# Oil Weathering

# Technical Documentation

# and

# Recommended Use Strategies

by

J A Galt

# Table of Contents


1Introduction....4
2Public Interface....5
2.1OilWx -creator: ....5
2.2Optional Customization Methods ....5
2.2.1MyOilObject.AddDistillationCut:....5
2.2.2MyOilObject.SetMaximumArea:....6
2.2.3MyOilObject.SetMousseOnsetPoint: ....6
2.2.4MyOilObject.SetMousseOnsetTime: ....6
2.2.5MyOilObject.SetWindRowSpacing: ....7
2.2.6MyOilObject.SetMinThickness:....7
2.3Final Initialization and Set of Model Environmental Variables ....7
2.4Time Integration of OilWx Object:....8
2.5State Variable Getters: ....9
3Spreading and Weathering Algorithms....10
3.1Creator....10
3.2Add Distillation Data ....11
3.3Set Maximum Area ....11
3.4Set Mousse Onset Point ....11
3.5Set Mousse Onset Time ....12
3.6Set Wind Row Spacing ....12
3.7Set Minimum Thickness ....12
3.8Set Environment ....12
3.9Step ....15
3.10Step Clean ....16
3.11SetDefaultPC....16
3.12SetUserPseudoComponents ....17
3.13Get Density ....18
3.14Get Thickness....19
3.15Get Product Thickness ....19
3.16Get Environmental Thickness....19
3.17Get C1 ....19
3.18Get C2 ....20
3.19Get Wind Row Spacing....20
3.20CalculateC2....20
3.21Vapor Pressure ....22
3.22Step Evaporation....22
3.23Step  Natural Dispersion ....23
3.24Step Water Content....23
3.25Calculate Viscosity ....24
3.26Calculate Area....24
3.27Area Modification....25
3.28Calculate Volume ....25

4References........26
5Appendices........27
5.1Appendix 1 – OilWx objects state variables........27
5.2Appendix 2 – Test Program Code........29
5.3Appendix 3 – Test Program Console (Spreadsheet) Output........33
5.4Appendix 4 – Graphic output from spreadsheet........34
5.5Appendix 5 – OilWx code........36

# 1 Introduction

This note will provide technical documentation and describe the intended use strategy for the OilWx routines. Formally, OilWx is a computer object which can be attached as an attribute to any identified host object. As such, it then provides oil characteristics, spreading, and weathering estimates for the host object. In modern computer applications based on objected oriented design an "object" is a referenced data type and is described in terms of its API (Application Programming Interface). This API is the public interface that the "object-OilWx" presents to the host (or client) program and specifies all of the ways that the host or client can: a) define the "object-OilWx" , b) pass information (client input) to the "object-OilWx" , or c) or receive state information (output from) the "object-OilWx" . The first part of this report will describe the OilWx's API. In addition to the API that specifies how the OilWx object interacts with its exterior programming environment, the object also contains an interior implementation part which contains the algorithms that change the state of the object. These simulate the physical processes associated with the spreading and weathering of a contiguous patch of floating oil and are not accessible to the host or client programming environment. The actual implementation of OilWx is broken into many subcomponents or code fragments that can be easily updated based on improved data or new insights and this refinement process does not change the existing API or require modification of the host program. The second part of this report describes the implementation algorithms of OilWx.

The initial use of the OilWx routines was intended to provide oil characteristics to a model of a contiguous patch of spilled oil (Response Options Calculator or ROC). Computational components of this host model then accounted for implementation of various hypothetical clean up methodologies to estimate efficacy of alternate response strategies.

An alternate use might call for an OilWx object to be attached to some fraction (or all) of the set of the Lagrangian particles in an oil trajectory model. This would provide a simple particle tracking model with advanced oil weathering and film thickness support.

The OilWx routines were initially written in JAVA, but could be easily translated to any modern computer language that is object oriented. It has already been ported to "Action Script" for use in ROC. It could also be quickly converted to (C++) or Python. The actual class definition is fast and compact requiring only about 1000 lines of code.

There are five general types of methods defined as part of the OilWx's API..

1. The first is the object creator that will initialize the object with a few oil property descriptors.
2. The second group of methods are optional and allow the user to customize the oil properties and specify some constraints that may apply during the weathering process.
3. The third type of object method is used to set environmental conditions that apply to the host object. This method must be called at least once to complete the OilWx's initialization, but it can then be repeatedly called if the host's environment changes (for example variable wind speed).
4. The forth group of methods are called to step the oil weathering processes in time, and there

are two options available (one that steps the time and a second alternate option that steps the time as well as explicitly removing a specified mass of oil representing clean up or removal activity associated with the spill response or some process that is not otherwise represented).

5. The final group of methods are so called "getters" that can be called on the OilWx object at any time to return the state variables describing the oil. (OilWx output for use in the host program.)

The following section of this note will describe the API (Application Programming Interface) components for each of the 5 groups of methods defined above. A final section will describe the actual oil weathering algorithms (implementation) that are used in the model stepping procedure.

# 2 Public Interface

The public interface to the OilWx object has a single creator that can be called by the host program to place an OilWx object into memory. It can then be manipulated by a number of methods that can be invoked using a reference to the OilWx object. This is a standard approach used in most object oriented languages.

## *2.1 OilWx -creator:*

The object creator is called using the "new" command and a short list of oil properties as in the following example.

**OilWx MyOilObject = new OilWx(api,vis,refT,vol)**

In this example the program returns an OilWx object called "MyOilObject" that has been initialized with an:

- an initial density specified as API Gravity
- an initial viscosity of vis in units of centistokes
- a reference temperature of refT (degrees C) where the viscosity is measured
- an initial volume in cubic meters

This newly created OilWx object is now available for a number of optional customization methods that can be used to go beyond the defaults that are built into the basic objects behavior.

## *2.2 Optional Customization Methods*

### 2.2.1 MyOilObject.AddDistillationCut:

The OilWx object represents the oil's many components in terms of a distillation curve that plots fraction percent of oil vs. boiling point. If this method is not called, the OilWx object will generate a statistically derived distillation curve approximating the continuous curve with ten pseudo components.

If on the other hand the user has oil specific distillation data, they can use this method to attach a custom distillation curve to the object by making the following call.

**MyOilObject.AddDistillationCut(cumulativePercent,boilingPoint)**

There are some restrictions associated with this method:

- this method must be call successively for each pseudo component
- a minimum of three components must be specified
- for each successive call both the cumulativePercent and boilingPoint must increase

If any of these three conditions is violated the final initialization routine (see SetEnvironment below) will return an error code. The host program should check for these error codes to prevent the OilWx model from running with an uncertain or erroneous data set.

### 2.2.2 MyOilObject.SetMaximumArea:

This method may be called if the user wishes to constrain the spreading routines in the OilWx object to a fixed maximum size. This would be used if the spilled volume was contained by ice, a berm, or a boomed region. The call to this method is:

**MyOilObject.SetMaximumArea(area)**

Where the area is entered as a method parameter in square meters.

### 2.2.3 MyOilObject.SetMousseOnsetPoint:

The algorithms that control the formation of water in oil emulsion (Mousse) typically will not begin until an initial fraction of the oil has evaporated. The default value for this coefficient is calculated by the model based on the initial oil parameterization. If on the other hand the user has additional data that characterizes the specific oil, they may override the default value for this oil emulsion onset value. The call to this method is:

**MyOilObject.SetMousseOnsetPoint(evaporatedFraction)**

Where evaporatedFraction is a fractional value in percent between 0 and 100.

### 2.2.4 MyOilObject.SetMousseOnsetTime:

Often during an oil spill the onset of oil in water emulsion formation is an observable. If at some time this is noticed the observer can modify the default behavior of the MyOilObject's behavior by setting this value. The call to this method is:

**MyOilObject.SetMousseOnsetTime(emulsionOnsetTime)**

Where emulsionOnsetTime is given in seconds. If this value is set it will override the fractional value (either the default or user set) that was described in the previous paragraph.

### 2.2.5 MyOilObject.SetWindRowSpacing:

The OilWx object has a sub model that calculates the relative oil film thickness caused by the formation of Langmuir cells or "wind rows". The default values used in this sub model is typically calculated from oil and environment parameters. It is also possible to observe the dominant wind row separation or spacing during an overflight. This method allows the user to override the internal computations by setting the number to the observed value. The call to this method is:

**MyOilObject.SetWindRowSpacing(distanceBetweenWindRows)**

Where distanceBetweenWindRows is given in meters.

### 2.2.6 MyOilObject.SetMinThickness:

Within the internal variables carried by the MyOilObject is a boolean flag that keeps track of whether the oil has spread to the point where the thickness is less than a user specified value. This gives the user a way to specify a limit as to when the oil can be ignored, recovery operations can be terminated, or computations or oil properties are no longer of interest. The user specified value can be set by a call to the method:

**MyOilObject.SetMinThickness(responseThicknesLimit)**

Where responseThicknessLimit is given in meters. This value does not affect any of the OilWx objects computations, but is available to the host object (see MyOilObject.is MinThickness below) so that it can take whatever action is appropriate.

## *2.3 Final Initialization and Set of Model Environmental Variables*

After all of the optional methods that are used to customize MyOilObject have been called if desired the final part of the object initialization is accomplished by a call to:

**MyOilObject.SetEnvironment(windSpeed,waterTemp,mixedLayerDepth)**

Where the input parameters are:

- The wind speed, windSpeed, is given in meters/second
- The water temperature, waterTemp, is given in degrees C
- The mixed layer depth, mixedLayerDepth, is given in meters.

This method must be called at least once as the last step in the objects initialization. After this initialization it can optionally be called any number of times to re-specify any or all of the environmental input parameters. If for example, MyOilObject was used in a trajectory model where the wind changed over time, repeated calls to MyOilObject.SetEnvironment could be used to define any step-wise variations.

The MyOilObject.SetEnvironment method will return an error code that provides some indication of whether the final initialization has obvious flaws that would make the spreading and oil weathering estimates suspect. The stand return codes are:

- positive integer – is the normal return and indicates the number of pseudo components used.

- -1 error – mass fraction values out of order in the pseudo components
- -2 error – indicates boiling point values out of order in the pseudo components
- -3 error – indicates mass fraction data < 0% in the pseudo components
- -4 error – indicates mass fraction data > 100% in the pseudo components
- -5 error – minimum boiling point < 0 deg C in the pseudo components
- -6 error – maximum boiling point > 1000 deg C in the pseudo components
- -7 error – not enough pseudo components < minNumUserPC (default 3)
- -8 error – model results suspect – oil may sink

After the initialization is complete the host program should check the return code from this method and for any negative values the computations should be terminated and input data checked.

## 2.4 Time Integration of OilWx Object:

After MyOilObject has been initialed it can be integrated over time to give an estimate of the various state variables that describe the weathering and spreading characteristics of the oil object that it represents. There are two public options for stepping forward in model time. The first of these is:

**MyOilObject.Step(stepLength)**

Where stepLength is the time in seconds over which the object is integrated. For example:

$$MyOilObject.Step(3600) = \int_{t_0}^{3600} MyOilObject\,dt$$

And this would advance all the weathering and spreading processes by one hour and update all the state variables accordingly.

The second public option for stepping the age of the object is like the first except that it allows to user to specify the removal of some of the oil volume due to some mitigation activity such as chemical dispersion, mechanical recovery or burning. This method would be called by:

**MyOilObject.Step(stepLength,volumeRecovered,oilOnlyFlag)**

where:

- stepLength is the integration time in seconds
- volumeRecovered is the amount of oil (in cubic meters) removed by the start of the time interval (since the last call to this method)
- oilOnlyFlag is a boolean flag which is set to true if the recovered volume is all oil and false if the volume is of an oil/mousse mixture.

This completes the description of all of the public methods that can be used to initialize, customize and time step (integrate) an OilWxObject. There remain a number of public methods that can report on the state variables that describe the OiWxObj at any moment in time.

### 2.5 State Variable Getters:

At any time during the life span of an OilWxObject a number of public methods are available to examine the state variables. These methods are all referred to as “getters” and they are “read only” in that when they are called by the host program they simple report back on a state variable, but never interact with or change the OilWxObject itself. The available “getters” are:

**MyOilObject.getDensity()** - returns oil density of oil/mousse in kg/m3

**MyOilObject.getViscosity()** - returns viscosity of oil/mousse in centistokes (cS)

**MyOilObject.getArea()** - returns area of oil slick in square meters

**MyOilObject.getSlickWidth()** - returns width of slick perpendicular to wind in meters

**MyOilObject.getThickness()** - returns thickness of oil without mousse in meters

**MyOilObject.getProdThickness()** - returns thickness of oil/mousse mixture in meters

**MyOilObject.getEnvirThickness()** - returns the maximum thickness (in meters) of the oil/mousse mixture in a contiguous slick including droplet transport and Langmuir processes.(ref)

**MyOilObject.getC1()** - returns the thickness ratio of the thick to mean portions of a contiguous slick due to droplet transport processes.

**MyOilObject.getC2()** - returns the thickness ratio of the thick to mean portions of a contiguous slick due to Langmuir processes (wind rows)

**MyOilObject.getWaterCont()** - returns the water fraction of the oil/mousse mixture in percent

**MyOilObject.getVolInit()** – returns the initial volume of the object in cubic meters

**MyOilObject.getPresentVol()** - returns the present oil volume of the object in cubic meters

**MyOilObject.getVolEvap()** - returns the volume of oil the object lost to evaporation in cubic meters

**MyOilObject.getVolNatDisp()** - returns the volume of oil the object lost to natural dispersion in cubic meters

**MyoilObject.getWindRowSpacing()** - returns the spacing between wind rows in meters

**MyOilObject.getMinThickness()** - returns the minimum thickness of the slick in meters

**MyOilObject.isMinThickness()** - returns true if minimum thickness has reached set limit, otherwise false

**MyOilObject.arePseudoCompSet()** - returns true if the oil's pseudo components are set (either default of user defined), otherwise false

**MyOilObject.areUserPseudoCompSet()** - returns true if user has specified pseudo components, otherwise false

**MyOilObject.isOilSlickLeft()** - returns true if mass balance indicates that some oil is left, otherwise false

**MyOilObject.getAreaMod()** - returns a fractional estimate (0 to 1.0) of the amount of the total slick

area that is so thin that it would appear as scattered sheen during an overflight

This completes the description of the public interface to OilWx objects. A simple example of a host program is presented in Appendix 2. This is a short JAVA program that creates an OilWx object, completes it initialization, and then integrates the time history. The "getters" are used to create output to the console which can then be pasted into a spreadsheet for graphing or additional calculations. A sample of the console output is shown in Appendix 3 and the graphed spreadsheet output is shown in Appendix 4.

This simple host program can be use to insure accuracy when porting the OilWx object code from one language to another and was used when moving the original JAVA code to ACTIONSCRIPT. The final spreadsheets could be subtracted from each other and uniform null results implied consistent coding of the algorithms.

# 3 Spreading and Weathering Algorithms

This section will list and describe the major algorithms that control the behavior of the OilWx object. These are the methods that set and manipulate the private internal state variables of the object. The host model generally does not have direct access to these variables and only modifies them through the public interface (described above). Internally the public interface methods often call private algorithms or methods that have direct access to the state variables, but this only occurs under the pre-defined control established by the public interface. Appendix 1 provides a complete list of the OilWx's state variables that can be accessed at any time by the objects methods. The host object will have access to a limited subset through the public interface.

The technical references for the algorithms that are used here come from three different sources. The first of these is the ADIOS® program distributed by NOAA, which has its technical manual included in the program help. In what follows, this will be referred to as (ADIOS). The second group of references were obtained via personal communications and review of notes from ADIOS development team members. These in general supported, and at times extended, the original ADIOS documentation. Particularly helpful were the discussions with Robert Jones (NOAA) and Roy Overstreet (NOAA-retired). Within this report these will be referenced as (personal communication – Jones) or (personal communication – Overstreet). The third reference for this report is the Genwest Systems, Inc Technical Note "Development of Spreading Algorithms for the MARO (Model for Assessment of Response Options) Calculator" (J.A.Galt and Roy Overstreet, 2009)

## *3.1 Creator*

The OilWx creator is a simple routine that sets up the object and over writes a set of oil descriptors and flags that will act as parameters to describe the default type oil if no additional information is provided during the initialization.

The program code is read only as follows:

```
1.   public OilWx(double inAPI,double inVis, double visRefTemp,double Volume)
2.        {
3.                  // set defaults for minimum oil
4.                  api=inAPI;
```

```
5.              refVis=inVis;
6.              refVisT=visRefTemp+273.18; // all internal temperatures are K
7.              distData=false;
8.              v0= Volume;
9.              age= 0.0;
10.         }
```

In line 6 we note that although the user inputs temperature in Centigrade computational values are converted to Kelvin

## *3.2 Add Distillation Data*

If the host program decides to provide distillation curve data this method is called a number of times, once for each of the pseudo components that is entered. On each call the following code is executed. As can be seen it first set a state variable flag to indicate that distillation data is set by the host program and then adds a data pair to the “massFrac” and “boilPt” vectors. No data checking is done during any of these calls, but before the data is used it will be checked to insure that it is consistent with minimum model standards as is described in the public interface above.

```
1.          public void AddDistillationCut(double clumMassFraction,double boilingPt)
2.          {
3.              distData= true;
4.              massFrac.add(clumMassFraction);
5.              boilPt.add(boilingPt+273.0);
6.      }
```

When the OilWx object is used by a host object that provides distillation data it may result in a product that has special characteristics and unusual weathering behavior. If this is a specifically blended distillation product this would be appropriate, but the end user should be aware that some distillation curves lead to products whose weathering characteristics may be particularly sensitive to small changes in input parameterization.

## *3.3 Set Maximum Area*

This method just sets a state variable to something other than the initial default which is Double.MAX_VALUE.

```
1.  public void SetMaximumArea(double area)
2.      {
3.          areaMax= area;
4.      }
```

## *3.4 Set Mousse Onset Point*

This method sets and overrides default values for some state variables used in the calculation of mousse formation.

```
1.  public void SetMousseOnsetPoint(double evaporatedFraction)
2.      {
3.          EmOnsetFraction= evaporatedFraction;
4.          EmOnsetTime=Double.MAX_VALUE;
5.          mousseOnsetFromUser=true;
6.      }
```

### 3.5 Set Mousse Onset Time

This method sets and overrides default values for some state variables used in the calculation of mousse formation.

```
1.   public void SetMouseOnsetTime(double emStartObserved)
2.       {
3.                   EmOnsetTime= emStartObserved;
4.                   EmOnsetFraction=Double.MAX_VALUE;
5.                   mousseOnsetFromUser=true;
6.       }
```

### 3.6 Set Wind Row Spacing

This method resets a default value for one of the state variables.

```
1.   public void SetWindRowSpacing(double distBetweenWR)
2.       {
3.                   mixLayer= distBetweenWR/3.0;;
4.       }
```

### 3.7 Set Minimum Thickness

This method resets a default value for one of the state variables.

```
1.   public void SetMinThickness(double minVal)
2.       {
3.                   minThickness= minVal;
4.       }
```

### 3.8 Set Environment

This is the first of the OilWx's methods that performs a significant number of tasks which include setting some state variables, checking the current state of various flags and calling additional private methods that perform component initialization based on whether the OilWx object is running with default values or has been modified with optional host program input.

The coding for this object is given below and is described as follows:

Line 3-8: the object is assumed to have the default number of pseudo components. This value is a variable hard-coded into the object initialization with a value of 10. During initial program testing a value of 5 was used to match the approach used in ADIOS, but some sensitivity to this choice was found in the output, so a span of numbers between 3 and 20 were tested. No significant variations were seen for 10 or more pseudo components and this number did not adversely impact the code execution speed so it became the default. Next, method variables are input with temperature converted to Kelvin and a minimum wind speed of 1 m/sec if the user has entered less than that value.

Line 9: A check is made of the flag to see if any pseudo components have been set. Once inside the block the flag is set to true. This insures that the following block of code will only be executed the first time the method is called. A check is then made to see if the host program (user) has set any custom pseudo components. If not, the internal private method SetDefaultPC(api) is called, otherwise the numberOfPC= SetUserPseudoComponents() is called instead. These methods will be described later and in either case they initialize all the pseudo components that describe the oil.

Line 21: This line of code checks whether the oil's chemical properties have been initialized. This is done by checking if the oil has been assigned an initial viscosity. If not the following block of code is executed.

Lines 23-29: In this block of code the initial viscosity of the oil is corrected for the environmental temperature and an expansion coefficient is calculated. These algorithms are taken directly from the formulation used in ADIOS and project development notes (personal communication Jones)

Lines 30-37: In this block of code the environmental temperature is used in a simple equation of state to determine the density of sea water. Then the environmental density of the base oil is calculated using the algorithms from ADIOS. The water and oil densities are compared and a warning flag is set if it looks like the oil might sink.

Lines 38-64: This block starts off by calling a private method CalculateVis() which contains encapsulated ADIOS code that calculates general viscosity, It then calculates a number of constants that are used in the algorithms that describe water droplet formation content, viscosity and density of oil and water droplet mixtures. These algorithms were taken from ADIOS code and development notes (personal communication Jones and Overstreet).

Lines 78-91: The final block of code in this method sets or resets (on multiple calls) the coefficients that are used in algorithms that calculate natural dispersion processes. These are generally similar to the methods used in ADIOS, but slightly more modern wave statistics were used and better references and development are given in (J.A.Galt and Roy Overstreet, 2009).

```
1.  public int SetEnvironment(double wind10,double waterTemp,double mixLayerDepth)
2.      {
3.              int numberOfPC= defaultNumPC;
4.              wTemp= waterTemp+273.18; // all internal temperatures are K
5.              wind= wind10;
6.              // check for zero wind default min = 1m/sec
7.              if(wind<1.0)wind=1.0;
8.              mixLayer= mixLayerDepth;
9.              if(!pseudoComponentsSet) // only set these once
10.             {
11.                     pseudoComponentsSet= true;
12.                     if(!distData){
13.                             // set default pesudo components
14.                             SetDefaultPC(api);
15.                     }else{
16.                             // set user defined distillation data
17.                             numberOfPC= SetUserPseudoComponents();
18.                             if(numberOfPC<0)return numberOfPC;
19.                     }
20.             }
21.             if(oilVis0==0.0) // this section should only be done once
22.             {
23.                     // initialize the constant used for vis change due to evap
24.                     double Ct= 5000.0; // constant K
25.                     double fac= Ct*(1.0/wTemp-1.0/refVisT);
26.                     oilVis0= refVis*Math.exp(fac);
27.                     CF= 1.5*Math.sqrt(oilVis0);
28.                     if(CF<2.0)CF= 2.0;
29.                     if(CF>12)CF= 12.0;
30.                     // set the water density at water temperature kg/m3
31.                     wRho= 1020*(1.0-1.12E-4*(waterTemp-4.0)); // add 2% for sea water
32.                     // set the initial oil density at water temperature from api
33.                     oilRho= 1000*(141.5/(131.5+api));
34.                     double refRhoT= 273.18+16.0;
35.                     oilRho*= (1.0-expCoefRho*(wTemp-refRhoT));
36.                     // check for sinking oil and set error flag
37.                     if(oilRho>wRho)numberOfPC=-8;
38.                     // initialize Mousse routine constants
```

```
39.                 double dyVis= oilRho*(1E-6)*CalculateVis();
40.                 if(dyVis>0.050)
41.                 {
42.                         Ymax= 0.9-0.0952*Math.log(dyVis/0.050);
43.                 }else{
44.                         Ymax= 0.9;
45.                 }
46.                 Smax= (6.0/dMin)*Ymax/(1.0-Ymax);  // used in water up take
47.                 Schk= (6.0/dMax)*Ymax/(1.0-Ymax);  // used in water up take
48.                 // calculate evaporation threshold for Mousse formation from ADIOS
49.                 double TBP= 532.98-3.1295*api;
50.                 double TG= 1356.7-247.36*Math.log(api);
51.                 double A1est= (483.0-TBP)/TG;
52.                 if(A1est<0.0)
53.                         {A1est=0.0;}
54.                 else if(A1est>0.4)
55.                         {A1est=0.4;}
56.                 double A2est= 1.0;
57.                 if(api<26.0)
58.                         {A2est=0.08;}
59.                 else if(api>50.0)
60.                         {A2est=0.303;}
61.                 else
62.                         {A2est= -1.038-0.78935*Math.log10(1.0/api);}
63.                 if(!mousseOnsetFromUser)EmOnsetFraction= 0.5*(A1est+A2est);
64.                 diag=EmOnsetFraction;
65.         }
66.         double k0y= 2.042*1e-6; // used in Roberts documentation
67.         Ky=(k0y)*wind*wind;
68.         Ks=6.0*Ky/dMax;
69.         if(L==0.0) // this section should only be done once
70.         {
71.                 // set up Initial gravitational spreading constants
72.                 L= Math.pow(v0,0.3333);
73.                 double fac1=Math.pow(L,1.1428);
74.                 double fac2=Math.pow(1E-6*CalculateVis(),-.4285); //convert cS => MKS
75.                 T0= fac1*fac2;
76.                 l0= Math.pow((L*L*T0*T0), 0.3333);
77.         }
78.         // calculate natural dispersion rate per sec
79.         double d= 5.48E-3*Math.pow(wind,5);
80.         d=Math.pow(d,0.57);
81.         // wave breaking after Ming and Farmer (JPO) 1994
82.         double f= 1.18e-3*Math.pow(wind,1.03);
83.         // set low wind cut off
84.         if(wind<3.0)f=0.0;
85.         double vEntrained= Math.pow(MaxDispSize,1.7)/1.7; //need to be integral
86.         double F= (1.0-Math.exp(-f*dtDropModel));
87.         DissapationFactor= F*vEntrained;
88.         // set Delvigne droplet wind dependent term
89.         DF= d*F;
90.         //diag=numberOfPC;
91.         return numberOfPC;
92. }
```

## *3.9 Step*

This method will integrate the OilWx object over a length of time specified by the input parameter which is given in seconds. It starts off the computation by checking to see if there is any oil left at the beginning of the time step.

Lines 3-12: This checks the sum of the pseudo components to find out if any oil remains. If the answer is no or if the oilRemainingFlag has been set to false in some other calculation, then the flag is confirmed and the method does nothing else.

Lines 14: If oil remains then the physical processes associated with evaporation is called through a private method to be described later.

Line 15: The remaining oil is decreased by the process of natural dispersion, by a call to another private method that will be described later.

Line 16: The remaining oil will be subject to the possibility of taking on water forming a water-in-oil emulsion (mousse) which once again is computationally carried out by a call to a private method that will be described later.

Lines 17-20: At this point the elongation of the slick is calculated for the integration time step as 3% of the wind speed and another half of a percent is added if the winds are strong enough to cause significant wind row formation (Langmuir circulation). Finally, a volumetric product thickness is calculated based on a private method which is then modified using the droplet differential thickness model (represented by yet another private method, getC1()). This follows the algorithms outlined in (J.A.Galt and Roy Overstreet, 2009).

Lines 21-27: The minimum thickness flag is checked and reset to reflect current conditions at the end of this integration time step and finally the age of the OilWx object is updated for the time step that has just been taken.

The code for this method is as follows:

```
1.  public void Step(double stepLengthSec)
2.      {
3.              //check if there is still any oil, if not don't step
4.              double moleSum= 0.0;  // total number of moles remaining in spill
5.              for(int i=0;i<pcVol.size();i++)
6.              {
7.                      moleSum+= pcVol.get(i);
8.              }
9.              if((moleSum==0.0)||(!oilRemainingFlag)){
10.                     oilRemainingFlag= false;
11.                     return;
12.             }
13.             // step processes
14.             StepEvaporation(stepLengthSec);
15.             StepNaturalDispersion(stepLengthSec);
16.             StepWaterContent(stepLengthSec);
17.             double windFac=0.03;
18.             if(wind>6.0)windFac+=0.005;
19.             windDistance+=windFac*wind*stepLengthSec;
20.             double chkThickness= getProdThickness()*getC1();
21.             if(chkThickness>minThickness)
22.             {
23.                     minThicknessFlag= true;
24.             }else{
25.                     minThicknessFlag= false;
26.             }
27.             age+=stepLengthSec;
28.     }
```

## *3.10 Step Clean*

This method does the same thing as the Step method after removing specified amount of oil from the OilWx object. It starts out by checking how much oil is present in all the remaining pseudo

components. It then checks to see if the host program specified the amount to be removed making sure to account for whether the host program specified the amount removed as pure oil or as an oil and water mixture (mousse). If the amount removed is more than the amount remaining, the oilRemaining flag is set to false and the method returns. If there is still oil remaining after the removal, it is removed from each of the pseudo components based on their relative volumetric presence. After this adjustment is made the method simply calls the usual Step method described above.

The method code is as follows:

```
1.  public void StepClean(double stepLengthSec,double stuffRecovered,boolean oilOnlyFlag)
2.      {
3.      //check if there is still any oil, if not don't step
4.      if(oilRemainingFlag)
5.          {
6.              // remove oil recovered
7.              int pcCount= pcVol.size();
8.              // find out how much is there
9.              double nowVolume= getPresentVol();
10.             // set recovered amount and discount water
11.             double oilRecovered= stuffRecovered;
12.             if(!oilOnlyFlag) oilRecovered*=(1.0-yFraction);
13.             // back out moles from each pseudo component
14.             if(oilRecovered>=nowVolume)
15.             {
16.                 oilRemainingFlag=false;
17.             }else{
18.                 for(int i=0;i<pcCount;i++){
19.                     double pcFrac= pcVol.get(i)*pcMV.get(i)/nowVolume;
20.                     pcVol.set(i,(pcVol.get(i)-
                    pcFrac*oilRecovered/pcMV.get(i)));
21.                 }
22.             }
23.         }
24.         Step(stepLengthSec);
25.     }
```

## 3.11 SetDefaultPC

This private method is called from SetEnvironment(). It initializes the pseudo components under the default options where the oil's API gravity is used with correlation data to estimate a distillation curve. The algorithms are based on the ADIOS development notes (personal communication Jones). The major difference between this and the ADIOS code is that here the default number of pseudo components is 10 rather than 5 (as explained previously).

The code for the method is as follows:

```
1.  private void SetDefaultPC(double api)
2.      {
3.          double T0=457.16-3.3447*api; // these two formula from Adios
4.          double dT=1356.7-247.36*Math.log(api);
5.          double sumVolMW= 0.0;
6.          double sumVol= 0.0;
7.          for(int i=0;i<defaultNumPC;i++)
8.          {
9.              double BP=T0+dT*((double)i+0.5)/defaultNumPC; //deg K
10.             double MV= (7.000E-5)-(2.102E-7)*BP+(1.000E-9)*BP*BP;
11.             double MW= 0.04132-(1.985E-4)*BP+(9.494E-7)*BP*BP;
12.             double vi= v0/defaultNumPC;
13.             pcVol.add(vi/MV); // moles in this cut m3/moles
```

```
14.                                     pcBP.add(BP);
15.                                     pcMV.add(MV);
16.                                     pcMW.add(MW);
17.                                     sumVolMW+= MW*vi/MV;
18.                                     sumVol+= vi/MV;
19.                             }
20.         }
```

## 3.12 SetUserPseudoComponents

This private method is called by SeEnvironment(). It initializes and sets the pseudo components representing the oil if the user or host program has specified customized distillation data. Each time the host object calls AddDistillationCut(), values are added to the state vectors "massFrac" and "boilPt".
Lines 3 - 29: The first part of this method scans these values and makes sure that they are a parametric representation of a monotonically increasing boiling point curve.
Lines 31- 34: This part of the code sets the bounding values for the boiling point curves.
Lines 34 – on: At this point the pseudo components are set up following the same algorithms that are used in the default case which is described in ADIOS documentation and development notes (personal communication, Jones)
The code for this method is as follows:

```
1.    private int SetUserPseudoComponents()
2.          {
3.                      int message= massFrac.size();
4.                      if(message<minMumUserPC)return -7;
5.                      double maxcut= 0.0;
6.                      double mincut= 100.0;
7.                      double maxtemp= 0.0;
8.                      double mintemp= 1000.0;
9.                      double valFrac0= 0.0;
10.                     double valBp0= 0.0;
11.                     for(int i=0;i<massFrac.size();i++)
12.                     {
13.                             double valFrac1= massFrac.get(i);
14.                             double valBp1= boilPt.get(i);
15.                             // update range
16.                             if(maxcut<valFrac1)maxcut= valFrac1;
17.                             if(mincut>valFrac1)mincut= valFrac1;
18.                             if(maxtemp<valBp1)maxtemp= valBp1;
19.                             if(mintemp>valBp1)mintemp= valBp1;
20.                             if((valFrac1-valFrac0)<0.0)return -1;
21.                             if((valBp1-valBp0)<0.0)return -2;
22.                             valFrac0= valFrac1;
23.                             valBp0= valBp1;
24.                     }
25.                     if(mincut<0.0)return -3;
26.                     if(maxcut>100.0)return -4;
27.                     if(mintemp<0.0)return -5;
28.                     if(maxtemp>1000.0)return -6;
29.                     numUserPC= message;
30.                     double[] T= new double[numUserPC+1];
31.                     T[0]=boilPt.get(0)-massFrac.get(0)*
32.                             (boilPt.get(1)-boilPt.get(0))/(massFrac.get(1)-massFrac.get(0));
33.                     T[numUserPC]=boilPt.get(numUserPC-1)+(100.0-massFrac.get(numUserPC-1))*
34.                             (boilPt.get(numUserPC-1)-boilPt.get(1))/(massFrac.get(numUserPC-1)-
            massFrac.get(1));
35.                     double[] M= new double[numUserPC+1];
36.                     M[0]=0.0;
37.                     M[numUserPC]=100.0;
38.                     for(int j=1;j<numUserPC;j++)
39.                     {
40.                             T[j]= boilPt.get(j-1);
41.                             M[j]= massFrac.get(j-1);
42.                     }
43.                     double sumVolMW= 0.0;
```

```
44.         double sumVol= 0.0;
45.         for (int k=1;k<numUserPC+1;k++)
46.         {
47.             double BP= 0.5*(T[k]+T[k-1]);
48.             double MV= (7.000E-5)-(2.102E-7)*BP+(1.000E-9)*BP*BP;
49.             double MW= 0.04132-(1.985E-4)*BP+(9.494E-7)*BP*BP;
50.             double vi= (M[k]-M[k-1])*v0/100.0;
51.             pcVol.add(vi/MV); // moles in this cut m3/moles
52.             pcBP.add(BP);
53.             pcMV.add(MV);
54.             pcMW.add(MW);
55.             sumVolMW+= MW*vi/MV;
56.             sumVol+= vi/MV;
57.         }
58.         averMW= sumVolMW/sumVol;
59.         // calculate evaporation rate constants
60.         double K0= 0.0048*1.3676*Math.pow(0.018/averMW,0.3333)*Math.pow(wind,0.7777);
61.         double R1= 8.205E-5; // gas constant (atmos m3)/(mol K)
62.         for(int i=0;i<pcVol.size();i++)
63.         {
64.             double vp= VaporPressure(pcBP.get(i));
65.             double moleRate= K0*vp/(R1*wTemp); // moles/(m2 sec)
66.             pcEvapRate.add(moleRate);
67.         }
68.         return message;
69.     }
```

## 3.13 Get Density

This method returns the density of the oil taking into account the increase due to evaporation and the decrease caused by the inclusion of water droplets (mousse). These are basically the same algorithms that are used in ADIOS. There is also a check to ensure that numerical round off will not force the evaporation processes to actually produce a sinking oil.

The code for this method is as follows:

```
1.  public double getDensity()
2.      {
3.          double rhoOil=oilRho*(1.0+0.18*getVolEvap()/v0);
4.          double delRho= (wRho-rhoOil)/wRho;
5.          if(delRho<1E-4)rhoOil=wRho*(1.0-1E-4);
6.          return rhoOil*(1.0-yFraction)+wRho*yFraction;
7.      }
```

## 3.14 Get Thickness

This method returns the calculated mean thickness of the slick by dividing the total volume of the oil by the calculated area of the contiguous slick size.

```
1.  public double getThickness()
2.      {
3.          return CalculateVolume()/CalculateArea(age);
4.      }
```

## 3.15 Get Product Thickness

This method returns the mean thickness of the slick based on the volume of the oil and water mousse mixture rather than just the oil.

```
1.  public double getProdThickness()
2.      {
3.          return getThickness()/(1.0-yFraction);
4.      }
```

## 3.16 Get Environmental Thickness

This method estimates the thickest portion of a contiguous slick by including maximum-to-mean ratio information due to oil droplet migration and wind row (Langmuir) cell formation. These processes are described in (J.A.Galt and Roy Overstreet, 2009).

```
1.  public double getEnvirThickness()
2.      {
3.              return getC1()*getC2()*getProdThickness();
4.      }
```

## 3.17 Get C1

This method implements the calculation of the maximum-to-mean ratio due to droplet migration as described by (J.A.Galt and Roy Overstreet, 2009). The algorithm is implemented in terms of a power law based on a non-dimensional parameterization. The non-dimensional parameter is the ratio of oil in the surface slick to the oil contained in sub-surface droplets and is calculated in Line 11 of the following code:

```
1.  public double getC1()
2.  {
3.              if(!isOilSlickLeft())return 0.0;
4.              double C1=1.0;
5.              // trap for low wind case
6.              if(wind<3.0)
7.              {
8.                      return C1;
9.              }
10.             // calculate the scale factor (SlickMass/DropletMass)
11.             double scale=getDensity()*getProdThickness()
                        1.   /CalculateDropletMass();
12.             C1= 1.0+2.2*(1.0-Math.exp(-Math.sqrt(3.0*scale)));
13.             return C1;
14.     }
```

## 3.18 Get C2

This method calculates the ratio of the maximum-to-mean oil thickness due to the formation of wind rows or Langmuir cells. This methodology is described in (J.A.Galt and Roy Overstreet, 2009) and is done by calling a private CalculateC2() which will be described later. Note that based on the definition given in this reference C2 is a multiplicative factor so it is the ratio calculated after the C1 correction has already be made.

```
1. public double getC2()
2.      {
3.              if(!isOilSlickLeft())return 0.0;
4.              double C1= getC1();
5.              double thicknessNow= C1*getProdThickness();
6.              double val= calculateC2(wind,getDensity(),thicknessNow);
7.              return val;
8.      }
```

## 3.19 Get Wind Row Spacing

This method returns the wind row spacing that is assumed to be 3 time the mixedLayer depth. The

default mixedLayer depth is 10 meters which could have been reset if the host program had supplied an alternate wind row spacing (ref 3.6 SetWindRowSpacing).

```
1.  public double getWindrowSpacing()
2.      {
3.          return 3.0*mixLayer;
4.      }
```

## 3.20 CalculateC2

This method contains the algorithm that calculates the ratio between the maximum and mean thickness in a contiguous oil slick caused by the formation of wind rows or Langmuir cell formation as described in (J.A.Galt and Roy Overstreet, 2009). The actual numerical model is run in its native code format and the results are then represented by a number of cases using a power law interpolation scheme. The actual algorithm is shown below and uses several wind speed cases each of which has its own curve fitting scheme

```
1.  private double calculateC2(double wind,double rho0,double thickness0)
2.      {
3.          // convert thickness in meters to mm
4.          double thick= 1000*thickness0;
5.          if(thick>1.0)thick=1.0;
6.          // default low wind cut off
7.          if(wind<3.0)return 1.0;
8.          double x= 2.0*(rho0-800.0)/(wRho-800.0);
9.          double cLow;
10.         double nLow;
11.         double cHigh;
12.         double nHigh;
13.         double c;
14.         double n;
15.         double val;
16.         if(wind<5.0)
17.         {
18.             double wCoef=(wind-3.0)/2.0;
19.             cLow= 0.8851+x*(-0.4312+x* 0.6034);
20.             nLow= - 0.8263+x*( 0.0466+x*-0.0432);
21.             cHigh= 2.3015+x*(-0.6293+x* 1.2263);
22.             nHigh= - 0.7527+x*(-0.0050+x*-0.0183);
23.             c=cHigh*wCoef+cLow*(1.0-wCoef);
24.             n=nHigh*wCoef+nLow*(1.0-wCoef);
25.             val= c*Math.pow(thick,n);
26.         }
27.         else if(wind<8.0)
28.         {
29.             double wCoef=(wind-5.0)/3.0;
30.             cLow= 2.3015+x*(-0.6293+x* 1.2263);
31.             nLow= - 0.7527+x*(-0.0050+x*-0.0183);
32.             cHigh= 5.1184+x*(-1.0233+x* 2.1188);
33.             nHigh= - 0.6799+x*(-0.0044+x*-0.0092);
34.             c=cHigh*wCoef+cLow*(1.0-wCoef);
35.             n=nHigh*wCoef+nLow*(1.0-wCoef);
36.             val= c*Math.pow(thick,n);
37.         }
38.         else if(wind<10.0)
39.         {
40.             double wCoef=(wind-8.0)/2.0;
41.             cLow= 5.1184+x*(-1.0233+x* 2.1188);
42.             nLow= - 0.6799+x*(-0.0044+x*-0.0092);
43.             cHigh= 6.9004+x*(-1.2544+x* 2.6376);
44.             nHigh= - 0.6600+x*(-0.0034+x*-0.0069);
45.             c=cHigh*wCoef+cLow*(1.0-wCoef);
46.             n=nHigh*wCoef+nLow*(1.0-wCoef);
47.             val= c*Math.pow(thick,n);
48.         }
```

```
49.                 // default wind > 10 m/s
50.                 else
51.                 {
52.                         c= 6.9004+x*(-1.2544+x* 2.6376);
53.                         n= - 0.6600+x*(-0.0034+x*-0.0069);
54.                         val= c*Math.pow(thick,n);
55.                 }
56.                 if(val>80.0)val=80.0; // low thickness cutoff
57.                 if(thickness0>0.001)val=val*Math.exp(-(thickness0-0.001)/0.001);// high
                                            thickness cutoff                    // back out C1 value
58.                 val= val/getC1();
59.                 if(val<1.0)val=1.0;
60.                 return val;
61.         }
```

## 3.21 Vapor Pressure

The method will calculate the vapor pressure of the oil (a pseudo component) based on its boiling point. The method comes from Lyman's book and is described in the ADIOS documentation and development notes (personal communication, Jones). The actual code is listed below:

```
1.    private double VaporPressure(double BP) //  vp in atmos
2.          {
3.                  // loop through the pc's for evaporation equations
4.                  double dZ= 0.97; // constant from Lyman
5.                  double R= 1.987; // gas constant (cal)/(mol K)
6.                  double dS= 8.75+R*Math.log(BP);  // Lyman 14-16 assume Kf=1.0
7.                  double c2= 0.19*BP-18.0; // Lyman 14-15
8.                  double T1= BP-c2;
9.                  double T2= wTemp-c2;
10.                 double fac1= dS*T1*T1/(dZ*R*BP);
11.                 double fac2= (T2-T1)/(T2*T1);
12.                 double vapP= Math.exp(fac1*fac2); // Lyman 14-14
13.                 return vapP;
14.         }
```

## 3.22 Step Evaporation

This method calculates the evaporation that takes place over a time step using the same basic pseudo component approach that is used in ADIOS and is explained in its technical documentation and development notes (personal communication, Jones). It should be noted however that OilWx's evaporation behavior will not be identical to ADIOS because the algorithm depends on "effective area" and the OilWx model calculates spreading (and thus area) in a different way. These differences show up in the actual slick foot print area (see algorithm description below) and in the thickness distribution within the slick area that allows some of the overall foot print to become so thin that it "sheens out" and no longer represents an effective evaporative area (see area modification algorithm below). The following code segment calculates the molar loses for each pseudo component:

```
1.    private void StepEvaporation(double stepLengthSec)
2.          {
3.                  // calculate the sum of the pc moles
4.                  double moleSum= 0.0;  // total number of moles in spill
5.                  for(int i=0;i<pcVol.size();i++)
6.                  {
7.                          moleSum+= pcVol.get(i);
8.                  }
9.                  double AModCoef=1.0;
10.                 double effArea= CalculateArea(age+stepLengthSec/2.0)*(1.0-yFraction)*(1.0-
                    AModCoef*AreaMod());
11.                 for(int i=0;i<pcVol.size();i++)
12.                 {
13.                         double downWindFactor=Math.pow(windDistance,0.1111);
14.                         if(downWindFactor<1.0)downWindFactor= 1.0;
```

```
15.                        double Dmi= pcEvapRate.get(i)*(pcVol.get(i)/moleSum)*
effArea*stepLengthSec/downWindFactor;
16.                        if(Dmi>pcVol.get(i))Dmi= pcVol.get(i);
17.                        // adjust the old molar volumes
18.                        pcVol.set(i, (pcVol.get(i)-Dmi));
19.                        // subtract real vol to evap sum
20.                        volumeEvap+=Dmi*pcMV.get(i);
21.                }
22.        }
```

## 3.23 Step Natural Dispersion

This method calculates the amount of oil that is lost from the slick due to natural dispersion. The basic approach is due to Delvigne's work and is the same as is described in (J.A.Galt and Roy Overstreet, 2009). It is also the same as used in ADIOS and described in its documentation with slight variations in the viscosity coefficients as described in the development notes (personal communication, Overstreet). The actual loss term is calculated in Line 7-8: which uses Overstreet's viscous formulation (CalculateDisConst()) and the Delvigne (wind dependent) energy factor calculated in the Set Environment method. There is also an area dependent term, so once again the dispersion losses in OilWx will not be identical to those calculated by ADIOS because of the differences in which these calculations are made. It should be noted that because of the way mousse formation is modeled, there are possible representation of distilled products with "user defined" pseudo components that interact through the viscosity and a sensitive feedback is seen in the dispersion results. That is, small changes in the blend formulation of a diesel may lead to large predicted differences in the amount of natural dispersion. The model code is as follows:

```
1.    private void StepNaturalDispersion(double stepLengthSec)
2.        {
3.                // if maximum area is restricted there is no dispersion
4.                if(areaMax!=Double.MAX_VALUE)return;
5.                // normal dispersion
6.                double AModCoef= 1.0;
7.                double massDis= CalculateDisConst()*DissapationFactor*(1.0-yFraction)
8.                                *CalculateArea(age)*(1.0- AmodCoef*AreaMod())*
stepLengthSec/dtDropModel; // kg of oil dispersed
9.                // convert to volume
10.               double volDis= massDis/getDensity(); // volume of droplets
11.               int pcCount= pcVol.size();
12.               double nowVolume= getPresentVol();
13.               // back out moles from each pseudo component
14.               if(volDis>nowVolume)
15.               {
16.                       oilRemainingFlag=false;
17.               }else{
18.                       for(int i=0;i<pcCount;i++){
19.                               double pcFrac= pcVol.get(i)*pcMV.get(i)/nowVolume;
20.                               pcVol.set(i,(pcVol.get(i)-pcFrac*volDis/pcMV.get(i)));
21.                       }
22.               }
23.               volumeDis+=volDis;
24.       }
```

## 3.24 Step Water Content

This method will calculate the water content at the end of an additional time step. It follows the same algorithm used in ADIOS and described in the development notes (personal communication, Jones). This method also calculates the mean droplet size which is a parameter used in the mousse viscosity calculation. (see next section)

```
1.  private void StepWaterContent(double stepLengthSec)
2.          {
3.                  // check if evaporation has started
4.                  double evapFraction= getVolEvap()/v0;
5.                  if(evapFraction<EmOnsetFraction)return;
6.                  // set a timestep for 1%change
7.                  double dt0= stepLengthSec;
8.                  double dt= dt0;
9.                  if(dt*Ky>0.01)dt=0.01/Ky;
10.                 while(dt0>0.0)
11.                 {
12.                         if(yFraction<Ymax)yFraction+=Ky*dt;
13.                         if(dSize>dMin)S+=Ks*dt;
14.                         if(yFraction>=Ymax)dSize= 0.6*yFraction/(S*(1.0-yFraction));
15.                         if(dt>dt0){dt0=dt;}else{dt0-=dt;}
16.                 }
17.                 if(yFraction>1.0)yFraction=1.0;
18.         }
```

## 3.25 Calculate Viscosity

This method calculates the oil/mousse mixture's viscosity in several steps. The algorithm is similar to the one used in ADIOS and described in the development notes (personal communication, Jones). The algorithm first estimates the change in the initial viscosity due to loss due by evaporation and then this value is augmented by an exponential, which depends on water content and water droplet size.

```
1.  private double CalculateVis()
2.          {
3.                  // calculate the change due to fraction evap
4.                  double fractionEvap= volumeEvap/v0;
5.                  double vis= oilVis0*Math.exp(CF*fractionEvap);
6.                  // calculate the change due to water content
7.                  // make 1st estimate using MacKays const as in Adios 1
8.                  double Cy= 0.65*dMin/dSize;
9.                  double fac= 2.5*yFraction/(1.0-Cy*yFraction);
10.                 vis= vis*Math.exp(fac);
11.                 return vis;
12.         }
```

## 3.26 Calculate Area

This method calculates the bounding area of a contiguous slick. Initially the spreading is assumed to be radial and dominated by gravitational slumping. This is essentially the first phase in classical Fay-Hoult spreading. After this phase, it is assumed that ocean surface dynamics dominate and the radial component of the spreading follows a power to the 3/2 spreading law as in the classical Richardson-Batchelor-Oboko studies. The downwind axis is then extended by three percent of wind speed and finally an additional one half a percent of the wind speed is added if the wind is greater than 6 m/sec. to represent the additional downwind velocity expected in (wind rows) convergence lines associated with Langmuir cells.

```
1.  private double CalculateArea(double time)
2.          {
3.                  double l1= 0;
4.                  if(time<T0) // still gravitational spreading
5.                  {
6.                          l1= Math.pow(L*L*time*time,0.333);
```

```
7.              }else //Richardson/Batchelor/Obuko
8.              {
9.                      double fac0= Math.pow(100*l0, 0.6666)+(0.006*(time-T0)); // convert to         cm to
    use Obuko const
10.                     l1= 0.01*Math.pow(fac0, 1.5); // convert back to m now that l1 is to          the first
    power
11.             }
12.             // correct the downwind axis for droplet drift
13.             double windFac=0.03;
14.             if(wind>6.0)windFac+=0.005;
15.             double l2= l1 + windFac*windDistance;
16.             double area= l1*l2;
17.             // check if constraints override spreading
18.             if(area>areaMax)area= areaMax;
19.             return area;
20.     }
```

## 3.27 Area Modification

This is a method that is unique to the OilWx object. The droplet spreading algorithm (J.A.Galt and Roy Overstreet, 2009) provides a statistical estimate of the relative thickness distribution along the downwind axis of a contiguous slick. In addition the OilWx object also keeps track of the losses in slick volume. If the average losses are greater than the expected thickness of some portion of the contiguous slick area then we can expect that this portion will have "sheened out" and that an overflight would report scattered sheen. This portion of the slick would no longer represent an active area of the slick for input into area dependent algorithms such as evaporation and natural dispersion calculations.

```
1.  private double AreaMod()
2.      {
3.              double ex= getC1()*getC2();
4.              double fac=(v0-getPresentVol())/v0;
5.              if((ex<1.)||(fac<1e-6))return 0.0; // round off error bug fixed 13-VI-09
6.              double x= Math.pow(fac/ex, 1/(ex-1.0));
7.              return x;
8.      }
```

## 3.28 Calculate Volume

This method calculates the volume of the oil in the slick (in cubic meters) by summing up the volumes in each of the pseudo components. The algorithm starts with molar volumes which is the internal unit used by the OilWx object.

```
1.  private double CalculateVolume()
2.      {
3.              if(!isOilSlickLeft())return 0;
4.              double presentVol= 0.0;
5.              for(int i=0;i<pcVol.size();i++)
6.              {
7.                      presentVol+= pcVol.get(i)*pcMV.get(i);
8.              }
9.              if(presentVol>v0)presentVol= v0; // model can not add to initial vol
10.             return presentVol;
11.     }
```

# 4 References

ADIOS (help manual) HAZMAT NOAA DOC

Galt, J. A. and Roy Overstreet (2009) Development of Spreading Algorithms for the MARO (Model for Assessment of Response Options) Calculator – Genwest Systems, Inc , Technical Note – May 2009

Jones, Robert (personal communication) HAZMAT NOAA DOC

Overstreet,Roy (personal communication) HAZMAT NOAA DOC – retired

# 5 Appendices

## 5.1 Appendix 1 – OilWx objects state variables

```
1.	private double api;
2.	private double oilRho;
3.	private double refVis;
4.	private double refVisT;
5.	private double oilVis0=0.0;
6.	private double CF=0.0;
7.	private boolean pseudoComponentsSet= false;
8.	private boolean distData= false;
9.	private double expCoefRho= 0.0008; //from Adios
10.	/** class variables pseudo components
11.	 */
12.	private int defaultNumPC= 10;
13.	private int numUserPC= 0; // number of user defined pseudo components
14.	private int minMumUserPC= 3;
15.	private Vector<Double> massFrac= new Vector<Double>();
16.	private Vector<Double> boilPt= new Vector<Double>();
17.	private Vector<Double> pcVol= new Vector<Double>(); // in moles
18.	private Vector<Double> pcBP= new Vector<Double>();  // boiling point k
19.	private Vector<Double> pcMV= new Vector<Double>();  // mole/m3
20.	private Vector<Double> pcMW= new Vector<Double>();
21.	private Vector<Double> pcEvapRate= new Vector<Double>();// evap rate moles/(m2     sec)
22.	private double averMW= 0.0;
23.	/** class environmental variables
24.	 */
25.	private double wTemp = 10.0;
26.	private double wRho;
27.	private double wind = 5;
28.	private double mixLayer= MIX_LAYER_DEFAULT;
29.	static final double MIX_LAYER_DEFAULT= 10.0;
30.	private boolean oilRemainingFlag= true;
31.	private double v0= 1.0;  //initial volume (cubic meters)
32.	private double volumeEvap= 0.0;
33.	private double volumeDis= 0.0;
34.	private double age= 0.0;  // time since spill (sec)
35.	private boolean minThicknessFlag= true;
36.	private double minThickness= 2E-6; // dark rainbow ~ 4 wavelengths of light
37.	/** class variables controlling spreading and dispersion
38.	 */
39.	private double dtDropModel= 30;
40.	private double DF=0.0; // wind dependent part of Delvigne droplet calculation
41.	private double MaxDispSize = 50E-6; // 50 microns
42.	private double DissapationFactor=0;
43.	private double areaMax= Double.MAX_VALUE; // maximum area of spill
44.	/** Mousse related class variables
45.	 */
46.	private double yFraction= 0.0; // fractional water content
47.	private double S=0.0;
48.	private double Ymax= 0;
49.	private double Smax=0; // used in water up take
50.	private double Schk=0; // used in water up take
51.	private double EmOnsetFraction= 1.0;
52.	private double EmOnsetTime= Double.MAX_VALUE;
53.	private boolean mousseOnsetFromUser=false;
54.	private double dMax= 10E-6; // 10 microns
55.	private double dMin= 1E-6;  // 1 micron
56.	private double dSize= dMax; // default to initial value
57.	private double Ks= 0.0; // from Roberts notes
58.	private double Ky= 0.0; // from Roberts notes
59.	/** Gravitational spreading coefficients
60.	 */
```

```
61.         private double L= 0.0;
62.         private double T0= 0.0;
63.         private double l0= 0.0;
64.         private double windDistance= 0.0;
65.         private double width= 0.0;
66.         /** public access test variable used for diagnostic output
67.          */
68.     public double diag= 0.0; // flag variable used during debuggung
```

## 5.2 Appendix 2 – Test Program Code

```
1.	package oilWeathering;
2.	
3.	public class RunDefaults {
4.	
5.		/** Test engine to look at OilWx behavior
6.		 * @param args
7.		 */
8.		public static void main(String[] args)
9.		{
10.			// set parameters
11.			double api= 32;
12.			double vis= 3.8;
13.			double refT= 38;
14.			double vol= 795;
15.			double wind= 5;
16.			double wTemp= 7.2;
17.			double layerDepth= 10;
18.			// write out header line
19.			System.out.println("OIL MODEL - Java"+'\t');
20.			StringBuffer header= new StringBuffer("api= "+api+"\t");
21.			header.append("vis= "+vis+"\t");
22.			header.append("refT= "+refT+"\t");
23.			header.append("vol= "+vol+"\t");
24.			header.append("wind= "+wind+"\t");
25.			header.append("wTemp= "+wTemp+"\t");
26.			header.append("layerD= "+layerDepth+"\r");
27.			System.out.println(header);
28.			// set run times
29.			int step= 600;
30.			int interval= 36;
31.			int run=720;
32.			StringBuffer parameters= new StringBuffer("step= "+step+" sec\t");
33.			parameters.append("interval= "+(interval*step)/3600+" hrs\t");
34.			parameters.append("run= "+(step*(run-1)/3600)+" hrs\r");
35.			System.out.println(parameters);
36.			System.out.println("\r");
37.			StringBuffer volume= new StringBuffer("vol\t");
38.			StringBuffer evap= new StringBuffer("evap\t");
39.			StringBuffer disp= new StringBuffer("disp\t");
40.			StringBuffer waterCont= new StringBuffer("waterCont\t");
41.			StringBuffer visc= new StringBuffer("visc\t");
42.			StringBuffer thickness= new StringBuffer("oil\t");
43.			StringBuffer prod= new StringBuffer("prod\t");
44.			StringBuffer envir= new StringBuffer("envir\t");
45.			StringBuffer missing= new StringBuffer("missing\t");
46.			StringBuffer area= new StringBuffer("area\t");
47.			StringBuffer areaMod= new StringBuffer("areaMod\t");
48.			StringBuffer C1val= new StringBuffer("C1\t");
49.			StringBuffer C2val= new StringBuffer("C2\t");
50.			StringBuffer Diag= new StringBuffer("diag\t");
51.	
52.	
53.			//create spill
54.			OilWx mySpill= new OilWx(api,vis,refT,vol);
55.	
56.			/*
57.			// temp check dist cuts
58.			mySpill.AddDistillationCut(1,221.1); // Oil thickness = NaN, Present Volume = 0
59.			mySpill.AddDistillationCut(10,260); // Oil thickness = NaN, Present Volume = 0
60.			mySpill.AddDistillationCut(50,398.8); // Oil thickness = NaN, Present Volume = 0
61.			mySpill.AddDistillationCut(14,125); // Oil thickness = NaN, Present Volume = 0
62.			mySpill.AddDistillationCut(19,150); // Oil thickness = NaN, Present Volume = 0
63.			mySpill.AddDistillationCut(25,175); // Oil thickness = NaN, Present Volume = 0
64.			mySpill.AddDistillationCut(29,200); // Oil thickness = NaN, Present Volume = 0
```

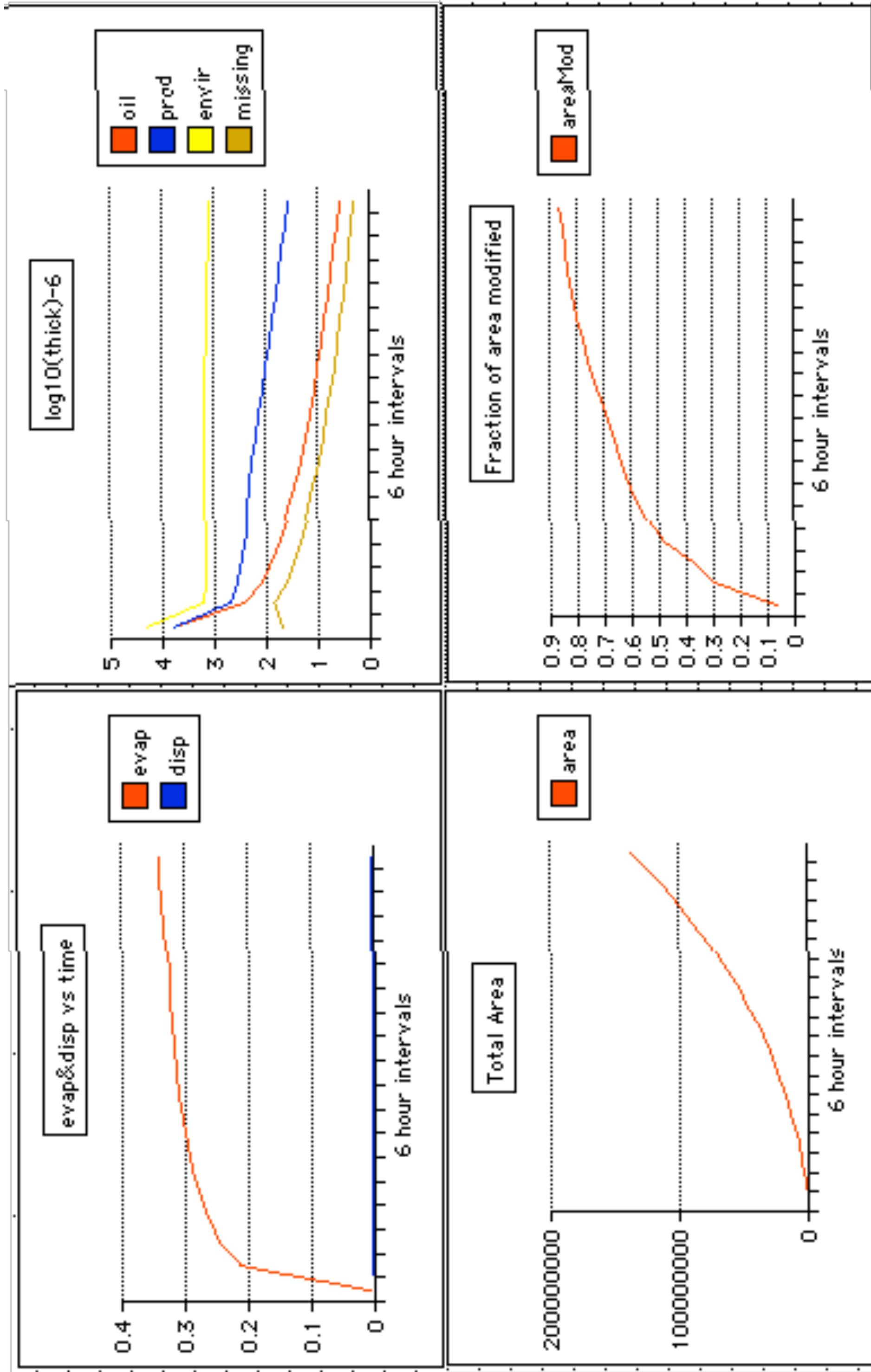
```
65.         mySpill.AddDistillationCut(34,225); // Oil thickness = NaN, Present Volume = 0
66.         mySpill.AddDistillationCut(40,250); // Oil thickness = NaN, Present Volume = 0
67.         mySpill.AddDistillationCut(46,275); // Oil thickness = NaN, Present Volume = 0
68.         mySpill.AddDistillationCut(50,200); // Oil thickness = NaN, Present Volume = 0
69.         mySpill.AddDistillationCut(56,225); // Oil thickness = NaN, Present Volume = 0
70.         mySpill.AddDistillationCut(61,250); // Oil thickness = NaN, Present Volume = 0
71.         mySpill.AddDistillationCut(66,275); // Oil thickness = NaN, Present Volume = 0
72.         mySpill.AddDistillationCut(72,300); // Oil thickness = NaN, Present Volume = 0
73.         */
74.
75.
76.         // set mousse constant
77.         //mySpill.SetMousseOnsetPoint(0.09);
78.
79.         // finish initialization
80.         int NPC= mySpill.SetEnvironment(wind, wTemp, layerDepth);
81.
82.         //System.out.println(NPC);
83.
84.
85.
86.         // output run results
87.         for(double i=0;i<run;i+=1)
88.         {
89.             /*
90.             // add this to check changing the environmenta conditions
91.             if(i%100==0){
92.                 NPC= mySpill.SetEnvironment(wind-i/100, wTemp, layerDepth);
93.                 mySpill.diag= wind-i/100;
94.             }
95.             */
96.
97.
98.             // if there is a problem this block will trap NaN's, print out an error indicating
99.             // the time step and which field took the hit.
100.            if(Double.isNaN(mySpill.getPresentVol())){System.out.println("\ni= "+i+"\t vol");break;}
101.            if(Double.isNaN(mySpill.getVolEvap())){System.out.println("\ni= "+i+"\t evap");break;}
102.            if(Double.isNaN(mySpill.getVolNatDisp())){System.out.println("\ni= "+i+"\t disp");break;}
103.            if(Double.isNaN(mySpill.getWaterCont())){System.out.println("\ni= "+i+"\t H2O");break;}
104.            if(Double.isNaN(mySpill.getViscosity())){System.out.println("\ni= "+i+"\t visc");break;}

105.            if(Double.isNaN(mySpill.getThickness())){System.out.println("\ni= "+i+"\t thickness");break;}
106.            if(Double.isNaN(mySpill.getProdThickness())){System.out.println("\ni= "+i+"\t product");break;}
107.            if(Double.isNaN(mySpill.getEnvirThickness())){System.out.println("\ni= "+i+"\t envir");break;}
108.            if(Double.isNaN(mySpill.getArea(mySpill.getAge()))){System.out.println("\ni= "+i+"\t area");break;}
109.            if(Double.isNaN(mySpill.getAreaMod())){System.out.println("\ni= "+i+"\t mod");break;}
110.            if(Double.isNaN(mySpill.getEnvirThickness())){System.out.println("\ni= "+i+"\t envir");break;}
111.            if(Double.isNaN(mySpill.getC1())){System.out.println("\ni= "+i+"\t C1");break;}
112.            if(Double.isNaN(mySpill.getC2())){System.out.println("\ni= "+i+"\t C2");break;}
113.
114.            //mySpill.StepClean(step,0.001*vol,true);
115.            mySpill.Step(step);
116.
117.            if(i%interval==0)
118.            {
119.                volume.append(mySpill.getPresentVol()/mySpill.getVolInit()+"\t");
120.                evap.append(mySpill.getVolEvap()/mySpill.getVolInit()+"\t");
121.                disp.append(mySpill.getVolNatDisp()/mySpill.getVolInit()+"\t");
122.                waterCont.append(mySpill.getWaterCont()+"\t");
123.                double logVis=Math.log10(mySpill.getViscosity());
124.                visc.append(logVis+"\t");
125.                if(mySpill.getThickness()==0.0){
126.                    double logGone= -3.0;
127.                    thickness.append(logGone+"\t");
128.                    prod.append(logGone+"\t");
129.                    envir.append(logGone+"\t");
130.                    missing.append(logGone+"\t");
131.                }else{
132.                    double logVal= 6.0+Math.log10(mySpill.getThickness());
133.                    thickness.append(logVal+"\t");
```

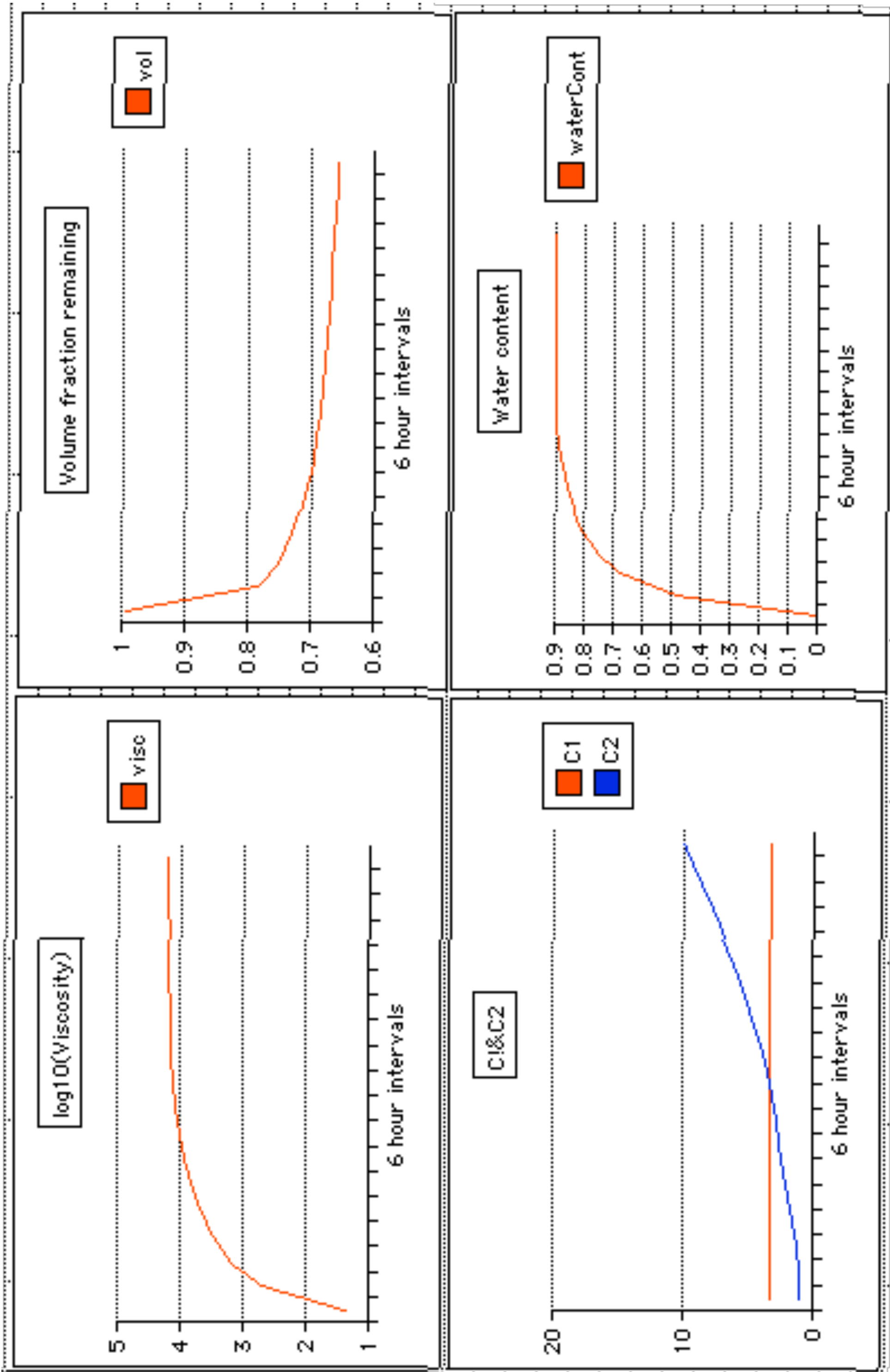
```
134.                                    logVal= 6.0+Math.log10(mySpill.getProdThickness());
135.                                    prod.append(logVal+"\t");
136.                                    logVal= 6.0+Math.log10(mySpill.getEnvirThickness());
137.                                    envir.append(logVal+"\t");
138.                                    double miss=(mySpill.getVolInit()-
    mySpill.getPresentVol())/mySpill.getArea(mySpill.getAge());
139.                                    logVal= 6.0+Math.log10(miss);
140.                                    missing.append(logVal+"\t");
141.                            }
142.                            area.append(mySpill.getArea(mySpill.getAge())+"\t");
143.                            mySpill.diag=mySpill.getSlickWidth();
144.                            areaMod.append(mySpill.getAreaMod()+"\t");
145.                            C1val.append(mySpill.getC1()+"\t");
146.                            C2val.append(mySpill.getC2()+"\t");
147.                            Diag.append(mySpill.diag+"\t");
148.                    }
149.            }
150.            volume.append("\r");
151.            evap.append("\r");
152.            disp.append("\r");
153.            waterCont.append("\r");
154.            visc.append("\r");
155.            thickness.append("\r");
156.            prod.append("\r");
157.            envir.append("\r");
158.            missing.append("\r");
159.            area.append("\r");
160.            areaMod.append("\r");
161.            System.out.println(volume);
162.            System.out.println(evap);
163.            System.out.println(disp);
164.            System.out.println(waterCont);
165.            System.out.println(visc);
166.            System.out.println(thickness);
167.            System.out.println(prod);
168.            System.out.println(envir);
169.            System.out.println(missing);
170.            System.out.println(area);
171.            System.out.println(areaMod);
172.            System.out.println(C1val);
173.            System.out.println(C2val);
174.            System.out.println(Diag);
175.            if(mySpill.areUserPseudoCompSet())System.out.print("User Defined\t");
176.            System.out.println(NPC+" PC's");
177.    }
178. }
```

## 5.3 Appendix 3 – Test Program Console (Spreadsheet) Output

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | OIL MODEL - Java | | | | | | | | | | | | | | | | | | | | |
| 2 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | | | | | | | | | | | | | | |
| 3 | [illegible] | [illegible] | [illegible] | | | | | | | | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | | | | | | | |
| 5 | vol | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 6 | evap | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 7 | disp | 0 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 8 | waterCont | 0 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 |
| 9 | visc | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 10 | oil | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 11 | prod | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 12 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 13 | missing | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 14 | area | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 15 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 16 | C1 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 17 | C2 | 1 | 1 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| 18 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |

## 5.4 Appendix 4 – Graphic output from spreadsheet





log10(Viscosity)
visc
5
4
3
2
1
6 hour intervals
Volume fraction remaining
vol
1
0.9
0.8
0.7
0.6
6 hour intervals
C!&C2
C1
C2
20
10
0
6 hour intervals
Water content
waterCont
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
6 hour intervals

## 5.5 Appendix 5 – OilWx code

```
1.  package oilWeathering;
2.  import java.text.NumberFormat;
3.  import java.util.Vector;
4.  /** =================== public interface =================
5.   * This is the basic Java class to handle oil weathering.
6.   * To use this class the calling program must create
7.   * an OilWx object using the constructor with the new
8.   * operator.
9.   * (version 1.0.3) 15-June-09
10.  * (version 1.0.4) 02-July-09 bug fix in water uptake & stepclean routine
11.  * (version 1.0.4) 10-July-09 bug fixs in visc & Nat disp routines
12.  * (version 1.0.4b) 14-July-09 fix variable wind wind spreading bug
13.  * (version 1.0.5) 03-Aug-09 fix 2nd call to SetEnvironment changing spreading
14.  * (version 2.xx1) 26-Sept-09 fix user minimum, added getSlickWidth, default to last EnOnset option
15.  *
16.  * Initialize by calls to:
17.  *             mySpill = new OilWx(API,viscosity,refTemp,Volume)
18.  *
19.  * Then call optional methods to override the default behavior
20.  *             mySpill.AddDistillationCut(cumulative%,boilingPt) // Minimum of three cuts
21.  *             mySpill.SetMaximumArea(area) // max area m2 - use for boomed spill
22.  *             mySpill.SetMousseOnsetPoint(%Evaporated) // 0 > value >100
23.  *             mySpill.SetMouseOnsetTime(emStartObserved) // age in seconds
24.  *             mySpill.SetWindRowSpacing(distBetweenWR) // m
25.  *             mySpill.SetMinThickness(responseThicknessLimit) // m
26.  *
27.  * Then call set environment method to finish initialization
28.  *             mySpill.SetEnvironment(windSpeed,waterTemp,mixedLayerDepth) int
29.  *
30.  *             => normal return is the number of  pseudo components, or:
31.  *             error -1 => mass fraction values out of order
32.  *             error -2 => boiling point values out of order
33.  *             error -3 => mass fraction data < 0%
34.  *             error -4 => mass fraction data > 100%
35.  *             error -5 => minimum boiling point < 0 degC
36.  *             error -6 => maximum boiling point > 1000 degC
37.  *             error -7 => not enough pseudo components < minMumUserPC
38.  *             error -8 => model results suspect oil may sink
39.  *
40.  * Then step through scenario by calls to either:
41.  *             mySpill.Step(stepLengthSec)
42.  *             mySpill.StepClean(stepLengthSec,stuffRecovered,oilOnlyFlag)
43.  *
44.  * Repeat calls to set environment can be used to change wind speed during a model run
45.  *
46.  * At any time during the model run state variables can be obtained by calls to:
47.  *             mySpill.getDensity()                    // kg/m2
48.  *             mySpill.getViscosity()          // cS
49.  *             mySpill.getArea()                       // m2
50.  *             mySpill.getSlickWidth()                 // m
51.  *             mySpill.getThickness()          // m thickness of oil without mousse
52.  *             mySpill.getProdThickness()   // m thickness of oil with mousse
53.  *             mySpill.getEnvirThickness()  // m thickness of oil C1*C2* mousse
54.  *             mySpill.getC1()                 // thickness coefficient (droplets)
55.  *             mySpill.getC2()                 // thickness coefficient (Langumir)
56.  *             mySpill.getWaterCont()                  // water fraction %
57.  *             mySpill.getVolInit()            // m3
58.  *             mySpill.getPresentVol()                 // m3
59.  *             mySpill.getVolEvap()            // m3
60.  *             mySpill.getVolNatDisp()                 // m3
61.  *             mySpill.getAge()                        // time (sec)
62.  *    mySpill.getWindRowSpacing()   // m
63.  *    mySpill.getMinThickness()             // m
64.  *    mySpill.isMinThickness()              // boolean
65.  *    mySpill.arePseudoCompSet()            // boolean
66.  *    mySpill.areUserPseudoCompSet()        // boolean
67.  *    mySpill.isOilSlickLeft()              // boolean
68.  *    mySpill.getAreaMod()                  // (0.0 < factor < 1.0)
```

```
69.    *
70.    * @author jag
71.    */
72.
73.
74.   /**==================== Class definition ==================
75.    *
76.    */
77.   public class OilWx {
78.
79.
80.           private double api;
81.           private double oilRho;
82.           private double refVis;
83.           private double refVisT;
84.           private double oilVis0=0.0;
85.           private double CF=0.0;
86.           private boolean pseudoComponentsSet= false;
87.           private boolean distData= false;
88.           private double expCoefRho= 0.0008; //from Adios
89.           /** class variables pseudo components
90.            */
91.           private int defaultNumPC= 10;
92.           private int numUserPC= 0; // number of user defined pseudo components
93.           private int minMumUserPC= 3;
94.           private Vector<Double> massFrac= new Vector<Double>();
95.           private Vector<Double> boilPt= new Vector<Double>();
96.           private Vector<Double> pcVol= new Vector<Double>(); // in moles
97.           private Vector<Double> pcBP= new Vector<Double>();  // boiling point k
98.           private Vector<Double> pcMV= new Vector<Double>();  // mole/m3
99.           private Vector<Double> pcMW= new Vector<Double>();
100.          private Vector<Double> pcEvapRate= new Vector<Double>();// evap rate moles/(m2 sec)
101.          private double averMW= 0.0;
102.          /** class environmental variables
103.           */
104.          private double wTemp = 10.0;
105.          private double wRho;
106.          private double wind = 5;
107.          private double mixLayer= MIX_LAYER_DEFAULT;
108.          static final double MIX_LAYER_DEFAULT= 10.0;
109.          private boolean oilRemainingFlag= true;
110.          private double v0= 1.0;  //initial volume (cubic meters)
111.          private double volumeEvap= 0.0;
112.          private double volumeDis= 0.0;
113.          private double age= 0.0;  // time since spill (sec)
114.          private boolean minThicknessFlag= true;
115.          private double minThickness= 2E-6; // dark rainbow ~ 4 wavelengths of light
116.          /** class variables controlling spreading and dispersion
117.           */
118.          private double dtDropModel= 30;
119.          private double DF=0.0; // wind dependent part of Delvigne droplet calculation
120.          private double MaxDispSize = 50E-6; // 50 microns
121.          private double DissapationFactor=0;
122.          private double areaMax= Double.MAX_VALUE; // maximum area of spill
123.          /** Mousse related class variables
124.           */
125.          private double yFraction= 0.0; // fractional water content
126.          private double S=0.0;
127.          private double Ymax= 0;
128.          private double Smax=0; // used in water up take
129.          private double Schk=0; // used in water up take
130.          private double EmOnsetFraction= 1.0;
131.          private double EmOnsetTime= Double.MAX_VALUE;
132.          private boolean mousseOnsetFromUser=false;
133.          private double dMax= 10E-6; // 10 microns
134.          private double dMin= 1E-6;  // 1 micron
135.          private double dSize= dMax; // default to initial value
136.          private double Ks= 0.0; // from Roberts notes
137.          private double Ky= 0.0; // from Roberts notes
138.          /** Gravitational spreading coefficients
```

```
139.     */
140.    private double L= 0.0;
141.    private double T0= 0.0;
142.    private double l0= 0.0;
143.    private double windDistance= 0.0;
144.    private double width= 0.0;
145.    /** public access test variable used for diagnostic output
146.     */
147.    public double diag= 0.0;
148.
149.    /** The constructor sets up an oil weathering class that
150.     * will carry out all the weathering operations and provide
151.     * output using standard Java bean get routines
152.     *
153.     * @param inAPI = API density of oil
154.     * @param inVis = initial viscosity of oil cS
155.     * @param visRefTemp = reference temperature for viscosity deg-C
156.     */
157.    public OilWx(double inAPI,double inVis, double visRefTemp,double Volume)
158.    {
159.            // set defaults for minimum oil
160.            api=inAPI;
161.            refVis=inVis;
162.            refVisT=visRefTemp+273.18; // all internal temperatures are K
163.            distData=false;
164.            v0= Volume;
165.            age= 0.0;
166.    }
167.    /** method to set or reset environmental variables
168.     * @param waterTemp = water temperature (deg C)
169.     * @param wind10 = wind speed at 10 m (m/sec)
170.     * @param mixLayerDepth = surface layer m (or OilWx.MIX_LAYER_DEFAULT)
171.     *
172.     * methods returns the number of pseudo components used or an error code
173.     */
174.    public int SetEnvironment(double wind10,double waterTemp,double mixLayerDepth)
175.    {
176.            int numberOfPC= defaultNumPC;
177.            wTemp= waterTemp+273.18; // all internal temperatures are K
178.            wind= wind10;
179.            // check for zero wind default min = 1m/sec
180.            if(wind<1.0)wind=1.0;
181.            mixLayer= mixLayerDepth;
182.            if(!pseudoComponentsSet) // only set these once
183.            {
184.                    pseudoComponentsSet= true;
185.                    if(!distData){
186.                            // set default pesudo components
187.                            SetDefaultPC(api);
188.                    }else{
189.                            // set user defined distillation data
190.                            numberOfPC= SetUserPseudoComponents();
191.                            if(numberOfPC<0)return numberOfPC;
192.                    }
193.            }
194.            if(oilVis0==0.0) // this section should only be done once
195.            {
196.                    // initialize the constant used for vis change due to evap
197.                    double Ct= 5000.0; // constant K
198.                    double fac= Ct*(1.0/wTemp-1.0/refVisT);
199.                    oilVis0= refVis*Math.exp(fac);
200.                    CF= 1.5*Math.sqrt(oilVis0);
201.                    if(CF<2.0)CF= 2.0;
202.                    if(CF>12)CF= 12.0;
203.                    // set the water density at water temperature kg/m3
204.                    wRho= 1020*(1.0-1.12E-4*(waterTemp-4.0)); // add 2% for sea water
205.                    // set the initial oil density at water temperature from api and waterTemp
206.                    oilRho= 1000*(141.5/(131.5+api));
207.                    double refRhoT= 273.18+16.0;
208.                    oilRho*= (1.0-expCoefRho*(wTemp-refRhoT));
```

```
209.                    // check for sinking oil and set error flag
210.                    if(oilRho>wRho)numberOfPC=-8;
211.                    // initialize Mousse routine constants
212.                    double dyVis= oilRho*(1E-6)*CalculateVis();
213.                    if(dyVis>0.050)
214.                    {
215.                            Ymax= 0.9-0.0952*Math.log(dyVis/0.050);
216.                    }else{
217.                            Ymax= 0.9;
218.                    }
219.                    Smax= (6.0/dMin)*Ymax/(1.0-Ymax);  // used in water up take
220.                    Schk= (6.0/dMax)*Ymax/(1.0-Ymax);  // used in water up take
221.                    // calculate evaporation threshold for Mousse formation from ADIOS
222.                    double TBP= 532.98-3.1295*api;
223.                    double TG= 1356.7-247.36*Math.log(api);
224.                    double A1est= (483.0-TBP)/TG;
225.                    if(A1est<0.0)
226.                            {A1est=0.0;}
227.                    else if(A1est>0.4)
228.                            {A1est=0.4;}
229.                    double A2est= 1.0;
230.                    if(api<26.0)
231.                            {A2est=0.08;}
232.                    else if(api>50.0)
233.                            {A2est=0.303;}
234.                    else
235.                            {A2est= -1.038-0.78935*Math.log10(1.0/api);}
236.                    if(!mousseOnsetFromUser)EmOnsetFraction= 0.5*(A1est+A2est);
237.
238.
239.                    diag=EmOnsetFraction;
240.            }
241.            double k0y= 2.042*1e-6; // used in Roberts documentation
242.            Ky=(k0y)*wind*wind;
243.            Ks=6.0*Ky/dMax;
244.            if(L==0.0) // this section should only be done once
245.            {
246.                    // set up Initial gravitational spreading constants
247.                    L= Math.pow(v0,0.3333);
248.                    double fac1=Math.pow(L,1.1428);
249.                    double fac2=Math.pow(1E-6*CalculateVis(),-.4285); //convert cS => MKS
250.                    T0= fac1*fac2;
251.                    l0= Math.pow((L*L*T0*T0), 0.3333);
252.            }
253.            // calculate natural dispersion rate per sec
254.            double d= 5.48E-3*Math.pow(wind,5);
255.            d=Math.pow(d,0.57);
256.            // wave breaking after Ming and Farmer (JPO) 1994
257.            double f= 1.18e-3*Math.pow(wind,1.03);
258.            // set low wind cut off
259.            if(wind<3.0)f=0.0;
260.            double vEntrained= Math.pow(MaxDispSize,1.7)/1.7; //need to be integral
261.            double F= (1.0-Math.exp(-f*dtDropModel));
262.            DissapationFactor= F*vEntrained;
263.            // set Delvigne droplet wind dependent term
264.            DF= d*F;
265.            //diag=numberOfPC;
266.            return numberOfPC;
267.    }
268.    //--------------------------------------------------
269.    //------ classes methods to customize weathering----
270.    //--------------------------------------------------
271.    /** Routine to add an entry to the user defined distillation cut
272.     * data.  Note that a minimum of "minMumUserPC" must be entered and
273.     * that for each component the cumMassFractio and the boilintPt
274.     * must be monotonically increasing.  After all components are
275.     * added call SetEnvironment which calls SetUserPseudoComponents
276.     */
277.    public void AddDistillationCut(double clumMassFraction,double boilingPt)
278.    {
```

```
279.                distData= true;
280.                massFrac.add(clumMassFraction);
281.                boilPt.add(boilingPt+273.0);
282.        }
283.        /** Method to set maximum area
284.         * @param area = maximum area of spill m2/ or Volume/minimumThickness m
285.         */
286.        public void SetMaximumArea(double area)
287.        {
288.                areaMax= area;
289.        }
290.        /** Method to set the onset evaporated value for mousse formation
291.         * @param evaporatedFraction
292.         */
293.        public void SetMousseOnsetPoint(double evaporatedFraction)
294.        {
295.                EmOnsetFraction= evaporatedFraction;
296.                EmOnsetTime=Double.MAX_VALUE;
297.                mousseOnsetFromUser=true;
298.        }
299.        /** Alternate method to set time for mousse formation
300.         * @param emStartObserved
301.         */
302.        public void SetMouseOnsetTime(double emStartObserved)
303.        {
304.                EmOnsetTime= emStartObserved;
305.                EmOnsetFraction=Double.MAX_VALUE;
306.                mousseOnsetFromUser=true;
307.        }
308.        public void SetWindRowSpacing(double distBetweenWR)
309.        {
310.                mixLayer= distBetweenWR/3.0;;
311.        }
312.        public void SetMinThickness(double minVal)
313.        {
314.                minThickness= minVal;
315.        }
316.        //--------------------------------------------------
317.        //-------- classes methods to step weathering-------
318.        //--------------------------------------------------
319.        /** Method steps the weathering object
320.         * @param stepLengthSec = length of time step (sec)
321.         */
322.        public void Step(double stepLengthSec)
323.        {
324.                //check if there is still any oil, if not don't step
325.                double moleSum= 0.0;  // total number of moles remaining in spill
326.                for(int i=0;i<pcVol.size();i++)
327.                {
328.                        moleSum+= pcVol.get(i);
329.                }
330.                if((moleSum==0.0)||(!oilRemainingFlag)){
331.                        oilRemainingFlag= false;
332.                        return;
333.                }
334.                // step processes
335.                StepEvaporation(stepLengthSec);
336.                StepNaturalDispersion(stepLengthSec);
337.                StepWaterContent(stepLengthSec);
338.                double windFac=0.03;
339.                if(wind>6.0)windFac+=0.005;
340.                windDistance+=windFac*wind*stepLengthSec;
341.                double chkThickness= getProdThickness()*getC1();
342.                if(chkThickness>minThickness)
343.                {
344.                        minThicknessFlag= true;
345.                }else{
346.                        minThicknessFlag= false;
347.                }
348.                age+=stepLengthSec;
```

```
349.        }
350.        /** Method steps the weathering object
351.         * @param stepLengthSec = length of time step (sec)
352.         * @param oil to be removed from last time step m3
353.         */
354.        public void StepClean(double stepLengthSec,double stuffRecovered,boolean oilOnlyFlag)
355.        {
356.                //check if there is still any oil, if not don't step
357.        if(oilRemainingFlag)
358.                {
359.                        // remove oil recovered
360.                        int pcCount= pcVol.size();
361.                        // find out how much is there
362.                        double nowVolume= getPresentVol();
363.                        // set recovered amount and discount water
364.                        double oilRecovered= stuffRecovered;
365.                        if(!oilOnlyFlag) oilRecovered*=(1.0-yFraction);
366.                        // back out moles from each pseudo component
367.                        if(oilRecovered>=nowVolume)
368.                        {
369.                                oilRemainingFlag=false;
370.                        }else{
371.                                for(int i=0;i<pcCount;i++){
372.                                        double pcFrac= pcVol.get(i)*pcMV.get(i)/nowVolume;
373.                                        pcVol.set(i,(pcVol.get(i)-pcFrac*oilRecovered/pcMV.get(i)));
374.                                }
375.                        }
376.                }
377.                Step(stepLengthSec);
378.        }
379.        //--------------------------------------------------
380.        //---------- return variable class methods ---------
381.        //--------------------------------------------------
382.        /** Method to return present density based of evap and water content
383.         *  units of kg/m3
384.         */
385.        public double getDensity()
386.        {
387.                double rhoOil=oilRho*(1.0+0.18*getVolEvap()/v0);
388.                double delRho= (wRho-rhoOil)/wRho;
389.                if(delRho<1E-4)rhoOil=wRho*(1.0-1E-4);
390.                return rhoOil*(1.0-yFraction)+wRho*yFraction;
391.        }
392.        /** Method to return present Viscosity
393.         *  units of cP
394.         */
395.        public double getViscosity()
396.        {
397.                return CalculateVis();
398.        }
399.        /** Method to return present Area
400.         *  units of m2
401.         */
402.        public double getArea(double time)
403.        {
404.                return CalculateArea(time);
405.        }
406.        /** Method to return Slick width perpendicular to the wind
407.         * units of m
408.         */
409.        public double getSlickWidth()
410.        {
411.                return width;
412.        }
413.        /** Method to return present Thickness of oil
414.         *  units of m
415.         */
416.        public double getThickness()
417.        {
418.                return CalculateVolume()/CalculateArea(age);
```

```
419.        }
420.        /** Method to return present Thickness of product oil/mousse
421.         *  units of m
422.         */
423.        public double getProdThickness()
424.        {
425.                return getThickness()/(1.0-yFraction);
426.        }
427.        /** Method to return present Thickness of
428.         *  product oil/mousse times C1 times C2
429.         *  units of m
430.         */
431.        public double getEnvirThickness()
432.        {
433.                return getC1()*getC2()*getProdThickness();
434.        }
435.        /** Method to return present C1
436.         *  base thickness for C1 calculation is product-thickness
437.         */
438.        public double getC1()
439.        {
440.                if(!isOilSlickLeft())return 0.0;
441.                double C1=1.0;
442.                // trap for low wind case
443.                if(wind<3.0)
444.                {
445.                        return C1;
446.                }
447.                // calculate the scale factor (SlickMass/DropletMass)
448.                double scale= getDensity()*getProdThickness()/CalculateDropletMass();
449.                C1= 1.0+2.2*(1.0-Math.exp(-Math.sqrt(3.0*scale)));
450.                return C1;
451.        }
452.        /** Method to return present C2
453.         *  base thickness for C2 calculation is C1*product-thickness
454.         */
455.        public double getC2()
456.        {
457.                if(!isOilSlickLeft())return 0.0;
458.                double C1= getC1();
459.                double thicknessNow= C1*getProdThickness();
460.                double val= calculateC2(wind,getDensity(),thicknessNow);
461.                return val;
462.        }
463.        /** this method returns the fraction of the spill area that has
464.         * not "sheened out"
465.         */
466.        public double getAreaMod()
467.        {
468.                return AreaMod();
469.        }
470.        /** Method to return present WaterCont
471.         *  units of %
472.         */
473.        public double getWaterCont()
474.        {
475.                return yFraction;
476.        }
477.        /** Method to return VolInit
478.         *  units of m3
479.         */
480.        public double getPresentVol()
481.        {
482.                if(!isOilSlickLeft())return 0.0;
483.                return CalculateVolume();
484.        }
485.        /** Method to return present VolInit
486.         *  units of m3
487.         */
488.        public double getVolInit()
```

```
489.	{
490.		return v0;
491.	}
492.	/** Method to return present VolEvap
493.	 *  units of m3
494.	 */
495.	public double getVolEvap()
496.	{
497.		return volumeEvap;
498.	}
499.	/** Method to return present VolNatDisp
500.	 *  units of m3
501.	 */
502.	public double getVolNatDisp()
503.	{
504.		return volumeDis;
505.	}
506.	/** Method to return present Age
507.	 *  units of sec
508.	 */
509.	public double getAge()
510.	{
511.		return age;
512.	}
513.	/** returns estimate of  wind row spacing 3*mixed layer depth
514.	 *
515.	 */
516.	public double getWindrowSpacing()
517.	{
518.		return 3.0*mixLayer;
519.	}
520.	/** returns the current value of minThickness
521.	 *
522.	 */
523.	public double getMinThickness()
524.	{
525.		return minThickness;
526.	}
527.	/** returns whether the max average area thickness is > minThickness
528.	 *
529.	 */
530.	public boolean isMinThickness()
531.	{
532.		return minThicknessFlag;
533.	}
534.	/** returns whether pseudo components are set
535.	 *
536.	 */
537.	public boolean arePseudoCompSet()
538.	{
539.		return pseudoComponentsSet;
540.	}
541.	/** returns whether user defined components are used
542.	 *
543.	 */
544.	public boolean areUserPseudoCompSet()
545.	{
546.		return distData;
547.	}
548.	/** returns whether there is any oil left in the slick
549.	 *
550.	 */
551.	public boolean isOilSlickLeft()
552.	{
553.		return oilRemainingFlag;
554.	}
555.	//-------------------------------------------------
556.	//--------------- internal class methods -----------
557.	//-------------------------------------------------
558.	/** set default pseudo components
```

```
559.         * will initialize the pseudo components
560.         * based on correlation data and the API
561.         */
562.        private void SetDefaultPC(double api)
563.        {
564.                double T0=457.16-3.3447*api; // these two formula come from Adios
565.                double dT=1356.7-247.36*Math.log(api);
566.                double sumVolMW= 0.0;
567.                double sumVol= 0.0;
568.                for(int i=0;i<defaultNumPC;i++)
569.                {
570.                        double BP=T0+dT*((double)i+0.5)/defaultNumPC; // temperature K
571.                        double MV= (7.000E-5)-(2.102E-7)*BP+(1.000E-9)*BP*BP;
572.                        double MW= 0.04132-(1.985E-4)*BP+(9.494E-7)*BP*BP;
573.                        double vi= v0/defaultNumPC;
574.                        pcVol.add(vi/MV); // moles in this cut m3/moles
575.                        pcBP.add(BP);
576.                        pcMV.add(MV);
577.                        pcMW.add(MW);
578.                        sumVolMW+= MW*vi/MV;
579.                        sumVol+= vi/MV;
580.
581.                }
582.                averMW= sumVolMW/sumVol;
583.                // calculate evaporation rate constants
584.                double K0= 0.0048*1.3676*Math.pow(0.018/averMW,0.3333)*Math.pow(wind,0.7777);
585.                double R1= 8.205E-5; // gas constant (atmos m3)/(mol K)
586.                for(int i=0;i<pcVol.size();i++)
587.                {
588.                        double vp= VaporPressure(pcBP.get(i));
589.                        double moleRate= K0*vp/(R1*wTemp); // moles/(m2 sec)
590.                        pcEvapRate.add(moleRate);
591.                }
592.        }
593.        /**
594.         * normal return is number of user defined pseudo components
595.         * error -1 => mass fraction values out of order
596.         * error -2 => boiling point values out of order
597.         * error -3 => mass fraction data < 0%
598.         * error -4 => mass fraction data > 100%
599.         * error -5 => minimum boiling point < 273 degK
600.         * error -6 => maximum boiling point > 1000 degK
601.         * error -7 => not enough pseudo components < minMumUserPC
602.         * error -8 => model results suspect oil may sink
603.         */
604.        private int SetUserPseudoComponents()
605.        {
606.                int message= massFrac.size();
607.                if(message<minMumUserPC)return -7;
608.                double maxcut= 0.0;
609.                double mincut= 100.0;
610.                double maxtemp= 0.0;
611.                double mintemp= 1000.0;
612.                double valFrac0= 0.0;
613.                double valBp0= 0.0;
614.                for(int i=0;i<massFrac.size();i++)
615.                {
616.                        double valFrac1= massFrac.get(i);
617.                        double valBp1= boilPt.get(i);
618.                        // update range
619.                        if(maxcut<valFrac1)maxcut= valFrac1;
620.                        if(mincut>valFrac1)mincut= valFrac1;
621.                        if(maxtemp<valBp1)maxtemp= valBp1;
622.                        if(mintemp>valBp1)mintemp= valBp1;
623.                        if((valFrac1-valFrac0)<0.0)return -1;
624.                        if((valBp1-valBp0)<0.0)return -2;
625.                        valFrac0= valFrac1;
626.                        valBp0= valBp1;
627.                }
628.                if(mincut<0.0)return -3;
```

```
629.			if(maxcut>100.0)return -4;
630.			if(mintemp<0.0)return -5;
631.			if(maxtemp>1000.0)return -6;
632.			numUserPC= message;
633.			double[] T= new double[numUserPC+1];
634.			T[0]=boilPt.get(0)-massFrac.get(0)*
635.				(boilPt.get(1)-boilPt.get(0))/(massFrac.get(1)-massFrac.get(0));
636.			T[numUserPC]=boilPt.get(numUserPC-1)+(100.0-massFrac.get(numUserPC-1))*
637.				(boilPt.get(numUserPC-1)-boilPt.get(1))/(massFrac.get(numUserPC-1)-massFrac.get(1));
638.			double[] M= new double[numUserPC+1];
639.			M[0]=0.0;
640.			M[numUserPC]=100.0;
641.			for(int j=1;j<numUserPC;j++)
642.			{
643.				T[j]= boilPt.get(j-1);
644.				M[j]= massFrac.get(j-1);
645.			}
646.			double sumVolMW= 0.0;
647.			double sumVol= 0.0;
648.			for (int k=1;k<numUserPC+1;k++)
649.			{
650.				double BP= 0.5*(T[k]+T[k-1]);
651.				double MV= (7.000E-5)-(2.102E-7)*BP+(1.000E-9)*BP*BP;
652.				double MW= 0.04132-(1.985E-4)*BP+(9.494E-7)*BP*BP;
653.				double vi= (M[k]-M[k-1])*v0/100.0;
654.				pcVol.add(vi/MV); // moles in this cut m3/moles
655.				pcBP.add(BP);
656.				pcMV.add(MV);
657.				pcMW.add(MW);
658.				sumVolMW+= MW*vi/MV;
659.				sumVol+= vi/MV;
660.			}
661.			averMW= sumVolMW/sumVol;
662.			// calculate evaporation rate constants
663.			double K0= 0.0048*1.3676*Math.pow(0.018/averMW,0.3333)*Math.pow(wind,0.7777);
664.			double R1= 8.205E-5; // gas constant (atmos m3)/(mol K)
665.			for(int i=0;i<pcVol.size();i++)
666.			{
667.				double vp= VaporPressure(pcBP.get(i));
668.				double moleRate= K0*vp/(R1*wTemp); // moles/(m2 sec)
669.				pcEvapRate.add(moleRate);
670.			}
671.			return message;
672.		}
673.		/** Method calculates the vapor pressure in atmospheres
674.		 */
675.		private double VaporPressure(double BP) //  vp in atmos
676.		{
677.			// loop through the pc's for evaporation equations
678.			double dZ= 0.97; // constant from Lyman
679.			double R= 1.987; // gas constant (cal)/(mol K)
680.			double dS= 8.75+R*Math.log(BP);  // Lyman 14-16 assume Kf=1.0
681.			double c2= 0.19*BP-18.0; // Lyman 14-15
682.			double T1= BP-c2;
683.			double T2= wTemp-c2;
684.			double fac1= dS*T1*T1/(dZ*R*BP);
685.			double fac2= (T2-T1)/(T2*T1);
686.			double vapP= Math.exp(fac1*fac2); // Lyman 14-14
687.			return vapP;
688.		}
689.		/** Method to step evaporation
690.		 */
691.		private void StepEvaporation(double stepLengthSec)
692.		{
693.			// calculate the sum of the pc moles
694.			double moleSum= 0.0;  // total number of moles in spill
695.			for(int i=0;i<pcVol.size();i++)
696.			{
697.				moleSum+= pcVol.get(i);
698.			}
```

```
699.            double AModCoef=1.0;
700.            double effArea= CalculateArea(age+stepLengthSec/2.0)*(1.0-yFraction)*(1.0-AModCoef*AreaMod());
701.            for(int i=0;i<pcVol.size();i++)
702.            {
703.                double downWindFactor=Math.pow(windDistance,0.1111);
704.                if(downWindFactor<1.0)downWindFactor= 1.0;
705.                double Dmi= pcEvapRate.get(i)*(pcVol.get(i)/moleSum)*effArea*stepLengthSec/downWindFactor;
706.                if(Dmi>pcVol.get(i))Dmi= pcVol.get(i);
707.                // adjust the old molar volumes
708.                pcVol.set(i, (pcVol.get(i)-Dmi));
709.                // subtract real vol to evap sum
710.                volumeEvap+=Dmi*pcMV.get(i);
711.            }
712.        }
713.        /** Method to step natural dispersion
714.         */
715.        private void StepNaturalDispersion(double stepLengthSec)
716.        {
717.            // if maximum area is restricted there is no dispersion
718.            if(areaMax!=Double.MAX_VALUE)return;
719.            // normal dispersion
720.            double AModCoef= 1.0;
721.            double massDis= CalculateDisConst()*DissapationFactor*(1.0-yFraction)
722.                            *CalculateArea(age)*(1.0-AModCoef*AreaMod())*stepLengthSec/dtDropModel; //
    kg of oil dispersed
723.
724.            // convert to volume
725.            double volDis= massDis/getDensity(); // volume of droplets
726.            int pcCount= pcVol.size();
727.            double nowVolume= getPresentVol();
728.            // back out moles from each pseudo component
729.            if(volDis>nowVolume)
730.            {
731.                oilRemainingFlag=false;
732.            }else{
733.                for(int i=0;i<pcCount;i++){
734.                    double pcFrac= pcVol.get(i)*pcMV.get(i)/nowVolume;
735.                    pcVol.set(i,(pcVol.get(i)-pcFrac*volDis/pcMV.get(i)));
736.                }
737.            }
738.            volumeDis+=volDis;
739.        }
740.        /** Method to step water content
741.         */
742.        private void StepWaterContent(double stepLengthSec)
743.        {
744.            // check if evaporation has started
745.            double evapFraction= getVolEvap()/v0;
746.            if((age>EmOnsetTime)||(evapFraction>EmOnsetFraction))
747.            {
748.                S= Smax-(Smax-S)*Math.exp(-Ks*stepLengthSec/Smax);
749.                if(S>Smax) S= Smax;
750.                // adjust the water content
751.                if(S<Schk){
752.                    yFraction= S*dMax/(6.0+S*dMax); // water content not max yet
753.                    dSize= dMax;
754.                }else{
755.                    yFraction= Ymax;
756.                    dSize= (6.0/S)*yFraction/(1.0-yFraction);
757.                }
758.                if(yFraction>1.0)yFraction=1.0;
759.            }
760.
761.
762.        }
763.        /** Method to calculate viscosity based on temperature
764.         * of water correction, fraction evaporated correction
765.         * and water content correction
766.         */
767.        private double CalculateVis()
```

```
768.        {
769.                // calculate the change due to fraction evap
770.                double fractionEvap= volumeEvap/v0;
771.                double vis= oilVis0*Math.exp(CF*fractionEvap);
772.                // calculate the change due to water content
773.                // coefficient 0.2 set based on Adios results
774.                double Cy= 0.50*(1.0+0.2*dMin/dSize);
775.                if(Cy>0.85)Cy= 0.85;
776.                if(Cy<0.35)Cy= 0.45;
777.                double fac= 2.5*yFraction/(1.0-Cy*yFraction);
778.                vis= vis*Math.exp(fac);
779.                return vis;
780.        }
781.        /** Method to calculate basic spill area based
782.         * on inital Fay and then Richarson/Batchlor/Obuko
783.         * plus droplet wind spreading
784.         */
785.        private double CalculateArea(double time)
786.        {
787.                double l1= 0;
788.                if(time<T0) // still gravitational spreading
789.                {
790.                        l1= Math.pow(L*L*time*time,0.333);
791.                }else //Richardson/Batchelor/Obuko
792.                {
793.                        double fac0= Math.pow(100*l0, 0.6666)+(0.006*(time-T0)); // convert to cm to use Obuko const
794.                        l1= 0.01*Math.pow(fac0, 1.5); // convert back to m now that l1 is to the first power
795.                }
796.                // correct the downwind axis for droplet drift
797.                width= l1;
798.                double l2= l1 + windDistance;
799.                double area= l1*l2;
800.                // check if constraints override spreading
801.                if(area>areaMax)area= areaMax;
802.                return area;
803.        }
804.        /** this method calculates a  reduction factor (0.0-1.0)in the area do to
805.         * oil loss in the thin areas
806.         * @return
807.         */
808.        private double AreaMod()
809.        {
810.                double ex= getC1()*getC2();
811.                double fac=(v0-getPresentVol())/v0;
812.                if((ex<1.)||(fac<1e-6))return 0.0; // round off error bug fixed 13-VI-09
813.                double x= Math.pow(fac/ex, 1/(ex-1.0));
814.                return x;
815.        }
816.        /** Method to return present physical volume
817.         * relative initial 1.0
818.         */
819.        private double CalculateVolume()
820.        {
821.                if(!isOilSlickLeft())return 0;
822.                double presentVol= 0.0;
823.                for(int i=0;i<pcVol.size();i++)
824.                {
825.                        presentVol+= pcVol.get(i)*pcMV.get(i);
826.                }
827.                if(presentVol>v0)presentVol= v0; // model can not add to initial vol
828.                return presentVol;
829.        }
830.        /** Method to set disapation factor C0 ref. Delvigne
831.         */
832.        private double CalculateDisConst()
833.        {
834.                // use method defined by Roy's formula
835.                return 2400.0*Math.exp(-73.682*Math.sqrt(1E-6*CalculateVis()));
836.        }
837.        /** method to calculate C2 using analytic fit to Roy's Langmuir Model
```

```
838.       * modified June 4,2009 to set high and low thickness  limits
839.       */
840.      private double calculateC2(double wind,double rho0,double thickness0)
841.      {
842.              // convert thickness in meters to mm
843.              double thick= 1000*thickness0;
844.              if(thick>1.0)thick=1.0;
845.              // default low wind cut off
846.              if(wind<3.0)return 1.0;
847.              double x= 2.0*(rho0-800.0)/(wRho-800.0);
848.              double cLow;
849.              double nLow;
850.              double cHigh;
851.              double nHigh;
852.              double c;
853.              double n;
854.              double val;
855.              if(wind<5.0)
856.              {
857.                      double wCoef=(wind-3.0)/2.0;
858.                      cLow= 0.8851+x*(-0.4312+x* 0.6034);
859.                      nLow= - 0.8263+x*( 0.0466+x*-0.0432);
860.                      cHigh= 2.3015+x*(-0.6293+x* 1.2263);
861.                      nHigh= - 0.7527+x*(-0.0050+x*-0.0183);
862.                      c=cHigh*wCoef+cLow*(1.0-wCoef);
863.                      n=nHigh*wCoef+nLow*(1.0-wCoef);
864.                      val= c*Math.pow(thick,n);
865.              }
866.              else if(wind<8.0)
867.              {
868.                      double wCoef=(wind-5.0)/3.0;
869.                      cLow= 2.3015+x*(-0.6293+x* 1.2263);
870.                      nLow= - 0.7527+x*(-0.0050+x*-0.0183);
871.                      cHigh= 5.1184+x*(-1.0233+x* 2.1188);
872.                      nHigh= - 0.6799+x*(-0.0044+x*-0.0092);
873.                      c=cHigh*wCoef+cLow*(1.0-wCoef);
874.                      n=nHigh*wCoef+nLow*(1.0-wCoef);
875.                      val= c*Math.pow(thick,n);
876.              }
877.              else if(wind<10.0)
878.              {
879.                      double wCoef=(wind-8.0)/2.0;
880.                      cLow= 5.1184+x*(-1.0233+x* 2.1188);
881.                      nLow= - 0.6799+x*(-0.0044+x*-0.0092);
882.                      cHigh= 6.9004+x*(-1.2544+x* 2.6376);
883.                      nHigh= - 0.6600+x*(-0.0034+x*-0.0069);
884.                      c=cHigh*wCoef+cLow*(1.0-wCoef);
885.                      n=nHigh*wCoef+nLow*(1.0-wCoef);
886.                      val= c*Math.pow(thick,n);
887.              }
888.              // default wind > 10 m/s
889.              else
890.              {
891.                      c= 6.9004+x*(-1.2544+x* 2.6376);
892.                      n= - 0.6600+x*(-0.0034+x*-0.0069);
893.                      val= c*Math.pow(thick,n);
894.              }
895.              if(val>80.0)val=80.0; // low thickness cutoff
896.              if(thickness0>0.001)val=val*Math.exp(-(thickness0-0.001)/0.001);// high thickness cutoff
897.              // back out C1 value
898.              val= val/getC1();
899.              if(val<1.0)val=1.0;
900.              return val;
901.      }
902.      /** method to calculate the Delvigne potntial droplet mass
903.       *  depending on the resent state of the oil's density,viscosity
904.       *  and environmental wind speed
905.       */
906.      private double CalculateDropletMass()
907.      {
```

```
908.                    // start with viscous and wind dependen parts
909.                    double dm0= CalculateDisConst()*DF;
910.                    // calculate droplet buoyancy contribution
911.                    double Wmix= 1.5*(0.026*Math.pow(wind,2.5));
912.                    double dt= 30; // time step
913.                    double dcn= 20; // number of mixing droplet classes
914.                    double d1= 0.0002; // upper limit for Stoke's drift
915.                    double d2= 0.01; // lower limit for Form Drag
916.                    double g= 9.8; // acceleration of gravity
917.                    double wVisc= 1.3e-6; // water viscosity
918.                    double dfac= g*(wRho-getDensity())/wRho;
919.                    // set parameters for droplet size classes
920.                    double tempY= yRise(dfac,d1,d2,wVisc,Wmix/dt); // droplet size to rise 1/2 Wmix in 1/2 dt
921.                    double dmax= tempY*(d2-d1)+d1;
922.                    double deld= dmax/dcn;
923.                    // add buoyancy factor
924.                    dm0*= deld*Math.pow(dmax,1.7)/1.7;
925.                    return dm0;
926.            }
927.            //================================================
928.            // calculate rise velocity stuff
929.            // dfac= g*(rhoWater-rhoOil)/(rhoWater) buoyance term
930.            // d1= 0.0002  upper limit for Stoke,s law cut
931.            // d2= 0.01  lower limit for Form drag
932.            // wVisc= 1.3e-6 water viscosity in mks units
933.            // y=(dia-d1)/(d2-d1) droplet diameter scaled to (d2-d1) interval
934.            //================================================
935.            /** following three methods are helpers to calculate the droplet
936.             * rise velocities used in scaling the droplet model
937.             */
938.            private double riseVel(double dfac,double d1,double d2,double wVisc,double y)
939.            {
940.                    if(y<0.0) // special case where d is in Stokes' drift range
941.                    {
942.                            double ds= y*(d2-d1)+d1;
943.                            return dfac*ds*ds/(18*wVisc);
944.                    }
945.                    if(y>=1.0) // special case where d is in form drag range
946.                    {
947.                            double df= y*(d2-d1)+d1;
948.                            return Math.sqrt(8.0*dfac*df/3.0);
949.                    }
950.                    double a1= dfac*d1*d1/(18*wVisc);
951.                    double a2= a1*(d2-d1)/(2.0*d1);
952.                    double a3= Math.sqrt(8.0*dfac*d2/3.0);
953.                    double a4= a3*(d2-d1)/(2.0*d2);
954.                    double c3= 2.0*a1+a2-2.0*a3+a4;
955.                    double c2= -3.0*a1-2.0*a2+3.0*a3-a4;
956.                    double c1= a2;
957.                    double c0= a1;
958.                    return ((c3*y+c2)*y+c1)*y+c0;
959.            }
960.            private double riseVelP(double dfac,double d1,double d2,double wVisc,double y)
961.            {
962.                    if(y<0.0) // special case where d is in stokes drift range
963.                    {
964.                            double ds= y*(d2-d1)+d1;
965.                            return 2.0*dfac*ds/(18*wVisc);
966.                    }
967.                    if(y>=1.0) // special case where d is in form drag range
968.                    {
969.                            double df= y*(d2-d1)+d1;
970.                            return Math.sqrt(8.0*dfac*df/3.0)/(2.0*df);
971.                    }
972.                    double a1= dfac*d1*d1/(18*wVisc);
973.                    double a2= a1*(d2-d1)/(2.0*d1);
974.                    double a3= Math.sqrt(8.0*dfac*d2/3.0);
975.                    double a4= a3*(d2-d1)/(2.0*d2);
976.                    double c3= 2.0*a1+a2-2.0*a3+a4;
977.                    double c2= -3.0*a1-2.0*a2+3.0*a3-a4;
```

```
978.                double c1= a2;
979.                return (3.0*c3*y+2.0*c2)*y+c1;
980.        }
981.        private double yRise(double dfac,double d1,double d2,double wVisc,double w0)
982.        {
983.                double y0= 0.5;
984.                double y1= 1.0;;
985.                double chk=1.0;
986.                int i= 0;
987.                while(chk>0.00005)
988.                {
989.                        y1= y0+(w0-riseVel(dfac,d1,d2,wVisc,y0))/riseVelP(dfac,d1,d2,wVisc,y0);
990.                        chk= Math.abs(y1-y0);
991.                        y0= y1;
992.                        i++;
993.                        if(i>25)break;
994.                }
995.                return y1;
996.        }
997. }
```