Appendix - Code

0.1 Raw Data

[NADP] = 8e-5 M
[ATP] = 1.5 mM
[ADP] = 0.5 mM
[NADPH] = 0.1 mM
[P] = 100 mM
[RuBP] = 0.15 mM
[O2] = 0.3 mM
[CO2] = 0.3 mM

In [1]:
```
%matplotlib inline
# All the imports
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from IPython.display import display, HTML
import ipywidgets as ipw
import logging

logging.disable(logging.CRITICAL)


avogadro = 6.022e23
```

In [2]:
```
chloroplast_volume = 3.3e-14  # Liters
grana_stack_height = 1.18e-7  # Meters
grana_stack_endcap_area = 1.6e-13 # Square meters
grana_per_chloroplast = 60

height_extension = 1.2  # Increase height by 10% on either end
radius_extension = 1.1  # Increase radius by 10%

single_grana_volume = (1e3 * grana_stack_height *  # Liters
                       grana_stack_endcap_area)
grana_volume = grana_per_chloroplast * single_grana_volume  # Liters

single_grana_extended_volume = (1e3 * height_extension *  # Liters
                                grana_stack_height *
                                radius_extension**2 *
                                grana_stack_endcap_area)

grana_extended_volume = (grana_per_chloroplast *  # Liters
                         single_grana_extended_volume)

stroma_volume = chloroplast_volume - grana_volume
```

```
        reaction_volume =  grana_extended_volume - grana_volume
        diffusion_volume = stroma_volume - reaction_volume


        print("Stroma Volume : {:.3e} Liters".format(stroma_volume))
        print("Reaction Volume : {:.3e} Liters".format(reaction_volume))
        print("Diffusion Volume : {:.3e} Liters".format(diffusion_volume))
```

```
Stroma Volume : 3.187e-14 Liters
Reaction Volume : 5.120e-16 Liters
Diffusion Volume : 3.136e-14 Liters
```

```
In [3]: reactant_map_1 = {"NADP": 0,
                         "ATP": 1,
                         "ADP": 2,
                         "NADPH": 3,
                         "O2": 4,
                         "CO2": 5}

        reactant_keys = list(reactant_map_1.keys())  # Stays in consistent order

        # Moles per liter
        reactant_conc_in_stroma = np.array([0.00008,  # NADP
                                            0.0015,   # ATP
                                            0.0005,   # ADP
                                            0.0001,   # NADPH
                                            0.0003,   # O2
                                            0.0003])  # CO2


        reactants_in_diffusion_vol = np.int_(diffusion_volume * avogadro *
                                             reactant_conc_in_stroma)
        reactants_in_reaction_vol = np.int_(reaction_volume * avogadro *
                                            reactant_conc_in_stroma)


        print("Reactants in Diffusion Volume")
        print("=============================")
        for reactant in reactant_keys:
            print("Total {} molecules in diffusion volume : {:}".format(
                    reactant,
                    reactants_in_diffusion_vol[reactant_map_1[reactant]]))

        print()
        print()

        print("Reactants in Reaction Volume")
```

```
        print("============================")
        for reactant in reactant_keys:
            print("Total {} molecules in reaction volume : {:}".format(
                    reactant,
                    reactants_in_reaction_vol[reactant_map_1[reactant]]))
```

```
Reactants in Diffusion Volume
============================
Total O2 molecules in diffusion volume : 5664625
Total NADP molecules in diffusion volume : 1510566
Total ATP molecules in diffusion volume : 28323129
Total NADPH molecules in diffusion volume : 1888208
Total ADP molecules in diffusion volume : 9441043
Total CO2 molecules in diffusion volume : 5664625


Reactants in Reaction Volume
============================
Total O2 molecules in reaction volume : 92502
Total NADP molecules in reaction volume : 24667
Total ATP molecules in reaction volume : 462512
Total NADPH molecules in reaction volume : 30834
Total ADP molecules in reaction volume : 154170
Total CO2 molecules in reaction volume : 92502
```

In [4]:

```
        # _d are in the diffusion volume,
        # _r are in the reaction volume
        reactant_names = ['CO2_d', 'O2_d',
                          'ATP_d', 'ADP_d',
                          'NADPH_d', 'NADP_d',
                          'CO2_r', 'O2_r',
                          'ATP_r', 'ADP_r',
                          'NADPH_r', 'NADP_r',
                          'Glucose']

        # Index map of the reactants
        reactant_map = {reactant:reactant_names.index(reactant)
                        for reactant in reactant_names}

        # For calculating diffusion parameters.
        reactant_masses = np.array([44.01,     # CO2_d
                                    32.00,     # O2_d
                                    507.18,    # ATP_d
```

```
                                427.20,     # ADP_d
                                745.42,     # NADPH_d
                                744.42,     # NADP_d
                                44.01,      # CO2_r
                                32.00,      # O2_r
                                507.18,     # ATP_r
                                427.20,     # ADP_r
                                745.42,     # NADPH_r
                                744.42])    # NADP_r


# Starting number of reactants
reactants = np.array([5664625,      # CO2_d
                      5664625,      # O2_d
                      28323129,     # ATP_d
                      9441043,      # ADP_d
                      1888208,      # NADPH_d
                      1510566,      # NADP_d
                      92502,        # CO2_r
                      92502,        # O2_r
                      462512,       # ATP_r
                      154170,       # ADP_r
                      30834,        # NADPH_r
                      24667,        # NADP_r
                      0])           # Glucose


# Specifies which reactants are in each reaction
reactions = np.array([
#                              Dark reaction
#                         Light reaction  |
#                           NADP diff. |  |
#                      NADPH diff. |  |  |
#                     ADP diff. |  |  |  |
#                  ATP diff. |  |  |  |  |
#           O2 int. diff. |  |  |  |  |  |
#       CO2 int. diff. |  |  |  |  |  |  |
#     O2 ext. diff. |  |  |  |  |  |  |  |
# CO2 ext. diff. |  |  |  |  |  |  |  |  |
#              |  |  |  |  |  |  |  |  |  |
             [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],  # CO2_d
             [0, 1, 0, 1, 0, 0, 0, 0, 0, 0],  # O2_d
             [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],  # ATP_d
             [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],  # ADP_d
             [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],  # NADPH_d
             [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],  # NADP_d
             [0, 0, 1, 0, 0, 0, 0, 0, 0, 1],  # CO2_r
             [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],  # O2_r
```

```
                [0, 0, 0, 0, 1, 0, 0, 0, 0, 1],  # ATP_r
                [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # ADP_r
                [0, 0, 0, 0, 0, 0, 1, 0, 0, 1],  # NADPH_r
                [0, 0, 0, 0, 0, 0, 0, 1, 1, 0],  # NADP_r
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]) # Glucose

        # Specifies the change in the reactants for each reaction
        state_change = np.array([
        #                                              Dark reaction
        #                                       Light reaction   |
        #                                       NADP diff.   |   |
        #                                NADPH diff.   |     |   |
        #                           ADP diff.   |     |     |   |
        #                      ATP diff.   |    |     |     |   |
        #                 O2 int. diff.   |    |     |     |   |
        #            CO2 int. diff.   |    |     |     |     |   |
        #       O2 ext. diff.   |     |    |     |     |     |   |
        # CO2 ext. diff. __|  __|  __|  __|  __|  __|  __|  __| __|
        #           |    |    |    |    |    |    |    |    |   |
                [100,   0, 100,   0,   0,   0,   0,   0,   0,   0],  # CO2_d
                [  0, 100,   0, 100,   0,   0,   0,   0,   0,   0],  # O2_d
                [  0,   0,   0,   0, 100,   0,   0,   0,   0,   0],  # ATP_d
                [  0,   0,   0,   0,   0, 100,   0,   0,   0,   0],  # ADP_d
                [  0,   0,   0,   0,   0,   0, 100,   0,   0,   0],  # NADPH_d
                [  0,   0,   0,   0,   0,   0,   0, 100,   0,   0],  # NADP_d
                [  0,   0,-100,   0,   0,   0,   0,   0,   0,  -6],  # CO2_r
                [  0,   0,   0,-100,   0,   0,   0,   0,   1,   0],  # O2_r
                [  0,   0,   0,   0,-100,   0,   0,   0,   3, -18],  # ATP_r
                [  0,   0,   0,   0,   0,-100,   0,   0,  -3,  18],  # ADP_r
                [  0,   0,   0,   0,   0,   0,-100,   0,   2, -12],  # NADPH_r
                [  0,   0,   0,   0,   0,   0,   0,-100,  -2,  12],  # NADP_r
                [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1]]) # Glucose

        # Keys are the columns of the reaction and
        # state change matrices
        reaction_map = {0:  "DM", # CO2 Membrane diffusion
                        1:  "DM", # O2 Membrane diffusion
                        2:  "DI", # CO2 internal diffusion
                        3:  "DI", # O2 internal diffusion
                        4:  "DI", # ATP diffusion
                        5:  "DI", # ADP diffusion
                        6:  "DI", # NADPH diffusion
                        7:  "DI", # NADP diffusion
                        8:  "R", # Light reaction
                        9:  "R"} # Dark reaction

In [26]: class Model(object):
             """Simple data structure for storing reaction info."""
```

```
    def __init__(self,
                 reactant_names,
                 reactant_map,
                 reactant_masses,
                 reactants,
                 reactions,
                 state_change,
                 reaction_map,
                 reaction_volume,
                 diffusion_volume):
        self.reactant_names = reactant_names
        self.reactant_map = reactant_map
        self.reactant_masses = np.sqrt(reactant_masses)
        self.reactants = reactants
        self.reactions = reactions
        self.state_change = state_change
        self.reaction_map = reaction_map
        # Avoid repeated division by doing it now.
        self.rvd = 1/(reaction_volume*avogadro)
        self.dvd = 1/(diffusion_volume*avogadro)
        self.log = []  # For counting reactions.

    def get_reactants(self):
        return self.reactants.copy()

    def update_reactants(self, reaction, sgn):
        self.log.append(reaction)
        self.reactants += sgn*self.state_change[:,reaction]



class Simulator(object):

    def __init__(self, model, params, callback):
        """Runs a stocastic chemical reaction simulation."""
        self.model = model
        self.params = params
        self.callback = callback

    def run(self, t):
        """
        Runs the simulation

        Arguments
        ---------
        t : float
            The start time of the reaction
```

```
        Returns
        -------
        T : ndarray
            A vector of the time points at which the
            reactants were updated.

        R : ndarray
            A (number of reactants) x (len(T)) matrix
            of the reactant counts at the time points in T.
        """
        # Output initialization
        T = [t]
        R = [self.model.get_reactants()]

        while T[-1] < self.params['time'] + t:

            # Calculate the current reaction rates
            # and the sign of the reaction (diffusion goes
            # two ways.)
            rates, sgn = self.calc_rates()
            #print(rates)

            # Get the time til update tau and
            # a random variable to pick which reaction
            # occurs.
            tau, r = self.calculate_update(rates)

            # Get the reaction from r and the reaction rates.
            reaction = self.get_reaction(rates, r)

            # Update the reactant counts.
            self.model.update_reactants(reaction, sgn[reaction])

            # Append results to the output
            T.append(T[-1] + tau)
            R.append(self.model.get_reactants())

            # Update the progress bar in the display widget.
            self.callback(tau)

        return T, R

    def calc_rates(self):
        """
        Calculates the instantaneous reaction rates
        from current species counts and parameters.
```

```
        Returns
        -------
        rates : ndarray
            Instantaneous reaction rates for all reactions.
        sgn : ndarray
            Vector of +/-1's indication the direction of
            each reaction.
        """

        # Turn counts into concentrations.
        conc = np.float_(self.model.get_reactants())
        conc[:6] = conc[:6]*self.model.dvd
        conc[6:] = conc[6:]*self.model.rvd

        # Initialize outputs
        rates = np.zeros(len(self.model.reaction_map))
        sgn = np.zeros(len(self.model.reaction_map), dtype = int)

        # Could speed up with by only updating reactants
        # changed on the previous step if necessary
        for i in range(len(rates)):
            # The three reaction types are processed differently

            if self.model.reaction_map[i] == 'DM':
                # Uses external CO2/O2 concentration
                rates[i], sgn[i] = self.calc_membrane_diffusion_rate(
                    i, conc[i])
            elif self.model.reaction_map[i] == 'DI':
                # Diffusion between internal compartments
                # Magic numbers (2 and 4) relate the reactants
                # of species S in the two internal compartments.
                # Should pass in info as a data structure.
                rates[i], sgn[i] = self.calc_internal_diffusion_rate(
                    i-2, conc[i-2], conc[i+4])
            else:
                # Straight up reaction rate calculation.  Sign is always
                # positive as our reactions are one-way.
                rates[i], sgn[i] = self.calc_reaction_rate(i, conc), 1

        return rates, sgn

    def calc_membrane_diffusion_rate(self, i, c):
        """
        Calculates diffusion rate and direction through
        chloroplast membrane.

        Arguments
        ---------
```

```
        i : int [0 or 1]
            Index representing the diffusion reaction of interest.
        c : float
            Concentration of the i-th reactant.

        Returns
        -------
        (rate, sgn) : (float, int)
            Reaction rate and direction.

        Note
        ----

        Based on the diffusion rate equation

            r = K*sqrt(T/m)

        scaled by the difference in concentration across
        the membrane.  Because osmosis is hard and not the
        point of the model.

        """
        # Reaction coefficient
        K = self.params['K_m']
        # Temperature
        T = np.sqrt(self.params['temp'])
        # Concentrations of external gasses
        g_c = [self.params['co2_conc'], self.params['o2_conc']]

        # Compute reaction sign, rate, then return
        d = g_c[i] - c
        return K*T/reactant_masses[i]*np.abs(d), int(np.sign(d))



    def calc_internal_diffusion_rate(self, i, c1, c2):
        """
        Calculates diffusion rate and direction between
        our internal compartments in the chloroplast

        Arguments
        ---------
        i : int
            Index representing the diffusion reaction of interest.
        c1, c2 : floats
            Concentrations of the reactant in the two internal
            compartments in the chloroplast.
```

```
        Returns
        -------
        (rate, sgn) : (float, int)
            Reaction rate and direction.

        Note
        ----
        Based on the diffusion rate equation

            r = K*sqrt(T/m)

        scaled by the difference in concentration
        (ie a discretized gradient)

        """

        # Reaction coefficient
        K = self.params['K_r']
        # Temperature
        T = np.sqrt(self.params['temp'])

        # Compute reaction sign, rate, then return
        d = c1 - c2
        return K*T/reactant_masses[i]*np.abs(d), int(-np.sign(d))


    def calc_reaction_rate(self, i, c):
        """
        Calculates the light and dark reaction rates

        Arguments
        ---------
        i : int [8 or 9]
            Index representing the chemical reaction of interest.
        c: ndarray
            Vector of concentrations for all reactants

        Returns
        -------
        rate : float
            Reaction rate

        Note
        ----

        Based on the chemical reaction rate equation

            d[X]/dt = -k_0*a*[X]*[Y]*[Z] + k_1*d*[X]*[U]*[V]
```

```
        modelling the reactions

                        k_0
        aX + bY + cZ ------>  (Products)


                      k_1
        (Reactants) ------> dX + eU + fV

        where X,Y,Z,U,V are chemical species and
        a,b,c,d,e,f are numbers that balance the
        chemical equation stoichiometrically.

        """
        # Get the appropriate rate coefficient
        k = self.params['k_l'] if i == 8 else self.params['k_d']
        # Extract the concentrations of reactants
        # involved in the current reaction.
        r = c*self.model.reactions[:, i]
        r = r[r.nonzero()]
        # To preserve dimensionality, raise the rate
        # constant to the power of the length of the reactants.
        # Rate is then K[X][Y][Z] for K = k**3 in this example.
        # Scaled by the coefficients when reactants are updated.
        return k**len(r)*np.product(r)

    @staticmethod
    def calculate_update(rates):
        """
        Draw from the joint probability distribution
        P(tau, r) over update time-steps and reactions
        based on the current instantaneous reaction rates.

        Arguments
        ---------
        rates : ndarray
            Instantaneous reaction rates for all reactions.

        Returns
        -------
        (tau, r) : (float, float)
            tau - next time-step
            r - variable to draw next reaction.
        """
        # Rate at which any reaction will happen
        total_rate = np.sum(rates)
        # Assume reactions are a Poisson process
```

```
            tau = -1/total_rate*np.log(np.random.random())
            # Get a variable uniformly distributed in
            # (0, total rate)
            r = total_rate*np.random.random()

            return tau, r

        @staticmethod
        def get_reaction(rates, r):
            """
            Produces the next reaction to occur.

            Arguments
            ---------
            rates : ndarray
                Instantaneous reaction rates for all reactions.
            r : float
                Random variable uniformly drawn from the
                interval (0, sum(rates)==total_reaction_rate)

            Returns
            -------
            i : int
                The index of the next reaction
            """

            # Look in the interval (0, rate_1)
            # if r found, select reaction 1, else
            # look in (rate_1, rate_2), loop.
            current = 0
            for i in range(len(rates)):
                current += rates[i]
                if current > r:
                    break
            return i
```

In [31]:
```
######################
# Model Parameters #
######################

# Setup here as a bunch of sliders for interactivity.

# Simulate how many seconds?
max_time = ipw.IntSlider(min=     100,
                         max=     10000000,
                         step=    100,
                         value=   1000,
                         continuous_update=False)
```

```
# External gas Concentrations (Moles/Liter)
ext_o2_conc = ipw.FloatSlider(min=   0.00001,
                              max=   0.00100,
                              step=  0.00001,
                              value= 0.00030,
                              continuous_update=False)
ext_co2_conc = ipw.FloatSlider(min=   0.00001,
                               max=   0.00100,
                               step=  0.00001,
                               value= 0.00030,
                               continuous_update=False)
# System Temperature (Kelvin)
temperature = ipw.IntSlider(min=   273,
                            max=   320,
                            step=    1,
                            value= 305,
                            continuous_update=False)


# Diffusion Parameters

# Cell Membrane
memb_diff = ipw.FloatSlider(min=   1,
                            max=   1000,
                            step=  1,
                            value= 1,
                            continuous_update=False)

# Reaction Compartment
int_diff = ipw.FloatSlider(min=   1,
                           max=   20,
                           step=  1,
                           value= 1,
                           continuous_update=False)


# Reaction Rates
light_rate = ipw.FloatSlider(min=    1,
                             max=    500,
                             step=   1,
                             value=  1,
                             continuous_update=False)

dark_rate = ipw.FloatSlider(min=     1,
                            max=    500,
                            step=   1,
                            value=  1,
```

```
                                        continuous_update=False)
```

In [32]:
```
def plot_the_plots(T, R):
    """
    Plots reactant concentrations near the
    thylakoid membrane over time.

    Arguments
    ---------
    T : ndarray
        Time steps at which the reactants
        were evaluated.
    R : ndarray
        (number_of_reactants) x (len(T)) array
        of reactant numbers at the time steps
        in T.
    """
    # Using model data names
    reacs = ['CO2_r', 'O2_r', 'ATP_r', 'ADP_r',
             'NADPH_r', 'NADP_r', 'Glucose']
    # Plot readable names
    names = ['Carbon Dioxide',
             'Oxygen',
             'ATP',
             'ADP',
             'NADPH',
             'NADP',
             'Glucose']

    # Convert from numbers to concentrations.
    R = R/(reaction_volume*avogadro)

    fig, ax = plt.subplots(len(names), 1, figsize=(12, 35))

    for i in range(len(reacs)):

        ax[i].plot(T, R[:, reactant_map[reacs[i]]], lw=4)
        ax[i].set_xlabel('Time', fontsize=15)
        ax[i].set_ylabel('{} Concentration [M]'.format(names[i]),
                         fontsize=15)

    plt.show();
```

In [33]:
```
# Lists are 'passed by reference'
# so updates in inner scopes (a function)
# propagate to outer scopes.
# So we make a copy to preserve our initial
```

```
        # conditions, but allow ourselves to run
        # the simulation repeatedly and concatenate
        # the data, giving us the ability to push
        # towards extrema in the model by moving around
        # sliders.
        r = reactants.copy()
        # For storing the long term data.
        R = []
        T = []
        C = []
        CONC = []
```

In [34]:
```
        # Setup a simulation progress bar
        # so we don't get mad and think our
        # computer froze.
        progress_bar = ipw.FloatProgress(min=0, max=max_time.value)
        display(progress_bar)
        def update_callback(dt):
            progress_bar.value += dt

        reaction_name_map = {0:  "CO2 Membrane diffusion",
                             1:  "O2 Membrane diffusion",
                             2:  "CO2 internal diffusion",
                             3:  "O2 internal diffusion",
                             4:  "ATP diffusion",
                             5:  "ADP diffusion",
                             6:  "NADPH diffusion",
                             7:  "NADP diffusion",
                             8:  "Light reaction",
                             9:  "Dark reaction"}

        # Interactive widgets!  With all our model parameters!
        @ipw.interact(max_time=max_time,
                      ext_o2_conc=ext_o2_conc,
                      ext_co2_conc=ext_co2_conc,
                      temperature=temperature,
                      memb_diff=memb_diff,
                      int_diff=int_diff,
                      light_rate=light_rate,
                      dark_rate=dark_rate,
                      __manual=True) # Makes us have to press a button
                                     # to start the simulation.
        def f(max_time, ext_o2_conc, ext_co2_conc,
              temperature, memb_diff, int_diff,
              light_rate, dark_rate):

            """
            Wrapper function for interactivity.
```

```
    Arguments
    ---------
    Parameters of the simulation, see above.

    """
    # Reset the progress bar.
    progress_bar.min=0
    progress_bar.max=max_time
    progress_bar.value=0

    # Package up the parameters.
    params = {'time': max_time,
              'o2_conc': ext_o2_conc,
              'co2_conc': ext_co2_conc,
              'temp': temperature,
              'K_m': memb_diff,
              'K_r': int_diff,
              'k_l': light_rate,
              'k_d': dark_rate}

    # Initialize the model.
    m = Model(reactant_names,
              reactant_map,
              reactant_masses,
              r,
              reactions,
              state_change,
              reaction_map,
              reaction_volume,
              diffusion_volume)

    # Initialize the simulator
    s = Simulator(m, params, update_callback)

    # Get the current simulation results
    if T:
        T_run, R_run = s.run(T[-1])
    else:
        T_run, R_run = s.run(0)

    # Append them to our long term results.
    T.extend(T_run)
    R.extend(R_run)

    # Counts the number of each reaction to occur.
    c = Counter(m.log)
    # Hang on to the results
```

```
            C.append(c)

            # Print the individual reaction counts
            print("Reaction Counts")
            print("---------------")
            for k, v in c.items():
                print("{} count: {}".format(reaction_name_map[k], v))
            print("Total : {}".format(np.sum([v for k, v in c.items()])))
            print()

            plot_the_plots(np.array(T_run), np.array(R_run))
```

```
        # The idea here is to pick a set of parameters and run the
        # simulation til you get a steady state.  Something will
        # level off near zero when this happens.  Then turn on/off
        # another parameter and run it for a few times.
        # You can then go to the next cell and plot cumulative
        # data to see the impact of the change on photosynthesis.
        #
        # Example:
        #
        # Leave everything at default rates and turn the dark_rate
        # up to max.  Eventually the NADPH will flatline near zero
        # though it'll bounce around due to a small amount of light
        # reactions and diffusion.  Then, turn on the light reactions.
        # Suddenly your the reactants for the dark reaction will
        # start being produced again and the glucose production rate
        # will take off.
```

```
Reaction Counts
---------------
CO2 Membrane diffusion count: 40289
CO2 internal diffusion count: 634
ATP diffusion count: 8
ADP diffusion count: 1
NADPH diffusion count: 16
NADP diffusion count: 14
Total : 40962
```

In [36]:
```
# Plot the cumulative data

plot_the_plots(np.array(T), np.array(R))
```