Fiber Diffusion Operations Engine

Alexander J. Zolan, John J. Hasenbein, Jeremy J. Tejada
Graduate Program in Operations Research and Industrial Engineering
Department of Mechanical Engineering
The University of Texas at Austin
Austin, Texas

Abstract

In this report, we describe a series of differential equations that model the flow of debris through a recirculation pool. Some of the debris attaches to a series of strainers with known filtration rates and other portions are absorbed in a sump. We detail a Python implementation of this model that tracks the amount of debris present on the strainers and in the pool over time. Finally, we present a validation of the implementation using a simplified example with constant flow rates and filtration efficiency.

1 Introduction

In response to the Nuclear Regulatory Commission’s General Safety Issue 191 (GSI-191), the South Texas Project Nuclear Operating Company (STPNOC) wishes to perform an assessment of the estimated change in core damage frequency (ΔCDF) related to the accumulation of fiber resulting from a loss of coolant accident (LOCA) event. The Fiber Diffusion Operations Engine (FIDOE) models the movement of debris generated by a LOCA event through the emergency core cooling system (ECCS).

Section 2 describes the ECCS, the differential equations that govern the flow of debris through the ECCS, and the model assumptions. Section 3 describes the Python implementation and its library dependencies. Section 3.4 illustrates the validity of the routine via a simplified example. Finally, Section 4 details the source code.

2 System Description

An overview of the flow model is shown in Figure 1. The transportable debris from the hypothesized LOCA moves down into the containment emergency sump forming a pool of water. The initial concentration of debris in the pool is $C_p(0) = \frac{M_p(0)}{V_p}$. We assume all the debris is uniformly distributed in the pool at time...
0. As such, there is no debris on the fiber or the core at time 0 \((M_k^s(0) = 0 \ \forall k\) and \(M_c(0) = 0\). The rate of accumulation of the debris on the strainer and the core is governed by the flow rate, the amount of debris that passed through the strainers and the amount of debris deposited on the core. In the formulation below, the core acts as a sink. The governing conservation equations are:

\[
\frac{d}{dt} M_k^s(t) = Q_k^s(t) C_p(t) f(M_k^s(t)), \forall k \in A, B, C \tag{1a}
\]

\[
\frac{d}{dt} M_c(t) = Q_c(t) C_p \left[\sum_{k \in A,B,C} \left(1 - f(M_k^s(t)) (1 - \gamma^k)Q_k^s(t)\right)\right]
\frac{\sum_{k \in A,B,C} (1 - \gamma^k)Q_k^s(t)}{\sum_{k \in A,B,C} (1 - \gamma^k)Q_k^s(t)} \tag{1b}
\]

\[
0 = \frac{d}{dt} M_p(t) + \frac{d}{dt} \sum_k M_k^s(t) + \frac{d}{dt} M_c(t), \tag{1c}
\]

where \(k\) is the strainer index. Wherever \(k\) appears the index is taken over all the values in \{A, B, C\}, i.e., the three strainers. In the sequel, we carry computations under the following assumptions:

1. \(f(M_s)\) is a fraction between 0 and 1, dependent on the amount of mass on the strainer. \[2, \text{Figure 13}\].
2. \(Q_s(\cdot)\) should be treated generally as a function of time to model pumps turning on and off (discrete tabular function).
3. \(Q_c(\cdot)\) is a known function of time (discrete tabular function).
4. \(V_p\) is a given constant value.
5. \(\gamma^k\) is a known fraction of the flow rate \(Q_k^s(t)\) that recirculates directly back to the strainer.
6. The initial mass on the core is \(M_c(0) = 0\).
7. The initial mass in the pool, \(M_p(0)\), is given.

3 Implementation

FIDOE is a Python script developed at the University of Texas at Austin under STPNOC grant BO4425 and is implemented on OS X for production. Apple distributes OS X (Release 10.10) with Python; however, Python was updated to a later version (Version 3.4.2) to run FIDOE. The open-source PANDAS library (http://pandas.pydata.org/) is used in the FIDOE implementation.

Validation of the conservation equations took place over a two month period during which time period changes were made to the original proposed formulation. Verification and validation of the software was performed by the University of Texas at Austin and STP. Verification and validation was independently performed by STP GSI-191 oversight personnel. YK риск, LLC also performed validation.

The FIDOE source code is densely commented and the code is self-explanatory. In this section, the elements of the Python module are described in more detail.

3.1 Input/Output Format

The inputs are in the form of two flat files, which are read via the function \texttt{ReadParams}, using Python’s ‘pandas’ library (http://pandas.pydata.org/). The first flat file is indexed by time and takes on the following form as an example:

\[
t, Q_{s,a}, Q_{s,b}, Q_{s,c}, Q_c \\
0, 9600, 9600, 9600, 2000 \\
5, 0, 9600, 9600, 1500
\]
The header $t$ represents the time index for any time series of inputs (in minutes), while the headers $Q_{s.a}$, $Q_{s.b}$, $Q_{s.c}$, and $Q_c$ represent the flow rates through the three strainers and the core, respectively, noting that in most practical cases the flow rates through the strainers will be constant. It is assumed that the flow rates in gallons per minute (gpm) through the strainers are known as an explicit function of time. These flow rates include the ECCS and CSS flows through each strainer. In the example above, the flow rate through the first strainer (A) would be 9600 gpm over the first 5 minutes, and 0 thereafter.

The second flat file consists of inputs that are constant over time and takes on the following form as an example:

```
Initial Mass:
M_p,0,87000
M_s.a,0,0
M_s.a,0,0
M_s.c,0,0
M_c,0,0
```

This describes the initial mass (in grams) of debris in the pool, the three trains, and on the core, respectively, at the start of the simulation. Additional inputs specified in this input file include the pool volume, strainer recirculation rates, the function type used to describe the relationship between debris on a strainer and the filtration fraction of that strainer, and parameters associated with that function type.

The output consists of a single time series flat file and a plot of debris on the strainers and the core over time, up to a given threshold. The output flat file takes on the following form:

```
t,M_s.a,M_s.b,M_s.c,M_c
0,9600,9600,9600,2000
5,0,9600,9600,1500
```

### 3.2 Class MassCalculator

The FIDOE module contains a single class, `MassCalculator`. The following tasks are completed in `MassCalculator` on initialization, with parameters as given by the function `ReadParams` as input:

- Read or set default pool volume (gallons) and initial mass in pool (grams)
- Read or set default initial mass on strainers
- Read or set default initial mass on core
- Read or set default $\gamma$, the percentage of water flowing back to the strainers
- Read or set default strainer flow rates, in gpm
- Read or set default core flow rate, in gpm
- Read or set default filtration rate for any strainer (as a function of mass on the strainer)
- Alert the user of any default values that are used, due to a lack of specified inputs

Within this class, there are several accessors.
3.2.1 Strainer and core flow rate retrieval

Strainer flow rates are obtained from three functions, getFlowRateStrainerA, getFlowRateStrainerB, and getFlowRateStrainerC. The core flow rate is obtained using the procedure getFlowRateCore. All four procedures take the time period as a single input, and return the flow rate out of the three ECCS strainers, in gpm. Time-dependent flows are used as read from values stored in the class at initialization.

3.2.2 Filtration Function by Strainer

The function getFiltrationRate receives a single input, the mass on the strainer, and returns the fraction of debris that will attach to the strainer (rather than pass through) given that mass. We assume that this filtration rate includes any potential losses due to shedding, as that is embedded in the equations calculated by Ogden and Morton [2]. The filtration function given in Ogden and Morton is a function of the mass on a strainer module, and there are 20 modules on each strainer, so we divide the mass input by 20 to arrive at the filtration function.

3.2.3 Rates of changes of debris by location

The functions getDeltaMassStrainerA, getDeltaMassStrainerA, and getDeltaMassStrainerC take the time (in minutes) and an input vector of the masses in the strainers, core and containment pool as inputs, and returns the rate of advection through strainers A, B, and C, respectively, using Equation 1a. The function getDeltaMassCore takes the same set of inputs and returns the rate of advection through the core, as described in Equation 1b this calls on the function getNetPassThroughRate, which calculates the sum of $\chi_A$, $\chi_B$, and $\chi_C$ at the bottom of Figure 2. Finally, the function getDeltaMassPool takes the same set 

Figure 2: Diagram of paths of advection to the core for a three train plant (trains A, B, and C) in containment after ECCS recirculation.

of inputs and returns the rate of change of debris in the pool, as given in Equation 1c. These equations are aggregated through the function getAllDeltas.
3.2.4 ODE Solver

The function \texttt{SolveForCoreMass} takes the time period as an input, and numerically solves the system of equations as given in \texttt{getAllDeltas} by calling the ‘LSODA’ solver, which is the default solver in Python and part of the ODEPACK suite of differential equation solvers \cite{odepack}.

3.3 Interface

After calling the module through a Python interpreter, the following procedure takes place:

- The user is prompted to enter the name or filepath of the time-indexed and constant inputs file, the maximum timespan to solve (in minutes), and the desired name of the results output.
- The input files are read via the function \texttt{ReadParams}.
- The class \texttt{MassCalculator} is initialized. Any missing inputs to the class are noted in the console output as are the default values used in their place.
- For 1000 points between zero minutes and the timespan given as input, the system of ODE’s given in \texttt{SolveForCoreMass}.
- Plots of mass on each strainer and on the core are created, and a table of these values are saved under the filename given as input.

3.4 Validation using a Simplified Example

To validate the Fiber Diffusion Operations Engine (FIDOE) program, we consider a case where all flow rates are constant with respect to time, and the filtration function at each strainer is constant regardless of the amount of debris it holds. We let \( \hat{f}, Q^k_s \), and \( Q_c \) be the constant filtration function, strainer flow rates, and core flow rate, respectively. Then, we can solve for the rates of change as follows:

\[
\frac{d}{dt}M^k_s(t) = \frac{Q^k_s \hat{f}}{V_p} M_p(t), \quad \forall k \in \{A, B, C\} \quad (2a)
\]

\[
\frac{d}{dt}M_c(t) = \frac{Q_c}{V_p} \left[ \sum_{k \in A, B, C} \left( (1 - \hat{f})(1 - \gamma^k) Q^k_s \right) \right] M_p(t) = \frac{Q_c(1 - \hat{f})}{V_p} \quad (2b)
\]

\[
0 = \frac{d}{dt}M_p(t) + \frac{d}{dt} \left[ \sum_{k \in A, B, C} M^k_s(t) \right] + \frac{d}{dt}M_c(t)
\]

\[
\Rightarrow \frac{d}{dt}M_p(t) = - \frac{d}{dt} \left[ \sum_{k \in A, B, C} M^k_s(t) \right] - \frac{d}{dt}M_c(t)
\]

\[
= - \sum_{k \in A, B, C} \left[ \frac{Q^k_s \hat{f}}{V_p} \right] M_p(t) - \frac{Q_c(1 - \hat{f})}{V_p} M_p(t)
\]

\[
= \frac{\left( \sum_{k \in A, B, C} \left[ Q^k_s \hat{f} \right] + Q_c(1 - \hat{f}) \right)}{V_p} M_p(t). \quad (2c)
\]

We see that the rate of change of mass in the pool is a constant times the mass in the pool itself, yielding a first-order differential equation. Given the initial mass in the pool \( M_p(0) \), the mass in the pool at time \( t \) is \( M_p(t) = M_p(0)e^{-Rt} \), where the constant \( R \) is defined as

\[
R = \left( \hat{f} \sum_{k \in A, B, C} [Q^k_s] + Q_c(1 - \hat{f}) \right) \frac{1}{V_p}.
\]
Since the mass in the pool $M_p(t)$ is the only time-dependent variable in determining the rate of accumulation on the strainers and core, and their relative rates are known as per [2], we solve for $\frac{d}{dt}M_k(t)$ and $\frac{d}{dt}M_c(t)$ as follows:

$$\frac{d}{dt}M_k(t) = \frac{Q_k \hat{f}}{V_p R} M_p(0) (1 - e^{-Rt}), \quad \forall k$$

$$\frac{d}{dt}M_c(t) = \frac{Q_c (1 - \hat{f})}{V_p R} M_p(0) (1 - e^{-Rt}).$$

### 3.5 Results

We now compare the output of FIDOE with the analytical solution described in Section 3.4, using an example with three strainers labeled $A, B, C$. The inputs to the system are given in Table 1, while Table 2 compares the calculated mass in FIDOE to the analytical solution across several time horizons. We see that FIDOE’s calculations match the analytical solution to several significant digits, confirming that the implementation matches the equations in the case of constant filtration.

Table 1: Parameters of FIDOE example, assuming constant filtration and strainer flow rates

<table>
<thead>
<tr>
<th>System Inputs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_p(0)$</td>
<td>50000</td>
</tr>
<tr>
<td>$V_p$</td>
<td>500000</td>
</tr>
<tr>
<td>$\hat{f}$</td>
<td>0.7</td>
</tr>
<tr>
<td>$Q^A$</td>
<td>3000</td>
</tr>
<tr>
<td>$Q^B$</td>
<td>2000</td>
</tr>
<tr>
<td>$Q^C$</td>
<td>1000</td>
</tr>
<tr>
<td>$Q_c$</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 2: Comparison of FIDOE output to analytical solution

<table>
<thead>
<tr>
<th>t</th>
<th>$M_k(1)$</th>
<th>$M_k(2)$</th>
<th>$M_k(3)$</th>
<th>$M_c(1)$</th>
<th>$M_c(2)$</th>
<th>$M_c(3)$</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2013.03</td>
<td>1342.02</td>
<td>671.01</td>
<td>57.52</td>
<td>2013.03</td>
<td>1342.02</td>
<td>671.01</td>
</tr>
<tr>
<td>20</td>
<td>2013.03</td>
<td>1342.02</td>
<td>671.01</td>
<td>57.52</td>
<td>2013.03</td>
<td>1342.02</td>
<td>671.01</td>
</tr>
<tr>
<td>50</td>
<td>8549.95</td>
<td>5699.97</td>
<td>2849.98</td>
<td>244.28</td>
<td>8549.95</td>
<td>5699.97</td>
<td>2849.98</td>
</tr>
<tr>
<td>100</td>
<td>14134.06</td>
<td>9422.71</td>
<td>4711.35</td>
<td>403.83</td>
<td>14134.06</td>
<td>9422.71</td>
<td>4711.35</td>
</tr>
<tr>
<td>200</td>
<td>29163.10</td>
<td>13442.07</td>
<td>6721.03</td>
<td>576.09</td>
<td>29163.10</td>
<td>13442.07</td>
<td>6721.03</td>
</tr>
<tr>
<td>300</td>
<td>22734.85</td>
<td>15156.57</td>
<td>7578.28</td>
<td>649.57</td>
<td>22734.85</td>
<td>15156.57</td>
<td>7578.28</td>
</tr>
<tr>
<td>400</td>
<td>23831.86</td>
<td>15887.91</td>
<td>7943.95</td>
<td>680.91</td>
<td>23831.86</td>
<td>15887.91</td>
<td>7943.95</td>
</tr>
<tr>
<td>500</td>
<td>24299.80</td>
<td>16199.87</td>
<td>8099.93</td>
<td>694.28</td>
<td>24299.80</td>
<td>16199.87</td>
<td>8099.93</td>
</tr>
<tr>
<td>600</td>
<td>24499.41</td>
<td>16332.94</td>
<td>8166.47</td>
<td>699.98</td>
<td>24499.41</td>
<td>16332.94</td>
<td>8166.47</td>
</tr>
<tr>
<td>700</td>
<td>24584.55</td>
<td>16380.70</td>
<td>8194.85</td>
<td>702.42</td>
<td>24584.55</td>
<td>16380.70</td>
<td>8194.85</td>
</tr>
<tr>
<td>800</td>
<td>24620.87</td>
<td>16413.91</td>
<td>8206.96</td>
<td>703.45</td>
<td>24620.87</td>
<td>16413.91</td>
<td>8206.96</td>
</tr>
<tr>
<td>900</td>
<td>24636.36</td>
<td>16424.24</td>
<td>8212.12</td>
<td>703.90</td>
<td>24636.36</td>
<td>16424.24</td>
<td>8212.12</td>
</tr>
<tr>
<td>1000</td>
<td>24642.97</td>
<td>16428.65</td>
<td>8214.32</td>
<td>704.08</td>
<td>24642.97</td>
<td>16428.65</td>
<td>8214.32</td>
</tr>
</tbody>
</table>

### 4 Source Code

**Fiber Diffusion Operations Engine (FIDOE)**

System of Differential Equations Solver

Alex Zolan

Updated May 27, 2015

The purpose of the program is to simulate debris moving through a recirculating pool from which strainers can filter out some
debris, and some of the debris that passes through the strainers may attach itself to the core. A differential equations solver (LSODA) handles the numerical solution of the differential equations.

import time
import scipy
import scipy.integrate
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas
import csv

class MassCalculator(object):
    
    def __init__(self, params):
        # pool volume (gallons) and initial mass in pool (grams)
        if "M_p_0" in params.keys():
            self.M_p_0 = params["M_p_0"]
        else:
            self.M_p_0 = 3000.0
            print "M_p_0 not in inputs. Default value of 3000 used."
        if "V_p" in params.keys():
            self.V_p = params["V_p"]
        else:
            self.V_p = 50000.0
            print "V_p not in inputs. Default value of 50000 used."
        # Initial mass on strainers
        if "M_s_a_0" in params.keys():
            self.M_s_a_0 = params["M_s_a_0"]
        else:
            self.M_s_a_0 = 0.0
            print "M_s_a_0 not in inputs. Default value of 0 used."
        if "M_s_b_0" in params.keys():
            self.M_s_b_0 = params["M_s_b_0"]
        else:
            self.M_s_b_0 = 0.0
            print "M_s_b_0 not in inputs. Default value of 0 used."
        if "M_s_c_0" in params.keys():
            self.M_s_c_0 = params["M_s_c_0"]
        else:
            self.M_s_c_0 = 0.0
            print "M_s_c_0 not in inputs. Default value of 0 used."
        # Initial mass on core
        if "M_c_0" in params.keys():
            self.M_c_0 = params["M_c_0"]
        else:
            self.M_c_0 = 0.0
            print "M_c_0 not in inputs. Default value of 0.0 used."
        # gamma, the percentage of water flowing back to the strainers
        if "gamma_a" in params.keys():
            self.gamma_a = params["gamma_a"]
        else:
self.gamma_a = 0.0
print "gamma_a not in inputs. Default value of 0.0 used."
if "gamma_b" in params.keys():
    self.gamma_b = params["gamma_b"]
else:
    self.gamma_b = 0.0
print "gamma_b not in inputs. Default value of 0.0 used."
if "gamma_c" in params.keys():
    self.gamma_c = params["gamma_c"]
else:
    self.gamma_c = 0.0
print "gamma_c not in inputs. Default value of 0.0 used."

# strainer flow rates in gallons per minute (gpm)
if "Q_s_a" in params.keys():
    self.Q_s_a = params["Q_s_a"]
else:
    self.Q_s_a = 1000.0
    print "Q_s_a not in inputs. Default value of 1000.0 used."
if "Q_s_b" in params.keys():
    self.Q_s_b = params["Q_s_b"]
else:
    self.Q_s_b = 1000.0
    print "Q_s_b not in inputs. Default value of 1000.0 used."
if "Q_s_c" in params.keys():
    self.Q_s_c = params["Q_s_c"]
else:
    self.Q_s_c = 1000.0
    print "Q_s_c not in inputs. Default value of 1000.0 used."

# core flow rate in gpm
if "Q_c" in params.keys():
    self.Q_c = params["Q_c"]
else:
    self.Q_c = 1600.0
    print "Q_c not in inputs. Default value of 1600.0 used."

# filtration function type definition = hybrid or linear function
if "function_type" in params.keys():
    self.filtration_function = params["function_type"]
else:
    print "Filtration function type not specified. Default of hybrid equation used."
    self.filtration_function = "hybrid"

# for a linear function, read in the slope and intercept.
# if none provided, assume constant filtration factor of 0.75.
if self.filtration_function == "linear":
    if "slope" in params.keys():
        self.slope = params["slope"]
    else:
        self.slope = 0.0
        print "slope (filtration function) not in inputs. Default of 0.0 used."
    if "intercept" in params.keys():
        self.intercept = params["intercept"]
    else:
        self.intercept = 0.75
        print "intercept (filtration function) not in inputs. Default of 0.75 used."

self.m = "N/A"
sel.b = "N/A"
sel.threshold = "N/A"
sel.delta = "N/A"
sel.a = "N/A"
# We assume the hybrid function if there's anything else specified,
# whether it's "hybrid" or anything not "linear" or "hybrid".
else:
    if self.filtration_function != "hybrid":
        print "Function not specified as hybrid or linear. Default of hybrid used."
    #filtration rate (function of mass)b
    if "m" in params.keys(): self.m = params["m"]
    else:
        self.m = 0.007741 #lower envelope
        print "m (filtration function) not in inputs. Default of 0.007741 used."
    if "b" in params.keys(): self.b = params["b"]
    else:
        self.b = 0.6560 #lower envelope
        print "b (filtration function) not in inputs. Default of 0.6560 used."
    if "M_c" in params.keys(): self.threshold = params["M_c"]
    else:
        self.threshold = 38.5 #lower envelope
        print "M_c (filtration function) not in inputs. Default of 38.5 used."
    if "delta" in params.keys(): self.delta = params["delta"]
    else:
        self.delta = 0.02968 #lower envelope
        print "delta (filtration function) not in inputs. Default of 0.02968 used."
    if "a" in params.keys(): self.a = params["a"]
    else:
        self.a = 1.0 #lower envelope
        #this upper bound is not expected to be used in most cases, so it is not
        #called out in the console.,
        print "a (filtration function) not in inputs. Default of 1.0 used."
    self.slope = "N/A"
    self.intercept = "N/A"

def getFlowRateStrainerA(self,t):
    """returns the flow rate out of strainer A, in gallons per minute.
    This function is assumed to be known with respect to time,
    but currently has only a constant."""
    if type(self.Q_s_a) == float: return self.Q_s_a
    else:
        #if not a constant, use the flow rate just before the time
        #period that exceeds the input t. otherwise, use the
        #last flow rate given
        if self.Q_s_a["t"][0] > t: return 0
        for i in range(1,len(self.Q_s_a["t"])):  
            if self.Q_s_a["t"][i] > t:
                return self.Q_s_a["vals"][i-1]
        return self.Q_s_a["vals"][0]

def getFlowRateStrainerB(self,t):
    """returns the flow rate out of strainer B, in gpm.
    This function is assumed to be known with respect to time,
    but currently has only a constant."""
    if type(self.Q_s_b) == float: return self.Q_s_b
    else:
        #if not a constant, use the flow rate just before the time
        #period that exceeds the input t. otherwise, use
# the last flow rate given
if self.Q_s_b['t'][0] > t: return 0
for i in range(1,len(self.Q_s_b['t'])):
    if self.Q_s_b['t'][i] > t:
        return self.Q_s_b['vals'][i-1]
return self.Q_s_b['vals'][-1]

def getFlowRateStrainerC(self,t):
    """returns the flow rate out of strainer C, in gpm.
    This function is assumed to be known with respect to time,
    but currently has only a constant.""
    if type(self.Q_s_c) == float: return self.Q_s_c
    else:
        # if not a constant, use the flow rate just before the time
        # period that exceeds the input t. otherwise, use the
        # last flow rate given
        if self.Q_s_c['t'][0] > t: return 0
        for i in range(1,len(self.Q_s_c['t'])):
            if self.Q_s_c['t'][i] > t:
                return self.Q_s_c['vals'][i-1]
        return self.Q_s_c['vals'][-1]

def getFlowRateCore(self, t):
    """returns the flow rate through the core, in gallons per minute.
    This function is assumed to be known with respect to time."""
    if type(self.Q_c) == float: return self.Q_c
    else:
        # if not a constant, use the flow rate just before the time
        # period that exceeds the input t. otherwise, use the
        # last flow rate given
        if self.Q_c['t'][0] > t: return 0
        for i in range(1,len(self.Q_c['t'])):
            if self.Q_c['t'][i] > t:
                return self.Q_c['vals'][i-1]
        return self.Q_c['vals'][-1]

def getFiltrationRate(self,mass):
    """returns the filtration rate (fraction between 0 and 1)
    of debris through the strainer. (Note the mass is total for
    a strainer, and there are 20 modules, with the filtration
    function relating to the per module mass - so we divide
    by 20 to get the per-module mass.
    mass -- amount of debris currently on the strainer (grams)
    retval - fraction between 0 and 1 indicating how the
    proportion of mass that is caught and added to the strainer
    """
    if self.filtration_function == "hybrid":
        if (mass/20.0) <= self.threshold:
            return (mass/20.0)*self.m + self.b
        else:
            return (self.threshold*self.m + self.b) + (self.a -
                self.threshold*self.m - self.b) * (1-scipy.exp( -self.delta * ((mass/20.0)-self.threshold) ) )
    else:
return max(0.0,min(1.0,(mass/20.0)*self.slope + self.intercept))

def getDeltaMassStrainerA(self, masses, t):
    """Calculates the rate of change of mass on strainer A.
    masses -- mass of debris in the different parts of the
    recirculation system:
    masses[0] = Pool (M_p)
    masses[1] = Strainer A (M_s_A)
    masses[2] = Strainer B (M_s_B)
    masses[3] = Strainer C (M_s_C)
    masses[4] = Core (M_c)
    t -- time
    retval -- rate of change of mass on Strainer A."""
    return (self.getFlowRateStrainerA(t) * (masses[0] / self.V_p) *
            self.getFiltrationRate(masses[1]) )

def getDeltaMassStrainerB(self, masses, t):
    """Calculates the rate of change of mass on strainer B.
    masses -- mass of debris in the different parts of the
    recirculation system:
    masses[0] = Pool (M_p)
    masses[1] = Strainer A (M_s_A)
    masses[2] = Strainer B (M_s_B)
    masses[3] = Strainer C (M_s_C)
    masses[4] = Core (M_c)
    t -- time
    retval -- rate of change of mass on Strainer B."""
    return (self.getFlowRateStrainerB(t) * (masses[0] / self.V_p) *
            self.getFiltrationRate(masses[2]) )

def getDeltaMassStrainerC(self, masses, t):
    """Calculates the rate of change of mass on strainer C.
    masses -- mass of debris in the different parts of the
    recirculation system:
    masses[0] = Pool (M_p)
    masses[1] = Strainer A (M_s_A)
    masses[2] = Strainer B (M_s_B)
    masses[3] = Strainer C (M_s_C)
    masses[4] = Core (M_c)
    t -- time
    retval -- rate of change of mass on Strainer C."""
    return (self.getFlowRateStrainerC(t) * (masses[0] / self.V_p) *
            self.getFiltrationRate(masses[3]) )

def getNetPassThroughRate(self,masses,t):
    """Calculates the weighted average pass-through rate of debris
    through the strainers and to the core.
    result is weighted by flow rate to the core
    (given by the gamma term and flow rate).
    masses -- mass of debris in the different parts of the system:
    masses[0] = Pool (M_p)
    masses[1] = Strainer A (M_s_A)
    masses[2] = Strainer B (M_s_B)
masses[3] = Strainer C (M_s_C)
masses[4] = Core (M_c)
t -- time

tval -- weighted average of debris filtered by the strainers"
if (self.getFlowRateStrainerA(t) + self.getFlowRateStrainerB(t) + self.getFlowRateStrainerC(t) == 0): return 1.0
else: return ( self.getFlowRateStrainerA(t) * (1-self.getFiltrationRate(masses[1])) * (1-self.gamma_a) + self.getFlowRateStrainerB(t) * (1-self.getFiltrationRate(masses[2])) * (1-self.gamma_b) + self.getFlowRateStrainerC(t) * (1-self.getFiltrationRate(masses[3])) * (1-self.gamma_c) ) / ( self.getFlowRateStrainerA(t) * (1-self.gamma_a) + self.getFlowRateStrainerB(t) * (1-self.gamma_b) + self.getFlowRateStrainerC(t) * (1-self.gamma_c) )

def getDeltaMassCore(self, masses, t):
    """Calculates the rate of change of debris on the core.""
    return (self.getFlowRateCore(t) * (masses[0] / self.V_p) * (self.getNetPassThroughRate(masses,t)) )

def getDeltaMassPool(self,masses,t):
    """Calculates the rate of change of debris in the pool.""
    return -1.0*( self.getDeltaMassCore(masses,t) + self.getDeltaMassStrainerA(masses, t) + self.getDeltaMassStrainerB(masses, t) + self.getDeltaMassStrainerC(masses, t) )

def getAllDeltas(self, masses, t):
    """Gets the rate of change of debris in all locations.""

def solveForCoreMass(self, t):
    """Runs the ODE integrator from Python's ODE library, with the delta functions and initial values arranged in order: pool, strainer A, B, C, and Core. Note: We use the library's default solver, LSODA, for this set of differential equations.""
    return scipy.integrate.odeint(self.getAllDeltas, scipy.array([self.M_p_0, self.M_s_a_0, self.M_s_b_0, self.M_s_c_0, self.M_c_0]), t, mxstep=10000000 )

def printEchoIn(self,filename = "echoin.csv"):
    """Prints all model parameters to file. Used for debugging"""
and I/O checking.

outfile = open(filename,'w')
outfile.write("Model parameters used:\n\n")
outfile.write("Filtration Function Type: %s\n" % self.filtration_function)
outfile.write("Filtration Function Parameter Values:\n")
outfile.write("m,%s\n" % self.m)
outfile.write("b,%s\n" % self.b)
outfile.write("M_c,%s\n" % self.threshold)
outfile.write("delta,%s\n" % self.delta)
outfile.write("a,%s\n" % self.a)
outfile.write("slope,%s\n" % self.slope)
outfile.write("intercept,%s\n" % self.intercept)
outfile.write("Initial Masses and Strainer Values:\n")
outfile.write("M_p_0,%s\n" % self.M_p_0)
outfile.write("V_p,%s\n" % self.V_p)
outfile.write("M_s_a_0,%s\n" % self.M_s_a_0)
outfile.write("M_s_b_0,%s\n" % self.M_s_b_0)
outfile.write("M_s_c_0,%s\n" % self.M_s_c_0)
outfile.write("gamma_a,%s\n" % self.gamma_a)
outfile.write("gamma_b,%s\n" % self.gamma_b)
outfile.write("gamma_c,%s\n" % self.gamma_c)
outfile.write("Flow Rates over time:\n")
if type(self.Q_s_a) == float: outfile.write("Q_s_a,%s\n" % self.Q_s_a)
else:
    outfile.write("t,Q_s_a\n")
    for idx in range(len(self.Q_s_a["t"])):
        outfile.write("%s,%s\n" % (self.Q_s_a["t"],self.Q_s_a["vals"])[idx])
    outfile.write("\n")
if type(self.Q_s_b) == float: outfile.write("Q_s_b,%s\n" % self.Q_s_b)
else:
    outfile.write("t,Q_s_b\n")
    for idx in range(len(self.Q_s_b["t"])):
        outfile.write("%s,%s\n" % (self.Q_s_b["t"],self.Q_s_b["vals"])[idx])
    outfile.write("\n")
if type(self.Q_s_c) == float: outfile.write("Q_s_c,%s\n" % self.Q_s_c)
else:
    outfile.write("t,Q_s_c\n")
    for idx in range(len(self.Q_s_c["t"])):
        outfile.write("%s,%s\n" % (self.Q_s_c["t"],self.Q_s_c["vals"])[idx])
    outfile.write("\n")
if type(self.Q_c) == float: outfile.write("Q_c,%s\n" % self.Q_c)
else:
    outfile.write("t,Q_c\n")
    for idx in range(len(self.Q_c["t"])):
        outfile.write("%s,%s\n" % (self.Q_c["t"],self.Q_c["vals"])[idx])
    outfile.write("\n")

def ReadParams(time_filename, initials_filename):
    """Serves as the input reader for this model. Assumes there is one file that reads as a table of time-based inputs and another file with initial and model values. The output is a dictionary that is used to initialize the MassCalculator class."
    """
    params = {"}
# read in initials and constants file
initials_file = csv.reader(open(initials_filename, 'rU'))
for line in initials_file:
    if len(line) > 1:
        try: params[line[0]] = float(line[1])
        except ValueError: params[line[0]] = line[1]
# read in time-based inputs file
time_df = pandas.read_csv(time_filename)
time_df = time_df[time_df.t == time_df.t].sort(['t']) # removes nan
# print time_df
params["Q_s_a"] = {}
params["Q_s_a"]['t'] = time_df.t.values
params["Q_s_a"]['vals'] = time_df.Q_s_a.values
params["Q_s_b"] = {}
params["Q_s_b"]['t'] = time_df.t.values
params["Q_s_b"]['vals'] = time_df.Q_s_b.values
params["Q_s_c"] = {}
params["Q_s_c"]['t'] = time_df.t.values
params["Q_s_c"]['vals'] = time_df.Q_s_c.values
params["Q_c"] = {}
params["Q_c"]['t'] = time_df.t.values
params["Q_c"]['vals'] = time_df.Q_c.values
print params["Q_c"]['t']
return params

if __name__ == '__main__':
time_filename = raw_input("Please enter the name of the time-indexed inputs file: ")
initials_filename = raw_input("Please enter the name of the constant inputs file: ")
timespan = float(raw_input("Please enter the desired timespan (minutes): "))
outfile = raw_input("Please enter the results filename (no extension): ")
""
# create_png = raw_input("Create graph summary of output (y/n)? ")
solver = MassCalculator(ReadParams(time_filename, initials_filename))
clock = time.time()
t = scipy.linspace(0,timespan,1001)
sol = solver.solveForCoreMass(t).T
elapsed = time.time() - clock
print "Calculations completed in "+str(elapsed)+" seconds. Creating output files."
# Creating csv table
output = open(outfile+".csv","w")
output.write("t,M_p,M_s_a,M_s_b,M_s_c,M_c\n")
for idx in range(len(sol[0])):
    output.write(str(t[idx])+","+str(sol[0][idx])+","+str(sol[1][idx])+","+str(sol[2][idx])+","+str(sol[3][idx])+","+str(sol[4][idx])+"\n")
output.close()
# Creating 2x2 figure of plots of debris levels over time.
# If plotting can't be done here, skip this step.
try:
    fig, axes = plt.subplots(2,2)
axes[0, 0].plot(t, sol[1])
axes[0, 0].set_title('Debris on strainer A over time')
axes[0, 1].plot(t, sol[2])
axes[0, 1].set_title('Debris on strainer B over time')
axes[1, 0].plot(t, sol[3])
axes[1, 0].set_title('Debris on strainer C over time')
axes[1, 1].plot(t, sol[4])
axes[1, 1].set_title('Debris on core over time')
plt.savefig(outfile + "_.png")
except TypeError: pass

# print model parameters
solver.printEchoIn()