        if not self.num01 >= 1:
            continue  # Try again.
        # otherwise, carry on.
        self.num03 += 1
        self.num02 -= 1
        self.num01 -= 1
        logType = '02 + 01 => 03'

    # 02 => 02
    elif reactionType <= 2/3:
        # do nothing
        logType = "02 => 02"
        pass

    # 02 => 2*01
    else:
        self.num02 -= 1
        self.num01 += 2
logType = "O2 => 2*O1"

# if we reach this point reaction was allowed, so leave the loop.
reactionSuccess = True

return logType

###################################################################
# changeO1():
# Decides what type of reaction an O1 molecule will go through and performs the appropriate math in order to conserve matter. Also returns a string for the log.
###################################################################
def changeO1(self):
    reactionSuccess = False
    logType = '' # log the type of reaction that occured.

    while not reactionSuccess:
        reactionType = random()
        # O1 => O1
        if reactionType <= 1/3:
            # do nothing
            logType = "O1 => O1"
            pass
        # O1 + O2 => O3
elif reactionType <= 2/3:
    # if not enough O2 atoms exist, don't perform this reaction.
    if not self.numO2 >= 1:
        continue  # try again
    # otherwise, carry on.
    self.numO3 += 1
    self.numO2 -= 1
    self.numO1 -= 1
    logType = "O1 + O2 => O3"
    
# 2*O1 => O2
else:
    # if not enough O1 atoms exist, don't perform this reaction.
    if not self.numO1 >= 2:
        continue  # try again
    self.numO2 += 1
    self.numO1 -= 2
    self.numO0 -= 1
    logType = "2*O1 => O2"

    # if we reach this point reaction was allowed, so leave the loop.
    reactionSuccess = True

    return logType

# checkForErrors():
# Function to run through various parts of the code to
# make sure that nothing goes against basic laws of
# physics. Specifically, we should not have a different
# number of oxygen atoms that we start with, nor should
# any of the particle counts be negative.

# moveTime():

# moveTime():

def checkForErrors(self):
    broken = False
    errorText = ''

    if not (self.numOAtoms == 3*self.numO3 + 2*self.numO2 +
            1*self.numO1):
        errorText = "Oxygen atoms not conserved."
        broken = True
    if not (self.numO3 >= 0):
        errorText += "Negative O3 particles!"
        broken = True
    if not (self.numO2 >= 0):
        errorText += "Negative O2 particles!"
        broken = True
    if not (self.numO1 >= 0):
        errorText += "Negative O1 particles!"
        broken = True

    return broken, errorText
This function is the main loop for this class. Decides how many photons to throw at the particles. Calls appropriate functions in order to trigger reactions. Also tracks progress of total of particles.

--------------------------------------------------

def moveTime(self):
    index = 0

    # open log file
    log = open(self.name + '_log.csv','w')

    broken = False # flag variable keeping track that we did not break physics.

    for time in range(int(600000)):
        sumOfParticles = self.numO3+self.numO2+self.numO1

        # Write to log: How any of each particle exists?
        log.write('{:>6}'.format(self.numO1) + ' , ' + '{:>6}'.format(self.numO2) + ' , ' + '{:>6}'.format(self.numO3) + ' , ' + '{:>6}'.format(3*self.numO3 + 2*self.numO2 + 1*self.numO1) + ' , ')

        # find fractional probabilities of hitting a particular particle based on number of particle.
        frac3 = self.numO3/sumOfParticles
        frac2 = self.numO2/sumOfParticles
        frac1 = self.numO1/sumOfParticles
##### Debugging and "Assert" statements section #####

```python
if not (abs(frac3+frac2+frac1 - 1.0) <= .0005): # assert this
to be true

errorText = "Bad Fraction"
broken = True

broken, errorText = self.checkForErrors()

if broken:
    print("Error: " + errorText)
    log.write("Error: " + errorText)
    break # leaves the loop, which will close the file and end
    the program

################################################ END ################################################

# what are we going to hit?

hitType = random()

if hitType <= frac3: # hit O3
    logType = "O3 , " + self.changeO3()

elif hitType <= frac2+frac3: # hit O2
    logType = "O2 , " + self.changeO2()

else: # hit O1
    logType = "O1 , " + self.changeO1()

# write to log: what type of particle was hit, and what reaction
did it undergo?

log.write(logType + '\n')
```
# Add each number of particles to their respective lists
# for plotting.
if index % 1000 == 0:
    self.trackO3.append(self.numO3)
    self.trackO2.append(self.numO2)
    self.trackO1.append(self.numO1)

    # keeping every single particle from being counted in plot.
    # (saves computational time.)

    index += 1

log.close()

############################################################
# plotStuff():
# responsible for plotting the graph of the three different
# types of molecules.
############################################################

def plotAndSave(self):
    plt.plot(self.trackO1,label='O1')
    plt.plot(self.trackO2,label='O2')
    plt.plot(self.trackO3,label='O3')
    plt.title(self.name)
    plt.legend(loc="upper right")
    plt.savefig(self.name+'.pdf')

    if self.show:
        plt.show()
# main():
# Driver Function. Creates system of oxygen particles.
# Calls functions to move time forward and plot the results.

def main():
    test = Oxygen()
    test.moveTime()
    test.plotAndSave()

if __name__ == "__main__":
    main()
The following is the code used in the second model (tracks position). This one includes some global variables that change significant details from the beginning, such as the radius of the ice grain, the reaction distance, and the number of photons to simulate. In addition, the constructor contains the variables to assign the initial composition of the ice grain, and the for loop in `Oxygen.moveTime(self)` contains some critical information for the reaction rates. It should be noted however, that the exact numbers used do not accurately represent the percent chance that a specific reaction will take place, as much of this information depends heavily on the location of other particles available.

One known bug exists in the logging function. I used the function `datetime.datetime.now()` to keep track of the amount of time spent on the simulation. By using it once at the beginning of the code, I was able to keep to print (and log) the amount of time that had elapsed since running the code by finding the difference between the `START` global variable and the value returned by the function. The bug is that it returns the in the format ”Days, Hours:Minutes:seconds.” In saving my log file as a CSV, after the first day, a new column is introduced into the log. Trying to read this back into a
NumPy array for plotting purposes caused an issue because of mismatched column numbers. This bug only causes issues if the code were to be automated, running several iterations and saving plots without human manipulation.

```python
#!/usr/bin/python3

# Author: Tyler Whitaker
#
# Oxygen Photolysis (with position): This code will be used
# as the template for the UV photolysis of oxygen atoms
# compounds in which the position of the particles are
# taken into account.
#

import numpy as np
from numpy.linalg import norm
import matplotlib.pyplot as plt
import random as ra
from datetime import datetime

#### Global variables to quickly edit parameters ####
REACTION_DISTANCE = 0.1
PHOTON_COUNT = 600000
FILE_NAME = 'photons' + str(PHOTON_COUNT)
RADIUS = 5
START = datetime.now()
```
SHOW = False

class Oxygen:
    def __init__(self):
        numO3 = 0     # Number of O3 molecules
        numO2 = 100000 # Number of O2 molecules
        numO1 = 0     # Number of O1 "molecules"
        self.numbers = [numO1,numO2,numO3]

        # Total sum of all Oxygen atoms. Used as backup to ensure
        # that we aren’t losing or gaining random atoms.
        self.numOAtoms = 3*numO3 + 2*numO2 + 1*numO1

        # Track number of each particle with respect to time for plotting
        self.trackO3 = []
        self.trackO2 = []
        self.trackO1 = []

        # are we goin to plot this time?
        self.show = False

############################################################
# initParticles():
# initialize the particles by placing them in raandom
# locations within a sphere. A set of 3D cartesian
# coordinates are created. If that position is outside of
# the radius ice grain (specified above), then we loop
# back to create a new position.

def initParticles(self):
    self.particles = []  # list for particles
    x = 0  # used to keep track of what type of particle we are creating.

    for num in self.numbers:  # self.numbers is the array of initial
        particle counts
        x += 1

        for i in range(num):  # loop for as many particles as needed.
            inRange = False  # flag to make sure that we create particles
                            # within our sphere.
            while not inRange:
                randX = ra.uniform(-5, 5)
                randY = ra.uniform(-5, 5)
                randZ = ra.uniform(-5, 5)
                vec = np.array([randX, randY, randZ])
                if norm(vec) <= RADIUS:  # check to make sure it is
                    within the sphere.
                    inRange = True

            # convert the randomized spherical vector to cartesian
            coordinates
            self.particles.append([vec, x])
densityCheck(self):
    # This is debugging function to check to make sure that
    # the density of the ice grain is reasonable. It does not
    # have to be run. It counts the particles that are around
    # each of the particle, and saves that information to a
    # csv file.

def densityCheck(self):
    length = len(self.particles)
    density = np.zeros(length, int)  # array of how any particles are
    # near each particle in our list.
    density -= 1  # all counts will include the particle being checked.
    particles = np.zeros([length, 3])

    # This converts the list of particles into an array of positions
    # for each particles.
    # This is desired because of the computational speed associated
    # with array math.
    for i in range(length):
        particles[i, 0] = self.particles[i][0][0]
        particles[i, 1] = self.particles[i][0][1]
        particles[i, 2] = self.particles[i][0][2]

    i = 0
    # for each particle in our list,
for alpha in particles:
    if i % 10000 == 0:
        print(i)  # visual tracker of progress.
    # create an array of separation vectors.
    sepVecs = alpha - particles
    # find the magnitude of each of those vectors
    sepMag = norm(sepVecs, axis=1)
    # find which particles are within the reaction range. Then, add
    # up how many are within range.
    density[i] = (sepMag <= REACTION_DISTANCE).sum()
    i += 1

np.savetxt('density.csv', density)

# triggerReaction():
# This method handles the reactions that are taking place
# and creates an equation for the reaction (if that
# information is desired). Random numbers are used to
# decide what type of reactions will occur and with what
# particle.

def triggerReaction(self, alpha, t):
    reactionType = 'O' + str(self.particles[alpha][1]) + ' '  

    if self.particles[alpha][1] == 1:  # O1 will never divide,  
        randNum = .75  # so don't give it the option.
else:
    # otherwise, generate a random number.
    randNum = ra.uniform(0,1)

# This is the section to either combine or do nothing.
if randNum >= 0.5:
    if self.particles[alpha][1] == 3:  # ОЗ will never combine
        potentialParticles = []  # so don’t bother looking
        through the particles.

    # find all the particles for potential reaction with
    particles[alpha]

else:
    # Conditions for potential particles:
    #  - sum of atom count must not exceed 3 (no О4 or higher)
    #  - must be within the specified range (see global
    #    variables above)
    #  - must not be self (alpha != beta)
    potentialParticles = [
        [self.particles[beta][1], beta]
        for beta in range(len(self.particles))
        if self.particles[alpha][1] + self.particles[beta][1] <= 3
        and
        norm(self.particles[beta][0]-self.particles[alpha][0])
        <= REACTION_DISTANCE
        and not all(self.particles[alpha][0] ==
                    self.particles[beta][0])]
# Selection of the particle

```python
if len(potentialParticles) == 0:  # if there are no potential particles
    # no reaction occurs
    reactionType += ' => O' + str(self.particles[alpha][1])
    beta = -5  # tag to make sure no reaction occurs

elif len(potentialParticles) == 1:  # if only one, then it will react with it.
    beta = potentialParticles[0][1]

else:  # if there are multiple, then select a random particle[beta]
    beta =
        potentialParticles[ra.randint(0, len(potentialParticles)-1)][1]

if beta != -5:  # beta = -5 is the code (see above) to mean no reaction occurred
    reactionType += ' + O' + str(self.particles[beta][1])
    # alpha crashes into and combined with beta
    self.numbers[self.particles[beta][1]-1] -= 1
    self.numbers[self.particles[alpha][1]-1] -= 1
    self.particles[beta][1] += self.particles[alpha][1]
    self.numbers[self.particles[beta][1]-1] += 1
    reactionType += ' => O' + str(self.particles[beta][1])
    # remove particle from the list that was hit (by photon) from
```
the list
self.particles.pop(alpha)

# if random number was out of bounds for the combine/do nothing reaction,
else:
    # then break the particle in two.
    reactionType += self.breakParticle(alpha)

    # return the reaction equation
    return reactionType

# breakParticle():
# This function breaks the particle into two pieces. It assumes that the particle will eject an O1. If the original particle alpha was O3, then it will become O2. These particles will be separated by a small distance. Imagine this as if the particle explodes and both pieces are sent in opposite directions.

def breakParticle(self, alpha):
    # random distance to separate the particles.
    dr = np.array([ra.uniform(0,1),ra.uniform(0,1),ra.uniform(0,1)])*.05
    # add new O1 to list of particles a small distance away from where alpha was.
    self.particles.append([self.particles[alpha][0]+dr,1])
    # add a particle to log
self.numbers[0] += 1

# move particle[alpha]
self.particles[alpha][0] -= dr

# subtracting particle from numbers
self.numbers[self.particles[alpha][1]-1] -= 1

# remove 0 from particle[alpha]
self.particles[alpha][1] -= 1

# adding particle to numbers
self.numbers[self.particles[alpha][1]-1] += 1

# return equation
return '=> O' + str(self.particles[alpha][1]) + ' + O' + str(self.particles[-1][1])

############################################################
# movetime():
# This function is responsible for moving time forward.
# Its body is a for loop that cycles through all of
# the photons that will be sent (determined by
# PHOTON_COUNT defined above) to the ice grain. This
# function will call logData() as needed in order to make
# sure that the log file and data are saved for graphing
# and analysis. Inside the loop, particles will be
# selected at random which will be hit by the incoming
# photon, at which triggerReaction().

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

def moveTime(self):
    # open a log file
log = open(FILE_NAME + '_Log.csv', 'w')

# top line is to label columns
log.write('Photon, Oxygens, O1, O2, O3, Particles, Time
')

# start for loop
for t in range(PHOTON_COUNT):
    # Do we need log the counts?
    self.logData(log, t)  # call logging function
    if t%1000 == 0:
        print(t, datetime.now() - START)  # report progress in terminal

    # pick random particle
    randParticle = ra.randint(0, len(self.particles)-1)

    ### This line is used if the reaction equations are desired.
    log.write(' Type: O' + str(self.particles[randParticle][1]) + ', ')

    ### similar situation with the reaction equation.
    reactionType = self.triggerReaction(randParticle, t)
    log.write(reactionType + ' 
')

    # Write to the log file one last time to save final counts.
    self.logData(log, PHOTON_COUNT)

log.close()
# logData():

# This function takes in the log file, finds how many
# particles of each type exist in the list of particles,
# and tracks them to the log file and local memory. This
# information will be used to plot the data later on, and
# analyze in a csv file. Formatting for the log file is
# also handled here.

# logData(self,log,t):
    def logData(self,log,t):
        log.write('{:>6}, '.format(t) + str(1*self.numbers[0] + 2*self.numbers[1] + 3*self.numbers[2]) + '
' + ', {:>6}'.format(self.numbers[0]) + ' , {:>6}'.format(self.numbers[1]) + ' , {:>6}'.format(self.numbers[2]) + '
' + ', {:>8} , '.format(len(self.particles)) + str(datetime.now() - START) + '
')

# plotStuff():

# plotStuff(self):
    def plotStuff(self):
        # uses the lists created in __init__() and modified in
        # logData() to generate a plot of the progression of the
        # particles over time (by photon count). Also saves the
        # graph as a pdf.-
data = np.genfromtxt(FILE_NAME + '_Log.csv', delimiter=',

data = data[1:,2:5]
plt.plot(data[:,0])
plt.plot(data[:,1])
plt.plot(data[:,2])
plt.xlabel("Number of Photons")
plt.ylabel("Number of Particles")
plt.title(FILE_NAME)
plt.legend(loc="upper right")
plt.savefig(FILE_NAME+'.pdf')

if SHOW:
    plt.show()

# main():
#  Driver function. Initializes the system and calls the
#  necessary methods to calculate things for the program.

def main():
    system = Oxygen()
    system.initParticles()
    system.moveTime()
    system.plotStuff()

    if __name__ == '__main__':
        print("starting",FILE_NAME)
        main()