UNIVERSITY OF CALIFORNIA,
IRVINE

Stochastic Models of Photoexcitation & Charge Accumulation for Solar Energy Conversion

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Materials Science and Engineering

by

Kevin Tkaczibson

Dissertation Committee:
Assistant Professor Shane Ardo, Chair
Associate Professor Matt Law
Assistant Professor Allon Hochbaum

2019

DEDICATION

To Emily,

You took a crazy leap of faith following me across the county and I am so unbelievably happy you did. You make every single part of my life more wonderful and none of this would ever have happened without your amazing support. I can't wait to see what the next adventure has in store for us.

Love, Kevin

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thank you to Prof. Shane Ardo who took me on despite my mixed research background and allowed me to have increasing freedom in guiding the direction of my projects over the last few years. Not all your crazy ideas are good ones but of all the crazy PIs I believe you certainly are one of the good ones. You seem to genuinely care for the well being of your students and prioritize them as human beings over optimizing your lab for research output.

Thanks to my current and former labmates: Joseph Cardon, Will White, Jen Glancy, Simon Luo, Eric Schwartz, Rohit Bhide, Leanna Schulte, Rylan Kautz, Cassidy Feltenberger, Zejie Chen, Gabe Phun, Nazila Farhang, Prof. Rohini Bala Chandran, Dr. Larry Renna, Prof. Jingyuan Liu, Dr Houman Yaghoubi, Dr Hsiang-Yun Chen, Dr. Chris Sanborn, Dr. David Fabian, Ron Reiter, Claudia Ramirez, Margherita Taddei, Greg Krueper, Tabitha Miller, Jackie Angsono, and Sam Nitz. This journey was so much better for your being a part of it. In particular, thanks to Joseph Cardon for your infinite patience and insightful discussions.

Thanks also to my friends outside of lab including (but not excluded to): Kelsey Cardon, Kevin Keator, Ed Jenner, Joanne Leadbetter, Sarah Royer, Anna Smith, Will Thrift, Peter Wagner, and Kristin Yamaka. You kept me sane during these last few years and are all around an awesome bunch!

Thanks to my family and most of all to Emily. You've been unbelievably supportive from start to finish and have helped make any of this possible.

This work was supported by the National Science Foundation under CHE – 1566160, the U.S. Department of Energy, Office of Energy Efficiency and Renewable Energy, Fuel Cell Technologies Incubator Program under Award No. DE-EE0006963, the School of Physical Sciences and the Henry Samueli School of Engineering at the University of California Irvine.

CURRICULUM VITAE

Kevin Tkaczibson

2010-14 B.S. in Physics, Carnegie Mellon University

2014-15 M.S. in Materials Science, University of California, Irvine

2015-2019 Ph.D. in Materials Science, University of California, Irvine

FIELD OF STUDY

Self-exchange election transfer modeling between surface anchored molecules

PUBLICATIONS

"Physical and electrochemical area determination of electrodeposited Ni, Co, and NiCo thin films" Gira, M.J., Tkacz, K.P. & Hampton, J.R. Nano Convergence. 2016.

"Investigating Saltwater Desalination by Electrodialysis and Curriculum Extensions To Introduce Students to the Chemical Physics of Polymeric Ion-Exchange Membranes." K. Tkacz, S. T. E. Nitz, W. White, S. Ardo*. Journal of Chemical Education. 2017

"Numerical Monte Carlo Simulations of Charge Transport across the Surface of Dye and Cocatalyst Modified Spherical Nanoparticles under Conditions of Pulsed or Continuous Illumination." K. Tkaczibson and S. Ardo*. Sustainable Energy & Fuels. 2019

ABSTRACT OF THE DISSERTATION

Stochastic Models of Photoexcitation and Charge Accumulation for Solar Energy Conversion

By

Kevin Tkaczibson

Doctor of Philosophy in Materials Science

University of California, Irvine, 2019

Assistant Professor Shane Ardo, Chair

A stochastic model has been created to efficiently explore the parameter space surrounding schemes for a new dye-sensitized solar cell and photoelectrochemical constructs. Models of a variety of semiconductor configurations that are populated with surface-anchored light-absorbing dye molecules and electrocatalysts are simulated under various illumination conditions. Absorption of light by dyes results charge transfer with the semiconductor scaffold to create mobile opposite charges in the dye layer that hop by self-exchange electron transfer across the semiconductor surface and ultimately accumulate on electrocatalysts or recombine. Additionally, simulations of transient absorption pulsed-laser experiments are compared to those of continuous illumination conditions to determine if pulsed-laser experiments can be used as an analog for real-world sunlight illumination conditions. Results from these simulations help to expand the current knowledgebase for these systems via rapid analysis of a wide range of parameter sets in order to better understand the limitations and possibilities for these and similar systems. It is our hope that these results are used by researchers to better focus their efforts in developing cost-competitive dye-sensitized solar energy conversion technologies.

INTRODUCTION

As the effects of global climate change become more apparent it is increasingly obvious that a paradigm shift is necessary in the sources from which we harvest our energy. On the forefront of possible sources are both wind and solar energy due to their relative abundance and ease of conversion into electricity. The leading solar energy conversion technology is the photovoltaic solar cell. Over 90% of the solar cell market is dominated by light-absorbers consisting of crystalline silicon, with the remaining fraction dominated by toxic crystalline CdTe. Although crystalline silicon is economically most viable, it is by no means the optimal light-absorber material for a solar cell, because it requires an expensive purification process, its indirect bandgap results in it being a weak light-absorber near its bandgap energy minimum, and recombination is dominated by Auger bulk non-radiative recombination, therefore decreasing its maximum efficiency versus other 1.1 eV bandgap materials with direct bandgap transitions.[1] Optimal solar cells contain light absorbers whose recombination is entirely radiative, which is the minimum rate of recombination for any solar energy conversion device.[2] Thus, opportunities remain to improve on commercial crystalline silicon solar cells using other solar cells that utilize lower-cost and more efficient materials. One such technology is the dye-sensitized solar

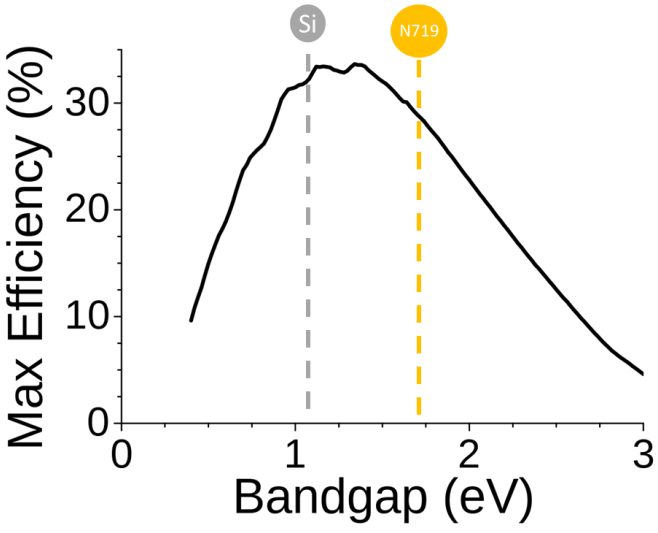


Figure 1. Maximum efficiency by bandgap or dye quasi-bandgap as determined by the Shockley–Queisser limit.

cells (DSSC), whose world record certified sunlight-to-electrical power conversion efficiency is presently only 14%.[3] While this falls below efficiencies of commercially available silicon solar cells (22.7%)[4], DSSCs may be advantageous for niche applications because of their possibility for low-temperature processing and manufacturing and tunability of the light-absorber to enable idealized bandgap energies and few non-radiative pathways for recombination.[5]

DSSCs make use of dye molecules as their light absorbing medium. They typically consist of a thin film of high-surface-area metal-oxide semiconductor with surface-anchored dyes, a heterogeneous-catalyst-coated counter electrode, and a redox-active electrolyte to mediate the charge between the two electrodes. Most of the state-of-the-art demonstrations

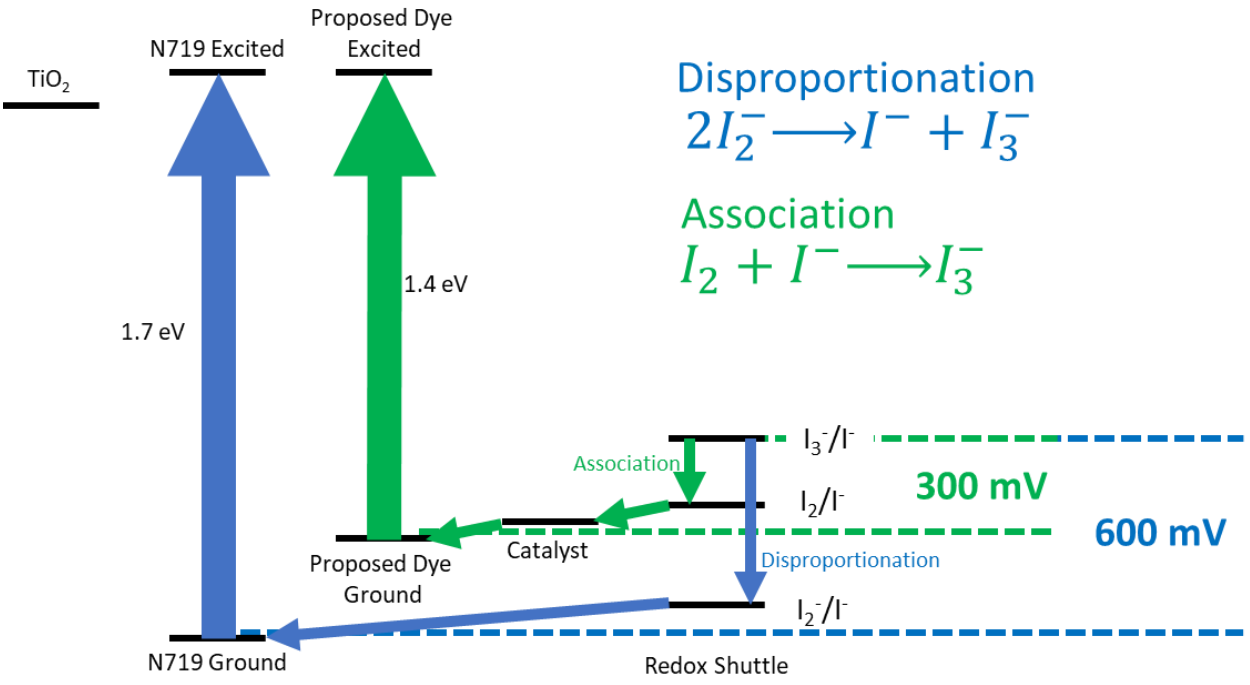


Figure 2. Scheme showing alternate kinetic pathway (in green) to dye regeneration which allows for less energy to be wasted and therefore a smaller quasi-bandgap dye that absorbs more light than traditional pathways such as the one shown in blue.

have historically utilized nanocrystalline anatase-phase TiO_2 as the metal-oxide semiconductor, platinum nanoparticles as the counter-electrode catalyst, and triiodide/iodide (I_3^-/I^-) in nitrile-based solvents as the electrolyte redox shuttle. Presently, even the leading DSSC dyes (N719, SM317, ADEKA-1+LEG4) have differences in energies of their lowest-energy vibronic states (E_{00}), which can be thought of as quasi-bandgaps, that are far from ideal (1.7 eV, 1.8 eV, 1.9 eV respectively) (Figure 1).[6] A major reason for this limitation in these DSSCs is the fact that the typical I_3^-/I^- redox shuttle used to regenerate the dyes wastes on the order of 500 mV as heat during disproportionation of two molecules of I_2^- into I_3^- and I^- (Figure 2). This loss could be mitigated by taking an alternate kinetic pathway by which I^- is directly oxidized into I_2, or even better yet, into I_3^-. The downside to these lower-energy pathways is that either reaction requires that two electrons are transferred, necessitating that two electron-holes accumulate at a single site. This is typically not possible on dye molecules and so the introduction of catalysts is required. These catalysts can receive electron-holes through electron transfer reactions from other molecules on the film that can propagate from dye molecule to dye molecule through self-exchange electron transfer reactions.[7] However, the process of charge accumulation on catalysts sites is both complex and not fully explored. While experimental efforts are eventually needed, initial stochastic modeling work on this sort of system has been done to better explore the parameter space of this system.

Such modeling allows for this highly complex system to be evaluated by easily adjusting single parameters in order to identify ideal conditions that are worth studying experimentally. Additionally, it helps to identify limits in certain systems so that experimentally it can be determined how well devices are performing relative to their

maximum potential performance. This thesis primarily describes a model created to evaluate such systems. Other similar models have been created before, but none are as thorough in terms of features desired as this one.[8,9] In addition to simply describing this model (primarily in the Appendices), Chapters 1 and 2 of this thesis describe studies that use this model to identify useful parameter spaces for DSSCs and related dye-sensitized photoelectrochemical cells that directly drive energy-storing fuel-forming reactions.

Chapter 1 aims to answer some basic questions including: "What is the ideal ratio of dyes to catalysts on a film?", "How greatly does an absorbance distribution such as the one created by the Beer-Lambert Law perturb the yield for catalyst turnover?", "What is the fluence dependence of the yield for catalyst turnover?", "How much more difficult is it to accumulate 4 electron-holes on one catalyst – as desired for O_2 evolution through water oxidation – as opposed to just 1 or 2 – as desired for iodide oxidation or proton reduction to H_2?", and "Are pulsed laser experiments a useful analog for studying materials behavior for systems under continuous illumination?".

Chapter 2 aims to answer some questions related to the geometry of the system. These include: "Does a series of separated particles accurately represent a morphologically complex film?", "To what degree does interparticle necking matter?", "Can this complex system more simply be represented by a square with periodic boundary conditions yet still result in similar conclusions?", "Does the use of polarized light for excitation perturb the results of the model?", and "What extent of surface coverage of dye molecules is necessary for a substantial percolation network to exist?"

The code for the models and comprehensive instructions for the use of that code are found in the appendices. The model was originally written in Mathematica and then

converted to Python but all studies herein were conducted using some version of the Mathematica model. Explanations of both versions of the model can be found in Appendix A (Mathematica) and Appendix C (Python) with full copies of the code for reference in Appendices B and D for Mathematic and Python, respectively.

Chapter 3 is an unrelated piece of work describing a scientific outreach experiment designed to educate middle school through undergraduate-level students about ion motion and desalination. Appendix E describes the outreach experiment discussed in Chapter 3 as well as provides pictorial instructions. Appendix F is another unrelated piece of work describing the design and construction of a slurry baggie solar water splitting Z-scheme reactor, which is described in detail in a prior review article from our group.[10]

CHAPTER 1. Preliminary Model Testing

Introduction

Designing and evaluating architectures for solar fuels generation are worthwhile academic research endeavors that may one day lead to an economically pertinent technology that enables long-term seasonable energy storage and/or a transportation fuel.[1–5] This type of energy storage is predicted to be necessary when society is powered by substantial renewable energy.[5] Architectures for solar fuels constructs generally fall into several broad categories. The most efficient designs consist of photovoltaic-grade materials with buried-junctions for effective photovoltaic action and that are protected from corrosion using chemically insulating overlayer coatings or direct electrical wiring to aqueous electrolytes where materials electrocatalysts perform the electrochemical reactions.[3,6–11] Other designs rely on coupled processes that together are much less well understood and often occur at semiconductor–liquid junctions with or without molecular electrocatalysts and/or dye sensitizers.[12–31] Each of these constructs has benefited from experimental and computational studies of its photophysical and photochemical processes in order to elucidate mechanistic details of operation and identify architectures that result in large power-conversion efficiencies. Some models capture bulk collective dynamics and overall photovoltaic performance using statistical ensemble models. For example, limiting physical processes in buried-junction designs and non-molecular photoelectrochemical designs have been simulated successfully using coupled differential equations that capture deterministic behaviors expected from statistical thermodynamics.[4,32–34] For mesoporous dye-sensitized designs, transport phenomena for redox-active species in solution and rates of electron transport between dyes and within mesoporous thin films have been modeled using various

methods with numerical results that are reasonably consistent with experimental observations.[35–44] However, a limitation of simulations that capture continuous and/or bulk behaviors is that they lack the granularity required to capture dynamics that occur at discrete molecular light absorbers and electrocatalysts. The molecularity of these photochemical designs can be studied using ab initio calculations, density functional theory, electronic structure determination, and molecular dynamics simulations. However, these atomic-level calculations are too fine-grained to capture dynamics that occur across nanometer-to-micron-sized regions consisting of hundreds to thousands of molecules that are critical in order to predict the overall function of the materials system. A modeling domain that is intermediate between these two size regimes is required to capture the micro-kinetic behavior of these systems on pertinent size scales. This need motivated us to develop a physically pertinent numerical modeling and simulation package based on a discrete-time random walk Monte Carlo method and that we will share publicly. It is the first of its kind that captures salient features of dye-sensitized and cocatalyst-modified constructs with the aim to help guide and progress the design of these systems to a practical level of device viability.

An enormous number of fundamental experiments have been conducted on dye-sensitized mesoporous thin films using a broad range of techniques.[45,46] To better understand observed behaviors related to charge transfer, Monte Carlo simulations have been performed that simulate Markovian micro-kinetic processes and quantify rates of electron and energy transfer between dyes only.[45,47] Some of the initial work was reported by Meyer and colleagues in the early 2000s, who modeled surface transport processes via classical *discrete-time* random walk Monte Carlo simulations across a two-dimensional

lattice with periodic boundary conditions.[48] Around the same time, Nelson, Durrant and colleagues introduced a mathematically rigorous model for charge recombination from these TiO_2 nanocrystallites to surface-bound dyes based on a *continuous-time* random walk model.[42,49–52] A critical assumption in this type of random walk model is that the walkers are independent and also that the location of the oxidized dye does not change appreciably on the timescale of the recombination process, which is not often a valid assumption.[37,53–62] Since that time, additional random walk Monte Carlo models have been reported for analogous processes and using computer code with similar features and limitations as first reported in the early 2000s.[41,63–67] In 2009, Ardo and Meyer were the first to incorporate specifically spherical nanoparticle supports into discrete-time random walk models, thus removing the need for periodic boundary conditions.[58,59] This was an important advance, because it remedied the non-physical limitation of the two-dimensional simulations, which over-counted regions on the particles near the poles and contained no accurate means to accurately quantify spherical polar angular position in three dimensions. Knowing the spatial positioning of each perturbed dye is important when modeling experimental data obtained using time-resolved polarization spectroscopy techniques that can be used to measure rates of transport across surfaces such as self-exchange electron transfer or energy transfer across nanometer-scale particles and on the nanosecond and longer timescales.[58,59] Since then other discrete-time random walk models have incorporated three-dimensional semiconductor nanoparticles[60,61,68,69] and even included surface-confined interparticle charge transport across necking regions.[67,70] Interparticle charge transport is an important process that captures dynamics occurring over the scale of several semiconductor nanoparticles. While our code described herein is also able to simulate interparticle charge

transport behavior, it is not utilized in our initial simulations because that process is of secondary importance to the majority intraparticle kinetic processes. Unique to our model, in comparison to all other prior models,[45,47,71,72] is that we identify kinetic parameters that lead to the most effective utilization of photons *for turnover of multiple-electron-transfer cocatalysts* under the simulated condition of pulsed-light excitation *or continuous illumination*. We report results from a series of parametric time-inhomogeneous random walk Monte Carlo simulation studies using isolated spherical nanoparticles arranged as a stack to mimic their spatial location as a thin film. These results are highly pertinent to dye-sensitized cocatalyst-modified semiconductor nanoparticles that constitute mesoporous photoelectrochemical electrodes or consist of colloidal suspensions.

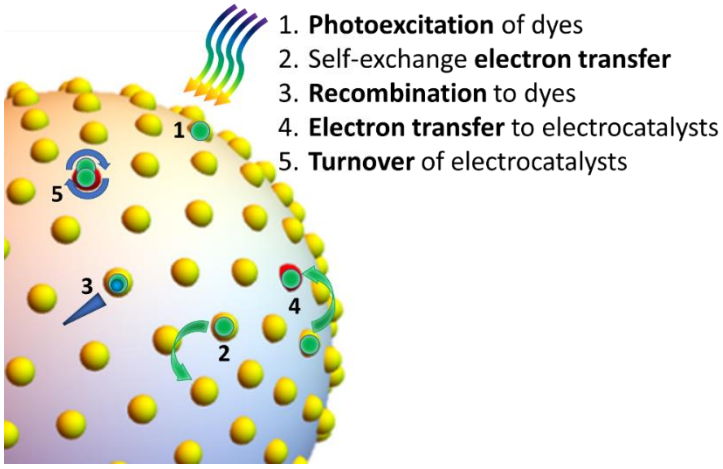


Figure 1.1. Model schematic showing the events that are included in the model to mimic the major kinetic processes that are operative in actual dye-sensitized photoelectrochemical constructs.

Experimental

Modeling Framework.

The architecture modeled is motivated by mesoporous thin films of nanoparticles that are commonly used in dye-sensitized photoelectrochemical constructs, where nominally identical spherical anatase TiO_2 nanocrystallites contain discrete surface-anchored light-absorbing moieties and redox-active electrocatalysts (Figure 1.1). In the case

of traditional dye-sensitized materials, both the light-absorbers and the electrocatalysts are molecules, but the model is general in that, for example, the light-absorbing units could be surface-confined material units like quantum dots or nanocrystalline regions with isolated optical transitions in the solid-state, or the electrocatalysts could be materials whose charge localization and transport follows a hopping or polaronic transport mechanism.[73] The model is able to simulate discrete processes that spatially exchange states, such as self-exchange Dexter or Förster energy transfer or self-exchange electron transfer initiated at an oxidized or reduced dye. Self-exchange electron transfer is the process assumed for the simulations performed herein with hops to only the closest adjacent dyes being possible, which is a valid assumption based on reasonable conditions and prior analyses.[74] The structure is incorporated into the model as 100 spheres that are positioned optically in series as a one-dimensional stack but that do not physically interact. The top sphere in this stack is considered to be at the surface of the thin film with subsequent spheres further down from the surface, at larger z-coordinates. The surfaces of these spheres are tessellated as icosahedra, using Wolfram Mathematica's built-in "Geodesate" function, which results in approximately evenly spaced points that represent possible locations of molecules. By tessellating 5 icosahedra, 252 points were generated on the surface of the sphere, with 240 hexagonally packed (6 adjacent points), and the remaining 12 pentagonally packed (5 adjacent points). It was not necessary to specifically identify the nanoparticle radius, molecule radius, film thickness, and film porosity, because they are all related and so only their relative sizes are pertinent. However, based on the values chosen for the number of locations for molecules per particle (252), the number of particles per stack (100), and the use of a stack to model a mesoporous film of ~50% porosity, the geometry is consistent with

characteristics of typical dye-sensitized mesoporous TiO_2 thin films.[60,74,75]Per particle, a specific number of these 252 points was chosen as positions of electrocatalysts that could be oxidized/reduced once or multiple times. Multiple transfers are desired in practical applications that make and break stable chemical bonds via multiple-electron/proton-transfer reactions. The position of each electrocatalyst on a single particle was chosen at random, with an additional option to evenly distribute the electrocatalysts over each particle such that each particle had the same number of electrocatalysts. The remaining points were chosen to be dyes and based on the pulsed photon fluence chosen for the experiment, locations for initial photoexcitation were chosen as a subset of dye positions. All simulations assumed unity quantum yield for rapid excited-state electron transfer between photoexcited dyes and the semiconductor support, such that photoexcitation always resulted in an oxidized/reduced dye molecule. For each semiconductor nanoparticle, its number of mobile electrons/holes was set equal to the number of oxidized/reduced molecular charges on its surface; however, the transport processes of the electrons/holes were not simulated. *Information regarding generation of initial conditions are described in more detail below.* The simulation proceeded by randomly choosing from a series of options at each timestep, including self-exchange electron transfer between two adjacent dyes or electrocatalysts, electron-transfer recombination between the semiconductor nanoparticle and an oxidized/reduced dye or electrocatalyst, photoexcitation of a ground-state dye – when conditions of continuous illumination were simulated – or doing nothing. When an electrocatalyst reached a redox state required for an electrocatalytic turnover event, the electrocatalyst was immediately regenerated and the same number of charges in the semiconductor nanoparticle were removed to simulate their collection elsewhere in the

system. This occurred repeatedly until all charge-separated states either recombined or drove electrocatalysis. *Information regarding this simulation loop and the resulting output data are described in more detail below.*

Generation of Initial Conditions. Initial assignment of photoexcited dyes, and therefore charge separated dyes, was performed multiple ways depending on the desired simulated condition. A set number of photoexcited dyes was either distributed over the entire stack or placed on each particle in the stack, e.g. for the case of 200 photoexcitations over the 100-particle stack ($<n_{pe}> = 2$), either 2 dye positions were chosen randomly per particle or 200 particle numbers and dye positions were chosen randomly across the entire stack. The assignment was made using weights incorporated via an assignment matrix, with weights based on one or more geometric considerations. One option for the assignment matrix was a Beer–Lambert law generation profile,

$$W_{\mathrm{BL}} = 10^{-\frac{n}{N}(\mathrm{Abs})} = 10^{\frac{n}{N}\log_{10}(T)} \qquad (1)$$

where the probability of photoexcitation decreases exponentially as the position of the dye is deeper in the stack, W_{BL} is the weight associated with a given position from the Beer–Lambert law weighting function and ranges from 0 to 1, n is the particle number in the stack, N is the total number of particles, Abs is the absorbance of the entire particle stack, and T is the fraction of transmitted light through the entire particle stack. As such, molecular positions closer to

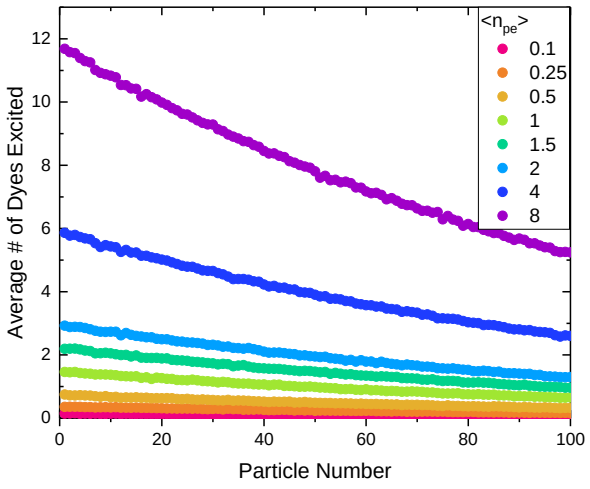


Figure 1.2. Simulated assignment of photoexcited dyes based on the Beer–Lambert law as a function of particle number/depth at the indicated excitation fluences and repeated a total of 50,000 times per condition.

the top of the particle stack were more likely to be photoexcited. This assignment, if repeated a statistically significant number of times, yields a distribution of excited dyes that follows an exponential decay with particle height, as predicted by the Beer–Lambert law (Figure 1.2). Another independent option for the assignment matrix was based on polarized light excitation and well-defined radial transition dipole moments for the surface-anchored dyes. This assignment weighs each position based on the inclination angle of the dye relative to the electric field vector of the polarized excitation light,

$$W_{\mathrm{A}} = \cos^2\theta \qquad (2)$$

where W_{A} is the weight associated with a given position from the anisotropy weighting function and θ is the angle between the normal from the center of a particle and the molecular position on its surface and the electric field vector of the polarized excitation light. Prior to performing the simulations, a list of data for each electrocatalyst, dye, and oxidized dye was generated that contained relevant parameters including molecule type (dye or electrocatalyst), recombination probability, hopping probability, oxidation state, and an array of positions for 5 or 6 adjacent molecules. Adjacent molecules were within 2.5 times the center-to-center distance between molecules the size of $[Ru^{II}(bpy)_3]^{2+}$ when in van der Waals contact. The recombination probability was set to be the same for electron transfer between the semiconductor and either an oxidized/reduced dye or an electrocatalyst. Also, when an oxidized/reduced dye is adjacent to an electrocatalyst, the hopping probability to the electrocatalyst was set to effectively 90%. *This latter point is described in more detail below.* These hopping and recombination probabilities were calculated from time constants ranging from 40 ns to 800 μs in steps of three points on a logarithmic scale. For each nanoparticle, the probability of recombination was scaled by the number of charges in that

semiconductor nanoparticle. A second list of information was generated that was updated at each Monte Carlo iteration during the simulation, which included the two-dimensional coordinates for all molecules that were altered from their initial state as (particle number, position number). At time zero this list only contained the locations of dyes that were initially photoexcited; as the initially photoexcited dyes became altered from their initial state over time these coordinates were replaced by those of other dyes or electrocatalysts.

Simulation Loop. After initially defining the state of the system at time equal to zero, Monte Carlo simulations were performed by looping over the list of molecules that were altered from their initial state (second list). For each, a probability, P_x, was assigned that ranged from 0 to 1 for the possible options of recombination, hopping to an adjacent point, or doing nothing, and with probabilities defined as follows,

$$P_x = \frac{t_{\text{step}}}{\tau_x} \tag{3}$$

where t_{step} is the amount of time between time points and τ_x is the ensemble average time constant for the process, x. At the end of each simulated timestep, the list of molecules was altered from its prior state and then the Monte Carlo process was repeated, assuming that a small and predefined timestep had passed. The value of the timestep varied and was chosen for each condition so that P_x as a percentage was < 1.1% for self-exchange electron transfer between dyes and was < 0.3% for recombination to oxidized/reduced dyes or electrocatalysts. The value of the timestep resulted in the probability of transferring a charge from an oxidized/reduced dye to an adjacent electrocatalyst being ~30%. This probability was set to be 27 times greater than the probability of transferring a charge between adjacent dyes via a self-exchange reaction in order to reflect the reasonable condition that electron transfer to/from an oxidized/reduced dye from/to an electrocatalyst is thermodynamically

Table 1.1. Values and expressions used for parameters in the Monte Carlo simulations.

Name	Value(s)	Unit
$\tau_{\text{hop(Dye–Dye)}}$	40, 80, 160, 400, 800, 1600, 4000, 8000, 16000, 40000, 80000, 160000, 400000, 800000	ns
$\tau_{\text{hop(Cat–Cat)}}$	$\tau_{\text{hop–DyetoDye}}$	ns
$\tau_{\text{hop(Dye–Cat)}}$	$\tau_{\text{hop–DyetoDye}}$ / 27	ns
$\tau_{\text{hop(Cat–Dye)}}$	$\tau_{\text{hop–DyetoDye}}$ x 10^{13}	ns
$\tau_{\text{recomb(SC–Dye)}}$ per particle	40, 80, 160, 400, 800, 1600, 4000, 8000, 16000, 40000, 80000, 160000, 400000, 800000	ns
$\tau_{\text{recomb(SC–Cat)}}$ per particle	$\tau_{\text{recomb–SCtoDye}}$	ns
time step, t_{step}	Minimum[3.75 x $\tau_{\text{hop–DyetoDye}}$, $\tau_{\text{recomb–SCtoDye}}$] / 350	ns
number of trials per data point	25	–
percent of incident light transmitted through the thin film	43.4	%
number of initially excited dyes per stack	10, 50, 100, 200, 400, 800, 2000, 4000, 8000, 16000	–
number of particles in the stack	100	–
number of molecular positions (points) per particle	252	–
percent surface coverage of molecules	100	%
maximum number of points adjacent to each molecule	6†	–
maximum redox state of electrocatalysts	1, 2, 4	–
number of electrocatalysts per stack	252	–
number of electrocatalysts per particle††	2	–
number of initial photoexcitation events per particle (n_{pe})††	1, 2, 4, 8, 20	–

favorable and thus much more probable. The exact 27-times-greater probability was chosen such that there was exactly a 90% probability this would occur after an oxidized/reduced dye on hexagonally packed sites became adjacent to an electrocatalyst, with a nominally lower probability of occurring on pentagonally packed sites. The derivation of this probability is shown in Equation 4.

$$\sum_{n=0}^{\infty}\left(\frac{27}{27+5}\right)\left(2\left(\frac{1}{27+5}\right)\right)^{n}=\left(\frac{27}{27+5}\right)\left(\frac{1}{1-\frac{2}{27+5}}\right)=\left(\frac{27}{27+5-2}\right)=0.9 \qquad (4)$$

This Monte Carlo process was repeated until no oxidized/reduced dyes remained. Specific parameters used for various model inputs are listed in Table 1.1. An additional option in the model was its ability to mimic conditions of continuous illumination, which incorporated repeated light excitation events. The initial number of photoexcited dyes was set to zero and after each timestep there was an additional probability for photoexcitation that scaled based

on the desired intensity of solar illumination and was weighted according to the assignment matrix calculated from the Beer–Lambert Law and polarization considerations. Simulations using the condition of continuous illumination were terminated after 25,000,000 iterations and all data was used to calculate time-averaged steady-state values. While these values included the initial data prior to reaching a steady-state condition, its influence on the average values was insignificant because its inclusion only resulted in a < 0.25% change in the value, on average.

Output Data.

During the simulation the number of times that an electrocatalyst is *completely* oxidized/reduced by a dye is recorded, because it is the most useful parameter to quantify the effectiveness that a condition drives solar photochemical transformations. From this it is possible to calculate the percent turnover, i.e. the percentage of photoexcitations that contributed to turnover of an electrocatalyst. Photoexcitations that contributed to electrocatalyst oxidation/reduction but did not result in electrocatalyst turnover did not count toward this total. To reduce computation time, photoexcitations that were not able to contribute to electrocatalyst turnover were identified and removed from the simulations before any timesteps had been performed. This occurred when a photoexcited dye was on a nanoparticle that either had zero electrocatalysts on its surface or had fewer photoexcited dyes than the maximum oxidation/reduction state of an electrocatalyst. When these photoexcited dyes were removed, they were counted as being unproductive toward electrocatalyst turnover. Data were collected under a wide range of starting conditions including varied maximum redox state of the electrocatalysts, number of electrocatalysts, initial excitation fluence (i.e. number of initially excited dyes), use of a Beer–Lambert law

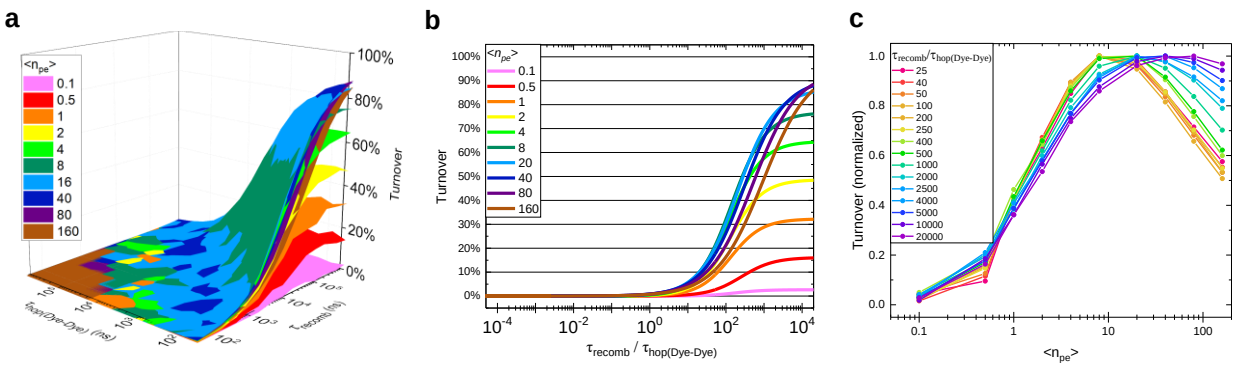


Figure 1.3. (a) Sheet plots representing the percentage of photoexcited dyes that ultimately *contribute to double oxidation/reduction of an electrocatalyst and turnover*, when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences. (b) Non-linear least squares sigmoidal best-fits of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant. (c) Plot of the data in panel a as a function of the initial pulsed-light excitation fluence at the indicated ratio of the recombination time constant to the hopping time constant.

distribution when assigning dye photoexcitations throughout the stack, self-exchange electron-transfer time constant between adjacent molecules, and electron-transfer recombination time constant between surface-anchored molecules and photo-generated charges in the semiconductor support.

Results and Discussion

General simulation conditions and data interpretation for the base case.

For each specific condition simulated herein, turnover percentage is reported as a function of 1 of 14 logarithmically-spaced hopping time constants, 1 of 14 logarithmically-spaced recombination time constants, and 1 of up to 10 logarithmically-spaced initial excitation fluences. These fluences are quantified as the average number of photoexcitations per particle over the stack of 100 particles and so a fluence of $<n_{pe}> = 2$ means that there were on average 2 photoexcitations per particle, or 200 photoexcitations created over the stack of 100 particles. Each combination of parameters for these three variables was simulated 25 times, and therefore 2500 semi-independent particles were analyzed resulting

in a total of 250 – 400,000 photoexcited dyes being averaged per condition. The particles are semi-independent in that electron transfer did not occur between molecules on separate particles, but that photoexcitation that followed the Beer–Lambert law generated an unequal number of initially photoexcited dyes on each particle such that particles nearer to the top of the film had more oxidized/reduced dyes while those farther from the top of the film had fewer oxidized/reduced dyes.

The data presented herein are reported as three-dimensional contour plots, one for each excitation fluence, as a function of the hopping time constant and recombination time constant (Figure 1.3a). The base case model used to obtain the data shown in Figure 1.3 included polarized Beer–Lambert law weighting to assign a distribution of photoexcited dyes, and electrocatalysts that could be maximally oxidized/reduced twice and occupied 1% of the possible molecular positions. Each three-dimensional contour plot for this condition changes monotonically as values on either axis increase, and therefore a series of single-fluence contour plots can easily be visualized as a series of three-dimensional sheet that spans all possible hopping and recombination time constants. This method of data visualization helps one identify the optimal fluence for ranges of kinetic parameters, as evidenced by sheet crossover. An example of this is the band of green shown crossing through the light blue sheet in Figure 1.3a as the percent turnover sharply increases. Visualizing the range of kinetic parameters that lead to band formation, i.e. crossing of two sheets, can provide insights into differences in nearly identical monotonic behavior. However, it is also apparent from Figure 1.3a that the percent turnovers are nearly the same for each ratio of the recombination time constant to the hopping time constant and thus, the observed independent variable is not the hopping time constant or the recombination time

constant but is instead their ratio. This means that a two-dimensional plot that captures the overall effect represented by the sheets can be generated using the recombination-to-hopping time-constant ratio as the independent variable. This is shown as two-dimensional plots in Figure 1.3b, which were obtained by recasting all points for each sheet shown in Figure 1.3a with the recombination-to-hopping time-constant ratio as the independent variable and fitting the data to the sigmoidal function shown using non-linear least-squares ($R^2 > 0.975$ except for the case of $\langle n_{pe} \rangle = 0.1$ which resulted in poor signal to noise). As fluence increases from $\langle n_{pe} \rangle = 0.1$ to $\langle n_{pe} \rangle = 8$, the maximum percent turnover increases monotonically but maintains the same functional form. From $\langle n_{pe} \rangle = 8$ up to the maximum of $\langle n_{pe} \rangle = 160$, the steep portion of the sigmoidal fit shifts to larger recombination-to-hopping time-constant ratios but still reaches the same maximum percent turnover. Larger recombination-to-hopping time-constant ratios are optimal because hopping is critical to electrocatalyst turnover while recombination is detrimental. The maximum percent turnover is only ~90%, because ~10% of dye photoexcitations occur on particles containing zero electrocatalysts based on the fact that electrocatalysts are distributed randomly at an average of 1% coverage per particle.

The two-dimensional plots shown in Figure 1.3b report the percent turnover as a function of the ratio of the kinetic parameters, and they span the range of excitation fluences. A variation on Figure 1.3b is shown in Figure 1.3c, where the parameters are rearranged so that the normalized percent turnover is reported as a function of the excitation fluence, and they span the range of ratios of the kinetic parameters where hopping is more probable than recombination. It is apparent from these data that an intermediate fluence is ideal for each specific recombination-to-hopping time-constant ratio. The generally downward concave shape to the data occurs because the complete twice-oxidation/reduction of each electrocatalyst is less likely to occur at low fluence while the equal-concentration second-order recombination behavior is more detrimental to the percent turnover at high fluence. However, even though these data show that higher fluences result in a smaller relative value for percent turnover, the overall rate of turnover events still increases at higher fluences, as seen in Figure 1.4. Also, as the recombination-to-hopping time-constant ratio increases, the

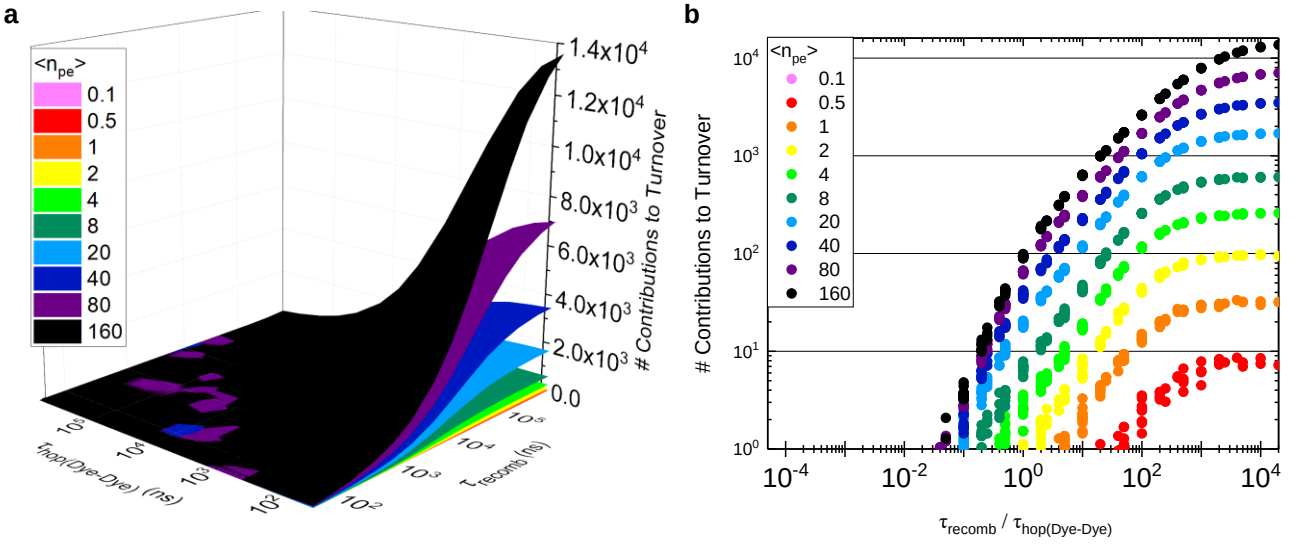


Figure 1.4. (a) Sheet plot representing the number of photoexcited dyes that ultimately *contribute to double oxidation/reduction of an electrocatalyst and turnover* when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences. (b) Representation of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant using base-10 logarithmic scaling of the y-axis values so that lower fluence data can be seen more clearly.

optimal fluence, indicated by the global maximum of the data, decreases slightly and then greatly increases because recombination is relatively slow and therefore equal-concentration second-order recombination does not outcompete photoexcitation until large fluences are used.

Effect of electrocatalyst behavior.

To understand the role that the redox state required for electrocatalyst turnover plays in the outcomes of the simulations, we performed simulations using electrocatalysts that each required only a single redox event for turnover (Figure 1.5). The general trends observed are very different than those observed for electrocatalysts requiring two redox events for turnover (Figure 1.3). For example, at lower fluences the probability of electrocatalyst turnover is small when it requires two redox events (Figure 1.3a,b, in pink) whereas the probability can be large when a single redox event is required for electrocatalyst turnover (Figure 1.5a,b, in pink). This drastically different behavior occurs at low fluences, because many photoexcitation events occur on particles where there are too few oxidized/reduced dyes to perform multiple redox reactions with any given electrocatalyst.

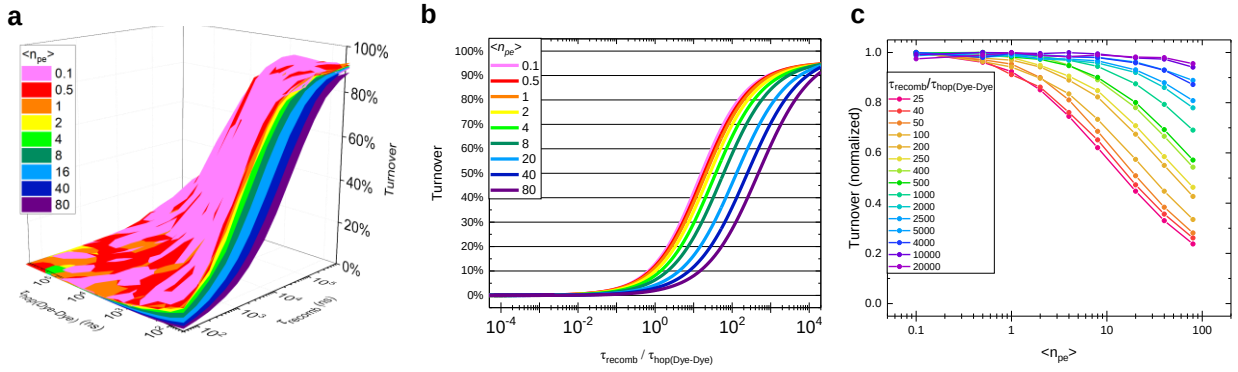


Figure 1.5. (a) Sheet plots representing the percentage of photoexcited dyes that ultimately *contribute to single oxidation/reduction of an electrocatalyst and turnover*, when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences. (b) Non-linear least squares sigmoidal best-fits of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant. (c) Plot of the data in panel a as a function of the initial pulsed-light excitation fluence at the indicated ratio of the recombination time constant to the hopping time constant.

As the fluence increases, photoexcitations become concentrated enough that they are reliably created in sufficient numbers to oxidize/reduce electrocatalysts once or twice as needed for turnover. However, then the limitation in the percent turnover is the ratio τ_{recomb} / $\tau_{hop(Dye–Dye)}$, where faster relative rates of hopping (small $\tau_{hop(Dye–Dye}$) are more beneficial to percent turnover (Figure 1.3b and Figure 1.5b), as described above. Another notable difference that arises from decreasing the number of redox events required for electrocatalyst turnover is shown in Figure 1.5c versus Figure 1.3c. Unlike the case when each electrocatalyst requires two redox events for turnover, single redox events at electrocatalysts are most likely to occur at the lowest fluences. At very high fluences, the relative percent turnover is small irrespective of the redox state required for electrocatalyst turnover. This behavior is almost entirely dictated by τ_{recomb} / $\tau_{hop(Dye–Dye)}$, where faster hopping (small $\tau_{hop(Dye–Dye)}$) and slower recombination (large τ_{recomb}) are optimal and conditions of higher fluence suffer from increased rates of recombination due to it being an equal-concentration second-order kinetic process in the number of oxidized/reduced molecules per particle. In summary, low fluence is optimum when electrocatalyst turnover requires single redox events. However, when electrocatalyst turnover requires two redox events, $<n_{pe}> \approx 10$ is optimum at small values of τ_{recomb} / $\tau_{hop(Dye–Dye)}$ and this optimal value for $<n_{pe}>$ increases as τ_{recomb} / $\tau_{hop(Dye–Dye)}$ increases (Figure 1.3c).

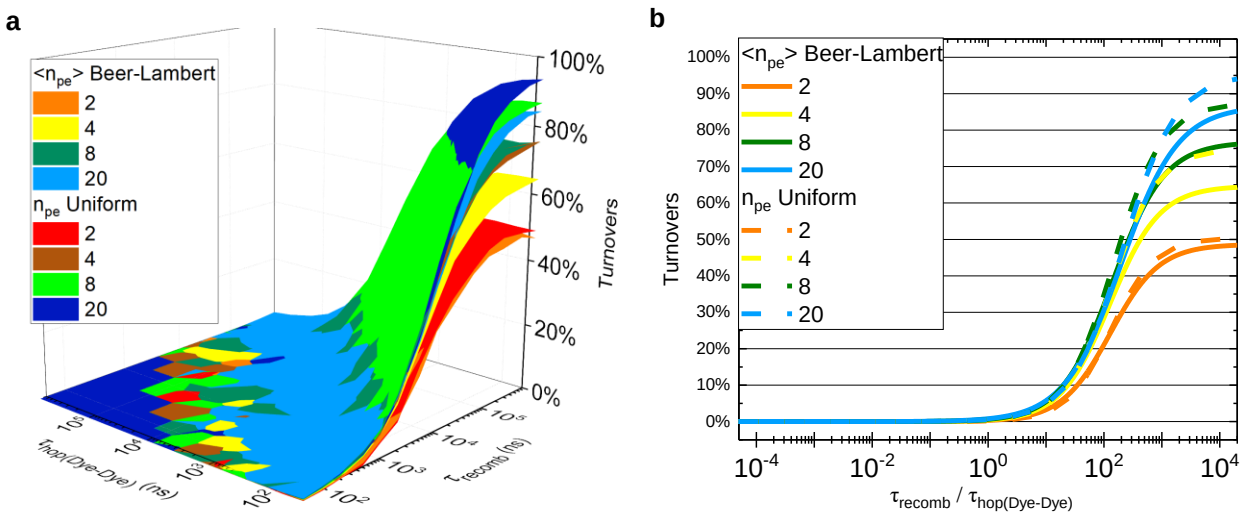


Figure 1.6. (a) Sheet plots representing the percentage of photoexcited dyes that ultimately contribute to double oxidation/reduction of an electrocatalyst and turnover when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences *that follow the Beer-Lambert law or a uniform distribution over the stack*. (b) Non-linear least squares sigmoidal best-fits of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant.

Effect of the Beer–Lambert law.

Use of the Beer–Lambert law to model the photoexcitation distribution in mesoporous thin films used in dye-sensitized solar cells is in general accurate for non-scattering films. However, to understand the influence that the photoexcitation profile has on electrocatalyst turnover we compared the condition where photoexcitation events followed a Beer–Lambert law distribution to the condition where the number of photoexcitation events was the same for each particle and therefore spatially homogeneous over the stack (Figure 1.6 and Figure 1.7, respectively). The lowest possible fluence resulting in homogenous photoexcitation events (n_{pe} = 1) resulted in only one oxidized/reduced dye per particle, and therefore a 0% chance of turnover for electrocatalysts requiring two or more redox events for turnover. In this case, use of the Beer–Lambert law distribution was beneficial. However, for all other values of $<n_{pe}>$ evaluated, the percent turnover is larger when uniform photoexcitation occurs instead of using a Beer–Lambert law excitation profile.

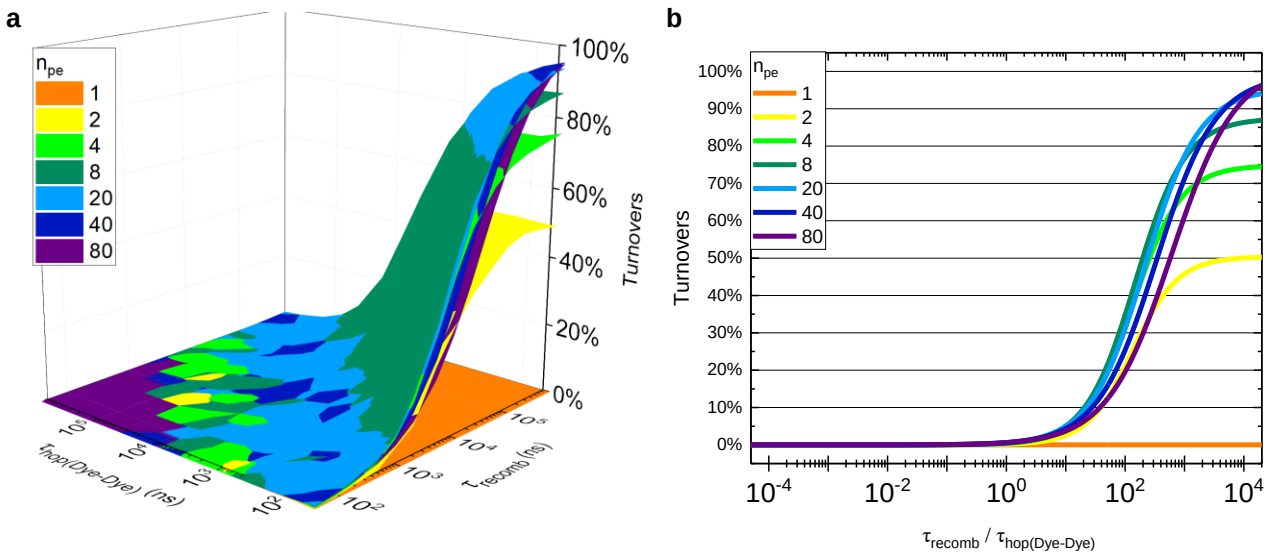


Figure 1.7. (a) Sheet plot representing the percentage of photoexcited dyes that ultimately contribute to double oxidation/reduction of an electrocatalyst and turnover when *electrocatalysts are present at exactly 2 per particle* at the indicated initial pulsed-light excitation *fluences as a uniform distribution over the stack*. (b) Non-linear least squares sigmoidal best-fits of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant.

For example, Figure 1.6 shows nearly identical plots for the dark green sheet (<n_{pe}> = 8, with Beer–Lambert law generation) and the brown sheet (n_{pe} = 4, without Beer–Lambert law generation) meaning that uniformly exciting dyes is approximately the same as having twice as many total excitations that follow a Beer–Lambert law distribution. This is because the Beer–Lambert law distribution often results in some photoexcitations that occur too sparsely to be useful and others that are so concentrated that the equal-concentration second-order nature of the recombination process results in more rapid loss of oxidized/reduced dyes. That is, toward the bottom of the stack it is likely that some photoexcitations occur on particles with no other photoexcitation events and therefore these events are never able to contribute to the two redox events required for turnover of an electrocatalyst. And at the top of the stack the rate of recombination is fast because these particles often have significantly more photoexcitations per particle than <n_{pe}>. Also, notably for the condition of <n_{pe}> = 2, uniform photoexcitation provides little benefit over photoexcitation that follows a Beer–Lambert law distribution, because the rates of equal-

concentration second-order recombination are not drastically different for particles with few photoexcitations. Collectively, these data suggest that optimal conditions include having a very thin layer of strongly-absorbing material or a thick layer of weakly-absorbing material. Alternatively, introducing scattering particles to more evenly distribute the incoming light across the stack is beneficial. Non-uniform photoexcitation is also problematic for fundamental studies of charge carrier dynamics and interfacial electron-transfer processes measured using transient absorption spectroscopy, because the ensemble kinetic behavior simultaneously reports on several simple first-order and/or second-order kinetic processes but under different initial excitation conditions. The aggregate transient absorption signal therefore not follow traditional kinetic models, which is a behavior that has been reported previously in the literature.[45,47,49–52,61]

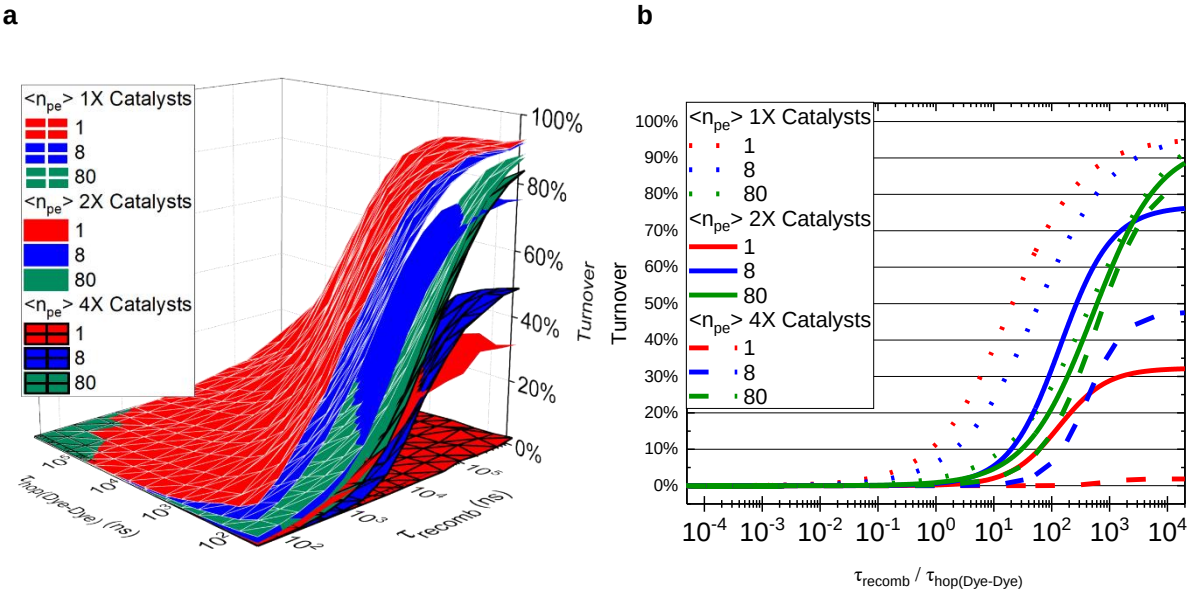


Figure 1.8. (a) Sheet plots representing the percentage of photoexcited dyes that ultimately *contribute to the indicated single (1X), double (2X), or quadruple (4X) oxidation/reduction of an electrocatalyst and turnover*, when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences. (b) Non-linear least squares sigmoidal best-fits of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant.

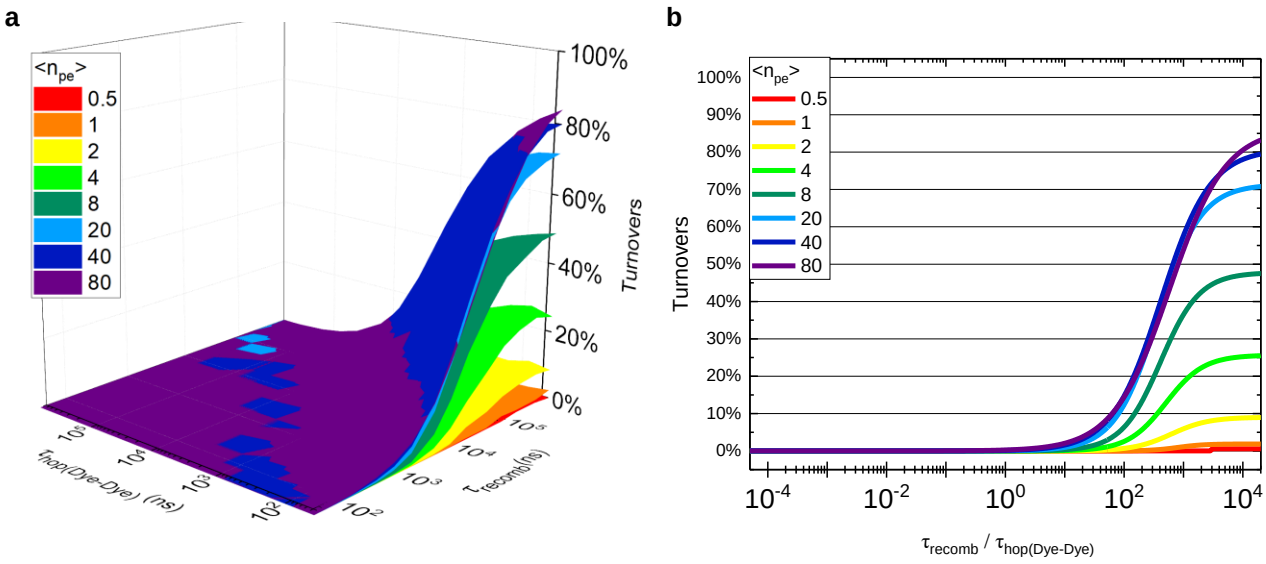


Figure 1.9. (a) Sheet plots representing the percentage of photoexcited dyes that ultimately *contribute to quadruple oxidation/reduction of an electrocatalyst and turnover* when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences. (b) Non-linear least squares sigmoidal best-fits of the data in panel a as a function of the ratio of the recombination time constant to the hopping time constant.

Effect of electrocatalyst valency.

While the model herein shows that it is more difficult to oxidize/reduce an electrocatalyst twice instead of just once, many reactions require even more than two redox events for electrocatalytic turnover. For example, oxidation of water to molecular dioxygen occurs via a four electron, four proton redox reaction, and in Nature's oxygen-evolving complex this net reaction is thought to occur via a single concerted O–O bond-forming step.[76] Because of the large interest in the oxygen evolution reaction, and other reactions requiring even more redox equivalents like molecular dinitrogen reduction to ammonia (6 electrons and 6 protons) and carbon dioxide reduction to methane (8 electrons and 8 protons), we performed simulations using electrocatalysts that are capable of accumulating 1, 2, or 4 charges prior to turnover, and did so at low ($<n_{pe}> = 1$), intermediate ($<n_{pe}> = 8$), and high ($<n_{pe}> = 80$) photon fluences. Figure 1.8 and Figure 1.9 show that in order to net oxidize/reduce an electrocatalyst four times, especially large fluences are required. However, this condition is not beneficial from a recombination perspective and therefore,

large values for τ_{recomb} / $\tau_{hop(Dye–Dye)}$ are needed to observe large values for percent turnover. These data follow the trends observed in Figure 1.3c where there is an optimal fluence that results in the largest percent turnover when τ_{recomb} / $\tau_{hop(Dye–Dye)}$ is large. Collectively, these data suggest that optimal fluence scales with the number of redox events required for turnover of an electrocatalyst.

Effect of electrocatalyst coverage.

Another parameter evaluated was the percent of positions occupied by electrocatalysts rather than dyes. In the base case, maximally twice oxidized/reduced electrocatalysts with a 1% coverage – or on average 2.52 electrocatalysts per particle – were used, and photoexcitation events were distributed according to a Beer–Lambert law distribution. In Figure 1.10, this condition is used to compare effects with other electrocatalyst coverages of 0.5%, 2%, and 4%, which correspond to on average 1.26, 5.04, and 10.08 electrocatalysts per particle, respectively. While 1% electrocatalyst coverage appears to be optimal when τ_{recomb} / $\tau_{hop(Dye–Dye)}$ is large, higher coverages are optimal as τ_{recomb} / $\tau_{hop(Dye–Dye)}$ decreases. With increasing coverage of electrocatalysts, accumulation of charges at electrocatalysts is more difficult, because the same number of oxidized/reduced dyes is diluted over a larger number of electrocatalyst sites. This limits percent turnover when τ_{recomb} / $\tau_{hop(Dye–Dye)}$ is large and therefore turnover is overall ineffective when there are too many electrocatalysts in the system. These simulation results are consistent with behavior that we observed previously via pulsed-laser spectroscopy experiments.[60] If τ_{recomb} / $\tau_{hop(Dye–Dye)}$ is small such that electrocatalyst turnover is poor, dilution of charges among electrocatalysts no longer limits the percent turnover and instead recombination is limiting. In these cases, having a larger coverage of electrocatalysts results in a larger percent

turnover because oxidizing/reducing an electrocatalyst occurs more frequently. This is clear from the data in Figure 1.10 where as τ_{recomb} / $\tau_{\text{hop(Dye–Dye)}}$ increases, the optimal coverage of electrocatalysts changes from 4% (green sheet) to 2% (light green sheet) and ultimately to 1% (yellow sheet). The condition of 1% coverage of electrocatalysts remains optimal under the fluences, electrocatalyst coverages, and values of τ_{recomb} / $\tau_{\text{hop(Dye–Dye)}}$ evaluated. However, the value of τ_{recomb} / $\tau_{\text{hop(Dye–Dye)}}$ where the optimal electrocatalyst coverage changes is dependent on the fluence. At low fluence, the size of the band in the sheet plot where the 1% electrocatalyst coverage condition is optimum is largest, while the transition of 4% to 1% electrocatalyst coverage being optimum occurs over the smallest region in the figure (Figure 1.10a). At high fluence, τ_{recomb} / $\tau_{\text{hop(Dye–Dye)}}$ must be near-optimal in order for 1% electrocatalyst coverage to be most effective at electrocatalyst turnover, and bands for both 2% and 4% electrocatalyst coverage are large (Figure 1.10d). This observation is extremely pertinent to dye-sensitized photoelectrochemical constructs, where most experimental demonstrations report that low coverages of electrocatalysts lead to the largest efficiencies for light-driven oxygen evolution through water oxidation.[77] Data from our simulations suggest that when the electrocatalyst coverage is relatively large (≥ 4%), optimal performance is observed at larger fluences and large values of τ_{recomb} / $\tau_{\text{hop(Dye–Dye)}}$. However, if lower fluences are used, our data suggest that a lower coverage of electrocatalysts is optimum.

A major challenge in using the results reported above to predict behaviors of dye-sensitized photoelectrochemical constructs is that most often efficiencies for light-driven oxygen

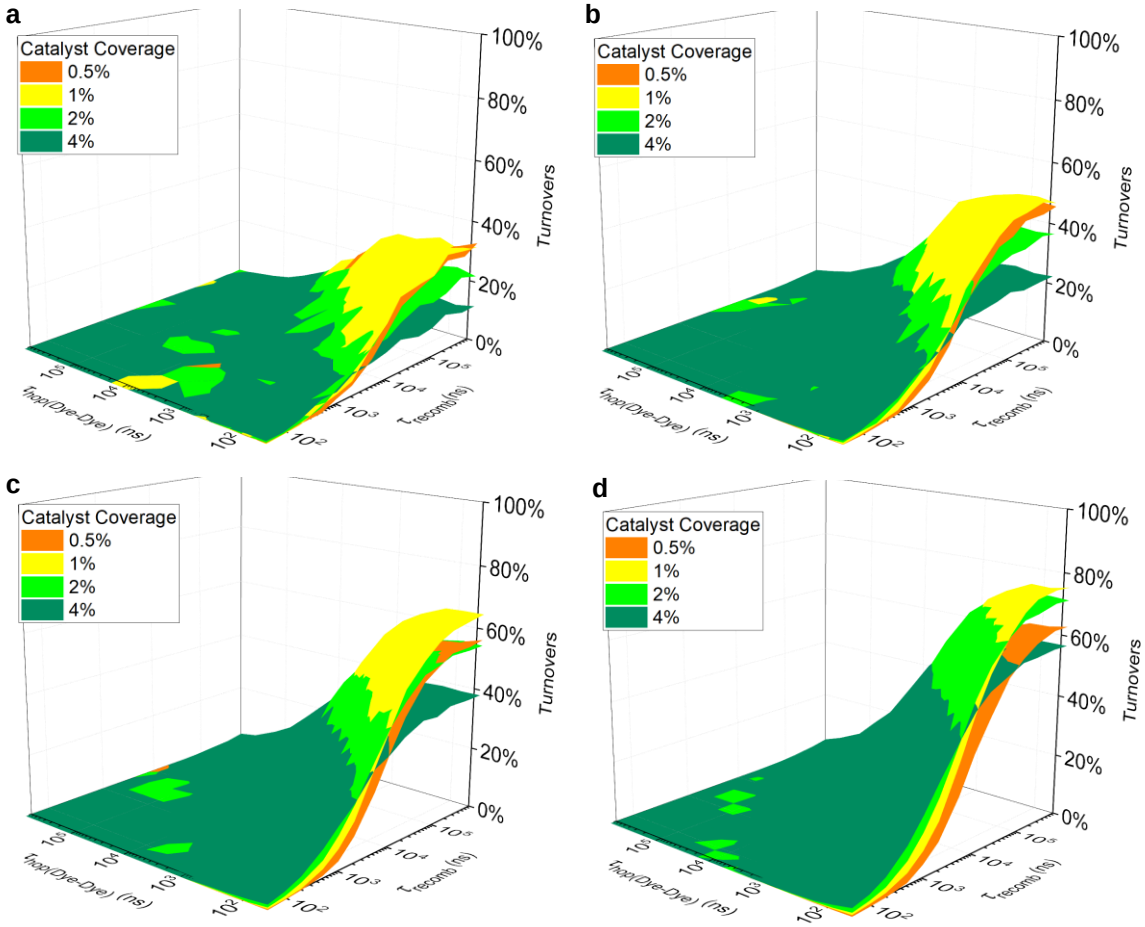


Figure 1.10. Sheet plots representing the percentage of photoexcited dyes that ultimately contribute to double oxidation/reduction of an electrocatalyst *when electrocatalyst are present at the indicated surface coverage* at the initial pulsed-light excitation fluence of (a) $\langle n_{pe} \rangle = 1$, (b) $\langle n_{pe} \rangle = 2$, (c) $\langle n_{pe} \rangle$

evolution through water oxidation are measured using conditions of continuous illumination and not initial pulsed-light excitation as simulated herein. This prompted us to quantify the percent electrocatalyst turnover during conditions of continuous illumination, which was mathematically implemented as a probability for light excitation at each step in the Monte Carlo simulation.

Effect of pulsed-light excitation versus continuous-wave illumination.

A major challenge in using the results reported above to predict behaviors of dye-sensitized photoelectrochemical constructs is that most often efficiencies for light-driven oxygen evolution through water oxidation are measured using conditions of continuous illumination and not initial pulsed-light excitation as simulated above. This prompted us to quantify the yield for electrocatalyst turnover during conditions of continuous illumination, which was mathematically implemented as a probability for light excitation at each step in the Monte Carlo simulation.

In order to realize efficiency gains in dye-sensitized photoelectrochemical constructs, detailed mechanisms and quantum yields for electron, charge, and energy transfer processes are necessary. Common techniques used to probe these processes include transient-absorption spectroscopy and time-resolved photoluminescence spectroscopy.[45] However, it is not known whether these pulsed-laser pump–probe techniques can replicate behaviors observed under practical conditions of continuous-wave illumination, which is the relevant condition for actual application of these photochemical materials systems. For this reason, we modeled the effects of repeated light excitation under conditions of solar-simulated illumination for a state-of-the-art dye-sensitized solar cell (~20 mA cm^{-2}) but under the caveat that surface-anchored electrocatalysts are present and that each requires one, two, or four redox events for turnover to mimic common conditions required for electrocatalytic reactions. For electrocatalysts requiring a single redox event for turnover, results from repeated light excitation at intensities of effectively 1 sun and 10 suns are in excellent agreement with results obtained using simulated initial pulsed light excitation at low fluences ($<n_{pe}>$ = 0.5–1.0 excitation) (Figure 1.11a). Data obtained for conditions of effectively 100 suns were very similar to those under lower light intensities, albeit with small

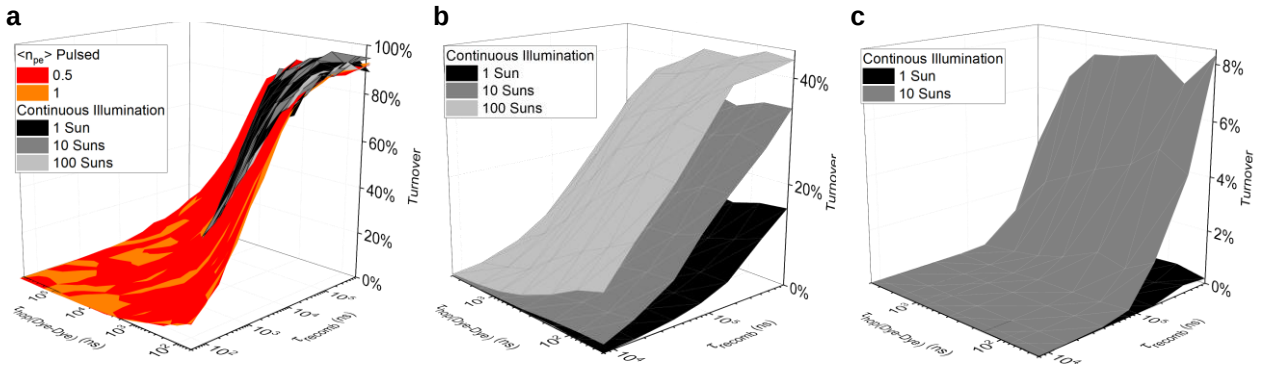


Figure 1.11. Sheet plots representing the percentage of photoexcited dyes that ultimately *contribute to (a) single, (b) double, or (c) quadruple oxidation/reduction of an electrocatalyst and turnover*, when electrocatalysts are present at 1% surface coverage at the indicated initial pulsed-light excitation fluences (colored sheets, taken from Figure 1.2) or continuous illumination solar-simulated fluences (grayscale sheets).

differences described in more detail below. When each electrocatalyst required two or more redox events for turnover, results over the same range of solar-simulated light intensities could not be reproduced by any condition utilizing initial pulsed-light excitation (Figure 1.11b and c). Also, it is clear from these data that turnover yields are no longer the same for each value of $\tau_{recomb}/\tau_{hop(Dye-Dye)}$, meaning that $\tau_{recomb}/\tau_{hop(Dye-Dye)}$ is not a reasonable single independent variable for these data and that all data in a single sheet can no longer be represented by a sigmoidal function in terms of $\tau_{recomb}/\tau_{hop(Dye-Dye)}$.The sensitivity of turnover yield to the light excitation condition depends on which time constant is varied. Starting at the optimal condition of small $\tau_{hop(Dye-Dye)}$ and large τ_{recomb}, turnover yield decreases substantially as the recombination time constant decreases; however, turnover yield is nearly constant as the hopping time constant increases. This suggests that the optimal condition is one where recombination is dictating the overall turnover yield, for electrocatalysts requiring two or four redox events for turnover (Figure 1.11b and c), but not for electrocatalysts requiring a single redox event for turnover (Figure 1.11a).

The plots shown in Figure 1.11are rich in information, but interpreting them when $\tau_{recomb}/\tau_{hop(Dye-Dye)}$ is not a good indicator of turnover yield is challenging. Therefore, we

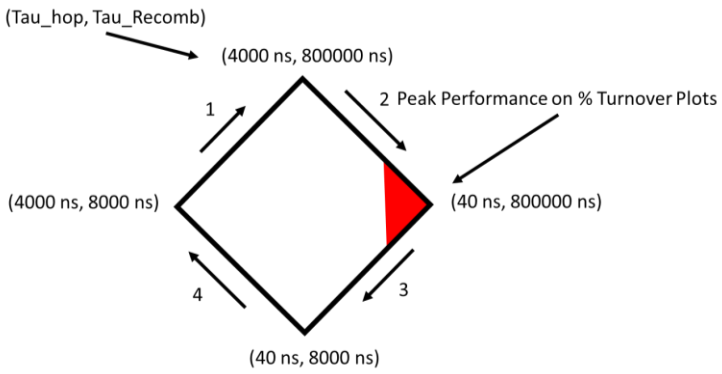


Figure 1.12. Schematic detailing the process used to create a panoramic plot by tracing the perimeter of the parameter space covered by the sheet plot as 1, 2, 3, and 4, to allow for facile two-dimensional viewing for a wide range of parameters.

decided to analyze the data under conditions where one time constant is fixed while the other time constant is varied. Because of the vastly different yields for electrocatalyst turnover under large and small time constant values, we decided that this analysis should be performed for multiple values of the fixed time constants, and therefore that the perimeter of the plots shown in Figure 1.11 would be most instructive and representative of the overall behavior. The resulting panoramic plots were constructed by starting at the condition where turnover yield is smallest, *i.e.* where $\tau_{hop(Dye-Dye)}$ is largest and τ_{recomb} is smallest, and reporting turnover yield as the time constants are stepped clockwise along the perimeter of the plots in Figure 1.11a and b. This protocol is shown schematically in Figure 1.12 and the resulting plots are shown in Figure 1.13a and b. As expected, the plots are nearly symmetric for the condition when electrocatalysts required a single redox event for turnover (Figure 1.13a), however the plots are clearly asymmetric for the condition when electrocatalysts required two redox events for turnover (Figure 1.13b). The causes of this asymmetry are due to the complex interplay of the competing kinetic processes. To understand which kinetic processes are rate-limiting for each set of time constants, it is useful to examine the steady-state number of oxidized/reduced molecules present on the surface of the nanoparticles as a function of the intensity of repeated light excitation (Figure 1.13c, d and 1.14). This is because the relationship between the number of charges present at steady-state, the photon

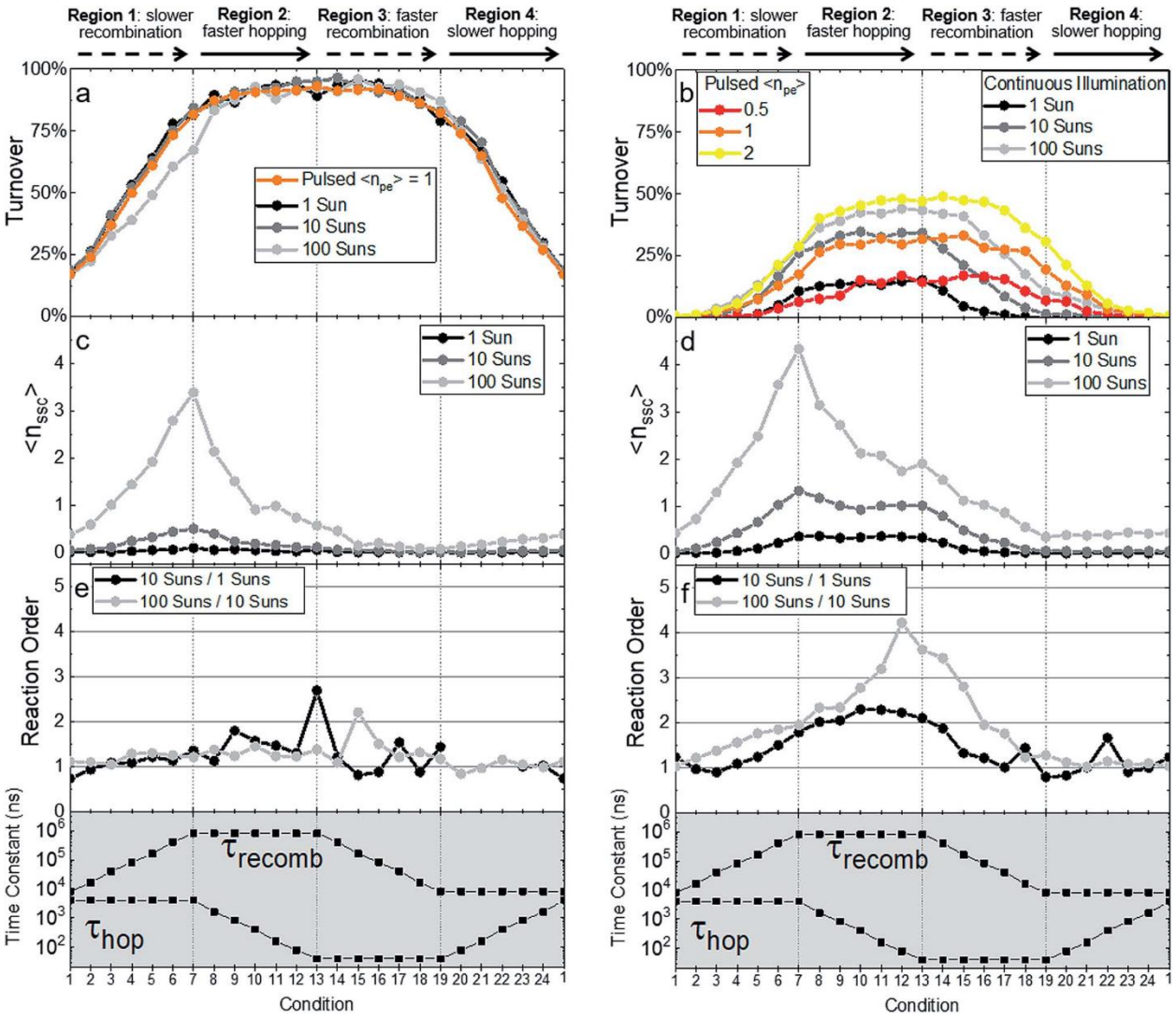


Figure 1.13 Panoramic plots tracing the perimeter of the parameter spaces covered by the sheet plots in (a) Figure 1.11a and (b) Figure 1.11b, with the greyed-out regions indicating the independent variables for all panels and the labels for regions 1, 2, 3, and 4 as descriptors for all panels. As references, panels (a) and (b) also contain data from the indicated initial pulsed-light excitation simulations (colored data, taken from Figure 1.2). Panoramic plots for the conditions in panels (a) and (b) showing the average number of molecular charges per particle at steady-state ($<n_{ssc}>$) as (c and d) raw data and (e and f) normalized to the data obtained using a 10-fold-lower photon fluence and converted into perceived reaction order in $<n_{ssc}>$.

fluence rate, and the charge loss mechanisms is well known based on detailed balance and Kirchhoff's current law. It follows that under steady-state conditions, per particle, the rate of generation of charges due to photon absorption ($G = I_{light}$) equals the rate of loss of charges due to recombination and electrocatalytic turnover (R), which by mass action has the following kinetic rate law,

$$R = k_1(n_{ssc})^{\nu_1} + k_2(n_{ssc})^{\nu_2} \quad (5)$$

where k_i represents the rate constants for the rate-limiting reactions, n_{ssc} is the steady-state number of charges on the nanoparticle, and vi represents the order of the reactions in n_{ssc}. Assuming that only one process with $\nu_i \neq 0$ dominates the loss term, R, the following log–log relation and derivative hold at steady-state (where *G = R*),

$$log_{10}[n_{ssc}] = \frac{1}{\nu} log_{10}[I_{light}] - \frac{1}{\nu} log_{10}[k], \quad (6a)$$

and so

$$\frac{d(log_{10}[n_{ssc}])}{d(log_{10}[I_{light}])} = \frac{1}{\nu} \quad (6b)$$

Further analysis of the equation for the slope (eqn (6b)) under the assumption that the rate-limiting loss mechanism does not change reveals that when the light intensity is increased by an order of magnitude, such that d $\log_{10}[I_{light}] = 1$, the following relations hold,

$$d(log_{10}[n_{ssc}]) = \frac{1}{\nu} = log_{10}[n_{ssc\_high}] - log_{10}[n_{ssc\_low}], \quad (7a)$$

and so

$$\nu = \frac{1}{log\left(\frac{n_{ssc\_high}}{n_{ssc\_low}}\right)} \quad (7b)$$

where high and low stand for the relative conditions of high and low light intensity. Using eqn (7b) and comparing the ratio of the steady-state number of charges per particle at several light excitation intensities (Figure 1.13e and f), one can glean the apparent order of the rate-limiting reaction for loss of charges and therefore, gain information as to the process that limits the yield for electrocatalyst turnover. Starting with the data in Figure 1.13c and d, these plots are clearly asymmetric, irrespective of whether the trends in electrocatalyst turnover yield are nearly symmetric (Figure 1.13a) or asymmetric (Figure 1.13b). This suggests that the number of charges present at steady-state is not the only indicator of the

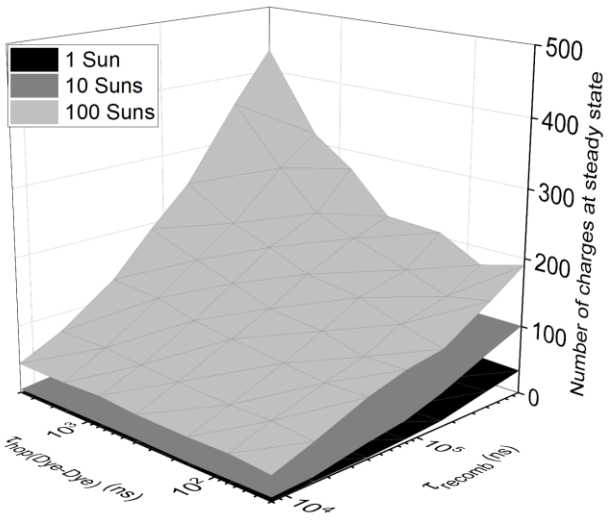


Figure 1.14. (a) Sheet plots – oriented like all other sheet plots – representing the steady-state number of oxidized/reduced species when electrocatalysts require double oxidation/reduction for turnover and are present at 1% surface coverage at the indicated continuous illumination solar-simulated fluences.

asymmetry in the trends for turnover yield. The number of steady state charges reaches a maximum value when both $\tau_{hop(Dye–Dye)}$ and τ_{recomb} are at their maximum value (Figure 1.13c and d, boundary 1|2), which is not the optimal condition for turnover yield. Irrespective, this condition makes sense because a large value for τ_{recomb} means that recombination is slow and a large value for $\tau_{hop(Dye–Dye)}$ means that a long time is required for an oxidized/reduced dye to encounter an electrocatalyst so that the charge can then be lost due to turnover. Under this condition, turnover yield is nearly the same as under the optimal condition where instead $\tau_{hop(Dye–Dye)}$ is small (Figure 1.13a and b). This means that when recombination is slow, hopping does not limit turnover yield, which is a conclusion that is consistent with the analysis of the data in Figure 1.11a and b.

Before analyzing the trends in the apparent orders of the rate-limiting reactions shown in Figure 1.13e and f, it is useful to understand how each reaction order is manifested in the data shown in Figure 1.13e and f and what reaction order is expected for each rate-limiting reaction. In order to determine the reaction order in the number of charges per particle for each kinetic process that results in loss of charge, simulations were performed using initial homogeneous pulsed-light excitation in the presence of only one kinetic process for loss of charge. Results from these hypothetical scenarios when n_{pe} is the same for each particle are shown in Figure 1.15. The rate of recombination was found to exhibit a second-

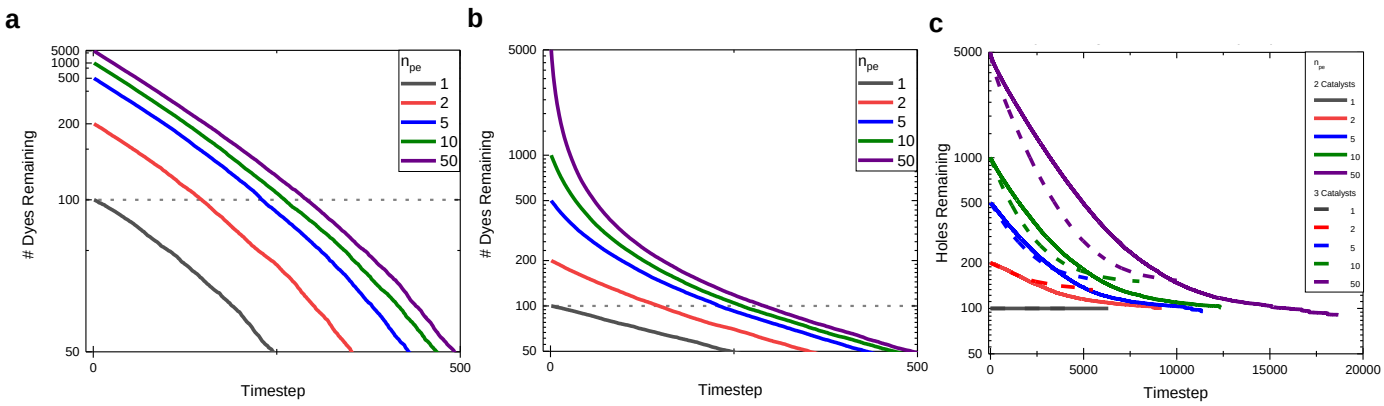


Figure 1.15. (a,b) Number of oxidized/reduced dyes remaining over time after the indicated initial uniform pulsed-light excitation fluences, in the absence of electrocatalysts. (c) Number of oxidized/reduced species remaining over time after the indicated initial uniform pulsed-light excitation fluences at the indicated uniform number of electrocatalysts per particle, in the absence of recombination. The y-axis in panel a is reciprocally scaled so that linear behavior indicates equal-concentration 2nd-order kinetic processes, while the y-axes in panels b and c are logarithmically scaled so that linear behavior indicates 1st-order kinetic processes. Kinetic parameters from best-fits of these data are shown in Table 1.2.

order dependence on $\langle n_{ssc} \rangle$ due to recombination having a first-order dependence on the number of oxidized/reduced molecules and a first-order dependence on the equal number of charges in the semiconductor nanoparticle (Figure 1.15a). However, when photon fluence was low such that $\langle n_{ssc} \rangle \leq 1$, a single recombination event per particle removed all of its charge carriers meaning that each particle only had a binary state of having zero or one charge-separated states and therefore the ensemble average behavior over all particles was in fact first-order in $\langle n_{ssc} \rangle$ (Figure 1.15b). Interestingly, the observed rate of electrocatalyst turnover was determined to be approximately first-order in the number of charges per particle over the majority of the time that oxidized/reduced dyes were present (Figure 1.15c), irrespective of the number of electrocatalysts per particle.

The data in Figure 1.13e suggest that under all light excitation intensities studied and irrespective of the values of $\tau_{hop(Dye-Dye)}$ and τ_{recomb}, there is a substantial first-order contribution from $\langle n_{ssc} \rangle$ to the loss of charges. At boundary 4|1, first-order behavior is expected because $\langle n_{ssc} \rangle < 1$ (Figure 1.13c) and ~80% of the molecular charges are lost due

to recombination (Figure 1.13a), which is manifest as a first-order dependence on the number of charges per particle, while the remaining charges contribute to electrocatalyst turnover, which is also first-order in the number of charges per particle. At boundaries 1|2, 2|3, and 3|4, first-order behavior is expected because >80% of the molecular charges contribute to electrocatalyst turnover (Figure 1.13a). However, notably, as observed at boundary 1|2, some second-order behavior is expected from the remaining molecular charges that are lost due to recombination (<20%). This is because at high fluence $\langle n_{ssc} \rangle > 1$ (Figure 1.13c) and even at low fluence some particles likely have $n_{scc} > 1$. It is challenging to draw additional conclusions from these data due to the near independence of the observed behavior on the time constants or light excitation conditions, and poor signal-to-noise for the lowest excitation condition. However, this is not the case for electrocatalysts that require two redox events for turnover (Figure 1.13f).

Table 1.2. Best-fit rate constants from the linear regions of the data in Figure 1.15.

	Recombination, *# excitations remaining* > 100 (Figure 1.15a)	**Recombination, *# excitations remaining < 100* (Figure 1.15b)**	**Turnover, *Initial (2 electrocatalysts per particle)* (Figure 1.15c)**	**Turnover, *initial (3 electrocatalysts per particle)* (Figure 1.15c)**
kinetics	**equal-concentration 2nd-order**	**1st-order**	**1st-order**	**1st-order**
n_{pe} **= 1**	–	1.23×10^{-3} timestep^{-1}	0 timestep^{-1}	0 timestep^{-1}
n_{pe} **= 2**	3.32×10^{-5} timestep^{-1}	1.43×10^{-3} timestep^{-1}	5.46×10^{-5} timestep^{-1}	5.92×10^{-5} timestep^{-1}
n_{pe} **= 5**	3.15×10^{-5} timestep^{-1}	1.34×10^{-3} timestep^{-1}	1.52×10^{-4} timestep^{-1}	1.86×10^{-4} timestep^{-1}
n_{pe} **= 10**	3.17×10^{-5} timestep^{-1}	1.38×10^{-3} timestep^{-1}	2.01×10^{-4} timestep^{-1}	2.80×10^{-4} timestep^{-1}
n_{pe} **= 50**	3.11×10^{-5} timestep^{-1}	1.47×10^{-3} timestep^{-1}	2.44×10^{-4} timestep^{-1}	3.55×10^{-4} timestep^{-1}
mean	319 ($\pm 9 \times 10^{-7}$) timestep^{-1}	137 ($\pm 9 \times 10^{-5}$) timestep^{-1}	–	–

For electrocatalysts that require two redox events for turnover, all processes that result in the loss of molecular charges require two oxidized/reduced dyes. However, this does not mean that charge loss will be second-order in the number of charges per particle, because rates of electrocatalyst turnover and rates of recombination when $<n_{ssc}> < 1$ exhibit first-order dependencies on the number of charges (Figure 1.15b and c). At boundary 4|1 and as expected, first-order behavior is dominant because at all light excitation intensities $<n_{ssc}> < 1$ (Figure 1.13d) and ~100% of the molecular charges are lost due to recombination (Figure 1.13b). At boundary 1|2, significant second-order behavior is expected because most molecular charges are lost due to recombination (>70%) (Figure 1.13b) and at high fluence $<n_{ssc}> > 1$ (Figure 1.13d) and even at low fluence some particles likely have $n_{ssc} > 1$. This same behavior occurs at boundary 2|3, although at high fluence, turnover yield is larger (Figure 1.13b) and so there is a more significant contribution from the first-order behavior of electrocatalyst turnover. However, under this condition the apparent order of n_{ssc} in the rate-limiting reaction for loss of charges is much larger than two, suggesting that turnover is larger than second-order in the number of charges per particle or that the observed rate constant for turnover increases at higher light excitation intensities. The data in Figure 1.15c suggest that for the electrocatalyst coverages and fluences used here, the rate constant increases considerably, which therefore explains the even larger apparent reaction order observed as a global maximum near boundary 2|3 (Figure 1.13f). Lastly, first-order behavior again dominates at boundaries 3|4 and 4|1 (Figure 1.13f), which is expected because turnover yield decreases to <10% in these regions (Figure 1.13b).

The consequences of the behavior and limiting mechanisms described above are important in that for the case of electrocatalysts that each requires one redox event for

turnover, the four boundaries at 4|1, 1|2, 2|3, and 3|4 show drastically different $<n_{ssc}>$ as a function of both illumination intensity and time constants, yet a nearly constant first-order contribution from n_{ssc} to the loss of charges, suggesting that the fluence dependence of the rate-limiting reaction for loss of charges is responsible for the symmetric trends in turnover yield observed for the data shown in Figure 1.13a. This also helps to explain the minor asymmetry in turnover yield observed at boundary 1|2 under 100 suns of repeated light excitation where $<n_{ssc}> > 1$ and turnover yield is only ~60%, meaning that second-order recombination occurs for ~40% of the oxidized/reduced molecules. This rationale also suggests that the more pronounced asymmetric trends in turnover yield observed in Figure 1.13b are due to the order of the photon fluence on the rate-limiting reaction for loss of charges. For these data, the approximate order of n_{ssc} in the rate-limiting reaction for loss of charges ranges from one (boundary 4|1), to two (boundary 1|2), to one but with variable observed rate constant for electrocatalytic turnover (boundary 2|3), and again to nearly one (boundary 3|4). The anomalous asymmetry in turnover yield is most apparent by comparing data near boundaries 1|2 and 3|4, which are regions that are symmetric in turnover yield at low fluence for electrocatalysts that require a single redox event for turnover (Figure 1.13a). The asymmetry in turnover yield for electrocatalysts that require two redox events for turnover is due to differences in the order of n_{ssc} in the rate-limiting reaction for loss of charges, which near boundary 1|2 is approximately two due to the equal-concentration second-order nature of the reaction, while near boundary 3|4 is approximately one due to ensemble effects.

The implications of these results are very important for dye-sensitized photoelectrochemical cells and related solar fuel constructs. It is clear that a range of

observed kinetic dependencies will exist for the various processes that are operative in dye-sensitized photoelectrochemical materials under constant solar-simulated illumination at 1–100 suns. This means that fitting data to simple kinetic models and analyzing trends in the resulting kinetic parameters will be greatly convoluted by whether each semiconductor nanoparticle has greater than or less than one charge at steady-state. In reality, this behavior is even more complex than reported herein because our models assumed that charges on oxidized/reduced dyes could not transport to other semiconductor particles, that all particles were identical in size, and that there was no distribution in the electronic states in the semiconductor or in the molecular states such that the kinetics could be described by straightforward traditional kinetic rate laws based on the law of mass action. Collectively, these data suggest that kinetic behaviors observed in dye-sensitized photoelectrochemical cells may not be due to heterogeneous environments or non-ideal kinetic processes, but rather the complex interplay of limiting regimes in chemical catalysis that are pertinent to these constructs. Data from these simulations also suggest that, experimentally, kinetics observed using pulsed-laser spectroscopies may represent a convolution of several

traditional kinetic equations even if a single underlying kinetic phenomenon is operative. This underscores an even broader conclusion from this study, which is the observation that for electrocatalysts that required multiple redox events for turnover, the conditions of initial pulsed-light excitation could not reproduce the behavior observed based on simulations that mimic the conditions of continuous illumination. Thus, fundamental time constants for kinetic processes must be obtained using any pulsed-laser fluence but then based on the values obtained, a specific pulsed-laser fluence must be used in order to predict the performance of the materials system under real-world sunlight illumination. This is

unfortunate because it requires a larger degree of experimental specificity and interpretation in order to perform meaningful experiments on these materials systems. For materials systems whose electrocatalysts only require that they are oxidized/ reduced once for turnover, a specific pulsed-laser fluence consistent with exciting approximately one dye per particle should mimic the performance under conditions of continuous illumination, assuming that the underlying material geometry, molecular arrangements, and mechanistic kinetic processes used in the models presented herein are accurate for the systems under study. These conclusions are consistent with experimental observations and analyses previously reported in the literature, which are conflicting on the mechanisms, kinetics processes, and even order of reactions in charges that are operative in dye-sensitized photoelectrochemical constructs[45,78,79] and therefore, this remains a very active area of research.

Conclusions

This work developed and reported a new and advanced model for charge transport across dye-sensitized materials that are most pertinent to photoelectrochemical cells for solar fuels constructs. Results from the model indicate the largest yields for electrocatalyst turnover occur when the ratio τ_{recomb} / $\tau_{\text{hop(Dye–Dye)}}$ is large and that while higher fluences result in larger absolute rates of electrocatalyst turnover, the yields decrease for electrocatalysts that require two oxidations/reductions for turnover. In general the model results also suggest that yield for turnover of these electrocatalysts was largest when the

total absorbance of the sample was low or scattering particles are introduced to randomize excitation over the thickness of the nanoparticle stack. Results also suggest that having 1% coverage of electrocatalysts, which equates to ~2.5 electrocatalysts per particle, maximizes the yield for turnover of electrocatalysts for the geometry and parameters considered in the model. The models also show that simulated continuous illumination can be attained through repeated light excitation and in this case observed kinetic behavior can be first-order or second-order in the number of charges per particle, or some linear combination of these processes. Under simulated 1 Sun excitation conditions incorporating dyes used in state-of-the-art dye-sensitized solar cells, on average less than one oxidized/reduced dye was present per particle at steady-state and the purely second-order kinetic processes for recombination resulted in ensemble first-order kinetic behavior due to the binary redox state of each nanoparticle. This suggests that for effective dye-sensitized photoelectrosynthetic cells for solar fuels production, a low coverage of electrocatalysts is best and depending on the illumination intensity and electron-transfer time constants, yields for electrocatalyst turnover can be quite high under solar simulated conditions.

CHAPTER 2. Geometric Considerations

Introduction

Systems similar to dye-sensitized solar cells have been the focus of numerous previous studies including our own as detailed in Chapter 1 of this thesis. These studies aim to better understand and optimize the experimental conditions that yield the best-performing devices. While our previous study focused on exploring different features of our developed model over a simple surface consisting of isolated spherical semiconductor particle supports, it lacked a key tie to real-world experiments in that photoexcitation events and molecules were isolated to a single particle within the larger surface. This meant that electron-holes on one particle would only be able to reach catalysts on that same particle, resulting in overall more restrictive percolation zones than real-world mesoporous thin films. Interparticle hopping is not entirely a new concept for implementation in these types of models but, as was the case previously, we believe the additional functionality that our model has over its predecessors offers a good opportunity to better understand these systems.

Experimental

Modeling framework

A model has been created that aims to simulate the structure of thin films mesoporous TiO_2 which is ostensibly made up of nearly identical spherical nanoparticles. Experimentally, this film is then coated by dye and catalyst molecules which is simulated by creating a distribution of molecular surface sites and assigning each to be a dye, catalyst, or nothing. Some dye molecules on this simulated surface are randomly assigned to be sites of photoexcitation. The simulation proceeds by stepping forward by a time step of fixed size

and allowing each electron-hole on the surface to hop to an adjacent molecule through self-exchange reactions, recombine with an electron injected into the bulk of the TiO_2, or else remain stationary. The probabilities for each of these events to happen is predetermined during simulation initialization based on input time constant values for hopping and recombination. The parameter of interest is, in general, the yield for electron-holes created by photoexcitation events ultimately resulting in complete oxidation of catalysts leading to catalyst turnover. Primarily, catalysts in this study require 2 electron-holes to oxidize completely. The underlying framework is for the most part the same as the previous study except for changes as described below.

Changes from our previous study

τ_{hop}	100000	50000	20000	10000	5000	2000	1000	500	200	100	50	20	10	5	2	1
τ_{recomb}	1000	1000000														
τ_{ratio1}	1E-2	2E-2	5E-2	1E-1	2E-1	5E-1	1E0	2E0	5E0	1E1	2E1	5E1	1E2	2E2	5E2	1E3
τ_{ratio2}	1E1	2E1	5E1	1E2	2E2	5E2	1E3	2E3	5E3	1E4	2E4	5E4	1E5	2E5	5E5	1E6

Table 2.1. Values of τ_{recomb} and τ_{recomb} used in these studies and the resulting τ_{ratio} values.

One takeaway from our previous report detailing results from Monte Carlo simulations of dye and cocatalyst modified spherical nanoparticles was that, in the case of models that simulated pulsed-light excitation, the time constant used to determine the probability of self-exchange electron transfer hopping between adjacent dyes and the time constant used to determine the probability of recombination between charges in the semiconductor and oxidized/reduced dyes were each not the true independent variable but rather that the ratio of those two time constants was the independent variable. Therefore, in

order to better structure this next study and save time on performing redundant and unnecessary simulations, the independent variables consisted of a one-dimensional array of hopping time constants each paired with only one of two different recombination time constants. This is in contrast to our previous study where the array of independent variables consisted of values resulting from a large two-dimensional matrix of hopping and recombination time constants resulting in a three-dimensional sheet plot. By using fewer combinations, a larger range of time constant ratios could be simulated with minimal experimental redundancy in order to speed up overall computation time. Some overlap in the time constant ratios was intentionally implemented as a precaution in case our previous assumption that the time constant ratio was the most appropriate independent variable proved to be invalid. In this way, 25 distinct time constant ratios were simulated in this work with 7 time constant ratios being in both sets of time constant ratios, as shown in Table 2.1.

Another important difference between our prior study and that reported herein is the process by which molecular sites are assigned. Instead of using Mathematica's built-in tessellation function to create a fixed geometric pattern of positions, a method based on Fibonacci spirals was used to evenly distribute points over the surface of a sphere. The benefits of this are two-fold. The greater benefit is that the spiral method scales to any number of points. Tessellation always resulted in the same pattern containing a very specific number of total points while the spiral method allows any number of points to be utilized. This allows for differing point densities on each identically-sized particle or the same point density on particles of different sizes. The other benefit of implementing Fibonacci spirals to determine positions on the spherical particles is that molecular positions on each particle are no longer all relatively the same. In the case of tessellation, each relative position on a

given particle is always the same while Fibonacci spirals generate differing relative positions per particle. Irrespective, in both cases all positions on each particle are rotated azimuthally and longitudinally based on the center of the particle to create a random relative orientation among particles.

The model utilized for the simulations herein also includes other differences versus our prior implementation of the model: (1) The height (*z*-coordinate) of a dye molecule is used to determine the probability of excitation according to the Beer–Lambert law generation profile, instead of the height of the particle to which the dye is anchored like in our prior study. This was a necessary change to accommodate non-linear particle geometries. (2) Each molecule is considered adjacent to all molecules within a fixed distance from its center, which results in a variable number of adjacent molecular positions, instead of fixing the number of adjacent molecular positions to 5 or 6 like in our prior study. For example, this means that molecules near necking regions between two particles have potentially more neighbors than molecules far from necking regions. Overall, this results in there being 6 – 8 adjacent molecular positions depending on the local geometry. (3) Photoexcitations that occur on particles that contain zero electrocatalysts or occur in too few number to result in electrocatalyst turnover are identified at the beginning of the simulation and the simulation is terminated once only electron/holes from these photoexcitations remain, instead of removing these photoexcitations from the outset like in our prior study. (4) Instead of assuming that hopping can occur between all molecules on a single particle like in our prior study, percolation zones are identified for each molecular position because in many cases particles touch and therefore adjacent molecules may be located on other particles. A percolation zone consists of the molecular positions that a charge can hop among

from a given oxidized/reduced molecule. In our prior study each particle was a separate percolation zone, because the particles did not touch, and all positions were occupied by a molecular dye or molecular electrocatalyst. For the more complex geometries modeled herein, many percolation zones do not contain all molecular positions on a particle but often contain molecular positions on several adjacent particles.

Surfaces

The major focus of this study is to model and simulate several surface geometries and other related surface conditions, instead of a surface that is made up of a stack of separate particles that are not in physical contact with each other like in our prior study and thus only influence each other in some cases through competitive light absorption. The surfaces modeled herein are defined in Figure 2.1 and in most cases are in physical contact, thus allowing molecules on different particles that are in close spatial proximity to be considered

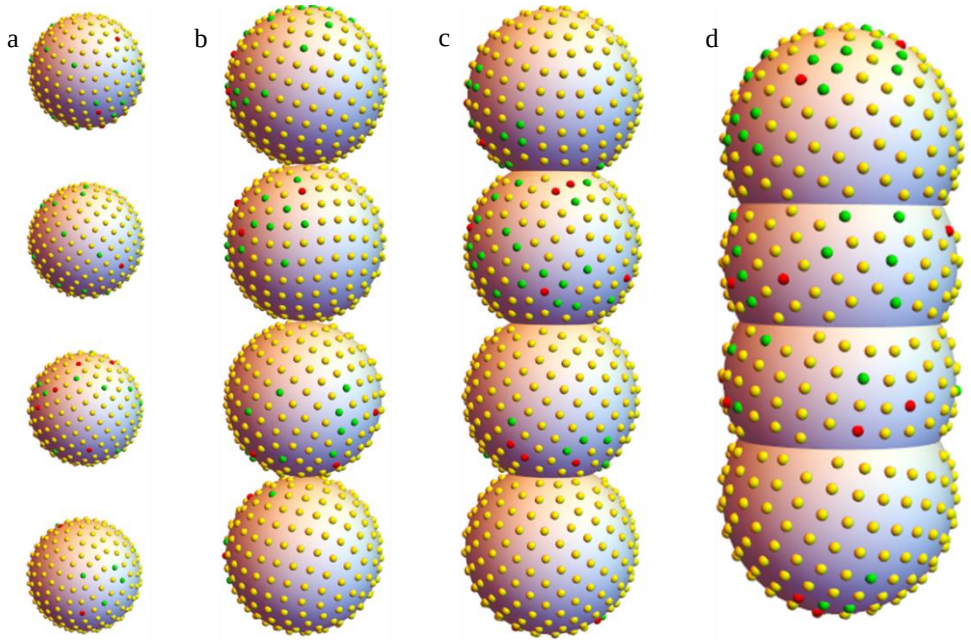


Figure 2.1. Different example modeling surfaces used in this study named as follows. (a) Separate. (b) Touching. (c) Necked. (d) Tube. 4 example particles shown for each geometry above where experimental conditions use stacks of 100 particles. Red and green spheres represent catalysts and photoexcitation positions respectively.

adjacent and resulting in inter-particle charge transport. This is very important, because it represents a more physical scenario encountered experimentally when studying typical mesoporous thin films utilized in dye-sensitized photoelectrochemical constructs. While the modeled particles change spatial configuration significantly, a stack of 100 particles is used like in our prior study. However, unique to the work herein is that some geometries have more particle overlap and therefore fewer possible molecular positions per particle. This means that in a stack of 100 particles, there is in total a different number of molecular positions on the entire surface. To account for this difference a fixed percentage of the total number of dye molecules is excited, rather than a fixed number of dye molecules per stack like in our prior study.

Results and discussion

Effect of surface geometry and photon fluence

The primary focus of this study is to compare the turnover yield of several geometric possibilities when considering the modeled surface. The most straightforward comparison is to test the same conditions over all four surfaces shown in Figure 2.1. The results of this are shown in Figure 2.2. Each subplot shows a different fluence condition simulated and all four possible surfaces as indicated. These results show that, as far as turnover yield is concerned, for optimal time constants, the Touching, Necked, and Tube surfaces all perform the same while the Separated particle surface is ~10% less effective. This is due to the fact that on a separated surface, it is much more likely to excite dyes on percolation zones, which in this case are particles, where there are no catalysts and so those excitations will never contribute to catalyst turnover. On the other surfaces, excitations may occur far from catalysts, but because all molecular positions contain a molecule, the entire surface is one

large percolation zone and thus there is always a possible pathway to eventually reach a catalyst. For this reason, the connected surfaces outperform the separate particles only when the time constant ratio is large and therefore electron-holes are able to hop many times between adjacent molecules before recombining. An unexpected but interesting result is that it does not seem to matter how connected a surface is in terms of necking such that surfaces with only one or two molecules that bridge charge transport between neighboring particles, as in the Touching structure, is just as effective as having many molecules bridge interparticle charge transport, as in the Tube structure.

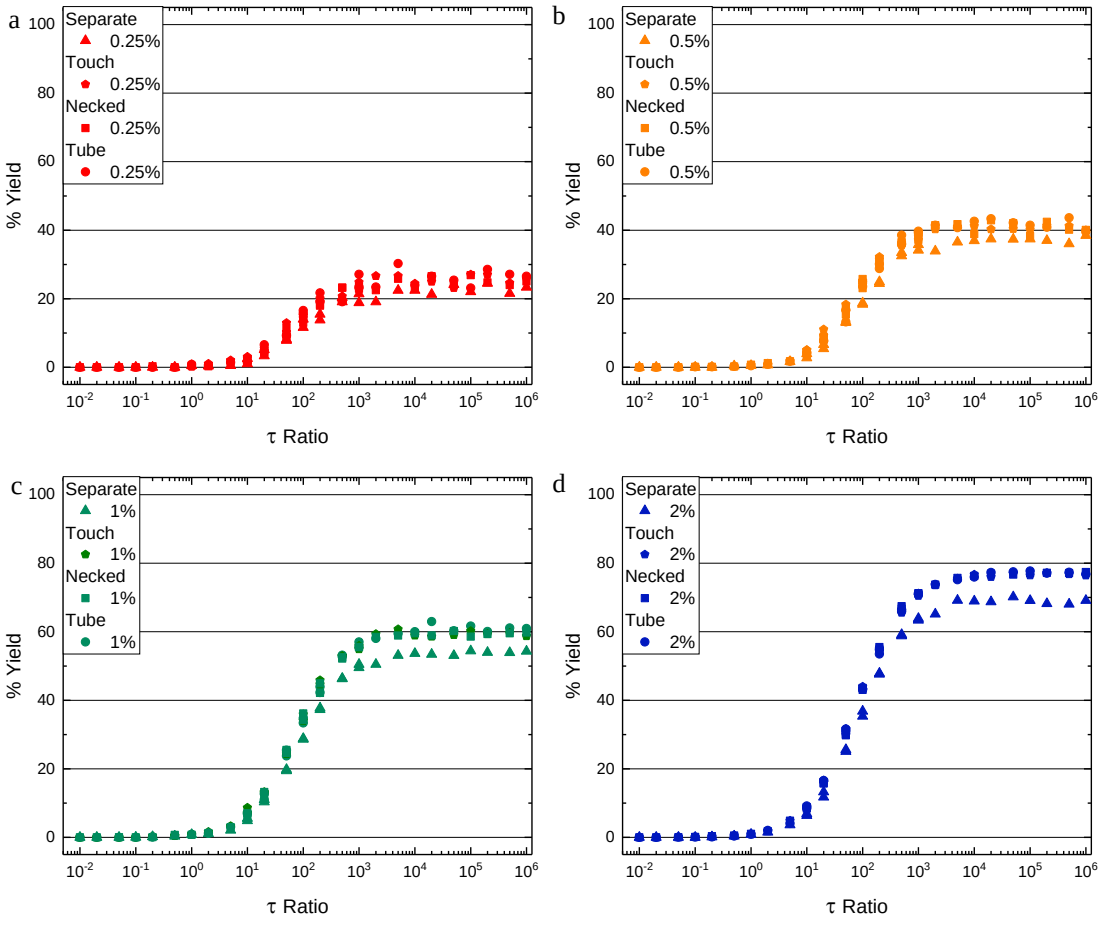


Figure 2.2. Plots showing yield for electrocatalyst turnover as a function of time constant ratio for each of four surfaces and under four different fluence conditions represented as the percentage of dyes on the surface that are initially photoexcited: (a) 0.25%, (b) 0.5%, (c) 1.0 %, (d) 2.0%.

The results from the direct surface comparison prompted the question of "How far do electron-holes travel if there are in a connected system?" A modification to the model was made which recorded the entire hopping path of an electron hole through its lifetime and a tally was made of the number of unique particles was that was visited during that lifetime. This was averaged over all excitations in a given experiment and the results are shown in Figure 2.3. One of the most striking takeaways is that the number of particles visited plateaus using conditions that are far smaller than the maximum time-constant ratio. The time constant ratio can essentially be thought of the number of random-walk hops an individual electron-hole can expect to take before recombining and so one would initially expect that the larger the ratio, the further the total distance traveled. However, the simulated systems have catalysts in them and an electron-hole will stop travelling once it reaches one. While the ratio may indicate that a much further distance is possible, these results show us that a catalyst is typically found far closer than the maximum possible travel distance. When viewing these results, it is also important to remember that the particles used in these surfaces are all initially the same size *prior to necking* and so the "particles" in the Tube surface are effectively much smaller in that there are fewer molecules to hop between before reaching the next particle.

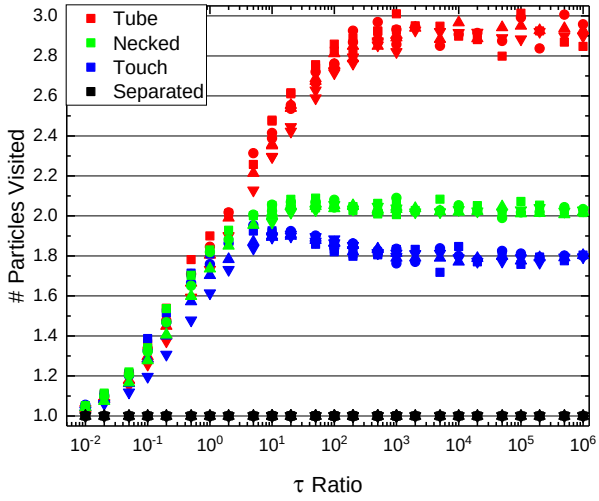


Figure 2.3. The average number of particles visited by an individual electron-hole during its lifetime averaged over 25 repetitions of 4 different fluences.

While it seems like electron-holes on Tube surfaces travel much further than those on Touching surfaces, the distances are actually the same, because the particles are just different sizes.

Maximum expectations

Initially, it seemed that the results from Figure 2.2a were higher than expected, which prompted us to question what the maximum turnover yield is given no limitation on hopping distance. For example, on the Separate surface, in order to contribute to turnover yield, electron-holes had to populate catalysts in a 2:1 ratio while at the same time the initial dye-to-catalyst ratio was 1:4. This combined with the fact that some electron-holes would inevitably recombine and others would be photogenerared on particles containing zero catalysts, made a >20% yield seem high.

A simpler model was created to simulate the sorting of electron-holes onto catalysts with no kinetic or recombination considerations. This model first took a specified number of catalysts and sorted them onto individual particles on the surface. It then took a specified number of photoexcitations and sorted them onto the individual particles on the surface which resulted in electron-holes on those particles. Then, for each particle, if there were both catalysts and electron-holes, holes were randomly sorted amongst the catalysts on their particle. Finally, the number of pairs of electron-holes on catalysts were tallied and counted toward turnover yield. This model identified the maximum possible yields, given only statistical constraints and not kinetic ones. For example, given a particle with 2 catalysts and 2 electron-holes, the ideal result would be that both electron-holes oxidize/reduce the same

catalyst and result in a single turnover event. However, it is just as likely in this example that each electron-hole oxidizes/reduces a different catalyst and zero turnover events occur. For this example situation the maximum expected yield is instead only 50% under the ideal circumstances. Determination of the maximum yield was performed on systems of 100 individual particles, 250 catalysts, and between 0 and 2000 photoexcitations. This was repeated for catalysts requiring 1, 2, 3, or 4 electron-holes to turnover and each datapoint taken was the average of 1000 such simulations. These results are shown in Figure 2.4.

Our previous concerns that, on a Separate surface with 1% catalyst coverage and photoexcitation of 0.25% of dye molecules, a result of >20% yield seemed high were apparently incorrect as this statistical model shows that such a system exhibits a maximum turnover yield of ~22%. The maximum values for fluence conditions of interest are indicated in Figure 2.4(b).

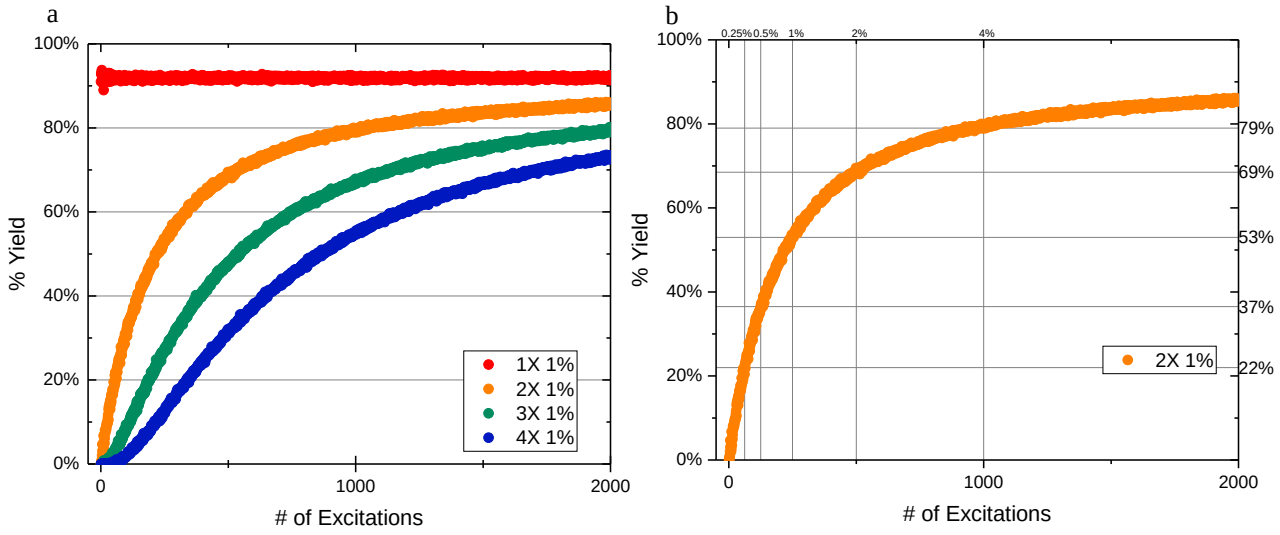


Figure 2.4. Plots representing the average percentage of photoexcited dyes that ultimately contribute to electrocatalyst turnover… based on sorting with no recombination considerations for a Separate particle surface. (a) Showing expected yield for 0-2000 excitations and for systems using catalysts requiring 1-4 electron-holes for turnover as indicated. (b) Showing only systems in which catalysts which require 2 electron-holes are needed with y-values indicated at relevant fluences.

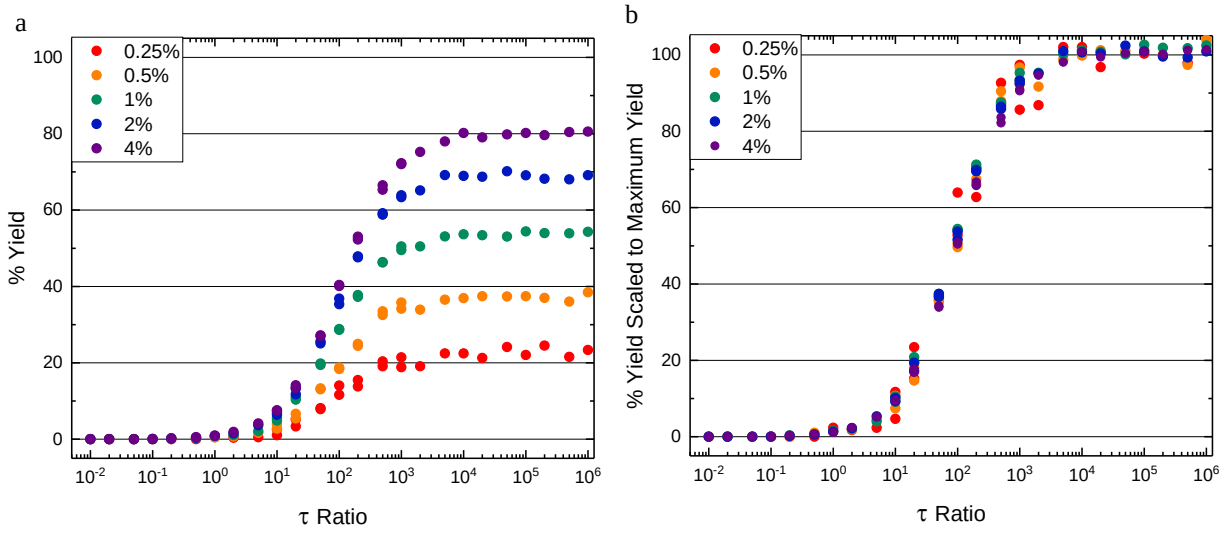


Figure 2.5. Plots showing the yield for catalyst turnover as a function of time constant ratio at fluences as indicated on a Separate particle surface. (a) Unscaled yield. (b) Yield scaled by pertinent maximum expected yield by fluence as indicated in Figure 2.4b.

Considering data for the Separate surface, each fluence condition resulted in nearly the same shaped plot but each plateaued at a different value as shown in Figure 2.5a. These values were then scaled by the maximum possible turnover yield to determine how each fluence condition performed relative to its maximum performance. When this scaling was done, as seen in Figure 2.5b, it was observed that the performance of each fluence condition was the same as all other fluence conditions. Two interesting conclusions result from this observation. The first of these is that only the time constant ratio determines how effectively an experiment performs relative to its maximum potential. That is not to say that all fluence conditions perform equally but rather that different fluence conditions do not have different requirements for the time constant ratio on their performance. The second conclusion is that maximum performance is reached well before the maximum simulated time constant ratio and that a time-constant ratio of 10^4 is sufficient for maximum performance of the Separate surface regardless of fluence.

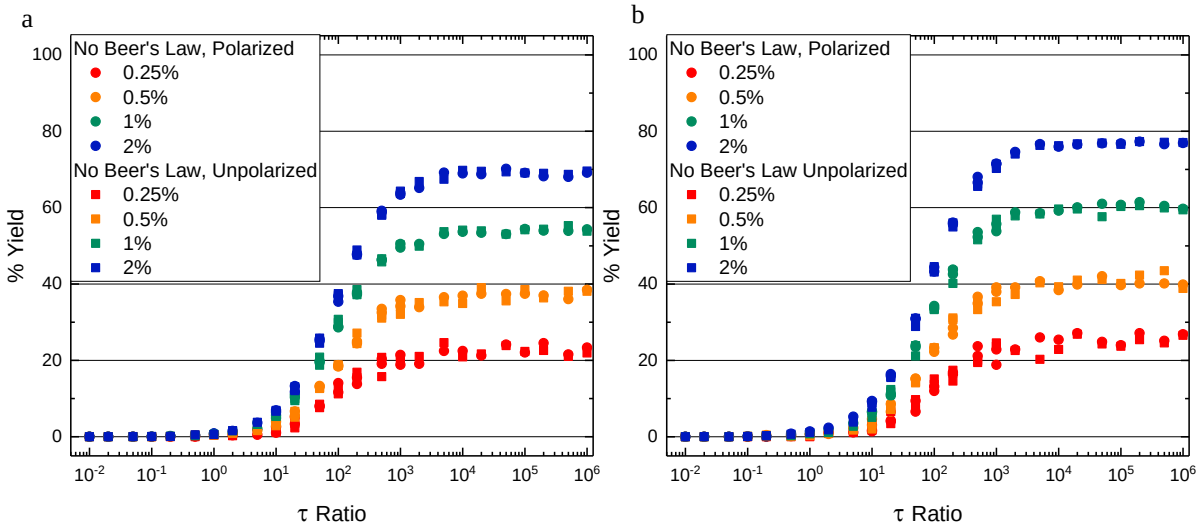


Figure 2.6. Plots showing yield for catalyst turnover comparing conditions where polarized light is used in the initial excitation or is not used for fluences as indicated and on (a) Separated particles. (b) A Tube.

Effect of photon polarization

Another major consideration was whether using polarized light excitation would have a significant impact on turnover yield. In this model, the use of polarized light was implemented by weighting the likelihood of photoexcitation for a molecule oriented along the axis of polarization higher than molecules oriented in another direction. A discussion of this weighting was made in our prior study. This weighting option was not examined in out prior study but could potentially be relevant with more complex geometries, such as those used here. Figures 2.6a and 2.6b show the results of this study on Separate particle and on a Tube respectively. The overall conclusion is that that polarized light has no effect on the turnover yield regardless of what Surface used or what fluence condition used.

Effect of vacant molecular positions

Another variable that seemed pertinent to examine in a geometric study was total molecular coverage shown in Figure 2.7. Experimentally, dying a film is an imperfect process and we hoped to examine the results when less than total molecular coverage is attained. As discussed above, a percolation network is important as electron-holes resulting from photoexcitation must be able to hop to catalysts. Here we examine how turnover yield is affected by the size of the percolation networks. The initial conditions of this study were the similar to previous studies with 1% of molecular sites being selected as catalyst sites and 1% of the remaining sites (dye sites) being selected as initially photoexcited. Only the 1% excitation fluence condition was used in this case. However, unlike previous studies a certain percentage of potential molecular positions were chosen to be vacant *before* catalyst selection. For example, in a Separate surface with 25,000 total molecular positions, 50% vacant spot coverage, 1% catalyst coverage, and 1% dye photoexcitation, 12,500 sites are randomly selected to be empty spots, 125 of the remaining sites are chosen to be catalyst sites, and 124 sites (rounded from 123.75) are chosen to be initially photoexcited dyes. The results of this study One of the first conclusions is that the percolation network does not begin to effect the turnover yield until nearly 50% of the molecular positions are vacant and that even at that coverage, electron-holes are still able to hop to catalysts if given enough time prior to recombining. This indicates that even with 50% coverage of vacant spots, a percolation network still exists but that it may not provide a direct path to hop to a catalyst and therefore the number of hops to reach a catalyst may be larger than when no molecular positions are vacant. Along with this, it seems that the percolation network is insufficient for significant catalyst turnover at 75% coverage of vacant spots and that between 50% and

75% coverage of vacant spots the percolation zones become significantly smaller and more discrete. Interestingly, when comparing the breakdown of the percolation network between the three connected surfaces, the Tube Surface performs the worst of the three followed by the Necked surface. This suggests that although the Tube surface seems like it should be the most robust in terms of percolation network, this is not the case and that perhaps this more complex percolation network is reliant on a larger fraction of its total molecular sites meaning that the removal of any sites is more likely to disrupt it.

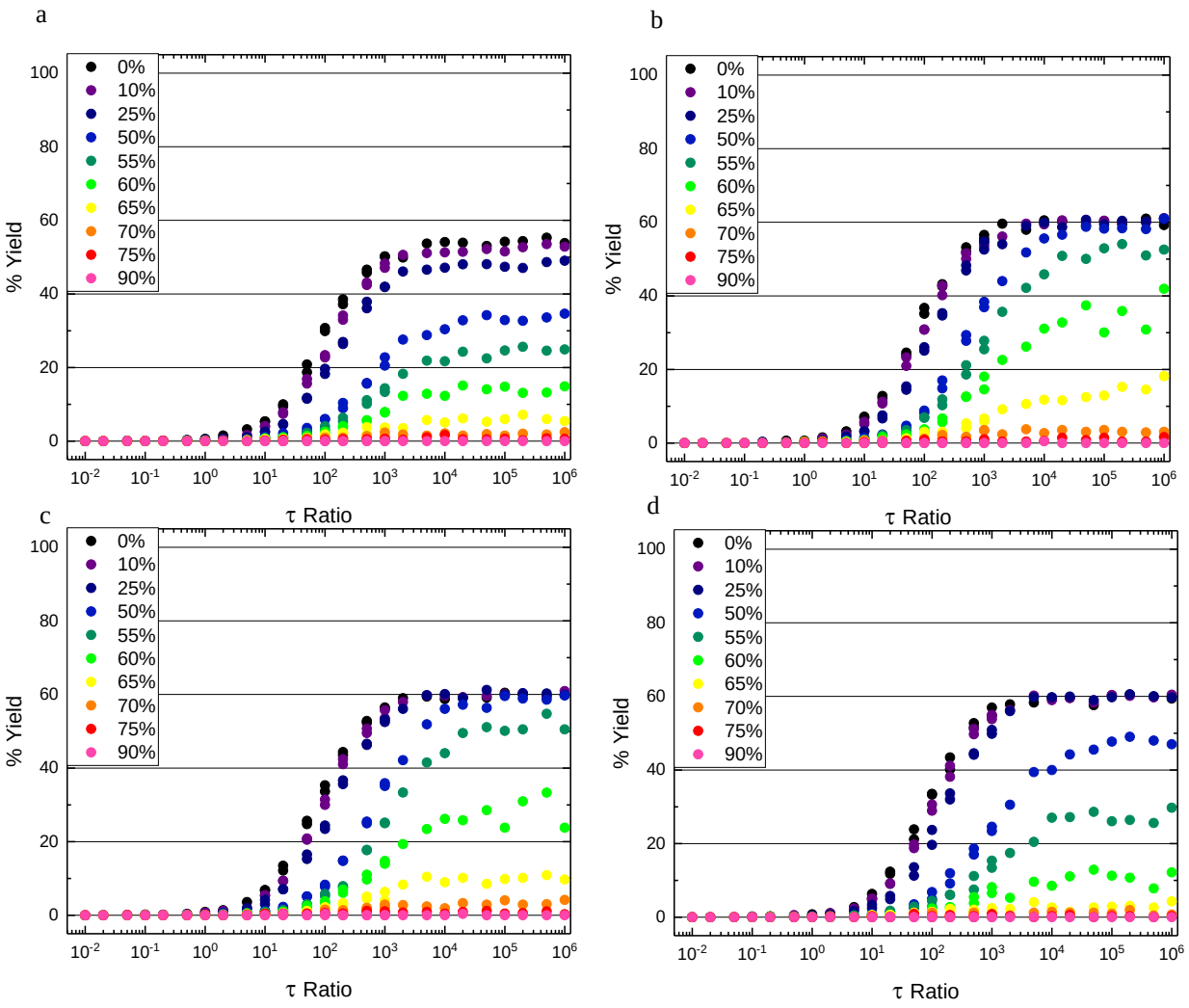


Figure 2.7. Plots showing the yield for catalyst turnover as a function of time constant ratio with a specified percentage of potential molecular positions left unoccupied as indicated on (a) Separate particles (b) a Touching Surface (c) a Necked Surface (d) a Tube Surface.

A second conclusion that can be drawn from these results is that the Separated system is more greatly affected by coverage of vacant spots as seen in Figure 2.7a. Even at only 10% coverage the turnover yield at high values of the time constant ratio has started to decrease noticeably and at no simulated time constant ratio do simulations reach the performance of those with 0% coverage of vacant spots. This may be due to the fact that a Separated particle surface is already made up of a number of smaller percolation zones and so is more susceptible to further disconnections within those zones to form multiple smaller percolation zones. It is more likely for an electron-hole to remain isolated from catalysts if it can only hop across a single particle. On the other hand, an electron-hole that is isolated from catalysts on its own starting particle may still be able to reach a catalyst if it can hop to another particle in any of the connected surfaces.

Effect of simplifying the surface geometry

Many of the conclusions of these geometric studies have thus far been that there are variables that do not significantly impact the yield for catalyst turnover in our system. This led us to question if the added complexity of our three-dimensional model over other models results in significantly different outcomes, besides time-resolved anisotropy information, or if a simpler two-dimensional geometry would be equivalent. To this end, a two-dimensional square model was created that can be simulated similarly to the three-dimensional models, but with several important differences. A depiction of this modeled surface is shown in Figure 2.8. One important difference is that every molecule in the model is located on one large square and therefore geometric considerations such as polarization and the Beer-Lambert Law are not relevant. This square consists of 158 molecular positions per

dimension, which results in a surface with nearly the same number of molecular sites as the Separated surface (158^2 = 24,964 as opposed to 250*100 = 25,000). Another important difference is that periodic boundary conditions are utilized so that opposite edges (right/left and top/bottom) are adjacent and therefore allow hopping to "wrap around" from one side to another. The final major difference between the three-previous dimensional models and this two-dimensional model is that election density is distributed homogenously. This is done in an attempt to effectively capture the equal-concentration second-order recombination behavior based on the concentration of injected electrons/holes in the semiconductor support while treating all molecular sites as connected. However, simply treating this surface as one large particle results in an extremely large recombination rate as all injected electrons are in that same particle. To correct for this, the relative electron density is scaled down by a factor equal to the dimension of the surface, 158 in this case. This

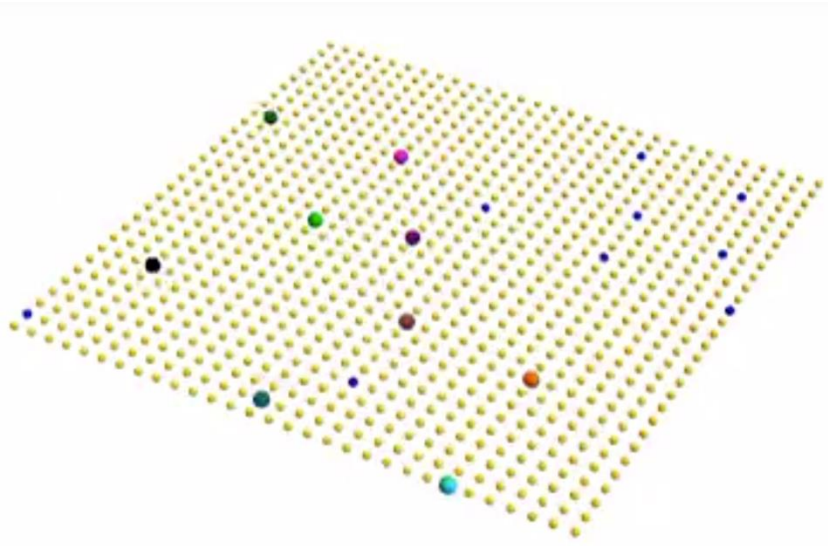


Figure 2.8. Depiction of a Square surface consisting of a 30 x 30 grid of molecules, where small blue dots represent catalysts molecules, multicolored dots represent photoexcitations, and the remainder of the molecular positions represent ground-state dye molecules. Actual simulations were performed using periodic boundary conditions, to allow for hopping between opposite edges of the square surface, and a 158 x 158 grid but that becomes difficult to visualize as a figure.

can be thought of as 158 particles each with 158 molecular positions and with the electrons distributed evenly across all particles. We think this is a reasonable comparison to the Separated surface containing 100 particles each with 250 molecular positions with slight confinement of surface charges to subsets of dyes in smaller percolation zones.

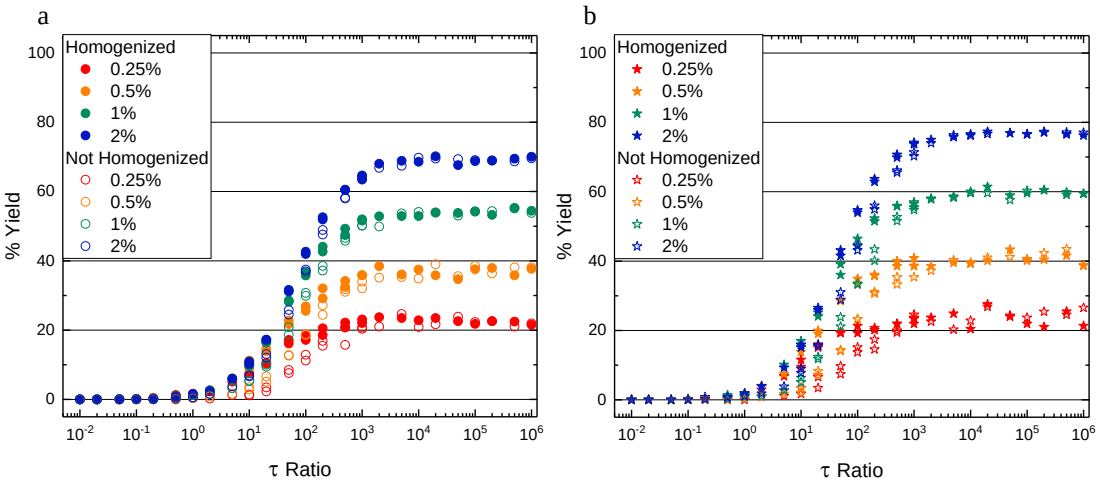


Figure 2.9. Plots comparing experiments in which electron density is homogenized across all positions vs those in which electron density remains fixed to particles which have electron-holes on them. (a) On a Separated surface. (b) On a Tube surface.

Before trying to compare these results to our previous results, it seemed prudent to first compare our previous results with those same surfaces while spreading all electron density homogenously over the entire surface. This was done on both a Separated surface and on a Tube surface and the results are shown in Figure 2.9. The primary difference between the homogenized and non-homogenized surfaces are in the mid-range of time constant ratios where turnover yields initially become non-negligible, where homogenized surfaces are more effective at turnover. This makes sense as this homogenization is a net shift of electron density away from electron-holes and so should strictly increase performance under conditions where the performance is recombination rate limited.

The other consideration that must be made when implementing the two-dimensional square surface is how to assign adjacent molecular positions. The two clear options in this case are to allow hopping only to strictly the 4 nearest neighbors or to also allow diagonal hopping and thus 8 adjacent molecular positions. Molecules on the three-dimensional surfaces typically had 6 adjacent molecular positions but varied between 4 and 8 depending on connectivity. By evaluating both two-dimensional neighboring options of 4 and 8 (Figure 2.10), the range of neighbors used for three-dimensional surfaces was spanned. The main

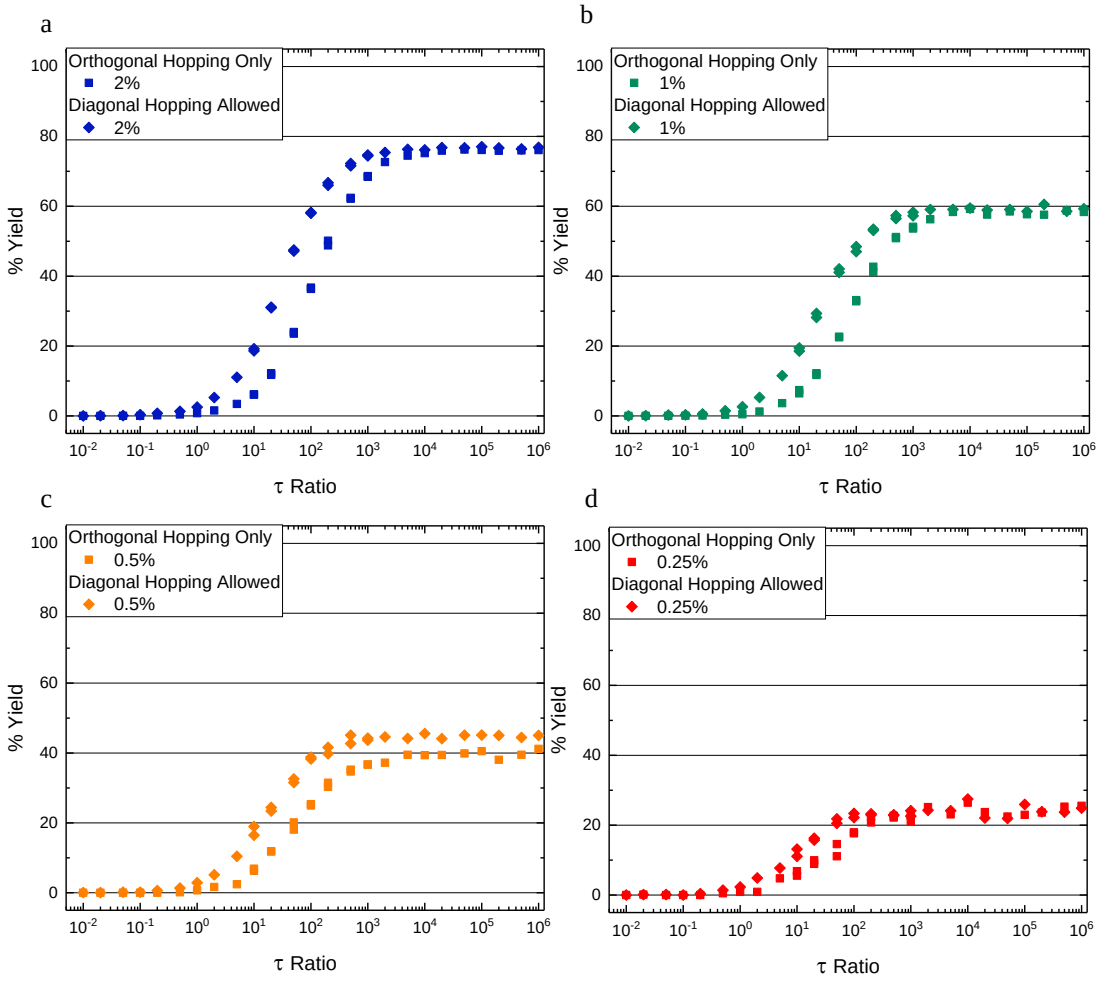


Figure 2.10. Plots showing experiments on a 158 x 158 molecule Square surface while allowing hopping only in orthogonal directions or allowing hopping to diagonally adjacent neighbors as indicated. Different fluence conditions separated into different plots for reading clarity. (a) exciting 2% of dye molecules. (b) exciting 1% of dye molecules. (c) exciting 0.5% of dye molecules. (d) exciting 0.25% of dye molecules.

impact that the number of adjacent molecular positions had in the two-dimensional model was on the minimum time constant ratio value that resulted in non-negligible turnover yield. This is reasonable because having the option to hop diagonally allows a hole to more quickly explore its surroundings and become adjacent to a catalyst at which point it very likely end up oxidizing that catalyst. The result is a range of turnover yields for each time constant ratio

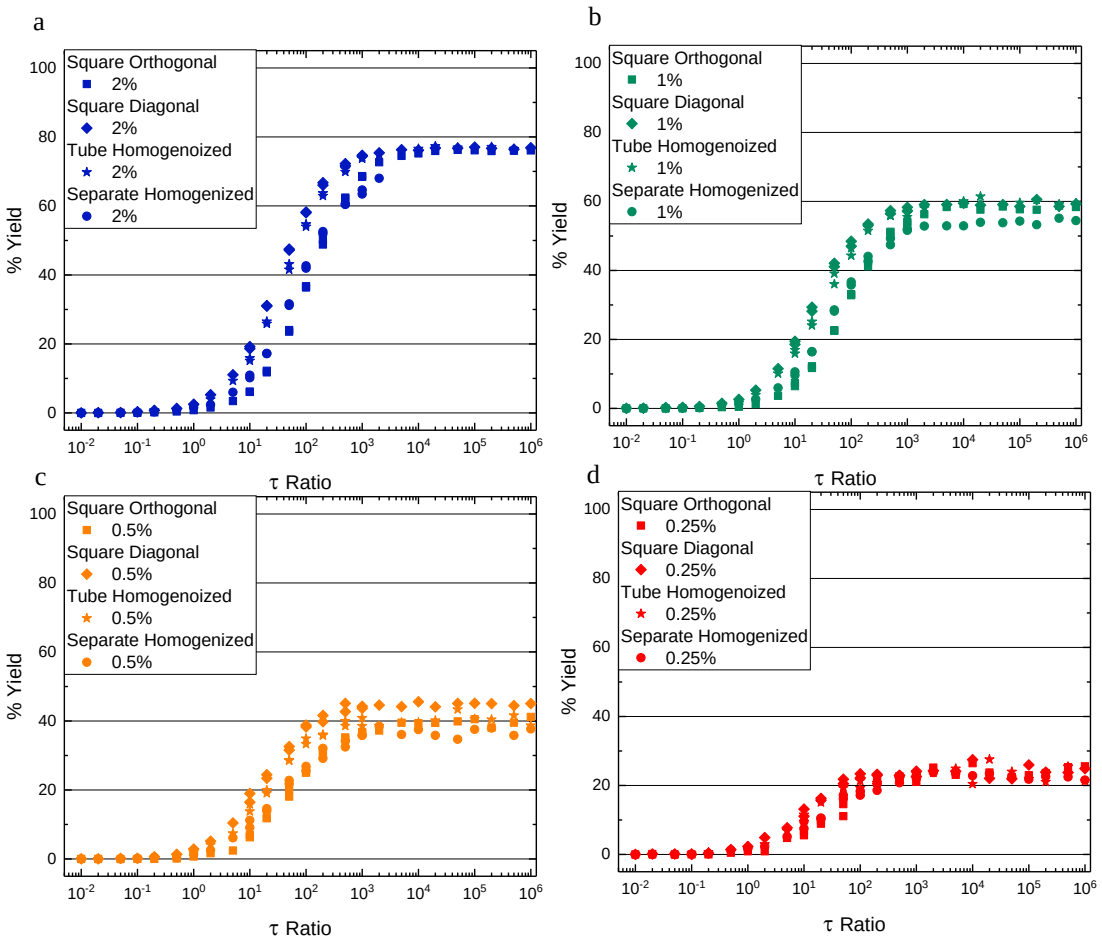


Figure 2.11. Plots comparing experiments on a 158 x 158 molecule Square surface with Separated and Tube surfaces that have had electron density homogenized. Different fluence conditions separated into different plots for reading clarity. (a) exciting 2% of dye molecules. (b) exciting 1% of dye molecules. (c) exciting 0.5% of dye molecules. (d) exciting 0.25% of dye molecules.

that span the conditions of 4 to 8 adjacent molecular positions for conditions of non-negligible turnover yield and non-near-optimal turnover yield.

With these considerations of electron density homogenization and 4 versus 8 nearest neighbors we analyze the differences in turnover yield for a two-dimensional Square surface in comparison to a three-dimensional Separated surface and Tube surface (Figure 2.11). These results are not terribly surprising in that the two variations of the two-dimensional square surface do bracket the results of the previously studied three-dimensional surfaces after they have been homogenized. A two-dimensional square surface that only allows for hopping to 4 adjacent molecular positions results in a smaller turnover yield than a homogenized three-dimensional Separated surface while a two-dimensional square surface that allows for hopping to 8 adjacent molecular positions results in a larger turnover yield than a homogenized three-dimensional Tube surface. Despite the fact that results from the two-dimensional square surface do not perfectly match those of either of these previously studied three-dimensional surfaces, it is clear that they are able to capture the overall behavior in turnover yield as a function of time constant ratio including peak performance yields at high values of the time constant ratio.

As a follow up to this study, a hexagonally packed surface was created which allows hopping to 6 nearest neighbors rather than to 4 or 8 nearest neighbors. This surface and a comparison to a homogenized Tube Surface are shown in Figure 2.12. This comparison shows that this hexagonally packed surface tracks the behavior of the Tube Surface well at all time constant ratios and that there are no significant deviations. From this we can conclude that, in terms of turnover yield, the topological information that needs to be accurately represented in the model includes only the number of nearest neighbors and the

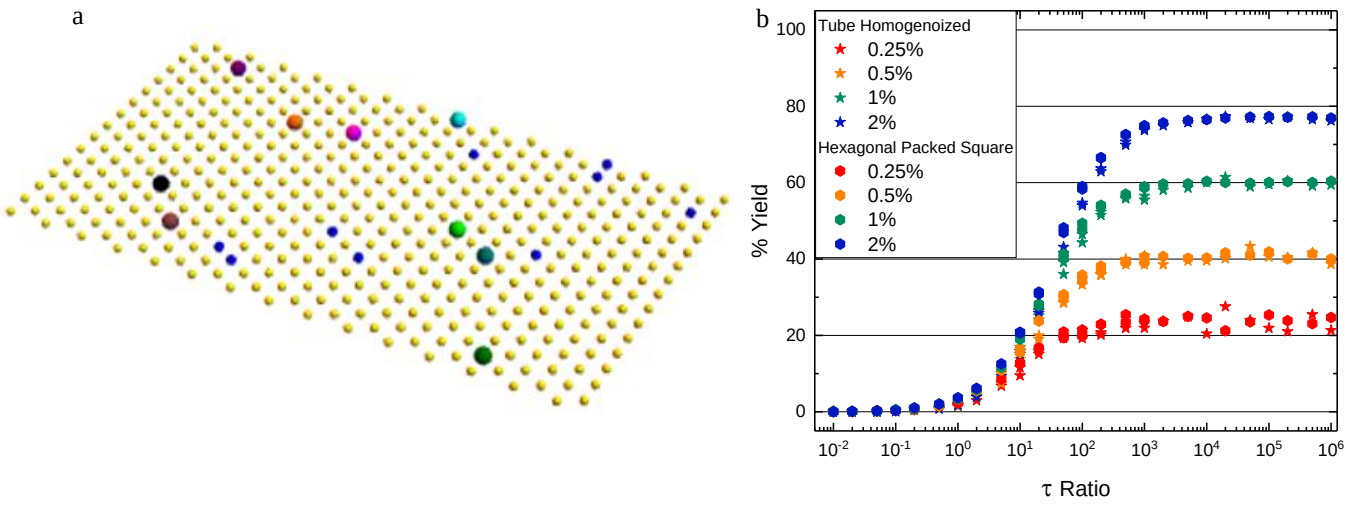


Figure 2.12. (a) Depiction of a planar surface which has been hexagonally packed with molecules such that each molecule has 6 nearest neighbors. (b) A comparison between results obtained from the surface shown in part (a) with those using a homogenized Tube surface as shown in Figure 2.9b and 2.11.

overall connectivity of the system. In this way, it is important to accurately represent both the short- and long-range order of the system being modeled but not necessary to specify the relative arrangement of these components.

Conclusions

This work continued the development of an advanced model for charge transport across dye-sensitized materials. Simulations show that, while using a more complex geometry which more closely resembles real experimental conditions, there is minimal difference in turnover yield after pulsed-light excitation conditions between three-dimensional surfaces with different types of interparticle connections. That is, yields are affected by the forming of connections between particles to allow interparticle hopping but changing the degree to which particle surfaces overlap does not have any significant effect. Results from simulations also indicate that the use of polarized light has little effect on

turnover yield. If fact, in comparison to a fully planar two-dimensional model incorporating periodic boundary conditions, simulations involving none of the geometric information or complexity of other models capture most of the behavior of those more complex models including maximum turnover yield performance. Finally, results show that for a Separated surface of individual particles, relative performance to maximum performance is independent of fluence used in the simulation.

CHAPTER 3. Outreach Development

Even though our planet contains large reservoirs of water as oceans, access to drinking water is a global concern, because large amounts of salt are fatal to humans. While electrochemical desalination, i.e. electrodialysis, is a process that is capable of generating potable water from saltwater, most chemistry curricula do not teach this process. Therefore, we developed a curriculum and accompanying low-cost activity to expose students from middle school to undergraduate studies to the concept of electrodialysis and the importance of polymeric ion-exchange membranes in the electrodialysis process. The curriculum provides background by introducing the students to issues of water access, current state-of-the-art solutions and the scenarios where each are optimal, and the urgent need for alternative and innovative processes for clean, potable water generation. The concepts and techniques presented in this curriculum cover those relevant to desalination, which encompass several physical phenomena that span multiple disciplines. The supporting activity that accompanies this curriculum allows students to perform electrodialysis and monitor the progress of the reaction using pH-sensitive dyes, which inherently includes many concepts that are relevant to general chemistry. The scientific depth of this curriculum is easily adjusted to challenge students at various levels of expertise.

Introduction

Scarcity of clean, potable water is a problem of immediate and enormous concern. It affects people in both developing nations and developed nations and is the root of numerous violent conflicts.[1,2] The United Nations has projected that in less than 15 years, nearly half of the global population will live in areas of water stress.[3] Moreover, most of these people will live in developing nations. In the United States, people in Southern California are

experiencing a prolonged drought, which affects the entire nation because California is the major supplier of domestic produce.[4] While water is relatively abundant on Earth, > 96% of it contains salt at concentrations that are unhealthy to humans and plants. Therefore, technologies to desalinate salt water and convert it to potable water are hugely important. In developed nations, with infrastructure to support an electric grid and the capital and labor to construct nearly billion-dollar desalination plants, reverse osmosis (RO) is the state-of-the-art commercial means used to desalinate ocean water.[5–7] The RO process occurs through pressurization of a container of salt water, which forces water molecules through a semipermeable membrane that excludes most solutes, including salt ions, resulting in less salty water on the other side of the membrane (Figure 3.1). Desalination by RO occurs when the external applied *pressure* opposes and exceeds the natural osmotic pressure between the salt water and the desalinated water on the other side of the membrane. Electrodialysis (ED) is a technique that is comparable to RO in terms of energy requirement.[8–12] Desalination by ED occurs when an external potential is applied between two chambers containing water of differing salinity such that the *electric potential* opposes and exceeds the natural chemical potential difference between the charged species in the solutions. This forces salt ions through a series of permselective polymeric ion-exchange membranes that each

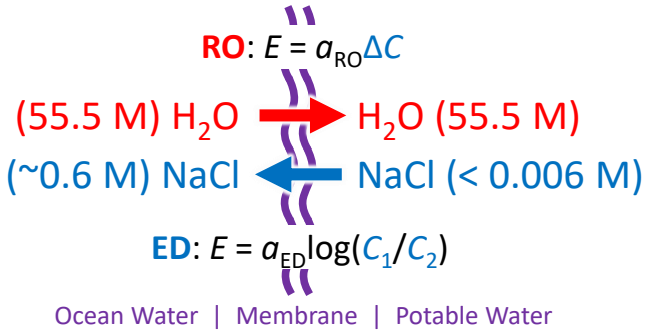


Figure 3.1. Diagram depicting two desalination processes: reverse osmosis (RO) and electrodialysis (ED). The goal of RO is to transport water (red) and the energy (E) requirement to do so is proportional to the difference in salt concentration across the membrane (purple wavy lines). The goal of ED is to transport salt (blue) and the E requirement to do so is proportional to the logarithm of the ratio of the salt concentrations on each side of the membrane.

predominantly transports ions of one charge type, resulting in net transport of ions away from the salty water thus desalinating it (Figure 3.1).

The activity presented herein demonstrates desalination by ED, as well as its relevance to concerns of global importance, i.e. desalination of salt water to potable levels. It introduces the chemical physics and mechanisms of polymeric ion-exchange membranes and techniques common to chemists and chemical engineers, notably pH, dialysis, and electrochemistry.

Technical Background

Two of the most important physical processes that dictate the performance of desalination by ED are mass transport (i.e. the rate) and membrane thermodynamics (i.e. energetics). Mass transport is simply that: the transport of mass. It constitutes a collection of processes that are common to plant-scale chemical engineering, including both RO and ED. A mathematical and physical description of basic transport phenomena are described in the Supporting Information. The thermodynamics of membrane processes for ED are primarily dictated by the net charge of the functional groups that are covalently bound, i.e. fixed, at the membrane surface. The most common charged groups are sulfonates for cation-exchange membranes (CEMs) and quaternary ammoniums for anion-exchange membranes (AEMs) (Figure 3.2).[13,14] The presence of these fixed charges affords CEMs with preferential transport of cations through their bulk and affords AEMs with preferential transport of anions through their bulk. A physical explanation of this basic membrane physics is

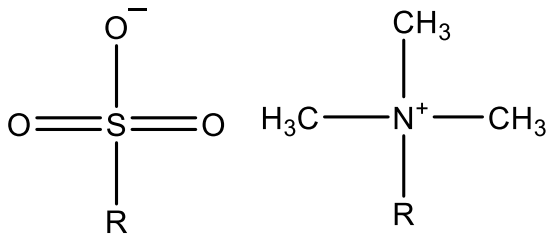


Figure 3.2. Most common fixed-charge groups found in ion-exchange membranes: sulfonates (left) in CEMs and quaternary ammoniums, e.g. trimethylammonium (right), in AEMs.

described in the Supporting Information. During desalination by ED, CEMs predominantly transport Na^+, but for the purpose of the activity associated with this curriculum Na^+ can be replaced by H^+, because they have the same charge and proton concentration is easier to visualize.

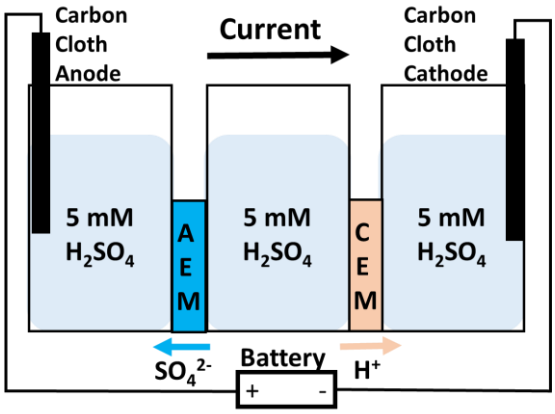


Figure 3.3. Diagram of cell setup consisting of three cuvettes, two ion-exchange membranes (cation-exchange membrane (CEM) and anion-exchange membrane (AEM)), and two carbon-cloth electrodes that are slid down the inside end faces of each cuvette and connected to a battery using alligator clips and wires. Also shown are the directions of predominant ion transport (SO_4^{2-}, H^+) through each membrane and net current flow during ED.

Experimental Methods

Electrochemical Setup

The experimental setup for the activity is shown in Figure 3.3 and consists of three chambers each constructed from a standard disposable 2.5 mL square-bottomed plastic cuvette. Holes (8 mm in diameter) are drilled near the bottom of one side of each of the two outer cuvettes and two holes are drilled through opposite sides of the middle cuvette. The holes are then covered in Teflon tape, by wrapped it around the walls of the cuvette twice, and then, using scissors, the tape covering each hole is cut out and removed. The Teflon tape serves as a gasket to form a water-tight seal with the membranes. Two commercial polymeric ion-exchange membranes are placed between pairs of the cuvettes and the completed cell is then clamped together with a C-clamp to ensure water-tight seals. By driving an anodic reaction (oxidation) in the chamber in contact with the AEM and a cathodic reaction (reduction) in the chamber in contact with the CEM, ions in the center cuvette are forced through the membranes based on their charge such that cation transport occurs predominantly through the CEM and anion transport occurs predominantly through the

AEM. This process desalinates (deionizes) the water in the central chamber via ED. From here on we refer to the outer chamber in contact with the AEM as the anode chamber and the outer chamber in contact with the CEM as the cathode chamber.

Figure 3.4. Thymol blue pH indicator at pH values of 1 to 10 in steps of 1 (from left to right).

Deionization with Acid

Desalination is a subset of a broader technique called deionization where instead of solely removing mineral salt ions, ions in general are removed, e.g. protons and conjugate bases. Both desalination and deionization obey the same physics that dictates the rate (mass transport) and energetics (membrane thermodynamics) of these processes. Replacement of salt with acid is useful for a low-cost version of this activity because acid concentration, i.e. as pH, is facile to measure precisely using a colorimetric indicator, whereas salt concentration is not. In the activity described herein, aqueous H_2SO_4 is used as the acid in place of NaCl, such that H^+ are transported by the CEM instead of Na^+ and SO_4^{2-} are transported by the AEM instead of Cl^-. Thymol blue is used as the colorimetric pH indicator. Thymol blue has two pK_a values (8.9 and 1.6) and therefore it exhibits two distinct color transitions (red-to-yellow-to-blue) based on the activity of protons in the solution it is dissolved in, i.e. pH.[15–17] The color changes occurring at low pH are important in this activity to monitor the deionization process of ~0.01 M acid (pH ≈ 2.0).

The nature of the current-carrying electrodes is not strict; however, in the activity described herein, carbon cloth electrodes were positioned inside the outer chambers and

connected to a 9 V battery. ED occurred with the application of any potential beyond that required to drive two electrochemical redox half reactions, which were likely $4\ H^+ + 4\ e^- \rightarrow 2\ H_2$ and $2\ H_2O \rightarrow O_2 + 4\ H^+$ (water electrolysis at 1.23 V) when H_2SO_4 was used. Therefore, 9 V was a large excess of potential but it helped facilitate rapid deionization; generally, the larger the potential the faster the rate of the electrochemical reactions. It is critical that the anode of the ED cell be connected to the '+' terminal of the battery and the cathode of the ED cell be connected to the '–' terminal of the battery. If not, the central chamber will become enriched in ions instead of depleted of ions as current is passed. After 30 minutes, the battery was disconnected and thymol blue pH indicator was added to each chamber. Do not add thymol blue while the battery is connected, because thymol blue reacts at the electrodes and degrades.

Figure 3.5. Digital photographs of the ED cell after 9 V was applied across the cell for 30 min and thymol blue pH indicator was added: (top) front view (bottom) top-down view. Originally, each of these three chambers contained 5 mM H_2SO_4 and was completely colorless.

Several possible variations to the activity are discussed in the Supporting Information and include using a different power source, type of electrode, pH indicator, type of acid and/or salt instead of acid, and operating the cell for a longer time.

Safety Considerations

This activity uses dilute but caustic solutions that are eye, skin, and respiratory irritants and therefore, skin contact, eye contact, and inhalation should be avoided through use of proper personal protective equipment. Moreover, prior to initiating the experiments

a procedure for safe disposal of these solutions must exist. Another potential hazard is electrical shock from the 9 V battery; therefore, during ED the battery and cell should not be handled. Moreover, after the ED process is complete the leads should be disconnected from the battery terminals in order to prevent the possibility of forming an electrical shunt that can result in high currents being passed and large dissipation of heat. Additionally, if this activity is performed with younger children, they can assemble the dry electrochemical cell but it is advised that a supervisor add the acid solution to each chamber, immerse the electrodes in solution, attach the battery, and disconnect the battery at the end of the experiment.

Results

After 30 minutes of electrodialysis using a 9 V battery and subsequent addition of thymol blue pH indicator to all three chambers originally containing transparent and colorless solutions, the outer chambers appeared red while the center chamber appeared orange due to successful deionization. The differences in color will be more (less) pronounced if the experiment is performed for more (less) than 30 minutes or a smaller (larger) volume of solution is used. The color directly relates to the pH of the solution and therefore to the number of free H^+ (H_3O^+) in solution.

Discussion

This activity was performed with > 25 middle-school students, high-school students, and undergraduate students majoring in chemistry. Formal survey feedback was obtained from the high-school students. Each student performed the activity but worked in pairs to

assist in more delicate procedural steps. About half of the high-school students obtained the intended results. Those that did not either did not make large enough holes in their Teflon tape or attached their battery backward. Greater than one third of the high-school students remarked that this activity was more enjoyable and a better learning experience than "Juice from Juice," a well-established outreach activity that the students also performed in the same day. "Juice from Juice" is a dye-sensitized solar cell activity where blackberry dyes are used to fabricate solar cells. It is based on the activity published by Smestad and Grätzel in 1998,[16] and more recently popularized by the Solar Center for Chemical Innovation and supported by the National Science Foundation. The most common feedback received from the high-school students was to increase the clarity of the instruction, such as related to background information, steps in the procedure, names of the components used, and conclusions based on observed outcomes. Several students also suggested that technical names for each component be listed to assist in describing the processes involved in the overall function. Adjustments were made to the curriculum in support of this feedback, such as emphasizing the importance of the orientation of the battery connection.

The following inquiry questions may be asked of students to promote critical thinking and to assess learning outcomes.

- Explain why the cathode (and anode) chamber become more acidic (or basic) over time.
- Explain why the color change due to thymol blue depends on the type of acid used.
- If the difference in color is subtle, experimentally how can the color change be increased in terms of the power source, time, and concentration of acid initially used?

- Explain why a color change of thymol blue is, or is not, observed with weaker acids such as vinegar (acetic acid)?
- Is a color change due to thymol blue expected if the solutions contain both acid and salt? Why or why not?

Conclusions

A hands-on scientific inquiry activity and accompanying curriculum have been developed to promote the importance of desalination to future science leaders. This activity allows for a wide distribution of subject-matter to be covered, including acid strength (pH), conductivity dependence on type and concentration of salt solution (i.e. electrolyte), pH indicators, spectrophotometry, polymeric ion-selective membranes, ionic circuits, osmotic pressure, among other topics. The flexibility in setup of this activity, which has many cost-effective options for implementation, allows for dissemination throughout many levels of science education and with a limited budget for supplies. We envision this being a core component in advanced chemistry courses at the middle-to-high-school level and general chemistry courses at the college level.

Conclusions

A model has been made which simulates the accumulation of electron-holes on catalyst sites in dye-sensitized solar cell like systems both under continuous illumination and during pulsed laser light experiments. Many simulated conditions have been tried to best determine what limitations these systems have experimentally and what parameters are crucial to model correctly. These findings as well as the model itself can be used by

researchers studying dye-sensitized solar cells or any system which relies on charge accumulation in trap sites.

Bibliography

Introduction

(1) Green, M.A. J Mater Sci: Mater Electron 2007, 18(Suppl 1), 15
(2) W. Shockley and H. Queisser. Journal of Applied Physics 1961, 32(3), 510
(3) Green, MA, Hishikawa, Y, Dunlop, ED, et al. Prog Photovolt Res Appl. 2019; 27: 3– 12
(4) X-Series Home Series|Sun Home. https://us.sunpower.com/solar-panels-technology/x-series-solar-panels. (Accessed April 24, 2019)
(5) Juan Bisquert, David Cahen, Gary Hodes, Sven Rühle, Arie Zaban. J. Phys. Chem. B 2004 108 (24), 8106-8118
(6) K. Kakiage, Y. Aoyama, T. Yano, K. Oya, J. Fujisawa and M. Hanaya. Chem. Commun., 2015, 51, 15894-15897
(7) S. Ardo and G. J. Meyer. Chemical Society Reviews 2009, 38(1), 115-164
(8) Higgins, G. T.; Bergeron, B. V.; Hasselmann, G. M.; Farzad, F.; Meyer, G. J. J. Phys. Chem. B 2006, 110 (6), 2598–2605
(9) Ke Hu, Kiyoshi C. D. Robson, Evan E. Beauvilliers, Eduardo Schott, Ximena Zarate, Ramiro Arratia-Perez, Curtis P. Berlinguette, and Gerald J. Meyer. Journal of the American Chemical Society 2014 136 (3), 1034-1046
(10) D. M. Fabian, S. Hu, N. Singh, F. A. Houle, T. Hisatomi, K. Domen, F. E. Osterloh, S. Ardo. Energy & Environmental Science 2015, 8(10), 2825-2850

Chapter 1

(1) B. D. James, G. N. Baum, J. Perez, K. N. Baum, DOE Contract Number GS-10F-009J, 2009, 1–128.
(2) Pinaud, B. A.; Benck, J. D.; Seitz, L. C.; Forman, A. J.; Chen, Z.; Deutsch, T. G.; James, B. D.; Baum, K. N.; Baum, G. N.; Ardo, S.; Wang, H.; Miller, E.; Jaramillo, T. F. *Energy Environ. Sci.* **2013**, *6* (7), 1983–2002.
(3) Ager, J. W.; Shaner, M. R.; Walczak, K. A.; Sharp, I. D.; Ardo, S. *Energy Environ. Sci.* **2015**, *8* (10), 2811–2824.
(4) Bala Chandran, R.; Breen, S.; Shao, Y.; Ardo, S.; Weber, A. Z. *Energy Environ. Sci.* **2018**, *11* (1), 115–135.
(5) Ardo, S.; Fernandez Rivas, D.; Modestino, M. A.; Schulze Greiving, V.; Abdi, F. F.; Alarcon Llado, E.; Artero, V.; Ayers, K.; Battaglia, C.; Becker, J.-P.; Bederak, D.; Berger, A.; Buda, F.; Chinello, E.; Dam, B.; Di Palma, V.; Edvinsson, T.; Fujii, K.; Gardeniers, H.; Geerlings, H.; H. Hashemi, S. M.; Haussener, S.; Houle, F.; Huskens, J.; James, B. D.; Konrad, K.; Kudo, A.; Kunturu, P. P.; Lohse, D.; Mei, B.; Miller, E. L.; Moore, G. F.; Muller, J.; Orchard, K. L.; Rosser, T. E.; Saadi, F. H.; Schüttauf, J.-W.; Seger, B.; Sheehan, S. W.; Smith, W. A.; Spurgeon, J.; Tang, M. H.; van de Krol, R.; Vesborg, P. C. K.; Westerik, P. *Energy Environ. Sci.* **2018**, *11* (10), 2768–2783.
(6) Verlage, E.; Hu, S.; Liu, R.; Jones, R. J. R.; Sun, K.; Xiang, C.; Lewis, N. S.; Atwater, H. A. *Energy Environ. Sci.* **2015**, *8* (11), 3166–3172.
(7) Jia, J.; Seitz, L. C.; Benck, J. D.; Huo, Y.; Chen, Y.; Ng, J. W. D.; Bilir, T.; Harris, J. S.; Jaramillo, T. F. *Nat. Commun.* **2016**, *7* (May), 1–6.
(8) Schüttauf, J.-W.; Modestino, M. A.; Chinello, E.; Lambelet, D.; Delfino, A.; Dominé, D.; Faes, A.; Despeisse, M.; Bailat, J.; Psaltis, D.; Moser, C.; Ballif, C. *J. Electrochem. Soc.* **2016**, *163* (10), F1177–F1181.
(9) Zhou, X.; Liu, R.; Sun, K.; Chen, Y.; Verlage, E.; Francis, S. A.; Lewis, N. S.; Xiang, C. *ACS Energy Lett.* **2016**, *1* (4), 764–770.
(10) Young, J. L.; Steiner, M. A.; Döscher, H.; France, R. M.; Turner, J. A.; Deutsch, T. G. *Nat. Energy* **2017**, *2* (4), 1–8.
(11) Yuvraj, S. *Modeling and experimental demonstration of an integrated photoelectrochemical hydrogen generator working under concentrated irradiation*, École Polytechnique Fédérale de Lausanne, Thesis, 2018.
(12) Li, F.; Yang, H.; Li, W.; Sun, L. *Joule* **2018**, *2* (1), 36–60.
(13) Kudo, A.; Miseki, Y. *Chem. Soc. Rev.* **2009**, *38* (1), 253–278.
(14) Alibabaei, L.; Brennaman, M. K.; Norris, M. R.; Kalanyan, B.; Song, W.; Losego, M. D.; Concepcion, J. J.; Binstead, R. a.; Parsons, G. N.; Meyer, T. J. *Proc. Nat. Acad. Sci. USA* **2013**, *110*, 20008–20013.
(15) Lin, F.; Boettcher, S. W. *Nat. Mater.* **2014**, *13* (1), 81–86.

(16) Warren, E. L.; Atwater, H. A.; Lewis, N. S. *J. Phys. Chem. C* **2014**, *118* (2), 747–759.
(17) Yu, Z.; Li, F.; Sun, L. *Energy Environ. Sci.* **2015**, *8* (3), 760–775.
(18) Zandi, O.; Hamann, T. W. *Nat. Chem.* **2016**, *8* (8), 778–783.
(19) Kalisman, P.; Nakibli, Y.; Amirav, L. *Nano Lett.* **2016**, *16* (3), 1776–1781.
(20) Kim, H. J.; Kearney, K. L.; Le, L. H.; Haber, Z. J.; Rockett, A. A.; Rose, M. J. *J. Phys. Chem. C* **2016**, *120* (45), 25697–25708.
(21) Utterback, J. K.; Grennell, A. N.; Wilker, M. B.; Pearce, O. M.; Eaves, J. D.; Dukovic, G. *Nat. Chem.* **2016**, *8* (11), 1061–1066.
(22) Pekarek, R. T.; Kearney, K.; Simon, B. M.; Ertekin, E.; Rockett, A. A.; Rose, M. J. *J. Am. Chem. Soc.* **2018**, *140* (41), 13223–13232.
(23) Wadsworth, B. L.; Khusnutdinova, D.; Moore, G. F. *J. Mater. Chem. A* **2018**, *6* (44), 21654–21665.
(24) Osterloh, F. E. *Chem. Soc. Rev.* **2013**, *42* (6), 2294–2320.
(25) Fabian, D. M.; Hu, S.; Singh, N.; Houle, F. A.; Hisatomi, T.; Domen, K.; Osterloh, F. E.; Ardo, S. *Energy Environ. Sci.* **2015**, *8* (10), 2825–2850.
(26) Maeda, K.; Domen, K. *J. Phys. Chem. Lett.* **2010**, *1* (18), 2655–2661.
(27) Hisatomi, T.; Kubota, J.; Domen, K. *Chem. Soc. Rev.* **2014**, *43* (22), 7520–7535.
(28) Wrighton, M. S. *Acc. Chem. Res.* **1979**, *12* (9), 303–310.
(29) Dominey, R. N.; Lewis, N. S.; Bruce, J. A.; Bookbinder, D. C.; Wrighton, M. S. *J. Am. Chem. Soc.* **1982**, *104* (2), 467–482.
(30) Moore, G. F.; Blakemore, J. D.; Milot, R. L.; Hull, J. F.; Song, H.; Cai, L.; Schmuttenmaer, C. A.; Crabtree, R. H.; Brudvig, G. W. *Energy Environ. Sci.* **2011**, *4* (7), 2389.
(31) Foley, J. M.; Price, M. J.; Feldblyum, J. I.; Maldonado, S. *Energy Environ. Sci.* **2012**, *5* (1), 5203–5220.
(32) Haussener, S.; Xiang, C.; Spurgeon, J. M.; Ardo, S.; Lewis, N. S.; Weber, A. Z. *Energy Environ. Sci.* **2012**, *5* (12), 9922–9935.
(33) Mills, T. J.; Lin, F.; Boettcher, S. W. *Phys. Rev. Lett.* **2014**, *112* (14), 148304.
(34) Singh, M. R.; Haussener, S.; Weber, A. Z. 2019; pp 500–536.
(35) Papageorgiou, N.; Grätzel, M.; Infelta, P. P. *Sol. Energy Mater. Sol. Cells* **1996**, *44* (4), 405–438.
(36) Papageorgiou, N.; Barbe, C.; Gratzel, M. *J. Phys. Chem. B* **1998**, *102* (21), 4156–4164.
(37) Bonhôte, P.; Gogniat, E.; Tingry, S.; Barbé, C.; Vlachopoulos, N.; Lenzmann, F.; Comte, P.; Grätzel, M. *J. Phys. Chem. B* **1998**, *102* (9), 1498–1507.
(38) Papageorgiou, N.; Liska, P.; Kay, A.; Grätzel, M. *J. Electrochem. Soc.* **1999**, *146* (3), 898–907.
(39) Bisquert, J. *J. Phys. Chem. B* **2002**, *106* (2), 325–333.
(40) Papageorgiou, N. *Coord. Chem. Rev.* **2004**, *248* (13–14), 1421–1446.
(41) Bisquert, J. *J. Phys. Chem. C* **2007**, *111* (46), 17163–17168.
(42) Ansari-Rad, M.; Anta, J. A.; Bisquert, J. *J. Phys. Chem. C* **2013**, *117* (32), 16275–16289.
(43) Gonzalez-Vazquez, J. P.; Oskam, G.; Anta, J. A. *J. Phys. Chem. C* **2012**, *116* (43), 22687–22697.
(44) Anta, J. A.; Casanueva, F.; Oskam, G. *J. Phys. Chem. B* **2006**, *110* (11), 5372–5378.
(45) Ardo, S.; Meyer, G. J. *Chem. Soc. Rev.* **2009**, *38* (1), 115–164.
(46) Hagfeldt, A.; Boschloo, G.; Sun, L.; Kloo, L.; Pettersson, H. *Chem. Rev.* **2010**, *110* (11), 6595–6663.
(47) Hu, K.; Meyer, G. J. *Langmuir* **2015**, *31* (41), 11164–11178.
(48) Trammell, S. A.; Yang, J.; Sykora, M.; Fleming, C. N.; Odobel, F.; Meyer, T. J. *J. Phys. Chem. B* **2001**, *105* (37), 8895–8904.
(49) Nelson, J. *Phys. Rev. B* **1999**, *59* (23), 15374–15380.
(50) Nelson, J.; Haque, S. A.; Klug, D. R.; Durrant, J. R. *Phys. Rev. B* **2001**, *63* (20), 205321.
(51) Nelson, J.; Chandler, R. E. *Coord. Chem. Rev.* **2004**, *248* (13–14), 1181–1194.
(52) Vaissier, V.; Mosconi, E.; Moia, D.; Pastore, M.; Frost, J. M.; De Angelis, F.; Barnes, P. R. F.; Nelson, J. *Chem. Mater.* **2014**, *26* (16), 4731–4740.
(53) Wang, Q.; Ito, S.; Grätzel, M.; Fabregat-Santiago, F.; Mora-Seró, I.; Bisquert, J.; Bessho, T.; Imai, H. *J. Phys. Chem. B* **2006**, *110* (50), 25210–25221.
(54) Anta, J. A.; Nelson, J.; Quirke, N. *Phys. Rev. B - Condens. Matter Mater. Phys.* **2002**, *65* (12), 1–10.
(55) Wang, Q.; Moser, J.; Grätzel, M. *J. Phys. Chem. B* **2005**, *109* (31), 14945–14953.
(56) Gao, F.; Wang, Y.; Shi, D.; Zhang, J.; Wang, M.; Jing, X.; Humphry-Baker, R.; Wang, P.; Zakeeruddin, S. M.; Grätzel, M. *J. Am. Chem. Soc.* **2008**, *130* (32), 10720–10728.
(57) Trammell, S. A.; Meyer, T. J. *J. Phys. Chem. B* **1999**, *103* (1), 104–107.
(58) Ardo, S.; Meyer, G. J. *J. Am. Chem. Soc.* **2010**, *132* (27), 9283–9285.

(59) Ardo, S.; Sun, Y.; Staniszewski, A.; Castellano, F. N.; Meyer, G. J. *J. Am. Chem. Soc.* **2010**, *132* (19), 6696–6709.
(60) Chen, H.; Ardo, S. *Nat. Chem.* **2017**, *10* (1), 17–23.
(61) Moia, D.; Szumska, A.; Vaissier, V.; Planells, M.; Robertson, N.; O'Regan, B. C.; Nelson, J.; Barnes, P. R. F. *J. Am. Chem. Soc.* **2016**, *138* (40), 13197–13206.
(62) Ardo, S. *Photoinduced Charge, Ion & Energy Transfer Processes at Transition-Metal Coordination Compounds Anchored to Mesoporous, Nanocrystalline Metal-Oxide Thin Films*, Johns Hopkins University, Thesis, 2010.
(63) Barzykin, A. V.; Tachiya, M. *J. Phys. Chem. B* **2002**, *106* (17), 4356–4363.
(64) Katoh, R.; Furube, A.; Barzykin, A. V.; Arakawa, H.; Tachiya, M. *Coord. Chem. Rev.* **2004**, *248* (13–14), 1195–1213.
(65) Barzykin, A. V.; Tachiya, M. *J. Phys. Chem. B* **2004**, *108* (24), 8385–8389.
(66) Higgins, G. T.; Bergeron, B. V.; Hasselmann, G. M.; Farzad, F.; Meyer, G. J. *J. Phys. Chem. B* **2006**, *110* (6), 2598–2605.
(67) Hu, K.; Robson, K. C. D.; Beauvilliers, E. E.; Schott, E.; Zarate, X.; Arratia-Perez, R.; Berlinguette, C. P.; Meyer, G. J. *J. Am. Chem. Soc.* **2014**, *136* (3), 1034–1046.
(68) Anta, J. A.; Mora-Seró, I.; Dittrich, T.; Bisquert, J. *Phys. Chem. Chem. Phys.* **2008**, *10* (30), 4478.
(69) Anta, J. A. *Energy Environ. Sci.* **2009**, *2* (4), 387.
(70) Anta, J. A.; Morales-Flórez, V. *J. Phys. Chem. C* **2008**, *112* (27), 10287–10293.
(71) Brennan, B. J.; Durrell, A. C.; Koepf, M.; Crabtree, R. H.; Brudvig, G. W. *Phys. Chem. Chem. Phys.* **2015**, *17* (19), 12728–12734.
(72) Brennan, B. J.; Regan, K. P.; Durrell, A. C.; Schmuttenmaer, C. A.; Brudvig, G. W. *ACS Energy Lett.* **2017**, *2* (1), 168–173.
(73) Rettie, A. J. E.; Chemelewski, W. D.; Emin, D.; Mullins, C. B. *J. Phys. Chem. Lett.* **2016**, *7* (3), 471–479.
(74) Ardo, S.; Meyer, G. J. *J. Am. Chem. Soc.* **2011**, *133* (39), 15384–15396.
(75) O'Regan, B. C.; Durrant, J. R. *Acc. Chem. Res.* **2009**, *42* (11), 1799–1808.
(76) Askerka, M.; Brudvig, G. W.; Batista, V. S. *Acc. Chem. Res.* **2017**, *50* (1), 41–48.
(77) Ashford, D. L.; Gish, M. K.; Vannucci, A. K.; Brennaman, M. K.; Templeton, J. L.; Papanikolas, J. M.; Meyer, T. J. *Chem. Rev.* **2015**, *115* (23), 13006–13049.
(78) Xu, P.; Gray, C. L.; Xiao, L.; Mallouk, T. E. *J. Am. Chem. Soc.* **2018**, *140* (37), 11647–11654.
(79) Brigham, E. C.; Meyer, G. J. *J. Phys. Chem. C* **2014**, *118* (15), 7886–7893.

Chapter 3

(1) Gleick, P. . *Environment* **1994**, *36* (3), 6–42.
(2) Gleick, P. H. *Int. Secur.* **1993**, *18* (1), 79–112.
(3) The United Nations Educational Scientific and Cultural Organization. *Managing Water under Uncertainty and Risk*; 2012; Vol. 1.
(4) California Department of Food and Agriculture. California Agricultural Statistics Review, **2015**, 1–126.
(5) Garbarini, G.; Eaton, R. *J. Chem. Educ.* **1971**, *48* (4), 226–230.
(6) Suess, M. J. *J. Chem. Educ.* **1971**, *48* (3), 190–192.
(7) Hecht, C. E. *J. Chem. Educ.* **1967**, *44* (1), 53–54.
(8) Kendall, A. I.; Gebauer-Fuelnegg, E. In *81st Meetings of the American Chemical Society*; 1931; pp 1634–1639.
(9) Shufle, J. A. *J. Chem. Educ.* **1961**, *38* (1), 17–19.
(10) García-García, V.; Montiel, V.; González-García, J.; Expósito, E.; Iniesta, J.; Bonete, P.; Inglés, M. *J. Chem. Educ.* **2000**, *77* (11), 1477–1479.
(11) Rozoy, E.; Boudesocque, L.; Bazinet, L. *J. Agric. Food Chem.* **2015**, *63* (2), 642–651.
(12) Behrman, A. S. *J. Chem. Educ.* **1929**, *6* (10), 1611–1618.
(13) Strathmann, H.; Grabowski, A.; Eigenberger, G. *Ind. Eng. Chem. Res.* **2013**, *52* (31), 10364–10379.
(14) Gaieck, W.; Ardo, S. *Rev. Adv. Sci. Eng.* **2014**, *3* (4), 277–287.
(15) Suzuki, C. *J. Chem. Educ.* **1990**, *68* (7), 588–589.
(16) Smestad, G. P.; Grätzel, M. *J. Chem. Educ.* **1998**, *75* (6), 752–756.
(17) Silva, C. R.; Pereira, R. B.; Sabadini, E. *J. Chem. Educ.* **2001**, *78* (7), 939–940.

Appendix F

(1) B. D. James, G. N. Baum, J. Perez, K. N. Baum, DOE Contract Number GS-10F-009J, 2009, 1–128.

(2) Pinaud, B. A.; Benck, J. D.; Seitz, L. C.; Forman, A. J.; Chen, Z.; Deutsch, T. G.; James, B. D.; Baum, K. N.; Baum, G. N.; Ardo, S.; Wang, H.; Miller, E.; Jaramillo, T. F. *Energy Environ. Sci.* **2013**, *6* (7), 1983–2002.

APPENDIX A. Modeling Guide in Mathematica

The Guide

A user's manual in cutting edge Monte Carlo modeling of intermolecular electron-hole hopping and accumulation at materials electrocatalyst sites on procedurally generated photo-sensitized mesoporous semiconductor surfaces

First there will be a high-level overview of the code and its capabilities. A second more in-depth explanation section by section will follow. For even more detailed explanations of code, please see in-line comments. The language here will be fairly colloquial, so this isn't terribly boring to write (so that I actually do it) and so that it isn't (as) terribly boring to read. The descriptions are given in the same order as the code for ease of reference and writing. It may be helpful to read this along with the code side by side.

The model – a program in two acts

The model is currently set up to run in two different pieces for reasons that will be made clear later. The first part creates a surface analogous to mesoporous TiO_2. The second runs a Monte Carlo experiment on the generated surface. These will both be discussed at a high level and then in detail separately. The overall goal of the program is to better understand charge accumulation at catalyst sites on photo-sensitized surfaces. A relevant example of such a system is TiO_2 sensitized with $Ru(bpy)_3$ like dyes in a Dye sensitized solar cell (DSSC). We model this type of system as a number of spherical particles parameterized in a number of ways that have set molecular sites which can hold, dyes, catalysts, or nothing at all. The catalysts act as thermodynamic sinks which can be oxidized or reduced multiple

times. This is crucial to the behavior of DSSC-like systems which require performing chemistry involving multiple electrons simultaneously. In our system, light interacts with the dye molecules which creates a mobile electron-hole pair. The electron then (we assume instantly and with 100% quantum yield) injects into the TiO_2 leaving behind a lone hole. Through electron-hole hopping this hole can "hop" from neighboring molecule to neighboring molecule until it either A) recombines with an electron in TiO_2 or B) reaches a catalyst. Once on the catalyst, the electron-hole may still recombine but once enough holes have gathered on the catalyst to perform the chemistry required, the catalyst fully turns over and the excitations are considered to have contributed to turnover. Determining what conditions and what systems lead to a large percentage of excitations turnover is the focus of this model. Additionally, kinetic information is recorded so we can determine how the system is progressing over time.

Act I – Creating the Surface

Overview

In this part, a surface to run future models on can be created with various settings. These include particle size ranges, particle necking ranges, whether to form a cluster or a stack, and the density of molecular positions on the surfaces of these particles. After settings are chosen, the code procedurally generates a surface one particle at a time. This starts with 1 particle and continually proposes additional particles. Each proposal is either determined to meet all settings or is rejected and a new proposal is made. This continues until the number of particles in the surface are as specified. Once particles are placed, molecular positions for each particle are set either using Mathematica's tessellation functionality (don't

use this) or using Fibonacci spirals which can be scaled to particle size. Once positions are chosen, particle neighbors are determined, and valid positions are determined. Particle neighbors are any particles in contact with a given particle meaning that molecular sites on any of a particle's neighbor list could potentially be neighbors of sites on that particle. Valid positions are positions which are not within a neighboring particle and which are also not too close to other molecular sites. Once valid positions are determined, the distance between each molecular site and every site that could potentially be a neighboring site is calculated and stored in a large table. From this table, the nearest neighbors for each molecular site are selected. Following this selection, all pertinent information needed to run a model is stored for this surface and may be exported later. Prior to exportation, a few graphics are displayed so we can get a decent idea of the surface we are about to save. Following the graphics, exportation will take the filename specified in the settings and save everything needed to run the model into a WDX file. After exportation there are two code blocks which allow further investigation into the surface created but are not actually needed to run the model. The first of these blocks display information related to the number and spacing of nearest neighbors. The second of these creates a graphic showing the percolation zones in the surface. For more details, everything above will now be discussed further below.

Detailed Walkthrough

Functions

There are two functions created in the first section. The first is pythag[] which is simply an application of the Pythagorean theorem in 3D. This is used preferentially over Mathematica's built in EuclideanDistance[] function because it is slightly faster since it does

not apply an Absolute value to each pair of distances. Since we're working with all real points, there is no need to take the absolute value after squaring.

This is a doozy and not necessary to understand from the beginning. The second function is a nasty looking thing but is somewhat simpler than it appears. This is the FloodFill[] function and is optionally used in the final block of code in the surface generation program. This function also appears in the main code where it is actually needed. Flood fill is a recursive algorithm used in many programs and is the same basic function used by that little paint can in Paint to color in a whole area. Paint is an easy example to explain how it works but you can also check out (https://en.wikipedia.org/wiki/Flood_fill) for some helpful gifs. Anyway, an example with paint: Say you have an image that is all black and white pixels and you want to color a certain area red. By clicking that red paint can in a white pixel, you select white as the target-color and red as the replacement color. The flood fill algorithm now takes over and proceeds as follows:

-If the current pixel is the target color, change to the replacement color and then call Flood Fill on the pixel above it, the pixel below it, the pixel to the right and the pixel to the left.

-If the pixel is not the target color (if it was black or already changed red) return and do nothing.

That's it! In this way, the algorithm changes the initial pixel and then checks all its neighbors and then checks all their neighbors and so on until it hits a pixel that isn't white and then is just leaves that pixel alone and doesn't check its neighbors. So eventually, all pixels that are of the same color that are in contact with each other get altered and all other pixels get left alone. So how does this have anything to do with the code? The model relies on electron-holes hopping from neighbor to neighbor until they reach a catalyst or

recombine. Depending on experimental settings, the recombination lifetime may be extremely long, and catalysts may be fairly scarce. This means there are potentially scenarios in which excitations occur on molecules which have no catalysts in their network or percolation zone. If this is the case, the electron-hole will continue to hop around until it recombines. If the recombination lifetime is very long relative to the hopping lifetime, this will take computationally forever that the model will have to be terminated. That sucks and is something to be avoided so before anything begins, a modified flood fill algorithm is used to identify which sections of the generated surface are in contact with each other. That way excitations occurring on molecules with no catalysts in their network can be identified and dealt with. More on that when discussing part II. The use of FloodFill[] in the surface generator is to identify what the percolation zones are. Often, the whole generated surface is mutually connected and so this is all a moot point. However, if the procedural generation happened to create a few patches of disconnected molecule sites, this is good to know ahead of time. For large surfaces, this will use an extremely large amount of RAM and so should be used only after saving and preparing for the computer used to hang for a bit. The actual implementation of the function will be discussed in the Functions section of part II so that I can move on to the actual program now.

Settings

Settings are probably the simplest to understand but also the most important to get right. Ideally, nothing outside of the settings section will ever need to be changed by a user. The *stackName* is simply the filename to be used if the surface generated is exported. *r1positions* is the number of molecular sites which will be placed on a particle of radius 1. Other size particles will have their number scaled so the packing density is the same.

moleculeRadius is the effective radius of each molecular site. This is analogous to the van Der Waals radius. This and all other radius settings are in units where 1 = 15 nm so that the standard size particles (which are 15 nm in radius) can have a radius of 1. Molecules will not be able to be placed within 2 molecular radii of each other on other particles (if they are too near each other near the particle boundaries) but will packed on the same particle as close as needed determined by *r1positions. neckingMin* and *neckingMax* set the amount of overlap that can occur between particles. This represents the fraction of the particles radius which can be overlapped by another particle. In the case that two particles are of different sizes, the fractional overlap is calculated from the smaller particle's perspective. In this way, necking of 0 means particles will all be tangent to each other, necking of 1 means the center of a particle will be at the surface of another, and necking <0 means particles will not be in contact. Initial necking of a proposed particle is chosen from a uniform distribution between the necking minimum and maximum. *partRadMin* and *parRadMax* are the ranges that a proposed particle's radius is bounded by. The choice of new radius is taken from a uniform distribution between the bounds. 1 is a standard size so this range should ideally be around 1. *numParticles* is simply the number of particles to be used in the surface. Usually 100 for large experiments and 10 for quick test trials. *FIB* is a Boolean that, if left true, will use Fibonacci Spirals to assign molecule positions around each particle. If set False, tessellation will be used and will not be as good and will not work if particles of different sizes are used. Leave FIB true unless there is a very good reason to change it. *Stack* is a Boolean that, if true, will always add new particles along the z-axis such that the final surface is a single column of particles. If False, a cluster that grows out in 3D will be created. *ClusterCompactness* is only relevant if *stack* is False and determines how tightly packed particles in a clustered surface

will be. Large positive values (10) will result in nearly spherical clusters of particles while large negative values (-10) will result is long dendritic chains of particles. *reach* is the number of molecular radii that an adjacent molecule can be away and still be considered a nearest neighbor. So, if *moleculeRadius* = 0.05 and *reach* = 3.0 then any molecule within 0.15 of a given molecule is that given molecule's neighbor. *reach* should be set such that each molecule has between 5-10 nearest neighbors. The graphics at the end can help evaluate if a reach value is appropriate.

Choosing Particles

Particles are chosen iteratively until the number set by *numParticles* have been chosen. The first particle is given a random radius and is set at (0,0,0). Each additional particle is added on to the surface starting from an existing particle. So, the 2nd particle will grow off the 1st and the 3rd will grow off either the 1st or the 2nd and so on. After the first particle, a new proposed particle is generated by choosing a new valid particle radius, choosing an existing particle to "Grow off of", choosing a random unit vector off that chosen particle, and then choosing a valid necking fraction with that chosen particle. At this point the proposed particle is a valid particle as far as the chosen existing particle but it must be verified that new particle doesn't overlap any other existing particles too much. The fractional overlap is calculated between the new particle and all existing particles and if any of them surpass the requirement set by *neckingMax* the particle is rejected. Otherwise the particle is set, and its position and radius are stored for future reference. In the case of particle rejection, the random proposal process begins from the start with a new radius, new attachment point, etc. The choosing of the attachment point is where *ClusterCompactness* is brought into use. Rather than give the choice of the attachment point a simple uniform

distribution the weight of a particle being chosen is equal to the reverse order of addition to the surface raised to the power of cluster compactness. So, if *ClusterCompactness* 3 and there were currently 5 particles in the cluster, the weights of particles being chosen as an attachment point would be [125,64,27,8,1] for particles [1,2,3,4,5] and particles are more likely to be added to older particles. If *ClusterCompactness* was -3, the weights would be [1/125,1/64,1/27,1/8,1] for particles [1,2,3,4,5] and particles would be more likely to be added to the newest particles. A uniform distribution can be achieved with a *ClusterCompactness* of 0.

Setting Sites

Sites are usually set with Fibonacci spirals, if this turned off, rather than generate an arrangement of sites, a set pattern with 252 sites per particle is used. This is not scaled for size. If *FIB* is true, a number of sites will be placed over each particle using math derived from Fibonacci spirals (golden spirals). More info on that here: http://blog.marmakoide.org/?p=1. Once sites have been chosen, particles must be randomly rotated so that the spiral axis is not always oriented in the same direction. This helps maintain an isotropic distribution over the surface. This rotation is done by, choosing two random angles. The inclination angle chosen between 0 and 180 degrees and the azimuthal angle chosen between 0 and 360 degrees are used to create a rotation matrix which is multiplied by the positions previously determined. The sites are then shifted over to their particle's center and scaled radially based on the radius. These newly calculated positions, *rotated positions,* are stored for further use.

Finding Nearest Particle Neighbors

All intermolecular hole hopping is based on nearest neighbors. That is, when an oxidized dye is given the opportunity to electron-hole transfer between sites, the list of sites it can transfer between are its nearest neighbors. The same is true when a dye is going to oxidize a catalyst. Pretty much everything in this model is based on these lists of nearest neighbors. This makes generating this list of neighbors **crucial** if anything meaningful is to be learned. On the plus side, if the list of nearest neighbors isn't generated correctly, the model fails so catastrophically that it crashes or else outputs data that is clearly meaningless. While this is a major problem, its nice that is easy to catch.

To begin finding nearest neighbors, the distances between all sites and all possibly neighboring sites must be calculated. Then the possible neighboring sites are sorted by distance and added to the list of neighbors for a given site until the next nearest neighbor is further away then the reach distance. This is easier said than done and this will be the bulk of computation time when running this surface generator. The reason that is this is a problem that scales as n^2m^2 where n is the number of particles and m is the number of sites per particle. This is because, initially all molecular sites are potential neighbors of each other and so the distance between all sites and all other sites must be calculated. We mitigate this somewhat by first identifying particle neighbors

We find particle neighbors in much the same way we will find molecule neighbors. All particles are potential neighbors of each other and so a cross-distance (Xdistance in the code in several places) table is created. Once created, we use this table to determine which particles are touching and identify them as neighbors. This is done by iterating over all particles and then for each particle iterating over all the particles again (including the first

particle) and determining if the center-to-center distance between both particles is less than their combined radii. If so, they are in contact and are therefore neighboring.

Determining Valid Sites

Not all sites are created equal. Some are inside neighboring particles. Some sites are too close to other sites. In the settings section, a molecular radius was specified. If a molecular site on another particle is within this radius, the two sites are too close and are set as invalid. Presently, both molecular sites are set as invalid even though in theory one could be left without any problems. This molecular radius checking only occurs between molecular sites and sites on **other** particles, not on the same particle. This effectively checks the necking/creased region where particles come together in which molecules might end up too close to each other. The molecular spacing between sites on the **same** particle is determined by the number of positions placed on a radius 1 particle and the Fibonacci spiral math. If molecules on the **same** particle end up too close together, this number of positions must be adjusted. With these considerations in mind, the code iterates over all particles and over all sites on these particles. Each site is assumed to be valid and then tested against a pair of *if* statements which check whether the site is inside a neighboring particle or too close to another site. If either of those checks fail, the site is invalid. If it passes both checks, the site is stored in the list of valid positions. After this process, the number of valid positions per part is calculated and stored for future iterating. To save time, this iteration is done alongside the iteration over all particles to determine neighboring particles.

Finding Nearest Molecule Neighbors

Next molecular neighbors will need to be determined but instead of every site in the whole system being a potential neighbor, only sites on neighboring particles will need to be

considered. This is an n^2 process (every particle checked against every other particle) and helps us reduce the nearest neighbor distance from n^2m^2 to something more like $3nm^2$ (assuming every particle has about 2 neighbors) which is a huge savings if n ~100 and m~1000.

Now that only molecular sites on neighboring particles must be considered, the process that was done on particles is repeated for each molecular site. The computationally expensive part is making the Xdistance table. The table is made by iterating over each particle and molecular site on that particle to create a row in this table. To create the row each potential neighboring site is iterated over and the distance to the original site is calculated. All of this is stored in a table of format:

{particle#, position#, distance to particle#position# from particle i position j}

where particle I, position j represents the original particle. Once this row is created, it is sorted by distance in ascending order and stored in the table. Once this is done for every molecular site, the table is ready to be used to assign nearest neighbors!

Once the Xdistance table is made, neighbors can be assigned. An empty table of NNs is created to store nearest neighbors. Next, for each position, a variable number of neighbors is added to this table. The reason that the number is variable is that, depending on surface geometry, the packing between sites will not be consistent. To do this, each position is iterated over and starting at the 2^{nd} nearest potential neighbor (skipping the first because each site is its own 1^{st} nearest "neighbor"), each site is checked against the reach limit to be a neighbor. If a potential neighbor is within the reach distance, it is added to the nearest neighbor list and the distance between the neighbors is recorded for later use. This continues until some potential nearest neighbor is not within reach. Once one potential neighbor fails,

all neighbors for a given site have been found and its time to move on to the next site and find its neighbors.

After this, the next step used to be to set inverse nearest neighbors. This is no longer done but I will leave the discussion of why this is no longer necessary for future reference. This section can be skipped. Once nearest neighbors are set, inverse nearest neighbors are set. A given site's list of inverse nearest neighbors' list is a list of all site for which the given site is a nearest neighbor. This was implemented when the qualification for being a nearest neighbor were on a rolling basis (a potential neighbor had to be not more than 5% further than the next previous neighbor) and so there wasn't necessarily a 1-to-1 correspondence between nearest neighbors (site A might have had site B as a nearest neighbor but site B might not have had site A as a neighbor). Now that there is a hard cutoff, this is largely redundant as the list of a particle's nearest neighbors and its inverse nearest neighbors are exactly the same. The downside to the hard cutoff is that, depending on the geometry of the system two points that are very nearly the same distance may not both be included or excluded. Sort of like if you get a B in a class with 89.99 % and someone else gets an A with 90.01%. The reason for this is that, again depending on the geometry, if points are spaced out just right, all points might end up being nearest neighbors if using a rolling cutoff. Using the grade scale as an example again, this would be like if, as long as you are within 1% of the person ahead of you, you would get the same grade as that person. This can lead to situations with students getting grades of: 90.5%, 89.7%, 89%, 88.4%, 87.8%... etc. and giving everyone an A because they were all within some limit to the next point. These are both contrived examples but these sorts of things will happen in the model. Both of these methods have their limitations and downsides but the hard cutoff is much more consistent and behaves as expected so that is

the nearest neighbor assignment method used. The point is that Inverse Nearest Neighbors are no longer useful and are a vestigial remnant of previous requirements. Much like the vermiform appendix in humans.

Alongside this sorting, a few geometric parameters are determined. Those are: molecular vectors (a vector for each molecule oriented from that molecule's particle's center radially outward; this is how the electric dipole moment of each molecule in the system is assumed to be oriented), molecular angles (an angle for each molecule which is the angle between its molecular vector and the polarization of the electric field of incident light: <0,1,0>), and the anisotropy contribution (a calculation based on the molecular angle used to determine polarization anisotropy). These things are only ever needed in the full model if anisotropy measurements are turned on and polarized light is used but it is simpler to calculate them once here and be done with it while were already iterating over the whole surface.

Graphics

Wow! You reached the end of the actual code! Now for some pictures! The code that runs these graphics is actually after the setting of the rotated positions and then again after the setting of the valid positions but since all the cells in Mathematica are grouped together, the graphics are displayed after the end of the cell.

The first graphic is a display of the particles in the system. If there are fewer than 15 particles in the system, the positions (before validation) will also be displayed. Here you can see that some positions are grouped very closer to others or else merging into other particles because these are all positions and not only valid ones. It is also worth noting that the size of the position spheres corresponds to the molecular radius specified in the beginning.

The second pair of graphics displays the position on the first 5 particles (as long as there at least 5) in different colors. One displays the valid positions while the other displays all positions. These help visualize the spherical overlap and what is getting cut off when necking occurs. If particles are not in contact and therefore all positions are valid, rather than display the same graphic twice, Mathematica takes a shortcut and simply puts a ^{2} at the corner of the graphic to show it is displayed twice. If there are fewer than 5 particles, nothing prints out.

The last graphic is a repeat of the first but this time showing only valid positions. This also includes two green spheres which show reach distance from two sites on particle 1. These are just two examples of what will be included in the nearest neighbor list for those two sites. These should ideally, encompass, as closely as possible, the nearest ring of sites to the site they are centered on and can be a helpful indicator of what is going wrong if the nearest neighbor lists aren't generating well. Again, this only displays if there are fewer than 15 particles in the system as, in most cases, displaying 1000s of positions on hundreds of particles is not worth the computational wait time. If there are more than 15 particles, this does not display at all as you already have a printout of the surface sans particles.

Exporting

If the surface generated look good, running this section will save a copy with the filename specified in the settings section into your Documents folder. This can take a few minutes depending on how large the surface generated was. Presently the filetype is a WDX which is a Mathematica specific extension that is the most efficient at storing a collection of Mathematica variables to be accessed later. It isn't usable by anything but Mathematica. It is

worth noting that Export[] does not ask permission before overwriting previous files so if you have a surface you want to keep but the filename is going to overwrite something, make sure to move the previous file out of documents or else change the filename (in the cell with Export[]).

Neighboring Statistics

Optional section! This section is used to create a breakdown of how the neighboring is distributed in the model. Nothing from this will be saved by exporting, this is purely for displaying information. The first thing that happens here is that the *positionDistances* list (which is a list of all distances between nearest neighbors) is flattened and converted to angstroms (instead of using the 1 = 15 nm units). Once in angstroms, the exponentially weighted average distance is calculated. If distance dependent hopping is enabled (in the main model) this is the distance that the specified hopping rate will be applied to. For example, if the specified hopping time constant is 40 ns, a hopping distance that is equal to this exponentially weighted average distance will have an effective time constant that is 40 ns while hopping distances that are shorter will have an effective hopping time constant that is <40 ns and hopping distances that are longer will have an effective time constant that is >40 ns. These will be exponentially distributed but the point that is the exponentially weighted average will be the one to receive the specified rate. In order to achieve this distribution, hopping rates are scaled so that once the exponentially scaling is applied, the desired distribution is achieved. This amounts to scaling everything by the inverse of the scaling factor that would be applied to the average. That was very wordy so hopefully an example (with pseudo made up numbers) will clarify:

Say you've got some distribution of hopping distances between nearest neighbors. You also specify that the hopping time constant should be 40 ns. You find that the **exponentially** weighted average of your distribution is 16 angstroms. If you were to apply the exponential scaling to this 16 angstrom distance, the scaling factor for that particular distance would be 10. That means, when the model runs, hops that are 16 angstroms long will have an effective time constant of 400 ns. This isn't correct because you want 40 ns to be the ***arithmetically*** averaged effective time constant. To correct for this, you scale all hopping factors by 1/10. This results in the ***exponential*** average distance having a total scale factor of 1 with an exponential distribution around it. Taking an ***arithmetic*** average at this point therefore also gives you a scale factor of 1. In this way, you specified 40 ns to be the hopping time constant and, on average, that is the hopping time constant, there is just an exponentially distributed series of scale factors that is set to 1 for the ***exponential*** average.

After this distribution is calculated, several things are printed out and several graphs are displayed. The printouts include the arithmetic and exponential average of the distances between neighbors, the closest neighbor (this is important because this is the neighbor that will potentially cause the hopping probability to shoot above 100%) , the scale factor that the closest neighbor receives, the average hopping factor (should always be 1), and the average correction factor (1/10 in the example above).

The first two graphs display the distribution of nearest neighbor distances, first in units of 1 = 15nm and then in angstroms. The third graph is a histogram of the number of nearest neighbors that a site has. Its good if this is a distribution around 6-8 with some tails on either side. It will change based on geometry but should reflect that the nearest ring to a site are included in its nearest neighbors. The final plot is a distribution of scaling factors for

the distance distribution of the system. The reason that it increases at lower values is that these scale factors are applied to the probability to hop rather than the hopping time constant. Notice that the scale factor for the value that is the average value is equal to 1. This is just a graphical display of what math will be applied in the main model and is only diagnostic and does not need to be run before exporting.

Percolation Zone Statistics

Another optional section! This applies the FloodFill[] function discussed above to the surface generated to determine if there are separate percolation zones. It then displays these zones in different colors (often the whole thing will be 1 color because everything is connected). This is done by iterating over each position in the system and checking to see if it has been assigned to a zone. If its still set to percolation zone 0, then a new zone has been found and FloodFill will be called here. So, the first time this happens, the zone will be set to 1. All other zones reached by FloodFill will then also be set to 1. The iteration moves on and if it finds another 0 that means it found a point disconnected from zone 1 and so sets this to zone 2. FloodFill then fills out more zone 2 sites. This goes on until every position has been assigned to a nonzero zone. While this is happening the zone sizes and positions are recorded for displaying graphics. It should be noted that if an individual zone is too large(usually if everything is connected and the surface is large) this will crash due to either exceeding the RAM limitations of the computer or the recursion depth. The recursion depth is set quite high before running this and if that is not enough, it can be increased. After that, the graphic displays and you've reached the end of the code!

Act II – Running the Model

Overview

The second program starts by importing the results from the first program. This is the scaffold which the model will run on. There are a number of user settings that allow for this surface to be used in different ways. Based on these user settings, the stackInfo table (the table which contains all information about every molecular site in the surface will be filled out. Almost immediately after the settings, the program will enter a series of loops which contain everything else. This is because, typically when running the model, we want to run it many times to build up meaningful statistics, and we also want to run with a number of different parameters. So instead of having to run the model, change parameters, run, change, run, change… almost the entire thing is placed inside a set of loops. These loops iterate over hopping and recombination constants so that modeled results under many conditions can be tested. Shortly after the start of these loops, a third loop, the trials loop, will start. This inner loop is simply to repeat conditions over and over and build up statistics.

Once inside the loop, it is time to start initializing the system. This will involve setting the type of each molecular site, determining recombination rates, determining hopping rates, and determining percolation zones. To begin, first vacant spots (sites with no molecule) are set followed by catalysts sites. Every other site is assumed to be a dye location and then a number of these dyes are chosen to be initially excited dyes. All of these choices can be controlled in the settings in terms of the number of each, whether they are randomly chosen over the surface or on a particle by particle basis, and whether light effects such as the Beer-Lambert law, or polarization should be considered when assigning excited dyes. Once all molecules have been set, the various hopping and recombination probabilities must

be set in the stackInfo table for future reference. Various counters are initialized, and molecular sites are sorted into percolation zones then we are ready to begin the main loop.

The main loop is where the simulation actually runs. The main loop has a very simple structure to it: At each timestep, give every charge a chance to do something, adjust the system accordingly, and move on. This typically continues until there are no more oxidized dyes left in the system. If CW mode is enabled then there is also a chance of adding more mobile charges (additional excitation events) to the system in which case the end condition is simply a number of timesteps having passed. The choices given to each charge are to recombine, hop to a neighboring molecule, or do nothing. A charge, in this case, is referring to an electron hole either on a dye or catalyst. If a catalysts is maximally oxidized, instead of being given a choice, it instead turns over (performs some chemistry). While all of these things are happening various parameters are tabulated for analysis later. Once the main loop is complete the program is essentially done. Some exporting of data files occurs amongst and after the ending of the other loops to output data of interest and then there are some extra optional cells for displaying graphics and making videos and stuff. The end!

Detailed Walk-Through

Importing

The first thing that happens is importing the surface and data from the previous program. This essentially lets us keep using variables from the surface as if we had just finished running it. This is nice because, often we want to run 10-50 simulations on the same surface and this way we know we are starting from the same point each time. We also save time not having to prepare the surface each time. Most variables here use the same names as when exporting previously.

Functions

There are 7 functions here used as support functions of which 6 are super straight forward.

probFromTau[], when given a time constant (tau) will return a probability that the event

associated with the time constant will happen in the span of the timestepsize of the model. For example, if the time constant for hopping is 40 ns that means on average there should be a hop every 40 ns. If the model's timestep size (calculated later) happens to be 1 ns, then there should be about 1/40 of a hop every timestep or else the chance for a hop to occur in 1 timestep should be 1/40.

dipoleOverlapFunc[] is a function that takes the relative angle between the electric dipole moment of a dye and the orientation of the electric field of incident polarized light and returns a value based on how well they overlap.

degeneracyOverlapFunc[] is a function that takes the same parameter as the dipoleOverlapFunc[] but instead returns a value based on the degeneracy of this angle over the surface of a sphere. *This is no longer needed as the spherically distributed positions no longer need to be corrected for angle degeneracy over a sphere.*

probAbsInit[] is now just the absolute value of the dipole overlap function. In previous iterations of this model it was necessary to also correct for degeneracy of points on a sphere (there are more points around the sphere at the equator than near the poles) but this is no longer the case.

lightIntensityBeersLaw[] takes a molecular depth fraction (the percent of the way through the film a given molecule is with the top molecule being 0% and the bottom molecule

being 100%) and provides a Beer's Law weighting for excitation. This weighting, if used in settings, will be applied when determining which dyes absorb incident light. The idea here is that when light is incident on a film(surface) some of the light is absorbed as it travels through the film. Particles lower down/further from the light source are receiving less light than those right on the surface. Therefore, molecules nearer to the light source are more likely to absorb light than those further down. Beer's Law (The Beer-Lambert Law) creates and exponential decay from 100% intensity to a set value specified by the percent light transmitted out the back. Experimentally, this value could be calculated by measuring the absorbance of a film. Presently, the light transmitted is always set to a fixed amount such that the molecule furthest from the light source receives a percent of light determined by fracTrans that the topmost molecule does.

distanceBetweenMolecules[] is exactly the same as pythag[] in the previous program but with a long name for no real reason. It applies the Pythagorean theorem to find the distance between two real points in 3D space.

FloodFill[] is a doozy for sure. Fortunately, I've already written this out once so rather than explain it again I can just refer you to FloodFill from the previous program.

Settings

There are a number of settings to be aware of while running this model. The first several are booleans followed by a couple of integers and then the settings for each molecule type.

absBLLaw, if set to True will use the Beer-Lambert Law to weight the assignment probability for the excitation of dyes according to lightIntensityBeersLaw[]. If set to false, all positions are weighted equally.

absAnisotropy, if set to True will make use of probAbsInit[] when weighting assignment probabilities for excitation of dye. This will simulate the use of polarized light which will favorably excite dyes whose electric dipole created by absorbing a photon is aligned with the direction of polarization of the electric field of the incoming light. Effectively this means that the tops and bottoms (the poles) of each particle will be more favorably excited than other locations. This creates areas of high concentration and low concentration in terms of excited dyes which then, by diffusion become homogenous. By watching the polarization anisotropy decay, one can back out the effective diffusion coefficient of the self-exchange hole transfer process.

CWmode, If enabled, will allow additional dyes to be excited over time as the system is Continuously Illuminated. If CWmode is enabled, the end condition will use the specified number of timesteps instead of waiting to eliminate all excited dyes. Also, it is likely that the number of initially excited number of dyes should be set to zero.

DistanceDependentHopping will change the effective hopping constants between nearest neighbors based on the distance between them. The arithmetic average of all hopping constants in the system will be the setting hopping constant. If False, all hopping constants will be the same regardless of distance.

ElectronSpreading, if enabled will blur out the volume of the surface that an injected electron is said to be in. If False, there is considered to be one injected electron in the volume of a given particle for every hole currently on a molecule on that particle's surface. If True,

each injected electron is “distributed” over several particles starting from an electron hole and taking relative particle size and neighboring into consideration. Doesn’t do anything in systems for disconnected particles. *This is entirely fabricated and is an approximation of a correction that could be made to distribute electron density. This has no basis in physics but is still probably better than nothing even if hard to justify in a paper.*

ElectronAreaScaling, if enabled will scale the effective electron density by a particle’s relative size. This makes the second order recombination behavior more “fair” when particles of different sizes are used. 1 electron-hole pair on a 10 nm particle and 1 electron-hole pair on a 30 nm shouldn’t have the same recombination probabilities. Doesn’t do anything for systems with equally size particles. *This is fabricated and has no basis in physics other than that this seems like a reasonable adjustment to make.*

electronDistributionHomogenized, if enabled, will assume that all electrons in the film are moving so quickly so as to all be effectively everywhere. This sets the electron density to be equal everywhere and sets it to be the sum of the total electrons in the film divided by the number of particles in the film.

maxOxState is the maximum oxidation state of catalysts to be use in this system. Typically set to 2 or 4. Once a catalyst has acquired this many charges, it will turnover (perform chemistry) and those charges will be removed from the system. This turnover occurs at the timestep following the one in which it was filled.

numTrials is the number of times to repeat the inner loop to build up statistics. 25 times is a good number to use for most things. If making a video of hopping, only 1 trial is needed. 2-5 times is a good number to use when testing new code.

Absorbance and *fracTrans* are only relevant if a Beer's Law distribution is used. If used, the Beer's Law exponential will decay down to the value of *fracTrans*. Absorbance is an alternative way to specify this if that is easier. If specifying *absorbance* rather than *fracTrans* directly, *fracTrans* should be set to 10^(-*absorbance*)

Dead Spot Settings! Dead spots are molecular sites on the surface which have no molecules. These cannot be hopped to and can simulate lower coverage dying of film or be used to see when percolation networks break down. *DSFixed* which, if true uses *DSperParticle* and if False uses *pctDS*. If using DSperParticle, a set number of dead spots will be selected for each particle and exactly that number will be used. If pctDS is used, a certain percentage of molecular sites over the whole surface will be chosen to be dead spots. This will be completely random.

Catalyst Setting operate almost exactly like dead spot settings! If *CatFixed* is True, a number (*CatperParticle*) of catalysts will be placed on each particle in the system. Otherwise, a percent (*pctCats*) of molecular sites in the system will be occupied by catalysts. 1 percent is about optimal under many conditions.

Dye Settings operate much the same way as dead spots and catalysts. If *DyesFixed* is True then a number (*DyesperParticle*) will be excited on each particle while if it is False then a number (*DyePerFilm*) will be excited over the whole film. It should be noted that all sites which are not dead spots or catalysts are, by default, dye location and that this does not affect that. This affects the number of *Initially Excited Dyes*. If UsePercent is set to True than the percent specified in *PctExcitedDyes* will overwrite the *DyesPerFilm* number. If using CWmode, starting with 0 or 1 dyes is usually good.

RecTurnovers, RecTimeBehavior, and *RecAnis* are used to specify whether to record (and export) behavior about the Turnover yield, species behavior over time, and anisotropy over time respectively.

RecHopPaths should only be used for smaller surfaces (<25 particles) unless you're using a powerful computer and willing to wait a long time. If enabled, this setting will keep a running list of every location that an electron hole has traveled to. This is useful for mapping out the movement of holes across a surface and for making videos of such. Note that this records every time a hole hops to a new location, not where it is at every timestep. Therefore, a playback of multiple holes hopping across the surface will not show the system over time but rather over sequential locations for each hole. This means that if a Dye 1 hops from A to B to C to D in the first 4 timesteps while dye 2 hops from W to X to Y to Z over the course of 50 timesteps, the two will appear to be hopping at the same rate.

Part is a quick way to separate two sets of hopping and recombination values wished to be run. By default, if an array of hopping and recombination values are both specified all combinations of those arrays are run. Often it is better to specify two subsets of such arrays and run one subset as part 1 and another as part 2. Other subsets could similarly be specified.

KHOP and KRECOMB should really be called τ_{hop} and $\tau_{recombine}$. These are the lists of time constants to iterate over in two external loops. So if 5 different hopping values and 3 different recombination values are specified, the system will run the simulation 15 times, using each pair once. Each of these times will be repeated a number of times equal to numTrials. Notice that these are specified in nanoseconds and the end of the array is multiplied by 10^{-9} so to specify a hopping rate of 40 ns, you would simply include 40 in the array, not 40 x 10^{-9}.

Initialization

This is a complicated model and therefore there a many things that need to be initialized. For many of them, the order in not important. For some it is. So, in the order that the code is currently in here are the things set up in this section. To begin some geometry parameters are identified. These could realistically be moved into the surface generation program, but they aren't computationally intensive, so it hasn't been worth it to move them yet. First the minimum and maximum height values are identified, and the total thickness is identified so that, if Beer's Law is used, molecular positions can be given a proper Beer's Law weighting. Additionally, the Particle sizes and particle regions (the volume over which an electron could be shared over) are determined so that if ElectronSpreading and ElectronAreaScaling are used the math is mostly done. Also, the effective molecular radius is specified here and for consistency should be left the same as it was by the surface generator. Another list created is the *oxList* which is a list of all the current oxidation state of every molecule on the surface.

Next the number of (excited) dyes and catalysts are determined based on whether they are to be distributed on a particle-by-particle basis or over the whole surface.

Next a number of empty tables are created to hold data. allHops will store a list of every hopping distance that was hopped to later backout an observed hopping constant. timeDecays will store the number of dyes remaining over time. CompTable is a table that will hold the Compiled Data for exporting after iterating over all hopping and recombination constants. ParameterPoint is a counter that counts what step of the time constant loops the simulation has reached so that data (which must be stored linearly) can be appropriately stored while looping over a 2-D grid).

Next, the time constant loops begin and the time constants are specified. The HRR (hopping-recombination ratio which is actually the recombination-hopping ratio) is determined. These values are placed in the header rows of the Compilation Table. More tables are initialized to store the turnovers per particle, catalysts reached, and total number of turnovers. More Counters are initialized to store the different potential automatic recombination cases. These count the times that the simulation has, automatically recombined electron-holes because they had no chance to contribute to catalyst turnover. Tables for the number of dyes remaining over time, a list of hopping distances, tracing parameters (if needed), and counters for the number of catalysts reached and number of catalysts turned over.

Next, many time constants are dealt with. The timestep size is calculated by finding the minimum size of effect we'd hope to see and dividing by 350. This should ensure that most processes in the simulation will have a small (~1%) chance to occur on any given timestep. This is to ensure that the probabilities are roughly constant timestep to timestep and that few events are missed. Following this, all other hopping constants are set. Although, in all experiments so far all recombination constants have remained the same, there is the option to provide different recombination constants for each oxidation state of a catalyst. If the max oxidation state is going to be greater than 4, additional rates would have to be appended to the tauRecombCatRates Table. Additionally, each catalyst can have two different recombination rates. One for fast recombination and one for slow. This is to roughly mimic the stretched exponential behavior seen by experimentalists. Each catalyst will be given randomly one of the two rates provided for each oxidation state. The weighting of which rate is given is determined by the population fraction. So, if popFrac1A = 0.38 38% of

catalysts sites will be given the time constant specified in tauRecombCatRates[[1, 1]] and 62% will be given the time constant specified by tauRecombCatRates[[1, 2]].

Next there are all of the time constants (taus) are converted to probabilities. This is essentially just dividing the time step size by each time constant and reflects the probability each of these things has to occur over the course of one timestep.

Finally, there are a few more counters that are initialized that keep track of the number of dyes left, the number of recombination events that have occurred from dyes, the number of non oxidized catalysts left, the number of recombination events that have occurred from each oxidation state of catalyst, the number of each oxidation state of catalyst at any given time, the initial and final charge distributions, the runStress(a running list of the number of items in the POI list at each timestep), the runTimes (a running list of how long it takes to complete each timestep), and the anisotropy which will be used as a place to store the polarization anisotropy value calculated each round.

Assigning Positions

The assignment of molecule positions first assigns dead spots, the catalysts and finally, determines which of the remaining dyes will be initially excited. To do this a few tables must be created. First a table to store all the relative likelihoods for a dye to absorb a photon is created. Initially every site is given the same weight but if Beer's Law is used or polarized light is used these will be changed later. Next is a table containing all possible choices for molecules. This table is in the format of (particle #, position #). Flattened Positions is the same list of coordinates but in a 1-D list instead of a 2-D list. Depending on whether molecules are to be chosen on a particle by particle basis or over the whole surface at once, having these possible choices in two different lists is very helpful.

Next the system is reset. That is, all possible positions are reset to be dyes. This is done so that each iteration through the trials loop or through each of the time constant loops allows for molecular arrangements to differ.

First to be assigned are dead spots. To start an empty list of chosen sites is created to hold future choices. Then, if dead spots are to be assigned on a particle by particle basis, the correct number of dead spots are chosen on each particle and appended to the list of choices. If dead spots are to be assigned over the whole film, the flattened positions list is used so that all possibilities are available at once. Once all choices have been made iterate over all the choices setting the type of the molecule to 3 in the stackInfo Table, the choice weighting to 0 (can't have these positions again) and mark down the positions remaining.

Next up is catalyst assignment which functions pretty much the same way as dead spot assignment. An empty array of catalyst choices is created and then depending on whether the number of catalysts is meant to be fixed over the surface or per particle, a number of selections are made and appended to the list. Once the array of catalyst choices are made, the stackInfo table is updated and the sites are made unavailable for future selection. Additionally, recombination rates are chosen and stored in the stackInfo table based on population fractions and time constants specified in initialization.

Finally, setting the (excited) dyes is a little different. First, each molecule has a molecular depth fraction calculated. This will be used to assign weighting if Beer's Law is used. Then an empty array for dye selections is created. Next, the choice of dye excitations must be weighted if Beer's Law or polarized light are used. For each of these modifications, the governing function (lightIntensityBeer'sLaw or probAbsInit) are mapped over all positions and the resulting matrix is multiplied against the choice weighting matrix. Once the

weighting matrices are taken care of, the selection proceeds the same as dead spots or catalysts. Dyes are chosen, using the determined weights, either by particle or over the whole surface. Once chosen, the oxList table is updated to reflect that these dyes are now oxidized.

Determining Percolation Zones

As discussed in the surface generation section previously, sorting molecular sites into percolation zones can be important to make sure simulations run smoothly. This is only really the case if there are multiple zones (if there are separate particles, sites isolated by dead spots, etc.). Determining percolation zones works the same way it did in the surface generation section. To start all positions are set to zone 0. Then Floodfill is called on the first position and it (along with all mutually connected positions are set to zone 1. Once the first all of Floodfill has run its course, the model iterates through all remaining positions to see if any are still set to zone 0. If so, that position must be in a different percolation zone than the first call and so Floodfill is called again starting from that new position and starts assigning positions in contact with it to zone 2. This continues until every position is assigned to a non-zero zone.

Once every position has had a zone number assigned to it, it is necessary to sort the positions into zones. Basically, we need to know what positions are in each zone not just what zone each position belongs to. Once positions are sorted into zones it is possible to count the number of catalysts and excitations in each zone. This is done after setting the hopping values but will be discussed here. This next step is to identify “hopeless excitations”. This is probably not the best name for these but these are essentially excitations which we know, from timestep 0, will never contribute to turnover. There are two reasons this may occur. The first way this happens is when excitations occur and have no catalysts in their

percolation zones and so will eventually recombine. The other way this happens is if there are not enough excitations in a given zone (even if there are catalysts) to turnover a catalyst. For example, if it takes 4 charges to turn over a catalysts and there are only 2 excitations that occur in a given zone, we know before things start that those excitations will never contribute. All excitations in a given zone are going to be hopeless or not so we can also identify hopeless zones as those containing hopeless excitations.

Identifying hopeless excitations is important because the simulation can be terminated once it has been reduced to only hopeless excitations. This can be identified as the time at which excited states only exist in hopeless zones. Often, this will be the time at which there are no more excited dyes at all and so all of this is unnecessary. However, in the case that there is an excited dye which has no possible catalysts to reach, the simulation would potentially continue on for hours or days waiting for this single dye to recombine. This is only a problem if the hopping rate is very fast and the recombination rate is very slow meaning that the expected number of timesteps to recombination can be extremely large. So, on the off chance that this is the case, its worth it to do a bit of this pre-sorting ahead of time so that a simulation that should last 30 second doesn't instead last 30 hours.

Setting Hopping Values

There are two things to consider when assigning hopping values: the number of dye and catalyst neighbors that a position has, and (potentially) the distance between each pair of neighbors. This latter consideration is only taken into account if distance dependent hopping is turned on. This process is done by iterating through each molecule in the system and filling in the stackInfo table appropriately. If a position contains a dye, it is given a dye

recombination rate and its list of hopping probabilities (scaled by a distance factor or not) are created appropriately neighbor by neighbor as each is identified as a dye or a catalyst. The same is true for catalysts with assignments of analogous recombination and hopping rates. All of these rates are stored in the stackInfo table for lookup in the main loop.

A few other minor things are taken care of at this point and throughout this section. A POI (point of interest (yes, another poor name choice)) list is created and randomly sorted. If traceRecord is turned on, the initial positions of every dye are stored so animations can be created later.

Next up is the main loop! Everything up to this point should run in a few seconds but the bulk of the program is up next.

Main Loop – Timestep Checks

The first part of every timestep is to see if the main loop should end. This is done by checking a variable called *EndCondition* which can be made true in two ways. The primary way this happens is when there are no more dyes in the POI list in non-hopeless (this is why I shouldn't name things) zones. This occurs when either there no more dyes remaining excited at all or when all remaining dyes are in hopeless zones (and therefore will eventually recombine). The second way *EndCondition* can be triggered is if CWmode is on and a certain number of timesteps has past. The reason the CWmode needs a separate condition is that the simulation starts at, and may often have, no excited dyes in the stack so the whole thing would end as soon as it began.

Other than checking for the end, there is a status update which occurs every 1000 timesteps and prints out (to the console) various statistics of interest so that we can continuously make sure things are running smoothly.

Main Loop – Bookkeeping

Bookkeeping consists of recording everything as specified by the settings. First up on this is to record the anisotropy value for the timestep. This is simply the sum of all anisotropy contributions of all molecules in the POI list scaled by the number of molecules in that list. This average anisotropy contribution should, for an ideal system start at 0.4 and decay down to 0 but depending on geometric considerations this may not always be the case. More info here:

https://en.wikipedia.org/wiki/Fluorescence_anisotropy

One of the next things to take care of is determining where the electrons are. Electrons are not handled explicitly in the model but they are the source of recombination, so we still need to know generally where they are. The primary assumption here is that when a dye is excited, an electron-hole pair is created after which the electron injects into the TiO_2 leaving a hole on the dye. The whole can hop around and is handled by the model but the electron is just assumed to be somewhere in the volume of the particle the hole is at the surface of. So a particle with 5 excited dyes/oxidized dyes on its surface is assumed to have 5 electrons within its volume. This is true even if all of holes on the surface didn't originally start out on this particular particle. So a hole which hops from dye to dye and ends up switching particles ends up bringing its electron along for the ride. This isn't terribly rigorous but as a rough approximation, its not terrible either. The point of all this is that at the start of each timestep we need to count up the number of holes on the surface of each particle, so we know how

many electrons are within each particle. This electron count can be modified in 2 ways both of which are completely fabricated and not rigorous so are typically not used but also seem like reasonable approximations. The first way is with *ElectronSpreading* which, if turned on allows some of the electron density in a particle to leak into neighboring particles. This is done by taking the total number of electrons in a given particle and distributing them across the given particle and all of its neighbors with the given particle being given a double weight compared to the others. These distributions are also weighted by a particles relative size (surface area being used as "size") so that a larger neighbor will take more of the distribution than a smaller neighbor. The other way to modify the electron count is with *AreaScaling*. This simply divides the electron density by the size (surface area) of a particle relative to a radius 1 particle. 1 electron in a radius 30 nm particle provides a lot more electron density than 1 electron in a 10 nm particle. Neither of these are physically rigorous but seem reasonable all the same. The reason we need to know the number of electrons (or electron density) per particle is that molecule recombination rates will be scaled by the number of electrons there are to recombine with.

The last thing to take care of in the bookkeeping step is to record the time behavior if specified. This is simply adding a row to a very large table which tabulates the various phenomenon at each timestep. This can get very large if running an experiment for a long time so it may be necessary under certain conditions to put this block inside a If[timestep%5==0,…]; statement (only perform this every 5 timesteps).

Main Loop – Iterating the POI List

This is really the main body of the whole thing. The overarching idea is that at every timestep, every molecule with a hole (every item on the POI list) will get a chance to have

that hole do something. That something can be recombine, hole-hop to a neighboring molecule, or do nothing. This is determined by a probabilistic choice determined by the hole's environment. So, for each element of the POI list we first need to pull all relevant information. The element being looked at is usually referred to as the "point" while the line of the stackInfo table which contains all of its information is referred to as the "position". This is done so it is faster (and shorter to read) to reference this information in the upcoming section.

The first thing that needs to happen for each molecule is that we need to decide what type of molecule we're dealing with. This is the first branching *Which* statement. These basically function as if else statements in other languages except are harder to read. If something is determined to be a dye, the next step is to pull all relevant information needed to construct the choice for that dye. That means pulling the recombination probability from the stackInfo table, pulling the list of nearest neighbors, and pulling the hopping probabilities between those nearest neighbors. Once this information is ready, a weighted choice is made. If the total probability of this weighted choice would exceed 100%, the probabilities are normalize such that they sum to exactly 100% and the option to do nothing is removed. This weighted choice consists of 3 real options: hop, recombine, or do nothing. The choice is constructed as a list which is equal to the number of neighbors a molecule has +2. If a molecule has X nearest neighbors and choice from 1-X will represent choosing to hop to that neighbor while a choice of X+1 will represent choosing to recombine and a choice of X+2 will represent a choice to do nothing.

A catalyst's choices are run the same way with the exception that if a catalyst starts at the maximum oxidation state, it will instead turnover and its choice will automatically be to do nothing for that round.

After a choice is made it has to be validated or else a new choice must be made. A valid choice is any choice to recombine, to do nothing, or to hop to a molecule which has not been maximally oxidized. This means that the only way for a choice to be invalid would be if the choice was made to hop to a catalyst that was already filled or to hop to an already oxidized dye. If this is the case, the choice is rejected and is taken again.

Once a choice is validated, it must be implemented. This is essentially just more bookkeeping and keeping track of what goes where. If the choice to hop, the relevant information for the hop target is pulled from the stackInfo table. Both the source and target molecules must have their oxidation states changed and the element on the POI list must be updated so that the target is now on the list. Additionally, if *traceRecord* is True, then the hop is recorded for purposes of creating animations later. If the choice is to recombine, oxidation states are updated and the element (if completely empty) if removed from the POI. The counter which steps forward is also decremented by one to account for the fact that an element was dropped from the list.

The only other real consideration in resolving these choices is that when a catalyst's oxidation state changes, the hopping probabilities of all neighboring molecules must be updated. This is because the model allows for different hopping rates to be used for an empty catalyst vs a 1^{st} oxidized catalyst vs a 2^{nd} oxidized catalyst. When a catalyst's oxidation state changes all of its neighbors are looked up and the hopping probability from them to the given

catalyst is updated to reflect this. This gets especially complicated in the unlikely event that there is a hopping even which occurs between two catalysts.

It should also be noted that a catalyst with multiple holes on it only receives one entry on the POI list and therefore only one "turn". For this reason, when a hole on a dye hops to a catalyst with at least one charge already, the entry on the POI is deleted rather than being reassigned.

Other than updating the status of the POI and the stackInfo table, various statistics are tabulated (recombinations, turnovers, etc. as they occur).

Main Loop – CW

In the case that an experiment is being run in CW mode (continuous illumination) there is also a chance that a new excitation is added to the system at each timestep. For no real reason at all, this is handled at the end of each timestep instead of with all the bookkeeping up front. First a random number is compared with the probability that an excitation occurs. If the random number is smaller than the excitation probability, a new excitation must be assigned to the system. This works the same way as previous assignments except that currently occupied sites must be taken into account and weighted as 0 in the weighted choice of where to place the excitation. Once chose, its appended to the POI list and its time to move on to the next time step.

Recording Data

The final section of code that runs in the main block is the part after the main loop which deals with exporting data in three different files (if specified). The first thing that happens after the main look is a check to see if the last trial of the main loop was the shortest. This is relevant because the data from the trials are averaged together but each trials will

not necessarily take the same number of timesteps. If one trial takes 5000 timesteps and the next one takes 8000, the arrays of data can't be averaged together until we make them the same size. We can do this in 2 ways: Make everything as long as the longest trial or make everything as short as the shortest trial. If we wanted to extend all of our short data to make it longer, we would need to fill in the ends with something which is a problem because we don't always have something appropriate to fill in there. For the number of dyes remaining we could easily fill in all zeros for all future timesteps. For number of recombinations however, we can't just tack on a bunch of zeros or even a bunch of whatever the last value was and be able to rely on that. So, for sake of simplicity and accuracy and to avoid having to determine the most accurate way to fabricate anything, we instead truncate data to match the shortest trial. This ensures that everything we have, even if less than we originally collected, is real. In order to do that we need to keep track of the shortest trial.

If turnovers are set to be recorded, the turnovers are averaged over the number of trials and then stored in a large table organized by hop and recombination rates. Both the number and percent of turnovers are stored. Following the end of the trials loop, but within the time constant loops, this table is exported. The table is exported after each pass through one of the time constant loops (overwriting the previous one each time) so that if the program crashes (or if a windows update restarts the computer ☹) the latest possible version of this table will have been saved.

If anisotropy and time behavior data are requested, those (which have been collected and organized during the bookkeeping step in the main loop) are truncated (down the to shortest trial) and added to the corresponding tables. These are also exported after each pass of the time constant loops but instead of overwriting each time, the parameter point is

appended to the filename to we end up getting *timebevaiorfile_1, timebevaiorfile_2, timebevaiorfile_3… etc.* which, while not elegant, is the only reliable way to save a set of potentially very large files without risking a memory overload. Ideally these would export at one multitabbed excel file but if the experiment takes more than a few thousand steps and there are many parameter points, Mathematica will run out of memory trying to save that monster file. So, a bunch of small files it is! They're kind of annoying to handle afterward but it only takes a short MATLAB script to stich them together as needed.

Following all the exporting, a simulation summary is printed out with a final sanity check to see how believable the data will be.

The End! Congrats on making it this far in this crazy mess of a program! Only one thing left and its totally optional and ancillary to the actual function.

Creating Videos

When making videos only one trial and one set of hopping and recombination constants needs to be used. This is because the animation blocks of code will pull the list of hops from the last trial to make the animation. It should also be noted that a very long experiment will result in a very long animation and possibly an insurmountably large export.

The first block of code in this section doesn't actually create an animation but rather a still image showing what is essentially the last frame of a potential animation. This is to quickly see if its worth trying to export by seeing if the catalysts end up on a side of the image that will be shown. It should also be noted that this image is rotatable like most Mathematica 3D images and that, like most Mathematica 3D images, will preserve that rotation upon being recreated. That is, if you create an image, spin it around to see the back, rerun an experiment, and then make a new image, the new image may also be turned around. Therefore, if you do

decide to rotate the still image created, be sure to delete that output cell once you are done so that Mathematica is forced to reset the rotation upon re-outputting. This is important because this default rotation will be the perspective that the animation will be in. This is, in theory, changeable but is way too much work and is much better just to run your quick animation experiment 50 time to find one that looks good from the front.

The way the still image works, and subsequent animations, is that each hole is given a color and a series of spheres in that color are created at every point that hole occupied according to the *DyeTraces* table. This table is a list of {particle #, position # coordinates} and so to create the actual spheres, the molecular positions must be looked up using these coordinates. All of these colored spheres are overlaid on an image containing all molecular spheres in yellow. Because these images involve a very large number of spheres, it is important to keep the surfaces used in these animation experiments small (5-10 particles max) if possible. Mathematica really doesn't like trying to animate 30,000 spheres in each frame.

The animations come in 4 types which are composed of two choices with two options each. Option 1 (which determines which block of code to use next) is whether you want an animation which follows the path of a single hole or animates all of them. Option 2 is whether or not you want to show a hole's hopping history or just its current location. This is controlled by simply commenting out the history bits as needed.

Making the animations works very similarly to making the still image. To start, the background is the starting layer of yellow molecules over the whole surface. After that hole positions are added on top as different colored spheres. For each frame, that index of the *DyeTraces* is used. So in frame 3, the third location that Dye#1,Dye#2,Dye#3,Dye#4,…etc.

was in is used as the location of that dye's sphere. Additionally, if histories were turned on, indexes up to that point are used as location for slightly smaller spheres of the same color.

Once created, there is not a great way to view the animation in Mathematica that doesn't take just as long as exporting and watching the video, so the next step is to export a video and see if it looks okay. The exporting process will take a long time (possibly a couple hours for a 2-3 minute animation) and the file size will be obnoxiously large. Unfortunately, I have not found a better export method or file type to use that Mathematica can handle and will playback on a PC. Making these animations is very much a trial and error process until you get something that looks good to you.

APPENDIX B. Model in Mathematica
Surface Generation

```
	startTime=AbsoluteTime[]; (*sets starting time for reference later*)

(*SUPPORTING FUNCTIONS*)
pythag[p1_,p2_]:=((p1[[1]]-p2[[1]])^2+(p1[[2]]-p2[[2]])^2+(p1[[3]]-p2[[3]])^2)^0.5; (*distance formula*)

FloodFill[part_,pos_]:=( (*used to find percolation zones*) (*only used in troubleshooting*)
 If[(stackInfo[[part,pos]][[1]]!=3)&&(percolationZone[[part,pos]]==0),
	percolationZone[[part,pos]]=zoneCount;
				If[Length[stackInfo[[part,pos]][[2]]]>=1,

	FloodFill[stackInfo[[part,pos]][[2]][[1]][[1]],stackInfo[[part,pos]][[2]][[1]][[2]]];
					If[Length[stackInfo[[part,pos]][[2]]]>=2,

	FloodFill[stackInfo[[part,pos]][[2]][[2]][[1]],stackInfo[[part,pos]][[2]][[2]][[2]]];
						If[Length[stackInfo[[part,pos]][[2]]]>=3,

	FloodFill[stackInfo[[part,pos]][[2]][[3]][[1]],stackInfo[[part,pos]][[2]][[3]][[2]]];
							If[Length[stackInfo[[part,pos]][[2]]]>=4,

	FloodFill[stackInfo[[part,pos]][[2]][[4]][[1]],stackInfo[[part,pos]][[2]][[4]][[2]]];
								If[Length[stackInfo[[part,pos]][[2]]]>=5,

	FloodFill[stackInfo[[part,pos]][[2]][[5]][[1]],stackInfo[[part,pos]][[2]][[5]][[2]]];
									If[Length[stackInfo[[part,pos]][[2]]]>=6,

	FloodFill[stackInfo[[part,pos]][[2]][[6]][[1]],stackInfo[[part,pos]][[2]][[6]][[2]]];

	If[Length[stackInfo[[part,pos]][[2]]]>=7,

	FloodFill[stackInfo[[part,pos]][[2]][[7]][[1]],stackInfo[[part,pos]][[2]][[7]][[2]]];

	If[Length[stackInfo[[part,pos]][[2]]]>=8,

	FloodFill[stackInfo[[part,pos]][[2]][[8]][[1]],stackInfo[[part,pos]][[2]][[8]][[2]]];
];
];
];
];
];
];
];
];

	];
 )
(*-----------------------------------------------------------------------------------------------------------------------------------------------*)
(*-----------------------------------------------------------------------------------------------------------------------------------------------*)
(*-----------------------------------------------------------------------------------------------------------------------------------------------*)
(*Stack Settings - No need to change anything outside of this section!*)
stackName = "10PartClusterForVid.wdx"; (*If the stack looks good, remember to export at the end. This is the
name that will be used.*)
```

```
r1positions = 250; (*The number of positions that will be spread over a particle with radius 1, other sizes particles will have position #s scaled*) (*A distance of 1 = 15 nm*)
moleculeRadius= 0.05; (*analogous to the van der Waals radius of a molecule*)
neckingMin= 0.25; (*minimum particle necking value*) (*Necking 0 is exactly touching. non zero values indicate the % of the smaller radius that is overlapped*)
neckingMax = 0.25; (*0.25 is a realistic value*)
partRadMin = 1.0; (*minimum radius possible for a particle*)
partRadMax =1.0; (*maximum radius possible for a particle*) (*A distance of 1 = 15 nm*)
numParticles = 10; (*How many particles will be in the surface created, if using more than 15, individual positions will not be plotted*)
FIB=True; (*True = use Fibonacci spirals to place points over each particle. SETTING FALSE MEANS Tessellation will be used instead, only allows 252 points per particle*)
stack = True; (*If true, particles will form a single column rather than a cluster*)
ClusterCompactness = 0; (*0 is totally random, positive make a tighter spherical cluster, negative makes a long dendritic cluster*)
reach = 7.0; (*multiplier which determines how many molecular radii limit being a nearest neighbor*)
(*----------*)
(*----------*)
(*----------*)
(*SETTING PARTICLES*)
partCenters = {{0,0,0}};  (*Initial particle center at zero. Table to store all particle centers*)
particleRadii = Table[1,{i,numParticles}]; (*A table to store the radii of al particles*)
particleRadii[[1]] = RandomReal[{partRadMin,partRadMax}]; (*initial particle radius*)
(*Iterates over all particles and attempts to find a place for them. This is done by choosing an existing particle, a random new particle size, a random direction from that existing particle and a valid necking  with that existing particle.
Then the new particle is checked against all existing particles to make sure it is not closer than the required by the maximum necking.
If valid, it is added to the stack, otherwise new random choices for all parameters are chosen*)
For[x=2,x<=numParticles,x++,
        validChoice = False;
        While[validChoice==False,
                centerChoice = RandomChoice[Reverse[Range[Length[partCenters]]]^ClusterCompactness->Range[Length[partCenters]]]; (*Weighted choice for particle to build off of*)
                currentCenter = partCenters[[centerChoice]];
                size = RandomReal[{partRadMin,partRadMax}];
                neck = size*RandomReal[{neckingMin,neckingMax}];
                If[stack==True, (*if a stack is to be formed, the next particle is always directly along the z-axis, otherwise choose a random point on a sphere*)
                        unit = {0,0,1};
                ,
                        unit = RandomPoint[Sphere[]];  (*Sphere[] function accounts for angular degeneracy when choosing points*)
                ];
                offset = Max[particleRadii[[centerChoice]]+(1-neck)*size,particleRadii[[centerChoice]]*(1-neck)+size]; (*the distance between the old and new particle centers while counting the neckign on the smaller of the two*)
                newCenter = currentCenter+offset*unit;
                validChoice=True;
                For[i=1,i<=Length[partCenters],i++,
                        dist =((newCenter[[1]]-partCenters[[i]][[1]])^2+(newCenter[[2]]-partCenters[[i]][[2]])^2+(newCenter[[3]]-partCenters[[i]][[3]])^2)^(0.5); (*finds the distance to all other particles*)
```

```
                        minC2CDist = Max[particleRadii[[i]]+(1-neckingMax)*size,particleRadii[[i]]*(1-
neckingMax)+size]; (*the minimum distance two particles can be to one another, with different particle sizes,
necking maximums are based on the smaller of the particles*)
                        If[dist<minC2CDist,
                                validChoice=False;
                        ];
                        ];
        ];
        partCenters = Append[partCenters,newCenter] ;(*stores the particle center to the list of centers*)
        particleRadii[[x]]=size; (*stores the particles radii in the list of particle radii*)
 ];
(*-------------------------------------------------------------------------------------------------------------------------------*)
(*-------------------------------------------------------------------------------------------------------------------------------*)
(*-------------------------------------------------------------------------------------------------------------------------------*)
(*SETTING POSITIONS*)
If[FIB==True, (*if using FIB spirals, scales the number of positions by the particle size. Otherwise 252 per
particle must be used*)
        positionsPerParticle = Table[Round[r1positions*particleRadii[[i]]^2],{i,1,numParticles}];
(*calculates the number of molecular positions per particle based on radius*)
 ,
        positionsPerParticle = Table[252,{i,1,numParticles}];
        Needs["PolyhedronOperations`"];
        psub =Round[Geodesate[PolyhedronData["Icosahedron", "Faces"],5][[1]][[14;;265]],0.0001];  (*Uses
icosahedrons to get particle positions if not using FIB spirals*)
 ];

positions = Table[0,{i,1,numParticles}]; (*Creates a table to hold all molecular positions as XYZ coordinates*)
rotatedPositions = Table[0,{i,1,numParticles}]; (*Creates a table to hold all molecular positions after they
have undergone two random rotations as XYZ coordinates*)
goldenAngle = Pi*(3.0-Sqrt[5.0]); (*stores the golden angle for use in math below*)

For[part=1,part<=numParticles,part++, (*iterates over all particles in the surface*)
        n = positionsPerParticle[[part]]; (*pulls the number of positions for the current particle*)
        c = partCenters[[part]]; (*pulls the particle center from the current particle*)
        rad = particleRadii[[part]];(*pulls the particle radii from the current particle*)
        If[FIB==True, (*if Fibonacci spirals are used, calculates xyz coordinates for the current particle based
on n,c,rad and the Golden Angle*)
                rho = Table[goldenAngle*i,{i,0,n}];
                z = Table[(1-1.0/n)-(2.0/n)*i,{i,0,n-1}];
                r = Table[Sqrt[1-z[[i]]^2],{i,1,n}];
                x = Table[r[[i]]*Cos[rho[[i]]],{i,1,n}];
                y = Table[r[[i]]*Sin[rho[[i]]],{i,1,n}];
                positions[[part]] = Table[{x[[i]],y[[i]],z[[i]]},{i,1,n}]; (*stores the positions calculated in the
positions table*)
                ,
                positions[[part]] = psub; (*if Fibonacci spirals are not used, all particles use the same
icosahedron tessellation position list*)
        ];
        theta = RandomReal[{0,360}]; (* chooses a random azimuthal angle*)
        phi = RandomReal[{0,180}]; (*chooses a random polar angle*)
        rot = {{Cos[phi],Sin[phi],0},{-
Cos[theta]*Sin[phi],Cos[theta]*Cos[phi],Sin[theta]},{Sin[theta]*Sin[phi],-Sin[theta]*Cos[phi],Cos[theta]}};
(*rotation matrix based on phi, theta*)
        rotatedPositions[[part]] = Table[(positions[[part]][[i]]).rot*rad+c,{i,1,n}]; (*calculates and stores the
rotated positions based on the rotation matrix*)
```

```
 ];
(*-----------------------------------------------------------------------------------------------------------------*)
(*These lines create and plot graphics displaying all particles in the surface and, if there are ≤15 particles in
the surface, also their positions*)
pointSpheres = Table[Sphere[partCenters[[i]],particleRadii[[i]]],{i,1,numParticles}];
molSpheres =
Table[Table[{Yellow,Sphere[rotatedPositions[[j]][[i]],moleculeRadius]},{i,1,positionsPerParticle[[j]]}],{j,1,nu
mParticles}];
If[numParticles<=15,Graphics3D[{pointSpheres,molSpheres},Boxed-
>False],Graphics3D[{pointSpheres},Boxed->False]] (*Plot what the initial surface will look like BEFORE
forced dead spots are removed*)
(*-----------------------------------------------------------------------------------------------------------------*)
(*-----------------------------------------------------------------------------------------------------------------*)
(*-----------------------------------------------------------------------------------------------------------------*)
(*DETERMINING NEIGHBORING PARTICLES*)
XPartDistances = {}; (*creates a cross distance table to be used in finding particle nearest neighbors*)
For[k=1,k<=numParticles,k++, (*iterates over every particle*)
        distances= Table[{i,pythag[partCenters[[k]],partCenters[[i]]]},{i,numParticles}]; (*calculates the
distances to every other particle from the current particle*)
        XPartDistances = Append[XPartDistances,distances]; (*adds the distance table to the cross distance
table*)
 ];
neighbors = Table[{},{i,numParticles}]; (*creates a table to store particle neighbors*)
validPositions = Table[{},{i,numParticles}]; (*creates a table to store molecular positions that are valid. This
will be the position list used from here on and the one exported*)
For[k=1,k<=numParticles,k++, (*iterates over all particle*)
        For[l=1,l<=numParticles,l++, (*for each particle k, iterates over all particles*)
                If[l!=k&&pythag[partCenters[[k]],partCenters[[l]]]<=particleRadii[[k]]+particleRadii[[l]],
(*if particles l and k are within their combined radii from each other and are not the same particle, they are
neighbors*)
                        neighbors[[k]] = Append[neighbors[[k]],l];
                ];
        ];
        For[j=1,j<=positionsPerParticle[[k]],j++, (*for each position on particle k, check to see if it within any
of the neighboring particles*)
                valid = True;
                For[i=1,i<=Length[neighbors[[k]]],i++,
                        neighborPart = neighbors[[k]][[i]];

        If[pythag[partCenters[[neighborPart]],rotatedPositions[[k]][[j]]]<=(particleRadii[[neighborPart]]+m
oleculeRadius),
                                valid=False;
                        ,
                                For[m=1,m<=positionsPerParticle[[neighborPart]],m++, (*also check to see
if it is too close to other molecules on separate particles*)
                                        intermoldist =
pythag[rotatedPositions[[neighborPart]][[m]],rotatedPositions[[k]][[j]]];
                                        If[intermoldist<(2*moleculeRadius),(*currently the TOO CLOSE
condition is that molecules have overlapping Van der Waals radii *)
                                                valid=False;
                                        ];
                                ];

                        ];
```

```
                    ];
                    If[valid,validPositions[[k]] = Append[validPositions[[k]],rotatedPositions[[k]][[j]]];];
(*stores the positions that are valid*)
          ];
          neighbors[[k]] = Append[neighbors[[k]],k]; (*stores the particle's neighboring particle*)
 ];
validPerPart = Table[Length[validPositions[[i]]],{i,1,numParticles}]; (*counts the number of valid positions
per particle*)
Print["Particle Neighbors Found!"];
Print["Time Passed = ", Floor[(AbsoluteTime[]-startTime)/60]," minutes,
",PaddedForm[Mod[(AbsoluteTime[]-startTime),60],{4,2}]," seconds"];
If[numParticles>=5, (*if there are at least 5 particles creates graphics showing the first 5 particles of rotated
positions and valid positions*)
 (*Plots ALL positions on the first 5 particles. Will get angry but can be ignored if there are fewer than 5
positions*)

ListPointPlot3D[{rotatedPositions[[1]],rotatedPositions[[2]],rotatedPositions[[3]],rotatedPositions[[4]],rotat
edPositions[[5]]},BoxRatios->Automatic,PlotStyle->PointSize[Large]]
  (*Plots ONLY VALID on the first 5 particles. Will get angry but can be ignored if there are fewer than 5
positions*)

ListPointPlot3D[{validPositions[[1]],validPositions[[2]],validPositions[[3]],validPositions[[4]],validPositions[
[5]]},BoxRatios->Automatic,PlotStyle->PointSize[Large]]
 ]

(*-------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*-------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*-------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*NEIGHBORING Positions*)
(*stackInfo Structure: {type,{NN},recomb prob,{Hopping Probs}} type 1 = abs, type 2 = cat, type 3 = dead
spot*)
stackInfo =Table[Table[{1,0,0,0},{j,1,validPerPart[[i]]}], {i,numParticles}]; (*creates the stackinfo table which
contains all reference information throughout the simulation*)
(*calculates an inclination angle for every valid position to be used in anisotropy studies*)
inclinationAngleArray = Table[Table[(180/Pi)*ArcTan[((validPositions[[i]][[j]]-
partCenters[[i]])[[2]])/((validPositions[[i]][[j]]-partCenters[[i]])[[1]])],{j,1,validPerPart[[i]]}],
{i,numParticles}];
xDistances = Table[Table[{},{j,1,validPerPart[[i]]}],{i,numParticles}]; (*cross distance table holding distance
from every particle to every potential neighboring particle*)


For[i=1,i<=numParticles,i++, (*iterates over all particles*)
          For[j=1,j<=validPerPart[[i]],j++, (*iterates over every position on particle i*)
                    xDist = {};
                    For[k=1,k<=Length[neighbors[[i]]],k++, (*iterates over all neighboring particles to particle
i*)
                              nP = neighbors[[i]][[k]];
                              For[m=1,m<=validPerPart[[nP]],m++, (*iterates over every position on the
neighboring particle*)
                                        xDist =
Append[xDist,{nP,m,pythag[validPositions[[i,j]],validPositions[[nP,m]]]}]; (*calculates the distance between a
position [i,j] and every possible neighbor *)
                              ];
                    ];
                    xDist = SortBy[xDist,Last]; (*sorts all possible neighboring positions by distance*)
```

```
                xDistances[[i,j]] = xDist; (*stores the position neighbors in the large cross-distance table by
Particle #, Position #, distance from [i,j]*)
        ];
 ];


If[numParticles<=15, (*repeat of previous graphic using yellow positions but now showing only valid
positions. Also shows two example nearest neighboring limits in green.*)
 pointSpheres = Table[Sphere[partCenters[[i]],particleRadii[[i]]],{i,1,numParticles}];
 molSpheres =
Table[Table[{Blue,Sphere[validPositions[[j]][[i]],moleculeRadius]},{i,1,Length[validPositions[[j]]]}],{j,1,numP
articles}];
 NNsphere = {Opacity[0.5],Green,Sphere[validPositions[[1]][[1]],moleculeRadius*reach]};
 NNsphere2 = {Opacity[0.5],Green,Sphere[validPositions[[1]][[-1]],moleculeRadius*reach]};
 Graphics3D[{pointSpheres,molSpheres,NNsphere,NNsphere2},Boxed->False] (*Plots surface after removal of
dead spots. Consider commenting this out if creating a large surface*)
 ]

Print["Position Neighbors Found!"];
(*SETS NEAREST NEIGHBORS*)
(*creates a bunch of tables to store relevant orientation information for each molecule*)
moleculeVectors = Table[Table[0,{j,1,validPerPart[[i]]}],{i,1,numParticles}];
moleculeAngles = Table[Table[0,{j,1,validPerPart[[i]]}],{i,1,numParticles}];
anisContribution = Table[Table[0,{j,1,validPerPart[[i]]}],{i,1,numParticles}];
positionDistances = Table[Table[{},{j,1,validPerPart[[i]]}],{i,numParticles}]; (*creates a table to store
distances between nearest neighbors*)
NNs =Table[Table[{},{j,1,validPerPart[[i]]}],{i,numParticles}]; (*creates a table to store nearest neighbor
positions for each position*)
For[i=1,i<=numParticles,i++, (*iterating over all particles*)
        For[j=1,j<=validPerPart[[i]],j++, (*iterating over positions*)
                        maxNN = 2; (*resets the INDEX of the maximum nearest neighbor to 3*)
                        While[(xDistances[[i]][[j]][[maxNN]][[3]])<=reach*moleculeRadius, (*while the
next neighbor is within the reaching radius, continue to append the NN and position distance*)
                                NNs[[i]][[j]] =
Append[NNs[[i]][[j]],{xDistances[[i]][[j]][[maxNN]][[1]],xDistances[[i]][[j]][[maxNN]][[2]]}];
                                positionDistances[[i]][[j]] =
Append[positionDistances[[i]][[j]],xDistances[[i]][[j]][[maxNN]][[3]]];
                                maxNN=maxNN+1; (*keep moving on to the next possible neighbor*)
                        ];
                        stackInfo[[i]][[j]][[2]]=NNs[[i]][[j]] ; (*stores the generated list of nearest neighbors
in the stack Info matrix*)

                moleculeVectors[[i]][[j]] = validPositions[[i]][[j]]-partCenters[[i]]; (*finds the radially
outward 3d vector that describes each molecules orientation*)
                moleculeAngles[[i]][[j]] =
ArcCos[Dot[moleculeVectors[[i]][[j]],{0,1,0}]/Norm[moleculeVectors[[i]][[j]]]]; (*calculates the angle made
between molecular orientations and light polarity*)
                anisContribution[[i]][[j]] = 1.5*(Cos[moleculeAngles[[i]][[j]]])^2 - 0.5; (*calculates the
contribution to the anisotropy a molecule would have if oxidized based on its angle*)

        ];
 ];
Print["Position Neighbors Set!"];
```

```
Print["Time Passed = ", Floor[(AbsoluteTime[]-startTime)/60]," minutes,
",PaddedForm[Mod[(AbsoluteTime[]-startTime),60],{4,2}]," seconds"];


(*Once you get a surface that looks good, make sure to run this to export it to Documents*)
Export[stackName, {stackInfo,    inclinationAngleArray, {{numParticles, validPerPart}},    validPositions,
PositionDistances, particleRadii, neighbors,    partCenters, moleculeAngles, anisContribution}];
```

Full Model

```
stackName = "TestStack.wdx";
stack=Import[stackName];
stackInfo = stack[[1]];
inclinationAngleArray = stack[[2]];
numParticles = stack[[3]][[1]][[1]];
positionsPerParticle = stack[[3]][[1]][[2]];
moleculePositions = stack[[4]];
NNDistances = stack[[5]];
particleRadii = stack[[6]];
particleNeighbors = stack[[7]]; (*contains neighboring particles of each particle. Each particle is at the end of its own list of neighbors *)
particleCenters = stack[[8]];
moleculeAngles = stack[[9]];
anisotropyContribution = stack[[10]];
(*SUPPORTING FUNCTIONS*)
probFromTau[x_]:=timeStepSize/x;
dipoleOverlapFunc[x_]:=(Cos[x])^2;
degeneracyAbsFunc[x_]:=Sin[x]   ;
probAbsInit[x_]:=Abs[dipoleOverlapFunc[x](*degeneracyAbsFunc[x]*)] ;      (* This allows polarized light absorption (for when excited by a laser), plus inclination angular degeneracy *)
lightIntensityBeersLaw[x_]:=10^(-((Log10[1/fracTrans]))*x)   ;
distanceBetweenMolecules[x1_,y1_,z1_,x2_,y2_,z2_]:=Sqrt[(x1-x2)^2+(y1-y2)^2+(z1-z2)^2] ;(*PYTHAG*)

FloodFill[part_,pos_]:=(
 If[(stackInfo[[part,pos]][[1]]!=3)&&(percolationZone[[part,pos]]==0),
          percolationZone[[part,pos]]=zoneCount;
                              If[Length[stackInfo[[part,pos]][[2]]]>=1,

          FloodFill[stackInfo[[part,pos]][[2]][[1]][[1]],stackInfo[[part,pos]][[2]][[1]][[2]]];
                                        If[Length[stackInfo[[part,pos]][[2]]]>=2,

          FloodFill[stackInfo[[part,pos]][[2]][[2]][[1]],stackInfo[[part,pos]][[2]][[2]][[2]]];
                                                  If[Length[stackInfo[[part,pos]][[2]]]>=3,

          FloodFill[stackInfo[[part,pos]][[2]][[3]][[1]],stackInfo[[part,pos]][[2]][[3]][[2]]];
                                                            If[Length[stackInfo[[part,pos]][[2]]]>=4,

          FloodFill[stackInfo[[part,pos]][[2]][[4]][[1]],stackInfo[[part,pos]][[2]][[4]][[2]]];
                                                                      If[Length[stackInfo[[part,pos]][[2]]]>=5,

          FloodFill[stackInfo[[part,pos]][[2]][[5]][[1]],stackInfo[[part,pos]][[2]][[5]][[2]]];
                                                                                If[Length[stackInfo[[part,pos]][[2]]]>=6,

          FloodFill[stackInfo[[part,pos]][[2]][[6]][[1]],stackInfo[[part,pos]][[2]][[6]][[2]]];

          If[Length[stackInfo[[part,pos]][[2]]]>=7,

          FloodFill[stackInfo[[part,pos]][[2]][[7]][[1]],stackInfo[[part,pos]][[2]][[7]][[2]]];

          If[Length[stackInfo[[part,pos]][[2]]]>=8,

          FloodFill[stackInfo[[part,pos]][[2]][[8]][[1]],stackInfo[[part,pos]][[2]][[8]][[2]]];
```

```
                                                                                                    ];
                                                                                          ];
                                                                                 ];
                                                                        ];
                                                               ];
                                                      ];
                                             ];
                                    ];
                  ];
 )
(*------------------------------------------------------------------------------------------------------------------------------*)
(*------------------------------------------------------------------------------------------------------------------------------*)
(*------------------------------------------------------------------------------------------------------------------------------*
NotebookSave[]; (*Saves notebook when cell is executed to save tears later*)
startTime=AbsoluteTime[];
$RecursionLimit = 50000;
date = ToString[Mod[DateList[][[1]],100]]<>ToString[DateList[][[2]]]<>ToString[DateList[][[3]]];
Print["Status: Build Loaded\n","Time Passed = ", Floor[(AbsoluteTime[]-startTime)/60]," minutes, ",PaddedForm[Mod[(AbsoluteTime[]-startTime),60],{4,2}]," seconds"];

seed=RandomPrime[{10^9,10^10}]; (*Uses the random number generator to get a random number that we will use to reset the number generator with a value we can store*)
(*seed = FromDigits["deadbeef"];  (*Sets the seed to a consistent value for testing. Comment this line out to use the random seed above*)*)
SeedRandom[seed]; (*Resets the random number generator with the seed determined above *)
(*---------------------------------------------------------------------------------------------------------------------------*)
(*------------------------------------------------------------------------------------------------------------------------------*)
(*------------------------------------------------------------------------------------------------------------------------------*)
(*RUN OPTIONS*)

absBLLaw=False; (*True assigns initially excited dyes according to Beers Law absorbance, False makes the assignments randomly*)
absAnisotropy=False;  (*True assigns initially excited dyes taking into account the angle on inclination with respect to the incoming light, False is random*)
CWmode = False; (*if true, only excites one dye initially and then allows excitations to happen during the simulation based on illumination*)

distanceDependantHopping = False; (*if true, the same number of nearest neighbors will be used but they will have different hopping probabilities*)
electronSpreading = False; (*non-physical approximation which allows electrons to spread slightly away from their hole*)
electronAreaScaling = False; (*non-physical approximation which scales electron density by particle surface area*)
(*electronDistributionFixed = False;*)
electronDistributionHomogenized = False; (*approximates that all electrons move fast enough that the whole surface receives the same electron density*)
maxTimeSteps = 2000; (*Maximum number of timesteps to take in a CW experiment*)
maxOxState=2; (*Maximum oxidation state that can be reached by the catalysts*)
CWSuns = 1;
numTrials = 3;
absorbance=0.044;  (* 99% is 0.256; 72% is 0.193 *)
fracTrans=0.10(*10^(-absorbance)*);

(*Dead Spots*)
DSFixed = False; (*If True, use per particle value, otherwise use per film value*)
```

```
DSperParticle = 0;
pctDS = 0.0; (*% of positions covered by deadspots*)
(*Catalysts*)
CatFixed = False; (*If True, use per particle value, otherwise use per film value*)
CatperParticle = 2;
pctCats = 1.0; (*1.0 = 1%*)
(*Dyes*)
DyesFixed = False; (*If True, use per particle value, otherwise use per film value*)
DyesperParticle = 10;
DyesPerFilm = 20;
UsePercent = True; (*allows a percent of a surfaces sites to be used rather than a fixed number*)
PctExcitedDyes = 1.0; (*1 = 1%, 100 = 100%*)
If[UsePercent==True,
        DyesPerFilm = Round[(PctExcitedDyes/100.0)*Total[positionsPerParticle]*(1-pctDS)*(1-pctCats)];
 ];
part = 1;

(*Recording*)
RecTurnovers = True;
RecTimeBehavior = True;
RecAnis = True;
RecHopPaths = False;


(*KHOP AND KRECOMB are arrays of all hopping and recombination time constants. They will all pair-wise be
iterated over*)
KHOP = {100000,50000,20000,10000,5000,2000,1000,500,200,100,50,20,10,5,2,1}*10^(-9);
If[part==1,
 KRECOMB = {1000}*10^(-9);
 ];
If[part==2,
 KRECOMB = {1000000}*10^(-9);
 ];
(*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*SETUP*)
maxZ = Max[moleculePositions[[;;,;;,3]]];
minZ = Min[moleculePositions[[;;,;;,3]]];
particleAreas = particleRadii^2;
particleRegions = Table[2*particleAreas[[i]],{i,1,numParticles}];
If[electronSpreading==True,
 For[x=1,x<=numParticles,x++,
        For[y=1,y<=Length[particleNeighbors[[x]]]-1,y++,
                particleRegions[[x]]=particleRegions[[x]]+particleAreas[[particleNeighbors[[x]][[y]]]];
                ];
        ];
 ];
stackHeight = maxZ-minZ; (*calculates film thickness*)
If[DyesFixed==True, (*overwrites set number of dyes if a set number per particle is specified*)
        numInitialExcitedDyes = numParticles*DyesperParticle;,
        numInitialExcitedDyes = DyesPerFilm;
 ];
If[CatFixed==True, (*Determines the number of catalyst on the film depending on whether there is a set
number per particle or film*)
        numCatalysts = numParticles*CatperParticle;,
```

```
        numCatalysts=Round[(pctCats/100)*Total[positionsPerParticle],1];
 ];
(*FILENAMES*)
turnoverFileName =
date<>"_Turn_"<>ToString[maxOxState]<>"X_"<>ToString[numInitialExcitedDyes]<>"Part"<>ToString[part]
<>"_Excitations_"<>StringTake[stackName,{1,-5}];
timeFileName =
date<>"_Time_"<>ToString[maxOxState]<>"X_"<>ToString[numInitialExcitedDyes]<>"Part"<>ToString[part]
<>"_Excitations_"<>StringTake[stackName,{1,-5}];
anisotropyFileName =
date<>"_Anis_"<>ToString[maxOxState]<>"X_"<>ToString[numInitialExcitedDyes]<>"Part"<>ToString[part]<
>"_Excitations_"<>StringTake[stackName,{1,-5}];
If[absBLLaw==True,
        turnoverFileName=turnoverFileName<>"_BL.CSV";
        timeFileName=timeFileName<>"_BL.XLSX";
        anisotropyFileName=anisotropyFileName<>"_BL.XLSX";
 ,
        turnoverFileName=turnoverFileName<>".CSV";
        timeFileName=timeFileName<>".XLSX";
        anisotropyFileName=anisotropyFileName<>".XLSX";
 ];

allHops = Table[{},{j,1,Length[KRECOMB]*Length[KHOP]}];
allTimes = Table[{},{j,1,Length[KRECOMB]*Length[KHOP]}];
allAnis = Table[{},{j,1,Length[KRECOMB]*Length[KHOP]}];

turnoverTable = Table[0,{j,1,Length[KHOP]*Length[KRECOMB]+1}];
turnoverTable[[1]] = {"TauRecomb (ns)","TauHop (ns)","TauRatio","Percent Turnovers","Number
Turnovers"};
ParameterPoint = 0; (*Counter for parameter point (Hopping & Recombination Constants Combination)*)
For[R = 1,R<=Length[KRECOMB],R++, (*Loops over all recombination rates*)
 For[H = 1,H<=Length[KHOP],H++, (*Loops over all hopping rates*)
  tauHopDyetoDye = KHOP[[H]]; (*Lifetime for Dye to Dye hopping*)
  tauRecombDye= KRECOMB[[R]] ;(*Lifetime for dye recombination*)(*467*(10^-6)*)
  ParameterPoint++; (*Keeps track of what iteration the model is on*)
  HRR = N[KRECOMB[[R]]/KHOP[[H]]]; (*Find the hopping-recombination ratio*)
  timeDecays = {};
  timeTable = Table[{{"Timestep","Time","Dyes Remaining","Charges","Turnovers","Dye
Recombinations","Catalyst Recombinations","Catalysts Remaining","Excitations"}},{i,1,numTrials}];
  anisTable = Table[{{"Timestep","Time","Excited Molecules","Anisotropy"}},{i,1,numTrials}];

  (*Creates Tables to store Data of interest*)
  turnoverTotals=0;

  (*Creates Counters for recording the number of dyes which are removed in the auto recombine step*)
  noCatRecombinations = 0;
  loneChargeRecombinations = 0;
  autoRecombinations = 0;
  hops = {};
  shortest = 10^10;
  For[trials = 1,trials<=numTrials,trials++, (*Loops over the number of trials specified to build up statistics*)
   EndCondition=False;
   If[RecHopPaths==True,
        DyeTraces = Table[{},{i,1,numInitialExcitedDyes}];
        DyeIds = Range[numInitialExcitedDyes];
```

```
        Print["Start: ",DyeIds];
    ];
    Print["Hop Rate: ",tauHopDyetoDye*10^9," ns","\nRecomb Rate: ",tauRecombDye*10^9," ns","\nParameter Point: ",ParameterPoint,"\nTrial: ",trials,"\nTime Passed = ", Floor[(AbsoluteTime[]-startTime)/60]," minutes"];
    (*Creates Counters for the number of times a catalyst has been found and the number of turnovers per trial*)
    turnovers = 0;
    numDyes = numInitialExcitedDyes;
    (*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
    (*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
    (*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
    (*PRIMARY PARAMETERS*)
    (*calculates the minimum effective time constant and makes sure the time step size is much smaller*)
    effectiveAnisTau = tauHopDyetoDye*3.75;
    minTau = Min[effectiveAnisTau,tauRecombDye];
    timeStepSize =  minTau/350.0; (*2.0*(10^-9)*) (*NANOSECONDS*) (*MUST BE SMALLER THAN FASTEST OBSERVABLE*)
    moleculeRadius=0.05; (*in units of 1 = 15nm*)
    tauExcite = (14.8*(10^-6))/CWSuns;
    (*sets Hopping rate from cat back to dye to be huge to prevent reverse hopping*)
    tauHopCattoDye = tauHopDyetoDye*100000000000;
    (*TABLES ARE CREATED TO STORE RATES FOR CATALYST BEHAVIOR. THESE MUST BE MANUALLY CHANGED BASED ON MAX OX STATE*)
    tauHopDyetoCat = Table[0,4,1]; (*Creates a table to store dye oxidation rates of catalysts from dyes*)
    tauHopDyetoCat[[1]] = tauHopDyetoDye/27; (* lifetime associated with hopping from a Dye to a catalyst that has 0 charges*)
    tauHopDyetoCat[[2]] = tauHopDyetoDye/27; (*27 is used here to ensure that an adjacent dye hops to a catalyst 90% of the time before hopping away from it.*)
    tauHopDyetoCat[[3]] =tauHopDyetoDye/27;
    tauHopDyetoCat[[4]] = tauHopDyetoDye/27;
    tauHopCattoCat = Table[0,4,1]; (*Creates a table to store catalyst oxidation rates of catalysts*)
    tauHopCattoCat[[1]] = tauHopDyetoDye; (*Lifetime associated with a hop from ANY catalyst to a cat0*)
    tauHopCattoCat[[2]] =tauHopDyetoDye;
    tauHopCattoCat[[3]] = tauHopDyetoDye;
    tauHopCattoCat[[4]] =tauHopDyetoDye;

    tauRecombCatRates = Table[0,4,2]; (*Creates a table to store recombination rates for catalysts*)
    tauRecombCatRates[[1,1]]=tauRecombDye;(*32.2*(10^-6); (*Fast lifetime for a single oxidized catalyst*)*)
    tauRecombCatRates[[1,2]]=tauRecombDye;(*692.0*(10^-6);(*Slow lifetime for a single oxidized catalyst*)*)
    tauRecombCatRates[[2,1]]=tauRecombDye;(*32.2*(10^-6);(*Fast lifetime for a double oxidized catalyst*)*)
    tauRecombCatRates[[2,2]]=tauRecombDye;(*692.0*(10^-6);(*Slow lifetime for a double oxidized catalyst*)*)
    tauRecombCatRates[[3,1]]=tauRecombDye;(*32.2*(10^-6);(*Fast lifetime for a triple oxidized catalyst*)*)
    tauRecombCatRates[[3,2]]=tauRecombDye;(*692.0*(10^-6);(*Slow lifetime for a triple oxidized catalyst*)*)
    tauRecombCatRates[[4,1]]=tauRecombDye;(*32.2*(10^-6);(*Fast lifetime for a quadruple oxidized catalyst*)*)
    tauRecombCatRates[[4,2]]=tauRecombDye;(*692.0*(10^-6);(*Slow lifetime for a quadruple oxidized catalyst*)*)
    popFrac1A = 0.38; (*Fraction of catalyst population that will undergo FAST recombination when singly Oxidized*)
    popFrac2A = 0.5; (*Fraction of catalyst population that will undergo FAST recombination when doubly Oxidized*)
    popFrac3A = 0.5; (*Fraction of catalyst population that will undergo FAST recombination when triply Oxidized*)
```

```
popFrac4A = 0.5;(*Fraction of catalyst population that will undergo FAST recombination when quadruply Oxidized*)

tauTurnover = 1*(10^-9); (*NANOSECONDS*) (*NOT USED*) (*Lifetime for a fully oxidized catalyst to perform chemistry*)
(*----------*)
(*----------*)
(*----------
*)(*SECONDARY PARAMETERS*)

(*Converts lifetimes to probabilities *)
probExcite = probFromTau[tauExcite];
probHopDyetoDye = probFromTau[tauHopDyetoDye];
probHopDyetoCat = probFromTau[tauHopDyetoCat];
probHopCattoDye = probFromTau[tauHopCattoDye];
probHopCattoCat = Map[probFromTau][tauHopCattoCat];(*converts all lifetimes into probabilities for oxidizing catalysts from dyes*)
probRecombCatRates = Map[probFromTau][tauRecombCatRates]; (*converts all lifetimes into probabilities for recombination over the timestep*)
probHopDyetoCat = Map[probFromTau][tauHopDyetoCat];(*converts all lifetimes into probabilities for oxidizing catalysts from dyes*)
probDyeRecomb = probFromTau[tauRecombDye]; (*Probability that a dye will recombine over the timestep*)
probTurnover = probFromTau[tauTurnover]; (*NOT USED*)

dyeRecombinations=0; (*prepares to count the number of dye recombinations that occur*)
catalystRecombinations=Table[0,maxOxState]; (*sets up an array to store the number of catalyst recombinations by oxidation state*)
catalystSpecies = Table[0,maxOxState]; (*creates an array to store the number of each catalyst species by oxidation state*)
anisotropy = 0; (*will be use to total the anisotropy after each pass of the loop*)
excitations=0;(*prepares to count the number of dye excitations that occur during the run*)

(*----------*)
(*----------*)
(*----------*)
(*----------*)
(*----------*)
(*----------*)
(*PRE SIMULATION PRINTOUT*)
Print["----------NEW RUN----------"];
(*----------*)
(*----------*)
(*----------*)
(*ASSIGNMENT OF CATALYST AND DYE POSITIONS*)
oxList = Table[Table[0,{j,1,positionsPerParticle[[i]]}],{i,numParticles}]; (*resets the table that stores the oxidation states of every molecule*)
openPositions = 0; (*resets a counter that will store the number of valid positions in the whole surface*)
choiceWeighting = Table[Table[1,{j,1,positionsPerParticle[[i]]}],{i,numParticles}]; (*initializes a table of selection weights per molecule*)
possiblePositions = Table[Table[{i,j},{j,1,positionsPerParticle[[i]]}],{i,numParticles}]; (*initializes the list of all possible positions*)
FlattenedPositions = Flatten[possiblePositions,1];
For[x=1,x<=numParticles,x++, (*iterates over all positions and resets them all to be non-oxidized dyes and counts them as open*)
```

```
        For[y=1,y<=Length[stackInfo[[x]]],y++,
            stackInfo[[x,y]][[1]]=1; (*Reset the position to be a dye position*)
            openPositions++;
        ];
    ];
    (*SETTING DEAD SPOTS*)
    DSArray = {}; (*creates an empty array to store empty molecular positions*)
    If[DSFixed, (*if the number of dead spots is to be fixed per particle, choose that many for each and add them to the DSArray*)
        For[x=1,x<=numParticles,x++,
            DSArray = AppendTo[DSArray,RandomSample[ choiceWeighting[[x]]->possiblePositions[[x]],DSperParticle]];
        ];
     ,(*if DS in not Fixed per particle, choose them randomly over the film*)
        numDeadSpots = Round[(pctDS/100.0)*openPositions]; (*number of spots assigned to be neither dyes nor catalysts*)
        DSArray = RandomSample[Flatten[choiceWeighting]->FlattenedPositions ,numDeadSpots];
    ];
    For[d=1,d<=Length[DSArray],d++, (*iterate over all the dead spots chosen, mark them as full positions, set their probability to be chosen to 0, and set their type in the stack info matrix*)
        dpos = DSArray[[d]];
        stackInfo[[dpos[[1]],dpos[[2]]]][[1]]=3;
        choiceWeighting[[dpos[[1]],dpos[[2]]]]=0;
        openPositions--;
    ];
    (*SETTING CATS*)
    CatArray  = {}; (*creates an array to store the positions of all catalysts in the surface*)
    If[CatFixed, (*if catalysts are to be fixed per particle, choose the appropriate number on each particle and store them*)
        For[x=1,x<=numParticles,x++,
            CatArray = AppendTo[CatArray,RandomSample[choiceWeighting[[x]]->possiblePositions[[x]],CatperParticle]];
        ];
     , (*If Cats are Not Fixed per particle, choose them randomly over the whole film*)
        numCats = Round[(pctCats/100.0)*openPositions];
        CatArray = RandomSample[Flatten[choiceWeighting]->FlattenedPositions,numCats];
    ];
    For[c=1,c<=Length[CatArray],c++,(*iterate over all catalysts chosen, mark the positions as full, set their probability for selection to 0, and choose additional parameters for them in the stack info matrix*)
        cpos = CatArray[[c]];
        stackInfo[[cpos[[1]],cpos[[2]]]][[1]]=2;
        choiceWeighting[[cpos[[1]],cpos[[2]]]]=0;
        recomb1 = RandomChoice[{popFrac1A,1-popFrac1A}->{probRecombCatRates[[1,1]],probRecombCatRates[[1,2]]}]; (*randomly choose a 1st recombination rate*)
        recomb2 = RandomChoice[{popFrac2A,1-popFrac2A}->{probRecombCatRates[[2,1]],probRecombCatRates[[2,2]]}]; (*randomly choose a 2nd recombination rate*)
        recomb3 = RandomChoice[{popFrac3A,1-popFrac3A}->{probRecombCatRates[[3,1]],probRecombCatRates[[3,2]]}]; (*randomly choose a 3rd recombination rate*)
        recomb4 = RandomChoice[{popFrac4A,1-popFrac4A}->{probRecombCatRates[[4,1]],probRecombCatRates[[4,2]]}]; (*randomly choose a 4th recombination rate*)
        stackInfo[[cpos[[1]],cpos[[2]]]][[3]]={recomb1,recomb2,recomb3,recomb4}; (*sets each catalysts 4 recombination rates*)
    ];
    (*calculate the fraction depth of each molecular position in the surface and stores in a table*)
```

```
moleculeDepthFraction = Table[Table[{(moleculePositions[[i]][[j]][[3]]-minZ)/stackHeight},{j,1,positionsPerParticle[[i]]}],{i,1,numParticles}];
(*SETTING DYES*)
DyeArray = {}; (*creates an array to store the positions of all initially excited dyes*)
If[absBLLaw, (*if Beers Law is to be used*)
	(*calculates the probability weight from Beer's Law for each molecule based on depth*)
	placementWeighting = Map[lightIntensityBeersLaw,moleculeDepthFraction];
	choiceWeighting = choiceWeighting*placementWeighting; (*factors in beers law weighting to open positions*)
 ];
If[absAnisotropy,choiceWeighting = choiceWeighting*Map[probAbsInit,moleculeAngles];];(*Apply Anisotropy effects if turned on*)
If[DyesFixed, (*If dye positions are to be fixed per particle, choose them per particle based on weight previously calculated*)
	For[x=1,x<=numParticles,x++,
		DyeArray = AppendTo[DyeArray,RandomSample[choiceWeighting[[x]]->possiblePositions[[x]],DyesperParticle]];
	];
 DyeArray = Flatten[DyeArray,1];
 ,(*chooses dyes randomly over the surface*)
	DyeArray = RandomSample[Flatten[choiceWeighting]-> FlattenedPositions,DyesPerFilm];
 ];
(*Set the position of the initially excited absorbers in the status Matrix*)
For[a=1, a<=Length[DyeArray],a++,(oxList[[(DyeArray[[a]][[1]]),(DyeArray[[a]][[2]])]])=1;];

(*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
(*Sorting dye positions into percolation zones*)
(*creates a table to store the percolation zone of each molecule*)
percolationZone = Table[Table[0,{j,1,positionsPerParticle[[i]]}],{i,1,numParticles}];
zoneCount=0; (*counter that keeps track of the number of zones there are so far*)
For[x=1,x<=numParticles,x++, (*iterates over all particles*)
	For[y=1,y<=positionsPerParticle[[x]],y++, (*iterates over all positions on particle x*)
		If[(stackInfo[[x,y]][[1]]!=3)&&(percolationZone[[x,y]]==0), (*if the molecule isn't a dead spot and it is still set to zone 0, flood*)
			zoneCount++ (*count the new zone discovered!*)
			FloodFill[x,y]; (*Run a recursive algorithm to identify all molecular positions which are connected to each other starting from here*)
		];
	];
 ];

Zones = Table[{},{i,1,zoneCount}]; (*creates a table to store the (particle, position) coordinates of all molecules by zone*)
ZonesPos = Table[{},{i,1,zoneCount}]; (*creates a table to store the XYZ coordinates of all molecules by zone*)
ZoneSizes = Table[{i,0},{i,1,zoneCount}]; (*creates a table to store the size of each zone*)

For[z=1,z<=zoneCount,z++, (*iterates over all zones*)
	For[x=1,x<=numParticles,x++, (*iterates over all particles*)
		For[y=1,y<=Length[stackInfo[[x]]],y++, (*iterates over all positions*)
			If[percolationZone[[x,y]]==z, (*if the molecule belongs to current zone, store its PP and XYZ coordinates*)
				Zones[[z]] = AppendTo[Zones[[z]],{x,y}];
```

```
                                (*ZonesPos[[z]] = AppendTo[ZonesPos[[z]],positions[[x,y]]];*) (*not actually used outside of troubleshooting*)
                            ];
                        ];
                ];
                ZoneSizes[[z]][[2]] =Length[Zones[[z]]]; (*calculates the size of each percolation zone*)
        ];
        (*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
        (*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)
        (*-----------------------------------------------------------------------------------------------------------------------------------------------------------*)

        If[distanceDependantHopping==True, (*if different nearest neighbors should receive different hopping weights based on distance*)
                allDistAng = Flatten[NNDistances]*15*10; (*creates an array off all nearest neighbor distances*)
                molRadAng = moleculeRadius*15*10; (*Van der Waals radius of molecules in angstroms*)
                tunnelFactor = 0.35;
                ExpAve=(-Log[Mean[Exp[(-tunnelFactor)*(allDistAng-
2*molRadAng)]]]/tunnelFactor)+2*molRadAng;
                AveFactor =Exp[(ExpAve-2*molRadAng)*(-0.35)];
                AveFactorInverse = 1.0/AveFactor;
                ];
                factors = {};
        (*iterates through all positions and counts the number of catalyst neighbors to each to assign weighted hopping probabilities*)
        For[x=1,x<=numParticles,x++, (*iterates over all the particles*)
                For[y=1,y<=positionsPerParticle[[x]],y++, (*iterates over each position on a particle*)
                        If[stackInfo[[x,y]][[1]]==1, (*currently looking at a dye*)
                                stackInfo[[x,y]][[3]]=probDyeRecomb;
                                neighbors = stackInfo[[x,y]][[2]]; (*retrieves the list of nearest neighbors for the given particle and position*)
                                hopProbs = Table[0,{Length[neighbors]}]; (*creates a table to store a molecule's hopping probability to each nearest neighbor*)
                                For[z=1,z<=Length[neighbors],z++, (*iterates over each neighbor in the list*)
                                        If[distanceDependantHopping==True, (*if distance dependence, calculate relative hopping weight neighbor by neighbor*)
                                        distanceFactor = AveFactorInverse*Exp[(-
tunnelFactor)*(NNDistances[[x,y]][[z]]*15*10-2*molRadAng)];
                                        factors = AppendTo[factors,distanceFactor];
                                        ,distanceFactor=1;]; (*if no distance dependence, weight everything by 1*)
                                        If[stackInfo[[neighbors[[z]][[1]],neighbors[[z]][[2]]]][[1]]==1,
hopProbs[[z]]=probHopDyetoDye*distanceFactor; ];(*Neighbor is a dye*)
                                        If[stackInfo[[neighbors[[z]][[1]],neighbors[[z]][[2]]]][[1]]==2,
hopProbs[[z]]=probHopDyetoCat[[1]]*distanceFactor; ];(*Neighbor is a catalyst*)
                                ];
                        ];
                        If[stackInfo[[x,y]][[1]]==2, (*currently looking at a catalyst*)
                                neighbors = stackInfo[[x,y]][[2]]; (*retrieves the list of nearest neighbors for the given particle and position*)
                                hopProbs = Table[0,{Length[neighbors]}]; (*creates a table to store a molecule's hopping probability to each nearest neighbor*)
                                For[z=1,z<=Length[neighbors],z++, (*iterates over each neighbor in the list*)
                                        If[stackInfo[[neighbors[[z]][[1]],neighbors[[z]][[2]]]][[1]]==1, (*Neighbor is a dye*)
                                                hopProbs[[z]]=probHopCattoDye;
                                                ];
```

```
                                        If[stackInfo[[neighbors[[z]][[1]],neighbors[[z]][[2]]]][[1]]==2, (*Neighbor is
a catalyst*)
                                                hopProbs[[z]]=probHopCattoCat[[1]];
                                        ];
                                ];
                        ];
                        stackInfo[[x,y]][[4]]=hopProbs; (*assigns the array of hopping probabilities to the position in
the stack info.*)
                ];
        ];
        POI = RandomSample[DyeArray]; (*randomly sorts the initial points of interest list to make order of action
arbitrary*)

        CatsPerParticle = Table[0,{i,1,numParticles}]; (*prepares a table to count the number of catalysts per
particle*)
        initialChargeDistPZ = Table[0,{zoneCount}]; (*prepares a table to count the number of excitations per
percolation zone*)
        CatsPerPZ = Table[0,{i,1,zoneCount}];(*prepares a table to count the number of catalysts per percolation
zone*)
        For
[x=1,x<=Length[POI],x++,initialChargeDistPZ[[percolationZone[[POI[[x]][[1]],POI[[x]][[2]]]]]]++;];(*counts
up the number of excitations on each particle*)
        For[cat=1,cat<=Length[CatArray],cat++,
CatsPerPZ[[percolationZone[[CatArray[[cat]][[1]],CatArray[[cat]][[2]]]]]]++;]; (*Counts Cats per percolation
Zone*)
        For[cat=1,cat<=Length[CatArray],cat++, CatsPerParticle[[CatArray[[cat]][[1]]]]++; ]; (*Counts the number
of Catalysts per particle and stores in Array*)
        hopelessZones = {}; (*creates an array that stores all zone with either 0 cats or fewer excitations than the
maximum number*)
        For[x=1,x<=Length[POI],x++, (*for each molecule currently with a charge (all dyes during this prestart
calculation)*)
                p = POI[[x]]; (*current molecule*)
                pz = percolationZone[[p[[1]]]][[p[[2]]]]; (*percolation zone of the molecule*)
                        If[(CatsPerPZ[[pz]]==0 )||(initialChargeDistPZ[[pz]]<maxOxState),(*If dyes can't contribute
to a full catalyst, do the recombination thing*)
                                If[CatsPerPZ[[pz]]==0 ,noCatRecombinations++;]; (*if the excitation is
hopeless because of lack of cats, record that*)
                                If[initialChargeDistPZ[[pz]]<maxOxState ,loneChargeRecombinations++;];
(*if the excitation is hopeless because it lacks fellow excitations in its zone, record that*)
                                If[!MemberQ[hopelessZones,pz], (*if the zone is hopeless and is not yet on
the list, add it*)
                                        hopelessZones = AppendTo[hopelessZones,pz];
                                ];
                        ];
        ];
        If[RecHopPaths==True,                   For[x=1,x<=numInitialExcitedDyes,x++,DyeTraces[[x]]                   =
AppendTo[DyeTraces[[x]],POI[[x]]];];
        (*-------------------------------------------------------------------------------------------------------------------------------------------------------*)
        (*-------------------------------------------------------------------------------------------------------------------------------------------------------*)
        (*-------------------------------------------------------------------------------------------------------------------------------------------------------*)
        Print["---------------------------------Beginning Main Loop-------------------------------------"];
        (*MAIN LOOP*)
        timestep=0; (*WOOOOOOO finally going to start doing stuff!!!!!!*)
        While[EndCondition==False,  (*keep going until there are no more excited dyes*)
                If[Mod[timestep,1000]==0, (*every 1000 timesteps*)
```

```
		If[And[numDyes<=0,CWmode==False],EndCondition=True;];
		sanityCheck = dyeRecombinations+numDyes+turnovers*maxOxState; (*performs a sanity check by adding up all the places excitations could have been lost*)
		For[state=1,state<=maxOxState,state++,
			sanityCheck = sanityCheck+state*catalystSpecies[[state]]+catalystRecombinations[[state]];
		];
	If[numCatalysts>0,
		allHopeless = True;  (*checks to make sure at least one dye remains in a non-hopeless zone*)
		For[x=1,x<=Length[POI],x++,
			p=POI[[x]];
			pz = percolationZone[[p[[1]]]][[p[[2]]]];

If[And[!MemberQ[hopelessZones,pz],stackInfo[[p[[1]],p[[2]]]][[1]]==1],allHopeless=False;];];
		If[allHopeless, (*if all remaining excitations are in hopeless zones, terminate the trial after bookkeeping*)
			numDyes=0;
			Print["FORESAKEN!"];
			For[x=1,x<=Length[POI],x++,
				p=POI[[x]];
				If[stackInfo[[p[[1]]]][[p[[2]]]][[1]]==1,
					autoRecombinations++;
					dyeRecombinations++;
				];
			];
			EndCondition=True;
		];
	];
		Print["Timestep: ",timestep,"\nParameter Point: ",ParameterPoint,"\nTurnovers = ",turnovers,"\nRecombinations from Dyes = ",dyeRecombinations,"\nDyes remaining excited = ",numDyes,"\nSingle catalysts remaining excited = ",catalystSpecies[[1]],"\nDouble catalysts remaining excited = ",catalystSpecies[[2]],"\nSanity Check: ",sanityCheck];
	];

	timestep++; (*keeps ticking away...*)
	If[RecAnis==True,
		anisotropy=0; (*Measure Anisotropy for the timestep*)
		For[x=1,x<=Length[POI],x++, anisotropy = anisotropy +anisotropyContribution[[POI[[x]][[1]],POI[[x]][[2]]]];];
		If[Length[POI]>0,anisotropy = anisotropy/Length[POI];,anisotropy=0;];
		timestepStress = 1.0*Length[POI];
		anisRow = {timestep,timestep*timeStepSize,timestepStress,anisotropy};
		anisTable[[trials]] = AppendTo[anisTable[[trials]],anisRow];
	];
	electronsPerParticle = Table[0,{numParticles}]; (*creates a table to count the number of injected electrons per particle*)
	electronDensityPerPart = Table[0,{numParticles}];
	charges = 0;
	numDyes = 0;
	numberCatalysts = 0;
  For [x=1,x<=Length[POI],x++, (*iterates over all currently excited molecules and counts which particle they are on. Assume electron stay local*)
		point = POI[[x]];
		charges=charges+oxList[[point[[1]],point[[2]]]];
		If[stackInfo[[point[[1]],point[[2]]]][[1]]==1,
```

```
                numDyes=numDyes+1,
                numberCatalysts=numberCatalysts+1;
        ];

    electronsPerParticle[[point[[1]]]]=electronsPerParticle[[point[[1]]]]+oxList[[point[[1]],point[[2]]]];
    ];
    If[electronSpreading==True,
        For [x=1,x<=numParticles,x++, (*iterates over all currently excited molecules and counts which particle they are on. Assume electron stay local*)

    electronDensityPerPart[[x]]=electronDensityPerPart[[x]]+electronsPerParticle[[x]]*(2*particleAreas[[x]])/(particleRegions[[x]]);
                For[y=1,y<=Length[particleNeighbors[[x]]]-1,y++,
                    neighbor = particleNeighbors[[x]][[y]];
                    electronDensityPerPart[[neighbor]] = electronDensityPerPart[[neighbor]]+electronsPerParticle[[x]]*(particleAreas[[neighbor]])/(particleRegions[[x]]);
                ];
        ];
    ,
        electronDensityPerPart = electronsPerParticle;
    ];
    If[electronAreaScaling==True,electronDensityPerPart = Table[electronDensityPerPart[[i]]/particleAreas[[i]],{i,1,numParticles}];];
    If[electronDistributionHomogenized==True,
        electronDensityPerPart = Table[Mean[electronDensityPerPart],{i,1,numParticles}];
    ];
    If[RecTimeBehavior==True,
        timeRow = {timestep,timestep*timeStepSize,1.0*numDyes,1.0*charges,1.0*turnovers,1.0*dyeRecombinations,1.0*Total[catalystRecombinations],1.0*numberCatalysts,1.0*excitations};
        timeTable[[trials]] = AppendTo[timeTable[[trials]],timeRow];
    ];
    For[mol=1,mol<=Length[POI],mol++, (*for every timestep, iterates through the list of Points of interest*)
        point = POI[[mol]]; (*pulls the x,y coordinates from the POI list for the current molecule we are looking at*)
        position=  stackInfo[[point[[1]],point[[2]]]]; (*uses the x,y coords to retrieve the relevant information about the molecule we are looking at*)
        Which[position[[1]]==1, (* CURRENTLY LOOKING AT A DYE*)
            probHop = position[[4]]; (*pulls the probabilities from the position info array*)
            NN = position[[2]];
            probDyeRecomb = position[[3]]*electronDensityPerPart[[point[[1]]]];
            choiceValid=False;
            While[choiceValid==False, (*makes a choice for what happens to the dye we are looking at*)

  If[Total[probHop]+probDyeRecomb<1,
                dyeChoice=RandomChoice[Flatten[{probHop,probDyeRecomb,(1-Total[probHop]-probDyeRecomb)}]-> Table[i,{i,Length[probHop]+2}]];
   , (*if the probabilities are greater than 1, don't allow the option to do nothing (still weights according to size)*)
   dyeChoice=RandomChoice[Flatten[{probHop,probDyeRecomb}]-> Table[i,{i,Length[probHop]+1}]];
   ];
                If[(dyeChoice>Length[probHop])
```

```
        ||((stackInfo[[NN[[dyeChoice]][[1]],NN[[dyeChoice]][[2]]]][[1]]==1 )
                                                &&
oxList[[NN[[dyeChoice]][[1]],NN[[dyeChoice]][[2]]]]==0)

        ||((stackInfo[[NN[[dyeChoice]][[1]],NN[[dyeChoice]][[2]]]][[1]]==2 )
                                                &&
oxList[[NN[[dyeChoice]][[1]],NN[[dyeChoice]][[2]]]]<maxOxState),
                                                choiceValid=True;
                                        ];
                                ];
                                Which[dyeChoice<=Length[probHop], (*Hop to another position*)
                                        hopTarget = NN[[dyeChoice]];(*determines which neighbor has
been selected*)
                                        hopTargetInfo = stackInfo[[ hopTarget[[1]],hopTarget[[2]]]];
                                        hops                                                    =
AppendTo[hops,(timeStepSize/(probHop[[dyeChoice]]))*10^9];
                                        If[hopTargetInfo[[1]]==1,(*Target is a dye*)
                                                oxList[[ hopTarget[[1]],hopTarget[[2]]]]++; (*increases the
oxidation state of the target*)
                                                oxList[[point[[1]],point[[2]]]]--; (*decreases ox state of the
source*)
                                                POI[[mol]]=hopTarget; (* target replaces source in POI
list*)

        If[RecHopPaths==True,DyeTraces[[DyeIds[[mol]]]]=AppendTo[DyeTraces[[DyeIds[[mol]]]],hopTarg
et];];
                                        ];
                                        If[hopTargetInfo[[1]]==2,(*Target is a catalyst*)
                                                initialHopTargetState                   =               oxList[[
hopTarget[[1]],hopTarget[[2]]]]; (*stores starting state for bookeeping*)
                                                oxList[[ hopTarget[[1]],hopTarget[[2]]]]++;(*increases the
oxidation state of the target*)
                                                targetNN=hopTargetInfo[[2]];
                                                For[z=1,z<=Length[targetNN],z++,
                                                        neighbor = targetNN[[z]];
                                                        targNeighInfo                           =
stackInfo[[neighbor[[1]],neighbor[[2]]]];
                                                        pos                                     =
FirstPosition[targNeighInfo[[2]],hopTarget][[1]];

        If[targNeighInfo[[1]]==1,stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopDyetoCat[[ini
tialHopTargetState+1]];];

        If[targNeighInfo[[1]]==2,stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopCattoCat[[init
ialHopTargetState+1]];];
                                                ];
                                                oxList[[point[[1]],point[[2]]]]--; (*decreases ox state of the
source*)
                                                POI[[mol]]=hopTarget; (* target replaces source in POI
list*)

        If[RecHopPaths==True,DyeTraces[[DyeIds[[mol]]]]=AppendTo[DyeTraces[[DyeIds[[mol]]]],hopTarg
et];];
```

```
                                                catalystSpecies[[initialHopTargetState+1]]++;    (*increase
the count of the catalyst type just created*)
                                                If[initialHopTargetState!=0,(*IF hopping to a previously
oxidized catalyst*)
                                                        catalystSpecies[[initialHopTargetState]]--;
                                                        POI=Delete[POI,mol] ; (*If catalyst was previously
oxidized, just remove point hopped from point of interest*)
                                                        If[RecHopPaths==True,DyeIds                    =
Delete[DyeIds,mol];];
                                                        mol--; (*moves backward one step in the loop to
account for deleting the current point in the list*)
                                                ];
                                        ];
                                , (* Branch of the Dye Choice Which Statement*)
                                dyeChoice==Length[probHop]+1,(*RECOMBINE*)
                                        oxList[[point[[1]],point[[2]]]]--; (*reduces the number of charges on
the dye*)
                                        POI=Delete[POI,mol];(*dyes only ever have 1 charge so after it is
gone, the dye is now just a normal point *)
                                        If[RecHopPaths==True,DyeIds = Delete[DyeIds,mol];];
                                        mol--; (*moves backward one step in the loop to account for deleting
the current point in the list*)

                                        dyeRecombinations++;
                                ];
                        ,(* Branch of the Particle Type Which Statement*)
                        position[[1]]==2,(*CURRENTLY LOOKING AT A CATALYST*)
                                catOxState = oxList[[point[[1]],point[[2]]]];
                                catRecombProb                                                          =
position[[3]][[catOxState]]*electronDensityPerPart[[point[[1]]]];
                                probHop= position[[4]];
                                NN = position[[2]];
                                choiceValid=False;

                                If[catOxState == maxOxState, (*Check to see if a Catalyst is fill, if so recombine
catalyst until empty*)
                                        catChoice = Length[probHop]+2;
                                        turnovers++;
                                        catalystSpecies[[maxOxState]]--;
                                        oxList[[point[[1]],point[[2]]]]=0;
                                        For[z=1,z<=Length[NN],z++,
                                                neighbor = NN[[z]];
                                                pos                                                    =
FirstPosition[stackInfo[[neighbor[[1]],neighbor[[2]]]][[2]],point ][[1]];
                                                If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==1,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopDyetoCat[[1]];
                                                ];
                                                If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==2,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopCattoCat[[1]];
                                                ];
                                        ];
```

```
                                        POI=Delete[POI,mol]; (*If the catalyst has no more charges on in, remove it from the list*)
                                        If[RecHopPaths==True,DyeIds = Delete[DyeIds,mol];];
                                        mol--; (*moves backward one step in the loop to account for deleting the current point in the list*)

                                , (*if not full, choose something to do*)
                        While[choiceValid==False,
                                catChoice=RandomChoice[Flatten[{probHop,catRecombProb,(1-Total[probHop]-catRecombProb)}]-> Table[i,{i,Length[probHop]+2}]];
                                If[
                                        Or[catChoice>Length[probHop],

        And[stackInfo[[NN[[catChoice]][[1]],NN[[catChoice]][[2]]]][[1]]==1 ,

        oxList[[NN[[catChoice]][[1]],NN[[catChoice]][[2]]]]==0],

        And[stackInfo[[NN[[catChoice]][[1]],NN[[catChoice]][[2]]]][[1]]==2 ,

        oxList[[NN[[catChoice]][[1]],NN[[catChoice]][[2]]]]<maxOxState]
                                        ],
                                        choiceValid=True;
                                ];
                        ];
    ]; (*end check full catalyst*)

                                If[catChoice<=Length[probHop], (*Hop to a target*)
                                        hopTarget = NN[[catChoice]];(*determines which neighbor has been selected*)
                                        hopTargetInfo = stackInfo[[ hopTarget[[1]],hopTarget[[2]]]];
                                        If[(hopTargetInfo[[1]]==1),(*hopping to a dye*)

                                                oxList[[ hopTarget[[1]],hopTarget[[2]]]]++;(*increase the oxidation state of target*)
                                                oxList[[point[[1]],point[[2]]]]--;(*reduces the oxidation state of the source*)
                                                For[z=1,z<=Length[NN],z++,
                                                        neighbor = NN[[z]];
                                                        pos = FirstPosition[stackInfo[[neighbor[[1]],neighbor[[2]]]][[2]],point ][[1]];

        If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==1,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopDyetoCat[[oxList[[point[[1]],point[[2]]]]+1]];];

        If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==2,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopCattoCat[[oxList[[point[[1]],point[[2]]]]+1]];];
                                                ];
                                                catalystSpecies[[catOxState]]--; (*reduces the number of catalysts of the type just lost*)
                                                POI=Append[POI,hopTarget];(*Adds the target dye to the point of interest list*)
```

```
                                        If[oxList[[point[[1]],point[[2]]]]==0, (*If the catalyst is now
empty, remove it*)
                                                POI=Delete[POI,mol];
                                                mol--; (*moves backward one step in the loop to
account for deleting the current point in the list*)
                                        , (*IF the catalyst is Not empty*)
                                                catalystSpecies[[catOxState-1]]++;    (*count    the
catalyst type just created*)
                                        ];
                                ]; (*END cat hop to a dye*)
                                If[(hopTargetInfo[[1]]==2),(*hopping to a catalyst*)
                                        hopTargetOxState                =                oxList[[
hopTarget[[1]],hopTarget[[2]]]];
                                        oxList[[ hopTarget[[1]],hopTarget[[2]]]]++ ;
                                        oxList[[point[[1]],point[[2]]]]--;
                                        targetNN=hopTargetInfo[[2]];
                                        For[z=1,z<=Length[targetNN],z++,
                                                neighbor = targetNN[[z]];
                                                pos                                             =
FirstPosition[stackInfo[[neighbor[[1]],neighbor[[2]]]][[2]],hopTarget][[1]];

        If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==1,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopDyetoCat[[oxList[[
hopTarget[[1]],hopTarget[[2]]]]]];];

        If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==2,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopCattoCat[[oxList[[
hopTarget[[1]],hopTarget[[2]]]]]];];
                                        ];
                                        For[z2=1,z2<=Length[NN],z2++,
                                                neighbor = NN[[z2]];
                                                pos                                             =
FirstPosition[stackInfo[[neighbor[[1]],neighbor[[2]]]][[2]],point ][[1]];

        If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==1,


        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopDyetoCat[[oxList[[point[[1]],point[[2]]
]]+1]];];

        If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==2,


        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopCattoCat[[oxList[[point[[1]],point[[2]]]
]+1]];];
                                        ];
                                        catalystSpecies[[catOxState]]--;
                                        catalystSpecies[[hopTargetOxState+1]]++;
                                        If[hopTargetOxState==0, (*IF hopping to a newly oxidized
catalyst*)
                                                POI=Append[POI,hopTarget]; (*Adds the target
catalyst to the point of interest list*)
                                        , (*if hopping to an already oxidized catalyst*)
```

```
                                                catalystSpecies[[hopTargetOxState]]--;  (*if a
catalyst was previously oxidized, reduce the count of the kind it previously was*)
                                        ];
                                        If[catOxState==1, (*if the catalyst hopped from is now
depleted (it was at 1)*)
                                                POI=Delete[POI,mol];
                                                mol--;, (*moves backward one step in the loop to
account for deleting the current point in the list*)
                                                catalystSpecies[[catOxState-1]]++; (*if a catalyst
has been reduced without being depleted add one catalyst of the type which it now is*)
                                        ];
                                        ]; (*END cat hop to another cat*)
                                ]; (*END Cat choice = HOP*)

                                If[catChoice==Length[probHop]+1, (*Recombine!*)

                                        oxState = oxList[[point[[1]],point[[2]]]];
                                        catalystRecombinations[[oxState]]++; (*increments the catalyst
recombination type that occurred*)
                                        catalystSpecies[[oxState]]--;
                                        oxList[[point[[1]],point[[2]]]]--;(*reduces the number of charges on
the catalyst*)
                                        For[z=1,z<=Length[NN],z++,
                                                neighbor = NN[[z]];
                                                pos                                             =
FirstPosition[stackInfo[[neighbor[[1]],neighbor[[2]]]][[2]],point ][[1]];
                                                If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==1,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopDyetoCat[[oxList[[point[[1]],point[[2]]
]]+1]];
                                                ];
                                                If[stackInfo[[neighbor[[1]],neighbor[[2]]]][[1]]==2,

        stackInfo[[neighbor[[1]],neighbor[[2]]]][[4]][[pos]]=probHopCattoCat[[oxList[[point[[1]],point[[2]]]
]+1]];
                                                ];

                                        ];
                                        If[oxState==1, (*If the catalyst only had 1 charge left*)
                                                POI=Delete[POI,mol]; (*If the catalyst has no more charges
on in, remove it from the list*)
                                                mol--;(*moves backward one step in the loop to account for
deleting the current point in the list*)
                                                If[RecHopPaths==True,DyeIds = Delete[DyeIds,mol];];
                                        , (*if the cat had more than one charge left*)
                                                catalystSpecies[[oxState-1]]++; (*if the catalyst wasn't
depleted, a reduced but still oxidized type was created*)
                                        ];
                                ];(*END Cat choice recombine*)
                        ]; (*END Molecule type Which Statement*)
                ];(*END OF POI LOOP*)

                If[CWmode==True, (*only excite new dyes if CW Mode is on*)
                        (*possibly excite a new dye this timestep*)
                        If[timestep>=maxTimeSteps,EndCondition=True;];
```

```
                    If[RandomReal[]<=probExcite,
                              excitations++;
                              choiceWeightingThisTime = choiceWeighting;

          For[x=1,x<=Length[POI],x++,choiceWeightingThisTime[[POI[[x,1]],POI[[x,2]]]]=0;];
                              newDye = Flatten[RandomSample[Flatten[choiceWeightingThisTime]->
Flatten[possiblePositions,1],1],1];
                              oxList[[newDye[[1]],newDye[[2]]]]=1;
                              POI = AppendTo[POI,newDye];
                    ];
          ];

   ];(*END OF MAIN LOOP*)
   If[timestep<shortest,shortest=timestep;];
   turnoverTotals=turnoverTotals+turnovers;
   ]; (*END OF TRIALS LOOP*)
  If[RecTurnovers==True,
          turnoverAverage = turnoverTotals*1.0/numTrials;
          turnoverPercent = turnoverAverage/numInitialExcitedDyes;
          turnoverRow =
{tauRecombDye*10^9,tauHopDyetoDye*10^9,HRR,turnoverPercent,turnoverAverage};
          turnoverTable[[ParameterPoint+1]]=turnoverRow;
   ];
  If[RecAnis==True,
          anisTableTrim=Table[anisTable[[i]][[1;;shortest+1]],{i,1,numTrials}];
          anisMean = Mean[anisTableTrim];
   ];
  If[RecTimeBehavior==True,
          timeTableTrim=Table[timeTable[[i]][[1;;shortest+1]],{i,1,numTrials}];
          timeMean = Mean[timeTableTrim];
   ];
  noCatRecombinations = 1.0*noCatRecombinations/numTrials;
  loneChargeRecombinations = 1.0*loneChargeRecombinations/numTrials;
  autoRecombinations = 1.0*autoRecombinations/numTrials;
  (*-----------------------------------------------------------------------------------------------------------------------*)
  (*-----------------------------------------------------------------------------------------------------------------------*)
  (*-----------------------------------------------------------------------------------------------------------------------*)
  (*EXPORTING*)
  If[RecAnis==True,
   anisotropyFileNameTemp = StringTake[anisotropyFileName,StringLength[anisotropyFileName]-
5]<>"_"<>ToString[ParameterPoint]<>StringTake[anisotropyFileName,-5];
   Export[anisotropyFileNameTemp,anisMean];
   ];
  If[RecTimeBehavior==True,
   timeFileNameTemp = StringTake[timeFileName,StringLength[timeFileName]-
5]<>"_"<>ToString[ParameterPoint]<>StringTake[timeFileName,-5];
   Export[timeFileNameTemp,timeMean];
   ];
  If[RecTurnovers==True,Export[turnoverFileName,turnoverTable];];
  sanityCheck = dyeRecombinations+numDyes+turnovers*maxOxState; (*Perform a sanity check to make sure
all bookkeeping adds up*)
  For[ox =1,ox<=maxOxState,ox++, sanityCheck= sanityCheck
+(ox*catalystSpecies[[ox]]+catalystRecombinations[[ox]]); ];
  totalTime = (AbsoluteTime[]-startTime)/60; (*determines how long it took the program to run from start to
final printout*)
```

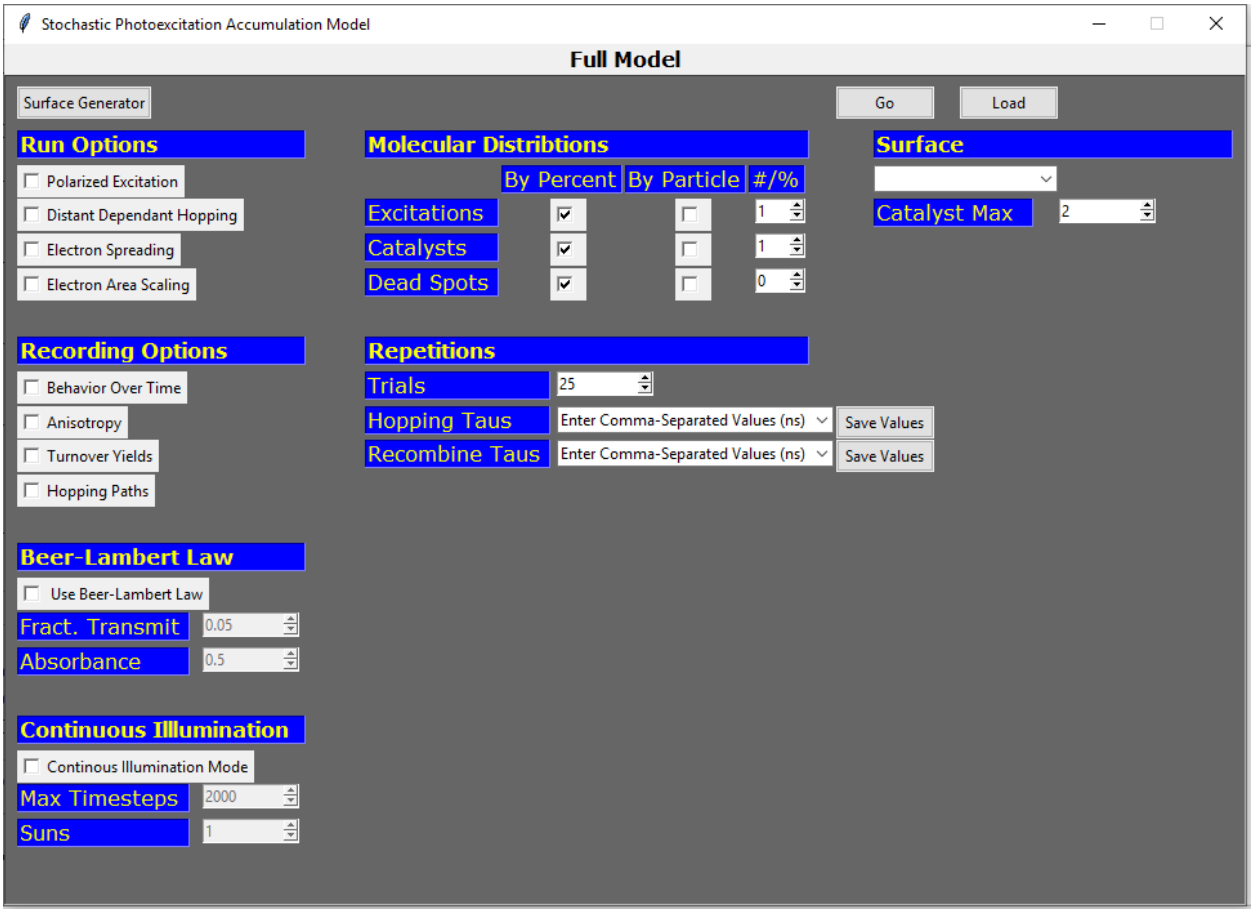
```
   (*POST SIMULATION PRINTOUT*)
   Print["--------------------------------------END RUN-----------------------------------------"];
   Print["Dye Excitations = ", excitations];
   Print["Turnovers = ",turnovers];
   Print["Recombinations from Dyes = ",dyeRecombinations];
   Print["Dyes remaining excited = ",numDyes];
   For[ox=1,ox<=maxOxState,ox++, Print[ToString[ox]," - Oxidized catalyst remaining excited =
",catalystSpecies[[ox]]];];
   For[ox=1,ox<=maxOxState,ox++, Print[ToString[ox]," - Oxidized catalyst recombined =
",catalystRecombinations[[ox]]];];
   Print["Sanity Check: ", sanityCheck," should be equal to ",numInitialExcitedDyes+excitations];
   Print["Total Time Passed = ", Floor[(AbsoluteTime[]-startTime)/60]," minutes,
",PaddedForm[Mod[(AbsoluteTime[]-startTime),60],{4,2}]," seconds"];

   allHops[[ParameterPoint]]=hops;
   If[RecTimeBehavior==True,allTimes[[ParameterPoint]]=timeMean;];
   ]; (*END OF HOP CONSTANT LOOP*)
 ]; (*END OF RECOMBINE CONSTANT LOOP*)
```

APPENDIX C. Modeling Guide in Python

This guide will serve as a user's manual for running Monte Carlo Simulations using the python model. This guide will not be as thorough as the one for the Mathematica code because most of the reasoning behind it is the same. Instead it will highlight the differences between the two. For more detailed explanations of the code please refer to the inline comments and for more detailed reasoning behind some of the decisions made, please see the Mathematica code guide.

Overview

At a high level, this program is used to model the accumulation of electron-holes on catalysts anchored to dye-sensitized nanoparticles. This accumulation is the result of these holes hopping between surface anchored molecules through self-exchange electron transfer processes as they as they move across the surface and eventually trap on catalysts sites. As with the Mathematica version of this program, there are two main functional parts: the creation of a surface, and the running of the model on a surface. This breakdown allows a surface created by the first part to be used over and over under different experimental conditions. One of the main differences with Python is that there are actually three pieces of code: the surface builder, the model runner, and the GUI interface which controls the other two. This allows conditions to be tested easily without having to worry about making typos while scrolling through 1000 lines of code to try and change one parameter here and another one there.



The GUI

The GUI has 2 panels. The first panel is the **Full Model** panel and controls the running of the model on a pregeneratred surface. The input parameters are sorted under relevant headings.

The first heading are global run options. These are four Boolean values which by default start as False. Checking any of these boxes will set them to true when the *Go* is pressed. *Polarized Excitation* controls whether an anisotropic distribution of initial excited states will be created based on using polarized light for excitation. *Distant Dependent Hopping* determines whether hopping to neighbor values will be weighted by distances

between those neighbors. If this option is selected, the hopping probability set according to the *Hopping Taus* will be modified based on inter-neighbor distances but the average hopping probability for the whole modeled surface will remain unchanged. *Electron Spreading* and *Electron Area Scaling* are non-physical, non-rigorous approximations which aim to correct for lack of modeled electron behavior in this model. Area Scaling will scale the electron density a given particle receives by that particle's surface area while the spreading function will share some electron density from a given particle with the neighboring particles based on their relative sizes. More details on these functions can be found in the other guide as well as in in-line comments.

The second heading controls options for what data are recorded during a simulation. *Behavior* over time will record many different statistics every timestep. These statistics will be averaged over a number of trials and truncated to the shortest trial. They will be stored in separate excel files by parameter point. *Anisotropy* will record the value of the polarization anisotropy of the system at every timestep. This value is averaged over trials and truncated to the shortest trial. An Excel file is created for these data for each parameter point. *Turnover Yields* records the fraction of initial photoexcitations which ultimately contribute to catalyst turnover in both number and percent. These values are averaged over a number of trials and exported as a large table organized by parameter point. *Hopping Paths* will record the path taken by every photoexcitation from start to finish. This will be overwritten each trial or parameter point and should only be done a single time to make a hopping video.

The *Beer-Lambert Law* section controls initial excitation distribution and whether a Beer-Lambert distribution should be applied to the weights of the positions for

photoexcitation. If the box is checked, the will be applied according to the value entered in the *Fractional Transmittance* box. This box may be filled directly or else the *Absorbance* box can be filled and the corresponding value will be calculated.

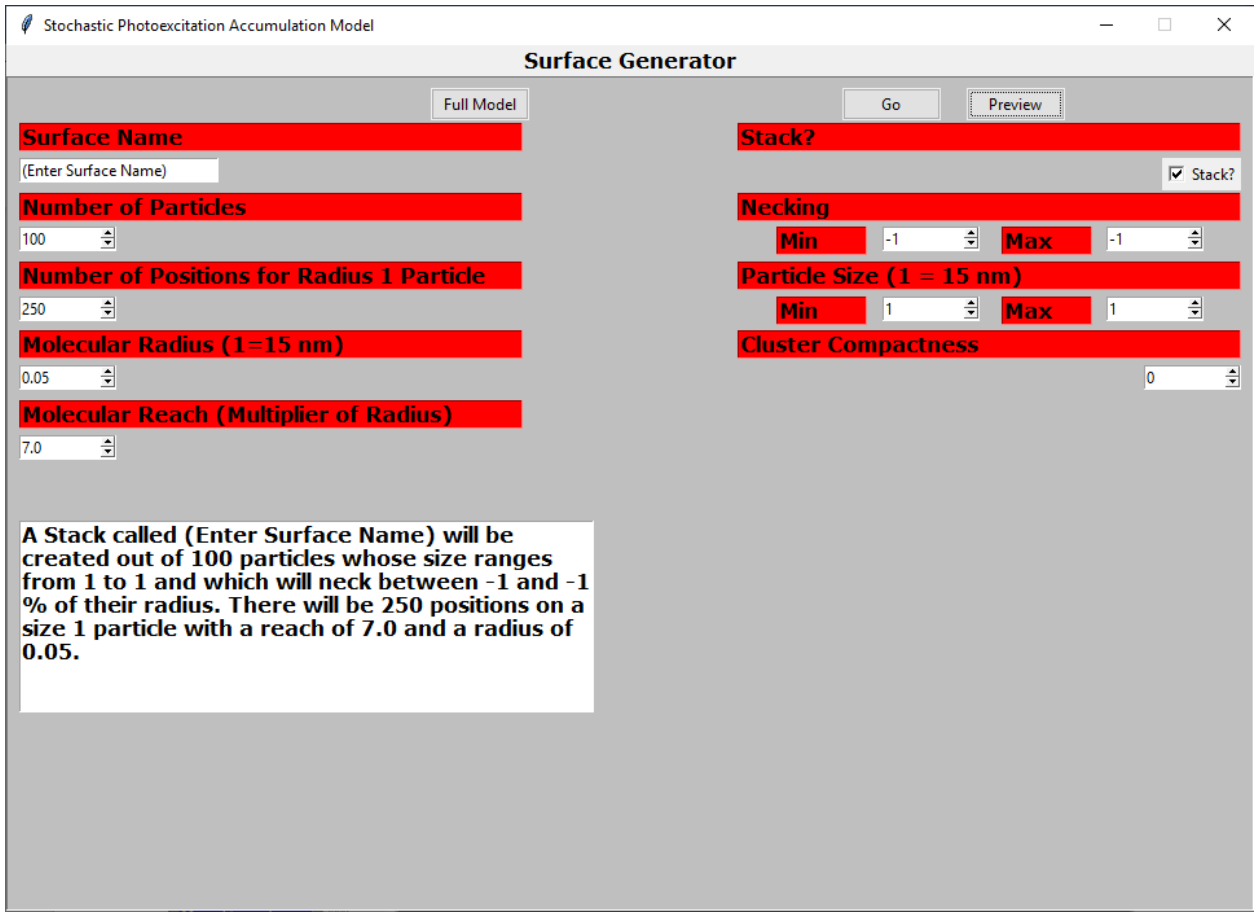
The *Continuous Illumination* section controls whether a simulation will include the possibility for additional photoexcitations to occur after each timestep (as under sun light illumination) or whether the only photoexcitation events will occur as timestep 0 (as in a pulsed laser experiment). If the box is checked there will an additional probability for photoexcitation events to occur based on the value in the *Suns* box. Additionally, an ending timestep will be set by *Max Timesteps* because having zero excited dyes is no longer a useful ending condition.

The *Molecular Distributions* section controls how photoexcitations, catalysts, and dead spots are distributed over the modeling surface. This essentially allows for 4 options for each and the number entered in the *#/%* column will be handled according to the first two columns for each row. This means that a specific number of excitations could be placed on each particle, a specific number of excitations could be placed over the whole surface (*By Particle* left unchecked), a certain percentage of each particle's molecules could be excited initially (only really relevant if there are different sizes of particles), or that a certain percentage of molecules will be excited over the whole film. This last option is the default option. These options are handled the same for each molecular species (excitation, dead spot, and catalyst) and all default to a percent-based usage over the whole film. It should also be noted that when using percentages that the percent entered will correspond to the percent of the total possible when it comes time to assign positions of each of the molecular species. During the simulation, dead spots are assigned first, followed by

catalysts, followed by excitations. This means that, for example, one could input 50% to be the total surface coverage for all 3 species and that 50% of all sites would be dead spots, 25% of all sites (50% of all non-dead spots) would be catalysts, and 12.5% of sites (50% of what is left) would be initially excited dyes.

The *Repetitions* section controls the loops that the simulation will run through. The *Trials* look is how many identical repetitions of each parameter point to make. In general, results from separate trials loops will be averaged together. The *Hopping Taus* and *Recombine Taus* each take in a string of comma separated values which are in units of nanometers. The simulation will run through all combinations of these values with each combination designated as a parameter point. For example if 3 Hop values are given and 2 Recombine values are given the simulation will run 6 times total (H1R1, H2R1, H3R1, H1R2, H2R2, H3R2). These values are the hopping and recombination time constants use in the simulation for an excited state (before weighting from other factors). If you plan to repeatedly use a set of values, you can save the set using the *Save Values* button. They will then appear in the corresponding drop-down menu the next time the program is opened. However, it should be noted that only one such set of values can be saved per running of the program. If many sets of value need to be entered in the drop-down menus quickly, the excel file containing those values can be found in the SavedSettings folder and edited manually.

The *Surface* section is where a pregenerated surface can be chosen from the drop-down menu. Alternatively, you can type in the name of the surface you want. The drop-down menu contains the scanned results of all surfaces in the Surfaces folder and so is a complete list of all surfaces it is possible to load. If something you want is not on this list,



typing it in won't work either. Below that is the entry for what the maximum oxidation state of a catalyst should be. This doesn't really belong here but also doesn't really belong in any other sections, so this seemed like where it fit the best.

Above the Surface section are two buttons. The *Go* button starts a simulation with the selected surface and with the input parameters. The *Load* Button loads a selected surface and displays a plot of it so you can visualize what is being loaded.

The second panel is the **Surface Generation** panel. This takes a number of parameters and can be used to create a surface by hitting the *Go* button. The preview button will generate a text description of the surface to be generated for proof reading if that is desired. It should be noted that if you run a surface generation, you must exit out of the GUI and reopen for it to appear in the surface drop-down menu.

Most of the variables here are fairly straightforward but a few are not so obvious. The *Surface Name* will be the file name used as well as the identifier used in the Run model panel. The *Number of Particles* is simply the number of particles to be used in the modeled surface. *Number of Positions* is the number of molecules that will be able to bind to a particle. This is the number used for a radius 1 particle and this value is scaled for particles of different sizes to maintain the same point density. The *Molecular Radius* is the size of the individual molecules and should be entered in units where 1 = 15 nm. This is done so that particles (which are meant to be 15nm in radius) can have a radius of 1 to keep certain math simple. This value should be near the van der waals radius of a modeled molecule. The default value is probably fine is almost all cases. *Molecular Reach* is a value that, when multiplied by the molecular radius gives the length (still in units of 1 = 15 nm) that two adjacent molecules must be within to be considered neighbors. All molecules within molecular radius* molecular reach of a given molecule are neighbors of that molecule. *Stack?* Is a true or false value which is by default true. If true, each particle added to the surface will be added to the bottom of the surface directly downward whereas if the value is false, particles may be added to the surface in any direction resulting in a cluster rather than a stack. *Necking* takes both a minimum and maximum value and ensures that every particle necks with a least one other particle by the minimum amount (in order to be attached to the surface) and that the overlap between any two molecules is no more than the maximum value (particles cannot be on top of each other). These values represent a fraction of a particle's radius that can overlap with another particle. If two particles have a necking value of 0, their surfaces are exactly touching each other, and any negative values result in particles that are not in contact at all. Meanwhile, a particle with a necking value of

1 (100%) with another will have its entire radius overlapped by the other particle such that the surface of the other particle passed through the center of the first particle. *Particle Size* also takes a range of values which set the range of possible particle sizes where 1 = 15 nm. These values specify particle radii and the distribution of sizes will be normal between the minimum and maximum specified values. *Cluster Compactness* is only relevant if making a cluster and not a stack. This value controls how tightly packed the cluster will be with large values (10) resulting in tight spherical clusters and large negative values (-10) resulting in long dendritic clusters. When choosing where to place a new particle in a cluster, first an existing particle is chosen to "grow off of". After this choice is made, a proposal particle is generated and tested against necking constraints, if it fails a new selection is made. This continues until the desired number of particles has been added and Cluster Compactness controls the weighting of the choice for a particle to grow off. More specifically, the choice for the growing particle is weighted by X^(-Cluster Compactness) where X is the particle number (order of addition to the cluster). In this way, large values of Cluster Compactness favor growing off the oldest particles in the surface while large negative values favor growing off the most recently added particles.

The Surface Builder

One of the biggest changes in the code between Mathematica and Python versions is that Python supports objects and other structures. So instead of organizing all the information about the surface in a table full of tables is it organized on a *Surface* object. This has many relevant fields stored on the object including *heightMax, heightMin, thickness, name, totalSites, numParticles*, and *particles*. Most of these are fairly self explanatory and correspond to the values calculated and stored in the stack file by the

Mathematica surface builder. The *particles* field is an array of *Particle* objects. Each of these *Particle* objects has many relevant fields including *center, radius, numPositions, particleNNs* (Particle Nearest Neighbors), *particleArea* (used for electron scaling functions), *particleRegions* (used for electron spreading functions), and *molecules.* The *molecules* field is, in turn, an array of *Molecule* objects with the relevant fields: *radii* (van Der Waals), *XYZ* (array of 3D coordinates), *inclinationAngle* (used for polarization effects), *anisotropyContribution*, *OxState*, typeOf, *NNs* (Nearest Neighbors), *NNdists* (distances to each Nearest Neighbor), *HopRates* (hopping rates to each nearest neighbor which may vary if distance dependent hopping is enabled), *RecombRate*, and *percolationZone* (a zone number assigned to sets of mutually connected molecules). By making use of structures the stored parameters can be accessed much more simply and in a self documenting way. For example, if you wanted to determine the type of the molecule at position 5 on the 3rd particle the Mathematica model requires you to write stackInfo[[3]][[5]][[1]] and simply know that the first entry in each molecule's stack info table is the type. In Python you would write SURF.particles[3].molecules[5].typeOf. While this is overall longer, is very clear what is being accessed which makes troubleshooting much easier.

On the whole, the general procedure follows the same pattern. First the designated number of particles are added to the surface one at a time. Once the particles are set, an array of positions for that particle is generated such that the positions are close to equally spaced out of over the particle's surface. Once positions have been established, particles neighboring each other are determined followed by the determination of molecular neighbors. Once neighbors are set global statistics, such at the number of molecules per

surface and the height max and height min are determined and stored. The surface is exported as a binary data stream as a “SURF” filetype.

The Full Model

The full model begins by importing the surface from the specified SURF file and then proceeds very similarly to the Mathematica version. One of the main syntactic differences is that Python allows for the iteration over any sorts of arrays and not just a series of numeric values. There are a number of cases where this is made use of an instead of iterating over the number of particles on a surface, for example, an iteration is carried out over the particles themselves. For example, “For x in SURF.particles” starts a loop where x is, at each iteration of the loop, a particle object from SURF.particles, not merely a counter counting out the nth particle. This helps keep things concise and somewhat easier to read.

Other than syntactic changes, the simulation proceeds very similarly as in the Mathematica version and so for further detail please see that model’s description. The order of some functions is different but the logical flow is still all the same.

APPENDIX D. Model in Python

The GUI

```
import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg,
NavigationToolbar2TkAgg
from matplotlib.figure import Figure
import random as rand
import matplotlib.pyplot as plt
import math
import statistics
import glob
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import os
import pickle
import csv
import surfaceBuilder as SB #this needs to be in the same folder
import RunModel as RM #this needs to be in the same folder
import tkinter as tk
from tkinter import ttk

FONT= ("Verdana", 12, "bold")
labelFont = ("Verdana", 12)

#loads a selected surface to create a visual preview. The aspect ratio will be wrong be
connectivity will be correct
def loadSurface(self,name):
   filepath = os.getcwd()
   with open(filepath+"/Surfaces/"+name+".SURF", "rb") as fileIn:
      unpickler = pickle.Unpickler(fileIn)
      SURF = unpickler.load()
   SB.plotSurface(SURF)
   doneText = tk.Text(self,height = 1, width = 40, font = FONT,wrap = "word")
   doneText.insert(1.0,"Done Importing "+name+"!")
   doneText.place(relx=0.51,rely=0.55,anchor='nw')
   print("Done Importing "+name+"!")

#runs the model with all input parameters after loading specified surface with pickle
def
RunFullModel(self,name,AnisTF,DDHTF,electSpreadTF,electScaleTF,timeRecTF,AnisRecTF,
TurnoverRecTF,hoppingPathTF,CWModeTF,maxTimeNum,sunsNum,BLTF,fractTransNum,
excitePCTTF,catPCTTF,DSPCTTF,excitePartTF,catPartTF,DSPartTF,exciteNum,catNum,DSN
um,trialsNum,hops,recombs,catMaxNum):
```

```
    filepath = os.getcwd()
    with open(filepath+"/Surfaces/"+name+".SURF", "rb") as fileIn:
        unpickler = pickle.Unpickler(fileIn)
        SURF = unpickler.load()

RM.RunModel(SURF,name,bool(AnisTF),bool(DDHTF),bool(electSpreadTF),bool(electScale
TF),bool(timeRecTF),bool(AnisRecTF),bool(TurnoverRecTF),bool(hoppingPathTF),bool(C
WModeTF),int(maxTimeNum),float(sunsNum),bool(BLTF),float(fractTransNum),bool(exci
tePCTTF),bool(catPCTTF),bool(DSPCTTF),bool(excitePartTF),bool(catPartTF),bool(DSPart
TF),float(exciteNum),float(catNum),float(DSNum),int(trialsNum),hops,recombs,int(catMax
Num))

#runs the surfacebuilder with all input parameters
def
runSurfaceGeneration(self,name,r1Pos,molRad,molReach,neckMin,neckMax,sizeMin,sizeM
ax,numPart,stack,compactness):

SB.buildSurface(str(name),int(r1Pos),float(molRad),float(neckMin),float(neckMax),float(si
zeMin),float(sizeMax),int(numPart),bool(stack),float(compactness),float(molReach))
    doneText = tk.Text(self,height = 1, width = 40, font = FONT,wrap = "word")
    doneText.insert(1.0,name+" Created Successfully!")
    doneText.place(relx=0.51,rely=0.55,anchor='nw')

#creates a textbox describing the input setting for proofreading before running surface
generation
def
previewSurfaceGeneration(self,name,r1Pos,molRad,molReach,neckMin,neckMax,sizeMin,si
zeMax,numPart,stack,compactness):
    if stack == True:
        stackString = "Stack"
    else:
        stackString = "Cluster"
    outString = "A "+stackString+" called "+name+" will be created out of "+str(numPart)+"
particles whose size ranges from "+str(sizeMin)+" to "+str(sizeMax)+" and which will neck
between "+str(neckMin)+" and "+str(neckMax)+" % of their radius. There will be
"+str(r1Pos)+" positions on a size 1 particle with a reach of "+str(molReach)+" and a radius
of "+str(molRad)+"."
    modelDescription = tk.Text(self, height = 8, width = 40, font =FONT,wrap="word")
    modelDescription.insert(1.0,outString)
    modelDescription.place(relx=0.01,rely=0.55,anchor='nw')


class Model(tk.Tk): #mostly black magic that makes the GUI work. Don't mess with it.
    def __init__(self, *args, **kwargs):
        tk.Tk.__init__(self, *args, **kwargs)
```

```
        tk.Tk.wm_title(self, "Stochastic Photoexcitation Accumulation Model") #controls title at
the top of the window

        container = tk.Frame(self)
        container.pack(side="top", fill="both", expand = False)
        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)
        self.frames = {}
        for F in (SurfaceGenerator, FullModel):
            frame = F(container, self)
            self.frames[F] = frame
            frame.grid(row=0, column=0, sticky="nsew")
        self.resizable(False,False)
        self.show_frame(FullModel) #starts the GUI on the Full Model tab
    def show_frame(self, cont):
        frame = self.frames[cont]
        frame.tkraise()


class SurfaceGenerator(tk.Frame): #controls the surface generation tab

    def __init__(self, parent, controller):#black magic
        tk.Frame.__init__(self,parent) #some more black magic
        #adds title to tab
        label = tk.Label(self, text="Surface Generator", font=FONT)
        label.pack()
        #controls window size
        modelWindowSize = tk.Text(self,height = 40,width = 120,bg = "gray75")
        modelWindowSize.pack()
        #adds Full model button
        button2 = ttk.Button(self, text="Full Model",command=lambda:
controller.show_frame(FullModel))
        button2.place(relx=.34,rely=0.05,anchor='nw')
        #adds Go button
        button4 = ttk.Button(self,
text="Go",command=lambda:runSurfaceGeneration(self,surfaceName.get(),r1Num.get(),m
olradNum.get(),molreachNum.get(),neckingMinNum.get(),neckingMaxNum.get(),sizeMinN
um.get(),sizeMaxNum.get(),partNum.get(),stackedTF.get(),compactNum.get()))
        button4.place(relx=.67,rely=0.05,anchor='nw')
        #adds preview Button
        button5 = ttk.Button(self,
text="Preview",command=lambda:previewSurfaceGeneration(self,surfaceName.get(),r1Nu
m.get(),molradNum.get(),molreachNum.get(),neckingMinNum.get(),neckingMaxNum.get(),
sizeMinNum.get(),sizeMaxNum.get(),partNum.get(),stackedTF.get(),compactNum.get()))
        button5.place(relx=.77,rely=0.05,anchor='nw')
        #adds Surface Name Section
```

```
        surfaceNameLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        surfaceNameLabel.insert(1.0,"Surface Name")
        surfaceNameLabel.place(relx=0.01,rely=0.09,anchor='nw')
        surfaceName = tk.Entry(self,width=25)
        surfaceName.insert(0,"(Enter Surface Name)")
        surfaceName.place(relx=0.01,rely=0.13,anchor='nw')
        #adds Stack? section
        stackedLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        stackedLabel.insert(1.0,"Stack?")
        stackedLabel.place(relx=0.99,rely=0.09,anchor='ne')
        stackedTF = tk.BooleanVar()
        s = tk.Checkbutton(self,text = "Stack?",variable = stackedTF)
        s.place(relx=0.99,rely=0.13,anchor='ne')
        s.select()
        #adds Number of Particles section
        partLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        partLabel.insert(1.0,"Number of Particles")
        partLabel.place(relx=0.01,rely=0.17,anchor='nw')
        partNum = tk.Spinbox(self,from_=1, to=200,width=10)
        partNum.delete(0,'end')
        partNum.insert(1,100)
        partNum.place(relx=0.01,rely=0.21,anchor='nw')
        #adds the Necking section
        neckingLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        neckingLabel.insert(1.0,"Necking")
        neckingLabel.place(relx=0.99,rely=0.17,anchor='ne')
        neckingMinLabel = tk.Text(self,height=1,width=6,background = 'red', font=FONT)
        neckingMinLabel.insert(1.0,"Min")
        neckingMinLabel.place(relx=0.69,rely=0.21,anchor='ne')
        neckingMaxLabel = tk.Text(self,height=1,width=6,background = 'red', font=FONT)
        neckingMaxLabel.insert(1.0,"Max")
        neckingMaxLabel.place(relx=0.87,rely=0.21,anchor='ne')

        neckingMinNum = tk.Spinbox(self,from_=-1, to=2,width=10,increment = 0.25,format =
"%.2f")
        neckingMinNum.delete(0,'end')
        neckingMinNum.insert(1,-1)
        neckingMinNum.place(relx=0.78,rely=0.21,anchor='ne')

        neckingMaxNum = tk.Spinbox(self,from_=-1, to=2,width=10,increment = 0.25,format =
"%.2f")
        neckingMaxNum.delete(0,'end')
        neckingMaxNum.insert(1,-1)
        neckingMaxNum.place(relx=0.96,rely=0.21,anchor='ne')
        #adds the Particle Size section
        sizeLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
```

```
        sizeLabel.insert(1.0,"Particle Size (1 = 15 nm)")
        sizeLabel.place(relx=0.99,rely=0.25,anchor='ne')
        sizeMinLabel = tk.Text(self,height=1,width=6,background = 'red', font=FONT)
        sizeMinLabel.insert(1.0,"Min")
        sizeMinLabel.place(relx=0.69,rely=0.29,anchor='ne')
        sizeMaxLabel = tk.Text(self,height=1,width=6,background = 'red', font=FONT)
        sizeMaxLabel.insert(1.0,"Max")
        sizeMaxLabel.place(relx=0.87,rely=0.29,anchor='ne')

        sizeMinNum = tk.Spinbox(self,from_=0.0, to=10,width=10,increment = 0.5,format =
"%.1f")
        sizeMinNum.delete(0,'end')
        sizeMinNum.insert(1,1)
        sizeMinNum.place(relx=0.78,rely=0.29,anchor='ne')

        sizeMaxNum = tk.Spinbox(self,from_=0.0, to=10,width=10,increment = 0.5,format =
"%.1f")
        sizeMaxNum.delete(0,'end')
        sizeMaxNum.insert(1,1)
        sizeMaxNum.place(relx=0.96,rely=0.29,anchor='ne')
        #adds the Particle Radius Section
        r1Label = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        r1Label.insert(1.0,"Number of Positions for Radius 1 Particle")
        r1Label.place(relx=0.01,rely=0.25,anchor='nw')
        r1Num = tk.Spinbox(self,from_=1, to=2000,width=10)
        r1Num.delete(0,'end')
        r1Num.insert(1,250)
        r1Num.place(relx=0.01,rely=0.29,anchor='nw')
        #adds the molecule radius section
        molradLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        molradLabel.insert(1.0,"Molecular Radius (1=15 nm)")
        molradLabel.place(relx=0.01,rely=0.33,anchor='nw')
        molradNum = tk.Spinbox(self,from_=0.01, to=0.1,width=10,increment =0.005)
        molradNum.delete(0,'end')
        molradNum.insert(1,0.05)
        molradNum.place(relx=0.01,rely=0.37,anchor='nw')
        #adds the molecule reach section
        molreachLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
        molreachLabel.insert(1.0,"Molecular Reach (Multiplier of Radius)")
        molreachLabel.place(relx=0.01,rely=0.41,anchor='nw')
        molreachNum = tk.Spinbox(self,from_=1, to=20,width=10)
        molreachNum.delete(0,'end')
        molreachNum.insert(1,7.0)
        molreachNum.place(relx=0.01,rely=0.45,anchor='nw')
        #adds the cluster compactness section
        compactLabel = tk.Text(self,height=1,width=35,background = 'red', font=FONT)
```

```
        compactLabel.insert(1.0,"Cluster Compactness")
        compactLabel.place(relx=0.99,rely=0.33,anchor='ne')
        compactNum = tk.Spinbox(self,from_=-10, to=10,width=10)
        compactNum.delete(0,'end')
        compactNum.insert(1,0)
        compactNum.place(relx=0.99,rely=0.37,anchor='ne')



class FullModel(tk.Frame): #controls the Full Model tab
    def __init__(self, parent, controller): #black magic
        tk.Frame.__init__(self, parent)#more black magic
        #adds title to tab
        label = tk.Label(self, text="Full Model", font=FONT)
        label.pack()
        labelBGColor = "Blue"
        labelTextColor = "Yellow"
        files = glob.glob('Surfaces/*.SURF')#scans current directory for files of ".SURF" type
using glob
        surfaceNames = [name[9:-5] for name in files] #creates a list of the Names of the
surfaces from files

        hopList = importHops() #imports lists of previously saved hopping rates
        recombList = importRecombs()#imports lists of previously saved recombination rates
        #controls  windows size
        modelWindowSize = tk.Text(self,height = 40,width = 120,bg = "gray40")
        modelWindowSize.pack()
        #adds the Surface Generation Button
        button = ttk.Button(self, text="Surface Generator",command=lambda:
controller.show_frame(SurfaceGenerator))
        button.place(relx=.01,rely=0.05,anchor='nw')
        #adds the Go button
        button4 = ttk.Button(self,
text="Go",command=lambda:RunFullModel(self,surface.get(),AnisTF.get(),DDHTF.get(),ele
ctSpreadTF.get(),electScaleTF.get(),timeRecTF.get(),AnisRecTF.get(),TurnoverRecTF.get(),
hoppingPathTF.get(),CWModeTF.get(),maxTimeNum.get(),sunsNum.get(),BLTF.get(),fract
TransNum.get(),excitePCTTF.get(),catPCTTF.get(),DSPCTTF.get(),excitePartTF.get(),catPar
tTF.get(),DSPartTF.get(),exciteNum.get(),catNum.get(),DSNum.get(),trialsNum.get(),hops.g
et(),recombs.get(),catMaxNum.get()))
        button4.place(relx=.67,rely=0.05,anchor='nw')
        #adds the Load Button
        button5 = ttk.Button(self,
text="Load",command=lambda:loadSurface(self,surface.get()))
        button5.place(relx=.77,rely=0.05,anchor='nw')
        #adds the Run Options Section
```

```
        heading1 = tk.Text(self,height=1,width=20,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading1.insert(1.0,"Run Options")
        heading1.place(relx=0.01,rely=0.1,anchor='nw')

        AnisTF=tk.BooleanVar()
        Anischeck = tk.Checkbutton(self,text= "Polarized Excitation", variable = AnisTF)
        Anischeck.place(relx=0.01,rely=0.14,anchor='nw')
        DDHTF=tk.BooleanVar()
        DDHcheck = tk.Checkbutton(self,text= "Distant Dependant Hopping", variable =
DDHTF)
        DDHcheck.place(relx=0.01,rely=0.18,anchor='nw')
        electSpreadTF=tk.BooleanVar()
        electrSpreadcheck = tk.Checkbutton(self,text= "Electron Spreading", variable =
electSpreadTF)
        electrSpreadcheck.place(relx=0.01,rely=0.22,anchor='nw')
        electScaleTF=tk.BooleanVar()
        electScalecheck = tk.Checkbutton(self,text= "Electron Area Scaling", variable =
electScaleTF)
        electScalecheck.place(relx=0.01,rely=0.26,anchor='nw')
        #adds the Recording Options Section
        heading2 = tk.Text(self,height=1,width=20,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading2.insert(1.0,"Recording Options")
        heading2.place(relx=0.01,rely=0.34,anchor='nw')
        timeRecTF=tk.BooleanVar()
        timeReccheck = tk.Checkbutton(self,text= "Behavior Over Time", variable = timeRecTF)
        timeReccheck.place(relx=0.01,rely=0.38,anchor='nw')
        AnisRecTF=tk.BooleanVar()
        AnisReccheck = tk.Checkbutton(self,text= "Anisotropy", variable = AnisRecTF)
        AnisReccheck.place(relx=0.01,rely=0.42,anchor='nw')
        TurnoverRecTF=tk.BooleanVar()
        TurnoverReccheck = tk.Checkbutton(self,text= "Turnover Yields", variable =
TurnoverRecTF)
        TurnoverReccheck.place(relx=0.01,rely=0.46,anchor='nw')
        hoppingPathTF=tk.BooleanVar()
        hoppingPathcheck = tk.Checkbutton(self,text= "Hopping Paths", variable =
hoppingPathTF)
        hoppingPathcheck.place(relx=0.01,rely=0.50,anchor='nw')
        #adds the Continuous Illumination section
        heading3 = tk.Text(self,height=1,width=20,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading3.insert(1.0,"Continuous Illumination")
        heading3.place(relx=0.01,rely=0.78,anchor='nw')
        CWModeTF=tk.BooleanVar()
```

```
        CWModecheck = tk.Checkbutton(self,text= "Continuous Illumination Mode", variable =
CWModeTF, command=lambda: toggleCWStates(maxTimeNum,sunsNum))
        CWModecheck.place(relx=0.01,rely=0.82,anchor='nw')

        maxTimeLabel = tk.Text(self,height=1,width=13,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        maxTimeLabel.insert(1.0,"Max Timesteps")
        maxTimeLabel.place(relx=0.01,rely=0.86,anchor='nw')
        maxTimeNum = tk.Spinbox(self,from_=0, to=20000,width=10)
        maxTimeNum.delete(0,'end')
        maxTimeNum.insert(1,2000)
        maxTimeNum.place(relx=0.16,rely=0.86,anchor='nw')
        maxTimeNum['state']='disabled'

        sunsLabel = tk.Text(self,height=1,width=13,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        sunsLabel.insert(1.0,"Suns ")
        sunsLabel.place(relx=0.01,rely=0.90,anchor='nw')
        sunsNum = tk.Spinbox(self,from_=0, to=1000,width=10)
        sunsNum.delete(0,'end')
        sunsNum.insert(1,1)
        sunsNum.place(relx=0.16,rely=0.90,anchor='nw')
        sunsNum['state']='disabled'

        #Adds the Beer's Law section
        heading5 = tk.Text(self,height=1,width=20,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading5.insert(1.0,"Beer-Lambert Law")
        heading5.place(relx=0.01,rely=0.58,anchor='nw')
        BLTF=tk.BooleanVar()
        BLcheck = tk.Checkbutton(self,text= " Use Beer-Lambert Law", variable = BLTF,
command=lambda: toggleBLStates(fractTransNum,absNum))
        BLcheck.place(relx=0.01,rely=0.62,anchor='nw')

        fractTransLabel = tk.Text(self,height=1,width=13,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        fractTransLabel.insert(1.0,"Fract. Transmit")
        fractTransLabel.place(relx=0.01,rely=0.66,anchor='nw')
        fractTransNum = tk.Spinbox(self,from_=0.000001,
to=1,width=10,increment=0.000001,command=
lambda:updateAbs(absNum,fractTransNum))
        fractTransNum.delete(0,'end')
        fractTransNum.insert(1,0.05)
        fractTransNum.place(relx=0.16,rely=0.66,anchor='nw')
        fractTransNum['state']='disabled'
```

```
        absLabel = tk.Text(self,height=1,width=13,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        absLabel.insert(1.0,"Absorbance")
        absLabel.place(relx=0.01,rely=0.70,anchor='nw')
        absNum = tk.Spinbox(self,from_=0, to=6,width=10,increment=0.1,command=
lambda:updatefractTrans(absNum,fractTransNum))
        absNum.delete(0,'end')
        absNum.insert(1,0.5)
        absNum.place(relx=0.16,rely=0.70,anchor='nw')
        absNum['state']='disabled'
        #adds the molecule distribution section
        heading4 = tk.Text(self,height=1,width=31,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading4.insert(1.0,"Molecular Distributions")
        heading4.place(relx=0.29,rely=0.1,anchor='nw')
        exciteLabel = tk.Text(self,height=1,width=10,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        exciteLabel.insert(1.0,"Excitations ")
        exciteLabel.place(relx=0.29,rely=0.18,anchor='nw')
        catLabel = tk.Text(self,height=1,width=10,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        catLabel.insert(1.0,"Catalysts ")
        catLabel.place(relx=0.29,rely=0.22,anchor='nw')
        DSLabel = tk.Text(self,height=1,width=10,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        DSLabel.insert(1.0,"Dead Spots ")
        DSLabel.place(relx=0.29,rely=0.26,anchor='nw')
        PCTLabel = tk.Text(self,height=1,width=9,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        PCTLabel.insert(1.0,"By Percent ")
        PCTLabel.place(relx=0.40,rely=0.14,anchor='nw')
        partLabel = tk.Text(self,height=1,width=9,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        partLabel.insert(1.0,"By Particle ")
        partLabel.place(relx=0.50,rely=0.14,anchor='nw')
        valueLabel = tk.Text(self,height=1,width=4,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        valueLabel.insert(1.0,"#/%")
        valueLabel.place(relx=0.60,rely=0.14,anchor='nw')
        excitePCTTF=tk.BooleanVar()
        excitePCTcheck = tk.Checkbutton(self, variable = excitePCTTF)

        excitePCTcheck.place(relx=0.44,rely=0.18,anchor='nw')
        excitePCTcheck.select()
        catPCTTF=tk.BooleanVar()
        catPCTcheck = tk.Checkbutton(self, variable = catPCTTF)
```

```
        catPCTcheck.place(relx=0.44,rely=0.22,anchor='nw')
        catPCTcheck.select()

        DSPCTTF=tk.BooleanVar()
        DSPCTcheck = tk.Checkbutton(self, variable = DSPCTTF)
        DSPCTcheck.place(relx=0.44,rely=0.26,anchor='nw')
        DSPCTcheck.select()

        excitePartTF=tk.BooleanVar()
        excitePartcheck = tk.Checkbutton(self, variable = excitePartTF)
        excitePartcheck.place(relx=0.54,rely=0.18,anchor='nw')
        catPartTF=tk.BooleanVar()
        catPartcheck = tk.Checkbutton(self, variable = catPartTF)
        catPartcheck.place(relx=0.54,rely=0.22,anchor='nw')
        DSPartTF=tk.BooleanVar()
        DSPartcheck = tk.Checkbutton(self, variable = DSPartTF)
        DSPartcheck.place(relx=0.54,rely=0.26,anchor='nw')
        exciteNum = tk.Spinbox(self,from_=0, to=2000,width=4)
        exciteNum.delete(0,'end')
        exciteNum.insert(1,1)
        exciteNum.place(relx=0.605,rely=0.18,anchor='nw')
        catNum = tk.Spinbox(self,from_=0, to=2000,width=4)
        catNum.delete(0,'end')
        catNum.insert(1,1)
        catNum.place(relx=0.605,rely=0.22,anchor='nw')
        DSNum = tk.Spinbox(self,from_=0, to=2000,width=4)
        DSNum.delete(0,'end')
        DSNum.insert(1,0)
        DSNum.place(relx=0.605,rely=0.26,anchor='nw')
        #adds the repetition/looping section
        heading5 = tk.Text(self,height=1,width=31,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading5.insert(1.0,"Repetitions")
        heading5.place(relx=0.29,rely=0.34,anchor='nw')
        trialsLabel = tk.Text(self,height=1,width=14,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        trialsLabel.insert(1.0,"Trials")
        trialsLabel.place(relx=0.29,rely=0.38,anchor='nw')
        trialsNum = tk.Spinbox(self,from_=0, to=1000,width=10)
        trialsNum.delete(0,'end')
        trialsNum.insert(1,25)
        trialsNum.place(relx=0.445,rely=0.38,anchor='nw')

        tauHopLabel = tk.Text(self,height=1,width=14,background = labelBGColor,
font=labelFont,fg = labelTextColor)
```

```
        tauHopLabel.insert(1.0,"Hopping Taus")
        tauHopLabel.place(relx=0.29,rely=0.42,anchor='nw')
        hops = tk.StringVar()
        hopEntry = ttk.Combobox(self,textvariable = hops,width = 32)
        hopEntry.place(relx=0.445,rely=0.42,anchor='nw')
        hopEntry.insert(0,"Enter Comma-Separated Values (ns)")
        hopEntry['values']=hopList
        SvHopButton = ttk.Button(self, text="Save Values",command=lambda:
saveHops(hops.get(),hopList))
        SvHopButton.place(relx=.67,rely=0.42,anchor='nw')

        tauRecombLabel = tk.Text(self,height=1,width=14,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        tauRecombLabel.insert(1.0,"Recombine Taus")
        tauRecombLabel.place(relx=0.29,rely=0.46,anchor='nw')
        recombs = tk.StringVar()
        recombEntry = ttk.Combobox(self,textvariable = recombs,width = 32)
        recombEntry.place(relx=0.445,rely=0.46,anchor='nw')
        recombEntry['values']=recombList
        recombEntry.insert(0,"Enter Comma-Separated Values (ns)")
        SvRecombButton = ttk.Button(self, text="Save Values",command=lambda:
saveRecombs(recombs.get(),recombList))
        SvRecombButton.place(relx=.67,rely=0.46,anchor='nw')
        #adds the surface selection section
        heading6 = tk.Text(self,height=1,width=25,background = labelBGColor, font=FONT,fg =
labelTextColor)
        heading6.insert(1.0,"Surface")
        heading6.place(relx=0.7,rely=0.1,anchor='nw')
        surface = tk.StringVar()
        surfaceBox = ttk.Combobox(self,textvariable = surface)
        surfaceBox.place(relx=0.7,rely = 0.14,anchor = 'nw')
        surfaceBox['values']=surfaceNames

        catMaxLabel = tk.Text(self,height=1,width=12,background = labelBGColor,
font=labelFont,fg = labelTextColor)
        catMaxLabel.insert(1.0,"Catalyst Max ")
        catMaxLabel.place(relx=0.7,rely=0.18,anchor='nw')

        catMaxNum = tk.Spinbox(self,from_=1, to=10,width=10)
        catMaxNum.delete(0,'end')
        catMaxNum.insert(1,2)
        catMaxNum.place(relx=0.85,rely=0.18,anchor='nw')
```

```
def saveHops(hops,hopList): #saves the currently input list of hopping constants to a
permanent list
   hopList = hopList+ [hops]
   exportCSV("hops.csv",hopList)

def importHops(): #scans the list of stored hopping constants and adds them to the drop
down menu
   filepath = os.getcwd()
   hopList = []
   csv.register_dialect('myDialect',quoting=csv.QUOTE_ALL,skipinitialspace=True)
   with open("SavedSettings/hops.csv", 'r') as csvFile:
      hopreader = csv.reader(csvFile, delimiter=',',dialect ='myDialect')
      for row in hopreader:
         hopList.append(row)
   return hopList[0]

def saveRecombs(recombs,recombList): #saves the currently input list of recombination
constants to a permanent list
   recombList = recombList+ [recombs]
   exportCSV("recombs.csv",recombList)

def importRecombs():#scans the list of stored recombination constants and adds them to
the drop down menu
   filepath = os.getcwd()
   recombList = []
   csv.register_dialect('myDialect',quoting=csv.QUOTE_ALL,skipinitialspace=True)
   with open("SavedSettings/recombs.csv", 'r') as csvFile:
      recombreader = csv.reader(csvFile, delimiter=',',dialect ='myDialect')
      for row in recombreader:
         recombList.append(row)
   return recombList[0]

def exportCSV(filename,dataCSV): #exports a CSV file
   csv.register_dialect('myDialect',quoting=csv.QUOTE_ALL,skipinitialspace=True)
   with open("SavedSettings/"+filename, 'w') as csvFile:
      writer = csv.writer(csvFile,delimiter=',',dialect ='myDialect',lineterminator = '\n' )
      writer.writerow(dataCSV)
   csvFile.close()


def toggleCWStates(maxTimeNum,sunsNum): #checks whether Continuous Illumination
mode is enabled and grays out options if it not
   if maxTimeNum['state'] == 'normal':
      maxTimeNum['state'] = 'disabled'
      sunsNum['state'] = 'disabled'
   else:
```

```
        maxTimeNum['state'] = 'normal'
        sunsNum['state'] = 'normal'


def toggleBLStates(fractTransNum,absNum): #checks whether Beer's Law is enabled and
grays out options if it not
    if fractTransNum['state'] == 'normal':
        fractTransNum['state'] = 'disabled'
        absNum['state'] = 'disabled'
    else:
        fractTransNum['state'] = 'normal'
        absNum['state'] = 'normal'

def updateAbs(absNum,fractTransNum): #each time a fractional transmission value is
changed, updates to the corresponding absorbance value
    absorb = -math.log10(float(fractTransNum.get()))
    absNum.delete(0,'end')
    absNum.insert(1,absorb)

def updatefractTrans(absNum,fractTransNum): #each time an absorbance value is
changed, updates to the corresponding fractional transmission value
    ft = 10**(-(float(absNum.get())))
    fractTransNum.delete(0,'end')
    fractTransNum.insert(1,ft)


app = Model()#black magic
app.mainloop() #black magic
```

The Surface Generator

```
import random as rand
import math
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import pickle
import os

def dist(p1,p2): # Pythagorean theorem in 3D assuming real coordinates
   return math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2+(p1[2]-p2[2])**2)

def weighted_choice(weights): # given a list of n weights, returns an integer 1-n weighted
by weights
   rnd = rand.random() * sum(weights)
   for i, w in enumerate(weights):
      rnd -= w
      if rnd < 0:
         return i
def plotSurface(SURF): # displays a 3D plot of the molecular surface
   X = flatten([[i.XYZ[0] for i in j.molecules] for j in SURF.particles])
   Y = flatten([[i.XYZ[1] for i in j.molecules] for j in SURF.particles])
   Z = flatten([[i.XYZ[2] for i in j.molecules] for j in SURF.particles])
   fig = plt.figure()
   ax = fig.add_subplot(111, projection='3d')
   Axes3D.scatter(ax,xs=X,ys=Y,zs=Z)
   plt.show()

def flatten(list):# analogous to flatten function in Mathematica. Condenses
multidimensional array
   return [item for sublist in list for item in sublist]


class Particle(): # defines a Particle class to store all necessary information
   def __init__(self):
      self.center = [0,0,0]
      self.radius= 1.0 # nm
      self.numPositions = 100
      self.molecules = []
      self.particleNNs = [] #Lists all neighboring particles in contact with this particle
      self.particleArea = self.radius**2
      self.particleRegions = 0

class Molecule():# defines a Molecule class to store all necessary information
   def __init__(self):
```

```
        self.radii = 0.5 #nm
        self.XYZ = [0,0,0]
        self.inclinationAngle = 0 # in degrees
        self.anisotropyContribution = 0
        self.OxState = 0
        self.typeOf = 1 #0 is DeadSpot, 1 is Dye, 2 is Catalyst
        self.NNs = []    #Will be initialized to an array of Nearest Neighbors
        self.NNdists = []
        self.HopRates = []  # Will be initialized to an array of hop rates to (Dye,Cat-
0,Cat1,Cat2...)
        self.RecombRate = 0 #Will be initialized to rate constant based on being in the
fast/slow population
        self.percolationZone = 0

class Surface(): # defines a Surface class to store all necessary information

    def  __init__(self):
        self.particles = []
        self.heightMax = 0
        self.heightMin = 0
        self.thickness = 0
        self.name = "name"
        self.totalSites = 0
        self.numParticles = 0

def
buildSurface(surfaceName,r1Pos,molRad,neckMin,neckMax,partRadMin,partRadMax,numP
art,stack,compactness,reach):
    #Determining the Particle Positions and sizes
    SURF = Surface()
    SURF.name = surfaceName
    SURF.particles.append(Particle()) #starts the Surface out with a single particle
    SURF.particles[0].radius =rand.random()*(partRadMax-partRadMin)+partRadMin #
gives that particle a random size

    for i in range(1,numPart): # adds particles to the surface until the specified number are
added
        validChoice = False #assumes the proposed particle is going to be invalid before trying
to find one that is valid
        while validChoice==False: #continuously proposes the next particle until a valid one is
selected
            size = rand.random()*(partRadMax-partRadMin)+partRadMin # chooses a random
size for proposal
            neck = size *(rand.random()*(neckMax-neckMin)+neckMin)#chooses a necking
fraction for proposal
```

```
        weights = [(j+1)**compactness for j in range(len(SURF.particles),0,-1)] # weights the
choice for particle to grow off of
        choice = weighted_choice(weights)#chooses a particle to grow off of
        currentCenter = SURF.particles[choice].center
        theta = rand.random()*math.pi*2 #chooses a pair of random angles to determine
how to attach proposed particle to particle being grown off of
        phi = math.acos(2*rand.random()-1)
        x = math.cos(theta)*math.sin(phi)
        y = math.sin(theta)*math.sin(phi)
        z = math.cos(phi)
        unit = [x,y,z]
        if stack==True: unit=[0,0,1] # if stack is true, ignores chosen angles and instead
always attaches along the z-axis
        offset = max((1-
neck)*size+SURF.particles[choice].radius,size+SURF.particles[choice].radius*(1-neck)) #
based on necking determines where proposed particle will be relative to the particle being
grown off of
        newCenter = np.add(currentCenter, [l*offset for l in unit])# determines where the
proposed particle will be
        validChoice=True # assumes that new particle will be valid before trying to prove it
is not
        for m in range(len(SURF.particles)): # iterates over every other particle in the
surface
            distance = math.sqrt((newCenter[0]-
SURF.particles[m].center[0])**2+(newCenter[1]-
SURF.particles[m].center[1])**2+(newCenter[2]-SURF.particles[m].center[2])**2) # finds
the distance to that particle
            minC2Cdist = max(size*(1-
neckMax)+SURF.particles[m].radius,size+SURF.particles[m].radius*(1-neckMax))# based
on both particle sizes and necking maximums, determines how close the proposed particle
can be to that particle
            if distance<minC2Cdist: #if the particle is too close, reject it
              validChoice=False

      SURF.particles.append(Particle()) # adds the now validated proposed particle (along
with its size and center) to the surface
      SURF.particles[i].center = newCenter.tolist()
      SURF.particles[i].radius = size
   print("Done Finding Particle Positions and Sizes!")

   #DETERMINING ALL POSITIONS
   goldenAngle = math.pi*(3-math.sqrt(5)) # stores the golden angle for reference in
making golden spirals
   for p in range(len(SURF.particles)): # iterates over all particles to set their molecules
      rad=SURF.particles[p].radius
      c = SURF.particles[p].center
```

```
      n = round(r1Pos*rad**2) #calculates the number of positions to be on each particle
based on its size and the number specified for a radius=1 particle
      rho = [goldenAngle*i for i in range(n)] # calculates XYZ coordinates based on
golden/Fibonacci spirals to evenly distribute them over a particle's surface
      z = [((1-1.0/n)-(2.0/n)*i) for i in range(n)]
      r = [ math.sqrt(1-z[i]**2) for i in range(n)]
      x = [(r[i]*math.cos(rho[i])) for i in range(n)]
      y = [(r[i]*math.sin(rho[i])) for i in range(n)]

      theta = rand.random()*math.pi*2 # chooses 2 random angles to rotate the particle by
so the spiral is oriented randomly
      phi = rand.random()*math.pi

      pos = np.array([x,y,z])# makes the XYZ coorinates into a matrix capable of rotating
      pos = pos.transpose()
      rot = [[math.cos(phi),math.sin(phi),0],[-
math.cos(theta)*math.sin(phi),math.cos(theta)*math.cos(phi),math.sin(theta)],[math.sin(t
heta)*math.sin(phi),-math.sin(theta)*math.cos(phi),math.cos(theta)]] # creates a rotation
matrix using phi and theta to rotate a particle both azimuthally and longitudinally
      rotpos = (np.matmul(pos,rot))*rad+c #multiplies the coordinates of the position
matrix by the rotation matrix

      SURF.particles[p].molecules = [Molecule() for m in range(n)] # creates an array of
molecules objects on the particle object
      for m in range(n): #iterates over all molecules and specifies their coordinates,
inclination angle and anisotropy contribution
         SURF.particles[p].molecules[m].XYZ = rotpos[m]
         SURF.particles[p].molecules[m].inclinationAngle = math.acos((np.dot((rotpos[m]-
c),[0,1,0]))/np.linalg.norm(rotpos[m]-c))
         SURF.particles[p].molecules[m].anisotropyContribution =
1.5*(math.cos(SURF.particles[p].molecules[m].inclinationAngle)**2 - 0.5)
   print("Done Assigning Rotated Positions!")


   neighbors = [[] for i in range(len(SURF.particles))] # creates an array that will
temporarily store particle neighbors
   validpositions = [[] for i in range(len(SURF.particles))] # creates an array that will store
all validated positions of molecules
   # this next section does two things. 1. It determines which particles neighboring each
other. 2. It checks to see which molecular positions are invalidated from either being inside
another particle or else overlapping with another molecule in necking regions
   for k in range(len(SURF.particles)): # iterates over all particles, particle k
      for l in range(len(SURF.particles)):# iterates over all particles again
         if (k!=l) and
dist(SURF.particles[k].center,SURF.particles[l].center)<=SURF.particles[k].radius+SURF.pa
```

```
rticles[l].radius: # if two distinct particles are within their combined radii of each other, they are neighbors
                neighbors[k].append(l)
      for j in range(len(SURF.particles[k].molecules)): # iterates over all molecules on Particle k
         valid = True
         for m in range(len(neighbors[k])): # iterates over all particles neighboring Particle K, particle m
            if dist(SURF.particles[neighbors[k][m]].center,SURF.particles[k].molecules[j].XYZ)<=SURF.particles[neighbors[k][m]].radius+molRad:
               valid = False # if the molecule on particle m is within particle k, it is not a valid position
            else:
               for n in range(len(SURF.particles[neighbors[k][m]].molecules)):#iterates over all molecules on particle m
                  if dist(SURF.particles[neighbors[k][m]].molecules[n].XYZ,SURF.particles[k].molecules[j].XYZ)<2*molRad:
                     valid = False# if one of the molecules on particle m is too close to one of the molecules on particle m, is not a valid position
         if valid==True:
            validpositions[k].append(j) # if a molecule passed these test and was validated, add it to the list of valid positions
      SURF.particles[k].molecules = [SURF.particles[k].molecules[i] for i in range(len(SURF.particles[k].molecules)) if i in validpositions[k]] # store the valid positions on the Surface
      neighbors[k].append(k) # store the particle k as one of its own neighbors to make the next step easier
      SURF.particles[k].particleNNs = neighbors[k] # store the particle neighbors on the Surface
   print("Done Validating molecule positions!")


   #Finding Molecule Distances
   xDistances = [[[] for j in range(len(SURF.particles[i].molecules))] for i in range(len(SURF.particles))]
   #this next section creates a cross-distance (xDistance) table to store the distance between every molecule with every molecule it could potentially
   # be a neighbor of. Potential neighbors include all molecules on its own particle as well as all molecules on every particle that is a neighboring
   #particle. The table is stored as [particle #k,position #m,distance to particle #i, position #j] for all i and j and it is sorted by the last entry
   for i in range(len(SURF.particles)): #i is a particle index
      for j in range(len(SURF.particles[i].molecules)): # j is a molecule index on particle #i
         xDist = []
```

```
        for k in SURF.particles[i].particleNNs: # k is a particle index
            for m in range(len(SURF.particles[k].molecules)): # m is a molecule index on
particle #k

xDist.append([k,m,dist(SURF.particles[i].molecules[j].XYZ,SURF.particles[k].molecules[m].
XYZ)])
        xDist.sort(key = lambda x: x[2]) # sorts the x-distance table by distance
        xDistances[i][j] =xDist #stores the x-distance table in the x-distances table


    #Finding molecule neighbors
    for i in range(len(SURF.particles)): #iterates over all the particles
        for j in range(len(SURF.particles[i].molecules)): #iterates over all molecules
            maxNN = 1 #assumes that every molecule will have at least one nearest neighbor
            NN = [] #creates an empty list to store molecular neighbors
            dists = [] # creates an empty list to store the inter-neighbor distances
            while xDistances[i][j][maxNN][2]<=reach*molRad: #until a potential neighbor too
far, continuously add more neighbors to a molecules NN list
                NN.append([xDistances[i][j][maxNN][0],xDistances[i][j][maxNN][1]]) # store the
particle#, position # of the neighbor
                dists.append(xDistances[i][j][maxNN][2]) # store the distance to the new neighbor
                maxNN+=1 #move on to the next neighbor
            SURF.particles[i].molecules[j].NNs = NN #store the list of nearest neighbors on the
Molecule object
            SURF.particles[i].molecules[j].NNdists = dists #store the list of neighbor distances on
the Molecule object
    print("Done Setting Neighbors!")

    Zs = flatten([[i.XYZ[2] for i in j.molecules] for j in SURF.particles]) # separates out all the
z coordinates in the entire surface
    SURF.heightMax = max(Zs) #stores the top of the surface
    SURF.heightMin = min(Zs) #stores the bottom of the surface
    SURF.thickness = SURF.heightMax-SURF.heightMin # stores the thickness of the surface
    SURF.totalSites = len(Zs) #stores the number of molecules on the surface
    SURF.numParticles = len(SURF.particles)#stores the number of particles on the surface



    filepath = os.getcwd() #determines what directory the file is being run from
    with open(filepath+"/Surfaces/"+surfaceName+".SURF", "wb") as fileOut: # opens a file
to store the surface in
        pickle.dump(SURF, fileOut)# converts the surface to a datastream through Pickle and
exports it into the opened file
    print("Done Exporting!")
```

The Full Model

```
import random
import numpy as np
import math
import os
import pickle
import datetime
import statistics
import csv

def flatten(list): # analogous to flatten function in Mathematica. Condenses
multidimensional array
   return [item for sublist in list for item in sublist]

def weighted_choice(weights): # given a list of n weights, returns an integer 1-n weighted
by weights
   rnd = random.random() * sum(weights)
   for i, w in enumerate(weights):
      rnd -= w
      if rnd < 0:
         return i

def exportCSV(filename,dataCSV): # takes a dataset and exports a CSV with a given
filename
   csv.register_dialect('myDialect',quoting=csv.QUOTE_NONE,skipinitialspace=True)
   with open("Data/"+filename, 'w') as csvFile:
      writer = csv.writer(csvFile,dialect ='myDialect',lineterminator = '\n' )
      writer.writerows(dataCSV)
   csvFile.close()

def FloodFill(startPoint,zoneCount,surface): #uses a FloodFill algorithm to identify all
mutually connected molecules in a percolation network
   if (surface.particles[startPoint[0]].molecules[startPoint[1]].typeOf != 0) and
(surface.particles[startPoint[0]].molecules[startPoint[1]].percolationZone == -1):
      surface.particles[startPoint[0]].molecules[startPoint[1]].percolationZone = zoneCount
      if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=1:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[0],zoneCount,surfa
ce)
         if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=2:
```

```
FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[1],zoneCount,surface)
        if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=3:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[2],zoneCount,surface)
            if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=4:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[3],zoneCount,surface)
                if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=5:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[4],zoneCount,surface)
                    if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=6:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[5],zoneCount,surface)
                        if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=7:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[6],zoneCount,surface)
                            if len(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs)>=8:

FloodFill(surface.particles[startPoint[0]].molecules[startPoint[1]].NNs[7],zoneCount,surface)



#RunModel takes many parameters from the GUI
def
RunModel(surface,name,AnisTF,DDHTF,electSpreadTF,electScaleTF,timeRecTF,AnisRecTF,TurnoverRecTF,hoppingPathTF,CWModeTF,maxTimeSteps,suns,BLTF,fracTrans,excitePCTTF,catPCTTF,DSPCTTF,excitePartTF,catPartTF,DSPartTF,exciteNum,catNum,DSNum,trials,hops,recombs,maxOxState):
    now = datetime.datetime.now()
    date = now.strftime("%y%m%d")#creates a datestring to use in filenames

    if excitePCTTF == True: #depending on distribution choices, appropriately interprets the excitation number input as a percent or a raw number
        numInitiallyExcitedDyes = exciteNum/100*surface.totalSites
    elif excitePartTF == True:
        numInitiallyExcitedDyes = exciteNum*len(surface.particles)
    else:
        numInitiallyExcitedDyes = exciteNum
```

```
   if catPCTTF == True: #depending on distribution choices, appropriately interprets the
excitation number input as a percent or a raw number
      numCatalysts = catNum/100*surface.totalSites
   elif catPartTF == True:
      numCatalysts = catNum*len(surface.particles)
   else:
      numCatalysts = catNum

   #Filenames
   turnoverFilename =
date+"_Turn_"+str(maxOxState)+"X"+str(int(numInitiallyExcitedDyes))+"_Excitations_"+na
me
   timeFilename =
date+"_Time_"+str(maxOxState)+"X"+str(int(numInitiallyExcitedDyes))+"_Excitations_"+na
me
   hopFilename =
date+"_Hop_"+str(maxOxState)+"X"+str(int(numInitiallyExcitedDyes))+"_Excitations"+na
me
   anisotropyFilename =
date+"_Anis_"+str(maxOxState)+"X"+str(int(numInitiallyExcitedDyes))+"_Excitations_"+na
me
   #sets filenames according to the date and whether Beer's Law is being used
   if BLTF==True:
      turnoverFilename = turnoverFilename+"_BL.CSV"
      timeFilename=timeFilename+"_BL.CSV"
      hopFilename=hopFilename+"_BL.CSV"
      anisotropyFilename=anisotropyFilename+"_BL.CSV"
   else:
      turnoverFilename = turnoverFilename+".CSV"
      timeFilename=timeFilename+".CSV"
      hopFilename=hopFilename+".CSV"
      anisotropyFilename=anisotropyFilename+".CSV"

   TauHop = [int(e) for e in hops.split(",")] # interprets the hopping constant string and
stores an array of hopping constants
   TauRecomb =[int(e) for e in recombs.split(",")] #interprets the recombinations constant
string and stores an array of recombination constants
   numParamPoints = len(TauHop)*len(TauRecomb)#calculates the number of parameter
points in the experiments based on the number of different time constants

   allTimes = [[] for i in range(numParamPoints)]# sets up a table to store time data
throughout the whole simulation
   allAnis = [[] for i in range(numParamPoints)]# sets up a table to store anisotropy data
throughout the whole simulation
```

```
   allHops = [[] for i in range(numParamPoints)]# sets up a table to store hopping path data throughout the whole simulation

   turnoverTable= [[] for i in range(numParamPoints+1)] #sets up a table to store turnover data throughout the whole simulation
   turnoverTable[0] = ["TauRecomb (ns)", "TauHop (ns)", "TauRatio", "Percent Turnovers", "Number Turnovers"] #creates a header for the turnover table

   parameterPoint = -1#starting off at parameter point -1 so the loop can increment to 0 in the beginning

   for R in TauRecomb: # starts a loop to repeat the experiment with every recombination constant
      R = R*(10**(-9)) #changes units of the recombination constant
      for H in TauHop: # starts a loop to repeat the experiment with every hopping constant
         H = H*(10**(-9)) #changes units of the hopping constant
         parameterPoint+=1 #steps forward to the next parameter point
         tauRatio = round(R/H,2) #calculates the ratio between the time constants
         timeDecays = [] # creates an array to store time data for this parameter point
         timeHeader = ["Timestep", "Time", "Dyes Remaining", "Charges", "Turnovers", "Dye Recombinations", "Catalyst Recombinations", "Catalysts Remaining", "Excitations"]
         timeTable = [[]for i in range(trials)]# creates an array to store time data for this parameter point
         anisTable = [[]for i in range(trials)]# creates an array to store anisotropy data for this parameter point
         anisHeader = ["Timestep", "Time", "Excited Molecules", "Anisotropy"] #creates a header to the anisotropy table
         turnoverTotals = 0 #initializes the turnover counter
         noCatRecombinations = 0 #initializes the no catalyst recombination counter
         loneChargeRecombinations = 0 #initializes the lone charge recombination counter
         autoRecombinations = 0 #initializes the autorecombination counter
         shortestTrial = 10**10 #uses a very large value to initialize the shortest trial length

         for trial in range(trials): #starts a loop to repeat a certain number of trials for statistical averaging
            endCondition = False
            print("---------------------------------NEW TRIAL---------------------------------")
            print("Hop Rate: "+str(round(H*10**9))+ " ns"+ "\nRecomb Rate: "+str(round(R*10**9))+ " ns"+ "\nParameter Point: "+ str(parameterPoint)+ "\nTrial: "+str(trial))
            turnovers = 0 #initializes the turnover counter
            numDyes = numInitiallyExcitedDyes
            effectiveAnisTau = H*3.75 #scales hopping time constant based on an experiment done long ago. Kind of arbitrary
            minTau = min(effectiveAnisTau,R)#determines the smallest time constant
```

```
        timestepSize = round(minTau/350,12)# sets the time step size much smaller than
the smallest time constant
        tauExcite = (14.8*(10**-6))/suns #calculates the time constant for generation
based on the number of suns of illumination

        tauHopCattoDye = H*1000000000 # sets the catalyst to dye hopping rate

        #creates a table of dye to catalysts hopping rates based on catalyst oxidation state
        tauHopDyetoCat = [H for i in range(4)]
        tauHopDyetoCat[0] = H/27
        tauHopDyetoCat[1] = H/27
        tauHopDyetoCat[2] = H/27
        tauHopDyetoCat[3] = H/27

        #creates a table of catalyst to catalyst hopping rates based on target catalysts
oxidation states
        tauHopCattoCat = [H for i in range(4)]
        tauHopCattoCat[0] = H
        tauHopCattoCat[1] = H
        tauHopCattoCat[2] = H
        tauHopCattoCat[3] = H

        #creates a table of fast and slow recombination rates for catalysts of oxidation
state 1-4
        tauRecombCat = [[R,R] for i in range(4)]

        popFracts = [0.5 for i in range(4)] # creates a table of population fractions so that
each catalyst oxidation state can be assigned to 1 of 2 recombination rates

        #converts time constants to probabilities by dividing them by the timestep size
        probExcite = timestepSize/tauExcite
        probHopDyetoDye = timestepSize/H
        probHopCattoDye = timestepSize/tauHopCattoDye
        probHopDyetoCat = [timestepSize/i for i in tauHopDyetoCat]
        probHopCattoCat = [timestepSize/i for i in tauHopCattoCat]
        probRecombCat = [[timestepSize/i[0],timestepSize/i[1]] for i in tauRecombCat]
        probRecombDye = timestepSize/R

        #initialize all the relevant counters
        dyeRecombinations = 0
        excitations = 0
        catalystRecombinations = [0 for i in range(maxOxState)]
        catalystSpecies = [0 for i in range(maxOxState)]
        anisotropy = 0
```

```
        #Assigning Dye and Catalyst Positions (and dead spots)

        for i in surface.particles: # resetting oxidation state of every molecule on the
surface and its type
          for j in i.molecules:
            j.OxState = 0
            j.typeOf = 1
            j.percolationZone =-1

        openPositions = surface.totalSites #keeps track of how many spots are remaining
to choose from
        choiceWeighting = [[1 for  i in range(len(surface.particles[j].molecules))]for j in
range(surface.numParticles)]#creates a table containing weights for all molecular positions
        possiblePositions = [[[j,i] for  i in range(len(surface.particles[j].molecules))]for j in
range(surface.numParticles)]#creates a table containing coordinates for all molecular
positions
        flattenedPositions = flatten(possiblePositions)# creates a flattened version of the
molecular positions table

        #assigning dead spots
        DSArray = [] #creates a table to store all dead spot positions
        if DSPartTF ==True:# if dead spots are to be chosen on a particle by particle basis
          for particle in range(surface.numParticles): # iterates over all particles
            if DSPCTTF == True: #if dead spots are chosen as a percent rather than a fixed
value
              n = round((DSNum/100)*len(surface.particles[particle].molecules))#
calculates the number of dead spots to be chosen
            else:
              n = DSNum #uses the provided dead spot number as the number to be
chosen
            DSArray.append(random.sample(possiblePositions[particle],n))#makes
choices for the current particle and adds them to the dead spot array
          DSArray = flatten(DSArray) #flattens out all the separate particle choices so that
the final array is 1D not 2D
        else: #if dead spots are to be chosen over the whole film
          if DSPCTTF == True: #if dead spots are chosen as a percent rather than a fixed
value
            n = round((DSNum/100)*openPositions) # calculates the number of dead
spots to be chosen
          else:
            n = DSNum #uses the provided dead spot number as the number to be chosen
          DSArray = random.sample(flattenedPositions,n) #chooses all the dead spots for
the entire film

        for d in DSArray: #sets the properties of the chosen dead spots
```

```
            choiceWeighting[d[0]][d[1]] = 0# sets the value to 0 in the weighting table so
that these positions cannot be chosen again
            surface.particles[d[0]].molecules[d[1]].typeOf = 3 #sets the type on the
molecules so they will be skipped over from here on out
          openPositions -=len(DSArray)#subtracts the number of dead spots from the
number of open positions to account for the fact that they are now unavailable

          #assigning Catalyst positions
          catArray = [] #creates a table to store all catalyst positions
          if catPartTF ==True: #if catalysts are to be chosen on a particle by particle basis
            for particle in range(surface.numParticles): # iterates over all particles
              if catPCTTF == True: #if catalysts are chosen as a percent rather than a fixed
value
                n = round((catNum/100)*len(surface.particles[particle].molecules)) #
calculates the number of catalysts to be chosen
              else:
                n = catNum #uses the provided catalysts number as the number to be chosen
              totalWeight = sum(choiceWeighting[particle]) #calculates the total weight of
all potential catalyst choices
              weights = [i/totalWeight for i in choiceWeighting[particle]] #calculates
weights for all possible choices
              cArray =
np.random.choice(range(len(possiblePositions[particle])),size=n,replace=False,
p=weights)#chooses the catalysts for the current particle
              catArray.append([possiblePositions[particle][c] for c in cArray]) #uses the
choices made to select positions for the catalyst array
            catArray = flatten(catArray) #flattens out all the separate particle choices so that
the final array is 1D not 2D
          else:#if catalysts are to be chosen over the whole film
            if catPCTTF == True: #if catalysts are chosen as a percent rather than a fixed
value
              n = round((catNum/100)*openPositions) # calculates the number of catalysts
to be chosen
            else:
              n = catNum #uses the provided catalysts number as the number to be chosen
            totalWeight = sum(flatten(choiceWeighting)) #calculates the total weight of all
potential catalyst choices
            weights =  [i/totalWeight for i in flatten(choiceWeighting)] #calculates weights
for all possible choices
            catArray =
np.random.choice(range(len(flattenedPositions)),size=n,replace=False, p=weights)
#chooses all the catalysts for the entire film
            catArray = [flattenedPositions[c] for c in catArray] #uses the choices made to
select positions for the catalyst array
```

```
        for c in catArray: #applies all catalyst properties to molecules chosen to be in the
catalyst array
            choiceWeighting[c[0]][c[1]] = 0 #sets the choice weighting to be 0 so this
position cannot be chosen again
            surface.particles[c[0]].molecules[c[1]].typeOf = 2 #sets the type to be catalyst
            recomb1 = np.random.choice(probRecombCat[0],size=1,p = [popFracts[0], 1 -
popFracts[0]])
            recomb2 = np.random.choice(probRecombCat[1],size=1,p = [popFracts[1], 1 -
popFracts[1]])
            recomb3 = np.random.choice(probRecombCat[2],size=1,p = [popFracts[2], 1 -
popFracts[2]])
            recomb4 = np.random.choice(probRecombCat[3],size=1,p = [popFracts[3], 1 -
popFracts[3]])
            surface.particles[c[0]].molecules[c[1]].RecombRate =
[recomb1,recomb2,recomb3,recomb4] #stores recombination rates for four oxidation
states
        openPositions -=len(catArray)    #removes the number of open positions now
occupied by catalysts

        #calculating BL weights if needed
        if BLTF == True: #if beers law is used modifies the weighting used for dye
excitation according to molecular depth and film thickness
            moleculeDepthFraction = [[(m.XYZ[2]-surface.heightMin)/surface.thickness for
m in p.molecules] for p in surface.particles] #calculates relative depth of all molecules in
the film
            T = np.log10(1/fracTrans) #transmission coefficient
            BLweighting = [[10**(-T*m) for m in p]for p in
moleculeDepthFraction]#calculates beers law weighting
            choiceWeighting = np.multiply(choiceWeighting,BLweighting) #applies Beer's
law weighting to weight matrix

        #calculating anisotropy weights if needed
        if AnisTF == True: #if polarized light is to be used, modifies the weighting used for
dye excitation based on dye inclination angle
            AnisWeighting = [[(math.cos(m.inclinationAngle))**2 for m in p.molecules]for p
in surface.particles] #calculates anisotropy weighting
            choiceWeighting = np.multiply(choiceWeighting,AnisWeighting)#applies
anisotropy weighting to weight matrix

        #assigning excitation positions
        exciteArray = [] #creates a table to store all initial excitation positions
        if excitePartTF ==True: #if excitations are to be chosen on a particle by particle
basis
            for particle in range(surface.numParticles): #iterates over all particles
                if excitePCTTF == True:  #if excitations are chosen as a percent rather than a
fixed value
```

```
                n = round((exciteNum/100)*len(surface.particles[particle].molecules)) #
calculates the number of excitation to be chosen
            else:
                n = exciteNum #uses the provided excitation number as the number to be
chosen
            totalWeight = sum(choiceWeighting[particle]) #calculates the total weight of
all potential excitation choices
            weights = [i/totalWeight for i in choiceWeighting[particle]]#calculates weights
for all possible choices
            eArray =
np.random.choice(range(len(possiblePositions[particle])),size=n,replace=False,
p=weights) #chooses the excitations for the current particle
            exciteArray.append([possiblePositions[particle][e] for e in eArray]) #uses the
choices made to select positions for the excitation array
        exciteArray = flatten(exciteArray) #flattens out all the separate particle choices
so that the final array is 1D not 2D
    else: #if excitations are to be chosen over the whole film
        if excitePCTTF == True: #if excitations are chosen as a percent rather than a
fixed value
            n = round((exciteNum/100)*openPositions) # calculates the number of
excitations to be chosen
        else:
            n = exciteNum #uses the provided excitation number as the number to be
chosen
        totalWeight = sum(flatten(choiceWeighting)) #calculates the total weight of all
potential excitation choices
        weights =  [i/totalWeight for i in flatten(choiceWeighting)] #calculates weights
for all possible choices
        exciteArray =
np.random.choice(range(len(flattenedPositions)),size=n,replace=False, p=weights)
#chooses all the excitations for the entire film
        exciteArray = [flattenedPositions[e] for e in exciteArray] #uses the choices made
to select positions for the excitation array

    for e in exciteArray: #sets the oxidation state of all excited dyes to 1
        surface.particles[e[0]].molecules[e[1]].OxState = 1
    openPositions -=len(exciteArray) #removes the chosen positions from the open
position counter


    #this next section uses the flood fill algorithm to identify percolation zones
    zoneCount = -1
    for x in range(surface.numParticles): #iterates over all particles
        for y in range(len(surface.particles[x].molecules)):#iterates over all molecules
on particle x
```

```
                if (surface.particles[x].molecules[y].typeOf != 0) and
(surface.particles[x].molecules[y].percolationZone == -1):
                    zoneCount +=1 #A new Zone has been found!
                    FloodFill([x,y],zoneCount,surface) #flood fill from the new zone and mark
with the current zone count
            zones = [[] for i in range(zoneCount+1)] #creates a table to store all zones
            zoneSizes = [[i,0] for i in range(zoneCount+1)]#creates a table to store the sizes of
all zones

            for z in range(zoneCount+1): #iterates over all zones
                for x in range(surface.numParticles):#iterates over all particles
                    for y in range(len(surface.particles[x].molecules)): #iterates over all molecules
on particle x
                        if surface.particles[x].molecules[y].percolationZone == z: # if molecule y
belongs to zone z, add it to the zone
                            zones[z].append([x,y])#adds molecule y to zone z
                zoneSizes[z][1] = len(zones[z])#calculates the size of all zones once sorting is
complete


            POI = random.sample(exciteArray,len(exciteArray)) #randomly sorts the
excitation array and creates the list which will be used to provide a turn for every
excitation at every timestep

            if DDHTF == True: #if distance dependent hopping is enabled
                NNdistances = flatten([j.NNdists for j in i.molecules for i in surface.particles])
#creates a flattened last of all nearest neighbor hopping distances
                NNdistancesAngstroms = [x*15*10 for x in NNdistances] #converts that list to
angstroms
                moleculeRadiusAngstroms = surface.particles[0].molecules[0].radii*15*10
#converts the molecule radius to angstroms
                tunnelFactor = 0.35 # sort of arbitrary number but is one people use

                expAve = -math.log10(statistics.mean([math.exp(b) for b in[(-tunnelFactor)*(x-
2*moleculeRadiusAngstroms) for x in
NNdistancesAngstroms]]))/tunnelFactor+2*moleculeRadiusAngstroms #calculates the
exponentially weighted average of the distance between neighbors
                aveFactor = math.exp((expAve - 2*moleculeRadiusAngstroms)*(-tunnelFactor))
#calculates the average distance factor by inverting the formula above
                aveFactorInverse = 1.0/aveFactor #inverts the averages distance factor

            for x in surface.particles: #iterates over all particles
                for y in x.molecules: #iterates over all molecules on particle x
                    if y.typeOf ==1: #Currently looking at a Dye
                        y.RecombRate = probRecombDye #sets the dye recombination probability
```

```
                hopProbs = [0 for i in range(len(y.NNs))] # creates a table to store hopping probabilities between neighboring molecules
                for z in range(len(y.NNs)): # iterates over all the neighbors of molecule y
                  if DDHTF ==True: distFactor = aveFactorInverse*math.exp((-tunnelFactor)*(y.NNdists[z]*15*10-2*moleculeRadiusAngstroms)) #if using distant dependent hopping, calculates the factor between molecule y and neighbor z
                  else:distFactor = 1 #if not using distant dependent hopping, the default factor is 1
                  if surface.particles[y.NNs[z][0]].molecules[y.NNs[z][1]].typeOf==1: hopProbs[z] = probHopDyetoDye*distFactor # if neighbor z is a dye applies the distance factor to the hopping probability between molecule y and neighbor z
                  else: hopProbs[z] = probHopDyetoCat[0]*distFactor # if neighbor z is a catalyst applies the distance factor to the hopping probability between molecule y and neighbor z
            if y.typeOf ==2: #Currently looking at a Catalyst
                hopProbs = [0 for i in range(len(y.NNs))] # creates a table to store hopping probabilities between neighboring molecules
                for z in range(len(y.NNs)): #iterates over all neighbors of molecule y
                  if DDHTF ==True: distFactor = aveFactorInverse*math.exp((-tunnelFactor)*(y.NNdists[z]*15*10-2*moleculeRadiusAngstroms)) #if using distant dependent hopping, calculates the factor between molecule y and neighbor z
                  else:distFactor = 1 #if not using distant dependent hopping, the default factor is 1
                  if surface.particles[y.NNs[z][0]].molecules[y.NNs[z][1]].typeOf==1: hopProbs[z] = probHopCattoDye*distFactor # if neighbor z is a dye applies the distance factor to the hopping probability between molecule y and neighbor z
                  else: hopProbs[z] = probHopCattoCat[0]*distFactor # if neighbor z is a catalyst applies the distance factor to the hopping probability between molecule y and neighbor z
            y.HopRates = hopProbs #stores the hopping rates for molecule y

        initialChargeDistPZ = [0 for i in range(zoneCount+1)] #creates a table to store the number of charges per percolation zone
        catsPerPZ = [0 for i in range(zoneCount+1)]#creates a table to store the number of catalysts per percolation zone
        hopelessZones = []#creates a list to store identified hopeless zones
        for x in POI:
initialChargeDistPZ[surface.particles[x[0]].molecules[x[1]].percolationZone]+=1 # counts the number of excitations per percolation zone
        for c in catArray:
catsPerPZ[surface.particles[c[0]].molecules[c[1]].percolationZone]+=1#counts the number of catalysts per percolation zone
        for z in range(zoneCount): #iterates over all percolation zones
          if (catsPerPZ[z] ==0) or (initialChargeDistPZ[z] < maxOxState):# if there are either no catalysts in the zone, or aren't enough excitations to turn over a single catalyst, the zone is hopeless
```

```
            if (catsPerPZ[z] ==0): noCatRecombinations += initialChargeDistPZ[z] #if there
are no catalysts in zone z, record that
            if (initialChargeDistPZ[z] < maxOxState):
loneChargeRecombinations+=initialChargeDistPZ[z]#if there are too few excitations in
zone z, record that
            hopelessZones.append(z) #add zone z to the hopeless zones list

        print("Beginning Main Loop")
        timestep=0 #start at the beginning, a very good place to start
        while endCondition==False: #continues until one end condition is met which
depends on run conditions
          if timestep%1000==0: #checks to evaluate end condition every 1000 timesteps
            if numDyes<=0 and CWModeTF==False: endCondition=True #if there are no
more excited dyes and Continuous Illumination isn't enabled, end
            if len(catArray) > 0: # if there were originally catalysts in the film, check to
make sure at least 1 non-hopeless zone is occupied
              allHopeless = True #assume dyes only exist in hopeless zones and then
check for the opposite
              for p in POI: #iterates over all currently excited molecules
                if surface.particles[p[0]].molecules[p[1]].typeOf == 1: # if the molecule is a
dye
                  if surface.particles[p[0]].molecules[p[1]].percolationZone not in
hopelessZones: allHopeless=False # if the molecule is in a zone not on the list of hopeless
zones
              if allHopeless == True: # if no excited dyes were found in zone not on the
hopeless zones list
                numDyes = 0 #reduce the number of dyes remaining to 0 (the remaining
dyes will all eventually recombine)
                print("FORESAKEN!") #declare that all hope is lost
                x=0# start a counter for the  following while loop
                while x<len(POI): #iterate over the list of excited molecules in a while loop
instead of a for loop to account for modifications of the POI
                  if surface.particles[POI[x][0]].molecules[POI[x][1]].typeOf==1: #was a
dye, carry out recombination for that dye
                    dyeRecombinations+=1 #tally the recombination event
                    autoRecombinations+=1 #tally the autorecombination event
                    POI.pop(x)#remove the dye from the POI
                  else:#if the excited molecule was a catalyst, just advance the counter to
the next molecule
                    x+=1

                endCondition=True #sets the end condition to true so that the loop will
end after the next iteration
            print("Timestep: "+str(timestep)+"\nParameter Point:
"+str(parameterPoint),"\nNumber of Dyes Left: ",numDyes)
          timestep+=1 #step forward in time
```

```
            if AnisRecTF == True: # if anisotropy recording is enabled, record for the
timestep
                anisotropy = 0 #resets the anisotropy counter
                for x in POI: anisotropy+=
surface.particles[x[0]].molecules[x[1]].anisotropyContribution   #iterates over all excited
molecules and adds their anisotropy contributions
                timestepStress = len(POI)#records the number of items in the POI
                if timestepStress> 0: anisotropy / timestepStress #scales the anisotropy to
calculate the average
                else: anisotropy = 0 #sets to 0 if there are no excited molecules left
                anisRow = [timestep,timestep*timestepSize,timestepStress,anisotropy] #
stores the anisotropy information for the timestep
                anisTable[trial].append(anisRow) #adds the current timestep row to the
anisotropy table

            electronsPerParticle = [0 for i in range(surface.numParticles)] #resets the
counter for number of electrons per particle
            electronDensityPerPart = [0 for i in range(surface.numParticles)] #resets the
counter for amount of electron density per particle
            charges = 0 # resets counters for this timestep before recalculating values
            numDyes = 0
            numCatalysts = 0
            for x in POI: # iterates over all excited molecules
              charges += surface.particles[x[0]].molecules[x[1]].OxState #counts the number
of charges total
              if surface.particles[x[0]].molecules[x[1]].typeOf == 1: #counts the number of
dyes per particle
                numDyes+=1
              else: #counts the number of catalysts per particle
                numCatalysts +=1

electronsPerParticle[x[0]]+=surface.particles[x[0]].molecules[x[1]].OxState#counts the
number of charges per particle
            if electSpreadTF == True: #if electron spreading is enabled, distribute electron
density between neighboring particles
              for x in range(surface.numParticles):#iterates over all particles

electronDensityPerPart+=electronsPerParticle[x]*(2*surface.particles[x].particleArea/surf
ace.particles[x].particleRegion)#adds two shares of the electron density from particle x to
particle x
                for y in range(len(surface.particles[x].particleNNs)):#iterates over all of
particle x's neighbors
                  neighbor = surface.particles[x].particleNNs[y]

electronDensityPerParticle[neighbor]+=electronsPerParticle[x]*(surface.particles[neighbo
```

```
r].particleArea/surface.particles[x].particleRegion)#adds one share of the electron density
from particle x to neighbor y

            else: #if electron spreading isn't used
                electronDensityPerPart = electronsPerParticle #electrons stay totally localized
            if electScaleTF == True: # if electron scaling is enabled, scale density by particle
size
                electronDensityPerPart =
[electronDensityPerPart[x]/suface.particles[x].particleArea for x in
range(surface.numParticles)]#scales the electron density by particle area (surface area)

            if timeRecTF == True: # if time recording is enabled, record all the following
statistics and store them in the timeTable
                timeRow =
[timestep,timestep*timestepSize,numDyes,charges,turnovers,dyeRecombinations,sum(cata
lystRecombinations),numCatalysts,excitations]
                timeTable[trial].append(timeRow)

            mol = 0#reset the molecule counter
            while mol < len(POI): #iterate over the POI list giving each molecule 1 turn
                point = POI[mol] #identifies the current molecule for easy reference
                position = surface.particles[point[0]].molecules[point[1]] # retrieves the set of
information about the current molecule for easy reference
                if position.typeOf == 1: #Currently looking at a dye
                    probRecomb = position.RecombRate*electronDensityPerPart[point[0]] #
retrieves the recombination probability for the dye
                    choiceValid=False # we haven't made a proper choice yet
                    while choiceValid == False: #until a proper choice has been made
                        weights = [i for i in position.HopRates] #assembled a list of weights to be
used for making a choice for this dye
                        weights.append(probRecomb)#add the recombination probability to the
weight list
                        if probRecomb+sum(position.HopRates)<1:#if the hopping and
recombination probabilities don’t add up to 1,
                            weights.append(1-probRecomb+sum(position.HopRates))#supplement
with a probability to do nothing
                        dyeChoice = weighted_choice(weights)#makes a weighted choice for the
current dye
                        if (dyeChoice >=len(position.HopRates)) or
(surface.particles[position.NNs[dyeChoice][0]].molecules[position.NNs[dyeChoice][1]].typ
eOf==1 and
surface.particles[position.NNs[dyeChoice][0]].molecules[position.NNs[dyeChoice][1]].OxSt
ate == 0) or
(surface.particles[position.NNs[dyeChoice][0]].molecules[position.NNs[dyeChoice][1]].typ
eOf==2 and
```

```
surface.particles[position.NNs[dyeChoice][0]].molecules[position.NNs[dyeChoice][1]].OxSt
ate < maxOxState):
                    choiceValid = True # if the choice was validated (was checked to make
sure not hopping to a full molecule)
                if dyeChoice < len(position.HopRates): #Hop to a neighboring position
                    hopTarget = position.NNs[dyeChoice]#identifies the neighbor being
hopped to for easy reference
                    hopTargetInfo =
surface.particles[hopTarget[0]].molecules[hopTarget[1]]#retrieves the info about the
target molecule for easy reference
                    if hopTargetInfo.typeOf == 1: #target is a dye
                        surface.particles[hopTarget[0]].molecules[hopTarget[1]].OxState +=1
#moves the hole to the new dye
                        surface.particles[point[0]].molecules[point[1]].OxState -=1 #takes the
hole away from the old dye
                        POI[mol]=hopTarget#changes the molecule listed in the POI to the new
one
                    else: #target is a catalyst
                        initialOxState = hopTargetInfo.OxState #records the initial oxidation
state of the target catalyst
                        surface.particles[hopTarget[0]].molecules[hopTarget[1]].OxState +=1
#moves the hole to the new catalyst
                        for z in range(len(hopTargetInfo.NNs)): #iterates through the target
catalyst's neighbors
                            neighbor = hopTargetInfo.NNs[z]#identifies the catalyst neighbor for
easy reference
                            targNeighborInfo =
surface.particles[neighbor[0]].molecules[neighbor[1]]#retrieves catalyst neighbor
information for reference
                            pos = targNeighborInfo.NNs.index(hopTarget) #identifies which index
neighbor the target catalyst was of its neighbor
                            if targNeighborInfo.typeOf == 1:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopDyetoCat[initialOxState] #changes the neighbor's hopping rate in accordance to
the catalysts new oxidation state and the fact that the  neighbor is a dye
                            if targNeighborInfo.typeOf == 2:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopCattoCat[initialOxState] #changes the neighbor's hopping rate in accordance to
the catalysts new oxidation state and the fact that the  neighbor is a catalyst
                        surface.particles[point[0]].molecules[point[1]].OxState -=1#removes the
hole from the old dye
                        POI[mol]=hopTarget#changes the molecule listed in the POI to the new
one
                        catalystSpecies[initialOxState]+=1#records the type of the catalyst
species changing
                        if initialOxState!=0:# if the catalyst was already oxidized
```

```
                        catalystSpecies[initialOxState-1]-=1 #removes the catalyst species that
is no longer present
                        POI.pop(mol)#removes the duplicate entry from the POI
                        mol-=1 #decrements the molecule counter to account for the fact that
the POI decreased in number


                    if dyeChoice == len(position.HopRates): #Recombine!
                      surface.particles[point[0]].molecules[point[1]].OxState -=1 #removes the
hole from the dye
                      POI.pop(mol) #removes the dye from the POI
                      dyeRecombinations+=1 #records the recombination event

                  if position.typeOf == 2: #Currently looking at a catalyst
                    choiceValid=False # we haven't made a proper choice yet
                    if position.OxState == maxOxState: #checks to see if the catalyst is full
                      catChoice = len(position.NNs)+2#sets the catalyst choice to do nothing
because turnover will be handled before the choice is made
                      choiceValid = True #sets the choice is valid so the choosing process can be
skipped
                      turnovers+=1 #records the turnover event
                      catalystSpecies[maxOxState-1]-=1 #reduces the number of maximally
oxidized catalyst species
                      surface.particles[point[0]].molecules[point[1]].OxState = 0#resets the
oxidation state of that catalyst to 0
                      for z in range(len(position.NNs)): #iterates over the catalysts neighbors
                        neighbor = position.NNs[z] #identifies the catalyst's neighbor for easy
reference
                        neighborInfo = surface.particles[neighbor[0]].molecules[neighbor[1]]
#retrieves the information about the catalyst's neighbor
                        pos = neighborInfo.NNs.index(point)#identifies which index neighbor
the target catalyst was of its neighbor
                        if neighborInfo.typeOf == 1:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopDyetoCat[0]#changes the neighbor's hopping rate in accordance to the catalysts
new oxidation state and the fact that the  neighbor is a dye
                        if neighborInfo.typeOf == 2:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopCattoCat[0]#changes the neighbor's hopping rate in accordance to the catalysts
new oxidation state and the fact that the  neighbor is a catalyst
                      POI.pop(mol) #removes the catalyst from the POI
                      mol-=1#decrements the molecule counter
                    initialOxState = surface.particles[point[0]].molecules[point[1]].OxState
#identifies the catalysts initial oxidation state
```

```
                probRecomb = position.RecombRate[position.OxState-
1]*electronDensityPerPart[point[0]]#calculates the recombination probability for the
catalyst
                while choiceValid == False: #until a valid choice has been made for this
catalyst
                    weights = [i for i in position.HopRates] #assembled a list of weights to be
used for making a choice for this catalyst
                    weights.append(probRecomb)#add the recombination probability to the
weight list
                    if probRecomb+sum(position.HopRates)<1:#if the hopping and
recombination probabilities don't add up to 1,
                        weights.append(1-probRecomb+sum(position.HopRates))#supplement
with a probability to do nothing
                    catChoice = weighted_choice(weights) #makes a weighted choice for the
current catalyst
                    if (catChoice >=len(position.HopRates)) or
(surface.particles[position.NNs[catChoice][0]].molecules[position.NNs[catChoice][1]].type
Of==1 and
surface.particles[position.NNs[catChoice][0]].molecules[position.NNs[catChoice][1]].OxSta
te == 0) or
(surface.particles[position.NNs[catChoice][0]].molecules[position.NNs[catChoice][1]].type
Of==2 and
surface.particles[position.NNs[catChoice][0]].molecules[position.NNs[catChoice][1]].OxSta
te < maxOxState):
                        choiceValid = True # if the choice was validated (was checked to make
sure not hopping to a full molecule)
                if catChoice < len(position.HopRates): #Hop to a neighboring position

                    hopTarget = position.NNs[catChoice]#identifies the neighbor being
hopped to for easy reference
                    hopTargetInfo =
surface.particles[hopTarget[0]].molecules[hopTarget[1]]#retrieves the information for the
hopping target
                    if hopTargetInfo.typeOf == 1: #target is a dye
                        surface.particles[hopTarget[0]].molecules[hopTarget[1]].OxState +=1
#moves the hole to the new dye
                        surface.particles[point[0]].molecules[point[1]].OxState -=1#removes the
hole from the old catalyst
                        for z in range(len(position.NNs)):#iterates through the neighbors of the
old catalyst
                            neighbor = position.NNs[z] #identifies the catalyst's neighbor for easy
reference
                            neighborInfo =
surface.particles[neighbor[0]].molecules[neighbor[1]]#retrieves the information about the
catalysts neighbor
```

```
                    pos = neighborInfo.NNs.index(point)#identifies which index neighbor
the target catalyst was of its neighbor
                    if neighborInfo.typeOf == 1:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopDyetoCat[initialOxState-1] #changes the neighbor's hopping rate in accordance to
the catalysts new oxidation state and the fact that the  neighbor is a dye
                    if neighborInfo.typeOf == 2:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopCattoCat[initialOxState-1] #changes the neighbor's hopping rate in accordance to
the catalysts new oxidation state and the fact that the  neighbor is a catalyst
                  catalystSpecies[initialOxState-1]-=1#removes the type of catalyst that is
no longer present
                  POI.append(hopTarget)#adds the newly oxidized dye to the POI
                  if initialOxState == 1: #if the catalyst is now depleted
                    POI.pop(mol)#remove the catalyst from the POI
                    mol-=1#steps  the molecule counter backward
                  else:# if the catalyst is not empty
                    catalystSpecies[initialOxState-2]+=1 #adds the type of catalyst that is
now present
                else: # target is a catalyst
                  hopTargetinitialOxState = hopTargetInfo.OxState#identifies the target's
initial oxidation state
                  surface.particles[hopTarget[0]].molecules[hopTarget[1]].OxState +=1
#moves the hole to the new catalyst
                  surface.particles[point[0]].molecules[point[1]].OxState -=1#removes the
hole from the old catalyst
                  for z in range(len(position.NNs)): #iterates over the old catalysts
neighbors
                    neighbor = position.NNs[z]#identifies the old catalyst's neighbor for
easy reference
                    neighborInfo = surface.particles[neighbor[0]].molecules[neighbor[1]]
#retrieves the information about the old catalysts neighbor
                    pos = neighborInfo.NNs.index(point) #identifies which index neighbor
the old catalyst was of its neighbor
                    if neighborInfo.typeOf == 1:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopDyetoCat[initialOxState-1] #changes the neighbor's hopping rate in accordance to
the old catalysts new oxidation state and the fact that the  neighbor is a dye
                    if neighborInfo.typeOf == 2:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopCattoCat[initialOxState-1] #changes the neighbor's hopping rate in accordance to
the old catalysts new oxidation state and the fact that the  neighbor is a catalyst
                  for z in range(len(hopTargetInfo.NNs)):#iterates over the new catalysts
neighbors
                    neighbor = hopTargetInfo.NNs[z]#identifies the new catalyst's
neighbor for easy reference
```

```
                    targNeighborInfo =
surface.particles[neighbor[0]].molecules[neighbor[1]] #retrieves the information about
the new catalysts neighbor
                    pos = targNeighborInfo.NNs.index(hopTarget) #identifies which index
neighbor the new catalyst was of its neighbor
                    if targNeighborInfo.typeOf == 1:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopDyetoCat[hopTargetinitialOxState]#changes the neighbor's hopping rate in
accordance to the new catalysts new oxidation state and the fact that the  neighbor is a dye
                    if targNeighborInfo.typeOf == 2:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
probHopCattoCat[hopTargetinitialOxState]#changes the neighbor's hopping rate in
accordance to the new catalysts new oxidation state and the fact that the  neighbor is a
catalyst
                  catalystSpecies[initialOxState-1]-=1 #reduces the count of the catalyst
species that is no longer present
                  catalystSpecies[hopTargetinitialOxState]+=1 #adds to the count of the
catalyst species that is now present
                  if hopTargetinitialOxState == 0: #if the target catalyst is newly oxidized
                    POI.append(hopTarget)#adds the new catalyst to the POI
                  else:# if the catalyst was already oxidized
                    catalystSpecies[hopTargetinitialOxState-1]-=1#reduces the count of
the catalyst species that has been removed
                  if initialOxState == 1: #if the original catalyst is now depleted
                    POI.pop(mol)#removes the old catalyst from the POI
                    mol-=1# steps the molecule counter back
                  else:# if the catalyst is not empty
                    catalystSpecies[initialOxState-2]+=1#adds the type of catalyst which
has now been created



                if catChoice == len(position.HopRates): #Recombine!
                  surface.particles[point[0]].molecules[point[1]].OxState -=1 #removes the
hole from the catalyst
                  catalystRecombinations[initialOxState-1]+=1 #records the recombination
event
                  for z in range(len(position.NNs)): #iterates over the catalysts neighbors
                    neighbor = position.NNs[z]#identifies the catalyst's neighbor for easy
reference
                    neighborInfo =
surface.particles[neighbor[0]].molecules[neighbor[1]]#retrieves the information about the
catalysts neighbor
                    pos = neighborInfo.NNs.index(point)
                    if neighborInfo.typeOf == 1:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] =
```

```
probHopDyetoCat[initialOxState-1]#changes the neighbor's hopping rate in accordance to the catalysts new oxidation state and the fact that the  neighbor is a dye
                    if neighborInfo.typeOf == 2:
surface.particles[neighbor[0]].molecules[neighbor[1]].HopRates[pos] = probHopCattoCat[initialOxState-1]#changes the neighbor's hopping rate in accordance to the catalysts new oxidation state and the fact that the  neighbor is a catalyst

                  catalystSpecies[initialOxState-1]-=1#reduces the count of the catalyst species that is no longer present
                  if initialOxState == 1: #if the catalyst is now depleted
                    POI.pop(mol) #removes the catalyst from the POI
                    mol-=1 #steps the molecule counter backward
                  else:# if the catalyst is not empty
                    catalystSpecies[initialOxState-2]+=1#adds to the count of the catalyst species that is now present
              mol+=1#moves on to the next molecule
            if CWModeTF == True: # if continuous illumination mode is enabled
              if timestep>maxTimeSteps: endCondition = True #if the simulation has run to the preset timestep, end
              if random.random()<= probExcite: #randomly determines whether or not to add an excitation this timestep
                excitations+=1 #adds to the counter of excitations
                choiceWeightingThisTime = choiceWeighting #retrieves the choiceWeighting table to decide where the excitation will occur
                for x in range(len(POI)):choiceWeightingThisTime[POI[x][0]][POI[x][1]]=0#iterates over all currently excite molecule and sets their probability of excitation to 0
                newDye = flattenedPositions[weighted_choice(flatten(choiceWeightingThisTime))]#makes a new excitation choice
                surface.particles[newDye[0]].molecules[newDye[1]].OxState=1#adds a hole to the chosen dye
                POI.append(newDye)#adds the dye to the POI

        #END OF THE MAIN LOOP
        print("----------------------END OF RUN-------------------------")#prints out some run statistics
        print("Number of initially Excited Dyes: "+str(len(exciteArray)))
        print("Number of Dyes Remaining: "+str(numDyes))
        print("Number of 1st Ox Catalysts: "+str(catalystSpecies[0]))
        if maxOxState>=2: print("Number of 2nd Ox Catalysts: "+str(catalystSpecies[1]))
        if maxOxState>=3: print("Number of 3rd Ox Catalysts: "+str(catalystSpecies[2]))
        if maxOxState>=4: print("Number of 4th Ox Catalysts: "+str(catalystSpecies[3]))
        if CWModeTF==True: print("Number of Excitations: "+ str(excitations))
        print("Number of Dye Recombinations: "+str(dyeRecombinations))
        print("Number of Catalyst Recombinations: "+str(sum(catalystRecombinations)))
```

```
            print("Number of Turnovers: "+str(turnovers))
            sanityCheck =
turnovers*maxOxState+numDyes+dyeRecombinations+sum(catalystRecombinations)+su
m([(i+1)*catalystSpecies[i] for i in range(len(catalystSpecies))])
            print("Sanity Check: "+str(sanityCheck)+" should equal "+
str(len(exciteArray)+excitations))

            if timestep<shortestTrial: shortestTrial=timestep #keeps track of which run in a
trials loop has been the shortest
            turnoverTotals+=turnovers#add the turnovers for this trial to the running total
        #END OF THE TRIALS LOOP
        if TurnoverRecTF == True: # if turnovers are being recorded
            turnoverAverage = turnoverTotals/trials #calculates an average turnover
            turnoverPercent =
maxOxState*turnoverAverage/numInitiallyExcitedDyes#calculates an average percent
turnover
            turnoverRow =
[round(R*10**9),round(H*10**9),tauRatio,turnoverPercent,turnoverAverage]#records the
turnover statistics
            turnoverTable[parameterPoint+1] = turnoverRow#adds the turnover stats for the
current parameter point to the turnover table
            exportCSV(turnoverFilename,turnoverTable)#exports the current turnover table
        if AnisRecTF == True:#if anisotropy is being recorded
            anisMean = [[sum([anisTable[i][k][j] for i in range(trials)])/(trials) for j in
range(4)] for k in range(shortestTrial)] #calculates anisotropy average over trials which is
truncated to the shortest trial
            anisMean.insert(0,anisHeader)#adds a header to the anisotropy table
            anisotropyFilenameTemp=anisotropyFilename[0:-
4]+"_"+str(parameterPoint)+"_"+anisotropyFilename[-4:len(anisotropyFilename)] #makes
a temporary filename for this parameter point
            exportCSV(anisotropyFilenameTemp,anisMean)#exports the anisotropy result for
the current parameter point
        if timeRecTF == True:#if time statistics are being recorded
            timeMean = [[sum([timeTable[i][k][j] for i in range(trials)])/(trials) for j in
range(9)] for k in range(shortestTrial)] #calculates time statistics average over trials which
is truncated to the shortest trial
            timeMean.insert(0,timeHeader)#adds a header to the time statistics table
            timeFilenameTemp=timeFilename[0:-
4]+"_"+str(parameterPoint)+"_"+timeFilename[-4:len(timeFilename)]#makes a temporary
filename for this parameter point
            exportCSV(timeFilenameTemp,timeMean)#exports the time statistics for the
current parameter point

        noCatRecombinations = noCatRecombinations/trials #calculates the average number
of recombinations for not having catalysts
```

```
        loneChargeRecombinations = loneChargeRecombinations/trials #calculates the
average number of recombinations for having too few excitations
        autoRecombinations = autoRecombinations/trials#calculates the average number of
recombinations made automatically for either of ther reasons above

    #END OF THE HOPPING LOOP
  #END OF THE RECOMBINATION LOOP
```

APPENDIX E. Water from Waves Procedure

Water from Waves Activity

Materials Needed

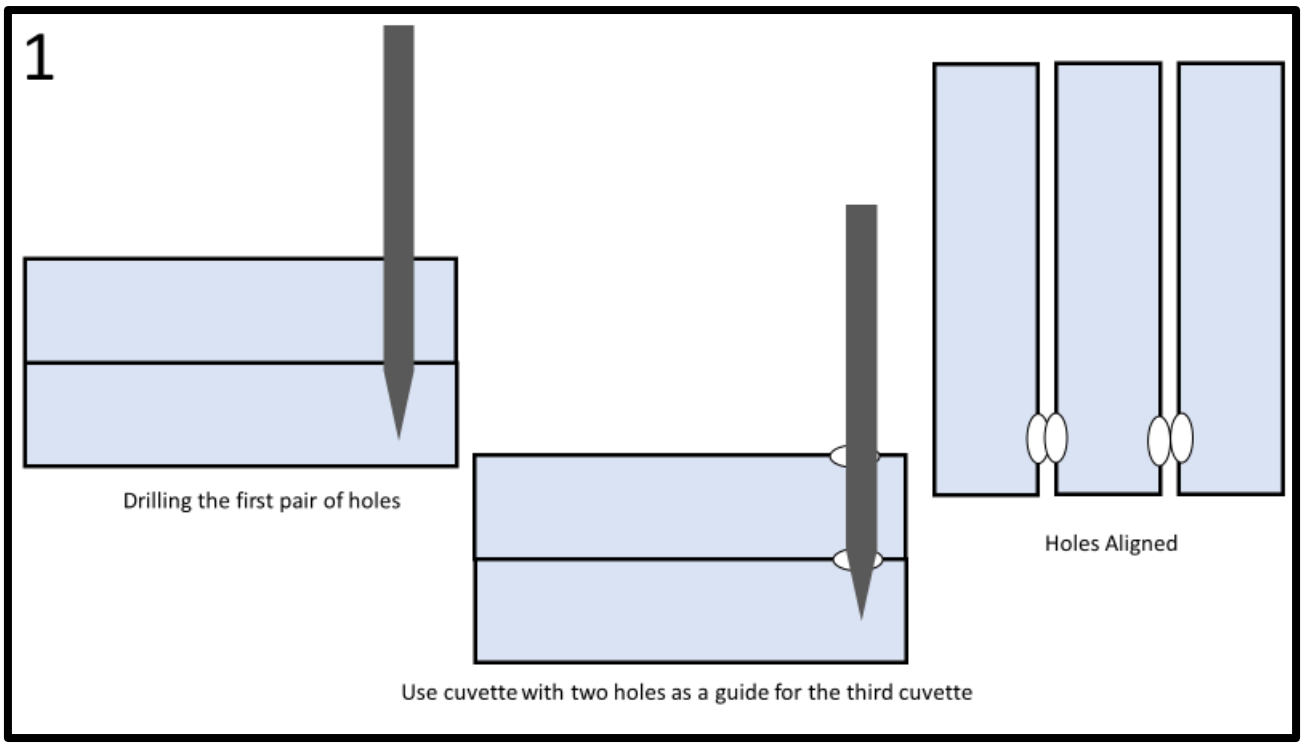
1. Anion exchange membranes, cut into 1 cm x 1 cm squares (1 ea)
2. Cation exchange membranes, cut into 1 cm x 1 cm squares (1 ea)
3. Standard plastic cuvettes, 2.5 mL (3 ea)
4. Teflon tape
5. Carbon cloth, cut into 0.9 cm by 4 cm strips (2 ea)
6. 9 volt battery (1 ea)
7. Alligator connection wires (2 ea)
8. Thymol blue pH indicator
9. Aqueous 5 mM H_2SO_4 solution (~4 mL ea)
10. A hand clamp (1 ea)
11. A hand drill or drill press
12. Scissors
13. Tweezers
14. Permanent marker

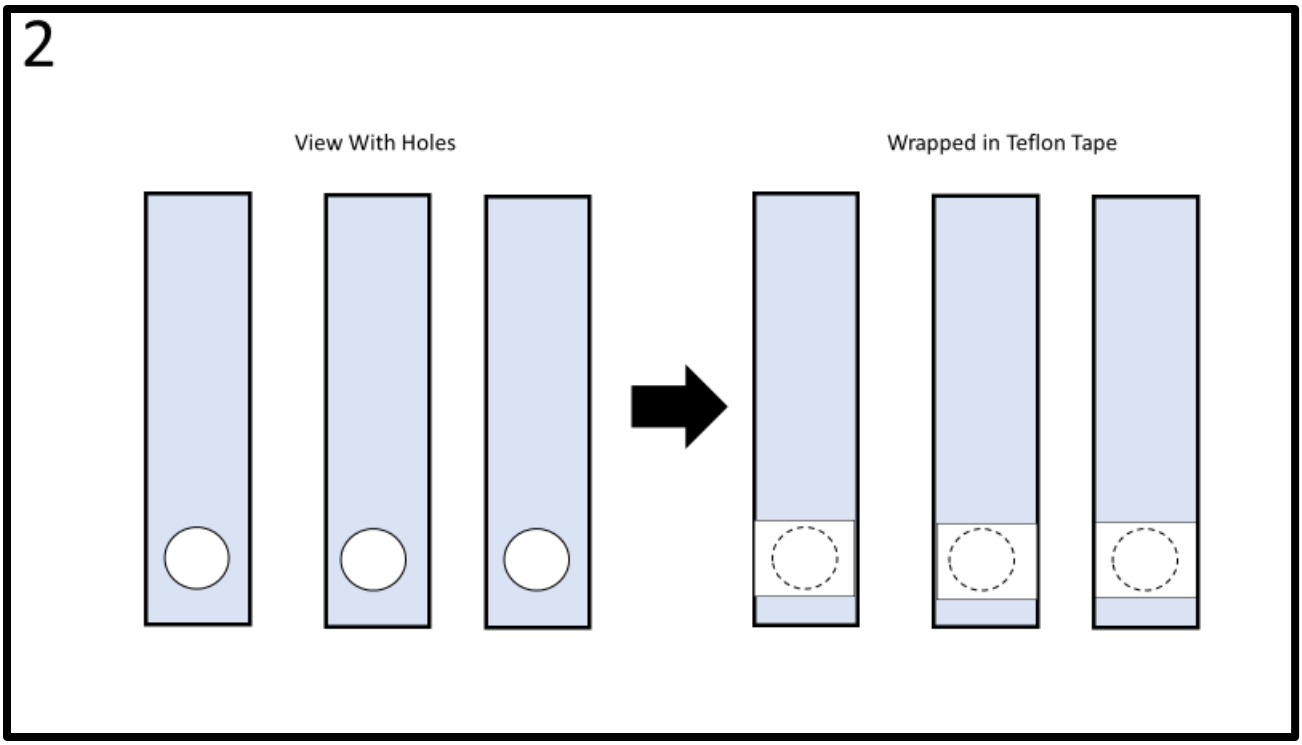
Personal Protective Equipment

1. Protective eyewear
2. Gloves
3. Long sleeves, long pants, and closed-toed shoes
4. Lab coats or aprons, if possible

Water from Waves Procedure

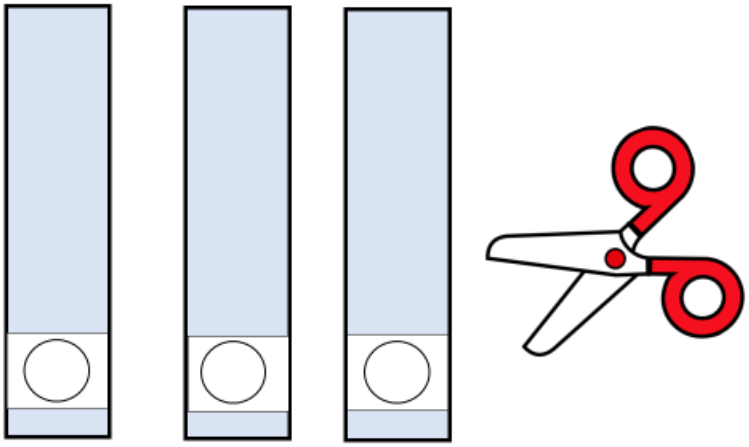
1) Drill 8 mm holes near bottom of plastic cuvettes. Two cuvettes should have holes drilled through one side while the third cuvette should have holes drilled through all the way. This is easiest to do by clamping a pair of cuvettes together and drilling one hole through one completely and through the first side of the other. Repeat the process for the third cuvette using the one with holes on both sides again as a guide.
2) Wrap Teflon tape over cuvette holes twice around. This is to ensure a tight seal when the cuvettes are brought together. The layers of Teflon tape used may need to be adjusted if cells leak based on thickness of tape and strength of clamps used.
3) Using scissors, cut holes in the Teflon tape over the drilled holes.
4) Label one side cuvette with a 'C' and the other side with an 'A'. These will be the cathode and anode chambers, respectively.
5) Using tweezers, stack the cuvettes and membranes in order: the cathode chamber hole side up, a CEM, the central chamber, an AEM, and the anode chamber hole side down. Make sure the membranes fully cover the holes and that all the holes are aligned as much as possible.
6) Clamp everything together while keeping it all aligned.
7) Rinse carbon cloth electrodes with deionized water. Slide a carbon cloth electrode onto each of the outside walls of the outer chambers
8) Attached alligator clip wires to carbon cloth electrodes in a way that does not pull them out. A good way to do this is to clamp onto both the electrode and the outer wall of the cuvette.
9) Fill each chamber approximately halfway with aqueous 5 mM H_2SO_4.
10) Attached the battery by connecting the alligator clip wire connected to the cathode to the '–' terminal on the battery and the alligator clip wire attached to the anode to the '+' terminal on the battery.
11) Bubbles will form on the surface of the electrodes. Allow this to run for approximately 30 minutes.
12) Carefully disconnect the battery.
13) Remove the electrodes.
14) Add a few drops of pH indicator to each chamber and notice the color differences. It may also help to fill another separate cuvette halfway with your starting acid and add a few drops of pH indicator for comparison.


1
Drilling the first pair of holes
Holes Aligned
Use cuvette with two holes as a guide for the third cuvette


2
View With Holes
Wrapped in Teflon Tape

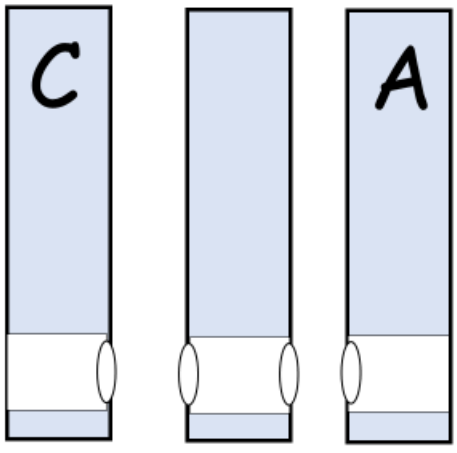
3

Cut out holes



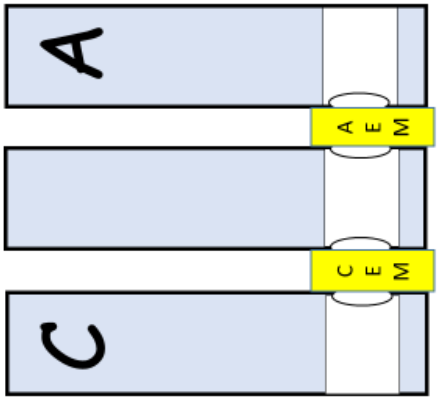
4

Front View and Labeled



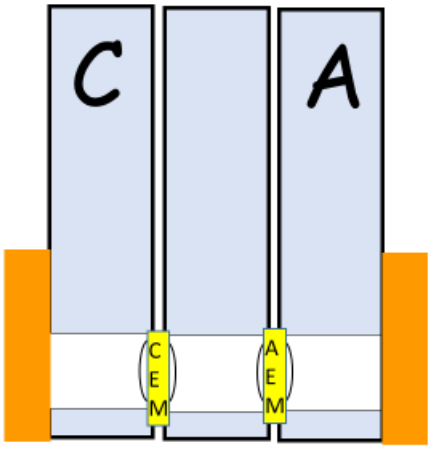
5

Aligning Compartments

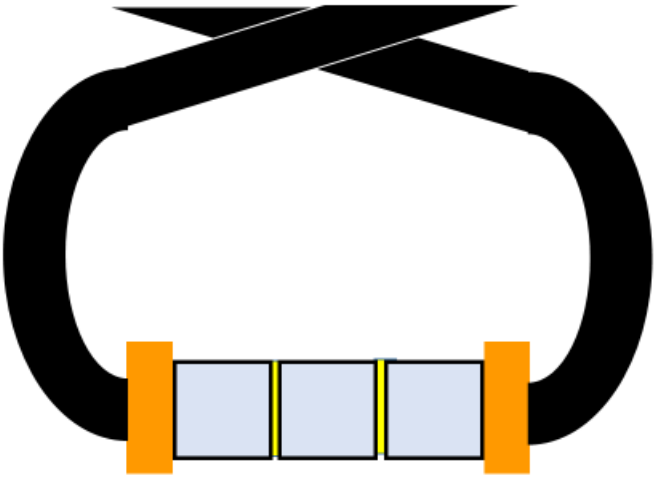


6

Clamp Together

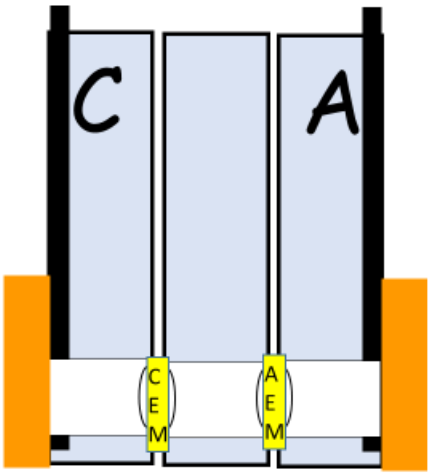


Top View

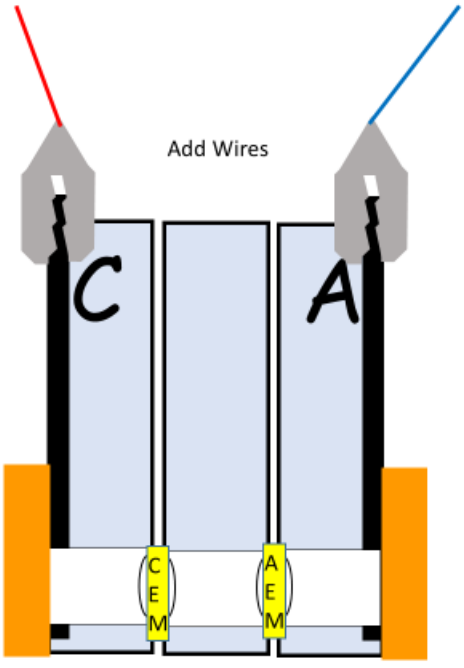


7

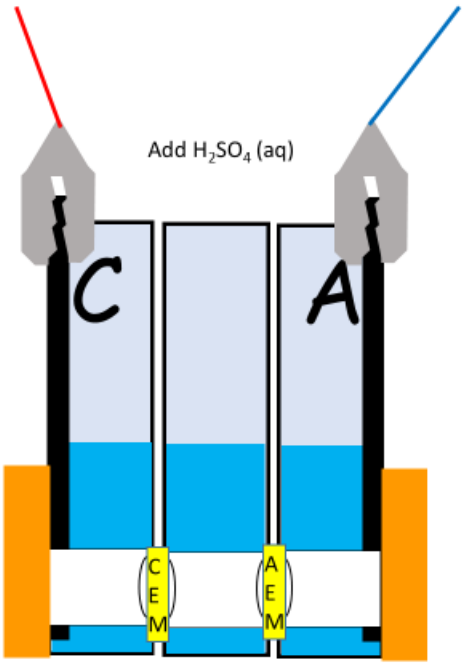
Add Electrodes



8



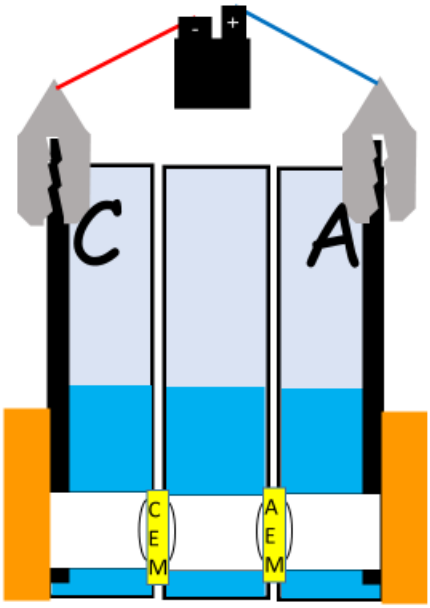
9



10

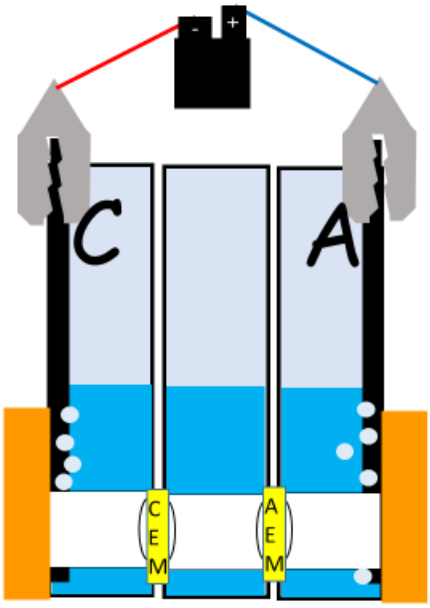
Note the battery orientation!

Attach Battery:
+ side to 'A' Chamber
– side to 'C' Chamber



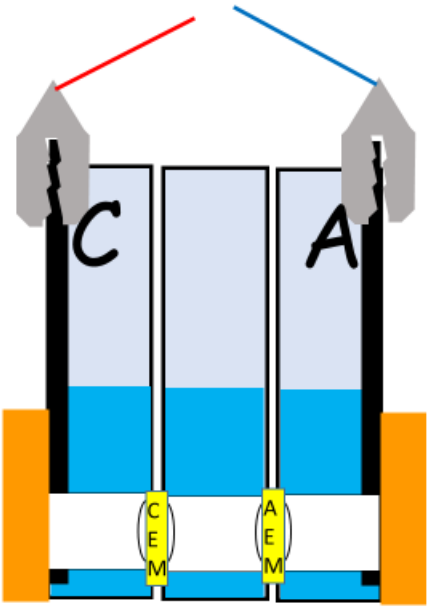
11

Allow to Run



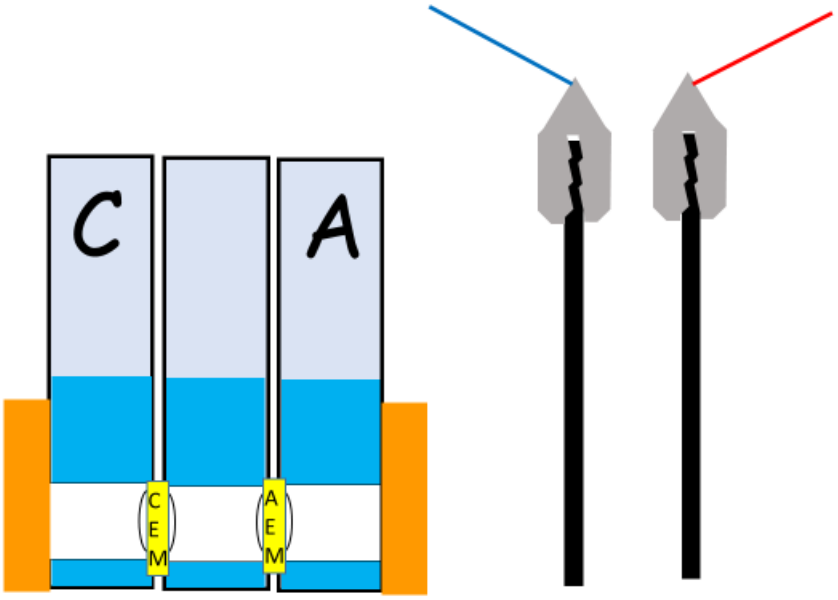
12

Disconnect

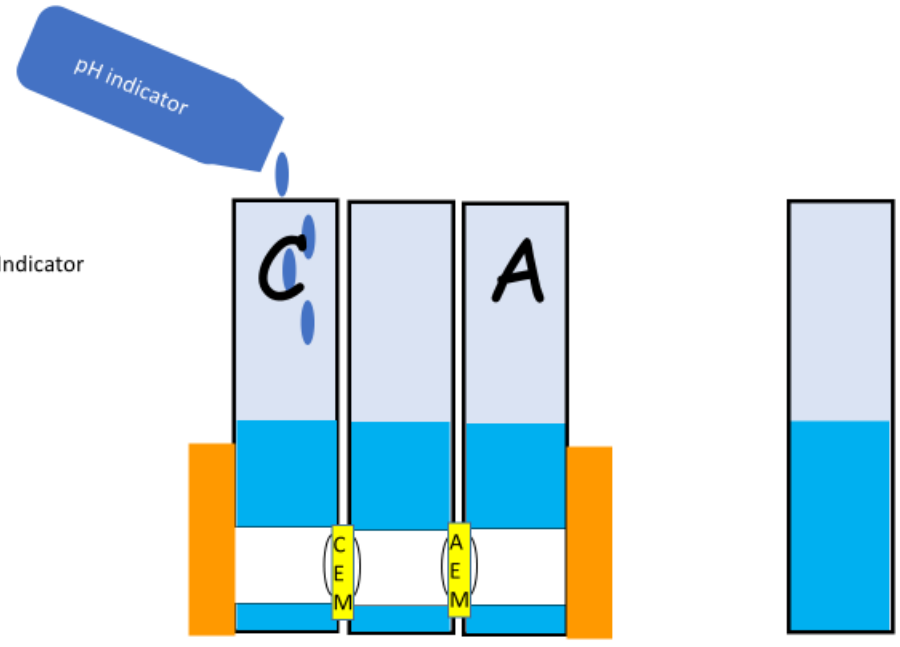


13

Disassemble

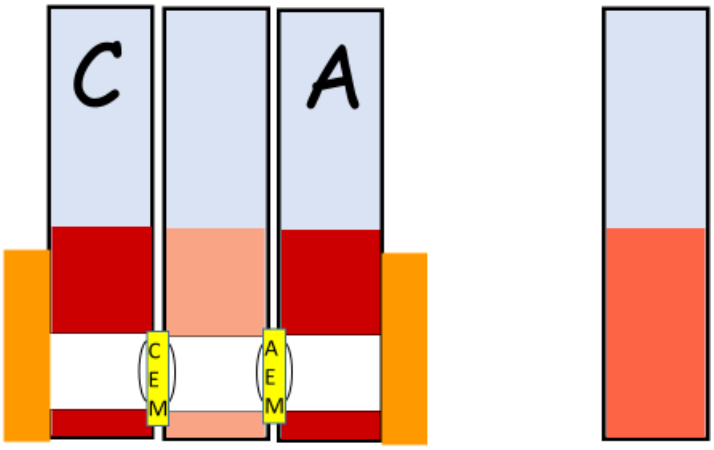


14



15

Notice color changes, and
Compare with starting solution



APPENDIX F. Z-Scheme Reactor Design and Fabrication

Scalable technologies for solar-energy conversion and storage must be efficient, robust, and inexpensive to manufacture. Recent techno-economic analyses of H_2 production suggest that it may be cost-competitive to create solar water splitting reactors using suspensions of nanoparticles to drive hydrogen and oxygen evolution chemistry.[1,2] These reactors could largely be made of flexible plastic to minimize manufacturing costs and would

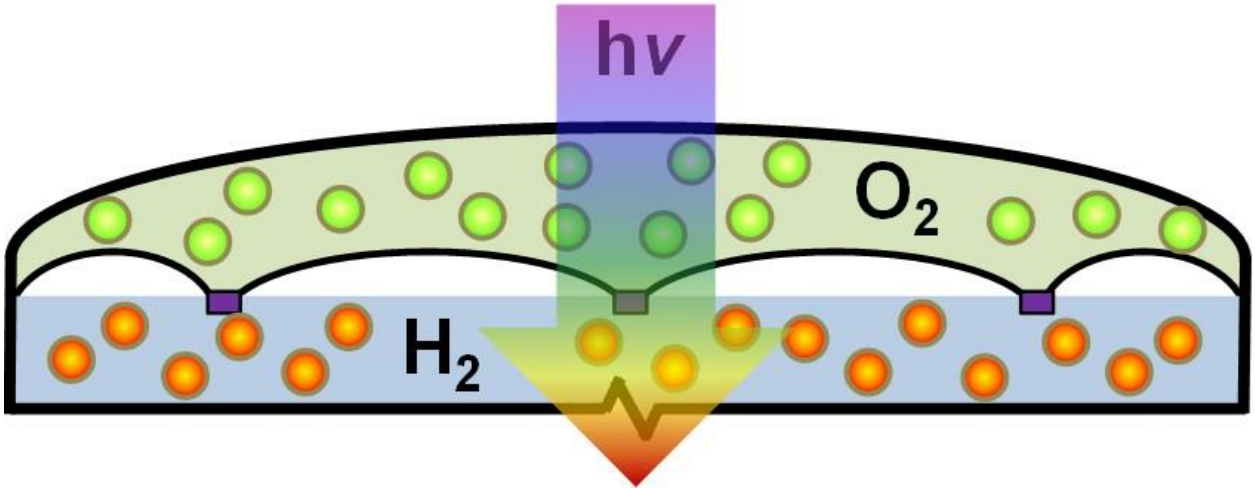


Figure F1. Ardo Group design concept for cost-competitive solar water splitting reactor. Oxygen and hydrogen are produced in separate chambers in series with each other.

evolve hydrogen and oxygen in separate chambers to both avoid the creation of explosive gas mixtures and to reduce the cost of separating them later. This could be achieved by filling each baggie with either hydrogen or oxygen evolving nanoparticles. These nanoparticles, when exposed to light would evolve the desired gas while simultaneously reducing or oxidizing a redox shuttle species. This redox shuttle species would be able to travel between chambers through vias which permit the shuttle to transport through while disallowing the transport of produced gasses or the nanoparticles themselves. In this way, the reactor would separately produce each product gas while doing no net chemistry to a redox shuttle while it ferries electronic and ionic charges between the chambers. The Ardo Labs' innovation on

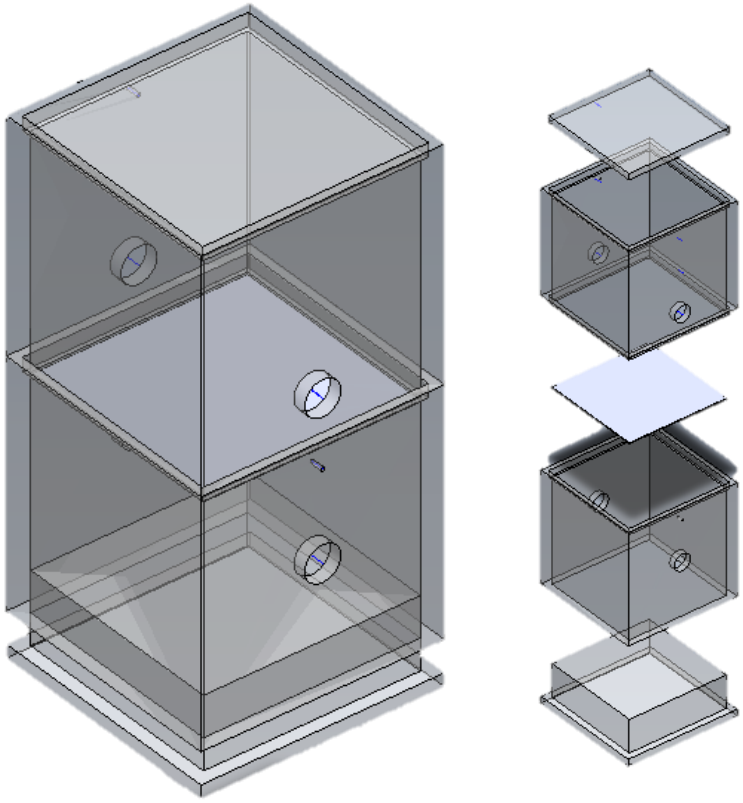


Figure F2. Original proposed design schematic of plexiglass reactor shown assembled on the left and in an exploded view on the right

this concept was to stack these chambers as shown in Figure F1 rather than placing them side-by-side. One of the major limitations with the side-by-side design was that it required the redox shuttle to travel great distances past a reasonable diffusion length and so it would require forced convection, likely pumping, to mix the redox shuttle between chambers. This more than doubled the project capital cost of the reactor.[1,2] Alternatively, the side-by-side design could make use of very many smaller chambers to limit diffusion requirements but this in turn greatly increased design and manufacturing complexity, and therefore cost. By placing the chambers on top of one another, they have much larger contact areas and small maximum redox shuttle transport distance. This does introduce additional competitive light absorption, which needs to be accounted for, but generally reduces projected costs by eliminating the need for active pumping and pipes and can result in increased efficiency due to tandem serial light abruption by the two sets of nanoparticles.

Figure F3. a) Gasket running around the top edge of the lower chamber. b) Clamping added to maintain seal between chambers.

My role in this project what to develop a laboratory-scale prototype reactor by which we could evaluate the reasonability of this proposed setup. The concept for the final field reactor was essentially two large acre-sized plastic bags, each less than 10 cm tall, that could lie on top of each other. That was challenging to evaluate on the small scale and so I developed a design made of plexiglass as shown in Figure F2. This design was rigid rather than flexible but would provide a reliable testing environment for the nanoparticles that our team was developing. The initial design was for a pair of chambers machined out of plexiglass, which could interlock with an intervening membrane separator. This membrane was supposed to be analogous to the vias in the proposed design concept. The bottom of the lower chamber also included a plunger so that the bottom chamber thickness could be controlled. The top chamber height could be controlled simply by changing the filling level of the solution. This way both chamber heights could be adjusted as the experimental conditions called for it. The top lid would be made of glass instead of plexiglass to allow for increased light transmission in the ultraviolet region of the electromagnetic spectrum.

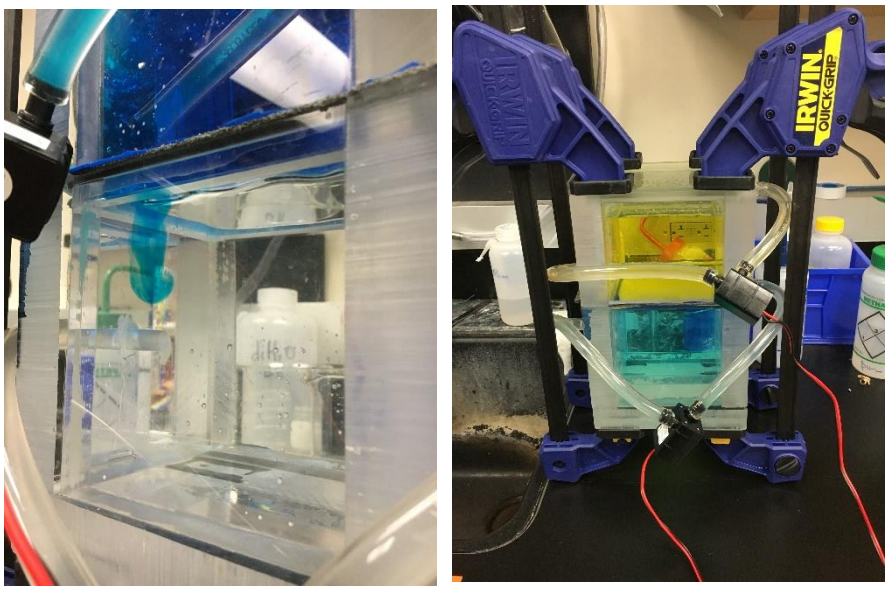


Figure F4. a) Leak test showing the membrane being circumvented. b) Long running leak test which showed little to no crossover

In the end, several parts of this design proved to be impractical and had to be adapted. One major adjustment was the base plunger. It turned out to be nearly impossible to get a reliable seal around this square plunger and so it was decided that instead the bottom of the lower chamber would simply be sealed on permanently and the bottom chamber height could be adjusted by placing plexiglass spacers into the bottom of the chamber. While not elegant, this was much easier to implement.

Another design challenge turned out to be the sealing of the two chambers while holding a membrane between them. The initial concept was to run a gasket around the top rim of the bottom chamber so that when the top chamber was place in it, a water-tight seal would be made. However, it turns out that it takes quite a lot of pressure to engage a gasket with that much surface area and so extra clamping was required to hold the two chambers together while engaging the gasket.

Finally, there were leakage problems around the membrane. Even though the chamber was now sealed effectively from the outside and able to pin a membrane material in place, the contact with the membrane was not water tight and therefore instead of

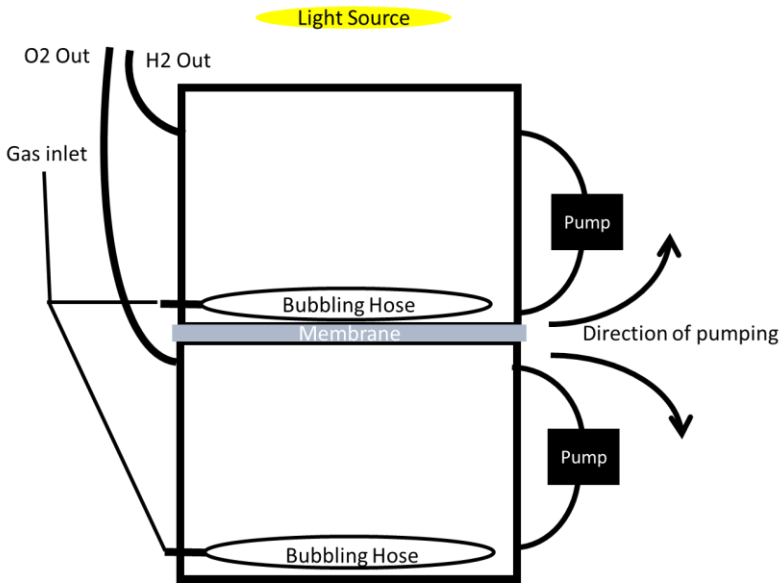


Figure F5. Schematic of reactor features and geometry

allowing for slow diffusion across a large area, water was able to pass from one chamber to another simply but going around the membrane. This defeats the purpose of having a membrane in the first place, so more leak testing had to be performed. To overcome this challenge, it was necessary to clamp down the membrane but not in a rough localized manner such as a gasket but rather with foam tape which would seal against the membrane without potentially tearing it. This conclusion was reached after testing a variety of sealing materials and using food coloring dyes as a leak indicator. These dyes were slightly larger that proposed redox shuttles would be but smaller than the nanoparticles we would be using and so should be able to slowly diffuse across a membrane. After much trial and error, foam tape was settled on as adequate as it made a decent seal although after leaving clamped for a long time would wear out and need periodic replacement.

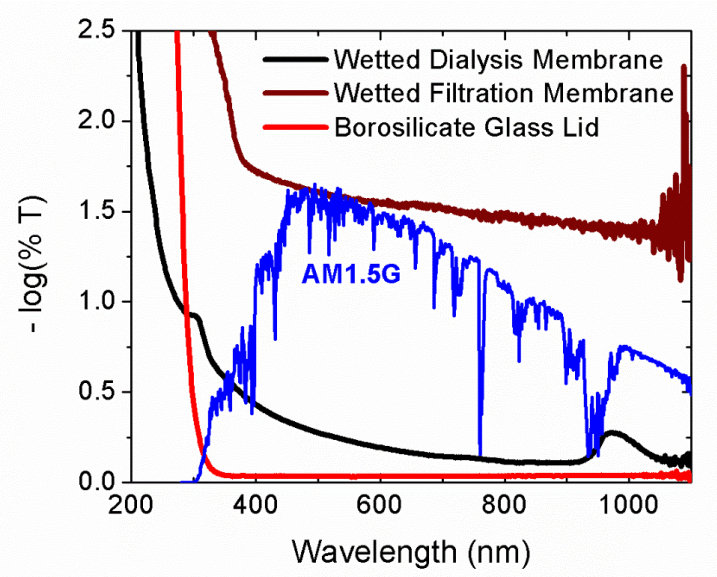


Figure F6. Absorption spectra comparing various membrane options

Once the basic chambers were machined and sealed, other features had to be added. Two gas vent ports were added to the top of each chamber such that produced gasses could be continually collected and analyzed using our inline mass spectrometer. Additionally, pumps were added to each chamber so that water could be flowed from near the membrane to far from the membrane in the same chamber with controlled flow rate. While the original design concept hoped to eliminate this sort of pumping altogether, being able to test how much pumping improved results was necessary.

Additionally, it was decided that it might be best to bubble gases through these chambers both to add to convective mixing as well as help with product collection. To implement this, aeration hoses were added to each chamber and connected to gas inlet ports added to the sides. At this point the reactor essentially reached all of our experimental capability expectations with the features implemented in Figure F5.

After reactor completion, a suitable membrane material had to be selected. This turned out to be a challenge because the membrane had to allow for the passage of both light and redox shuttle while excluding nanoparticles and product gasses. On top of that it had to

	Dialysis Membrane	Snyder ultrafiltration Membrane	Genpore Plastic	Polyvinyl Membrane
Transparent		Opaque	Opaque	
Dye Diffusion	Slow		No diffusion	
NP diffusion	Some leakage	–	–	Macroscopic Holes
Physically Robust				Fell apart

Table F1. Potential membrane materials evaluation.

be physically robust and capable of operating in non-neutral pH conditions as required by many of the nanoparticles of interest. It also had to be inexpensive because this was supposed to span acres in practice. Turns out, there is no such membrane that exists today and thus this is a challenge for future research, which was beyond the scope of our funded project.