The project discussed in this report involved the construction of randomized test matrices for detonation dilution studies. The tests to be performed will necessitate the variance of diluent gas, fuel and oxidizer equivalence ratio, and the level of dilution. Fuel and oxidizer species will be held constant throughout the tests, which will be blocked by diluent species for practical purposes. Multiple replicates of each test will be needed to ensure statistical viability, along with a randomized testing order within blocks. To enable this, a series of python scripts have been produced which generate multiple replicates of a randomized test matrix. In addition to the appropriate gas combinations the test matrix includes information to enable automated valve activation, to estimate the amount of each gas used, and to estimate the resulting detonation wave speed.

At each combination of factor levels, a list of pressures was calculated indicating the total pressure at which each gas valve should be shut off. This is accompanied by an estimation of the mass of each gas used per test and a predicted detonation velocity, which will be used for accounting purposes and sanity checking wave speed measurements, respectively. Detonation velocities were calculated at each set point using the Shock and Detonation Toolbox (SDToolbox) from Caltech's Explosion Dynamics Laboratory, and mass estimations were calculated using the geometry of the detonation tube along with Cantera.

The proposed grading rubric is as follows:

- Detonation speed calculation returns appropriate value: 10 points
- Equivalence ratio calculation returns appropriate value
  - for a single input condition: 5 points
  - for a list of input conditions: 5 points
- Dilution ratio calculation returns appropriate value
  - for a single input condition: 5 points
  - for a list of input conditions: 10 points
- Mass estimation returns appropriate value
  - for a single input condition: 5 points
  - for a list of input conditions: 5 points
- Test matrix is successfully output to .csv: 40 points
- Test matrix is randomized: 5 points

In order to assess the accuracy of velocity calculations, the Chapman-Jouguet (C-J) detonation velocities calculated in this project are compared to known values in literature. This comparison is shown graphically in Figure 1, demonstrating the validity of the C-J velocity calculations made using SDToolbox. Given that the calculated and expected values from the literature overlap, the calculated values are determined to be appropriate.

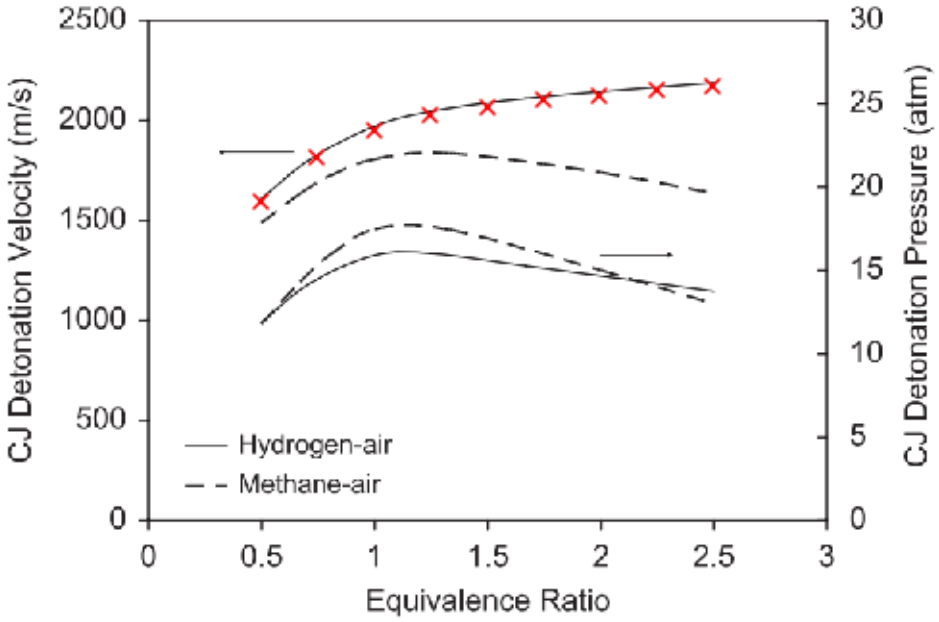

Figure 1: Calculated C-J speeds for H_2-Air (red) overlaid on a plot of C-J calculations performed by Ciccarelli and Dorofeev using STANJAN [1].

Gas masses, valve cutoff pressures, and mole fractions were verified by manually calculating them for two equivalence ratios. The results for undiluted and diluted mixtures are shown in Tables 1 and 2, respectively. Valve cutoff pressures were calculated by summing the partial pressures of each gas in the test chamber with a fill order of:

1. Fuel: $P_{cutoff} = P_f = P_{test} x_f$
2. Oxidizer: $P_{cutoff} = P_f = P_{test}(x_f + x_o)$
3. Diluent: $P_{cutoff} = P_f = P_{test}(x_f + x_o + x_d)$

where x indicates the mole fraction of each species and P_{test} indicates the initial chamber pressure of the test, which in this case is one atmosphere. Mole fractions of fuel and oxidizer were calculated from the equivalence ratio, ϕ using the relationship

$$\phi = \frac{n_f/n_o}{(n_f/n_o)_{stoich}}$$

where n_f and n_o are the number of moles of fuel and oxidizer, and the ratio $(n_f/n_o)_{stoich}$ is determined by the stoichiometric reaction $H_2 + \frac{1}{2} O_2 \longrightarrow H_2O$. For the diluted cases,

the equivalence ratio was maintained while the mole fractions (the number of moles of each component normalized by the total number of moles in the reaction) of fuel and oxidizer were reduced, with the remainder of the mixture being the desired amount of diluent gas. The masses of each species were calculated using the overall tube volume and the ideal gas law

$$P_i = \rho_i \frac{\bar{R}}{MW} T$$

where P_i and ρ_i represent the partial pressure and partial density of the ith species. These tables show no difference between the manually calculated values and the values generated by the python script.

	Undiluted												
	ϕ	1						0.75					
	Molecule	Moles	Mole Frac	F/A	P_i (kPa)	P_{cutoff} (kPa)	mass (kg)	Moles	Mole Frac	F/A	P_i (kPa)	P_{cutoff} (kPa)	mass (kg)
Hand Calc	H_2	1.00000	0.66667	2	67550	67550	0.00572	1.00000	0.6	1.5	60795	60795	0.00515
	O_2	0.50000	0.33333		33775	101325	0.04541	0.66667	0.4		40530	101325	0.05450
Cantera	H_2	-	0.66667	2	67550	67550	0.00572	-	0.6	1.5	60795	60795	0.00515
	O_2	-	0.33333		33775	101325	0.04541	-	0.4		40530	101325	0.05450

Table 1: Manually calculated vs. Cantera values with no dilution at a mass fraction of 20% for equivalence ratios of 1 and 0.75

	Diluted												
	ϕ	1						0.75					
	Molecule	Moles	Mole Frac	F/A	P_i (kPa)	P_{cutoff} (kPa)	mass (kg)	Moles	Mole Frac	F/A	P_i (kPa)	P_{cutoff} (kPa)	mass (kg)
Hand Calc	H_2	1.00000	0.5333	2	54040	54040	0.004578	1.00000	0.48	1.5	48636	48636	0.00412
	O_2	0.50000	0.2667		27020	81060	0.03633	0.66667	0.32		32424	81060	0.04360
	Ar	0.30000	0.2000		20265	101325	0.034017	0.33333	0.2		20265	101325	0.03402
Cantera	H_2	-	0.5333	2	54040	54040	0.004578	-	0.48	1.5	48636	48636	0.00412
	O_2	-	0.2667		27020	81060	0.03633	-	0.32		32424	81060	0.04360
	Ar	-	0.2000		20265	101325	0.034017	-	0.2		20265	101325	0.03402

Table 2: Manually calculated vs. Cantera values with Argon dilution at a mass fraction of 20% for equivalence ratios of 1 and 0.75

Finally, Table 3 shows the factor level combinations for each replicate of the test matrix and allows for visual verification of the randomization of the test order both within blocks and between replicates.

	Replicate 1		Replicate 2		Replicate 3		Replicate 4	
Diluent	ϕ	Diluent MF	ϕ	Diluent MF	ϕ	Diluent MF	ϕ	Diluent MF
None	0.75	0	1	0	1	0	1.25	0
	1	0	0.75	0	0.75	0	0.75	0
	1.25	0	1.25	0	1.25	0	1	0
Ar	1	0.3	0.75	0.2	1.25	0.3	1.25	0.2
	1	0.1	1.25	0.2	1	0.3	1	0.1
	1	0.2	1	0.1	1	0.2	1.25	0.1
	1.25	0.3	0.75	0.1	1.25	0.1	1.25	0.3
	0.75	0.2	1	0.3	1.25	0.2	1	0.2
	0.75	0.3	0.75	0.3	0.75	0.3	0.75	0.3
	1.25	0.1	1	0.2	1	0.1	0.75	0.2
	1.25	0.2	1.25	0.1	0.75	0.2	0.75	0.1
	0.75	0.1	1.25	0.3	0.75	0.1	1	0.3
N_2	1.25	0.2	0.75	0.1	1	0.1	1.25	0.3
	1.25	0.3	1.25	0.3	1	0.2	1.25	0.1
	1	0.2	1	0.3	1.25	0.1	1	0.2
	0.75	0.1	1.25	0.1	0.75	0.2	0.75	0.1
	1	0.3	0.75	0.2	0.75	0.3	0.75	0.2
	1	0.1	1.25	0.2	0.75	0.1	1.25	0.2
	0.75	0.3	0.75	0.3	1	0.3	1	0.3
	0.75	0.2	1	0.1	1.25	0.2	1	0.1
	1.25	0.1	1	0.2	1.25	0.3	0.75	0.3
CO_2	1	0.1	1.25	0.1	1	0.1	1.25	0.1
	1	0.2	1	0.1	1.25	0.1	1.25	0.3
	1	0.3	0.75	0.3	0.75	0.2	1	0.2
	1.25	0.2	1.25	0.2	1.25	0.2	1	0.1
	0.75	0.1	1	0.3	1.25	0.3	0.75	0.3
	0.75	0.2	0.75	0.1	1	0.3	0.75	0.1
	1.25	0.1	1.25	0.3	1	0.2	1.25	0.2
	0.75	0.3	0.75	0.2	0.75	0.3	0.75	0.2
	1.25	0.3	1	0.2	0.75	0.1	1	0.3

Table 3: Test conditions from each replicate, showing the randomization of each testing block.

Based on these results, the following grades are proposed:

- Detonation speed calculation returns appropriate value: 10/10
- Equivalence ratio calculation returns appropriate value
  - for a single input condition: 5/5
  - for a list of input conditions: 10/10
- Dilution ratio calculation returns appropriate value
  - for a single input condition: 5/5
  - for a list of input conditions: 10/10

- Mass estimation returns appropriate value
  - for a single input condition: 5/5
  - for a list of input conditions: 10/10
- Test matrix is successfully output to .csv: 40/40
- Test matrix is randomized: 5/5

The full generated test matrices are included with this report as .csv files, as well as `detonation.py` and `generate_test_matrix.py`. `detonation.py` defines the tools used to manipulate Cantera and SDToolbox, while `generate_test_matrix.py` uses these tools to build the randomized test matrices detailed in this report. If you wish to run the code yourself, please install or ensure you have the following packages:

- Cantera
- SDToolbox (http://shepherd.caltech.edu/EDL/public/cantera/html/SD_Toolbox/index.html)
- pandas
- random
- itertools
- os
- multiprocessing
- sys

These scripts run well on both my workstation and laptop, both of which are Windows based. If you have any issues running them please let me know and I will do my best to help. Thank you.

References

[1] G. Ciccarelli and S. Dorofeev, "Flame acceleration and transition to detonation in ducts," Progress in Energy and Combustion Science, vol. 34, no. 4, pp. 499–550, aug 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0360128507000639

detonation.py

```
#!/usr/bin/env python
"""
Created on Mon Feb 26 16:23:21 2018

@author: cartemic
"""

import SDToolbox as sd
import cantera as ct


class Velocity():
    '''
    A velocity object for easy unit envforcement to prevent the user from
    being too dumb. The only units here are m/s because I have no reason to
    use other units in this application. They can be added later if needed.
    '''
    units = set(['m/s'])

    def __init__(self, velocity, unit=None):
        # duck type it for ease of use
        try:
            self.value = velocity.value
            self.unit = velocity.unit
        except AttributeError:
            try:
                if unit is None:
                    unit = 'm/s'
                    print('No unit given. Assuming ' + unit)
                elif unit not in Velocity.units:
                    raise ValueError
                self.value = float(velocity)
                self.unit = unit
            except ValueError:
                print('Bogus velocity input')
                raise ValueError

    def __str__(self):
        return str(self.value) + ' ' + self.unit

    def __repr__(self):
        return str(self)


class Pressure():
    '''
    A pressure object for easy unit conversion to prevent the user from
    being too dumb. Units default to Pascals if not specified.
    '''
    units = set(['Pa', 'kPa', 'psia', 'atm'])

    def __init__(self, pressure, unit=None):
        # duck type it for ease of use
        try:
            self.value = pressure.value
            self.unit = pressure.unit
        except AttributeError:
            try:
```

```
                if unit is None:
                    unit = 'Pa'
                    print('No unit given. Assuming ' + unit)
                elif unit not in Pressure.units:
                    raise ValueError
                self.value = float(pressure)
                self.unit = unit
            except ValueError:
                print('Bogus pressure input')
                raise ValueError

    def to_Pa(self):
        '''
        Converts a pressure object to Pa
        '''
        if self.unit == 'Pa':
            return self
        elif self.unit == 'kPa':
            return Pressure(self.value * 1000., 'Pa')
        elif self.unit == 'psia':
            return Pressure(self.value / 0.00014503773800722, 'Pa')
        else:
            return Pressure(self.value * 101325, 'Pa')

    def __str__(self):
        return str(self.value) + ' ' + self.unit

    def __repr__(self):
        return str(self)

    def __mul__(self, other):
        return Pressure(other * self.value, self.unit)

    def __rmul__(self, other):
        return self * other


class Temperature():
    '''
    A temperature object for easy unit conversion to prevent the user from
    being too dumb. Units default to Kelvin if not specified.
    '''
    units = set(['K', 'C', 'F', 'R'])

    def __init__(self, temperature, unit=None):
        # duck type it for ease of use
        try:
            self.value = temperature.value
            self.unit = temperature.unit
        except AttributeError:
            try:
                if unit is None:
                    unit = 'K'
                    print('No unit given. Assuming ' + unit)
                elif unit not in Temperature.units:
                    raise ValueError
                self.value = float(temperature)
                self.unit = unit
            except ValueError:
                print('Bogus temperature input')
```

```
                raise ValueError

    def to_Kelvin(self):
        '''
        Converts a temperature object to Kelvin
        '''
        if self.unit == 'K':
            return self
        elif self.unit == 'C':
            return Temperature(self.value + 273.15, 'K')
        elif self.unit == 'F':
            return Temperature((self.value - 32.) * 5./9., 'C').to_Kelvin()
        else:
            return Temperature(self.value - 459.67, 'F').to_Kelvin()

    def __str__(self):
        return str(self.value) + ' ' + self.unit

    def __repr__(self):
        return str(self)

    def __mul__(self, other):
        return Temperature(other * self.value, self.unit)

    def __rmul__(self, other):
        return self * other


class Detonation():
    '''
    Docstring goes here
    '''
    # has a cj velocity and mass loss from tank estimate
    def __init__(self, init_pressure, init_temp, fuel, oxidizer, equivalence=1,
                 mechanism='gri30.cti'):
        # define chemical mechanism for cantera and sdtoolbox
        self.mechanism = mechanism

        # ensure good pressure input
        try:
            self.P = init_pressure.to_Pa()
        except AttributeError:
            print('Bad pressure given, assuming units are Pascals')
            self.P = Pressure(init_pressure, 'Pa')

        # ensure good temperature input
        try:
            self.T = init_temp.to_Kelvin()
        except AttributeError:
            print('Bad temperature given, assuming units are Kelvin')
            self.T = Temperature(init_temp, 'K')

        # initialize undiluted gas solution in Cantera
        self.undiluted = ct.Solution(mechanism)
        self.undiluted.TP = (self.T.value, self.P.value)

        # make sure the user input species that are in the mechanism file
        good_species = self.undiluted.species_names
        if fuel in good_species:
            self.fuel = fuel
```

```
        else:
            print('Bad␣Fuel')
            raise ValueError
        if oxidizer in good_species:
            self.oxidizer = oxidizer
        else:
            print('Bad␣Oxidizer')
            raise ValueError

        # set the default equivalence ratio
        self.set_equivalence(equivalence)

    def set_equivalence(self, equivalence_ratio):
        '''
        Sets the equivalence ratio of the undiluted mixture using Cantera
        '''
        equivalence_ratio = float(equivalence_ratio)

        # set the equivalence ratio
        self.undiluted.set_equivalence_ratio(equivalence_ratio,
                                             self.fuel,
                                             self.oxidizer)
        try:
            self.add_diluent(self.diluent, self.diluent_mol_frac)
        except AttributeError:
            pass

        # ensure good inputs were given and record new equivalence ratio
        if sum([self.undiluted.X > 0][0]) < 2:
            print('You␣can\t␣detonate␣that,␣ya␣dingus')
            raise ValueError
        self.phi = equivalence_ratio

    def get_mixture_string(self, diluted=False):
        '''
        Gets a mixture string from either the diluted or undiluted Cantera
        solution object, which is then used to calculate CJ velocity using
        SDToolbox
        '''
        if diluted:
            cantera_solution = self.diluted
        else:
            cantera_solution = self.undiluted
        mixture_list = []
        for i, species in enumerate(cantera_solution.species_names):
            if cantera_solution.X[i] > 0:
                mixture_list.append(species + ':' +
                                    str(cantera_solution.X[i]))
        return '␣'.join(mixture_list)

    def CJ_Velocity(self, diluted=False):
        '''
        Calculates CJ velocity using SDToolbox
        '''
        [cj_speed, _] = sd.CJspeed(self.P.value, self.T.value,
                                   self.get_mixture_string(diluted),
                                   self.mechanism, 0)
        return Velocity(cj_speed, 'm/s')

    def add_diluent(self, diluent, mole_fraction):
```

```
        '''
        Adds a diluent to an undiluted mixture, keeping the same equivalence
        ratio.
        '''
        # make sure diluent is available in mechanism and isn't the fuel or ox
        if diluent not in self.undiluted.species_names:
            print('Bad diluent:', diluent)
            raise ValueError
        elif diluent in [self.fuel, self.oxidizer]:
            print('You can\'t dilute with fuel or oxidizer!')
            raise ValueError
        elif mole_fraction > 1.:
            print('Bro, do you even mole fraction?')
            raise ValueError

        self.diluent = diluent
        self.diluent_mol_frac = mole_fraction

        # collect undiluted mole fractions
        mole_fractions = self.undiluted.mole_fraction_dict()

        # add diluent and adjust mole fractions so they sum to 1
        new_fuel = (1 - mole_fraction) * mole_fractions[self.fuel]
        new_ox = (1 - mole_fraction) * mole_fractions[self.oxidizer]
        x = '{0}: {1} {2}: {3} {4}: {5}'.format(diluent,
                                                mole_fraction,
                                                self.fuel,
                                                new_fuel,
                                                self.oxidizer,
                                                new_ox)
        try:
            # add to diluted cantera solution
            self.diluted.X = x
        except AttributeError:
            # create cantera solution if one doesn't exist
            self.diluted = ct.Solution(self.mechanism)
            self.diluted.TPX = (self.T.value, self.P.value, x)

    def __str__(self):
        return '''
This is my detonation.
There are many like it, but this one is mine.

My detonation is my best friend.
It is my life.
I must master it as I must master my life.

Without me my detonation is useless.
Without my detonation I am useless.
I must initiate my detonation true.
I must initiate better than my academic rivals who are trying to outpublish me.
I must collect data before he collects my data.
I will...

My detonation and I know that what counts in academia is not the detonations
we initiate, the data we collect, nor the analysis we conduct.
We know that it is the impact factors that count.
We will publish...

My detonation is human, even as I, because it is my life.
```

```
Thus, I will learn it as a brother.
I will learn its weaknesses, its strength, its parameters, its diluents,
its causes and its effects.
I will keep my detonation tube clean and ready.
We will become part of each other.
We will...

Before Joe Shepherd, I swear this creed.
My detonation and I are defenders of my degree.
We are the masters of our academic rivals.
We are the saviors of my degree.

So be it, until victory is Oregon State's and there is no enemy but data.
'''

    def __repr__(self):
        return str(self)

    def get_mass(self, tube_volume_m3, diluted=False):
        '''
        The cantera concentration function is used to collect
        species concentrations, in kmol/m^3, which are then multiplied by
        the molecular weights in kg/kmol to get the density in kg/m^3. This
        is then multiplied by the tube volume to get the total mass of each
        component.
        '''
        if diluted:
            cantera_solution = self.diluted
        else:
            cantera_solution = self.undiluted
        mixture_list = []
        for i, species in enumerate(cantera_solution.species_names):
            if cantera_solution.X[i] > 0:
                R = 8314 / cantera_solution.molecular_weights[i]
                P = self.P.value * cantera_solution.X[i]
                T = self.T.value
                rho = P/(R*T)
                mixture_list.append([species, rho * tube_volume_m3])

        return dict(mixture_list)

    def get_pressures(self, diluted=False):
        '''
        Cantera is used to get the mole fractions of each species, which are
        then multiplied by the initial pressure to get each partial pressure.
        '''
        if diluted:
            cantera_solution = self.diluted
        else:
            cantera_solution = self.undiluted
        mixture_list = []
        for i, species in enumerate(cantera_solution.species_names):
            if cantera_solution.X[i] > 0:
                mixture_list.append([species,
                                     self.P.value *
                                     cantera_solution.X[i]])
        return dict(mixture_list)


if __name__ == '__main__':
```

```
T = Temperature(20, 'C')
P = Pressure(1, 'atm')
test = Detonation(P, T, 'H2', 'O2')

# check CJ
from matplotlib import pyplot as plt
plt.close('all')
equivs = [i * 0.25 + 0.5 for i in xrange(9)]
cj_velocities = []
for phi in equivs:
    # stoich:
    # H2 + 0.5(O2 + 3.76N2) -> H2O + (3.76/2)N2
    # phi = (F/A)/(F/A)st = (A/F)st/(A/F)
    # A/F = (A/F)st / phi = (0.5/1) / phi
    mols_H2 = 1.
    mols_O2 = 0.5 / phi
    mols_N2 = 0.5 / phi * 3.76
    gases = 'H2: {0} O2: {1} N2: {2}'.format(mols_H2, mols_O2, mols_N2)
    [cj, _] = sd.CJspeed(P.to_Pa().value,
                         T.to_Kelvin().value,
                         gases,
                         'gri30.cti',
                         0)
    cj_velocities.append(cj)

# load in plot from cicarelli and dorofeev
cj_plot = plt.imread('cj_speeds_original.png')

# I don't want to have to re-use ginput every run, just copy and paste
# image coordinates here
image_coords = [(121.81653225806453, 420.03002016129028),
                (654.875, 30.42921370967747)]

# shift calculated values to where they should be on the imported plot
phi2 = [phi/3. * (image_coords[1][0]-image_coords[0][0]) +
        image_coords[0][0] for phi in equivs]
cj2 = [v/2500 * (image_coords[1][1]-image_coords[0][1]) +
       image_coords[0][1] for v in cj_velocities]

# overlay calculated cj velocities
plt.imshow(cj_plot)
plt.plot(phi2, cj2, 'rx')
plt.axis('off')
plt.savefig('cj_speeds.png')
```

generate_test_matrix.py

```
#!/usr/bin/env python
"""
Created on Mon Feb 26 16:23:21 2018

@author: cartemic
"""

import detonation as det
import pandas as pd
import random
import itertools
import os
import multiprocessing as mp
import sys

# %% define functions


def add_row(self, row):
    # add a row to a pandas dataframe, code from
    # https://stackoverflow.com/questions/10715965/
    # add-one-row-in-a-pandas-dataframe
    self.loc[len(self.index)] = row


pd.DataFrame.add_row = add_row


def first_replicate(initial_pressure,
                    initial_temperature,
                    fuel_string,
                    oxidizer_string,
                    equivalence_list,
                    current_diluent,
                    diluent_mf_list,
                    tube_volume_m3,
                    df,
                    call_number,
                    mp_out):

    # initialize a detonation
    my_detonation = det.Detonation(initial_pressure,
                                   initial_temperature,
                                   fuel_string,
                                   oxidizer_string)

    # no diluent means no diluent mass fraction
    if current_diluent == 'None':
        diluent_mf_list = [0]

    # build list of all test combinations
    test_conditions = list(itertools.product(*[equivalence_list,
                                               [current_diluent],
                                               diluent_mf_list]))

    for j in xrange(len(test_conditions)):
        # select a random test condition
        [equivalence, diluent, mf] = test_conditions.pop(
```

```
                random.randrange(len(test_conditions)))
        data = [diluent, equivalence, mf,
                [], [], [], [], [], [], []]

        # update detonation for each diluent, equivalence, and mf
        my_detonation.set_equivalence(equivalence)
        if current_diluent == 'None':
            # undiluted case
            diluted = False

        else:
            # diluted case
            my_detonation.add_diluent(diluent, mf)
            diluted = True

        # calculate partial pressures of each component. Filling
        # will be done fuel -> oxidizer -> diluent, so partials
        # will be added in this order in order to get cutoff
        # pressures for each component.
        pressures = my_detonation.get_pressures(diluted)
        masses = my_detonation.get_mass(tube_volume_m3, diluted)
        data[3] = pressures[fuel_string]
        data[4] = data[3] + pressures[oxidizer_string]
        try:
            data[5] = data[4] + pressures[current_diluent]
        except (KeyError):
            data[5] = 'N/A'

        # calculate cj
        data[6] = my_detonation.CJ_Velocity(diluted).value

        # calculate mass used
        data[7] = masses[fuel_string]
        data[8] = masses[oxidizer_string]
        try:
            data[9] = masses[current_diluent]
        except (KeyError):
            data[9] = 0.
        # put into sheet
        df.add_row(data)
    sys.stdout.flush()
    mp_out.put((call_number, df))


def User_input(user_query, desired_type):
    flag = True
    while flag:
        information = raw_input(user_query+' ').split()
        try:
            if desired_type is list:
                return information

            output = desired_type(*information)
            if isinstance(output, type(None)):
                raise ValueError
            else:
                return output
                flag = False
        except (TypeError, ValueError):
            print('Error: could not convert to ' + str(desired_type))
```

```
            print('Try␣again.\n')


def Output_results(dataframe_list):
    for i, df in enumerate(dataframe_list):
        file_name = 'Test_matrix_replicate_' + str(i+1) + '.csv'
        if os.path.exists(file_name):
            os.remove(file_name)
        df.to_csv(file_name, index=False)


def Generate(initial_pressure,
             initial_temperature,
             fuel_string,
             oxidizer_string,
             diluent_str_list,
             equivalence_list,
             diluent_mf_list,
             tube_volume_m3,
             replicates):

    initial_pressure = det.Pressure(initial_pressure)
    initial_temperature = det.Temperature(initial_temperature)

    mp_out = mp.Queue(replicates)

    # make sure there is an undiluted case
    if 'None' not in diluent_str_list:
        # Insert at beginning of list. Uses a string because a NoneType causes
        # problems with the dataframe reordering later in the script.
        diluent_str_list.insert(0, 'None')

    # list of colunm names for pandas dataframe
    column_titles = ['Diluent',
                     'Equivalence',
                     'Diluent␣MF',
                     'Fuel␣Pressure␣(Pa)',
                     'Oxidizer␣Pressure␣(Pa)',
                     'Diluent␣Pressure␣(Pa)',
                     'CJ␣Speed␣(m/s)',
                     'Fuel␣Used␣(kg)',
                     'Oxidizer␣Used␣(kg)',
                     'Diluent␣Used␣(kg)']

    # empty list for storing dataframes
    df_list = []
    for replicate in xrange(replicates):
#         print 'replicate', replicate
        df = pd.DataFrame(columns=column_titles)
        # initialize blank list of test conditions
        if replicate == 0:
            # block by diluent
            # http://sebastianraschka.com/Articles/2014_multiprocessing.html
            num_diluents = len(diluent_str_list)
            processes = []
            for n in xrange(num_diluents):
                print '␣␣␣␣Starting␣', diluent_str_list[n], 'on␣core', n
                processes.append(mp.Process(target=first_replicate,
                                            args=(initial_pressure,
                                                  initial_temperature,
```

```
                                                fuel_string,
                                                oxidizer_string,
                                                equivalence_list,
                                                diluent_str_list[n],
                                                diluent_mf_list,
                                                tube_volume_m3,
                                                df.copy(),
                                                n,
                                                mp_out)))
                processes[n].start()

            # end parallel processes
            print
            for n, p in enumerate(processes):
                p.join(None)
                print '␣␣␣␣Finished', diluent_str_list[n], 'matrix'

            # collect output
            diluent_blocks = sorted([mp_out.get() for p in processes])
            df = pd.concat([block[1] for block in diluent_blocks])

        else:
            # randomize first dataframe
            # loop through blocked diluents
            for i, diluent in enumerate(diluent_str_list):
                # separate out current diluent
                sub_df = df_list[0][df_list[0].Diluent == diluent]

                # randomize this block and add it to the current dataframe
                new_order = [int(idx) for idx in sub_df.index]
                random.shuffle(new_order)
                df = pd.concat([df, sub_df.reindex(new_order)])

        # append current dataframe to list
        df_list.append(df)
    return df_list


# %% main program


if __name__ == '__main__':
    P0 = det.Pressure(1, 'atm')
    T0 = det.Temperature(70, 'F')
    tube_volume_m3 = 0.1028
    fuel = 'H2'
    oxidizer = 'O2'
    diluents = ['AR', 'N2', 'CO2']
    equivalence = [0.75, 1, 1.25]
    diluent_mf = [0.1, 0.2, 0.3]
    replicates = 4
    test = Generate(P0,
                    T0,
                    fuel,
                    oxidizer,
                    diluents,
                    equivalence,
                    diluent_mf,
                    tube_volume_m3,
                    replicates)
```

```
Output_results(test)
```