Ecosystem

Version 1.0.0

Generated by Doxygen 1.8.13

Contents

1 Introduction

1.1 Copyright Statement

Copyright

1.2 General Information

The source code contained within the ecosystem project was designed as a standalone tool set for performing numerical modeling and data analyses associated with adsorption phenomena in both gaseous and aqueous systems. Many of the lower level tools are general enough to be applied to any system you desire to be modeled. Such algorithms included are Krylov subspace methods for linear systems and a Jacobian-Free Newton-Krylov method for non-linear systems. There is also a templated matrix object for generic matrix construction and modification. For specific information on each individual kernel, navigate through the class and file indices or table of contents.

Warning

2 Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Class Documentation

5.1 ActivityDistribution Class Reference

C++ Object for determining the activity-size distribution.

```
#include <kea.h>
```

Public Member Functions

- ActivityDistribution ()
  Default constructor.
- ∼ActivityDistribution ()
  Default destructor.
- void set_model_type (asd_model type)
  Set the model_type parameter.
- void set_capfis_ratio (double val)
  Set the capfis_ratio parameter.
- void set_neutrons_emit (double val)
  Set the neutrons_emit parameter.

- void set_fusion_yield (double val)

  Set the fusion_yield parameter.
- void set_fission_yield (double val)

  Set the fission_yield parameter.
- void set_total_yield (double val)

  Set the total_yield parameter.
- void set_casing_cap (double val)

  Set the casing_cap parameter.
- void set_casing_den (double val)

  Set the casing_den parameter.
- void set_casing_thickness (double val)

  Set the casing_thickness parameter.
- void set_casing_mw (double val)

  Set the casing_mw parameter.
- void set_casing_thermal (double val)

  Set the casing_thermal parameter.
- void set_soil_thermal (double val)

  Set the soil_thermal parameter.
- void set_soil_scattering (double val)

  Set the soil_scattering parameter.
- void set_weapon_thermal (double val)

  Set the weapon_thermal parameter.
- void set_size_cutoff (double val)

  Set the size_cutoff parameter.
- void set_burst_height (double val)

  Set the burst_height parameter.
- void set_escape_fraction (double val)

  Set the escape_fraction parameter.
- void set_volatile_fraction (double val)

  Set the volatile_fraction parameter.
- void set_soil_capture_fraction (double val)

  Set the soil_capture_fraction parameter.
- void delete_casing_components ()

  Function to remove all casing components and parameters.
- void add_casing_component (std::string name, double frac)

  Function to add casing components and corresponding molefraction.
- void verify_casing_components ()

  Function to check casing components for errors and correct.
- void delete_fractionation ()

  Function to remove all maps with isotope fractionation.
- void delete_capture_fractions ()

  Function to delete all maps with neutron capture fractions.
- asd_model get_model_type ()

  Get the model_type parameter.
- double get_capfis_ratio ()

  Get the capfis_ratio parameter.
- double get_neutrons_emit ()

  Get the neutrons_emit parameter.
- double get_fusion_yield ()

  Get the fusion_yield parameter.
- double get_fission_yield ()

Get the fission_yield parameter.

- double get_total_yield ()

  Get the total_yield parameter.
- double get_casing_cap ()

  Get the casing_cap parameter.
- double get_casing_den ()

  Get the casing_den parameter.
- double get_casing_thickness ()

  Get the casing_thickness parameter.
- double get_casing_mw ()

  Get the casing_mw parameter.
- double get_casing_thermal ()

  Get the casing_thermal parameter.
- double get_soil_thermal ()

  Get the soil_thermal parameter.
- double get_soil_scattering ()

  Get the soil_scattering parameter.
- double get_weapon_thermal ()

  Get the weapon_thermal parameter.
- double get_size_cutoff ()

  Get the size_cutoff parameter.
- double get_burst_height ()

  Get the burst_height parameter.
- double get_escape_fraction ()

  Get the escape_fraction parameter.
- double get_volatile_fraction ()

  Get the volatile_fraction parameter.
- double get_soil_capture_fraction ()

  Get the soil_capture_fraction parameter.
- FissionProducts & getNuclides (double size_k)

  Returns reference to the FissionProducts object for the kth size class.
- double getRadioactivity (double size_k)

  Returns the radioactivity of the kth particle size class.
- std::map< double, FissionProducts > & getNuclideDistributionMap ()

  Returns reference to the Nuclide Size Distribution Map.
- std::map< double, double > & getRadioactivityDistributionMap ()

  Returns reference to the radioactivity-size distribution map.
- void compute_neutrons_emit (double fission, double fusion)

  Compute the neutrons emitted per fission (atoms/fission)
- void compute_casing_mw ()

  Compute the casing MW (must have initialized materials first)
- void compute_casing_thermal ()

  Compute the casing thermal capture cross-section.
- void compute_soil_thermal (std::map< std::string, double > &soil_atom_frac, std::map< std::string, Atom > &soil_atom)

  Compute the soil thermal capture cross-section.
- void compute_soil_scattering (std::map< std::string, double > &soil_atom_frac, std::map< std::string, Atom > &soil_atom)

  Compute the soil scattering cross-section.
- void compute_weapon_thermal (FissionProducts &weapon)

  Compute the weapon thermal capture cross-section.

- void compute_casing_capfrac ()

    Functions below compute all the capture fractions and will call the above functions for thermal and scattering info.
- void compute_soil_capfrac (std::map< std::string, double > &soil_atom_frac, std::map< std::string, Atom > &soil_atom)
- void compute_weapon_capfrac (FissionProducts &weapon)
- void compute_casing_cap ()

    Compute the casing capture fraction (Sigma)
- void compute_escape_fraction ()

    Compute the casing escape fraction.
- void compute_volatile_fraction (double h, double W)

    Compute the neutron fraction entering soil given h (m) and W (kT)
- void compute_soil_capture_fraction (std::map< std::string, double > &soil_atom_frac, std::map< std::string, Atom > &soil_atom)

    Compute the fraction of neutrons captured by soil.
- void initialize_fractionation (FissionProducts &yields, yaml_cpp_class &data)

    Initialize fractionation via yield data.
- int evaluate_initial_fractionation ()

    Go through all soil, casing, and weapon data to add nuclides.
- void evaluate_freiling_ratios (double solid_time, double solid_temp)

    Evaluate the initial fractionation to the solidification time.
- void evalute_freiling_dist (std::map< double, double > &part_conc)

    Evaluate the freiling distribution.
- void evalute_freiling_tompkins_dist (std::map< double, double > &part_hist)

    Evaluate the freiling-tompkins distribution.
- void evalute_mod_freiling_dist (std::map< double, double > &part_hist)

    Evaluate the modified freiling distribution.
- void evalute_mod_freiling_tompkins_dist (std::map< double, double > &part_hist)

    Evaluate the modified freiling-tompkins distribution.
- void evalute_distribution (std::map< double, double > &part_conc, std::map< double, double > &part_hist)

    Call and evaluate the appropriate distribution function.
- void distribute_nuclides (std::map< double, double > &part_hist)

    Distribute nuclides on particle sizes according to distribution fractions.
- void evaluate_fractionation (std::string file_name, bool file_out, double solid_time, double stab_time)

    Run fractionation simulation to cloud stabilization time and print results to file.
- void simulate_fractionation (double start_time, double end_time)

    Performs a simulation between two time events and records the results (no file output)

Protected Attributes

- asd_model model_type

    Type of activity-size distribution model to use.
- double capfis_ratio

    Neutron capture-to-fission ratio for induced activity.
- double neutrons_emit

    Below are all the parameters associated with the induced-soil-activity models.
- double fusion_yield

    Fusion yield in kT (Wfu)
- double fission_yield

    Fission yield in kT (Wfis)
- double total_yield

Total weapon yield in kT (W)
- double casing_cap

  Weapon casing capture (Sigma)
- double casing_den

  Weapon casing material density in g/cm^3 (rho_c)
- double casing_thickness

  Weapon casing material thickness in cm (X)
- double casing_mw

  Weapon casing average molecular weight in g/mol (A)
- double casing_thermal

  Weapon casing average thermal neutron x-sec in barns (sigma_c)
- double soil_thermal

  Soil material average thermal neutron x-sec in barns (sigma_s)
- double soil_scattering

  Soil material average neutron scattering in barns (sigma_ssc)
- double weapon_thermal

  Weapone material thermal neutron x-sec in barns.
- double size_cutoff

  Size cutoff point for the Freiling-Tompkins distributions in um (D)
- std::map< std::string, Atom > casing_atom

  Stores a map of casing atom components (key is the atom)
- std::map< std::string, double > casing_atom_frac

  Stores a map of casing atom components (key is the atom)
- std::map< std::string, Molecule > casing_mat

  Weapon casing molecular composition.
- std::map< std::string, double > casing_frac

  Weapon casing molefractions.
- double burst_height

  Weapon burst height above ground (ft)
- double escape_fraction

  Neutron fraction that escapes casing (e^-Sigma∗X)
- double volatile_fraction

  Neutron fraction that enters volatilized soil.
- double soil_capture_fraction

  Neutron fraction that is captured by soil.
- FissionProducts initial_frac

  Below are the parameters associated with the activity-size distributions.
- std::map< double, FissionProducts > nuc_fractionation

  Fractionation of nuclides with particle size (um)
- std::map< double, double > radioactivity_dist

  Distribution of radioactivity (in Bq) with particle size (um)
- std::map< int, double > total_moles

  Total moles of nuclides for each mass number chain.
- std::map< int, double > refractory_moles

  Refractory moles of nuclides for each mass number chain.
- std::map< int, double > freiling_numbers

  Map of the Freiling ratio numbers (b_i)
- std::unordered_map< int, std::unordered_map< double, double > > distribution

  Map of the normalized distributions for nuclides [i] ==> mass num [k] ==> particle size.
- std::map< std::string, double > casing_capfrac

  Neutron fractional captures by all atomic materials.

- std::map< std::string, double > soil_capfrac

  Soil neutron capture fractions (by atom symbol)
- std::map< std::string, double > weapon_capfrac

  Weapon neutron capture fractions (by isotope name)

5.1.1 Detailed Description

C++ Object for determining the activity-size distribution.

This object inherits from FissionProducts and will determine the activity-size distributions of nuclides in a nuclear debris cloud. It will also be used to determine the induced activity in soil particles and from weapon material absorbtion of neutrons. While this kernel is developed independently from CRANE, it will be coupled with the size distributions and other parameters from CRANE. Then CRANE and KEA will be implemented together to fully describe the nuclear debris cloud post-detonation and to the time of cloud stabilization.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 ActivityDistribution()

```
ActivityDistribution::ActivityDistribution ( )
```

Default constructor.

5.1.2.2 ∼ActivityDistribution()

```
ActivityDistribution::∼ActivityDistribution ( )
```

Default destructor.

5.1.3 Member Function Documentation

5.1.3.1 set_model_type()

```
void ActivityDistribution::set_model_type (
            asd_model type )
```

Set the model_type parameter.

5.1.3.2 set_capfis_ratio()

```
void ActivityDistribution::set_capfis_ratio (
             double val )
```

Set the capfis_ratio parameter.

5.1.3.3 set_neutrons_emit()

```
void ActivityDistribution::set_neutrons_emit (
             double val )
```

Set the neutrons_emit parameter.

5.1.3.4 set_fusion_yield()

```
void ActivityDistribution::set_fusion_yield (
             double val )
```

Set the fusion_yield parameter.

5.1.3.5 set_fission_yield()

```
void ActivityDistribution::set_fission_yield (
             double val )
```

Set the fission_yield parameter.

5.1.3.6 set_total_yield()

```
void ActivityDistribution::set_total_yield (
             double val )
```

Set the total_yield parameter.

5.1.3.7 set_casing_cap()

```
void ActivityDistribution::set_casing_cap (
             double val )
```

Set the casing_cap parameter.

5.1.3.8 set_casing_den()

```
void ActivityDistribution::set_casing_den (
             double val )
```

Set the casing_den parameter.

5.1.3.9 set_casing_thickness()

```
void ActivityDistribution::set_casing_thickness (
             double val )
```

Set the casing_thickness parameter.

5.1.3.10 set_casing_mw()

```
void ActivityDistribution::set_casing_mw (
             double val )
```

Set the casing_mw parameter.

5.1.3.11 set_casing_thermal()

```
void ActivityDistribution::set_casing_thermal (
             double val )
```

Set the casing_thermal parameter.

5.1.3.12 set_soil_thermal()

```
void ActivityDistribution::set_soil_thermal (
             double val )
```

Set the soil_thermal parameter.

5.1.3.13 set_soil_scattering()

```
void ActivityDistribution::set_soil_scattering (
             double val )
```

Set the soil_scattering parameter.

5.1.3.14 set_weapon_thermal()

```
void ActivityDistribution::set_weapon_thermal (
             double val )
```

Set the weapon_thermal parameter.

5.1.3.15 set_size_cutoff()

```
void ActivityDistribution::set_size_cutoff (
             double val )
```

Set the size_cutoff parameter.

5.1.3.16 set_burst_height()

```
void ActivityDistribution::set_burst_height (
             double val )
```

Set the burst_height parameter.

5.1.3.17 set_escape_fraction()

```
void ActivityDistribution::set_escape_fraction (
             double val )
```

Set the escape_fraction parameter.

5.1.3.18 set_volatile_fraction()

```
void ActivityDistribution::set_volatile_fraction (
             double val )
```

Set the volatile_fraction parameter.

5.1.3.19 set_soil_capture_fraction()

```
void ActivityDistribution::set_soil_capture_fraction (
             double val )
```

Set the soil_capture_fraction parameter.

5.1.3.20 delete_casing_components()

```
void ActivityDistribution::delete_casing_components ( )
```

Function to remove all casing components and parameters.

5.1.3.21 add_casing_component()

```
void ActivityDistribution::add_casing_component (
             std::string name,
             double frac )
```

Function to add casing components and corresponding molefraction.

5.1.3.22 verify_casing_components()

```
void ActivityDistribution::verify_casing_components ( )
```

Function to check casing components for errors and correct.

5.1.3.23 delete_fractionation()

```
void ActivityDistribution::delete_fractionation ( )
```

Function to remove all maps with isotope fractionation.

5.1.3.24 delete_capture_fractions()

```
void ActivityDistribution::delete_capture_fractions ( )
```

Function to delete all maps with neutron capture fractions.

5.1.3.25 get_model_type()

```
asd_model ActivityDistribution::get_model_type ( )
```

Get the model_type parameter.

5.1.3.26 get_capfis_ratio()

```
double ActivityDistribution::get_capfis_ratio ( )
```

Get the capfis_ratio parameter.

5.1.3.27 get_neutrons_emit()

```
double ActivityDistribution::get_neutrons_emit ( )
```

Get the neutrons_emit parameter.

5.1.3.28 get_fusion_yield()

```
double ActivityDistribution::get_fusion_yield ( )
```

Get the fusion_yield parameter.

5.1.3.29 get_fission_yield()

```
double ActivityDistribution::get_fission_yield ( )
```

Get the fission_yield parameter.

5.1.3.30 get_total_yield()

```
double ActivityDistribution::get_total_yield ( )
```

Get the total_yield parameter.

5.1.3.31 get_casing_cap()

```
double ActivityDistribution::get_casing_cap ( )
```

Get the casing_cap parameter.

5.1.3.32 get_casing_den()

```
double ActivityDistribution::get_casing_den ( )
```

Get the casing_den parameter.

5.1.3.33 get_casing_thickness()

```
double ActivityDistribution::get_casing_thickness ( )
```

Get the casing_thickness parameter.

5.1.3.34 get_casing_mw()

```
double ActivityDistribution::get_casing_mw ( )
```

Get the casing_mw parameter.

5.1.3.35 get_casing_thermal()

```
double ActivityDistribution::get_casing_thermal ( )
```

Get the casing_thermal parameter.

5.1.3.36 get_soil_thermal()

```
double ActivityDistribution::get_soil_thermal ( )
```

Get the soil_thermal parameter.

5.1.3.37 get_soil_scattering()

```
double ActivityDistribution::get_soil_scattering ( )
```

Get the soil_scattering parameter.

5.1.3.38 get_weapon_thermal()

```
double ActivityDistribution::get_weapon_thermal ( )
```

Get the weapon_thermal parameter.

5.1.3.39 get_size_cutoff()

```
double ActivityDistribution::get_size_cutoff ( )
```

Get the size_cutoff parameter.

5.1.3.40 get_burst_height()

```
double ActivityDistribution::get_burst_height ( )
```

Get the burst_height parameter.

5.1.3.41 get_escape_fraction()

```
double ActivityDistribution::get_escape_fraction ( )
```

Get the escape_fraction parameter.

5.1.3.42 get_volatile_fraction()

```
double ActivityDistribution::get_volatile_fraction ( )
```

Get the volatile_fraction parameter.

5.1.3.43 get_soil_capture_fraction()

```
double ActivityDistribution::get_soil_capture_fraction ( )
```

Get the soil_capture_fraction parameter.

5.1.3.44 getNuclides()

```
FissionProducts& ActivityDistribution::getNuclides (
             double size_k )
```

Returns reference to the FissionProducts object for the kth size class.

5.1.3.45 getRadioactivity()

```
double ActivityDistribution::getRadioactivity (
             double size_k )
```

Returns the radioactivity of the kth particle size class.

5.1.3.46 getNuclideDistributionMap()

```
std::map<double, FissionProducts>& ActivityDistribution::getNuclideDistributionMap ( )
```

Returns reference to the Nuclide Size Distribution Map.

5.1.3.47 getRadioactivityDistributionMap()

```
std::map<double, double>& ActivityDistribution::getRadioactivityDistributionMap ( )
```

Returns reference to the radioactivity-size distribution map.

5.1.3.48 compute_neutrons_emit()

```
void ActivityDistribution::compute_neutrons_emit (
             double fission,
             double fusion )
```

Compute the neutrons emitted per fission (atoms/fission)

5.1.3.49 compute_casing_mw()

```
void ActivityDistribution::compute_casing_mw ( )
```

Compute the casing MW (must have initialized materials first)

5.1.3.50 compute_casing_thermal()

```
void ActivityDistribution::compute_casing_thermal ( )
```

Compute the casing thermal capture cross-section.

5.1.3.51 compute_soil_thermal()

```
void ActivityDistribution::compute_soil_thermal (
             std::map< std::string, double > & soil_atom_frac,
             std::map< std::string, Atom > & soil_atom )
```

Compute the soil thermal capture cross-section.

5.1.3.52 compute_soil_scattering()

```
void ActivityDistribution::compute_soil_scattering (
            std::map< std::string, double > & soil_atom_frac,
            std::map< std::string, Atom > & soil_atom )
```

Compute the soil scattering cross-section.

5.1.3.53 compute_weapon_thermal()

```
void ActivityDistribution::compute_weapon_thermal (
            FissionProducts & weapon )
```

Compute the weapon thermal capture cross-section.

5.1.3.54 compute_casing_capfrac()

```
void ActivityDistribution::compute_casing_capfrac ( )
```

Functions below compute all the capture fractions and will call the above functions for thermal and scattering info.

5.1.3.55 compute_soil_capfrac()

```
void ActivityDistribution::compute_soil_capfrac (
            std::map< std::string, double > & soil_atom_frac,
            std::map< std::string, Atom > & soil_atom )
```

5.1.3.56 compute_weapon_capfrac()

```
void ActivityDistribution::compute_weapon_capfrac (
            FissionProducts & weapon )
```

5.1.3.57 compute_casing_cap()

```
void ActivityDistribution::compute_casing_cap ( )
```

Compute the casing capture fraction (Sigma)

5.1.3.58 compute_escape_fraction()

```
void ActivityDistribution::compute_escape_fraction ( )
```

Compute the casing escape fraction.

5.1.3.59 compute_volatile_fraction()

```
void ActivityDistribution::compute_volatile_fraction (
            double h,
            double W )
```

Compute the neutron fraction entering soil given h (m) and W (kT)

5.1.3.60 compute_soil_capture_fraction()

```
void ActivityDistribution::compute_soil_capture_fraction (
            std::map< std::string, double > & soil_atom_frac,
            std::map< std::string, Atom > & soil_atom )
```

Compute the fraction of neutrons captured by soil.

5.1.3.61 initialize_fractionation()

```
void ActivityDistribution::initialize_fractionation (
            FissionProducts & yields,
            yaml_cpp_class & data )
```

Initialize fractionation via yield data.

5.1.3.62 evaluate_initial_fractionation()

```
int ActivityDistribution::evaluate_initial_fractionation ( )
```

Go through all soil, casing, and weapon data to add nuclides.

5.1.3.63 evaluate_freiling_ratios()

```
void ActivityDistribution::evaluate_freiling_ratios (
            double solid_time,
            double solid_temp )
```

Evaluate the initial fractionation to the solidification time.

5.1.3.64 evalute_freiling_dist()

```
void ActivityDistribution::evalute_freiling_dist (
            std::map< double, double > & part_conc )
```

Evaluate the freiling distribution.

5.1.3.65 evalute_freiling_tompkins_dist()

```
void ActivityDistribution::evalute_freiling_tompkins_dist (
            std::map< double, double > & part_hist )
```

Evaluate the freiling-tompkins distribution.

5.1.3.66 evalute_mod_freiling_dist()

```
void ActivityDistribution::evalute_mod_freiling_dist (
            std::map< double, double > & part_hist )
```

Evaluate the modified freiling distribution.

5.1.3.67 evalute_mod_freiling_tompkins_dist()

```
void ActivityDistribution::evalute_mod_freiling_tompkins_dist (
            std::map< double, double > & part_hist )
```

Evaluate the modified freiling-tompkins distribution.

5.1.3.68 evalute_distribution()

```
void ActivityDistribution::evalute_distribution (
            std::map< double, double > & part_conc,
            std::map< double, double > & part_hist )
```

Call and evaluate the appropriate distribution function.

5.1.3.69 distribute_nuclides()

```
void ActivityDistribution::distribute_nuclides (
            std::map< double, double > & part_hist )
```

Distribute nuclides on particle sizes according to distribution fractions.

5.1.3.70 evaluate_fractionation()

```
void ActivityDistribution::evaluate_fractionation (
             std::string file_name,
             bool file_out,
             double solid_time,
             double stab_time )
```

Run fractionation simulation to cloud stabilization time and print results to file.

5.1.3.71 simulate_fractionation()

```
void ActivityDistribution::simulate_fractionation (
             double start_time,
             double end_time )
```

Performs a simulation between two time events and records the results (no file output)

5.1.4 Member Data Documentation

5.1.4.1 model_type

```
asd_model ActivityDistribution::model_type  [protected]
```

Type of activity-size distribution model to use.

5.1.4.2 capfis_ratio

```
double ActivityDistribution::capfis_ratio  [protected]
```

Neutron capture-to-fission ratio for induced activity.

5.1.4.3 neutrons_emit

```
double ActivityDistribution::neutrons_emit  [protected]
```

Below are all the parameters associated with the induced-soil-activity models.

Neutrons emitted per fission in atoms/fission (No)

5.1.4.4 fusion_yield

```
double ActivityDistribution::fusion_yield  [protected]
```

Fusion yield in kT (Wfu)

5.1.4.5 fission_yield

```
double ActivityDistribution::fission_yield  [protected]
```

Fission yield in kT (Wfis)

5.1.4.6 total_yield

```
double ActivityDistribution::total_yield  [protected]
```

Total weapon yield in kT (W)

5.1.4.7 casing_cap

```
double ActivityDistribution::casing_cap  [protected]
```

Weapon casing capture (Sigma)

5.1.4.8 casing_den

```
double ActivityDistribution::casing_den  [protected]
```

Weapon casing material density in g/cm^3 (rho_c)

5.1.4.9 casing_thickness

```
double ActivityDistribution::casing_thickness  [protected]
```

Weapon casing material thickness in cm (X)

5.1.4.10 casing_mw

```
double ActivityDistribution::casing_mw  [protected]
```

Weapon casing average molecular weight in g/mol (A)

5.1.4.11 casing_thermal

```
double ActivityDistribution::casing_thermal  [protected]
```

Weapon casing average thermal neutron x-sec in barns (sigma_c)

5.1.4.12 soil_thermal

```
double ActivityDistribution::soil_thermal  [protected]
```

Soil material average thermal neutron x-sec in barns (sigma_s)

5.1.4.13 soil_scattering

```
double ActivityDistribution::soil_scattering  [protected]
```

Soil material average neutron scattering in barns (sigma_ssc)

5.1.4.14 weapon_thermal

```
double ActivityDistribution::weapon_thermal  [protected]
```

Weapone material thermal neutron x-sec in barns.

5.1.4.15 size_cutoff

```
double ActivityDistribution::size_cutoff  [protected]
```

Size cutoff point for the Freiling-Tompkins distributions in um (D)

5.1.4.16 casing_atom

```
std::map<std::string, Atom> ActivityDistribution::casing_atom  [protected]
```

Stores a map of casing atom components (key is the atom)

5.1.4.17 casing_atom_frac

```
std::map<std::string, double> ActivityDistribution::casing_atom_frac  [protected]
```

Stores a map of casing atom components (key is the atom)

5.1.4.18 casing_mat

```
std::map<std::string, Molecule> ActivityDistribution::casing_mat  [protected]
```

Weapon casing molecular composition.

5.1.4.19 casing_frac

```
std::map<std::string, double> ActivityDistribution::casing_frac  [protected]
```

Weapon casing molefractions.

5.1.4.20 burst_height

```
double ActivityDistribution::burst_height  [protected]
```

Weapon burst height above ground (ft)

5.1.4.21 escape_fraction

```
double ActivityDistribution::escape_fraction  [protected]
```

Neutron fraction that escapes casing (e^-Sigma∗X)

5.1.4.22 volatile_fraction

```
double ActivityDistribution::volatile_fraction  [protected]
```

Neutron fraction that enters volatilized soil.

5.1.4.23 soil_capture_fraction

```
double ActivityDistribution::soil_capture_fraction  [protected]
```

Neutron fraction that is captured by soil.

5.1.4.24 initial_frac

```
FissionProducts ActivityDistribution::initial_frac  [protected]
```

Below are the parameters associated with the activity-size distributions.

Initial fractionation prior to size differentiation

5.1.4.25 nuc_fractionation

```
std::map<double, FissionProducts> ActivityDistribution::nuc_fractionation  [protected]
```

Fractionation of nuclides with particle size (um)

5.1.4.26 radioactivity_dist

```
std::map<double, double> ActivityDistribution::radioactivity_dist  [protected]
```

Distribution of radioactivity (in Bq) with particle size (um)

5.1.4.27 total_moles

```
std::map<int, double> ActivityDistribution::total_moles  [protected]
```

Total moles of nuclides for each mass number chain.

5.1.4.28 refractory_moles

```
std::map<int, double> ActivityDistribution::refractory_moles  [protected]
```

Refractory moles of nuclides for each mass number chain.

5.1.4.29 freiling_numbers

```
std::map<int, double> ActivityDistribution::freiling_numbers  [protected]
```

Map of the Freiling ratio numbers (b_i)

5.1.4.30 distribution

```
std::unordered_map<int, std::unordered_map<double, double> > ActivityDistribution::distribution
[protected]
```

Map of the normalized distributions for nuclides [i] ==> mass num [k] ==> particle size.

5.1.4.31 casing_capfrac

```
std::map<std::string, double> ActivityDistribution::casing_capfrac  [protected]
```

Neutron fractional captures by all atomic materials.

Casing neutron capture fractions (by atom symbol)

5.1.4.32 soil_capfrac

```
std::map<std::string, double> ActivityDistribution::soil_capfrac  [protected]
```

Soil neutron capture fractions (by atom symbol)

5.1.4.33 weapon_capfrac

```
std::map<std::string, double> ActivityDistribution::weapon_capfrac  [protected]
```

Weapon neutron capture fractions (by isotope name)

The documentation for this class was generated from the following file:

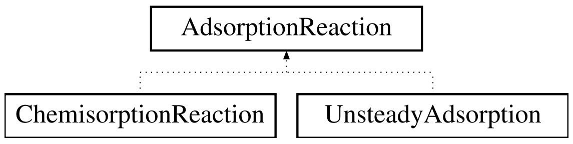
- kea.h

5.2 AdsorptionReaction Class Reference

Adsorption Reaction Object.

```
#include <shark.h>
```

Inheritance diagram for AdsorptionReaction:



Public Member Functions

- AdsorptionReaction ()

    Default Constructor.
- ∼AdsorptionReaction ()

    Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int n)

    Function to call the initialization of objects sequentially.
- void Display_Info ()

    Display the adsorption reaction information (PLACE HOLDER)
- void modifyDeltas (MassBalance &mbo)

    Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

    Find and set the adsorbed species indices for each reaction object.
- int checkAqueousIndices ()

    Function to check and report errors in the aqueous species indices.
- void setActivityModelInfo (int(∗act)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data), const void ∗act_data)

    Function to set the surface activity model and data pointer.
- void setAqueousIndex (int rxn_i, int species_i)

    Set the primary aqueous species index for the ith reaction.
- int setAqueousIndexAuto ()

    Automatically sets the primary aqueous species index based on reactions.
- void setActivityEnum (int act)

    Set the surface activity enum value.
- void setMolarFactor (int rxn_i, double m)

Set the molar factor for the ith reaction (mol/mol)

- void setVolumeFactor (int i, double v)

  Set the ith volume factor for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

  Set the ith area factor for the species list (m^2/mol)
- void setSpecificArea (double a)

  Set the specific area for the adsorbent (m^2/kg)
- void setSpecificMolality (double a)

  Set the specific molality for the adsorbent (mol/kg)
- void setSurfaceCharge (double c)

  Set the surface charge of the uncomplexed ligands.
- void setTotalMass (double m)

  Set the total mass of the adsorbent (kg)
- void setTotalVolume (double v)

  Set the total volume of the system (L)
- void setAreaBasisBool (bool opt)

  Set the basis boolean directly.
- void setSurfaceChargeBool (bool opt)

  Set the boolean for inclusion of surface charging.
- void setBasis (std::string option)

  Set the basis of the adsorption problem from the given string arg.
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent to the given string.
- void setChargeDensityValue (double a)

  Set the value of the charge density parameter to a (C/m^2)
- void setIonicStrengthValue (double a)

  Set the value of the ionic strength parameter to a (mol/L)
- void setActivities (Matrix< double > &x)

  Set the values of activities in the activity matrix.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- double calculateActiveFraction (const Matrix< double > &x)

  Calculates the fraction of the surface that is active and available.
- double calculateSurfaceChargeDensity (const Matrix< double > &x)

  Function to calculate the surface charge density based on concentrations.
- double calculateLangmuirMaxCapacity (int i)

  Calculates the theoretical maximum capacity for adsorption in reaction i.
- double calculateLangmuirEquParam (const Matrix< double > &x, const Matrix< double > &gama, int i)

  Calculates the equivalent Langmuir isotherm equilibrium parameter.
- double calculateLangmuirAdsorption (const Matrix< double > &x, const Matrix< double > &gama, int i)

  Calculates the equivalent Langmuir adsorption by forming the Langmuir-like parameters.
- double calculatePsi (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.

- double calculateAqueousChargeExchange (int i)

  Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int i)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_↩ perm, int i)

  Calculates the residual for the ith reaction in the system.
- Reaction & getReaction (int i)

  Return reference to the ith reaction object in the adsorption object.
- double getMolarFactor (int i)

  Get the ith reaction's molar factor for adsorption (mol/mol)
- double getVolumeFactor (int i)

  Get the ith volume factor (species not involved return zeros) (cm^3/mol)
- double getAreaFactor (int i)

  Get the ith area factor (species not involved return zeros) (m^2/mol)
- double getActivity (int i)

  Get the ith activity factor for the surface species.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getSpecificMolality ()

  Get the specific molality of the adsorbent (mol/kg)
- double getSurfaceCharge ()

  Get the surface charge of the adsorbent.
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- int getNumberRxns ()

  Get the number of reactions involved in the adsorption object.
- int getAdsorbIndex (int i)

  Get the index of the adsorbed species in the ith reaction.
- int getAqueousIndex (int i)

  Get the index of the primary aqueous species in the ith reaction.
- int getActivityEnum ()

  Return the enum representing the choosen activity function.
- bool isAreaBasis ()

  Returns true if we are in the Area Basis, False if in Molar Basis.
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getAdsorbentName ()

  Returns the name of the adsorbent as a string.

Protected Attributes

- MasterSpeciesList ∗ List

  Pointer to the MasterSpeciesList object.
- int(∗ surface_activity)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data)

  Pointer to a surface activity model.
- const void ∗ activity_data

  Pointer to the data structure needed for surface activities.
- int act_fun

  Enumeration of the activity function being used for the surface phase.
- std::vector< double > area_factors

  List of the van der Waals areas associated with surface species (m^2/mol)
- std::vector< double > volume_factors

  List of the van der Waals volumes of each surface species (cm^3/mol)
- std::vector< int > adsorb_index

  List of the indices for the adsorbed species in the reactions.
- std::vector< int > aqueous_index

  List of the indices for the primary aqueous species in the reactions.
- std::vector< double > molar_factor

  List of the number of ligands needed to form one mole of adsorption in each reaction.
- Matrix< double > activities

  List of the activities calculated by the activity model.
- double specific_area

  Specific surface area of the adsorbent (m^2/kg)
- double specific_molality

  Specific molality of the adsorbent - moles of ligand per kg sorbent (mol/kg)
- double surface_charge

  Charge of the uncomplexed surface ligand species.
- double total_mass

  Total mass of the adsorbent in the system (kg)
- double total_volume

  Total volume of the system (L)
- double ionic_strength

  Ionic Strength of the system used to adjust equilibria constants (mol/L)
- double charge_density

  Surface charge density of the adsorbent used to adjust equilbria (C/m^2)
- int num_rxns

  Number of reactions involved in the adsorption equilibria.
- bool AreaBasis

  True = Adsorption on an area basis, False = Adsorption on a ligand basis.
- bool IncludeSurfCharge

  True = Includes surface charging corrections, False = Does not consider surface charge.
- std::string adsorbent_name

  Name of the adsorbent for this object.

Private Attributes

- std::vector< Reaction > ads_rxn

  List of reactions involved with adsorption.

5.2.1 Detailed Description

Adsorption Reaction Object.

C++ Object to handle data and functions associated with forumlating adsorption equilibrium reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 AdsorptionReaction()

```
AdsorptionReaction::AdsorptionReaction ( )
```

Default Constructor.

5.2.2.2 ∼AdsorptionReaction()

```
AdsorptionReaction::∼AdsorptionReaction ( )
```

Default Destructor.

5.2.3 Member Function Documentation

5.2.3.1 Initialize_Object()

```
void AdsorptionReaction::Initialize_Object (
            MasterSpeciesList & List,
            int n )
```

Function to call the initialization of objects sequentially.

5.2.3.2 Display_Info()

```
void AdsorptionReaction::Display_Info ( )
```

Display the adsorption reaction information (PLACE HOLDER)

5.2.3.3 modifyDeltas()

```
void AdsorptionReaction::modifyDeltas (
            MassBalance & mbo )
```

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.2.3.4 setAdsorbIndices()

```
int AdsorptionReaction::setAdsorbIndices ( )
```

Find and set the adsorbed species indices for each reaction object.

This function searches through the Reaction objects in AdsorptionReaction to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.2.3.5 checkAqueousIndices()

```
int AdsorptionReaction::checkAqueousIndices ( )
```

Function to check and report errors in the aqueous species indices.

5.2.3.6 setActivityModelInfo()

```
void AdsorptionReaction::setActivityModelInfo (
              int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void
*data) act,
              const void * act_data )
```

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.2.3.7 setAqueousIndex()

```
void AdsorptionReaction::setAqueousIndex (
              int rxn_i,
              int species_i )
```

Set the primary aqueous species index for the ith reaction.

5.2.3.8 setAqueousIndexAuto()

```
int AdsorptionReaction::setAqueousIndexAuto ( )
```

Automatically sets the primary aqueous species index based on reactions.

This function will go through all species and all reactions in the adsorption object and automatically set the primary aqueous species index based on the stoicheometry of the reaction. It will also check and make sure that the primary aqueous index species appears opposite of the adsorbed species in the reactions. Note: This function assumes that the adsorbed indices have already been set.

5.2.3.9 setActivityEnum()

```
void AdsorptionReaction::setActivityEnum (
            int act )
```

Set the surface activity enum value.

5.2.3.10 setMolarFactor()

```
void AdsorptionReaction::setMolarFactor (
            int rxn_i,
            double m )
```

Set the molar factor for the ith reaction (mol/mol)

5.2.3.11 setVolumeFactor()

```
void AdsorptionReaction::setVolumeFactor (
            int i,
            double v )
```

Set the ith volume factor for the species list (cm^3/mol)

5.2.3.12 setAreaFactor()

```
void AdsorptionReaction::setAreaFactor (
            int i,
            double a )
```

Set the ith area factor for the species list (m^2/mol)

5.2.3.13 setSpecificArea()

```
void AdsorptionReaction::setSpecificArea (
            double a )
```

Set the specific area for the adsorbent (m^2/kg)

5.2.3.14 setSpecificMolality()

```
void AdsorptionReaction::setSpecificMolality (
            double a )
```

Set the specific molality for the adsorbent (mol/kg)

5.2.3.15 setSurfaceCharge()

```
void AdsorptionReaction::setSurfaceCharge (
             double c )
```

Set the surface charge of the uncomplexed ligands.

5.2.3.16 setTotalMass()

```
void AdsorptionReaction::setTotalMass (
             double m )
```

Set the total mass of the adsorbent (kg)

5.2.3.17 setTotalVolume()

```
void AdsorptionReaction::setTotalVolume (
             double v )
```

Set the total volume of the system (L)

5.2.3.18 setAreaBasisBool()

```
void AdsorptionReaction::setAreaBasisBool (
             bool opt )
```

Set the basis boolean directly.

5.2.3.19 setSurfaceChargeBool()

```
void AdsorptionReaction::setSurfaceChargeBool (
             bool opt )
```

Set the boolean for inclusion of surface charging.

5.2.3.20 setBasis()

```
void AdsorptionReaction::setBasis (
             std::string option )
```

Set the basis of the adsorption problem from the given string arg.

5.2.3.21 setAdsorbentName()

```
void AdsorptionReaction::setAdsorbentName (
             std::string name )
```

Set the name of the adsorbent to the given string.

5.2.3.22 setChargeDensityValue()

```
void AdsorptionReaction::setChargeDensityValue (
             double a )
```

Set the value of the charge density parameter to a (C/m^2)

5.2.3.23 setIonicStrengthValue()

```
void AdsorptionReaction::setIonicStrengthValue (
             double a )
```

Set the value of the ionic strength parameter to a (mol/L)

5.2.3.24 setActivities()

```
void AdsorptionReaction::setActivities (
             Matrix< double > & x )
```

Set the values of activities in the activity matrix.

5.2.3.25 calculateAreaFactors()

```
void AdsorptionReaction::calculateAreaFactors ( )
```

Calculates the area factors used from the van der Waals volumes.

5.2.3.26 calculateEquilibria()

```
void AdsorptionReaction::calculateEquilibria (
             double T )
```

Calculates all equilibrium parameters as a function of temperature.

5.2.3.27 setChargeDensity()

```
void AdsorptionReaction::setChargeDensity (
             const Matrix< double > & x )
```

Calculates and sets the current value of charge density.

5.2.3.28 setIonicStrength()

```
void AdsorptionReaction::setIonicStrength (
             const Matrix< double > & x )
```

Calculates and sets the current value of ionic strength.

5.2.3.29 callSurfaceActivity()

```
int AdsorptionReaction::callSurfaceActivity (
             const Matrix< double > & x )
```

Calls the activity model and returns an int flag for success or failure.

5.2.3.30 calculateActiveFraction()

```
double AdsorptionReaction::calculateActiveFraction (
             const Matrix< double > & x )
```

Calculates the fraction of the surface that is active and available.

5.2.3.31 calculateSurfaceChargeDensity()

```
double AdsorptionReaction::calculateSurfaceChargeDensity (
             const Matrix< double > & x )
```

Function to calculate the surface charge density based on concentrations.

This function is used to calculate the surface charge density of the adsorbed species based on the charges and concentrations of the adsorbed species. The calculation is used to correct the adsorption equilibria constant based on a localized surface charge balance. This requires that you know the molality of the uncomplexed ligand species on the surface, as well as the specific surface area for the adsorbent.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.2.3.32 calculateLangmuirMaxCapacity()

```
double AdsorptionReaction::calculateLangmuirMaxCapacity (
            int i )
```

Calculates the theoretical maximum capacity for adsorption in reaction i.

This function is used to calculate the current maximum capacity of a species for a given adsorption reaction using the concentrations and activities of other species in the system. You must pass the index of the reaction of interest. The index of the species of interest is determined from the adsorb_index object. Note: This is only true if the stoicheometry for the adsorbed species is 1.

Parameters

i	index of the reaction of interest for the adsorption object

5.2.3.33 calculateLangmuirEquParam()

```
double AdsorptionReaction::calculateLangmuirEquParam (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            int i )
```

Calculates the equivalent Langmuir isotherm equilibrium parameter.

This function will take in the current aqueous activities and calculate an effective Langmuir adsorption parameter for use in determining the adsorption in the system. It uses the system temperature as well to calculate equilibrium. Note: This is only true if the stoicheometry for the adsorbed species is 1.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
i	index of the reaction of interest for the adsorption object

5.2.3.34 calculateLangmuirAdsorption()

```
double AdsorptionReaction::calculateLangmuirAdsorption (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            int i )
```

Calculates the equivalent Langmuir adsorption by forming the Langmuir-like parameters.

This function will use the calculateLangmuirMaxCapacity and calculateLangmuirEquParam functions to approximate the adsorption of the ith reaction given the concentration of aqueous species, activities, and temperature. Note: This is only true if the stoicheometry for the adsorbed species is 1.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
i	index of the reaction of interest for the adsorption object

5.2.3.35 calculatePsi()

```
double AdsorptionReaction::calculatePsi (
            double sigma,
            double T,
            double I,
            double rel_epsilon )
```

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.2.3.36 calculateAqueousChargeExchange()

```
double AdsorptionReaction::calculateAqueousChargeExchange (
            int i )
```

Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.

This function will look at all aqueous species involved in the ith adsorption reaction and sum up their stoicheometries and charges to see what the net change in charge is caused by the adsorption of charged species in solution. It is then used to adjust or correct the equilibrium constant for the given adsorption reaction.

Parameters

i	index of the reaction of interest for the adsorption object

5.2.3.37 calculateEquilibriumCorrection()

```
double AdsorptionReaction::calculateEquilibriumCorrection (
            double sigma,
            double T,
            double I,
            double rel_epsilon,
            int i )
```

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
i	index of the reaction of interest for the adsorption object

5.2.3.38 Eval_Residual()

```
double AdsorptionReaction::Eval_Residual (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            double T,
            double rel_perm,
            int i )
```

Calculates the residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.2.3.39 getReaction()

```
Reaction& AdsorptionReaction::getReaction (
            int i )
```

Return reference to the ith reaction object in the adsorption object.

5.2.3.40 getMolarFactor()

```
double AdsorptionReaction::getMolarFactor (
             int i )
```

Get the ith reaction's molar factor for adsorption (mol/mol)

5.2.3.41 getVolumeFactor()

```
double AdsorptionReaction::getVolumeFactor (
             int i )
```

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

5.2.3.42 getAreaFactor()

```
double AdsorptionReaction::getAreaFactor (
             int i )
```

Get the ith area factor (species not involved return zeros) (m^2/mol)

5.2.3.43 getActivity()

```
double AdsorptionReaction::getActivity (
             int i )
```

Get the ith activity factor for the surface species.

5.2.3.44 getSpecificArea()

```
double AdsorptionReaction::getSpecificArea ( )
```

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.2.3.45 getSpecificMolality()

```
double AdsorptionReaction::getSpecificMolality ( )
```

Get the specific molality of the adsorbent (mol/kg)

5.2.3.46 getSurfaceCharge()

```
double AdsorptionReaction::getSurfaceCharge ( )
```

Get the surface charge of the adsorbent.

5.2.3.47 getBulkDensity()

```
double AdsorptionReaction::getBulkDensity ( )
```

Calculate and return bulk density of adsorbent in system (kg/L)

5.2.3.48 getTotalMass()

```
double AdsorptionReaction::getTotalMass ( )
```

Get the total mass of adsorbent in the system (kg)

5.2.3.49 getTotalVolume()

```
double AdsorptionReaction::getTotalVolume ( )
```

Get the total volume of the system (L)

5.2.3.50 getChargeDensity()

```
double AdsorptionReaction::getChargeDensity ( )
```

Get the value of the surface charge density (C/m^2)

5.2.3.51 getIonicStrength()

```
double AdsorptionReaction::getIonicStrength ( )
```

Get the value of the ionic strength of solution (mol/L)

5.2.3.52 getNumberRxns()

```
int AdsorptionReaction::getNumberRxns ( )
```

Get the number of reactions involved in the adsorption object.

5.2.3.53 getAdsorbIndex()

```
int AdsorptionReaction::getAdsorbIndex (
             int i )
```

Get the index of the adsorbed species in the ith reaction.

5.2.3.54 getAqueousIndex()

```
int AdsorptionReaction::getAqueousIndex (
             int i )
```

Get the index of the primary aqueous species in the ith reaction.

5.2.3.55 getActivityEnum()

```
int AdsorptionReaction::getActivityEnum ( )
```

Return the enum representing the choosen activity function.

5.2.3.56 isAreaBasis()

```
bool AdsorptionReaction::isAreaBasis ( )
```

Returns true if we are in the Area Basis, False if in Molar Basis.

5.2.3.57 includeSurfaceCharge()

```
bool AdsorptionReaction::includeSurfaceCharge ( )
```

Returns true if we are considering surface charging during adsorption.

5.2.3.58 getAdsorbentName()

```
std::string AdsorptionReaction::getAdsorbentName ( )
```

Returns the name of the adsorbent as a string.

5.2.4 Member Data Documentation

5.2.4.1 List

```
MasterSpeciesList* AdsorptionReaction::List  [protected]
```

Pointer to the MasterSpeciesList object.

5.2.4.2 surface_activity

```
int(* AdsorptionReaction::surface_activity) (const Matrix< double > &logq, Matrix< double >
&activity, const void *data)  [protected]
```

Pointer to a surface activity model.

This is a function pointer for a surface activity model. The function must accept the log of the surface concentrations as an argument (logq) and provide the activities for each species (activity). The pointer data is used to pass any additional arguments needed.

Parameters

logq	matrix of the log (base 10) of surface concentrations of all species
activity	matrix of activity coefficients for all surface species (must be overriden)
data	pointer to a data structure needed to calculate activities

5.2.4.3 activity_data

```
const void* AdsorptionReaction::activity_data  [protected]
```

Pointer to the data structure needed for surface activities.

5.2.4.4 act_fun

```
int AdsorptionReaction::act_fun  [protected]
```

Enumeration of the activity function being used for the surface phase.

5.2.4.5 area_factors

```
std::vector<double> AdsorptionReaction::area_factors  [protected]
```

List of the van der Waals areas associated with surface species (m^2/mol)

5.2.4.6 volume_factors

```
std::vector<double> AdsorptionReaction::volume_factors  [protected]
```

List of the van der Waals volumes of each surface species (cm^3/mol)

5.2.4.7 adsorb_index

```
std::vector<int> AdsorptionReaction::adsorb_index  [protected]
```

List of the indices for the adsorbed species in the reactions.

5.2.4.8 aqueous_index

```
std::vector<int> AdsorptionReaction::aqueous_index  [protected]
```

List of the indices for the primary aqueous species in the reactions.

5.2.4.9 molar_factor

```
std::vector<double> AdsorptionReaction::molar_factor  [protected]
```

List of the number of ligands needed to form one mole of adsorption in each reaction.

5.2.4.10 activities

```
Matrix<double> AdsorptionReaction::activities  [protected]
```

List of the activities calculated by the activity model.

5.2.4.11 specific_area

```
double AdsorptionReaction::specific_area  [protected]
```

Specific surface area of the adsorbent (m^2/kg)

5.2.4.12 specific_molality

```
double AdsorptionReaction::specific_molality  [protected]
```

Specific molality of the adsorbent - moles of ligand per kg sorbent (mol/kg)

5.2.4.13 surface_charge

```
double AdsorptionReaction::surface_charge  [protected]
```

Charge of the uncomplexed surface ligand species.

5.2.4.14 total_mass

```
double AdsorptionReaction::total_mass  [protected]
```

Total mass of the adsorbent in the system (kg)

5.2.4.15 total_volume

```
double AdsorptionReaction::total_volume  [protected]
```

Total volume of the system (L)

5.2.4.16 ionic_strength

```
double AdsorptionReaction::ionic_strength  [protected]
```

Ionic Strength of the system used to adjust equilibria constants (mol/L)

5.2.4.17 charge_density

```
double AdsorptionReaction::charge_density  [protected]
```

Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

5.2.4.18 num_rxns

```
int AdsorptionReaction::num_rxns  [protected]
```

Number of reactions involved in the adsorption equilibria.

5.2.4.19 AreaBasis

```
bool AdsorptionReaction::AreaBasis  [protected]
```

True = Adsorption on an area basis, False = Adsorption on a ligand basis.

5.2.4.20 IncludeSurfCharge

```
bool AdsorptionReaction::IncludeSurfCharge  [protected]
```

True = Includes surface charging corrections, False = Does not consider surface charge.

5.2.4.21 adsorbent_name

```
std::string AdsorptionReaction::adsorbent_name  [protected]
```

Name of the adsorbent for this object.

5.2.4.22 ads_rxn

```
std::vector<Reaction> AdsorptionReaction::ads_rxn  [private]
```

List of reactions involved with adsorption.

The documentation for this class was generated from the following file:

- shark.h

5.3 ARNOLDI_DATA Struct Reference

Data structure for the construction of the Krylov subspaces for a linear system.

```
#include <lark.h>
```

Public Attributes

- int k

  Desired size of the Krylov subspace.
- int iter

  Actual size of the Krylov subspace.
- double beta

  Normalization parameter.
- double hp1

  Additional row element of H (separate storage for holding)
- bool Output = true

  True = print messages to console.
- std::vector< Matrix< double > > Vk

  (N) x (k) orthonormal vector basis stored as a vector of column matrices
- Matrix< double > Hkp1

  (k+1) x (k) upper Hessenberg matrix
- Matrix< double > yk

  (k) x (1) vector search direction
- Matrix< double > e1

  (k) x (1) orthonormal vector with 1 in first position
- Matrix< double > w

  (N) x (1) interim result of the matrix_vector multiplication
- Matrix< double > v

  (N) x (1) holding cell for the column entries of Vk and other interims
- Matrix< double > sum

  (N) x (1) running sum of subspace vectors for use in altering w

5.3.1 Detailed Description

Data structure for the construction of the Krylov subspaces for a linear system.

C-style object used in conjunction with the Arnoldi algorithm to construct an orthonormal basis and upper Hessenberg representation of a given linear operator. This is used to solve a linear system both iteratively (i.e., in conjunction with GMRESLP) and directly (i.e., in conjunction with FOM). Alternatively, you can just store the factorized components for later use in another routine.

5.3.2 Member Data Documentation

5.3.2.1 k

```
int ARNOLDI_DATA::k
```

Desired size of the Krylov subspace.

5.3.2.2 iter

```
int ARNOLDI_DATA::iter
```

Actual size of the Krylov subspace.

5.3.2.3 beta

```
double ARNOLDI_DATA::beta
```

Normalization parameter.

5.3.2.4 hp1

```
double ARNOLDI_DATA::hp1
```

Additional row element of H (separate storage for holding)

5.3.2.5 Output

```
bool ARNOLDI_DATA::Output = true
```

True = print messages to console.

5.3.2.6 Vk

```
std::vector< Matrix<double> > ARNOLDI_DATA::Vk
```

(N) x (k) orthonormal vector basis stored as a vector of column matrices

5.3.2.7 Hkp1

```
Matrix<double> ARNOLDI_DATA::Hkp1
```

(k+1) x (k) upper Hessenberg matrix

5.3.2.8 yk

`Matrix<double> ARNOLDI_DATA::yk`

(k) x (1) vector search direction

5.3.2.9 e1

`Matrix<double> ARNOLDI_DATA::e1`

(k) x (1) orthonormal vector with 1 in first position

5.3.2.10 w

`Matrix<double> ARNOLDI_DATA::w`

(N) x (1) interim result of the matrix_vector multiplication

5.3.2.11 v

`Matrix<double> ARNOLDI_DATA::v`

(N) x (1) holding cell for the column entries of Vk and other interims

5.3.2.12 sum

`Matrix<double> ARNOLDI_DATA::sum`

(N) x (1) running sum of subspace vectors for use in altering w

The documentation for this struct was generated from the following file:

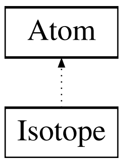
- lark.h

5.4 Atom Class Reference

Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions)

`#include <eel.h>`

Inheritance diagram for Atom:



Public Member Functions

- Atom ()

  Default Constructor.
- ∼Atom ()

  Default Destructor.
- void Register (std::string Symbol)

  Register an atom object by symbol.
- void Register (int number)

  Register an atom object by number.
- void editAtomicWeight (double AW)

  Manually changes the atomic weight.
- void editOxidationState (int state)

  Manually changes the oxidation state.
- void editProtons (int proton)

  Manually changes the number of protons.
- void editNeutrons (int neutron)

  Manually changes the number of neutrons.
- void editElectrons (int electron)

  Manually changes the number of electrons.
- void editValence (int val)

  Manually changes the number of valence electrons.
- void editRadii (double r)

  Manually changes the van der Waals radii.
- void editMeltingPoint (double val)

  Manually changes the melting point.
- void editBoilingPoint (double val)

  Manually changes the boiling point.
- void editThermalXSection (double val)

  Manually changes the thermal cross section.
- void editScatterXSection (double val)

  Manually changes the scattering cross section.
- void removeProton ()

  Manually removes 1 proton and adjusts weight.
- void removeNeutron ()

  Manually removes 1 neutron and adjusts weight.
- void removeElectron ()

  Manually removes 1 electron from valence.
- double AtomicWeight ()

  Returns the current atomic weight (g/mol)
- int OxidationState ()

  Returns the current oxidation state.
- int Protons ()

  Returns the current number of protons.
- int Neutrons ()

  Returns the current number of neutrons.
- int Electrons ()

  Returns the current number of electrons.
- int BondingElectrons ()

  Returns the number of electrons available for bonding.
- double AtomicRadii ()

Returns the current van der Waals radii (in angstroms)

- double MeltingPoint ()

  Returns the melting point.
- double BoilingPoint ()

  Returns the boiling point.
- double ThermalXSection ()

  Returns the thermal cross section.
- double ScatterXSection ()

  Returns the scattering cross section.
- double KShellEnergy ()

  Returns the K-shell energy (in keV)
- std::string AtomName ()

  Returns the name of the atom.
- std::string AtomSymbol ()

  Returns the symbol of the atom.
- std::string AtomCategory ()

  Returns the category of the atom.
- std::string AtomState ()

  Returns the state of the atom.
- int AtomicNumber ()

  Returns the atomic number of the atom.
- void DisplayInfo ()

  Displays Atom information to console.

Protected Attributes

- double atomic_weight

  Holds the atomic weight of the atom.
- int oxidation_state

  Holds the oxidation state of the atom.
- int protons

  Holds the number of protons in the atom.
- int neutrons

  Holds the number of neutrons in the atom.
- int electrons

  Holds the number of electrons in the atom.
- int valence_e

  Holds the number of valence electrons in the atom.
- double atomic_radii

  Holds the van der Waals radii of the element (in angstroms)
- double melting_point

  Holds the melting point of the element (in K)
- double boiling_point

  Holds the boiling point of the element (in K)
- double thermal_x_sec

  Holds the thermal neutron cross section of the element (in barns)
- double scatter_x_sec

  Holds the scattering neutron cross section of the element (in barns)
- double K_shell_energy

  Holds the K-Shell binding energy of electrons (in keV)

- std::vector< double > L_shell_energy

    Holds the list of L-Shell binding energies (in keV)
- std::string Name

    Holds the name of the atom.
- std::string Symbol

    Holds the atomic symbol for the atom.
- std::string Category

    Holds the category of the atom (e.g., Alkali Metal)
- std::string NaturalState

    Holds the natural state of the atom (e.g., Gas)
- int atomic_number

    Holds the atomic number of the atom.

5.4.1 Detailed Description

Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions)

C++ class object holding data and functions associated with atoms. Objects can be registered at the time of object construction, or after declaring an Atom object. Registration can be done via the atomic symbol or atomic number. Valid atoms go from Hydrogen (1) to Oganesson (118).

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Atom()

```
Atom::Atom ( )
```

Default Constructor.

5.4.2.2 ∼Atom()

```
Atom::∼Atom ( )
```

Default Destructor.

5.4.3 Member Function Documentation

5.4.3.1 Register() [1/2]

```
void Atom::Register (
            std::string Symbol )
```

Register an atom object by symbol.

5.4.3.2 Register() [2/2]

```
void Atom::Register (
             int number )
```

Register an atom object by number.

5.4.3.3 editAtomicWeight()

```
void Atom::editAtomicWeight (
             double AW )
```

Manually changes the atomic weight.

5.4.3.4 editOxidationState()

```
void Atom::editOxidationState (
             int state )
```

Manually changes the oxidation state.

5.4.3.5 editProtons()

```
void Atom::editProtons (
             int proton )
```

Manually changes the number of protons.

5.4.3.6 editNeutrons()

```
void Atom::editNeutrons (
             int neutron )
```

Manually changes the number of neutrons.

5.4.3.7 editElectrons()

```
void Atom::editElectrons (
             int electron )
```

Manually changes the number of electrons.

5.4.3.8 editValence()

```
void Atom::editValence (
            int val )
```

Manually changes the number of valence electrons.

5.4.3.9 editRadii()

```
void Atom::editRadii (
            double r )
```

Manually changes the van der Waals radii.

5.4.3.10 editMeltingPoint()

```
void Atom::editMeltingPoint (
            double val )
```

Manually changes the melting point.

5.4.3.11 editBoilingPoint()

```
void Atom::editBoilingPoint (
            double val )
```

Manually changes the boiling point.

5.4.3.12 editThermalXSection()

```
void Atom::editThermalXSection (
            double val )
```

Manually changes the thermal cross section.

5.4.3.13 editScatterXSection()

```
void Atom::editScatterXSection (
            double val )
```

Manually changes the scattering cross section.

5.4.3.14 removeProton()

```
void Atom::removeProton ( )
```

Manually removes 1 proton and adjusts weight.

5.4.3.15 removeNeutron()

```
void Atom::removeNeutron ( )
```

Manually removes 1 neutron and adjusts weight.

5.4.3.16 removeElectron()

```
void Atom::removeElectron ( )
```

Manually removes 1 electron from valence.

5.4.3.17 AtomicWeight()

```
double Atom::AtomicWeight ( )
```

Returns the current atomic weight (g/mol)

5.4.3.18 OxidationState()

```
int Atom::OxidationState ( )
```

Returns the current oxidation state.

5.4.3.19 Protons()

```
int Atom::Protons ( )
```

Returns the current number of protons.

5.4.3.20 Neutrons()

```
int Atom::Neutrons ( )
```

Returns the current number of neutrons.

5.4.3.21 Electrons()

```
int Atom::Electrons ( )
```

Returns the current number of electrons.

5.4.3.22 BondingElectrons()

```
int Atom::BondingElectrons ( )
```

Returns the number of electrons available for bonding.

5.4.3.23 AtomicRadii()

```
double Atom::AtomicRadii ( )
```

Returns the current van der Waals radii (in angstroms)

5.4.3.24 MeltingPoint()

```
double Atom::MeltingPoint ( )
```

Returns the melting point.

5.4.3.25 BoilingPoint()

```
double Atom::BoilingPoint ( )
```

Returns the boiling point.

5.4.3.26 ThermalXSection()

```
double Atom::ThermalXSection ( )
```

Returns the thermal cross section.

5.4.3.27 ScatterXSection()

```
double Atom::ScatterXSection ( )
```

Returns the scattering cross section.

5.4.3.28 KShellEnergy()

```
double Atom::KShellEnergy ( )
```

Returns the K-shell energy (in keV)

5.4.3.29 AtomName()

```
std::string Atom::AtomName ( )
```

Returns the name of the atom.

5.4.3.30 AtomSymbol()

```
std::string Atom::AtomSymbol ( )
```

Returns the symbol of the atom.

5.4.3.31 AtomCategory()

```
std::string Atom::AtomCategory ( )
```

Returns the category of the atom.

5.4.3.32 AtomState()

```
std::string Atom::AtomState ( )
```

Returns the state of the atom.

5.4.3.33 AtomicNumber()

```
int Atom::AtomicNumber ( )
```

Returns the atomic number of the atom.

5.4.3.34 DisplayInfo()

```
void Atom::DisplayInfo ( )
```

Displays Atom information to console.

5.4.4 Member Data Documentation

5.4.4.1 atomic_weight

```
double Atom::atomic_weight  [protected]
```

Holds the atomic weight of the atom.

5.4.4.2 oxidation_state

```
int Atom::oxidation_state  [protected]
```

Holds the oxidation state of the atom.

5.4.4.3 protons

```
int Atom::protons  [protected]
```

Holds the number of protons in the atom.

5.4.4.4 neutrons

```
int Atom::neutrons  [protected]
```

Holds the number of neutrons in the atom.

5.4.4.5 electrons

```
int Atom::electrons  [protected]
```

Holds the number of electrons in the atom.

5.4.4.6 valence_e

```
int Atom::valence_e  [protected]
```

Holds the number of valence electrons in the atom.

5.4.4.7 atomic_radii

```
double Atom::atomic_radii  [protected]
```

Holds the van der Waals radii of the element (in angstroms)

5.4.4.8 melting_point

```
double Atom::melting_point  [protected]
```

Holds the melting point of the element (in K)

5.4.4.9 boiling_point

```
double Atom::boiling_point  [protected]
```

Holds the boiling point of the element (in K)

5.4.4.10 thermal_x_sec

```
double Atom::thermal_x_sec  [protected]
```

Holds the thermal neutron cross section of the element (in barns)

5.4.4.11 scatter_x_sec

```
double Atom::scatter_x_sec  [protected]
```

Holds the scattering neutron cross section of the element (in barns)

5.4.4.12 K_shell_energy

```
double Atom::K_shell_energy  [protected]
```

Holds the K-Shell binding energy of electrons (in keV)

5.4.4.13 L_shell_energy

```
std::vector<double> Atom::L_shell_energy  [protected]
```

Holds the list of L-Shell binding energies (in keV)

5.4.4.14 Name

```
std::string Atom::Name  [protected]
```

Holds the name of the atom.

5.4.4.15 Symbol

```
std::string Atom::Symbol  [protected]
```

Holds the atomic symbol for the atom.

5.4.4.16 Category

```
std::string Atom::Category  [protected]
```

Holds the category of the atom (e.g., Alkali Metal)

5.4.4.17 NaturalState

```
std::string Atom::NaturalState  [protected]
```

Holds the natural state of the atom (e.g., Gas)

5.4.4.18 atomic_number

```
int Atom::atomic_number  [protected]
```

Holds the atomic number of the atom.

The documentation for this class was generated from the following file:

- eel.h

5.5 BACKTRACK_DATA Struct Reference

Data structure for the implementation of Backtracking Linesearch.

```
#include <lark.h>
```

Public Attributes

- int fun_call = 0

  Number of function calls made during line search.
- double alpha = 1e-4

  Scaling parameter for determination of search step size.
- double rho = 0.1

  Scaling parameter for to change step size by.
- double lambdaMin =DBL_EPSILON

  Smallest allowable step length.
- double normFkp1

  New residual norm of the Newton step.
- bool constRho = false

  True = use a constant value for rho.
- Matrix< double > Fk

  Old residual vector of the Newton step.
- Matrix< double > xk

  Old solution vector of the Newton step.

5.5.1 Detailed Description

Data structure for the implementation of Backtracking Linesearch.

C-style object used in conjunction with the Backtracking Linesearch algorithm to smooth out convergence of Netwon based iterative methods for non-linear systems of equations. The actual algorithm has been separated from the interior of the Newton method so that it can be included in any future Newton based iterative methods being developed.

5.5.2 Member Data Documentation

5.5.2.1 fun_call

```
int BACKTRACK_DATA::fun_call = 0
```

Number of function calls made during line search.

5.5.2.2 alpha

```
double BACKTRACK_DATA::alpha = 1e-4
```

Scaling parameter for determination of search step size.

5.5.2.3 rho

```
double BACKTRACK_DATA::rho = 0.1
```

Scaling parameter for to change step size by.

5.5.2.4 lambdaMin

```
double BACKTRACK_DATA::lambdaMin =DBL_EPSILON
```

Smallest allowable step length.

5.5.2.5 normFkp1

```
double BACKTRACK_DATA::normFkp1
```

New residual norm of the Newton step.

5.5.2.6 constRho

```
bool BACKTRACK_DATA::constRho = false
```

True = use a constant value for rho.

5.5.2.7 Fk

```
Matrix<double> BACKTRACK_DATA::Fk
```

Old residual vector of the Newton step.

5.5.2.8 xk

```
Matrix<double> BACKTRACK_DATA::xk
```

Old solution vector of the Newton step.

The documentation for this struct was generated from the following file:

- lark.h

5.6 BiCGSTAB_DATA Struct Reference

Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems.

```
#include <lark.h>
```

Public Attributes

- int maxit = 0

    Maximum allowable iterations - default = min(2∗vector_size,1000)
- int iter = 0

    Actual number of iterations.
- bool breakdown

    Boolean to determine if the method broke down.
- double alpha

    Step size parameter for next solution.
- double beta

    Step size parameter for search direction.
- double rho

    Scaling parameter for alpha and beta.
- double rho_old

    Previous scaling parameter for alpha and beta.
- double omega

    Scaling parameter and additional step length.
- double omega_old

    Previous scaling parameter and step length.
- double tol_rel = 1e-6

    Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

    Absolution tolerance for convergence - default = 1e-6.
- double res

    Absolute residual norm.
- double relres

    Relative residual norm.
- double relres_base

    Initial residual norm.
- double bestres

    Best found residual norm.
- bool Output = true

    True = print messages to console.
- Matrix< double > x

    Current solution to the linear system.
- Matrix< double > bestx

    Best found solution to the linear system.
- Matrix< double > r

    Residual vector for the linear system.
- Matrix< double > r0

    Initial residual vector.
- Matrix< double > v

    Search direction for p.

- Matrix< double > p

  Search direction for updating.
- Matrix< double > y

  Preconditioned search direction.
- Matrix< double > s

  Residual updating vector.
- Matrix< double > z

  Preconditioned residual updating vector.
- Matrix< double > t

  Search direction for resdidual updates.

5.6.1 Detailed Description

Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems.

C-style object used in conjunction with the Bi-Conjugate Gradient STABalized (BiCGSTAB) algorithm to solve a linear system of equations. This algorithm is generally more efficient than any GMRES or GCR variant, but may not always reduce the residual at each step. However, if used with preconditioning, then this algorithm is very efficient, especially when used for solving grid-based linear systems.

5.6.2 Member Data Documentation

5.6.2.1 maxit

```
int BiCGSTAB_DATA::maxit = 0
```

Maximum allowable iterations - default = min(2$*$vector_size,1000)

5.6.2.2 iter

```
int BiCGSTAB_DATA::iter = 0
```

Actual number of iterations.

5.6.2.3 breakdown

```
bool BiCGSTAB_DATA::breakdown
```

Boolean to determine if the method broke down.

5.6.2.4 alpha

```
double BiCGSTAB_DATA::alpha
```

Step size parameter for next solution.

5.6.2.5 beta

```
double BiCGSTAB_DATA::beta
```

Step size parameter for search direction.

5.6.2.6 rho

```
double BiCGSTAB_DATA::rho
```

Scaling parameter for alpha and beta.

5.6.2.7 rho_old

```
double BiCGSTAB_DATA::rho_old
```

Previous scaling parameter for alpha and beta.

5.6.2.8 omega

```
double BiCGSTAB_DATA::omega
```

Scaling parameter and additional step length.

5.6.2.9 omega_old

```
double BiCGSTAB_DATA::omega_old
```

Previous scaling parameter and step length.

5.6.2.10 tol_rel

```
double BiCGSTAB_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.6.2.11 tol_abs

```
double BiCGSTAB_DATA::tol_abs = 1e-6
```

Absolution tolerance for convergence - default = 1e-6.

5.6.2.12 res

```
double BiCGSTAB_DATA::res
```

Absolute residual norm.

5.6.2.13 relres

```
double BiCGSTAB_DATA::relres
```

Relative residual norm.

5.6.2.14 relres_base

```
double BiCGSTAB_DATA::relres_base
```

Initial residual norm.

5.6.2.15 bestres

```
double BiCGSTAB_DATA::bestres
```

Best found residual norm.

5.6.2.16 Output

```
bool BiCGSTAB_DATA::Output = true
```

True = print messages to console.

5.6.2.17 x

```
Matrix<double> BiCGSTAB_DATA::x
```

Current solution to the linear system.

5.6.2.18 bestx

```
Matrix<double> BiCGSTAB_DATA::bestx
```

Best found solution to the linear system.

5.6.2.19 r

```
Matrix<double> BiCGSTAB_DATA::r
```

Residual vector for the linear system.

5.6.2.20 r0

```
Matrix<double> BiCGSTAB_DATA::r0
```

Initial residual vector.

5.6.2.21 v

```
Matrix<double> BiCGSTAB_DATA::v
```

Search direction for p.

5.6.2.22 p

```
Matrix<double> BiCGSTAB_DATA::p
```

Search direction for updating.

5.6.2.23 y

```
Matrix<double> BiCGSTAB_DATA::y
```

Preconditioned search direction.

5.6.2.24 s

```
Matrix<double> BiCGSTAB_DATA::s
```

Residual updating vector.

5.6.2.25 z

```
Matrix<double> BiCGSTAB_DATA::z
```

Preconditioned residual updating vector.

5.6.2.26 t

```
Matrix<double> BiCGSTAB_DATA::t
```

Search direction for resdidual updates.

The documentation for this struct was generated from the following file:

- lark.h

5.7 Cardinal Class Reference

C++ object for coupling cloud rise and activity distribution.

```
#include <cardinal.h>
```

Public Member Functions

- Cardinal ()

    Default constructor.
- ∼Cardinal ()

    Default destructor.
- void setConsoleOut (bool val)

    Function to set the console output parameter.
- bool isConsoleOut ()

    Returns the ConsoleOut parameter.
- Crane & getCloudRise ()

    Return reference to the cloud rise simulation data.
- ActivityDistribution & getActivity ()

    Return reference to the activity distribution data.
- int readInputFile (const char ∗input)

    Read the yaml input file.
- int readAtomsphereFile (const char ∗input)

    Read the atomspheric data file.
- int setupCloudRiseSimulation (FILE ∗outfile)

    Setup the cloud rise simulations (post-read)
- int readDatabaseFiles (const char ∗path)

    Read the Nuclides and Fission Yields from the given path.
- int setupActivityDistribution ()

    Setup the activity distribution simulations (post-read)
- int runSimulations (int unc, std::string nuc_file)

    Run all the simulations (with uncertainty option) default = 0.

Private Attributes

- ActivityDistribution activity

    Object for the activity-size distributions.
- Crane cloudrise

    Object for the cloud rise simulations.
- FissionProducts yields

    Object for the fission product yields.
- Dove dove

    Object for the ODE solver.
- yaml_cpp_class nuc_data

    Object for the nuclide data.
- yaml_cpp_class yield_data

    Object for the yield data.
- yaml_cpp_class input_file

    Object for the input file.
- bool ConsoleOut

    Boolean used to determine whether or not to print info to the console.

5.7.1 Detailed Description

C++ object for coupling cloud rise and activity distribution.

Object will contain all data and function associated with running the cloud rise and activity-size distribution simulations coupled together. User is required to provide three inputs to run the CARDINAL_SCENARIO: (i) Input Control File, (ii) Atomspheric Data file, and (iii) Common Path to Static Database files. All of this information is necessary in order to fully run and couple simulations together.

5.7.2 Constructor & Destructor Documentation

5.7.2.1 Cardinal()

```
Cardinal::Cardinal ( )
```

Default constructor.

5.7.2.2 ∼Cardinal()

```
Cardinal::∼Cardinal ( )
```

Default destructor.

5.7.3 Member Function Documentation

5.7.3.1 setConsoleOut()

```
void Cardinal::setConsoleOut (
             bool val )
```

Function to set the console output parameter.

5.7.3.2 isConsoleOut()

```
bool Cardinal::isConsoleOut ( )
```

Returns the ConsoleOut parameter.

5.7.3.3 getCloudRise()

```
Crane& Cardinal::getCloudRise ( )
```

Return reference to the cloud rise simulation data.

5.7.3.4 getActivity()

```
ActivityDistribution& Cardinal::getActivity ( )
```

Return reference to the activity distribution data.

5.7.3.5 readInputFile()

```
int Cardinal::readInputFile (
             const char * input )
```

Read the yaml input file.

5.7.3.6 readAtomsphereFile()

```
int Cardinal::readAtomsphereFile (
             const char * input )
```

Read the atomspheric data file.

5.7.3.7 setupCloudRiseSimulation()

```
int Cardinal::setupCloudRiseSimulation (
            FILE * outfile )
```

Setup the cloud rise simulations (post-read)

5.7.3.8 readDatabaseFiles()

```
int Cardinal::readDatabaseFiles (
            const char * path )
```

Read the Nuclides and Fission Yields from the given path.

5.7.3.9 setupActivityDistribution()

```
int Cardinal::setupActivityDistribution ( )
```

Setup the activity distribution simulations (post-read)

5.7.3.10 runSimulations()

```
int Cardinal::runSimulations (
            int unc,
            std::string nuc_file )
```

Run all the simulations (with uncertainty option) default = 0.

5.7.4 Member Data Documentation

5.7.4.1 activity

```
ActivityDistribution Cardinal::activity  [private]
```

Object for the activity-size distributions.

5.7.4.2 cloudrise

```
Crane Cardinal::cloudrise  [private]
```

Object for the cloud rise simulations.

5.7.4.3 yields

```
FissionProducts Cardinal::yields  [private]
```

Object for the fission product yields.

5.7.4.4 dove

```
Dove Cardinal::dove  [private]
```

Object for the ODE solver.

5.7.4.5 nuc_data

```
yaml_cpp_class Cardinal::nuc_data  [private]
```

Object for the nuclide data.

5.7.4.6 yield_data

```
yaml_cpp_class Cardinal::yield_data  [private]
```

Object for the yield data.

5.7.4.7 input_file

```
yaml_cpp_class Cardinal::input_file  [private]
```

Object for the input file.

5.7.4.8 ConsoleOut

```
bool Cardinal::ConsoleOut  [private]
```

Boolean used to determine whether or not to print info to the console.

The documentation for this class was generated from the following file:

- cardinal.h

5.8 CGS_DATA Struct Reference

Data structure for the implementation of the CGS algorithm for non-symmetric linear systems.

```
#include <lark.h>
```

Public Attributes

- int maxit = 0

  Maximum allowable iterations - default = min(2∗vector_size,1000)
- int iter = 0

  Actual number of iterations.
- bool breakdown

  Boolean to determine if the method broke down.
- double alpha

  Step size parameter for next solution.
- double beta

  Step size parameter for search direction.
- double rho

  Scaling parameter for alpha and beta.
- double sigma

  Scaling parameter and additional step length.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolution tolerance for convergence - default = 1e-6.
- double res

  Absolute residual norm.
- double relres

  Relative residual norm.
- double relres_base

  Initial residual norm.
- double bestres

  Best found residual norm.
- bool Output = true

  True = print messages to console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual vector for the linear system.
- Matrix< double > r0

  Initial residual vector.
- Matrix< double > u

  Search direction for v.
- Matrix< double > w

  Updates sigma and u.
- Matrix< double > v

  Search direction for x.

- Matrix< double > p

    Preconditioning result for w, z, and matvec for Ax.
- Matrix< double > c

    Holds the matvec result between A and p.
- Matrix< double > z

    Full search direction for x.

5.8.1 Detailed Description

Data structure for the implementation of the CGS algorithm for non-symmetric linear systems.

C-style object to be used in conjunction with the Conjugate Gradient Squared (CGS) algorithm to solve linear systems of equations. This algorithm is slightly less computational work than BiCGSTAB, but is much less stable. As a result, I do not recommend using this algorithm unless you also use some form of preconditioning.

5.8.2 Member Data Documentation

5.8.2.1 maxit

```
int CGS_DATA::maxit = 0
```

Maximum allowable iterations - default = min(2∗vector_size,1000)

5.8.2.2 iter

```
int CGS_DATA::iter = 0
```

Actual number of iterations.

5.8.2.3 breakdown

```
bool CGS_DATA::breakdown
```

Boolean to determine if the method broke down.

5.8.2.4 alpha

```
double CGS_DATA::alpha
```

Step size parameter for next solution.

5.8.2.5 beta

```
double CGS_DATA::beta
```

Step size parameter for search direction.

5.8.2.6 rho

```
double CGS_DATA::rho
```

Scaling parameter for alpha and beta.

5.8.2.7 sigma

```
double CGS_DATA::sigma
```

Scaling parameter and additional step length.

5.8.2.8 tol_rel

```
double CGS_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.8.2.9 tol_abs

```
double CGS_DATA::tol_abs = 1e-6
```

Absolution tolerance for convergence - default = 1e-6.

5.8.2.10 res

```
double CGS_DATA::res
```

Absolute residual norm.

5.8.2.11 relres

```
double CGS_DATA::relres
```

Relative residual norm.

5.8.2.12 relres_base

```
double CGS_DATA::relres_base
```

Initial residual norm.

5.8.2.13 bestres

```
double CGS_DATA::bestres
```

Best found residual norm.

5.8.2.14 Output

```
bool CGS_DATA::Output = true
```

True = print messages to console.

5.8.2.15 x

```
Matrix<double> CGS_DATA::x
```

Current solution to the linear system.

5.8.2.16 bestx

```
Matrix<double> CGS_DATA::bestx
```

Best found solution to the linear system.

5.8.2.17 r

```
Matrix<double> CGS_DATA::r
```

Residual vector for the linear system.

5.8.2.18 r0

```
Matrix<double> CGS_DATA::r0
```

Initial residual vector.

5.8.2.19 u

```
Matrix<double> CGS_DATA::u
```

Search direction for v.

5.8.2.20 w

```
Matrix<double> CGS_DATA::w
```

Updates sigma and u.

5.8.2.21 v

```
Matrix<double> CGS_DATA::v
```

Search direction for x.

5.8.2.22 p

```
Matrix<double> CGS_DATA::p
```

Preconditioning result for w, z, and matvec for Ax.

5.8.2.23 c

```
Matrix<double> CGS_DATA::c
```

Holds the matvec result between A and p.

5.8.2.24 z

```
Matrix<double> CGS_DATA::z
```

Full search direction for x.

The documentation for this struct was generated from the following file:

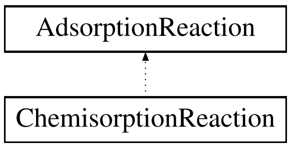
- lark.h

5.9 ChemisorptionReaction Class Reference

Chemisorption Reaction Object.

`#include <shark.h>`

Inheritance diagram for ChemisorptionReaction:



Public Member Functions

- ChemisorptionReaction ()

    Default Constructor.
- ∼ChemisorptionReaction ()

    Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int n)

    Function to call the initialization of objects sequentially.
- void Display_Info ()

    Display the adsorption reaction information.
- void modifyMBEdeltas (MassBalance &mbo)

    Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

    Find and set the adsorbed species indices for each reaction object.
- int setLigandIndex ()

    Find and set the ligand species index.
- int setDeltas ()

    Find and set all the delta values for the site balance.
- void setActivityModelInfo (int(∗act)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data), const void ∗act_data)

    Function to set the surface activity model and data pointer.
- void setActivityEnum (int act)

    Set the surface activity enum value.
- void setDelta (int i, double v)

    Set the ith delta factor for the site balance.
- void setVolumeFactor (int i, double v)

    Set the ith volume factor for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

    Set the ith area factor for the species list (m^2/mol)
- void setSpecificArea (double a)

    Set the specific area for the adsorbent (m^2/kg)
- void setSpecificMolality (double a)

    Set the specific molality for the adsorbent (mol/kg)
- void setTotalMass (double m)

    Set the total mass of the adsorbent (kg)
- void setTotalVolume (double v)

    Set the total volume of the system (L)

- void setSurfaceChargeBool (bool opt)

  Set the boolean for inclusion of surface charging.
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent to the given string.
- void setChargeDensityValue (double a)

  Set the value of the charge density parameter to a (C/m^2)
- void setIonicStrengthValue (double a)

  Set the value of the ionic strength parameter to a (mol/L)
- void setActivities (Matrix< double > &x)

  Set the values of activities in the activity matrix.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- double calculateSurfaceChargeDensity (const Matrix< double > &x)

  Function to calculate the surface charge density based on concentrations.
- double calculateElecticPotential (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateAqueousChargeExchange (int i)

  Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int i)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_RxnResidual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int i)

  Calculates the residual for the ith reaction in the system.
- double Eval_SiteBalanceResidual (const Matrix< double > &x)

  Calculates the residual for the overall site balance.
- Reaction & getReaction (int i)

  Return reference to the ith reaction object in the adsorption object.
- double getDelta (int i)

  Get the ith delta factor for the site balance.
- double getVolumeFactor (int i)

  Get the ith volume factor (species not involved return zeros) (cm^3/mol)
- double getAreaFactor (int i)

  Get the ith area factor (species not involved return zeros) (m^2/mol)
- double getActivity (int i)

  Get the ith activity factor for the surface species.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getSpecificMolality ()

  Get the specific molality of the adsorbent (mol/kg)
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)

- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- int getNumberRxns ()

  Get the number of reactions involved in the adsorption object.
- int getAdsorbIndex (int i)

  Get the index of the adsorbed species in the ith reaction.
- int getLigandIndex ()

  Get the index of the ligand species.
- int getActivityEnum ()

  Return the enum representing the choosen activity function.
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getAdsorbentName ()

  Returns the name of the adsorbent as a string.

Protected Attributes

- int ligand_index

  Index of the ligand for all reactions.
- std::vector< double > Delta

  Vector of weights (i.e., deltas) used in the site balance.

Private Attributes

- std::vector< Reaction > ads_rxn

  List of reactions involved with adsorption.

Additional Inherited Members

5.9.1 Detailed Description

Chemisorption Reaction Object.

C++ Object to handle data and functions associated with forumlating adsorption equilibrium reactions in a aqueous mixture based on chemisorption mechanisms. Each unique surface in a system will require an instance of this structure. This is very similar to AdsorptionReaction, however, this will include a site balance residual that will allow us to consider protonation and deprotonation of the ligands.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 ChemisorptionReaction()

```
ChemisorptionReaction::ChemisorptionReaction ( )
```

Default Constructor.

5.9.2.2 ∼ChemisorptionReaction()

```
ChemisorptionReaction::∼ChemisorptionReaction ( )
```

Default Destructor.

5.9.3 Member Function Documentation

5.9.3.1 Initialize_Object()

```
void ChemisorptionReaction::Initialize_Object (
            MasterSpeciesList & List,
            int n )
```

Function to call the initialization of objects sequentially.

5.9.3.2 Display_Info()

```
void ChemisorptionReaction::Display_Info ( )
```

Display the adsorption reaction information.

5.9.3.3 modifyMBEdeltas()

```
void ChemisorptionReaction::modifyMBEdeltas (
            MassBalance & mbo )
```

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.9.3.4 setAdsorbIndices()

```
int ChemisorptionReaction::setAdsorbIndices ( )
```

Find and set the adsorbed species indices for each reaction object.

This function searches through the Reaction objects in ChemisorptionReaction to find the adsorbed species and their indices to set that information in the adsorb_index structure. Function will return 0 if successful and -1 on a failure.

5.9.3.5 setLigandIndex()

```
int ChemisorptionReaction::setLigandIndex ( )
```

Find and set the ligand species index.

This function searches through the Reaction objects in ChemisorptionReaction to find the ligand species and its index to set that information in the ligand_index structure. Function will return 0 if successful and -1 on a failure.

5.9.3.6 setDeltas()

```
int ChemisorptionReaction::setDeltas ( )
```

Find and set all the delta values for the site balance.

This function searches through all reaction object instances for the stoicheometry of the ligand in each adsorption reaction. That stoicheometry serves as the basis for determining the site balance. NOTE: the delta for the ligand is set automatically in the setLigandIndex() function, so we can ignore that species. In addition, this function must be called after setLigandIndex() and setAdsorbIndices() are called and after the stoicheometry of each reaction has been determined.

5.9.3.7 setActivityModelInfo()

```
void ChemisorptionReaction::setActivityModelInfo (
             int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void
*data) act,
             const void * act_data )
```

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.9.3.8 setActivityEnum()

```
void ChemisorptionReaction::setActivityEnum (
             int act )
```

Set the surface activity enum value.

5.9.3.9 setDelta()

```
void ChemisorptionReaction::setDelta (
             int i,
             double v )
```

Set the ith delta factor for the site balance.

5.9.3.10 setVolumeFactor()

```
void ChemisorptionReaction::setVolumeFactor (
             int i,
             double v )
```

Set the ith volume factor for the species list (cm^3/mol)

5.9.3.11 setAreaFactor()

```
void ChemisorptionReaction::setAreaFactor (
             int i,
             double a )
```

Set the ith area factor for the species list (m^2/mol)

5.9.3.12 setSpecificArea()

```
void ChemisorptionReaction::setSpecificArea (
             double a )
```

Set the specific area for the adsorbent (m^2/kg)

5.9.3.13 setSpecificMolality()

```
void ChemisorptionReaction::setSpecificMolality (
             double a )
```

Set the specific molality for the adsorbent (mol/kg)

5.9.3.14 setTotalMass()

```
void ChemisorptionReaction::setTotalMass (
             double m )
```

Set the total mass of the adsorbent (kg)

5.9.3.15 setTotalVolume()

```
void ChemisorptionReaction::setTotalVolume (
            double v )
```

Set the total volume of the system (L)

5.9.3.16 setSurfaceChargeBool()

```
void ChemisorptionReaction::setSurfaceChargeBool (
            bool opt )
```

Set the boolean for inclusion of surface charging.

5.9.3.17 setAdsorbentName()

```
void ChemisorptionReaction::setAdsorbentName (
            std::string name )
```

Set the name of the adsorbent to the given string.

5.9.3.18 setChargeDensityValue()

```
void ChemisorptionReaction::setChargeDensityValue (
            double a )
```

Set the value of the charge density parameter to a (C/m^2)

5.9.3.19 setIonicStrengthValue()

```
void ChemisorptionReaction::setIonicStrengthValue (
            double a )
```

Set the value of the ionic strength parameter to a (mol/L)

5.9.3.20 setActivities()

```
void ChemisorptionReaction::setActivities (
            Matrix< double > & x )
```

Set the values of activities in the activity matrix.

5.9.3.21 calculateAreaFactors()

```
void ChemisorptionReaction::calculateAreaFactors ( )
```

Calculates the area factors used from the van der Waals volumes.

5.9.3.22 calculateEquilibria()

```
void ChemisorptionReaction::calculateEquilibria (
             double T )
```

Calculates all equilibrium parameters as a function of temperature.

5.9.3.23 setChargeDensity()

```
void ChemisorptionReaction::setChargeDensity (
             const Matrix< double > & x )
```

Calculates and sets the current value of charge density.

5.9.3.24 setIonicStrength()

```
void ChemisorptionReaction::setIonicStrength (
             const Matrix< double > & x )
```

Calculates and sets the current value of ionic strength.

5.9.3.25 callSurfaceActivity()

```
int ChemisorptionReaction::callSurfaceActivity (
             const Matrix< double > & x )
```

Calls the activity model and returns an int flag for success or failure.

5.9.3.26 calculateSurfaceChargeDensity()

```
double ChemisorptionReaction::calculateSurfaceChargeDensity (
             const Matrix< double > & x )
```

Function to calculate the surface charge density based on concentrations.

This function is used to calculate the surface charge density of the adsorbed species based on the charges and concentrations of the adsorbed species. The calculation is used to correct the adsorption equilibria constant based on a localized surface charge balance. This requires that you know the molality of the uncomplexed ligand species on the surface, as well as the specific surface area for the adsorbent.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.9.3.27 calculateElecticPotential()

```
double ChemisorptionReaction::calculateElecticPotential (
            double sigma,
            double T,
            double I,
            double rel_epsilon )
```

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.9.3.28 calculateAqueousChargeExchange()

```
double ChemisorptionReaction::calculateAqueousChargeExchange (
            int i )
```

Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.

This function will look at all aqueous species involved in the ith adsorption reaction and sum up their stoicheometries and charges to see what the net change in charge is caused by the adsorption of charged species in solution. It is then used to adjust or correct the equilibrium constant for the given adsorption reaction.

Parameters

i	index of the reaction of interest for the adsorption object

5.9.3.29 calculateEquilibriumCorrection()

```
double ChemisorptionReaction::calculateEquilibriumCorrection (
            double sigma,
            double T,
            double I,
            double rel_epsilon,
            int i )
```

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
i	index of the reaction of interest for the adsorption object

5.9.3.30 Eval_RxnResidual()

```
double ChemisorptionReaction::Eval_RxnResidual (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            double T,
            double rel_perm,
            int i )
```

Calculates the residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.9.3.31 Eval_SiteBalanceResidual()

```
double ChemisorptionReaction::Eval_SiteBalanceResidual (
            const Matrix< double > & x )
```

Calculates the residual for the overall site balance.

This function will provide a system residual for the site/ligand balance for the Chemisorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.9.3.32 getReaction()

```
Reaction& ChemisorptionReaction::getReaction (
             int i )
```

Return reference to the ith reaction object in the adsorption object.

5.9.3.33 getDelta()

```
double ChemisorptionReaction::getDelta (
             int i )
```

Get the ith delta factor for the site balance.

5.9.3.34 getVolumeFactor()

```
double ChemisorptionReaction::getVolumeFactor (
             int i )
```

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

5.9.3.35 getAreaFactor()

```
double ChemisorptionReaction::getAreaFactor (
             int i )
```

Get the ith area factor (species not involved return zeros) (m^2/mol)

5.9.3.36 getActivity()

```
double ChemisorptionReaction::getActivity (
             int i )
```

Get the ith activity factor for the surface species.

5.9.3.37 getSpecificArea()

```
double ChemisorptionReaction::getSpecificArea ( )
```

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.9.3.38 getSpecificMolality()

```
double ChemisorptionReaction::getSpecificMolality ( )
```

Get the specific molality of the adsorbent (mol/kg)

5.9.3.39 getBulkDensity()

```
double ChemisorptionReaction::getBulkDensity ( )
```

Calculate and return bulk density of adsorbent in system (kg/L)

5.9.3.40 getTotalMass()

```
double ChemisorptionReaction::getTotalMass ( )
```

Get the total mass of adsorbent in the system (kg)

5.9.3.41 getTotalVolume()

```
double ChemisorptionReaction::getTotalVolume ( )
```

Get the total volume of the system (L)

5.9.3.42 getChargeDensity()

```
double ChemisorptionReaction::getChargeDensity ( )
```

Get the value of the surface charge density (C/m^2)

5.9.3.43 getIonicStrength()

```
double ChemisorptionReaction::getIonicStrength ( )
```

Get the value of the ionic strength of solution (mol/L)

5.9.3.44 getNumberRxns()

```
int ChemisorptionReaction::getNumberRxns ( )
```

Get the number of reactions involved in the adsorption object.

5.9.3.45 getAdsorbIndex()

```
int ChemisorptionReaction::getAdsorbIndex (
             int i )
```

Get the index of the adsorbed species in the ith reaction.

5.9.3.46 getLigandIndex()

```
int ChemisorptionReaction::getLigandIndex ( )
```

Get the index of the ligand species.

5.9.3.47 getActivityEnum()

```
int ChemisorptionReaction::getActivityEnum ( )
```

Return the enum representing the choosen activity function.

5.9.3.48 includeSurfaceCharge()

```
bool ChemisorptionReaction::includeSurfaceCharge ( )
```

Returns true if we are considering surface charging during adsorption.

5.9.3.49 getAdsorbentName()

```
std::string ChemisorptionReaction::getAdsorbentName ( )
```

Returns the name of the adsorbent as a string.

5.9.4 Member Data Documentation

5.9.4.1 ligand_index

`int ChemisorptionReaction::ligand_index [protected]`

Index of the ligand for all reactions.

5.9.4.2 Delta

`std::vector<double> ChemisorptionReaction::Delta [protected]`

Vector of weights (i.e., deltas) used in the site balance.

5.9.4.3 ads_rxn

`std::vector<Reaction> ChemisorptionReaction::ads_rxn [private]`

List of reactions involved with adsorption.

The documentation for this class was generated from the following file:

- shark.h

5.10 ConstReaction Class Reference

ConstReaction class is an object for information and functions associated with the Generic Reaction.

`#include <crow.h>`

Public Member Functions

- ConstReaction ()

    Default constructor.
- ∼ConstReaction ()

    Default destructor.
- void InitializeSolver (Dove &Solver)

    Function to initialize the ConstReaction object from the Dove object.
- void SetIndex (int index)

    Function to set the index of the function/variable of interest.
- void SetForwardRate (double rate)

    Function to se the forward rate constant of the reaction.
- void SetReverseRate (double rate)

    Function to se the reverse rate constant of the reaction.
- void InsertStoichiometry (int i, double v)

    Insert a Stoichiometric value to the existing map.
- double getForwardRate ()

    Function to return the forward reaction rate constant.
- double getReverseRate ()

    Function to return the reverse reaction rate constant.
- int getIndex ()

    Function to return the primary variable index.
- std::map< int, double > & getStoichiometryMap ()

    Function to return reference to the Stoichiometry map object.

Protected Attributes

- double forward_rate

  Reaction rate constant associated with reactants.
- double reverse_rate

  Reaction rate constant associated with products.
- int main_index

  Variable index for the variable of interest.
- std::map< int, double > stoic

  Map of Stoichiometric coefficients for the reaction (access by var index)
- Dove ∗ SolverInfo

  Pointer to the Dove Object.

5.10.1 Detailed Description

ConstReaction class is an object for information and functions associated with the Generic Reaction.

This is a C++ style object designed to store and operate on the generic representation of a reaction mechanism. In this object, the reaction parameters are treated as constants and do not change with temperature. This object can be inherited from to add functionality that will compute reaction parameters as a function of system temperature. In addition, this object will contribute user functions (rate functions and jacobians) to DOVE, which will be interfaced through CROW_DATA (see below the class definition).

5.10.2 Constructor & Destructor Documentation

5.10.2.1 ConstReaction()

```
ConstReaction::ConstReaction ( )
```

Default constructor.

5.10.2.2 ∼ConstReaction()

```
ConstReaction::∼ConstReaction ( )
```

Default destructor.

5.10.3 Member Function Documentation

5.10.3.1 InitializeSolver()

```
void ConstReaction::InitializeSolver (
             Dove & Solver )
```

Function to initialize the ConstReaction object from the Dove object.

5.10.3.2 SetIndex()

```
void ConstReaction::SetIndex (
             int index )
```

Function to set the index of the function/variable of interest.

5.10.3.3 SetForwardRate()

```
void ConstReaction::SetForwardRate (
             double rate )
```

Function to se the forward rate constant of the reaction.

5.10.3.4 SetReverseRate()

```
void ConstReaction::SetReverseRate (
             double rate )
```

Function to se the reverse rate constant of the reaction.

5.10.3.5 InsertStoichiometry()

```
void ConstReaction::InsertStoichiometry (
             int i,
             double v )
```

Insert a Stoichiometric value to the existing map.

5.10.3.6 getForwardRate()

```
double ConstReaction::getForwardRate ( )
```

Function to return the forward reaction rate constant.

5.10.3.7 getReverseRate()

```
double ConstReaction::getReverseRate ( )
```

Function to return the reverse reaction rate constant.

5.10.3.8 getIndex()

```
int ConstReaction::getIndex ( )
```

Function to return the primary variable index.

5.10.3.9 getStoichiometryMap()

```
std::map<int, double>& ConstReaction::getStoichiometryMap ( )
```

Function to return reference to the Stoichiometry map object.

5.10.4 Member Data Documentation

5.10.4.1 forward_rate

```
double ConstReaction::forward_rate  [protected]
```

Reaction rate constant associated with reactants.

5.10.4.2 reverse_rate

```
double ConstReaction::reverse_rate  [protected]
```

Reaction rate constant associated with products.

5.10.4.3 main_index

```
int ConstReaction::main_index  [protected]
```

Variable index for the variable of interest.

5.10.4.4 stoic

```
std::map<int, double> ConstReaction::stoic  [protected]
```

Map of Stoichiometric coefficients for the reaction (access by var index)

5.10.4.5 SolverInfo

```
Dove* ConstReaction::SolverInfo  [protected]
```

Pointer to the Dove Object.

The documentation for this class was generated from the following file:

- crow.h

5.11 Crane Class Reference

CRANE object to hold data and functions associated with Cloud Rise.

```
#include <crane.h>
```

Public Member Functions

- Crane ()

  Default Constructor.
- ∼Crane ()

  Default Destructor.
- void display_part_hist ()
- void display_part_conc ()
- void display_soil_characteristics ()
- void set_eps (double val)

  Set the eps parameter.
- void set_grav (double val)

  Set the grav parameter.
- void set_k (double val)

  Set the k parameter.
- void set_k2 (double val)

  Set the k2 parameter.
- void set_k3 (double val)

  Set the k3 parameter.
- void set_k6 (double val)

  Set the k6 parameter.
- void set_mu (double val)

  Set the mu parameter.
- void set_k_temp (double val)

  Set the k_temp parameter.
- void set_apparent_temp (double val)

Set the apparent_temp parameter.
- void set_apparent_amb_temp (double val)

  Set the apparent_amb_temp parameter.
- void set_q_x (double val)

  Set the q_x parameter.
- void set_q_xe (double val)

  Set the q_xe parameter.
- void set_beta_prime (double val)

  Set the beta_prime parameter.
- void set_xe (double val)

  Set the xe parameter.
- void set_char_vel (double val)

  Set the char_vel parameter.
- void set_vert_rad (double val)

  Set the vert_rad parameter.
- void set_horz_rad (double val)

  Set the horz_rad parameter.
- void set_virtual_mass (double val)

  Set the virtual_mass parameter.
- void set_gas_const (double val)

  Set the gas_const parameter.
- void set_latent_heat (double val)

  Set the latent_heat parameter.
- void set_sigma_turbulence (double val)

  Set the sigma_turbulence parameter.
- void set_mean_spec_heat (double val)

  Set the mean_spec_heat parameter.
- void set_actual_spec_heat (double val)

  Set the actual_spec_heat parameter.
- void set_spec_heat_water (double val)

  Set the spec_heat_water parameter.
- void set_spec_heat_entrain (double val)

  Set the spec_heat_entrain parameter.
- void set_spec_heat_entrain_integral (double val)

  Set the spec_heat_entrain_integral parameter.
- void set_spec_heat_conds (double val)

  Set the spec_heat_conds parameter.
- void set_cloud_volume (double val)

  Set the cloud_volume parameter.
- void set_equil_temp (double val)

  Set the equil_temp parameter.
- void set_total_mass_fallout_rate (double val)

  Set the total_mass_fallout_rate parameter.
- void set_surf_area (double val)

  Set the surf_area parameter.
- void set_shear_ratio (double val)

  Set the shear_ratio parameter.
- void set_shear_vel (double val)

  Set the shear_vel parameter.
- void set_part_density (double val)

  Set the part_density parameter.

- void set_adjusted_height (double val)

  Set the adjusted_height parameter.
- void set_bomb_yield (double val)

  Set the bomb_yield parameter.
- void set_force_factor (double val)

  Set the force_factor parameter.
- void set_det_alt (double val)

  Set the det_alt parameter.
- void set_burst_height (double val)

  Set the burst_height parameter.
- void set_ground_alt (double val)

  Set the ground_alt parameter.
- void set_energy_frac (double val)

  Set the energy_frac parameter.
- void set_eccentricity (double val)

  Set the eccentricity parameter.
- void set_air_density (double val)

  Set the air_density parameter.
- void set_air_viscosity (double val)

  Set the air_viscosity parameter.
- void set_slip_factor (double val)

  Set the slip_factor parameter.
- void set_davies_num (double val)

  Set the davies_num parameter.
- void set_min_dia (double val)

  Set the min_dia parameter.
- void set_max_dia (double val)

  Set the max_dia parameter.
- void set_mean_dia (double val)

  Set the mean_dia parameter.
- void set_std_dia (double val)

  Set the std_dia parameter.
- void set_num_bins (int val)

  Set the num_bins parameter.
- void set_cloud_mass (double val)

  Set the cloud_mass parameter.
- void set_entrained_mass (double val)

  Set the entrained_mass parameter.
- void set_cloud_rise (double val)

  Set the cloud_rise parameter.
- void set_cloud_alt (double val)

  Set the cloud_alt parameter.
- void set_x_water_vapor (double val)

  Set the x_water_vapor parameter.
- void set_w_water_conds (double val)

  Set the w_water_conds parameter.
- void set_s_soil (double val)

  Set the s_soil parameter.
- void set_temperature (double val)

  Set the temperature parameter.
- void set_energy (double val)

Set the energy parameter.

- void set_current_time (double val)

  Set the current_time parameter.
- void set_vapor_pressure (double val)

  Set the vapor_pressure parameter.
- void set_sat_vapor_pressure (double val)

  Set the sat_vapor_pressure parameter.
- void set_solidification_temp (double val)

  Set the solidification_temp parameter.
- void set_vaporization_temp (double val)

  Set the vaporization_temp parameter.
- void set_initial_soil_mass (double val)

  Set the initial_soil_mass parameter.
- void set_initial_soil_vapor (double val)

  Set the initial_soil_vapor parameter.
- void set_initial_water_mass (double val)

  Set the initial_water_mass parameter.
- void set_initial_air_mass (double val)

  Set the initial_air_mass parameter.
- void set_current_amb_temp (double val)

  Set the current_amb_temp parameter.
- void set_current_atm_press (double val)

  Set the current_atm_press parameter.
- void set_includeShearVel (bool val)

  Set the includeShearVel parameter.
- void set_isSaturated (bool val)

  Set the isSaturated parameter.
- void set_isSolidified (bool val)

  Set the isSolidified parameter.
- void set_ConsoleOut (bool val)

  Set the ConsoleOut parameter.
- void set_FileOut (bool val)

  Set the FileOut parameter.
- void set_saturation_time (double val)

  Set the saturation_time parameter.
- void set_solidification_time (double val)

  Set the solidification_time parameter.
- void set_stabilization_time (double val)

  Set the stabilization_time parameter.
- void set_isTight (bool val)

  Set the isTight parameter.
- void set_useCustomDist (bool val)

  Set the useCustomDist parameter.
- void set_cloud_density (double val)

  Set the cloud_density parameter.
- void set_horz_rad_change (double val)

  Set the horz_rad_change parameter.
- void set_energy_switch (double val)

  Set the energy_switch parameter.
- void set_alt_top (double val)

  Set the alt_top parameter.

- void set_alt_bottom (double val)

    Set the alt_bottom parameter.
- void set_alt_top_old (double val)

    Set the alt_top_old parameter.
- void set_alt_bottom_old (double val)

    Set the alt_bottom_old parameter.
- void set_rise_top (double val)

    Set the rise_top parameter.
- void set_rise_bottom (double val)

    Set the rise_bottom parameter.
- void set_CloudFile (FILE ∗file)

    Set the CloudFile parameter.
- double get_eps ()

    Get the eps parameter.
- double get_grav ()

    Get the grav parameter.
- double get_k ()

    Get the k parameter.
- double get_k2 ()

    Get the k2 parameter.
- double get_k3 ()

    Get the k3 parameter.
- double get_k6 ()

    Get the k6 parameter.
- double get_mu ()

    Get the mu parameter.
- double get_k_temp ()

    Get the k_temp parameter.
- double get_apparent_temp ()

    Get the apparent_temp parameter.
- double get_apparent_amb_temp ()

    Get the apparent_amb_temp parameter.
- double get_q_x ()

    Get the q_x parameter.
- double get_q_xe ()

    Get the q_xe parameter.
- double get_beta_prime ()

    Get the beta_prime parameter.
- double get_xe ()

    Get the xe parameter.
- double get_char_vel ()

    Get the char_vel parameter.
- double get_vert_rad ()

    Get the vert_rad parameter.
- double get_horz_rad ()

    Get the horz_rad parameter.
- double get_virtual_mass ()

    Get the virtual_mass parameter.
- double get_gas_const ()

    Get the gas_const parameter.
- double get_latent_heat ()

Get the latent_heat parameter.
- double get_sigma_turbulence ()

  Get the sigma_turbulence parameter.
- double get_mean_spec_heat ()

  Get the mean_spec_heat parameter.
- double get_actual_spec_heat ()

  Get the actual_spec_heat parameter.
- double get_spec_heat_water ()

  Get the spec_heat_water parameter.
- double get_spec_heat_entrain ()

  Get the spec_heat_entrain parameter.
- double get_spec_heat_entrain_integral ()

  Get the spec_heat_entrain_integral parameter.
- double get_spec_heat_conds ()

  Get the spec_heat_conds parameter.
- double get_cloud_volume ()

  Get the cloud_volume parameter.
- double get_equil_temp ()

  Get the equil_temp parameter.
- double get_total_mass_fallout_rate ()

  Get the total_mass_fallout_rate parameter.
- double get_surf_area ()

  Get the surf_area parameter.
- double get_shear_ratio ()

  Get the shear_ratio parameter.
- double get_shear_vel ()

  Get the shear_vel parameter.
- double get_part_density ()

  Get the part_density parameter.
- double get_adjusted_height ()

  Get the adjusted_height parameter.
- double get_bomb_yield ()

  Get the bomb_yield parameter.
- double get_force_factor ()

  Get the force_factor parameter.
- double get_det_alt ()

  Get the det_alt parameter.
- double get_burst_height ()

  Get the burst_height parameter.
- double get_ground_alt ()

  Get the ground_alt parameter.
- double get_energy_frac ()

  Get the energy_frac parameter.
- double get_eccentricity ()

  Get the eccentricity parameter.
- double get_air_density ()

  Get the air_density parameter.
- double get_air_viscosity ()

  Get the air_viscosity parameter.
- double get_slip_factor ()

  Get the slip_factor parameter.

- double get_davies_num ()

    Get the davies_num parameter.
- double get_min_dia ()

    Get the min_dia parameter.
- double get_max_dia ()

    Get the max_dia parameter.
- double get_mean_dia ()

    Get the mean_dia parameter.
- double get_std_dia ()

    Get the std_dia parameter.
- int get_num_bins ()

    Get the num_bins parameter.
- double get_cloud_mass ()

    Get the cloud_mass parameter.
- double get_entrained_mass ()

    Get the entrained_mass parameter.
- double get_cloud_rise ()

    Get the cloud_rise parameter.
- double get_cloud_alt ()

    Get the cloud_alt parameter.
- double get_x_water_vapor ()

    Get the x_water_vapor parameter.
- double get_w_water_conds ()

    Get the w_water_conds parameter.
- double get_s_soil ()

    Get the s_soil parameter.
- double get_temperature ()

    Get the temperature parameter.
- double get_energy ()

    Get the energy parameter.
- double get_current_time ()

    Get the current_time parameter.
- double get_vapor_pressure ()

    Get the vapor_pressure parameter.
- double get_sat_vapor_pressure ()

    Get the sat_vapor_pressure parameter.
- double get_solidification_temp ()

    Get the solidification_temp parameter.
- double get_vaporization_temp ()

    Get the vaporization_temp parameter.
- double get_initial_soil_mass ()

    Get the initial_soil_mass parameter.
- double get_initial_soil_vapor ()

    Get the initial_soil_vapor parameter.
- double get_initial_water_mass ()

    Get the initial_water_mass parameter.
- double get_initial_air_mass ()

    Get the initial_air_mass parameter.
- double get_current_amb_temp ()

    Get the current_amb_temp parameter.
- double get_current_atm_press ()

Get the current_atm_press parameter.
- bool get_includeShearVel ()

  Get the includeShearVel parameter.
- bool get_isSaturated ()

  Get the isSaturated parameter.
- bool get_isSolidified ()

  Get the isSolidified parameter.
- double get_part_size (int i)

  Get the i-th particle size parameter.
- double get_settling_rate (double Dj)

  Get the settling_rate associated with size Dj.
- double get_settling_rate_old (double Dj)

  Get the settling_rate_old associated with size Dj.
- bool get_ConsoleOut ()

  Get the ConsoleOut parameter.
- bool get_FileOut ()

  Get the FileOut parameter.
- Matrix< double > & get_part_conc_var ()

  Get the part_conc_var parameter.
- double get_saturation_time ()

  Get the saturation_time parameter.
- double get_solidification_time ()

  Get the solidification_time parameter.
- double get_stabilization_time ()

  Get the stabilization_time parameter.
- bool get_isTight ()

  Get the isTight parameter.
- bool get_useCustomDist ()

  Get the useCustomDist parameter.
- double get_cloud_density ()

  Get the cloud_density parameter.
- double get_horz_rad_change ()

  Get the horz_rad_change parameter.
- double get_energy_switch ()

  Get the energy_switch parameter.
- double get_alt_top ()

  Get the alt_top parameter.
- double get_alt_bottom ()

  Get the alt_bottom parameter.
- double get_alt_top_old ()

  Get the alt_top parameter.
- double get_alt_bottom_old ()

  Get the alt_bottom parameter.
- double get_rise_top ()

  Get the rise_top parameter.
- double get_rise_bottom ()

  Get the rise_bottom parameter.
- std::map< double, double > & get_part_conc ()

  Get the part_conc map parameter.
- std::map< double, double > & get_part_hist ()

  Get the part_hist map parameter.

- std::map< std::string, double > & get_soil_molefrac ()

  Get the soil_molefrac map parameter.
- std::map< std::string, Molecule > & get_soil_comp ()

  Get the soil_comp map parameter.
- std::map< std::string, double > & get_soil_atom_frac ()

  Get the soil_atom_frac map parameter.
- std::map< std::string, Atom > & get_soil_atom ()

  Get the soil_atom map parameter.
- void compute_beta_prime (double x, double s, double w)

  Function to compute ratio of cloud gas density to local density.
- void compute_q_x (double x)

  Function to compute virtual temperature ratio for x.
- void compute_q_xe (double xe)

  Function to compute virtual temperature ratio for xe.
- void compute_apparent_temp (double T, double x)

  Function to compute apparent temp (T∗) given T and x.
- void compute_apparent_amb_temp (double Te, double xe)

  Function to compute apparent amb temp (Te∗) given Te and xe.
- void compute_char_vel (double u, double E)

  Function to compute characteristic velocity.
- void compute_air_viscosity (double T)

  Function to compute air viscosity.
- void compute_vapor_pressure (double P, double x)

  Function to compute vapor pressure in cloud given P and x.
- void compute_sat_vapor_pressure (double T)

  Function to compute saturation vapor pressure given T.
- void compute_xe (double Te, double P, double HR)

  Function to compute ambient air water ratio given Te, P, and HR.
- void compute_air_density (double P, double x, double T)

  Function to compute air density given P, x, and T.
- void compute_spec_heat_entrain (double T)

  Function to compute specific heat of entrained air given T.
- void compute_spec_heat_water (double T)

  Function to compute specific heat of water vapor given T.
- void compute_spec_heat_conds (double T)

  Function to compute specific heat of condensed matter given T.
- void compute_actual_spec_heat (double T, double x)

  Function to compute the actual specific heat of the cloud given T and x.
- void compute_k_temp (double T)

  Function to compute temperature factor based on given T.
- void compute_mean_spec_heat (double T, double x, double s, double w)

  Function to compute mean specific heat given T, x, s, and w.
- void compute_cloud_volume (double m, double x, double s, double w, double T, double P)

  Function to compute cloud volume.
- void compute_cloud_density (double m, double x, double s, double w, double T, double P)

  Function to compute cloud density.
- void compute_vert_rad (double z)

  Function to compute vertical cloud radius given z.
- void compute_horz_rad (double m, double x, double s, double w, double T, double P, double z)

  Function to compute horizontal radius.
- void compute_sigma_turbulence (double E, double z)

Function to compute sigma turbulence given E and z.
- void compute_surf_area (double m, double x, double s, double w, double T, double P, double z)

    Function to compute cloud surface area.
- void compute_shear_vel (Matrix< double > v_top, Matrix< double > v_bot)
- void compute_shear_ratio (double m, double x, double s, double w, double T, double P, double z, double u, double E, Matrix< double > v_top, Matrix< double > v_bot)

    Function to compute the shear_ratio based on the fundamental variables and atmospheric parameters.
- void compute_slip_factor (double Dj, double T, double P)

    Function to compute the slip_factor given Dj, T, and P.
- void compute_davies_num (double Dj, double m, double x, double s, double w, double T, double P)

    Function to compute davies_num given the conditions.
- void compute_settling_rate (double Dj, double m, double x, double s, double w, double T, double P)
- void compute_total_mass_fallout_rate (double m, double x, double s, double w, double T, double P, double z, const Matrix< double > &n)

    Function to compute total_mass_fallout_rate based on all given variables and parameters.
- void compute_energy_switch (double W)

    Function to compute energy switch from W.
- void compute_k (double W)

    Function to compute cloud rise yield from W.
- void compute_k2 (double W)

    Function to compute power function yield from W.
- void compute_mu (double W)

    Function to compute energy yield from W.
- void compute_force_factor (double W)

    Function to compute the force factor from W.
- void compute_equil_temp (double W)

    Function to compute T_ri from W.
- void create_part_hist (double min, double max, int size, double avg, double std)

    Function to compute normalized particle histogram.
- void compute_det_alt (double gz, double hb)

    Function to compute the det_alt given ground alt and burst height.
- void compute_initial_cloud_alt (double W, double gz, double hb)

    Function to compute initial cloud_alt.
- void compute_initial_current_time (double W, double gz, double hb)

    Function to compute initial time after detonation.
- void compute_initial_temperature (double W, double gz, double hb)

    Function to compute initial temperature of fallout cloud.
- void compute_initial_soil_mass (double W, double gz, double hb)

    Function to compute initial soil mass in cloud.
- void compute_initial_part_hist (double W, double gz, double hb, int size)

    Function to compute part_hist from W, gz, hb, and size.
- void compute_initial_air_mass (double W, double gz, double hb)

    Function to compute initial air mass in cloud.
- void compute_initial_water_mass (double W, double gz, double hb)

    Function to compute initial water mass in cloud.
- void compute_initial_entrained_mass (double W, double gz, double hb)

    Function to compute initial entrained mass in cloud.
- void compute_initial_cloud_mass (double W, double gz, double hb)

    Function to compute initial cloud mass.
- void compute_initial_s_soil (double W, double gz, double hb)

    Function to compute initial soil fraction.

- void compute_initial_x_water_vapor (double W, double gz, double hb)

  Function to compute initial water fraction.
- void compute_initial_cloud_volume (double W, double gz, double hb)

  Function to compute initial cloud volume.
- void compute_initial_horz_rad (double W, double gz, double hb)

  Function to compute initial horizontal radius of cloud.
- void compute_initial_vert_rad (double W, double gz, double hb)

  Function to compute initial vertical cloud radius.
- void compute_initial_cloud_rise (double W, double gz, double hb)

  Function to compute initial cloud rise velocity.
- void compute_initial_energy (double W, double gz, double hb)

  Function to compute initial energy for cloud.
- void compute_initial_part_conc (double W, double gz, double hb, int size)

  Function to compute initial particle concentrations.
- void compute_initial_virtual_mass (double W, double gz, double hb)

  Function to compute initial virtual mass of cloud.
- void compute_adjusted_height (double W, double gz, double hb)

  Function to compute the adjusted height of the cloud.
- void delete_particles ()

  Function to remove all existing particle information.
- void compute_spec_heat_entrain_integral (double T, double Te)

  Function to compute integral of spec_heat_entrain from Te to T.
- double return_spec_heat_water_integral (double T, double Te)

  Function to return integral of spec_heat_water from Te to T.
- double return_spec_heat_conds_integral (double T, double Te)

  Function to return integral of spec_heat_conds from Te to T.
- void add_amb_temp (double z, double Te)

  Function to add a temperature Te at altitude z.
- void add_atm_press (double z, double P)

  Function to add a pressure P at altitude z.
- void add_rel_humid (double z, double HR)

  Function to add a relative humidity HR at altitude z.
- void add_wind_vel (double z, double vx, double vy)

  Function to add a wind velocity by components at altitude z.
- void create_default_atmosphere ()

  Function to create a default atmosphere from -600 m to 49,800 m.
- void create_default_wind_profile ()

  Function to create a default wind profile from 216 m to 31,023 m.
- void delete_atmosphere ()

  Function to remove existing atmosphere profile from memory.
- void delete_wind_profile ()

  Function to remove existing wind profile from memory.
- double return_amb_temp (double z)

  Function to return the ambient temperature (K) given an altitude z (m)
- double return_atm_press (double z)

  Function to return the atmospheric pressure (Pa) given an altitude z (m)
- double return_rel_humid (double z)

  Function to return the relative humidity (%) given an altitude z (m)
- Matrix< double > return_wind_vel (double z)

  Function to return wind velocity vector (m/s) given an altitude z (m)
- double return_wind_speed (double z)

Function to return wind speed (m/s) given an altitude (m)

- void compute_current_amb_temp (double z)

    Function to set the current_amb_temp parameter given an altitude (m)

- void compute_current_atm_press (double z)

    Function to set the current_amb_press parameter given an altitude (m)

- void add_solid_param (std::string name, int pow, double param)

    Function to add a solidification parameter based on oxide name.

- void add_vapor_param (std::string name, int pow, double param)

    Function to add a vaporization parameter based on oxide name.

- void create_default_soil_components ()

    Function to setup the default soil component parameters.

- void delete_soil_components ()

    Function to remove all soil components and parameters.

- void add_soil_component (std::string name, double frac)

    Function to add soil components and corresponding molefraction.

- void verify_soil_components ()

    Function to check soil components for errors and correct.

- void compute_solidification_temp ()

    Function to compute soil solidification temperature based on components.

- void compute_vaporization_temp ()

    Function to compute soil vaporization temperature based on components.

- void compute_initial_soil_vapor ()

    Function to compute the initial vaporized soil mass (kg)

- void compute_alt_top (double z, double Hc)

    Function to compute cloud cap top given center z and height Hc.

- void compute_alt_bottom (double z, double Hc)

    Function to compute cloud cap bottom given center z and height Hc.

- void compute_rise_top (double z_new, double z_old, double dt)

    Function to compute cloud cap top rise given changes in top altitudes.

- void compute_rise_bottom (double z_new, double z_old, double dt)

    Function to compute cloud cap bottom rise given changes in bottom altitudes.

- Matrix< double > & return_parcel_alt_top ()

    Function to return matrix of parcels and particle sizes for top of parcel.

- Matrix< double > & return_parcel_alt_bot ()

    Function to return matrix of parcels and particle sizes for bottom of parcel.

- Matrix< double > & return_parcel_rad_top ()

    Function to return matrix of parcels and particle sizes for radius of top of parcel.

- Matrix< double > & return_parcel_rad_bot ()

    Function to return matrix of parcels and particle sizes for radius of bottom of parcel.

- Matrix< double > & return_parcel_conc ()

    Function to return matrix of parcel concentrations (row = parcel, col = particle size)

- int read_atmosphere_profile (const char ∗profile)

    Function to read atmospheric data from input file (return 0 on success and -1 on failure)

- int read_conditions (Dove &dove, yaml_cpp_class &yaml)

    Function to read yaml input for Simulation Conditions.

- void establish_initial_conditions (Dove &dove, double W, double gz, double hb, int bins, bool includeShear, bool isTight)

    Function will establish initial conditions, setup variables names, and register functions.

- int read_dove_options (Dove &dove, FILE ∗file, yaml_cpp_class &yaml)

    Function to read yaml input for ODE options.

- void establish_dove_options (Dove &dove, FILE ∗file, bool fileout, bool consoleout, integrate_subtype inttype, timestep_type timetype, precond_type type, double tol, double dtmin, double dtmax, double dtmin_conv, double t_out, double endtime)

    Function to establish DOVE conditions and options.
- int read_pjfnk_options (Dove &dove, yaml_cpp_class &yaml)

    Function to read yaml input for solver options.
- void establish_pjfnk_options (Dove &dove, krylov_method lin_method, linesearch_type linesearch, bool linear, bool precon, bool nl_out, bool l_out, int max_nlit, int max_lit, int restart, int recursive, double nl_abstol, double nl_reltol, double l_abstol, double l_reltol)

    Function to establish PJFNK conditions and options.
- int read_wind_profile (yaml_cpp_class &yaml)

    Function to read yaml input for solver options.
- void estimate_parameters (Dove &dove)

    Function to estimate all parameters prior to simulating a single timestep.
- void perform_postprocessing (Dove &dove)

    Function to compute post-processing information to establish cloud stem.
- void store_variables (Dove &dove)

    Function to store all variable values to be updated from Dove simulations.
- void print_header (Dove &dove)

    Function to print addition headers to the output file.
- void print_information (Dove &dove, bool initialPhase)

    Function to print additional information to output file.
- int run_crane_simulation (Dove &dove)

    Function to run the simulation to completion.

Protected Attributes

- double eps

    Ratio of molecular wieghts for water-vapor and dry air (eps)
- double grav

    Acceleration from gravity (m/s^2) (g)
- double k

    Dimensionless initial cloud rise yield (k)
- double k2

    Dimensionless power function yield (k2)
- double k3

    Dimensionless kinetic energy yield (k3)
- double k6

    Dimensionless wind shear factor (k6)
- double mu

    Dimensionless energy yield for given explosion (mu)
- double k_temp

    Dimensionless temperature factor for specific heat (k(T,T_ri))
- double apparent_temp

    Apparent temperature of cloud (K) (T∗)
- double apparent_amb_temp

    Apparent ambient temperature at specific altitude (K) (Te∗)
- double q_x

    Ratio of apparent to actual temperature in the cloud (q(x))
- double q_xe

Ratio of apparent to actual ambient temperature (q(xe))
- double beta_prime

  Ratio of cloud gas density to total cloud density (beta')
- double xe

  Ratio of water-vapor to dry air for ambient air at given altitude (xe)
- double char_vel

  Characteristic velocity for cloud rise (m/s) (v)
- double vert_rad

  Vertical radius of the cloud shape (m) (Hc)
- double horz_rad

  Horizontal radius of the cloud shape (m) (Rc)
- double virtual_mass

  Initial virtual mass of the cloud (kg) (m_i')
- double gas_const

  Gas law constant for dry air (J/kg/K) (Ra)
- double latent_heat

  Latent heat of evaporation of water (or ice) (J/kg) (L)
- double sigma_turbulence

  Rate of dissipation of turbulent energy per mass (K/s) (SIGMA)
- double mean_spec_heat

  Weighted mean specific heat of the cloud (J/kg/K) (c_pBAR)
- double actual_spec_heat

  Actual specific heat of the cloud mass (J/kg/K) (c_p(T))
- double spec_heat_water

  Specific heat of water vapor (J/kg/K) (c_pw(T))
- double spec_heat_entrain

  Specific heat of entrained air (J/kg/K) (c_pa(T))
- double spec_heat_entrain_integral

  Integrated Specific heat of entrained air (J/kg) (integral(c_pa(T)∗dT))
- double spec_heat_conds

  Specific heat of condensed matter (J/kg/K) (c_s(T))
- double cloud_volume

  Volume of the debris cloud (m^3) (V)
- double equil_temp

  Initial thermal equilibrium temperature (K) (T_ri)
- double total_mass_fallout_rate

  Overall rate of particle fallout (kg/s) (p(t))
- double surf_area

  Surface area of the cloud (m^2) (S)
- double shear_ratio

  (Surf_area/Vol)∗mu∗char_vel {Can be swapped to include shear (1/s)} ((S/V)∗mu∗v)
- double shear_vel

  Magnitude of shear velocity impacting cloud shape (m/s) (v_s)
- double part_density

  Average density of soil particles in cloud (kg/m^3) (rho_p)
- double adjusted_height

  Initial adjusted height of the cloud (m) (z')
- double bomb_yield

  Size of nuclear bomb or explosion (kT) (W)
- double force_factor

  Dimensionless factor of explosive force (F)

- double det_alt

  Height/altitude of detontation above mean sea level (m) (h)
- double burst_height

  Height of detonation above ground level (m) (h_b)
- double ground_alt

  Height/altitude of the ground level above sea level (m) (h_gz)
- double energy_frac

  Fraction of energy available to heat the air (phi)
- double eccentricity

  Eccentricity of an oblate spheriod (default = 0.75) (e)
- double air_density

  Density of air in cloud (kg/m^3) (rho)
- double air_viscosity

  Viscosity of air in cloud (kg/m/s) (eta)
- double slip_factor

  Slip factor for particle settling (-) (s)
- double davies_num

  Unitless number for particle settling analysis (-) (ND)
- double vapor_pressure

  Vapor pressure inside the cloud at cloud altitude (Pa) (Pv)
- double sat_vapor_pressure

  Saturation vapor pressure inside the cloud (Pa) (Pws)
- double solidification_temp

  Temperature at which soil debris will solidify (K) (Ts)
- double vaporization_temp

  Temperature at which soil debris will vaporize (K) (Tm)
- double initial_soil_mass

  Initial value for soil mass in debris cloud (kg) (m_ri)
- double initial_soil_vapor

  Initial value for the vaporized amount of soil (kg) (m_vi)
- double initial_water_mass

  Initial value for water mass in debris cloud (kg) (m_wi)
- double initial_air_mass

  Initial value for air mass in debris cloud (kg) (m_ai)
- double current_amb_temp

  Current value of ambient temperature (set based on atm profile) (Te)
- double current_atm_press

  Current value of atmospheric pressure (set based on atm profile) (P)
- bool includeShearVel

  Boolean statement used to include (true) or ignore (false) wind shear.
- bool isSaturated

  Boolean state used to determine whether or not to use Saturated Functions.
- bool isSolidified

  Boolean state used to determine whether or not the soils have solidified.
- bool isTight

  Boolean state used to determine whether or not to use Tight Coulpling.
- bool useCustomDist

  Boolean state used to determine whether or not to use a Custom Particle Distribution.
- bool ConsoleOut

  Boolean state used to determine whether or not to include console output.
- bool FileOut

Boolean state used to determine whether or not to include file output.

- std::map< double, double > amb_temp

  Ambient Temperature (K) at various altitudes (m) (Te)
- std::map< double, double > atm_press

  Atmospheric Pressure (Pa) at various altitudes (m) (P)
- std::map< double, double > rel_humid
- std::map< double, Matrix< double > > wind_vel

  Wind Velocities (m/s) at various altitudes (m) (v_a)
- std::map< double, double > part_hist

  Normalized Histogram of Particle Distribution by size (um)
- std::map< double, double > settling_rate

  Particle settling rate (m/s) by particle size (um) (f_j)
- std::vector< double > part_size

  Particle sizes listed in order from smallest to largest (um)
- double min_dia

  Minimum particle diameter for distribution (um) (dmin)
- double max_dia

  Maximum particle diameter for distribution (um) (dmax)
- double mean_dia

  Mean particle diameter for distribution (um) (d50)
- double std_dia

  Standard deviation for lognormal distribution (sigma)
- int num_bins

  Number of desired size bins for particles (N)
- std::map< std::string, Atom > soil_atom

  Stores a map of soil atom components (key is the atom)
- std::map< std::string, double > soil_atom_frac

  Stores a map of soil atom components (key is the atom)
- std::map< std::string, Molecule > soil_comp

  Stores the soil component molecule information.
- std::map< std::string, double > soil_molefrac
- std::unordered_map< std::string, std::map< int, double > > solid_params

  Polynominal parameters for specific oxides in soil used to determine the solidification temperature.
- std::unordered_map< std::string, std::map< int, double > > vapor_params

  Polynominal parameters for specific oxides in soil used to determine the vaporization temperature.
- double cloud_mass

  List of Variables to be solved for by Dove.
- double entrained_mass

  Mass of entrained materials in debris cloud (Mg) (m_ent)
- double cloud_rise

  Rate of cloud rise through atmosphere (m/s) (u)
- double cloud_alt

  Altitude of the cloud above mean sea level (m) (z)
- double x_water_vapor

  Mixing ratio for water vapor to dry air (kg/kg) (x)
- double w_water_conds

  Mixing ratio for condensed water to dry air (kg/kg) (w)
- double s_soil

  Mixing ratio for suspended soils to dry air (kg/kg) (s)
- double temperature

  Temperature of the air in the cloud (K) (T)

- double energy

  Mass-less Kinetic Energy in the cloud (J/kg) (E)
- std::map< double, double > part_conc

  Number of particles per volume (Gp/m^3) given size (um) (n_j(t))
- double current_time

  Current time since explosion (s) (t)
- Matrix< double > part_conc_var

  Storage matrix for particle concentrations (Gp/m^3) in order of size.
- double saturation_time

  Time at which saturation has occurred (s)
- double solidification_time

  Time at which the melted debris has solidified (s)
- double stabilization_time

  Time at which the debris cloud has stabilized (s)
- double cloud_density

  Density of the cloud materials (kg/m^3)
- double horz_rad_change

  Change in horizontal radius between time steps (m) (d(Rc))
- double energy_switch

  Energy switch parameter for determining termination (J/kg)
- double t_count

  Place holder for a time variable to determine when output is printed.
- double alt_top

  Top of the cloud cap (m) (zt)
- double alt_bottom

  Bottom of the cloud cap (m) (zb)
- double alt_top_old

  Old top of the cloud cap (m) (zt_old)
- double alt_bottom_old

  Old bottom of the cloud cap (m) (zb_old)
- double rise_top

  Cloud rise for top of cloud cap (m/s) (ut)
- double rise_bottom

  Cloud rise for the bottom of cloud cap (m/s) (ub)
- Matrix< double > parcel_alt_top

  Top of stem of ith parcel for jth particle size (m) (zt_ij)
- Matrix< double > parcel_alt_bot

  Bottom of the stem of ith parcel for jth particle size (m) (zb_ij)
- Matrix< double > parcel_rad_top

  Radius of the Top of stem of ith parcel for jth particle size (m) (Rt_ij)
- Matrix< double > parcel_rad_bot

  Radius of the Bottom of the stem of ith parcel for jth particle size (m) (Rb_ij)
- std::map< double, double > settling_rate_old

  Old Particle settling rate (m/s) by particle size (um) (f_j)
- Matrix< double > parcel_conc

  Concentration of dust inside each parcel (Gp/m^3)
- FILE ∗ CloudFile

  Output file to show help plot cloud growth over time.
- double t_cloud_out

  Time to print out cloud profiles to output file.
- double t_cloud_count

  Counter for the cloud output frequency.

5.11.1 Detailed Description

CRANE object to hold data and functions associated with Cloud Rise.

This is a C++ object that contains data and functions associated with calculating debris cloud rise from nuclear detonations. This object must be passed to the Dove object and registered as the user defined data structure. Then, inside of residual functions developed for each non-linear variable, this object must be appropriately dereferenced so that its members and functions can be called appropriately.

Note

Crane interfaces with Dove, but does not contain an instance of Dove. Also, Crane does not contain data for the non-linear variables since Dove holds the vectors of non-linear variables.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 Crane()

```
Crane::Crane ( )
```

Default Constructor.

5.11.2.2 ∼Crane()

```
Crane::∼Crane ( )
```

Default Destructor.

5.11.3 Member Function Documentation

5.11.3.1 display_part_hist()

```
void Crane::display_part_hist ( )
```

5.11.3.2 display_part_conc()

```
void Crane::display_part_conc ( )
```

5.11.3.3 display_soil_characteristics()

```
void Crane::display_soil_characteristics ( )
```

5.11.3.4 set_eps()

```
void Crane::set_eps (
             double val )
```

Set the eps parameter.

5.11.3.5 set_grav()

```
void Crane::set_grav (
             double val )
```

Set the grav parameter.

5.11.3.6 set_k()

```
void Crane::set_k (
             double val )
```

Set the k parameter.

5.11.3.7 set_k2()

```
void Crane::set_k2 (
             double val )
```

Set the k2 parameter.

5.11.3.8 set_k3()

```
void Crane::set_k3 (
             double val )
```

Set the k3 parameter.

5.11.3.9 set_k6()

```
void Crane::set_k6 (
            double val )
```

Set the k6 parameter.

5.11.3.10 set_mu()

```
void Crane::set_mu (
            double val )
```

Set the mu parameter.

5.11.3.11 set_k_temp()

```
void Crane::set_k_temp (
            double val )
```

Set the k_temp parameter.

5.11.3.12 set_apparent_temp()

```
void Crane::set_apparent_temp (
            double val )
```

Set the apparent_temp parameter.

5.11.3.13 set_apparent_amb_temp()

```
void Crane::set_apparent_amb_temp (
            double val )
```

Set the apparent_amb_temp parameter.

5.11.3.14 set_q_x()

```
void Crane::set_q_x (
            double val )
```

Set the q_x parameter.

5.11.3.15 set_q_xe()

```
void Crane::set_q_xe (
             double val )
```

Set the q_xe parameter.

5.11.3.16 set_beta_prime()

```
void Crane::set_beta_prime (
             double val )
```

Set the beta_prime parameter.

5.11.3.17 set_xe()

```
void Crane::set_xe (
             double val )
```

Set the xe parameter.

5.11.3.18 set_char_vel()

```
void Crane::set_char_vel (
             double val )
```

Set the char_vel parameter.

5.11.3.19 set_vert_rad()

```
void Crane::set_vert_rad (
             double val )
```

Set the vert_rad parameter.

5.11.3.20 set_horz_rad()

```
void Crane::set_horz_rad (
             double val )
```

Set the horz_rad parameter.

5.11.3.21 set_virtual_mass()

```
void Crane::set_virtual_mass (
            double val )
```

Set the virtual_mass parameter.

5.11.3.22 set_gas_const()

```
void Crane::set_gas_const (
            double val )
```

Set the gas_const parameter.

5.11.3.23 set_latent_heat()

```
void Crane::set_latent_heat (
            double val )
```

Set the latent_heat parameter.

5.11.3.24 set_sigma_turbulence()

```
void Crane::set_sigma_turbulence (
            double val )
```

Set the sigma_turbulence parameter.

5.11.3.25 set_mean_spec_heat()

```
void Crane::set_mean_spec_heat (
            double val )
```

Set the mean_spec_heat parameter.

5.11.3.26 set_actual_spec_heat()

```
void Crane::set_actual_spec_heat (
            double val )
```

Set the actual_spec_heat parameter.

5.11.3.27 set_spec_heat_water()

```
void Crane::set_spec_heat_water (
             double val )
```

Set the spec_heat_water parameter.

5.11.3.28 set_spec_heat_entrain()

```
void Crane::set_spec_heat_entrain (
             double val )
```

Set the spec_heat_entrain parameter.

5.11.3.29 set_spec_heat_entrain_integral()

```
void Crane::set_spec_heat_entrain_integral (
             double val )
```

Set the spec_heat_entrain_integral parameter.

5.11.3.30 set_spec_heat_conds()

```
void Crane::set_spec_heat_conds (
             double val )
```

Set the spec_heat_conds parameter.

5.11.3.31 set_cloud_volume()

```
void Crane::set_cloud_volume (
             double val )
```

Set the cloud_volume parameter.

5.11.3.32 set_equil_temp()

```
void Crane::set_equil_temp (
             double val )
```

Set the equil_temp parameter.

5.11.3.33 set_total_mass_fallout_rate()

```
void Crane::set_total_mass_fallout_rate (
             double val )
```

Set the total_mass_fallout_rate parameter.

5.11.3.34 set_surf_area()

```
void Crane::set_surf_area (
             double val )
```

Set the surf_area parameter.

5.11.3.35 set_shear_ratio()

```
void Crane::set_shear_ratio (
             double val )
```

Set the shear_ratio parameter.

5.11.3.36 set_shear_vel()

```
void Crane::set_shear_vel (
             double val )
```

Set the shear_vel parameter.

5.11.3.37 set_part_density()

```
void Crane::set_part_density (
             double val )
```

Set the part_density parameter.

5.11.3.38 set_adjusted_height()

```
void Crane::set_adjusted_height (
             double val )
```

Set the adjusted_height parameter.

5.11.3.39 set_bomb_yield()

```
void Crane::set_bomb_yield (
             double val )
```

Set the bomb_yield parameter.

5.11.3.40 set_force_factor()

```
void Crane::set_force_factor (
             double val )
```

Set the force_factor parameter.

5.11.3.41 set_det_alt()

```
void Crane::set_det_alt (
             double val )
```

Set the det_alt parameter.

5.11.3.42 set_burst_height()

```
void Crane::set_burst_height (
             double val )
```

Set the burst_height parameter.

5.11.3.43 set_ground_alt()

```
void Crane::set_ground_alt (
             double val )
```

Set the ground_alt parameter.

5.11.3.44 set_energy_frac()

```
void Crane::set_energy_frac (
             double val )
```

Set the energy_frac parameter.

5.11.3.45 set_eccentricity()

```
void Crane::set_eccentricity (
             double val )
```

Set the eccentricity parameter.

5.11.3.46 set_air_density()

```
void Crane::set_air_density (
             double val )
```

Set the air_density parameter.

5.11.3.47 set_air_viscosity()

```
void Crane::set_air_viscosity (
             double val )
```

Set the air_viscosity parameter.

5.11.3.48 set_slip_factor()

```
void Crane::set_slip_factor (
             double val )
```

Set the slip_factor parameter.

5.11.3.49 set_davies_num()

```
void Crane::set_davies_num (
             double val )
```

Set the davies_num parameter.

5.11.3.50 set_min_dia()

```
void Crane::set_min_dia (
             double val )
```

Set the min_dia parameter.

5.11.3.51 set_max_dia()

```
void Crane::set_max_dia (
             double val )
```

Set the max_dia parameter.

5.11.3.52 set_mean_dia()

```
void Crane::set_mean_dia (
             double val )
```

Set the mean_dia parameter.

5.11.3.53 set_std_dia()

```
void Crane::set_std_dia (
             double val )
```

Set the std_dia parameter.

5.11.3.54 set_num_bins()

```
void Crane::set_num_bins (
             int val )
```

Set the num_bins parameter.

5.11.3.55 set_cloud_mass()

```
void Crane::set_cloud_mass (
             double val )
```

Set the cloud_mass parameter.

5.11.3.56 set_entrained_mass()

```
void Crane::set_entrained_mass (
             double val )
```

Set the entrained_mass parameter.

5.11.3.57 set_cloud_rise()

```
void Crane::set_cloud_rise (
             double val )
```

Set the cloud_rise parameter.

5.11.3.58 set_cloud_alt()

```
void Crane::set_cloud_alt (
             double val )
```

Set the cloud_alt parameter.

5.11.3.59 set_x_water_vapor()

```
void Crane::set_x_water_vapor (
             double val )
```

Set the x_water_vapor parameter.

5.11.3.60 set_w_water_conds()

```
void Crane::set_w_water_conds (
             double val )
```

Set the w_water_conds parameter.

5.11.3.61 set_s_soil()

```
void Crane::set_s_soil (
             double val )
```

Set the s_soil parameter.

5.11.3.62 set_temperature()

```
void Crane::set_temperature (
             double val )
```

Set the temperature parameter.

5.11.3.63 set_energy()

```
void Crane::set_energy (
            double val )
```

Set the energy parameter.

5.11.3.64 set_current_time()

```
void Crane::set_current_time (
            double val )
```

Set the current_time parameter.

5.11.3.65 set_vapor_pressure()

```
void Crane::set_vapor_pressure (
            double val )
```

Set the vapor_pressure parameter.

5.11.3.66 set_sat_vapor_pressure()

```
void Crane::set_sat_vapor_pressure (
            double val )
```

Set the sat_vapor_pressure parameter.

5.11.3.67 set_solidification_temp()

```
void Crane::set_solidification_temp (
            double val )
```

Set the solidification_temp parameter.

5.11.3.68 set_vaporization_temp()

```
void Crane::set_vaporization_temp (
            double val )
```

Set the vaporization_temp parameter.

5.11.3.69 set_initial_soil_mass()

```
void Crane::set_initial_soil_mass (
             double val )
```

Set the initial_soil_mass parameter.

5.11.3.70 set_initial_soil_vapor()

```
void Crane::set_initial_soil_vapor (
             double val )
```

Set the initial_soil_vapor parameter.

5.11.3.71 set_initial_water_mass()

```
void Crane::set_initial_water_mass (
             double val )
```

Set the initial_water_mass parameter.

5.11.3.72 set_initial_air_mass()

```
void Crane::set_initial_air_mass (
             double val )
```

Set the initial_air_mass parameter.

5.11.3.73 set_current_amb_temp()

```
void Crane::set_current_amb_temp (
             double val )
```

Set the current_amb_temp parameter.

5.11.3.74 set_current_atm_press()

```
void Crane::set_current_atm_press (
             double val )
```

Set the current_atm_press parameter.

5.11.3.75 set_includeShearVel()

```
void Crane::set_includeShearVel (
             bool val )
```

Set the includeShearVel parameter.

5.11.3.76 set_isSaturated()

```
void Crane::set_isSaturated (
             bool val )
```

Set the isSaturated parameter.

5.11.3.77 set_isSolidified()

```
void Crane::set_isSolidified (
             bool val )
```

Set the isSolidified parameter.

5.11.3.78 set_ConsoleOut()

```
void Crane::set_ConsoleOut (
             bool val )
```

Set the ConsoleOut parameter.

5.11.3.79 set_FileOut()

```
void Crane::set_FileOut (
             bool val )
```

Set the FileOut parameter.

5.11.3.80 set_saturation_time()

```
void Crane::set_saturation_time (
             double val )
```

Set the saturation_time parameter.

5.11.3.81 set_solidification_time()

```
void Crane::set_solidification_time (
             double val )
```

Set the solidification_time parameter.

5.11.3.82 set_stabilization_time()

```
void Crane::set_stabilization_time (
             double val )
```

Set the stabilization_time parameter.

5.11.3.83 set_isTight()

```
void Crane::set_isTight (
             bool val )
```

Set the isTight parameter.

5.11.3.84 set_useCustomDist()

```
void Crane::set_useCustomDist (
             bool val )
```

Set the useCustomDist parameter.

5.11.3.85 set_cloud_density()

```
void Crane::set_cloud_density (
             double val )
```

Set the cloud_density parameter.

5.11.3.86 set_horz_rad_change()

```
void Crane::set_horz_rad_change (
             double val )
```

Set the horz_rad_change parameter.

5.11.3.87 set_energy_switch()

```
void Crane::set_energy_switch (
             double val )
```

Set the energy_switch parameter.

5.11.3.88 set_alt_top()

```
void Crane::set_alt_top (
             double val )
```

Set the alt_top parameter.

5.11.3.89 set_alt_bottom()

```
void Crane::set_alt_bottom (
             double val )
```

Set the alt_bottom parameter.

5.11.3.90 set_alt_top_old()

```
void Crane::set_alt_top_old (
             double val )
```

Set the alt_top_old parameter.

5.11.3.91 set_alt_bottom_old()

```
void Crane::set_alt_bottom_old (
             double val )
```

Set the alt_bottom_old parameter.

5.11.3.92 set_rise_top()

```
void Crane::set_rise_top (
             double val )
```

Set the rise_top parameter.

5.11.3.93 set_rise_bottom()

```
void Crane::set_rise_bottom (
             double val )
```

Set the rise_bottom parameter.

5.11.3.94 set_CloudFile()

```
void Crane::set_CloudFile (
             FILE * file )
```

Set the CloudFile parameter.

5.11.3.95 get_eps()

```
double Crane::get_eps ( )
```

Get the eps parameter.

5.11.3.96 get_grav()

```
double Crane::get_grav ( )
```

Get the grav parameter.

5.11.3.97 get_k()

```
double Crane::get_k ( )
```

Get the k parameter.

5.11.3.98 get_k2()

```
double Crane::get_k2 ( )
```

Get the k2 parameter.

5.11.3.99 get_k3()

```
double Crane::get_k3 ( )
```

Get the k3 parameter.

5.11.3.100 get_k6()

```
double Crane::get_k6 ( )
```

Get the k6 parameter.

5.11.3.101 get_mu()

```
double Crane::get_mu ( )
```

Get the mu parameter.

5.11.3.102 get_k_temp()

```
double Crane::get_k_temp ( )
```

Get the k_temp parameter.

5.11.3.103 get_apparent_temp()

```
double Crane::get_apparent_temp ( )
```

Get the apparent_temp parameter.

5.11.3.104 get_apparent_amb_temp()

```
double Crane::get_apparent_amb_temp ( )
```

Get the apparent_amb_temp parameter.

5.11.3.105 get_q_x()

```
double Crane::get_q_x ( )
```

Get the q_x parameter.

5.11.3.106 get_q_xe()

```
double Crane::get_q_xe ( )
```

Get the q_xe parameter.

5.11.3.107 get_beta_prime()

```
double Crane::get_beta_prime ( )
```

Get the beta_prime parameter.

5.11.3.108 get_xe()

```
double Crane::get_xe ( )
```

Get the xe parameter.

5.11.3.109 get_char_vel()

```
double Crane::get_char_vel ( )
```

Get the char_vel parameter.

5.11.3.110 get_vert_rad()

```
double Crane::get_vert_rad ( )
```

Get the vert_rad parameter.

5.11.3.111 get_horz_rad()

```
double Crane::get_horz_rad ( )
```

Get the horz_rad parameter.

5.11.3.112 get_virtual_mass()

```
double Crane::get_virtual_mass ( )
```

Get the virtual_mass parameter.

5.11.3.113 get_gas_const()

```
double Crane::get_gas_const ( )
```

Get the gas_const parameter.

5.11.3.114 get_latent_heat()

```
double Crane::get_latent_heat ( )
```

Get the latent_heat parameter.

5.11.3.115 get_sigma_turbulence()

```
double Crane::get_sigma_turbulence ( )
```

Get the sigma_turbulence parameter.

5.11.3.116 get_mean_spec_heat()

```
double Crane::get_mean_spec_heat ( )
```

Get the mean_spec_heat parameter.

5.11.3.117 get_actual_spec_heat()

```
double Crane::get_actual_spec_heat ( )
```

Get the actual_spec_heat parameter.

5.11.3.118 get_spec_heat_water()

```
double Crane::get_spec_heat_water ( )
```

Get the spec_heat_water parameter.

5.11.3.119 get_spec_heat_entrain()

```
double Crane::get_spec_heat_entrain ( )
```

Get the spec_heat_entrain parameter.

5.11.3.120 get_spec_heat_entrain_integral()

```
double Crane::get_spec_heat_entrain_integral ( )
```

Get the spec_heat_entrain_integral parameter.

5.11.3.121 get_spec_heat_conds()

```
double Crane::get_spec_heat_conds ( )
```

Get the spec_heat_conds parameter.

5.11.3.122 get_cloud_volume()

```
double Crane::get_cloud_volume ( )
```

Get the cloud_volume parameter.

5.11.3.123 get_equil_temp()

```
double Crane::get_equil_temp ( )
```

Get the equil_temp parameter.

5.11.3.124 get_total_mass_fallout_rate()

```
double Crane::get_total_mass_fallout_rate ( )
```

Get the total_mass_fallout_rate parameter.

5.11.3.125 get_surf_area()

```
double Crane::get_surf_area ( )
```

Get the surf_area parameter.

5.11.3.126 get_shear_ratio()

```
double Crane::get_shear_ratio ( )
```

Get the shear_ratio parameter.

5.11.3.127 get_shear_vel()

```
double Crane::get_shear_vel ( )
```

Get the shear_vel parameter.

5.11.3.128 get_part_density()

```
double Crane::get_part_density ( )
```

Get the part_density parameter.

5.11.3.129 get_adjusted_height()

```
double Crane::get_adjusted_height ( )
```

Get the adjusted_height parameter.

5.11.3.130 get_bomb_yield()

```
double Crane::get_bomb_yield ( )
```

Get the bomb_yield parameter.

5.11.3.131 get_force_factor()

```
double Crane::get_force_factor ( )
```

Get the force_factor parameter.

5.11.3.132 get_det_alt()

```
double Crane::get_det_alt ( )
```

Get the det_alt parameter.

5.11.3.133 get_burst_height()

```
double Crane::get_burst_height ( )
```

Get the burst_height parameter.

5.11.3.134 get_ground_alt()

```
double Crane::get_ground_alt ( )
```

Get the ground_alt parameter.

5.11.3.135 get_energy_frac()

```
double Crane::get_energy_frac ( )
```

Get the energy_frac parameter.

5.11.3.136 get_eccentricity()

```
double Crane::get_eccentricity ( )
```

Get the eccentricity parameter.

5.11.3.137 get_air_density()

```
double Crane::get_air_density ( )
```

Get the air_density parameter.

5.11.3.138 get_air_viscosity()

```
double Crane::get_air_viscosity ( )
```

Get the air_viscosity parameter.

5.11.3.139 get_slip_factor()

```
double Crane::get_slip_factor ( )
```

Get the slip_factor parameter.

5.11.3.140 get_davies_num()

```
double Crane::get_davies_num ( )
```

Get the davies_num parameter.

5.11.3.141 get_min_dia()

```
double Crane::get_min_dia ( )
```

Get the min_dia parameter.

5.11.3.142 get_max_dia()

```
double Crane::get_max_dia ( )
```

Get the max_dia parameter.

5.11.3.143 get_mean_dia()

```
double Crane::get_mean_dia ( )
```

Get the mean_dia parameter.

5.11.3.144 get_std_dia()

```
double Crane::get_std_dia ( )
```

Get the std_dia parameter.

5.11.3.145 get_num_bins()

```
int Crane::get_num_bins ( )
```

Get the num_bins parameter.

5.11.3.146 get_cloud_mass()

```
double Crane::get_cloud_mass ( )
```

Get the cloud_mass parameter.

5.11.3.147 get_entrained_mass()

```
double Crane::get_entrained_mass ( )
```

Get the entrained_mass parameter.

5.11.3.148 get_cloud_rise()

```
double Crane::get_cloud_rise ( )
```

Get the cloud_rise parameter.

5.11.3.149 get_cloud_alt()

```
double Crane::get_cloud_alt ( )
```

Get the cloud_alt parameter.

5.11.3.150 get_x_water_vapor()

```
double Crane::get_x_water_vapor ( )
```

Get the x_water_vapor parameter.

5.11.3.151 get_w_water_conds()

```
double Crane::get_w_water_conds ( )
```

Get the w_water_conds parameter.

5.11.3.152 get_s_soil()

```
double Crane::get_s_soil ( )
```

Get the s_soil parameter.

5.11.3.153 get_temperature()

```
double Crane::get_temperature ( )
```

Get the temperature parameter.

5.11.3.154 get_energy()

```
double Crane::get_energy ( )
```

Get the energy parameter.

5.11.3.155 get_current_time()

```
double Crane::get_current_time ( )
```

Get the current_time parameter.

5.11.3.156 get_vapor_pressure()

```
double Crane::get_vapor_pressure ( )
```

Get the vapor_pressure parameter.

5.11.3.157 get_sat_vapor_pressure()

```
double Crane::get_sat_vapor_pressure ( )
```

Get the sat_vapor_pressure parameter.

5.11.3.158 get_solidification_temp()

```
double Crane::get_solidification_temp ( )
```

Get the solidification_temp parameter.

5.11.3.159 get_vaporization_temp()

```
double Crane::get_vaporization_temp ( )
```

Get the vaporization_temp parameter.

5.11.3.160 get_initial_soil_mass()

```
double Crane::get_initial_soil_mass ( )
```

Get the initial_soil_mass parameter.

5.11.3.161 get_initial_soil_vapor()

```
double Crane::get_initial_soil_vapor ( )
```

Get the initial_soil_vapor parameter.

5.11.3.162 get_initial_water_mass()

```
double Crane::get_initial_water_mass ( )
```

Get the initial_water_mass parameter.

5.11.3.163 get_initial_air_mass()

```
double Crane::get_initial_air_mass ( )
```

Get the initial_air_mass parameter.

5.11.3.164 get_current_amb_temp()

```
double Crane::get_current_amb_temp ( )
```

Get the current_amb_temp parameter.

5.11.3.165 get_current_atm_press()

```
double Crane::get_current_atm_press ( )
```

Get the current_atm_press parameter.

5.11.3.166 get_includeShearVel()

```
bool Crane::get_includeShearVel ( )
```

Get the includeShearVel parameter.

5.11.3.167 get_isSaturated()

```
bool Crane::get_isSaturated ( )
```

Get the isSaturated parameter.

5.11.3.168 get_isSolidified()

```
bool Crane::get_isSolidified ( )
```

Get the isSolidified parameter.

5.11.3.169 get_part_size()

```
double Crane::get_part_size (
             int i )
```

Get the i-th particle size parameter.

5.11.3.170 get_settling_rate()

```
double Crane::get_settling_rate (
             double Dj )
```

Get the settling_rate associated with size Dj.

5.11.3.171 get_settling_rate_old()

```
double Crane::get_settling_rate_old (
             double Dj )
```

Get the settling_rate_old associated with size Dj.

5.11.3.172 get_ConsoleOut()

```
bool Crane::get_ConsoleOut ( )
```

Get the ConsoleOut parameter.

5.11.3.173 get_FileOut()

```
bool Crane::get_FileOut ( )
```

Get the FileOut parameter.

5.11.3.174 get_part_conc_var()

```
Matrix<double>& Crane::get_part_conc_var ( )
```

Get the part_conc_var parameter.

5.11.3.175 get_saturation_time()

```
double Crane::get_saturation_time ( )
```

Get the saturation_time parameter.

5.11.3.176 get_solidification_time()

```
double Crane::get_solidification_time ( )
```

Get the solidification_time parameter.

5.11.3.177 get_stabilization_time()

```
double Crane::get_stabilization_time ( )
```

Get the stabilization_time parameter.

5.11.3.178 get_isTight()

```
bool Crane::get_isTight ( )
```

Get the isTight parameter.

5.11.3.179 get_useCustomDist()

```
bool Crane::get_useCustomDist ( )
```

Get the useCustomDist parameter.

5.11.3.180 get_cloud_density()

```
double Crane::get_cloud_density ( )
```

Get the cloud_density parameter.

5.11.3.181 get_horz_rad_change()

```
double Crane::get_horz_rad_change ( )
```

Get the horz_rad_change parameter.

5.11.3.182 get_energy_switch()

```
double Crane::get_energy_switch ( )
```

Get the energy_switch parameter.

5.11.3.183 get_alt_top()

```
double Crane::get_alt_top ( )
```

Get the alt_top parameter.

5.11.3.184 get_alt_bottom()

```
double Crane::get_alt_bottom ( )
```

Get the alt_bottom parameter.

5.11.3.185 get_alt_top_old()

```
double Crane::get_alt_top_old ( )
```

Get the alt_top parameter.

5.11.3.186 get_alt_bottom_old()

```
double Crane::get_alt_bottom_old ( )
```

Get the alt_bottom parameter.

5.11.3.187 get_rise_top()

```
double Crane::get_rise_top ( )
```

Get the rise_top parameter.

5.11.3.188 get_rise_bottom()

```
double Crane::get_rise_bottom ( )
```

Get the rise_bottom parameter.

5.11.3.189 get_part_conc()

```
std::map<double, double>& Crane::get_part_conc ( )
```

Get the part_conc map parameter.

5.11.3.190 get_part_hist()

```
std::map<double, double>& Crane::get_part_hist ( )
```

Get the part_hist map parameter.

5.11.3.191 get_soil_molefrac()

```
std::map<std::string, double>& Crane::get_soil_molefrac ( )
```

Get the soil_molefrac map parameter.

5.11.3.192 get_soil_comp()

```
std::map<std::string, Molecule>& Crane::get_soil_comp ( )
```

Get the soil_comp map parameter.

5.11.3.193 get_soil_atom_frac()

```
std::map<std::string, double>& Crane::get_soil_atom_frac ( )
```

Get the soil_atom_frac map parameter.

5.11.3.194 get_soil_atom()

```
std::map<std::string, Atom>& Crane::get_soil_atom ( )
```

Get the soil_atom map parameter.

5.11.3.195 compute_beta_prime()

```
void Crane::compute_beta_prime (
             double x,
             double s,
             double w )
```

Function to compute ratio of cloud gas density to local density.

5.11.3.196 compute_q_x()

```
void Crane::compute_q_x (
             double x )
```

Function to compute virtual temperature ratio for x.

5.11.3.197 compute_q_xe()

```
void Crane::compute_q_xe (
             double xe )
```

Function to compute virtual temperature ratio for xe.

5.11.3.198 compute_apparent_temp()

```
void Crane::compute_apparent_temp (
             double T,
             double x )
```

Function to compute apparent temp (T∗) given T and x.

5.11.3.199 compute_apparent_amb_temp()

```
void Crane::compute_apparent_amb_temp (
             double Te,
             double xe )
```

Function to compute apparent amb temp (Te∗) given Te and xe.

5.11.3.200 compute_char_vel()

```
void Crane::compute_char_vel (
             double u,
             double E )
```

Function to compute characteristic velocity.

5.11.3.201 compute_air_viscosity()

```
void Crane::compute_air_viscosity (
             double T )
```

Function to compute air viscosity.

5.11.3.202 compute_vapor_pressure()

```
void Crane::compute_vapor_pressure (
             double P,
             double x )
```

Function to compute vapor pressure in cloud given P and x.

5.11.3.203 compute_sat_vapor_pressure()

```
void Crane::compute_sat_vapor_pressure (
             double T )
```

Function to compute saturation vapor pressure given T.

5.11.3.204 compute_xe()

```
void Crane::compute_xe (
             double Te,
             double P,
             double HR )
```

Function to compute ambient air water ratio given Te, P, and HR.

5.11.3.205 compute_air_density()

```
void Crane::compute_air_density (
             double P,
             double x,
             double T )
```

Function to compute air density given P, x, and T.

5.11.3.206 compute_spec_heat_entrain()

```
void Crane::compute_spec_heat_entrain (
             double T )
```

Function to compute specific heat of entrained air given T.

5.11.3.207 compute_spec_heat_water()

```
void Crane::compute_spec_heat_water (
             double T )
```

Function to compute specific heat of water vapor given T.

5.11.3.208 compute_spec_heat_conds()

```
void Crane::compute_spec_heat_conds (
             double T )
```

Function to compute specific heat of condensed matter given T.

5.11.3.209 compute_actual_spec_heat()

```
void Crane::compute_actual_spec_heat (
             double T,
             double x )
```

Function to compute the actual specific heat of the cloud given T and x.

5.11.3.210 compute_k_temp()

```
void Crane::compute_k_temp (
             double T )
```

Function to compute temperature factor based on given T.

5.11.3.211 compute_mean_spec_heat()

```
void Crane::compute_mean_spec_heat (
             double T,
             double x,
             double s,
             double w )
```

Function to compute mean specific heat given T, x, s, and w.

5.11.3.212 compute_cloud_volume()

```
void Crane::compute_cloud_volume (
             double m,
             double x,
             double s,
             double w,
             double T,
             double P )
```

Function to compute cloud volume.

5.11.3.213 compute_cloud_density()

```
void Crane::compute_cloud_density (
             double m,
             double x,
             double s,
             double w,
             double T,
             double P )
```

Function to compute cloud density.

5.11.3.214 compute_vert_rad()

```
void Crane::compute_vert_rad (
             double z )
```

Function to compute vertical cloud radius given z.

5.11.3.215 compute_horz_rad()

```
void Crane::compute_horz_rad (
             double m,
             double x,
             double s,
             double w,
             double T,
             double P,
             double z )
```

Function to compute horizontal radius.

5.11.3.216 compute_sigma_turbulence()

```
void Crane::compute_sigma_turbulence (
             double E,
             double z )
```

Function to compute sigma turbulence given E and z.

5.11.3.217 compute_surf_area()

```
void Crane::compute_surf_area (
             double m,
             double x,
             double s,
             double w,
             double T,
             double P,
             double z )
```

Function to compute cloud surface area.

5.11.3.218 compute_shear_vel()

```
void Crane::compute_shear_vel (
            Matrix< double > v_top,
            Matrix< double > v_bot )
```

Function to compute the shear_vel based on v_top and v_bot

5.11.3.219 compute_shear_ratio()

```
void Crane::compute_shear_ratio (
            double m,
            double x,
            double s,
            double w,
            double T,
            double P,
            double z,
            double u,
            double E,
            Matrix< double > v_top,
            Matrix< double > v_bot )
```

Function to compute the shear_ratio based on the fundamental variables and atmospheric parameters.

5.11.3.220 compute_slip_factor()

```
void Crane::compute_slip_factor (
            double Dj,
            double T,
            double P )
```

Function to compute the slip_factor given Dj, T, and P.

5.11.3.221 compute_davies_num()

```
void Crane::compute_davies_num (
            double Dj,
            double m,
            double x,
            double s,
            double w,
            double T,
            double P )
```

Function to compute davies_num given the conditions.

5.11.3.222 compute_settling_rate()

```
void Crane::compute_settling_rate (
             double Dj,
             double m,
             double x,
             double s,
             double w,
             double T,
             double P )
```

Function to compute the settling rates of specific particle size

5.11.3.223 compute_total_mass_fallout_rate()

```
void Crane::compute_total_mass_fallout_rate (
             double m,
             double x,
             double s,
             double w,
             double T,
             double P,
             double z,
             const Matrix< double > & n )
```

Function to compute total_mass_fallout_rate based on all given variables and parameters.

5.11.3.224 compute_energy_switch()

```
void Crane::compute_energy_switch (
             double W )
```

Function to compute energy switch from W.

5.11.3.225 compute_k()

```
void Crane::compute_k (
             double W )
```

Function to compute cloud rise yield from W.

5.11.3.226 compute_k2()

```
void Crane::compute_k2 (
             double W )
```

Function to compute power function yield from W.

5.11.3.227 compute_mu()

```
void Crane::compute_mu (
             double W )
```

Function to compute energy yield from W.

5.11.3.228 compute_force_factor()

```
void Crane::compute_force_factor (
             double W )
```

Function to compute the force factor from W.

5.11.3.229 compute_equil_temp()

```
void Crane::compute_equil_temp (
             double W )
```

Function to compute T_ri from W.

5.11.3.230 create_part_hist()

```
void Crane::create_part_hist (
             double min,
             double max,
             int size,
             double avg,
             double std )
```

Function to compute normalized particle histogram.

5.11.3.231 compute_det_alt()

```
void Crane::compute_det_alt (
             double gz,
             double hb )
```

Function to compute the det_alt given ground alt and burst height.

5.11.3.232 compute_initial_cloud_alt()

```
void Crane::compute_initial_cloud_alt (
            double W,
            double gz,
            double hb )
```

Function to compute initial cloud_alt.

5.11.3.233 compute_initial_current_time()

```
void Crane::compute_initial_current_time (
            double W,
            double gz,
            double hb )
```

Function to compute initial time after detonation.

5.11.3.234 compute_initial_temperature()

```
void Crane::compute_initial_temperature (
            double W,
            double gz,
            double hb )
```

Function to compute initial temperature of fallout cloud.

5.11.3.235 compute_initial_soil_mass()

```
void Crane::compute_initial_soil_mass (
            double W,
            double gz,
            double hb )
```

Function to compute initial soil mass in cloud.

5.11.3.236 compute_initial_part_hist()

```
void Crane::compute_initial_part_hist (
            double W,
            double gz,
            double hb,
            int size )
```

Function to compute part_hist from W, gz, hb, and size.

5.11.3.237 compute_initial_air_mass()

```
void Crane::compute_initial_air_mass (
             double W,
             double gz,
             double hb )
```

Function to compute initial air mass in cloud.

5.11.3.238 compute_initial_water_mass()

```
void Crane::compute_initial_water_mass (
             double W,
             double gz,
             double hb )
```

Function to compute initial water mass in cloud.

5.11.3.239 compute_initial_entrained_mass()

```
void Crane::compute_initial_entrained_mass (
             double W,
             double gz,
             double hb )
```

Function to compute initial entrained mass in cloud.

5.11.3.240 compute_initial_cloud_mass()

```
void Crane::compute_initial_cloud_mass (
             double W,
             double gz,
             double hb )
```

Function to compute initial cloud mass.

5.11.3.241 compute_initial_s_soil()

```
void Crane::compute_initial_s_soil (
             double W,
             double gz,
             double hb )
```

Function to compute initial soil fraction.

5.11.3.242 compute_initial_x_water_vapor()

```
void Crane::compute_initial_x_water_vapor (
            double W,
            double gz,
            double hb )
```

Function to compute initial water fraction.

5.11.3.243 compute_initial_cloud_volume()

```
void Crane::compute_initial_cloud_volume (
            double W,
            double gz,
            double hb )
```

Function to compute initial cloud volume.

5.11.3.244 compute_initial_horz_rad()

```
void Crane::compute_initial_horz_rad (
            double W,
            double gz,
            double hb )
```

Function to compute initial horizontal radius of cloud.

5.11.3.245 compute_initial_vert_rad()

```
void Crane::compute_initial_vert_rad (
            double W,
            double gz,
            double hb )
```

Function to compute initial vertical cloud radius.

5.11.3.246 compute_initial_cloud_rise()

```
void Crane::compute_initial_cloud_rise (
            double W,
            double gz,
            double hb )
```

Function to compute initial cloud rise velocity.

5.11.3.247 compute_initial_energy()

```
void Crane::compute_initial_energy (
            double W,
            double gz,
            double hb )
```

Function to compute initial energy for cloud.

5.11.3.248 compute_initial_part_conc()

```
void Crane::compute_initial_part_conc (
            double W,
            double gz,
            double hb,
            int size )
```

Function to compute initial particle concentrations.

5.11.3.249 compute_initial_virtual_mass()

```
void Crane::compute_initial_virtual_mass (
            double W,
            double gz,
            double hb )
```

Function to compute initial virtual mass of cloud.

5.11.3.250 compute_adjusted_height()

```
void Crane::compute_adjusted_height (
            double W,
            double gz,
            double hb )
```

Function to compute the adjusted height of the cloud.

5.11.3.251 delete_particles()

```
void Crane::delete_particles ( )
```

Function to remove all existing particle information.

5.11.3.252 compute_spec_heat_entrain_integral()

```
void Crane::compute_spec_heat_entrain_integral (
             double T,
             double Te )
```

Function to compute integral of spec_heat_entrain from Te to T.

5.11.3.253 return_spec_heat_water_integral()

```
double Crane::return_spec_heat_water_integral (
             double T,
             double Te )
```

Function to return integral of spec_heat_water from Te to T.

5.11.3.254 return_spec_heat_conds_integral()

```
double Crane::return_spec_heat_conds_integral (
             double T,
             double Te )
```

Function to return integral of spec_heat_conds from Te to T.

5.11.3.255 add_amb_temp()

```
void Crane::add_amb_temp (
             double z,
             double Te )
```

Function to add a temperature Te at altitude z.

5.11.3.256 add_atm_press()

```
void Crane::add_atm_press (
             double z,
             double P )
```

Function to add a pressure P at altitude z.

5.11.3.257 add_rel_humid()

```
void Crane::add_rel_humid (
             double z,
             double HR )
```

Function to add a relative humidity HR at altitude z.

5.11.3.258 add_wind_vel()

```
void Crane::add_wind_vel (
             double z,
             double vx,
             double vy )
```

Function to add a wind velocity by components at altitude z.

5.11.3.259 create_default_atmosphere()

```
void Crane::create_default_atmosphere ( )
```

Function to create a default atmosphere from -600 m to 49,800 m.

5.11.3.260 create_default_wind_profile()

```
void Crane::create_default_wind_profile ( )
```

Function to create a default wind profile from 216 m to 31,023 m.

5.11.3.261 delete_atmosphere()

```
void Crane::delete_atmosphere ( )
```

Function to remove existing atmosphere profile from memory.

5.11.3.262 delete_wind_profile()

```
void Crane::delete_wind_profile ( )
```

Function to remove existing wind profile from memory.

5.11.3.263 return_amb_temp()

```
double Crane::return_amb_temp (
             double z )
```

Function to return the ambient temperature (K) given an altitude z (m)

5.11.3.264 return_atm_press()

```
double Crane::return_atm_press (
             double z )
```

Function to return the atmospheric pressure (Pa) given an altitude z (m)

5.11.3.265 return_rel_humid()

```
double Crane::return_rel_humid (
             double z )
```

Function to return the relative humidity (%) given an altitude z (m)

5.11.3.266 return_wind_vel()

```
Matrix<double> Crane::return_wind_vel (
             double z )
```

Function to return wind velocity vector (m/s) given an altitude z (m)

5.11.3.267 return_wind_speed()

```
double Crane::return_wind_speed (
             double z )
```

Function to return wind speed (m/s) given an altitude (m)

5.11.3.268 compute_current_amb_temp()

```
void Crane::compute_current_amb_temp (
             double z )
```

Function to set the current_amb_temp parameter given an altitude (m)

5.11.3.269 compute_current_atm_press()

```
void Crane::compute_current_atm_press (
             double z )
```

Function to set the current_amb_press parameter given an altitude (m)

5.11.3.270 add_solid_param()

```
void Crane::add_solid_param (
             std::string name,
             int pow,
             double param )
```

Function to add a solidification parameter based on oxide name.

5.11.3.271 add_vapor_param()

```
void Crane::add_vapor_param (
             std::string name,
             int pow,
             double param )
```

Function to add a vaporization parameter based on oxide name.

5.11.3.272 create_default_soil_components()

```
void Crane::create_default_soil_components ( )
```

Function to setup the default soil component parameters.

5.11.3.273 delete_soil_components()

```
void Crane::delete_soil_components ( )
```

Function to remove all soil components and parameters.

5.11.3.274 add_soil_component()

```
void Crane::add_soil_component (
             std::string name,
             double frac )
```

Function to add soil components and corresponding molefraction.

5.11.3.275 verify_soil_components()

```
void Crane::verify_soil_components ( )
```

Function to check soil components for errors and correct.

5.11.3.276 compute_solidification_temp()

```
void Crane::compute_solidification_temp ( )
```

Function to compute soil solidification temperature based on components.

5.11.3.277 compute_vaporization_temp()

```
void Crane::compute_vaporization_temp ( )
```

Function to compute soil vaporization temperature based on components.

5.11.3.278 compute_initial_soil_vapor()

```
void Crane::compute_initial_soil_vapor ( )
```

Function to compute the initial vaporized soil mass (kg)

5.11.3.279 compute_alt_top()

```
void Crane::compute_alt_top (
             double z,
             double Hc )
```

Function to compute cloud cap top given center z and height Hc.

5.11.3.280 compute_alt_bottom()

```
void Crane::compute_alt_bottom (
             double z,
             double Hc )
```

Function to compute cloud cap bottom given center z and height Hc.

5.11.3.281 compute_rise_top()

```
void Crane::compute_rise_top (
             double z_new,
             double z_old,
             double dt )
```

Function to compute cloud cap top rise given changes in top altitudes.

5.11.3.282 compute_rise_bottom()

```
void Crane::compute_rise_bottom (
             double z_new,
             double z_old,
             double dt )
```

Function to compute cloud cap bottom rise given changes in bottom altitudes.

5.11.3.283 return_parcel_alt_top()

```
Matrix<double>& Crane::return_parcel_alt_top ( )
```

Function to return matrix of parcels and particle sizes for top of parcel.

5.11.3.284 return_parcel_alt_bot()

```
Matrix<double>& Crane::return_parcel_alt_bot ( )
```

Function to return matrix of parcels and particle sizes for bottom of parcel.

5.11.3.285 return_parcel_rad_top()

```
Matrix<double>& Crane::return_parcel_rad_top ( )
```

Function to return matrix of parcels and particle sizes for radius of top of parcel.

5.11.3.286 return_parcel_rad_bot()

```
Matrix<double>& Crane::return_parcel_rad_bot ( )
```

Function to return matrix of parcels and particle sizes for radius of bottom of parcel.

5.11.3.287 return_parcel_conc()

```
Matrix<double>& Crane::return_parcel_conc ( )
```

Function to return matrix of parcel concentrations (row = parcel, col = particle size)

5.11.3.288 read_atmosphere_profile()

```
int Crane::read_atmosphere_profile (
             const char * profile )
```

Function to read atmospheric data from input file (return 0 on success and -1 on failure)

5.11.3.289 read_conditions()

```
int Crane::read_conditions (
             Dove & dove,
             yaml_cpp_class & yaml )
```

Function to read yaml input for Simulation Conditions.

5.11.3.290 establish_initial_conditions()

```
void Crane::establish_initial_conditions (
             Dove & dove,
             double W,
             double gz,
             double hb,
             int bins,
             bool includeShear,
             bool isTight )
```

Function will establish initial conditions, setup variables names, and register functions.

5.11.3.291 read_dove_options()

```
int Crane::read_dove_options (
             Dove & dove,
             FILE * file,
             yaml_cpp_class & yaml )
```

Function to read yaml input for ODE options.

5.11.3.292 establish_dove_options()

```
void Crane::establish_dove_options (
            Dove & dove,
            FILE * file,
            bool fileout,
            bool consoleout,
            integrate_subtype inttype,
            timestep_type timetype,
            precond_type type,
            double tol,
            double dtmin,
            double dtmax,
            double dtmin_conv,
            double t_out,
            double endtime )
```

Function to establish DOVE conditions and options.

5.11.3.293 read_pjfnk_options()

```
int Crane::read_pjfnk_options (
            Dove & dove,
            yaml_cpp_class & yaml )
```

Function to read yaml input for solver options.

5.11.3.294 establish_pjfnk_options()

```
void Crane::establish_pjfnk_options (
            Dove & dove,
            krylov_method lin_method,
            linesearch_type linesearch,
            bool linear,
            bool precon,
            bool nl_out,
            bool l_out,
            int max_nlit,
            int max_lit,
            int restart,
            int recursive,
            double nl_abstol,
            double nl_reltol,
            double l_abstol,
            double l_reltol )
```

Function to establish PJFNK conditions and options.

5.11.3.295 read_wind_profile()

```
int Crane::read_wind_profile (
            yaml_cpp_class & yaml )
```

Function to read yaml input for solver options.

5.11.3.296 estimate_parameters()

```
void Crane::estimate_parameters (
            Dove & dove )
```

Function to estimate all parameters prior to simulating a single timestep.

5.11.3.297 perform_postprocessing()

```
void Crane::perform_postprocessing (
            Dove & dove )
```

Function to compute post-processing information to establish cloud stem.

5.11.3.298 store_variables()

```
void Crane::store_variables (
            Dove & dove )
```

Function to store all variable values to be updated from Dove simulations.

5.11.3.299 print_header()

```
void Crane::print_header (
            Dove & dove )
```

Function to print addition headers to the output file.

5.11.3.300 print_information()

```
void Crane::print_information (
            Dove & dove,
            bool initialPhase )
```

Function to print additional information to output file.

5.11.3.301 run_crane_simulation()

```
int Crane::run_crane_simulation (
             Dove & dove )
```

Function to run the simulation to completion.

5.11.4 Member Data Documentation

5.11.4.1 eps

```
double Crane::eps  [protected]
```

Ratio of molecular wieghts for water-vapor and dry air (eps)

5.11.4.2 grav

```
double Crane::grav  [protected]
```

Acceleration from gravity (m/s^2) (g)

5.11.4.3 k

```
double Crane::k  [protected]
```

Dimensionless initial cloud rise yield (k)

5.11.4.4 k2

```
double Crane::k2  [protected]
```

Dimensionless power function yield (k2)

5.11.4.5 k3

```
double Crane::k3  [protected]
```

Dimensionless kinetic energy yield (k3)

5.11.4.6 k6

```
double Crane::k6  [protected]
```

Dimensionless wind shear factor (k6)

5.11.4.7 mu

```
double Crane::mu  [protected]
```

Dimensionless energy yield for given explosion (mu)

5.11.4.8 k_temp

```
double Crane::k_temp  [protected]
```

Dimensionless temperature factor for specific heat (k(T,T_ri))

5.11.4.9 apparent_temp

```
double Crane::apparent_temp  [protected]
```

Apparent temperature of cloud (K) (T*)

5.11.4.10 apparent_amb_temp

```
double Crane::apparent_amb_temp  [protected]
```

Apparent ambient temperature at specific altitude (K) (Te*)

5.11.4.11 q_x

```
double Crane::q_x  [protected]
```

Ratio of apparent to actual temperature in the cloud (q(x))

5.11.4.12 q_xe

```
double Crane::q_xe  [protected]
```

Ratio of apparent to actual ambient temperature (q(xe))

5.11.4.13 beta_prime

```
double Crane::beta_prime  [protected]
```

Ratio of cloud gas density to total cloud density (beta')

5.11.4.14 xe

```
double Crane::xe  [protected]
```

Ratio of water-vapor to dry air for ambient air at given altitude (xe)

5.11.4.15 char_vel

```
double Crane::char_vel  [protected]
```

Characteristic velocity for cloud rise (m/s) (v)

5.11.4.16 vert_rad

```
double Crane::vert_rad  [protected]
```

Vertical radius of the cloud shape (m) (Hc)

5.11.4.17 horz_rad

```
double Crane::horz_rad  [protected]
```

Horizontal radius of the cloud shape (m) (Rc)

5.11.4.18 virtual_mass

```
double Crane::virtual_mass  [protected]
```

Initial virtual mass of the cloud (kg) (m_i')

5.11.4.19 gas_const

```
double Crane::gas_const  [protected]
```

Gas law constant for dry air (J/kg/K) (Ra)

5.11.4.20 latent_heat

```
double Crane::latent_heat  [protected]
```

Latent heat of evaporation of water (or ice) (J/kg) (L)

5.11.4.21 sigma_turbulence

```
double Crane::sigma_turbulence  [protected]
```

Rate of dissipation of turbulent energy per mass (K/s) (SIGMA)

5.11.4.22 mean_spec_heat

```
double Crane::mean_spec_heat  [protected]
```

Weighted mean specific heat of the cloud (J/kg/K) (c_pBAR)

5.11.4.23 actual_spec_heat

```
double Crane::actual_spec_heat  [protected]
```

Actual specific heat of the cloud mass (J/kg/K) (c_p(T))

5.11.4.24 spec_heat_water

```
double Crane::spec_heat_water  [protected]
```

Specific heat of water vapor (J/kg/K) (c_pw(T))

5.11.4.25 spec_heat_entrain

```
double Crane::spec_heat_entrain  [protected]
```

Specific heat of entrained air (J/kg/K) (c_pa(T))

5.11.4.26 spec_heat_entrain_integral

```
double Crane::spec_heat_entrain_integral  [protected]
```

Integrated Specific heat of entrained air (J/kg) (integral(c_pa(T)∗dT))

5.11.4.27 spec_heat_conds

```
double Crane::spec_heat_conds  [protected]
```

Specific heat of condensed matter (J/kg/K) (c_s(T))

5.11.4.28 cloud_volume

```
double Crane::cloud_volume  [protected]
```

Volume of the debris cloud (m^3) (V)

5.11.4.29 equil_temp

```
double Crane::equil_temp  [protected]
```

Initial thermal equilibrium temperature (K) (T_ri)

5.11.4.30 total_mass_fallout_rate

```
double Crane::total_mass_fallout_rate  [protected]
```

Overall rate of particle fallout (kg/s) (p(t))

5.11.4.31 surf_area

```
double Crane::surf_area  [protected]
```

Surface area of the cloud (m^2) (S)

5.11.4.32 shear_ratio

```
double Crane::shear_ratio  [protected]
```

(Surf_area/Vol)∗mu∗char_vel {Can be swapped to include shear (1/s)} ((S/V)∗mu∗v)

5.11.4.33 shear_vel

```
double Crane::shear_vel  [protected]
```

Magnitude of shear velocity impacting cloud shape (m/s) (v_s)

5.11.4.34 part_density

```
double Crane::part_density [protected]
```

Average density of soil particles in cloud (kg/m^3) (rho_p)

5.11.4.35 adjusted_height

```
double Crane::adjusted_height [protected]
```

Initial adjusted height of the cloud (m) (z')

5.11.4.36 bomb_yield

```
double Crane::bomb_yield [protected]
```

Size of nuclear bomb or explosion (kT) (W)

5.11.4.37 force_factor

```
double Crane::force_factor [protected]
```

Dimensionless factor of explosive force (F)

5.11.4.38 det_alt

```
double Crane::det_alt [protected]
```

Height/altitude of detontation above mean sea level (m) (h)

5.11.4.39 burst_height

```
double Crane::burst_height [protected]
```

Height of detonation above ground level (m) (h_b)

5.11.4.40 ground_alt

```
double Crane::ground_alt [protected]
```

Height/altitude of the ground level above sea level (m) (h_gz)

5.11.4.41 energy_frac

```
double Crane::energy_frac  [protected]
```

Fraction of energy available to heat the air (phi)

5.11.4.42 eccentricity

```
double Crane::eccentricity  [protected]
```

Eccentricity of an oblate spheriod (default = 0.75) (e)

5.11.4.43 air_density

```
double Crane::air_density  [protected]
```

Density of air in cloud (kg/m^3) (rho)

5.11.4.44 air_viscosity

```
double Crane::air_viscosity  [protected]
```

Viscosity of air in cloud (kg/m/s) (eta)

5.11.4.45 slip_factor

```
double Crane::slip_factor  [protected]
```

Slip factor for particle settling (-) (s)

5.11.4.46 davies_num

```
double Crane::davies_num  [protected]
```

Unitless number for particle settling analysis (-) (ND)

5.11.4.47 vapor_pressure

```
double Crane::vapor_pressure  [protected]
```

Vapor pressure inside the cloud at cloud altitude (Pa) (Pv)

5.11.4.48 sat_vapor_pressure

```
double Crane::sat_vapor_pressure  [protected]
```

Saturation vapor pressure inside the cloud (Pa) (Pws)

5.11.4.49 solidification_temp

```
double Crane::solidification_temp  [protected]
```

Temperature at which soil debris will solidify (K) (Ts)

5.11.4.50 vaporization_temp

```
double Crane::vaporization_temp  [protected]
```

Temperature at which soil debris will vaporize (K) (Tm)

5.11.4.51 initial_soil_mass

```
double Crane::initial_soil_mass  [protected]
```

Initial value for soil mass in debris cloud (kg) (m_ri)

5.11.4.52 initial_soil_vapor

```
double Crane::initial_soil_vapor  [protected]
```

Initial value for the vaporized amount of soil (kg) (m_vi)

5.11.4.53 initial_water_mass

```
double Crane::initial_water_mass  [protected]
```

Initial value for water mass in debris cloud (kg) (m_wi)

5.11.4.54 initial_air_mass

```
double Crane::initial_air_mass  [protected]
```

Initial value for air mass in debris cloud (kg) (m_ai)

5.11.4.55 current_amb_temp

```
double Crane::current_amb_temp  [protected]
```

Current value of ambient temperature (set based on atm profile) (Te)

5.11.4.56 current_atm_press

```
double Crane::current_atm_press  [protected]
```

Current value of atmospheric pressure (set based on atm profile) (P)

5.11.4.57 includeShearVel

```
bool Crane::includeShearVel  [protected]
```

Boolean statement used to include (true) or ignore (false) wind shear.

5.11.4.58 isSaturated

```
bool Crane::isSaturated  [protected]
```

Boolean state used to determine whether or not to use Saturated Functions.

5.11.4.59 isSolidified

```
bool Crane::isSolidified  [protected]
```

Boolean state used to determine whether or not the soils have solidified.

5.11.4.60 isTight

```
bool Crane::isTight  [protected]
```

Boolean state used to determine whether or not to use Tight Coulpling.

5.11.4.61 useCustomDist

```
bool Crane::useCustomDist  [protected]
```

Boolean state used to determine whether or not to use a Custom Particle Distribution.

5.11.4.62 ConsoleOut

```
bool Crane::ConsoleOut  [protected]
```

Boolean state used to determine whether or not to include console output.

5.11.4.63 FileOut

```
bool Crane::FileOut  [protected]
```

Boolean state used to determine whether or not to include file output.

5.11.4.64 amb_temp

```
std::map<double, double> Crane::amb_temp  [protected]
```

Ambient Temperature (K) at various altitudes (m) (Te)

5.11.4.65 atm_press

```
std::map<double, double> Crane::atm_press  [protected]
```

Atmospheric Pressure (Pa) at various altitudes (m) (P)

5.11.4.66 rel_humid

```
std::map<double, double> Crane::rel_humid  [protected]
```

Relative Humidity (%) at various altitudes (m) (HR)

5.11.4.67 wind_vel

```
std::map<double, Matrix<double> > Crane::wind_vel  [protected]
```

Wind Velocities (m/s) at various altitudes (m) (v_a)

Velocities stored in x (0,0) and y (1,0) components at a given altitude

5.11.4.68 part_hist

```
std::map<double, double> Crane::part_hist  [protected]
```

Normalized Histogram of Particle Distribution by size (um)

5.11.4.69 settling_rate

```
std::map<double, double> Crane::settling_rate  [protected]
```

Particle settling rate (m/s) by particle size (um) (f_j)

5.11.4.70 part_size

```
std::vector<double> Crane::part_size  [protected]
```

Particle sizes listed in order from smallest to largest (um)

5.11.4.71 min_dia

```
double Crane::min_dia  [protected]
```

Minimum particle diameter for distribution (um) (dmin)

5.11.4.72 max_dia

```
double Crane::max_dia  [protected]
```

Maximum particle diameter for distribution (um) (dmax)

5.11.4.73 mean_dia

```
double Crane::mean_dia  [protected]
```

Mean particle diameter for distribution (um) (d50)

5.11.4.74 std_dia

```
double Crane::std_dia  [protected]
```

Standard deviation for lognormal distribution (sigma)

5.11.4.75 num_bins

```
int Crane::num_bins  [protected]
```

Number of desired size bins for particles (N)

5.11.4.76 soil_atom

```
std::map<std::string, Atom> Crane::soil_atom  [protected]
```

Stores a map of soil atom components (key is the atom)

5.11.4.77 soil_atom_frac

```
std::map<std::string, double> Crane::soil_atom_frac  [protected]
```

Stores a map of soil atom components (key is the atom)

5.11.4.78 soil_comp

```
std::map<std::string, Molecule> Crane::soil_comp  [protected]
```

Stores the soil component molecule information.

5.11.4.79 soil_molefrac

```
std::map<std::string, double> Crane::soil_molefrac  [protected]
```

Stores the molefraction of the soil components

5.11.4.80 solid_params

```
std::unordered_map<std::string, std::map<int, double> > Crane::solid_params  [protected]
```

Polynominal parameters for specific oxides in soil used to determine the solidification temperature.

5.11.4.81 vapor_params

```
std::unordered_map<std::string, std::map<int, double> > Crane::vapor_params  [protected]
```

Polynominal parameters for specific oxides in soil used to determine the vaporization temperature.

5.11.4.82 cloud_mass

```
double Crane::cloud_mass  [protected]
```

List of Variables to be solved for by Dove.

These variables are stored internally by Dove, but will be placed into these parameters below after completing a solve for convenience. Total mass of the debris cloud (Mg) (m)

5.11.4.83 entrained_mass

```
double Crane::entrained_mass  [protected]
```

Mass of entrained materials in debris cloud (Mg) (m_ent)

5.11.4.84 cloud_rise

```
double Crane::cloud_rise  [protected]
```

Rate of cloud rise through atmosphere (m/s) (u)

5.11.4.85 cloud_alt

```
double Crane::cloud_alt  [protected]
```

Altitude of the cloud above mean sea level (m) (z)

5.11.4.86 x_water_vapor

```
double Crane::x_water_vapor  [protected]
```

Mixing ratio for water vapor to dry air (kg/kg) (x)

5.11.4.87 w_water_conds

```
double Crane::w_water_conds  [protected]
```

Mixing ratio for condensed water to dry air (kg/kg) (w)

5.11.4.88 s_soil

```
double Crane::s_soil  [protected]
```

Mixing ratio for suspended soils to dry air (kg/kg) (s)

5.11.4.89 temperature

```
double Crane::temperature  [protected]
```

Temperature of the air in the cloud (K) (T)

5.11.4.90 energy

```
double Crane::energy  [protected]
```

Mass-less Kinetic Energy in the cloud (J/kg) (E)

5.11.4.91 part_conc

```
std::map<double, double> Crane::part_conc  [protected]
```

Number of particles per volume (Gp/m^3) given size (um) (n_j(t))

5.11.4.92 current_time

```
double Crane::current_time  [protected]
```

Current time since explosion (s) (t)

5.11.4.93 part_conc_var

```
Matrix<double> Crane::part_conc_var  [protected]
```

Storage matrix for particle concentrations (Gp/m^3) in order of size.

5.11.4.94 saturation_time

```
double Crane::saturation_time  [protected]
```

Time at which saturation has occurred (s)

5.11.4.95 solidification_time

```
double Crane::solidification_time  [protected]
```

Time at which the melted debris has solidified (s)

5.11.4.96 stabilization_time

```
double Crane::stabilization_time  [protected]
```

Time at which the debris cloud has stabilized (s)

5.11.4.97 cloud_density

```
double Crane::cloud_density  [protected]
```

Density of the cloud materials (kg/m^3)

5.11.4.98 horz_rad_change

```
double Crane::horz_rad_change  [protected]
```

Change in horizontal radius between time steps (m) (d(Rc))

5.11.4.99 energy_switch

```
double Crane::energy_switch  [protected]
```

Energy switch parameter for determining termination (J/kg)

5.11.4.100 t_count

```
double Crane::t_count  [protected]
```

Place holder for a time variable to determine when output is printed.

5.11.4.101 alt_top

```
double Crane::alt_top  [protected]
```

Top of the cloud cap (m) (zt)

5.11.4.102 alt_bottom

```
double Crane::alt_bottom  [protected]
```

Bottom of the cloud cap (m) (zb)

5.11.4.103 alt_top_old

```
double Crane::alt_top_old  [protected]
```

Old top of the cloud cap (m) (zt_old)

5.11.4.104 alt_bottom_old

```
double Crane::alt_bottom_old [protected]
```

Old bottom of the cloud cap (m) (zb_old)

5.11.4.105 rise_top

```
double Crane::rise_top [protected]
```

Cloud rise for top of cloud cap (m/s) (ut)

5.11.4.106 rise_bottom

```
double Crane::rise_bottom [protected]
```

Cloud rise for the bottom of cloud cap (m/s) (ub)

5.11.4.107 parcel_alt_top

```
Matrix<double> Crane::parcel_alt_top [protected]
```

Top of stem of ith parcel for jth particle size (m) (zt_ij)

5.11.4.108 parcel_alt_bot

```
Matrix<double> Crane::parcel_alt_bot [protected]
```

Bottom of the stem of ith parcel for jth particle size (m) (zb_ij)

5.11.4.109 parcel_rad_top

```
Matrix<double> Crane::parcel_rad_top [protected]
```

Radius of the Top of stem of ith parcel for jth particle size (m) (Rt_ij)

5.11.4.110 parcel_rad_bot

```
Matrix<double> Crane::parcel_rad_bot [protected]
```

Radius of the Bottom of the stem of ith parcel for jth particle size (m) (Rb_ij)

5.11.4.111 settling_rate_old

```
std::map<double, double> Crane::settling_rate_old  [protected]
```

Old Particle settling rate (m/s) by particle size (um) (f_j)

5.11.4.112 parcel_conc

```
Matrix<double> Crane::parcel_conc  [protected]
```

Concentration of dust inside each parcel (Gp/m^3)

5.11.4.113 CloudFile

```
FILE* Crane::CloudFile  [protected]
```

Output file to show help plot cloud growth over time.

5.11.4.114 t_cloud_out

```
double Crane::t_cloud_out  [protected]
```

Time to print out cloud profiles to output file.

5.11.4.115 t_cloud_count

```
double Crane::t_cloud_count  [protected]
```

Counter for the cloud output frequency.

The documentation for this class was generated from the following file:

- crane.h

5.12 CROW_DATA Struct Reference

Primary data structure for the CROW routines.

```
#include <crow.h>
```

Public Attributes

- std::unordered_map< int, ConstReaction > const_reacts

    Map of constant reaction objects used in CROW (access by variable index)
- std::unordered_map< int, MultiConstReaction > multi_const_reacts

    Map of multiple constant reaction objects used in CROW (access by variable index)
- std::unordered_map< int, InfiniteBath > inf_bath

    Map of Infinite Bath objects used in CROW (access by variable index)
- Dove SolverInfo

    Dove object that holds all information associated with the solver.
- FILE ∗ OutputFile

    Pointer to the output file for CROW.
- bool FileOutput = true

    Boolean to determine whether or not to print results to a file.
- yaml_cpp_class yaml_object

    yaml object to read and access digitized yaml documents (see yaml_wrapper.h)

5.12.1 Detailed Description

Primary data structure for the CROW routines.

This is a c-style data structure used to house all CROW data necessary to perform simulations on systems of non-linear reaction equations. It is the primary structure that will interface with DOVE and be passed to the DOVE object as the user_data structure. Nested within this structure will be all the parameter information necessary to delineate and evaluate the functions registered in DOVE.

5.12.2 Member Data Documentation

5.12.2.1 const_reacts

```
std::unordered_map<int, ConstReaction> CROW_DATA::const_reacts
```

Map of constant reaction objects used in CROW (access by variable index)

5.12.2.2 multi_const_reacts

```
std::unordered_map<int, MultiConstReaction> CROW_DATA::multi_const_reacts
```

Map of multiple constant reaction objects used in CROW (access by variable index)

5.12.2.3 inf_bath

```
std::unordered_map<int, InfiniteBath> CROW_DATA::inf_bath
```

Map of Infinite Bath objects used in CROW (access by variable index)

5.12.2.4 SolverInfo

```
Dove CROW_DATA::SolverInfo
```

Dove object that holds all information associated with the solver.

5.12.2.5 OutputFile

```
FILE* CROW_DATA::OutputFile
```

Pointer to the output file for CROW.

5.12.2.6 FileOutput

```
bool CROW_DATA::FileOutput = true
```

Boolean to determine whether or not to print results to a file.

5.12.2.7 yaml_object

```
yaml_cpp_class CROW_DATA::yaml_object
```

yaml object to read and access digitized yaml documents (see yaml_wrapper.h)

The documentation for this struct was generated from the following file:

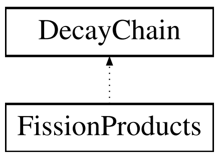
- crow.h

5.13 DecayChain Class Reference

DecayChain object to hold and store a set of unique isotopes in a branched decay chain.

```
#include <ibis.h>
```

Inheritance diagram for DecayChain:



Public Member Functions

- DecayChain ()

    Default constructor.
- ∼DecayChain ()

    Default destructor.
- void DisplayList ()

    Display list of nuclides to console.
- void DisplayStableList ()

    Display list of stable nuclides to the console.
- void DisplayInfo ()

    Display final nuclides and their parents and branches.
- void DisplayStableInfo ()

    Display stable nuclides and their direct parents and branches.
- void DisplayMap ()

    Display the coefficient map to the console.
- void DisplayEigenMap ()

    Display the eigenvector map to the console.
- void loadNuclides (yaml_cpp_class &data)

    Function to load the nuclide library into the pointer.
- void unloadNuclides ()

    Delete the pointer to nuclide library to free space.
- int registerInitialNuclide (std::string isotope_name)

    Register an initial nuclide by name (e.g., H-2)
- int registerInitialNuclide (std::string symb, int iso)

    Register an initial nuclide by symbol (e.g., H) and isotope number (e.g., 2)
- int registerInitialNuclide (int atom_num, int iso_num)

    Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)
- int registerInitialNuclide (std::string isotope_name, double ic)

    NOTE: The below functions will register isotopes with their initial conditions as well.
- int registerInitialNuclide (std::string symb, int iso, double ic)

    Register an initial nuclide by symbol (e.g., H) and iso number (e.g., 2)
- int registerInitialNuclide (int atom_num, int iso_num, double ic)

    Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)
- void setWarnings (bool opt)

    Set the warnings boolean value.
- void setConsoleOut (bool opt)

    Set the ConsoleOut boolean value.
- void setThreshold (double val)

    Set the threshold value for half-life (in sec)
- void createChains ()

    Function to create unique list of final nuclides from decay chains of initial.
- void formEigenvectors ()

    Function to produce eigenvectors from coefficient matrix.
- int verifyEigenSoln ()

    Function will verify that the eigenvectors and eigenvalues are correct.
- void calculateFractionation (double t)

    Function to calculate the isotope fractionation given a time t in seconds.
- void calculateIonizationRate (std::vector< Atom > &atoms, std::vector< double > &mass_fracs, double density, double potential)

    Calculate the ionization rate given a list of atoms and their mass fractions in a particular media.

- void print_results (FILE ∗file, time_units units, double end_time, int points)

    Function to print results to file based on end_time and number of points.
- int read_conditions (yaml_cpp_class &yaml)

    Read the Runtime conditions for the simulation case.
- int read_isotopes (yaml_cpp_class &yaml)

    Read the Isotope conditions for the simulation case.
- int run_simulation (std::string file_name)

    Runs the simulation set up by the input files.
- double returnUnstableFractionation (int i, double t)

    Return the fractionation of the ith unstable nuclide.
- double returnStableFractionation (int i, double t)

    Return the fractionation of the ith stable nuclide.
- double returnFractionation (std::string iso_name, double t)

    Return the fractionation of the given nuclide.
- int getNumberNuclides ()

    Return the number of nuclides in the decay chain.
- int getNumberStableNuclides ()

    Return the number of stable nuclides.
- double getIonizationRate ()

    Return the stored ionization rate (in ion-pairs/second)
- int getIsotopeIndex (std::string iso_name)

    Return the unstable isotope index that corresponds to the given name.
- int getStableIsotopeIndex (std::string iso_name)

    Return the stable isotope index that corresponds to the given name.
- std::vector< int > & getParentList (int i)

    Return the vector list of parents for the ith isotope in the nuclide list.
- std::vector< int > & getStableParentList (int i)

    Return the vector list of parents for the ith stable isotope.
- std::vector< int > & getBranchList (int i, int j)

    Return the vector list of branch fractions for the jth parent of the ith nuclide.
- std::vector< int > & getStableBranchList (int i, int j)

    Return the list of branch fractions for the jth parent of the ith stable nuclide.
- Isotope & getIsotope (int i)

    Return the ith isotope in the nuclide list.
- Isotope & getStableIsotope (int i)

    Return the ith stable isotope.
- Isotope & getIsotope (std::string iso_name)

    Return the isotope (Stable or Unstable) that corresponds to the given name.

Protected Member Functions

- void roughInsertSort (Isotope iso)

    Insert an isotope to the initial nuclide list and sort according to isotope number.
- void finalSort ()

    Sort the list of nuclides after creating the chains.
- std::vector< Isotope > sameIsoNumSort (std::vector< Isotope > &list)

    Return a sorted list of isotopes given reference to another list.
- void fillOutBranchData ()

    Function to fill out all branching data and parent indices.
- void fillOutCoefMap ()

    Function to fill out all coefficients in the map.

Protected Attributes

- yaml_cpp_class ∗ nuclides

    Pointer to a yaml object storing the digital library of all nuclides.
- int time_steps

    Integer option to hold number of time steps to simulate.
- double end_time

    Time at which to end decay simulations.
- time_units t_units

    Units of time for which the end time is given.
- bool VerifyEigen

    Boolean option to check eigenvector solution.
- bool PrintChain

    Boolean option to print decay chain data to output file.
- bool PrintResults

    Boolean option to print simulation results to output file.
- bool PrintSparsity

    Boolean option to print sparsity pattern to output file.
- bool Warnings

    Boolean is True if you want to print warnings to console.
- double avg_eig_error

    Stores the average error in eigen solution.
- double hl_threshold

    Half-life value (in seconds) at which 99% of isotope has been converted.
- bool ConsoleOut

    Boolean is True if you want to print console messages.
- double ionization_rate

    Total number of ion-pairs produced per second from this decay chain.

Private Attributes

- std::vector< Isotope > nuc_list

    List of (ith) nuclides that make up the decay chain.
- std::unordered_map< std::string, int > nuc_map

    Map of unstable nuclides by isotope name (maps to index in nuc_list)
- std::vector< Isotope > stable_list

    List of (ith) stable nuclides that terminate decay chains.
- std::unordered_map< std::string, int > stable_map

    Map of stable nuclides by isotope name (maps to index in stable_list)
- std::vector< std::vector< int > > parents

    List of the indices that each isotope has from the nuc_list.
- std::vector< std::vector< std::vector< int > > > branches

    List of the indices for all branches of a parent to an isotope in nuc_list.
- std::vector< std::map< int, double > > CoefMap

    Coefficient Map for matrix representing the ODE system.
- std::vector< std::vector< int > > stable_parents

    List of the indices that each stable isotope has from the stable_list.
- std::vector< std::vector< std::vector< int > > > stable_branches

    List of the indices for all branches of a parent to a stable isotope in stable_list.
- std::vector< std::map< int, double > > stable_CoefMap

    Coefficient Map for matrix representing the ODE system for stable isotopes.
- std::vector< std::map< int, double > > EigenMap

    Map for all eigenvalues (row x column)

5.13.1 Detailed Description

DecayChain object to hold and store a set of unique isotopes in a branched decay chain.

C++ style object that will contain a list of unique nuclides that can be used to numerically solve a decay chain system. User will provide a list of initial nuclides present and this object will then use that information to build the list of all possible nuclides that can be formed. One of the key features of this object is that the nuclide list will be unique (i.e., no duplicate nuclides) so that we can iterate through that list to apply the branch fractions and decay constants to solve the fractionation as a function of time.

First, create chains of isotope from the initial nuclides given. Then, reduce chains to remove redundant isotopes and order the isotopes in final_nuc list to ensure that only the highest mass number isotopes are listed first. Lastly, fill out the parents and branches lists to allow for direct access to all necessary decay information to speed up computation and evaluation of the chains.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 DecayChain()

```
DecayChain::DecayChain ( )
```

Default constructor.

5.13.2.2 ∼DecayChain()

```
DecayChain::∼DecayChain ( )
```

Default destructor.

5.13.3 Member Function Documentation

5.13.3.1 DisplayList()

```
void DecayChain::DisplayList ( )
```

Display list of nuclides to console.

5.13.3.2 DisplayStableList()

```
void DecayChain::DisplayStableList ( )
```

Display list of stable nuclides to the console.

5.13.3.3 DisplayInfo()

```
void DecayChain::DisplayInfo ( )
```

Display final nuclides and their parents and branches.

5.13.3.4 DisplayStableInfo()

```
void DecayChain::DisplayStableInfo ( )
```

Display stable nuclides and their direct parents and branches.

5.13.3.5 DisplayMap()

```
void DecayChain::DisplayMap ( )
```

Display the coefficient map to the console.

5.13.3.6 DisplayEigenMap()

```
void DecayChain::DisplayEigenMap ( )
```

Display the eigenvector map to the console.

5.13.3.7 loadNuclides()

```
void DecayChain::loadNuclides (
             yaml_cpp_class & data )
```

Function to load the nuclide library into the pointer.

5.13.3.8 unloadNuclides()

```
void DecayChain::unloadNuclides ( )
```

Delete the pointer to nuclide library to free space.

5.13.3.9 registerInitialNuclide() [1/6]

```
int DecayChain::registerInitialNuclide (
             std::string isotope_name )
```

Register an initial nuclide by name (e.g., H-2)

5.13.3.10 registerInitialNuclide() [2/6]

```
int DecayChain::registerInitialNuclide (
            std::string symb,
            int iso )
```

Register an initial nuclide by symbol (e.g., H) and isotope number (e.g., 2)

5.13.3.11 registerInitialNuclide() [3/6]

```
int DecayChain::registerInitialNuclide (
            int atom_num,
            int iso_num )
```

Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)

5.13.3.12 registerInitialNuclide() [4/6]

```
int DecayChain::registerInitialNuclide (
            std::string isotope_name,
            double ic )
```

NOTE: The below functions will register isotopes with their initial conditions as well.

Register an initial nuclide by name (e.g., H-2)

5.13.3.13 registerInitialNuclide() [5/6]

```
int DecayChain::registerInitialNuclide (
            std::string symb,
            int iso,
            double ic )
```

Register an initial nuclide by symbol (e.g., H) and iso number (e.g., 2)

5.13.3.14 registerInitialNuclide() [6/6]

```
int DecayChain::registerInitialNuclide (
            int atom_num,
            int iso_num,
            double ic )
```

Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)

5.13.3.15 setWarnings()

```
void DecayChain::setWarnings (
             bool opt )
```

Set the warnings boolean value.

5.13.3.16 setConsoleOut()

```
void DecayChain::setConsoleOut (
             bool opt )
```

Set the ConsoleOut boolean value.

5.13.3.17 setThreshold()

```
void DecayChain::setThreshold (
             double val )
```

Set the threshold value for half-life (in sec)

5.13.3.18 createChains()

```
void DecayChain::createChains ( )
```

Function to create unique list of final nuclides from decay chains of initial.

5.13.3.19 formEigenvectors()

```
void DecayChain::formEigenvectors ( )
```

Function to produce eigenvectors from coefficient matrix.

5.13.3.20 verifyEigenSoln()

```
int DecayChain::verifyEigenSoln ( )
```

Function will verify that the eigenvectors and eigenvalues are correct.

5.13.3.21 calculateFractionation()

```
void DecayChain::calculateFractionation (
             double t )
```

Function to calculate the isotope fractionation given a time t in seconds.

This function must be called after createChains() and after formEigenvectors(). It will use the eigenvector solution to estimate the isotope concentrations for each isotope in the chain at the given time t. Those concentrations are based on values given for the initial concentrations of each isotope and are stored in each isotope object as the current concentration value.

Use an analytical solution based on linear combinations of eigenvectors.

5.13.3.22 calculateIonizationRate()

```
void DecayChain::calculateIonizationRate (
             std::vector< Atom > & atoms,
             std::vector< double > & mass_fracs,
             double density,
             double potential )
```

Calculate the ionization rate given a list of atoms and their mass fractions in a particular media.

This function uses the Linear Energy Transfer function to calculate the average ionization potential of this nuclide decaying in a media made up of the given set of atoms and their respective mass fractions. The result is stored in the ionization_coeff parameter and can be accessed via the IonizationCoeff() return function. As the density and/or mass fraction of the media changes, this function needs to be re-called to update that ionization rate.

Parameters

atoms	reference to a list of atoms in the media
mass_fracs	reference to a list of mass fractions for the list of atoms
density	density of the media in g/cm^3
potential	ionization potential of the media in eV

5.13.3.23 print_results()

```
void DecayChain::print_results (
             FILE * file,
             time_units units,
             double end_time,
             int points )
```

Function to print results to file based on end_time and number of points.

This function will open an output file named IBIS_Results.txt and print the simulation results to that file starting from time = 0 s to end_time in (s). The integer 'points' is used to determine how many points at which the calculation of fractionation will take place. The output file will be opened and closed within this function so the user is not responsible for keeping track of the file. All output is printed to the 'output/' folder from the working directory.

5.13.3.24 read_conditions()

```
int DecayChain::read_conditions (
            yaml_cpp_class & yaml )
```

Read the Runtime conditions for the simulation case.

5.13.3.25 read_isotopes()

```
int DecayChain::read_isotopes (
            yaml_cpp_class & yaml )
```

Read the Isotope conditions for the simulation case.

5.13.3.26 run_simulation()

```
int DecayChain::run_simulation (
            std::string file_name )
```

Runs the simulation set up by the input files.

5.13.3.27 returnUnstableFractionation()

```
double DecayChain::returnUnstableFractionation (
            int i,
            double t )
```

Return the fractionation of the ith unstable nuclide.

5.13.3.28 returnStableFractionation()

```
double DecayChain::returnStableFractionation (
            int i,
            double t )
```

Return the fractionation of the ith stable nuclide.

5.13.3.29 returnFractionation()

```
double DecayChain::returnFractionation (
            std::string iso_name,
            double t )
```

Return the fractionation of the given nuclide.

5.13.3.30 getNumberNuclides()

```
int DecayChain::getNumberNuclides ( )
```

Return the number of nuclides in the decay chain.

5.13.3.31 getNumberStableNuclides()

```
int DecayChain::getNumberStableNuclides ( )
```

Return the number of stable nuclides.

5.13.3.32 getIonizationRate()

```
double DecayChain::getIonizationRate ( )
```

Return the stored ionization rate (in ion-pairs/second)

5.13.3.33 getIsotopeIndex()

```
int DecayChain::getIsotopeIndex (
             std::string iso_name )
```

Return the unstable isotope index that corresponds to the given name.

5.13.3.34 getStableIsotopeIndex()

```
int DecayChain::getStableIsotopeIndex (
             std::string iso_name )
```

Return the stable isotope index that corresponds to the given name.

5.13.3.35 getParentList()

```
std::vector<int>& DecayChain::getParentList (
             int i )
```

Return the vector list of parents for the ith isotope in the nuclide list.

5.13.3.36 getStableParentList()

```
std::vector<int>& DecayChain::getStableParentList (
            int i )
```

Return the vector list of parents for the ith stable isotope.

5.13.3.37 getBranchList()

```
std::vector<int>& DecayChain::getBranchList (
            int i,
            int j )
```

Return the vector list of branch fractions for the jth parent of the ith nuclide.

5.13.3.38 getStableBranchList()

```
std::vector<int>& DecayChain::getStableBranchList (
            int i,
            int j )
```

Return the list of branch fractions for the jth parent of the ith stable nuclide.

5.13.3.39 getIsotope() [1/2]

```
Isotope& DecayChain::getIsotope (
            int i )
```

Return the ith isotope in the nuclide list.

5.13.3.40 getStableIsotope()

```
Isotope& DecayChain::getStableIsotope (
            int i )
```

Return the ith stable isotope.

5.13.3.41 getIsotope() [2/2]

```
Isotope& DecayChain::getIsotope (
            std::string iso_name )
```

Return the isotope (Stable or Unstable) that corresponds to the given name.

5.13.3.42 roughInsertSort()

```
void DecayChain::roughInsertSort (
            Isotope iso ) [protected]
```

Insert an isotope to the initial nuclide list and sort according to isotope number.

5.13.3.43 finalSort()

```
void DecayChain::finalSort ( ) [protected]
```

Sort the list of nuclides after creating the chains.

5.13.3.44 sameIsoNumSort()

```
std::vector<Isotope> DecayChain::sameIsoNumSort (
            std::vector< Isotope > & list ) [protected]
```

Return a sorted list of isotopes given reference to another list.

All isotopes in the list given will have the same isotope number, so we are sorting the list based on partents and daughters.

5.13.3.45 fillOutBranchData()

```
void DecayChain::fillOutBranchData ( ) [protected]
```

Function to fill out all branching data and parent indices.

5.13.3.46 fillOutCoefMap()

```
void DecayChain::fillOutCoefMap ( ) [protected]
```

Function to fill out all coefficients in the map.

5.13.4 Member Data Documentation

5.13.4.1 nuclides

```
yaml_cpp_class* DecayChain::nuclides [protected]
```

Pointer to a yaml object storing the digital library of all nuclides.

5.13.4.2 time_steps

```
int DecayChain::time_steps [protected]
```

Integer option to hold number of time steps to simulate.

5.13.4.3 end_time

```
double DecayChain::end_time [protected]
```

Time at which to end decay simulations.

5.13.4.4 t_units

```
time_units DecayChain::t_units [protected]
```

Units of time for which the end time is given.

5.13.4.5 VerifyEigen

```
bool DecayChain::VerifyEigen [protected]
```

Boolean option to check eigenvector solution.

5.13.4.6 PrintChain

```
bool DecayChain::PrintChain [protected]
```

Boolean option to print decay chain data to output file.

5.13.4.7 PrintResults

```
bool DecayChain::PrintResults [protected]
```

Boolean option to print simulation results to output file.

5.13.4.8 PrintSparsity

```
bool DecayChain::PrintSparsity [protected]
```

Boolean option to print sparsity pattern to output file.

5.13.4.9 Warnings

```
bool DecayChain::Warnings [protected]
```

Boolean is True if you want to print warnings to console.

5.13.4.10 avg_eig_error

```
double DecayChain::avg_eig_error [protected]
```

Stores the average error in eigen solution.

5.13.4.11 hl_threshold

```
double DecayChain::hl_threshold [protected]
```

Half-life value (in seconds) at which 99% of isotope has been converted.

5.13.4.12 ConsoleOut

```
bool DecayChain::ConsoleOut [protected]
```

Boolean is True if you want to print console messages.

5.13.4.13 ionization_rate

```
double DecayChain::ionization_rate [protected]
```

Total number of ion-pairs produced per second from this decay chain.

5.13.4.14 nuc_list

```
std::vector<Isotope> DecayChain::nuc_list [private]
```

List of (ith) nuclides that make up the decay chain.

5.13.4.15 nuc_map

```
std::unordered_map< std::string, int> DecayChain::nuc_map [private]
```

Map of unstable nuclides by isotope name (maps to index in nuc_list)

5.13.4.16 stable_list

```
std::vector<Isotope> DecayChain::stable_list [private]
```

List of (ith) stable nuclides that terminate decay chains.

5.13.4.17 stable_map

```
std::unordered_map< std::string, int> DecayChain::stable_map [private]
```

Map of stable nuclides by isotope name (maps to index in stable_list)

5.13.4.18 parents

```
std::vector< std::vector<int> > DecayChain::parents [private]
```

List of the indices that each isotope has from the nuc_list.

List of indices for all parents of an isotope in the list of all isotopes.

parents[i] = list of parent indices for the ith isotope parents[i][j] = jth parent index of the ith isotope

5.13.4.19 branches

```
std::vector< std::vector< std::vector<int> > > DecayChain::branches [private]
```

List of the indices for all branches of a parent to an isotope in nuc_list.

List of indices for direct access to necessary information to develop the chain scheme.

branches[i] = list of a list of all branch indices (by parent) for ith isotope branches[i][j] = list of all branches that contribute to formation of ith isotope by jth parent branches[i][j][k] = kth branch index for the jth parent that forms the ith isotope

5.13.4.20 CoefMap

```
std::vector< std::map<int, double> > DecayChain::CoefMap [private]
```

Coefficient Map for matrix representing the ODE system.

5.13.4.21 stable_parents

```
std::vector< std::vector<int> > DecayChain::stable_parents [private]
```

List of the indices that each stable isotope has from the stable_list.

List of indices for all parents of a stable isotope in the list of all isotopes.

stable_parents[i] = list of parent indices for the ith isotope stable_parents[i][j] = jth parent index of the ith isotope

5.13.4.22 stable_branches

```
std::vector< std::vector< std::vector<int> > > DecayChain::stable_branches  [private]
```

List of the indices for all branches of a parent to a stable isotope in stable_list.

List of indices for direct access to necessary information to develop the chain scheme.

stable_branches[i] = list of a list of all branch indices (by parent) for ith isotope stable_branches[i][j] = list of all branches that contribute to formation of ith isotope by jth parent stable_branches[i][j][k] = kth branch index for the jth parent that forms the ith isotope

5.13.4.23 stable_CoefMap

```
std::vector< std::map<int, double> > DecayChain::stable_CoefMap  [private]
```

Coefficient Map for matrix representing the ODE system for stable isotopes.

5.13.4.24 EigenMap

```
std::vector< std::map<int, double> > DecayChain::EigenMap  [private]
```

Map for all eigenvalues (row x column)

The documentation for this class was generated from the following file:

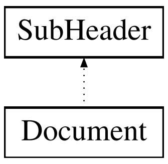
- ibis.h

5.14 Document Class Reference

Object for the various documents in the yaml file.

```
#include <yaml_wrapper.h>
```

Inheritance diagram for Document:



Public Member Functions

- Document ()

  Default constructor.
- ∼Document ()

  Default destructor.
- Document (const Document &doc)

  Copy constructor.
- Document (std::string name)

  Constructor by name.
- Document (const KeyValueMap &map)

  Constructor by existing map.
- Document (std::string name, const KeyValueMap &map)

  Constructor by name and map.
- Document (std::string key, const Header &head)

  Constructor by single header.
- Document & operator= (const Document &doc)

  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)

  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const

  Return the ValueType at the given key.
- Header & operator() (const std::string key)

  Return the Header reference at the given key.
- Header operator() (const std::string key) const

  Return the Header at the given key.
- std::map< std::string, Header > & getHeadMap ()

  Return the reference to the Header Map.
- KeyValueMap & getDataMap ()

  Return the reference to the KeyValueMap.
- Header & getHeader (std::string key)

  Return reference to the Header in map at the key.
- std::map< std::string, Header >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, Header >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, Header >::const_iterator begin () const

  Returns a const iterator pointing to the begining of the list.
- std::map< std::string, Header >::iterator begin ()

  Returns an iterator pointing to the begining of the list.
- void clear ()

  Clear out info in the Document.
- void resetKeys ()

  Set all keys in the map to match names of the headers.
- void changeKey (std::string oldKey, std::string newKey)

  Change a given oldKey in the header map to the newKey given.
- void revalidateAllKeys ()

  Resets and validates keys in header and subheader maps.
- void addPair (std::string key, std::string val)

  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int t)

Adds a pair object and asserts a type.
- void setName (std::string name)

  Set the name of the Document.
- void setAlias (std::string alias)

  Set the alias of the Document.
- void setNameAliasPair (std::string n, std::string a, int s)

  Set the name, alias, and state of the document.
- void setState (int state)

  Set the state of the Document.
- void DisplayContents ()

  Display the contents of the Document.
- void addHeadKey (std::string key)

  Add a key to the Header without a header object.
- void copyAnchor2Alias (std::string alias, Header &ref)

  Find the anchor in the map, and copy to the Header reference given.
- int size ()

  Return the size of the header map.
- std::string getName ()

  Return the name of the document.
- std::string getAlias ()

  Return the alias of the document.
- int getState ()

  Return the state of the document.
- bool isAlias ()

  Returns true if the document is an alias.
- bool isAnchor ()

  Returns true if the document is an anchor.
- Header & getAnchoredHeader (std::string alias)

  Returns reference to the anchored header, if any.
- Header & getHeadFromSubAlias (std::string alias)

  Returns reference to the Header that contains a Sub with the given alias.

Private Attributes

- std::map< std::string, Header > Head_Map

  Map of headers contained within the document.

Additional Inherited Members

5.14.1 Detailed Description

Object for the various documents in the yaml file.

C++ Object for the documents in a yaml input file as denoted by a Key: followed by — (three dashes) and ending with a ... (three dots). A single yaml file can have multiple document structures and each document structure can have multiple headers (which have sub-headers and key-values) and key-value-pairs. This is the larges single object in the yaml file itself.

Just like Header, this object also inherits from SubHeader and therefore has access to its protected members. You can use access to those members to establish the KeyValuePairs in the Document, name the Document, and give the Document an alias or anchor value.

5.14.2 Constructor & Destructor Documentation

5.14.2.1 Document() [1/6]

```
Document::Document ( )
```

Default constructor.

5.14.2.2 ∼Document()

```
Document::∼Document ( )
```

Default destructor.

5.14.2.3 Document() [2/6]

```
Document::Document (
             const Document & doc )
```

Copy constructor.

5.14.2.4 Document() [3/6]

```
Document::Document (
             std::string name )
```

Constructor by name.

5.14.2.5 Document() [4/6]

```
Document::Document (
             const KeyValueMap & map )
```

Constructor by existing map.

5.14.2.6 Document() [5/6]

```
Document::Document (
             std::string name,
             const KeyValueMap & map )
```

Constructor by name and map.

5.14.2.7 Document() [6/6]

```
Document::Document (
             std::string key,
             const Header & head )
```

Constructor by single header.

5.14.3 Member Function Documentation

5.14.3.1 operator=()

```
Document& Document::operator= (
             const Document & doc )
```

Equals overload.

5.14.3.2 operator[]() [1/2]

```
ValueTypePair& Document::operator[] (
             const std::string key )
```

Return the ValueType reference at the given key.

5.14.3.3 operator[]() [2/2]

```
ValueTypePair Document::operator[] (
             const std::string key ) const
```

Return the ValueType at the given key.

5.14.3.4 operator()() [1/2]

```
Header& Document::operator() (
             const std::string key )
```

Return the Header reference at the given key.

5.14.3.5 operator()() [2/2]

```
Header Document::operator() (
             const std::string key ) const
```

Return the Header at the given key.

5.14.3.6 getHeadMap()

```
std::map<std::string, Header>& Document::getHeadMap ( )
```

Return the reference to the Header Map.

5.14.3.7 getDataMap()

```
KeyValueMap& Document::getDataMap ( )
```

Return the reference to the KeyValueMap.

5.14.3.8 getHeader()

```
Header& Document::getHeader (
             std::string key )
```

Return reference to the Header in map at the key.

5.14.3.9 end() [1/2]

```
std::map<std::string, Header>::const_iterator Document::end ( ) const
```

Returns a const iterator pointing to the end of the list.

5.14.3.10 end() [2/2]

```
std::map<std::string, Header>::iterator Document::end ( )
```

Returns an iterator pointing to the end of the list.

5.14.3.11 begin() [1/2]

```
std::map<std::string, Header>::const_iterator Document::begin ( ) const
```

Returns a const iterator pointing to the begining of the list.

5.14.3.12 begin() [2/2]

```
std::map<std::string, Header>::iterator Document::begin ( )
```

Returns an iterator pointing to the begining of the list.

5.14.3.13 clear()

```
void Document::clear ( )
```

Clear out info in the Document.

5.14.3.14 resetKeys()

```
void Document::resetKeys ( )
```

Set all keys in the map to match names of the headers.

5.14.3.15 changeKey()

```
void Document::changeKey (
             std::string oldKey,
             std::string newKey )
```

Change a given oldKey in the header map to the newKey given.

5.14.3.16 revalidateAllKeys()

```
void Document::revalidateAllKeys ( )
```

Resets and validates keys in header and subheader maps.

5.14.3.17 addPair() [1/2]

```
void Document::addPair (
             std::string key,
             std::string val )
```

Adds a pair object to the map (with only strings)

5.14.3.18 addPair() [2/2]

```
void Document::addPair (
             std::string key,
             std::string val,
             int t )
```

Adds a pair object and asserts a type.

5.14.3.19 setName()

```
void Document::setName (
             std::string name )
```

Set the name of the Document.

5.14.3.20 setAlias()

```
void Document::setAlias (
             std::string alias )
```

Set the alias of the Document.

5.14.3.21 setNameAliasPair()

```
void Document::setNameAliasPair (
             std::string n,
             std::string a,
             int s )
```

Set the name, alias, and state of the document.

5.14.3.22 setState()

```
void Document::setState (
             int state )
```

Set the state of the Document.

5.14.3.23 DisplayContents()

```
void Document::DisplayContents ( )
```

Display the contents of the Document.

5.14.3.24 addHeadKey()

```
void Document::addHeadKey (
             std::string key )
```

Add a key to the Header without a header object.

5.14.3.25 copyAnchor2Alias()

```
void Document::copyAnchor2Alias (
             std::string alias,
             Header & ref )
```

Find the anchor in the map, and copy to the Header reference given.

5.14.3.26 size()

```
int Document::size ( )
```

Return the size of the header map.

5.14.3.27 getName()

```
std::string Document::getName ( )
```

Return the name of the document.

5.14.3.28 getAlias()

```
std::string Document::getAlias ( )
```

Return the alias of the document.

5.14.3.29 getState()

```
int Document::getState ( )
```

Return the state of the document.

5.14.3.30 isAlias()

```
bool Document::isAlias ( )
```

Returns true if the document is an alias.

5.14.3.31 isAnchor()

```
bool Document::isAnchor ( )
```

Returns true if the document is an anchor.

5.14.3.32 getAnchoredHeader()

```
Header& Document::getAnchoredHeader (
            std::string alias )
```

Returns reference to the anchored header, if any.

5.14.3.33 getHeadFromSubAlias()

```
Header& Document::getHeadFromSubAlias (
            std::string alias )
```

Returns reference to the Header that contains a Sub with the given alias.

5.14.4 Member Data Documentation

5.14.4.1 Head_Map

```
std::map<std::string, Header> Document::Head_Map  [private]
```

Map of headers contained within the document.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.15 DOGFISH_DATA Struct Reference

Primary data structure for running the DOGFISH application.

```
#include <dogfish.h>
```

Public Attributes

- unsigned long int total_steps = 0

  Total number of solver steps taken.
- double time_old = 0.0

  Old value of time (hrs)
- double time = 0.0

  Current value of time (hrs)
- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- bool DirichletBC = false

  False = uses film mass transfer for BC, True = Dirichlet BC.
- bool NonLinear = false

  False = Solve directly, True = Solve iteratively.
- double t_counter = 0.0

  Counter for the time output.
- double t_print

  Print output at every t_print time (hrs)
- int NumComp

  Number of species to track.
- double end_time

  Units: hours.
- double total_sorption_old

  Per mass or volume of single fiber.
- double total_sorption

  Per mass or volume of single fiber.
- double fiber_length

  Units: um.
- double fiber_diameter

  Units: um.
- double fiber_specific_area

  Units: m^2/kg.
- FILE ∗ OutputFile

  Output file pointer to the output file for postprocesses and results.
- double(∗ eval_R)(int i, int l, const void ∗data)

  Function pointer to evaluate retardation coefficient.
- double(∗ eval_DI)(int i, int l, const void ∗data)

  Function pointer to evaluate intraparticle diffusivity.
- double(∗ eval_kf)(int i, const void ∗data)

  Function pointer to evaluate film mass transfer coefficient.
- double(∗ eval_qs)(int i, const void ∗data)

  Function pointer to evaluate fiber surface concentration.
- const void ∗ user_data

  Data structure for users info to calculate all parameters.
- std::vector< FINCH_DATA > finch_dat

  Data structure for FINCH_DATA objects.
- std::vector< DOGFISH_PARAM > param_dat

  Data structure for DOGFISH_PARAM objects.

5.15.1 Detailed Description

Primary data structure for running the DOGFISH application.

C-style object to hold information for the adsorption simulations. Contains function pointers and other data structures. This information is passed around to other functions used to simulate the fiber diffusion physics.

5.15.2 Member Data Documentation

5.15.2.1 total_steps

```
unsigned long int DOGFISH_DATA::total_steps = 0
```

Total number of solver steps taken.

5.15.2.2 time_old

```
double DOGFISH_DATA::time_old = 0.0
```

Old value of time (hrs)

5.15.2.3 time

```
double DOGFISH_DATA::time = 0.0
```

Current value of time (hrs)

5.15.2.4 Print2File

```
bool DOGFISH_DATA::Print2File = true
```

True = results to .txt; False = no printing.

5.15.2.5 Print2Console

```
bool DOGFISH_DATA::Print2Console = true
```

True = results to console; False = no printing.

5.15.2.6 DirichletBC

```
bool DOGFISH_DATA::DirichletBC = false
```

False = uses film mass transfer for BC, True = Dirichlet BC.

5.15.2.7 NonLinear

```
bool DOGFISH_DATA::NonLinear = false
```

False = Solve directly, True = Solve iteratively.

5.15.2.8 t_counter

```
double DOGFISH_DATA::t_counter = 0.0
```

Counter for the time output.

5.15.2.9 t_print

```
double DOGFISH_DATA::t_print
```

Print output at every t_print time (hrs)

5.15.2.10 NumComp

```
int DOGFISH_DATA::NumComp
```

Number of species to track.

5.15.2.11 end_time

```
double DOGFISH_DATA::end_time
```

Units: hours.

5.15.2.12 total_sorption_old

```
double DOGFISH_DATA::total_sorption_old
```

Per mass or volume of single fiber.

5.15.2.13 total_sorption

```
double DOGFISH_DATA::total_sorption
```

Per mass or volume of single fiber.

5.15.2.14 fiber_length

```
double DOGFISH_DATA::fiber_length
```

Units: um.

5.15.2.15 fiber_diameter

```
double DOGFISH_DATA::fiber_diameter
```

Units: um.

5.15.2.16 fiber_specific_area

```
double DOGFISH_DATA::fiber_specific_area
```

Units: m^2/kg.

5.15.2.17 OutputFile

```
FILE* DOGFISH_DATA::OutputFile
```

Output file pointer to the output file for postprocesses and results.

5.15.2.18 eval_R

```
double(* DOGFISH_DATA::eval_R) (int i, int l, const void *data)
```

Function pointer to evaluate retardation coefficient.

5.15.2.19 eval_DI

```
double(* DOGFISH_DATA::eval_DI) (int i, int l, const void *data)
```

Function pointer to evaluate intraparticle diffusivity.

5.15.2.20 eval_kf

```
double(* DOGFISH_DATA::eval_kf) (int i, const void *data)
```

Function pointer to evaluate film mass transfer coefficient.

5.15.2.21 eval_qs

```
double(* DOGFISH_DATA::eval_qs) (int i, const void *data)
```

Function pointer to evaluate fiber surface concentration.

5.15.2.22 user_data

```
const void* DOGFISH_DATA::user_data
```

Data structure for users info to calculate all parameters.

5.15.2.23 finch_dat

```
std::vector<FINCH_DATA> DOGFISH_DATA::finch_dat
```

Data structure for FINCH_DATA objects.

5.15.2.24 param_dat

```
std::vector<DOGFISH_PARAM> DOGFISH_DATA::param_dat
```

Data structure for DOGFISH_PARAM objects.

The documentation for this struct was generated from the following file:

- dogfish.h

5.16 DOGFISH_PARAM Struct Reference

Data structure for species-specific parameters.

```
#include <dogfish.h>
```

Public Attributes

- double intraparticle_diffusion

  Units: um^2/hr.
- double film_transfer_coeff

  Units: um/hr.
- double surface_concentration

  Units: mol/kg.
- double initial_sorption

  Units: mol/kg.
- double sorbed_molefraction

  Molefraction of sorbed species.
- Molecule species

  Adsorbed species Molecule Object.

5.16.1 Detailed Description

Data structure for species-specific parameters.

C-style object to hold information on all adsorbing species. Parameters are given descriptive names to indicate what each is for.

5.16.2 Member Data Documentation

5.16.2.1 intraparticle_diffusion

```
double DOGFISH_PARAM::intraparticle_diffusion
```

Units: um^2/hr.

5.16.2.2 film_transfer_coeff

```
double DOGFISH_PARAM::film_transfer_coeff
```

Units: um/hr.

5.16.2.3 surface_concentration

```
double DOGFISH_PARAM::surface_concentration
```

Units: mol/kg.

5.16.2.4 initial_sorption

```
double DOGFISH_PARAM::initial_sorption
```

Units: mol/kg.

5.16.2.5 sorbed_molefraction

```
double DOGFISH_PARAM::sorbed_molefraction
```

Molefraction of sorbed species.

5.16.2.6 species

```
Molecule DOGFISH_PARAM::species
```

Adsorbed species Molecule Object.

The documentation for this struct was generated from the following file:

- dogfish.h

5.17 Dove Class Reference

Dynamic ODE-solver with Various Established methods (DOVE) object.

```
#include <dove.h>
```

Public Member Functions

- Dove ()
  Default constructor.
- ∼Dove ()
  Default destructor.
- void set_numfunc (int i)
  Set the number of functions to solve and reserve necessary space.
- void set_timestep (double d)
  Set the value of the time step.
- void set_timestepmin_converged (double d)
  Set the value of the minimum time step after simulation converged.
- void set_timestepmin (double dmin)
  Set the value of the minimum time step.
- void set_timestepmax (double dmax)
  Set the value of the maximum time step.
- void set_endtime (double e)
  Set the value of the end time.
- void set_starttime (double s)

Set the value of the starting time.
- void set_integrationtype (integrate_subtype type)

  Set the type of integration scheme to use.
- void set_timestepper (timestep_type type)

  Set the time stepper scheme type.
- void set_preconditioner (precond_type type)

  Set the type of preconditioner to use.
- void set_outputfile (FILE ∗file)

  Set the output file for simulation results.
- void set_userdata (const void ∗data)

  Set the user defined data structure.
- void set_initialcondition (int i, double ic)

  Set the initial condition of variable i to value ic.
- void set_initialcondition (std::string name, double ic)

  Set the initial condition of variable name to value ic.
- void set_variableName (int i, std::string name)

  Set the name of variable i to the given string (both i and name should be unique)
- void set_variableSteadyState (int i)

  Set the ith variable to be steady-state (i.e., var_steady[i] = true)
- void set_variableSteadyState (std::string name)

  Set the named varibale to steady-state (i.e., var_steady[i] = true)
- void set_variableSteadyStateAll ()

  Set all variables to be steady-state.
- void set_output (bool choice)

  Set the value of DoveOutput (True if you want console messages)
- void set_fileoutput (bool choice)

  Set the value of DoveFileOuput (True if you want results printed to file)
- void set_headeroutput (bool choice)

  Set the value of DoveHeader (True if you want header printed to file)
- void set_tolerance (double tol)

  Set the value of residual/error tolerance desired.
- void set_t_out (double v)

  Set the value of the t_out parameter.
- void set_defaultNames ()

  Set all the variable names to default values (does not set var_names_hash values)
- void set_defaultCoeffs ()

  Set all coeff functions to the default.
- void set_defaultJacobis ()

  Set all Jacobians to the default (only does the diagonals!)
- void set_defaultStates ()

  Set all var_steady values to false for each variable (unsteady variables are default)
- void set_NonlinearAbsTol (double tol)

  Set the value of nonlinear absolute tolerance.
- void set_NonlinearRelTol (double tol)

  Set the value of nonlinear relative tolerance.
- void set_LinearAbsTol (double tol)

  Set the value of linear absolute tolerance.
- void set_LinearRelTol (double tol)

  Set the value of linear relative tolerance.
- void set_NonlinearOutput (bool choice)

  Sets the non-linear output information according to user choice.

- void set_LinearOutput (bool choice)

  Sets the linear output information according to user choice.
- void set_Preconditioning (bool choice)

  Sets the boolean to determine whether or not to include preconditioning.
- void set_LinearMethod (krylov_method choice)

  Sets the linear solver method to user choice.
- void set_LineSearchMethod (linesearch_type choice)

  Sets the line search method to the user choice.
- void set_MaxNonLinearIterations (int it)

  Set the maximum number of non-linear iterations.
- void set_MaxLinearIterations (int it)

  Set the maximum number of linear iterations (or number of restarts)
- void set_RestartLimit (int it)

  Sets the number of iterations before restarting.
- void set_RecursionLevel (int level)

  Sets the maximum level of recursion for the KMS method.
- void set_LinearStatus (bool choice)

  Sets the boolean to determine whether or not to treat as linear (true = Linear)
- void registerFunction (int i, double(∗func)(int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove))

  Register the ith user function.
- void registerFunction (std::string name, double(∗func)(int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove))

  Register the named user function.
- void registerCoeff (int i, double(∗coeff)(int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove))

  Register the ith time coeff function.
- void registerCoeff (std::string name, double(∗coeff)(int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove))

  Register the named time coeff function.
- void registerJacobi (int i, int j, double(∗jac)(int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove))

  Register the i-jth element of jacobian.
- void registerJacobi (std::string func_name, std::string var_name, double(∗jac)(int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove))

  Register the named element of jacobian.
- void print_header (bool addNewLine)

  Function to print out a header to output file.
- void print_newresult (bool addNewLine)

  Function to print out the new result of n+1 time level.
- void print_result (bool addNewLine)

  Function to print out the old result of n time level.
- void createJacobian ()

  Function to create and store a numerical jacobian.
- Matrix< double > & getNumJacobian ()

  Return reference to the numerical jacobian.
- Matrix< double > & getCurrentU ()

  Return reference to the n level solution.
- Matrix< double > & getOldU ()

  Return reference to the n-1 level solution.
- Matrix< double > & getNewU ()

Return reference to the n+1 level solution.

- int getVariableIndex (std::string name) const

  Return the index of the variable whose name matches the given string (checks hash table)
- std::string getVariableName (int i)

  Return the name of the variable based on the given index.
- double getMaxRate ()

  Returns the value of the maximum rate of change for all variables.
- double getCurrentU (int i, const Matrix< double > &u) const

  Return the value of the n level solution for variable i.
- double getOldU (int i, const Matrix< double > &u) const

  Return the value of the n-1 level solution for variable i.
- double getNewU (int i, const Matrix< double > &u) const

  Return the value of the n+1 level solution for variable i.
- double getCurrentU (std::string name, const Matrix< double > &u) const

  Return the value of the n level solution for variable of given name.
- double getOldU (std::string name, const Matrix< double > &u) const

  Return the value of the n-1 level solution for variable of given name.
- double getNewU (std::string name, const Matrix< double > &u) const

  Return the value of the n+1 level solution for variable of given name.
- double coupledTimeDerivative (int i, const Matrix< double > &u) const

  Return the value of the ith variable's time derivative.
- double coupledTimeDerivative (std::string name, const Matrix< double > &u) const

  Return the value of the named variable's time derivative.
- double coupledDerivativeTimeDerivative (int i, int j, const Matrix< double > &u) const

  Return the value of the ith variable's time derivative's jth derivative.
- const void ∗ getUserData ()

  Return pointer to user data.
- int getNumFunc () const

  Return the number of functions.
- double getTimeStep () const

  Return the current time step.
- double getTimeStepOld () const

  Return the old time step.
- double getEndTime () const

  Return value of end time.
- double getCurrentTime () const

  Return the value of current time.
- double getOldTime () const

  Return the value of the previous time.
- double getOlderTime () const

  Return the value of the older previous time.
- double getStartTime () const

  Return the value of the start time.
- double getMinTimeStep ()

  Return the value of the minimum time step.
- double getMaxTimeStep ()

  Return the value of the maximum time step.
- double getOutputTime ()

  Return the time that output is printed on.
- FILE ∗ getFile ()

  Return the pointer to the output file.

- bool hasConverged ()

    Returns state of convergence.
- double getNonlinearResidual ()

    Returns the current value of the non-linear residual.
- double getNonlinearRelativeRes ()

    Returns the current value of the non-linear relative residual.
- bool allSteadyState () const

    Returns a boolean to determine whether or not all variables are steady (true = all steady)
- bool isSteadyState (int i) const

    Returns true if the ith variable is steady-state.
- integrate_type getIntegrationType ()

    Returns the type of integration method in use.
- integrate_subtype getIntegrationMethod ()

    Returns the method of time integration.
- timestep_type getTimeStepper ()

    Returns the time stepper method.
- precond_type getPreconditioner ()

    Returns the preconditioner type.
- linesearch_type getLinesearchMethod ()

    Returns the method of line search.
- krylov_method getLinearMethod ()

    Returns the linear method used.
- bool isPreconditioned ()

    Returns true if using preconditioning.
- double getLinearToleranceABS ()

    Returns the value of linear tolerance (absolute)
- double getLinearToleranceREL ()

    Returns the value of linear tolerance (relative)
- double getNonlinearToleranceABS ()

    Returns the value of nonlinear tolerance (absolute)
- double getNonlinearToleranceREL ()

    Returns the value of nonlinear tolerance (relative)
- int getMaxNonlinearIterations ()

    Returns the maximum allowable nonlinear iterations.
- int getMaxLinearIterations ()

    Returns the maximum allowable linear iterations.
- bool isLinear ()

    Returns true if the system is being treated as linear.
- int getRestartLevel ()

    Returns the number of iterations allowed before restart for Krylov methods.
- int getRecursionLevel ()

    Returns the number of recursion levels allow for Krylov preconditioning.
- bool isValidName (std::string name)

    Returns true if the given name is a variable in the DOVE system.
- std::map< int, double(∗)(int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)> & getJacobiMap (int i)

    Function to return a reference to the Jacobian Matrix map at the ith row of the matrix.
- double ComputeTimeStep ()

    Returns a computed value for the next time step.
- double Eval_Func (int i, const Matrix< double > &u, double t)

    Evaluate user function i at given u matrix and time t.

- double Eval_Coeff (int i, const Matrix< double > &u, double t)

    Evaluate user time coefficient function i at given u matrix and time t.
- double Eval_Jacobi (int i, int j, const Matrix< double > &u, double t)

    Evaluate user jacobian function for (i,j) at given u matrix and time t.
- int solve_timestep ()

    Function to solve a single time step.
- void validate_precond ()

    Function to validate and set preconditioning pointer.
- void validate_linearsteps ()

    Function to check and validate the number of linear iterations.
- void validate_method ()

    Function to check and validate the time integration method.
- void update_states ()

    Function to update the stateful information.
- void update_timestep ()

    Function to update the timestep for the simulation.
- void reset_all ()

    Reset all the states.
- int solve_all ()

    Function to solve the system of equations and print results to file (returns 0 on success)
- int solve_FE ()

    Solver function for explicit-FE method.
- int solve_RK4 ()

    Solver function for explicit-RK4 method.
- int solve_RKF ()

    Solver function for explicit-RKF method.

Protected Attributes

- Matrix< std::string > var_names

    Matrix of variable names (access names by index in numerical order)
- std::unordered_map< std::string, int > var_names_hash

    Hash table of variable names and corresponding indices (access index by name)
- Matrix< int > var_steady

    Matrix of boolean args used to dictate which variables are considered steady-state (if any)
- Matrix< double > un

    Matrix for nth level solution vector.
- Matrix< double > unp1

    Matrix for n+1 level solution vector.
- Matrix< double > unm1

    Matrix for n-1 level solution vector.
- double dt

    Time step between n and n+1 time levels.
- double dt_old

    Time step between n and n-1 time levels.
- double time_end

    Time on which to end the ODE simulations.
- double time_start

    Time on which to start the ODE simulations.
- double time

Value of current time.
- double time_old

  Value of previous time.
- double time_older

  Value of older previous time.
- double dtmin

  Minimum allowable time step.
- double dtmin_con

  Minimum allowable time step if solution has converged.
- double dtmax

  Maximum allowable time step.
- double tolerance

  Residual tolerance desired (or level of accuracy desired)
- double t_count

  Counter designed to keep track of how often we print results.
- double t_out

  Direct Dove to only print results after specific time increments.
- integrate_type int_type

  Type of time integration to use.
- integrate_subtype int_sub

  Subtype of time integration scheme to use.
- timestep_type timestepper

  Type of time stepper to be used.
- precond_type preconditioner

  Type of preconditioner to use.
- linesearch_type line_type

  Type of linesearch method to use.
- FILE $*$ Output

  File to where simulation results will be place.
- int num_func

  Number of functions in the system of ODEs.
- bool Converged

  Boolean to hold information on whether or not last step converged.
- bool DoveOutput

  Boolean to determine whether or not to print Dove messages to console.
- bool DoveFileOutput

  Boolean to determine whether or not to print Dove ouput to the file.
- bool DoveHeader

  Boolean to determine whether or not to print the Dove header to file.
- bool Preconditioner

  Boolean to determine whether or not to use a preconditioner.
- bool Linear

  Boolean to determine whether or not to treat problem as linear.
- bool AllSteadyState

  Boolean to determine whether or not all variables are steady (true = all steady-state)
- int linmax

  Users requested maximum number of linear steps.
- int timesteps

  Running count of number of time steps taken.
- Matrix$<$ double($*$)(int i, const Matrix$<$ double $>$ &u, double t, const void $*$data, const Dove &dove)$>$ user↩_func

Matrix object for user defined rate functions.

- Matrix< double(∗)(int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)> user↩ _coeff

    Matrix object for user defined time coefficients (optional)
- std::vector< std::map< int, double(∗)(int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)> > user_jacobi

    A vector of Maps for user defined Jacobian elements (optional)
- const void ∗ user_data

    Pointer for user defined data structure.
- PJFNK_DATA newton_dat
- int(∗ residual)(const Matrix< double > &x, Matrix< double > &F, const void ∗res_data)

    Function pointer for the residual function of DOVE.
- int(∗ precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗precon_data)

    Function pointer for the preconditioning operation of DOVE.
- NUM_JAC_DATA jac_dat

    Data structure for making Numerical Jacobian Matrices.
- Matrix< double > Jacobian

    Matrix to hold numerical jacobian.

5.17.1 Detailed Description

Dynamic ODE-solver with Various Established methods (DOVE) object.

This class structure creates a C++ object that can be used to solve coupled systems of Ordinary Differential Equations. A user will interface with this object by creating functions that evaluate the right-hand side of an ODE based on the given variable set. Those functions will collectively create the system to solve numerically using either explicit or implicit methods. The choice of methods can be set by the user, or the object will default to the Backwards-Euler implicit method for stability.

User functions for the right-hand side are written as...

du_i/dt = user_function_i(const Matrix<double> &u, const void ∗data_struct)

In some cases, there is a need to include a time coefficient on the left-hand side of the rate expression. For those cases, the user may also provide a time coefficient function...

user_time_coeff_i(const Matrix<double> &u, const void ∗data_struct) ∗ du_i/dt = user_function_i(...)

For most implicit problems, the ODE system must be solved iteratively using a Newton-style method. In these cases, the user may also provide functions for Jacobian matrix elements...

user_jacobi_element_i_j(const Matrix<double> &u, const void ∗data_struct)

All of these above functions are to be put into Matrices inside of the Dove class object so that Dove will call those functions when it needs to be called. Data structures for all function calls are optional and are to be defined by the user to contain whatever parameter information is needed for their particular problem.

5.17.2 Constructor & Destructor Documentation

5.17.2.1 Dove()

```
Dove::Dove ( )
```

Default constructor.

5.17.2.2 ∼Dove()

```
Dove::∼Dove ( )
```

Default destructor.

5.17.3 Member Function Documentation

5.17.3.1 set_numfunc()

```
void Dove::set_numfunc (
             int i )
```

Set the number of functions to solve and reserve necessary space.

5.17.3.2 set_timestep()

```
void Dove::set_timestep (
             double d )
```

Set the value of the time step.

5.17.3.3 set_timestepmin_converged()

```
void Dove::set_timestepmin_converged (
             double d )
```

Set the value of the minimum time step after simulation converged.

5.17.3.4 set_timestepmin()

```
void Dove::set_timestepmin (
             double dmin )
```

Set the value of the minimum time step.

5.17.3.5 set_timestepmax()

```
void Dove::set_timestepmax (
             double dmax )
```

Set the value of the maximum time step.

5.17.3.6 set_endtime()

```
void Dove::set_endtime (
             double e )
```

Set the value of the end time.

5.17.3.7 set_starttime()

```
void Dove::set_starttime (
             double s )
```

Set the value of the starting time.

5.17.3.8 set_integrationtype()

```
void Dove::set_integrationtype (
             integrate_subtype type )
```

Set the type of integration scheme to use.

5.17.3.9 set_timestepper()

```
void Dove::set_timestepper (
             timestep_type type )
```

Set the time stepper scheme type.

5.17.3.10 set_preconditioner()

```
void Dove::set_preconditioner (
             precond_type type )
```

Set the type of preconditioner to use.

5.17.3.11 set_outputfile()

```
void Dove::set_outputfile (
             FILE * file )
```

Set the output file for simulation results.

5.17.3.12 set_userdata()

```
void Dove::set_userdata (
             const void * data )
```

Set the user defined data structure.

5.17.3.13 set_initialcondition() [1/2]

```
void Dove::set_initialcondition (
             int i,
             double ic )
```

Set the initial condition of variable i to value ic.

5.17.3.14 set_initialcondition() [2/2]

```
void Dove::set_initialcondition (
             std::string name,
             double ic )
```

Set the initial condition of variable name to value ic.

5.17.3.15 set_variableName()

```
void Dove::set_variableName (
             int i,
             std::string name )
```

Set the name of variable i to the given string (both i and name should be unique)

5.17.3.16 set_variableSteadyState() [1/2]

```
void Dove::set_variableSteadyState (
             int i )
```

Set the ith variable to be steady-state (i.e., var_steady[i] = true)

5.17.3.17 set_variableSteadyState() [2/2]

```
void Dove::set_variableSteadyState (
             std::string name )
```

Set the named varibale to steady-state (i.e., var_steady[i] = true)

5.17.3.18 set_variableSteadyStateAll()

```
void Dove::set_variableSteadyStateAll ( )
```

Set all variables to be steady-state.

5.17.3.19 set_output()

```
void Dove::set_output (
             bool choice )
```

Set the value of DoveOutput (True if you want console messages)

5.17.3.20 set_fileoutput()

```
void Dove::set_fileoutput (
             bool choice )
```

Set the value of DoveFileOuput (True if you want results printed to file)

5.17.3.21 set_headeroutput()

```
void Dove::set_headeroutput (
             bool choice )
```

Set the value of DoveHeader (True if you want header printed to file)

5.17.3.22 set_tolerance()

```
void Dove::set_tolerance (
             double tol )
```

Set the value of residual/error tolerance desired.

5.17.3.23 set_t_out()

```
void Dove::set_t_out (
            double v )
```

Set the value of the t_out parameter.

5.17.3.24 set_defaultNames()

```
void Dove::set_defaultNames ( )
```

Set all the variable names to default values (does not set var_names_hash values)

5.17.3.25 set_defaultCoeffs()

```
void Dove::set_defaultCoeffs ( )
```

Set all coeff functions to the default.

5.17.3.26 set_defaultJacobis()

```
void Dove::set_defaultJacobis ( )
```

Set all Jacobians to the default (only does the diagonals!)

5.17.3.27 set_defaultStates()

```
void Dove::set_defaultStates ( )
```

Set all var_steady values to false for each variable (unsteady variables are default)

5.17.3.28 set_NonlinearAbsTol()

```
void Dove::set_NonlinearAbsTol (
            double tol )
```

Set the value of nonlinear absolute tolerance.

5.17.3.29 set_NonlinearRelTol()

```
void Dove::set_NonlinearRelTol (
            double tol )
```

Set the value of nonlinear relative tolerance.

5.17.3.30 set_LinearAbsTol()

```
void Dove::set_LinearAbsTol (
            double tol )
```

Set the value of linear absolute tolerance.

5.17.3.31 set_LinearRelTol()

```
void Dove::set_LinearRelTol (
            double tol )
```

Set the value of linear relative tolerance.

5.17.3.32 set_NonlinearOutput()

```
void Dove::set_NonlinearOutput (
            bool choice )
```

Sets the non-linear output information according to user choice.

5.17.3.33 set_LinearOutput()

```
void Dove::set_LinearOutput (
            bool choice )
```

Sets the linear output information according to user choice.

5.17.3.34 set_Preconditioning()

```
void Dove::set_Preconditioning (
            bool choice )
```

Sets the boolean to determine whether or not to include preconditioning.

5.17.3.35 set_LinearMethod()

```
void Dove::set_LinearMethod (
             krylov_method choice )
```

Sets the linear solver method to user choice.

5.17.3.36 set_LineSearchMethod()

```
void Dove::set_LineSearchMethod (
             linesearch_type choice )
```

Sets the line search method to the user choice.

5.17.3.37 set_MaxNonLinearIterations()

```
void Dove::set_MaxNonLinearIterations (
             int it )
```

Set the maximum number of non-linear iterations.

5.17.3.38 set_MaxLinearIterations()

```
void Dove::set_MaxLinearIterations (
             int it )
```

Set the maximum number of linear iterations (or number of restarts)

5.17.3.39 set_RestartLimit()

```
void Dove::set_RestartLimit (
             int it )
```

Sets the number of iterations before restarting.

5.17.3.40 set_RecursionLevel()

```
void Dove::set_RecursionLevel (
             int level )
```

Sets the maximum level of recursion for the KMS method.

5.17.3.41 set_LinearStatus()

```
void Dove::set_LinearStatus (
             bool choice )
```

Sets the boolean to determine whether or not to treat as linear (true = Linear)

5.17.3.42 registerFunction() [1/2]

```
void Dove::registerFunction (
             int i,
             double(*)(int i, const Matrix< double > &u, double t, const void *data, const
Dove &dove) func )
```

Register the ith user function.

This function will register the ith user function into the object. That function must accept as arguments the function identifier i, a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifier i can be used to conveniently define coupling between nieghboring elements/variables in the system. In other words, the int i denotes not only the function being registered, but also the primary coupled variable for the function.

i.e., du_i/dt = Func(u_i all other u)

This will allow for this framework to also handle PDEs, whose coupling between ith and jth variables is usually done via nieghboring variables (i.e., u_i in a 1-D PDE couples with u_i-1 and u_i+1). A similar relational scheme is workable with multiple dimensions. Additional information about the coupling between the ith variable and other variables can be passed to the function via the void data pointer.

Note

> You are allowed to point to the same user function for all i, but you must make sure that the resulting system is non-singular (i.e., use argument i passed to the function to denote interally which function you are at).

5.17.3.43 registerFunction() [2/2]

```
void Dove::registerFunction (
             std::string name,
             double(*)(int i, const Matrix< double > &u, double t, const void *data, const
Dove &dove) func )
```

Register the named user function.

This function will register the named user function into the object. That function must accept as arguments the function identifier i, a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifier i can be used to conveniently define coupling between nieghboring elements/variables in the system. In other words, the int i denotes not only the function being registered, but also the primary coupled variable for the function.

i.e., du_i/dt = Func(u_i all other u)

This will allow for this framework to also handle PDEs, whose coupling between ith and jth variables is usually done via nieghboring variables (i.e., u_i in a 1-D PDE couples with u_i-1 and u_i+1). A similar relational scheme is workable with multiple dimensions. Additional information about the coupling between the ith variable and other variables can be passed to the function via the void data pointer.

Note

> You are allowed to point to the same user function for all i, but you must make sure that the resulting system is non-singular (i.e., use argument i passed to the function to denote interally which function you are at).

5.17.3.44 registerCoeff() [1/2]

```
void Dove::registerCoeff (
              int i,
              double(*)(int i, const Matrix< double > &u, double t, const void *data, const
Dove &dove) coeff )
```

Register the ith time coeff function.

This function will register the ith coeff function into the object. That function must accept as arguments the coefficient identifier i, a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifier i can be used to conveniently define identify where the coefficient may be applied spatially. In other words, if solving a PDE, the time coefficient may be a function of location in space, which can be potentially identified by int i.

For example, in 1-D space, the distance x can be computed as x = dx∗i for a regular grid.

5.17.3.45 registerCoeff() [2/2]

```
void Dove::registerCoeff (
              std::string name,
              double(*)(int i, const Matrix< double > &u, double t, const void *data, const
Dove &dove) coeff )
```

Register the named time coeff function.

This function will register the named coeff function into the object. That function must accept as arguments the coefficient identifier i, a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifier i can be used to conveniently define identify where the coefficient may be applied spatially. In other words, if solving a PDE, the time coefficient may be a function of location in space, which can be potentially identified by int i.

For example, in 1-D space, the distance x can be computed as x = dx∗i for a regular grid.

5.17.3.46 registerJacobi() [1/2]

```
void Dove::registerJacobi (
              int i,
              int j,
              double(*)(int i, int j, const Matrix< double > &u, double t, const void *data,
const Dove &dove) jac )
```

Register the i-jth element of jacobian.

This function will register the (i,j) jacobian function into the object. That function must accept as arguments the jacobi identifiers (i and j), a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifiers i and j can be used to determine which Jacobian function this should be, thus allowing a user to potentially reference the same function for all Jacobi elements, but return different results based on matrix location.

Jacobian elements are as follows: J_ij = d(func_i)/d(u_j) derivative of ith function with respect to jth variable.

Note

The jacobian information is used only in preconditioning actions taken by DOVE. The type of preconditioning can be choosen by the user. There are standard types of preconditioning available.

5.17.3.47 registerJacobi() [2/2]

```
void Dove::registerJacobi (
             std::string func_name,
             std::string var_name,
             double(*)(int i, int j, const Matrix< double > &u, double t, const void *data,
const Dove &dove) jac )
```

Register the named element of jacobian.

This function will register the named jacobian function into the object. That function must accept as arguments the jacobi identifiers (i and j), a constant Matrix for variables u, a double for time t, and a void data pointer. All of this information is required to be in the function parameters, but is not required to be used by the function. The indentifiers i and j can be used to determine which Jacobian function this should be, thus allowing a user to potentially reference the same function for all Jacobi elements, but return different results based on matrix location.

Jacobian elements are as follows: J_ij = d(func_i)/d(u_j) derivative of ith function with respect to jth variable.

Note

The jacobian information is used only in preconditioning actions taken by DOVE. The type of preconditioning can be choosen by the user. There are standard types of preconditioning available.

5.17.3.48 print_header()

```
void Dove::print_header (
             bool addNewLine )
```

Function to print out a header to output file.

5.17.3.49 print_newresult()

```
void Dove::print_newresult (
             bool addNewLine )
```

Function to print out the new result of n+1 time level.

5.17.3.50 print_result()

```
void Dove::print_result (
             bool addNewLine )
```

Function to print out the old result of n time level.

5.17.3.51 createJacobian()

```
void Dove::createJacobian ( )
```

Function to create and store a numerical jacobian.

5.17.3.52 getNumJacobian()

```
Matrix<double>& Dove::getNumJacobian ( )
```

Return reference to the numerical jacobian.

5.17.3.53 getCurrentU() [1/3]

```
Matrix<double>& Dove::getCurrentU ( )
```

Return reference to the n level solution.

5.17.3.54 getOldU() [1/3]

```
Matrix<double>& Dove::getOldU ( )
```

Return reference to the n-1 level solution.

5.17.3.55 getNewU() [1/3]

```
Matrix<double>& Dove::getNewU ( )
```

Return reference to the n+1 level solution.

5.17.3.56 getVariableIndex()

```
int Dove::getVariableIndex (
             std::string name ) const
```

Return the index of the variable whose name matches the given string (checks hash table)

5.17.3.57 getVariableName()

```
std::string Dove::getVariableName (
             int i )
```

Return the name of the variable based on the given index.

5.17.3.58 getMaxRate()

```
double Dove::getMaxRate ( )
```

Returns the value of the maximum rate of change for all variables.

5.17.3.59 getCurrentU() [2/3]

```
double Dove::getCurrentU (
              int i,
              const Matrix< double > & u ) const
```

Return the value of the n level solution for variable i.

5.17.3.60 getOldU() [2/3]

```
double Dove::getOldU (
              int i,
              const Matrix< double > & u ) const
```

Return the value of the n-1 level solution for variable i.

5.17.3.61 getNewU() [2/3]

```
double Dove::getNewU (
              int i,
              const Matrix< double > & u ) const
```

Return the value of the n+1 level solution for variable i.

5.17.3.62 getCurrentU() [3/3]

```
double Dove::getCurrentU (
              std::string name,
              const Matrix< double > & u ) const
```

Return the value of the n level solution for variable of given name.

5.17.3.63 getOldU() [3/3]

```
double Dove::getOldU (
              std::string name,
              const Matrix< double > & u ) const
```

Return the value of the n-1 level solution for variable of given name.

5.17.3.64 getNewU() [3/3]

```
double Dove::getNewU (
            std::string name,
            const Matrix< double > & u ) const
```

Return the value of the n+1 level solution for variable of given name.

5.17.3.65 coupledTimeDerivative() [1/2]

```
double Dove::coupledTimeDerivative (
            int i,
            const Matrix< double > & u ) const
```

Return the value of the ith variable's time derivative.

5.17.3.66 coupledTimeDerivative() [2/2]

```
double Dove::coupledTimeDerivative (
            std::string name,
            const Matrix< double > & u ) const
```

Return the value of the named variable's time derivative.

5.17.3.67 coupledDerivativeTimeDerivative()

```
double Dove::coupledDerivativeTimeDerivative (
            int i,
            int j,
            const Matrix< double > & u ) const
```

Return the value of the ith variable's time derivative's jth derivative.

5.17.3.68 getUserData()

```
const void* Dove::getUserData ( )
```

Return pointer to user data.

5.17.3.69 getNumFunc()

```
int Dove::getNumFunc ( ) const
```

Return the number of functions.

5.17.3.70 getTimeStep()

```
double Dove::getTimeStep ( ) const
```

Return the current time step.

5.17.3.71 getTimeStepOld()

```
double Dove::getTimeStepOld ( ) const
```

Return the old time step.

5.17.3.72 getEndTime()

```
double Dove::getEndTime ( ) const
```

Return value of end time.

5.17.3.73 getCurrentTime()

```
double Dove::getCurrentTime ( ) const
```

Return the value of current time.

5.17.3.74 getOldTime()

```
double Dove::getOldTime ( ) const
```

Return the value of the previous time.

5.17.3.75 getOlderTime()

```
double Dove::getOlderTime ( ) const
```

Return the value of the older previous time.

5.17.3.76 getStartTime()

```
double Dove::getStartTime ( ) const
```

Return the value of the start time.

5.17.3.77 getMinTimeStep()

```
double Dove::getMinTimeStep ( )
```

Return the value of the minimum time step.

5.17.3.78 getMaxTimeStep()

```
double Dove::getMaxTimeStep ( )
```

Return the value of the maximum time step.

5.17.3.79 getOutputTime()

```
double Dove::getOutputTime ( )
```

Return the time that output is printed on.

5.17.3.80 getFile()

```
FILE* Dove::getFile ( )
```

Return the pointer to the output file.

5.17.3.81 hasConverged()

```
bool Dove::hasConverged ( )
```

Returns state of convergence.

5.17.3.82 getNonlinearResidual()

```
double Dove::getNonlinearResidual ( )
```

Returns the current value of the non-linear residual.

5.17.3.83 getNonlinearRelativeRes()

```
double Dove::getNonlinearRelativeRes ( )
```

Returns the current value of the non-linear relative residual.

5.17.3.84 allSteadyState()

```
bool Dove::allSteadyState ( ) const
```

Returns a boolean to determine whether or not all variables are steady (true = all steady)

5.17.3.85 isSteadyState()

```
bool Dove::isSteadyState (
             int i ) const
```

Returns true if the ith variable is steady-state.

5.17.3.86 getIntegrationType()

```
integrate_type Dove::getIntegrationType ( )
```

Returns the type of integration method in use.

5.17.3.87 getIntegrationMethod()

```
integrate_subtype Dove::getIntegrationMethod ( )
```

Returns the method of time integration.

5.17.3.88 getTimeStepper()

```
timestep_type Dove::getTimeStepper ( )
```

Returns the time stepper method.

5.17.3.89 getPreconditioner()

```
precond_type Dove::getPreconditioner ( )
```

Returns the preconditioner type.

5.17.3.90 getLinesearchMethod()

```
linesearch_type Dove::getLinesearchMethod ( )
```

Returns the method of line search.

5.17.3.91 getLinearMethod()

```
krylov_method Dove::getLinearMethod ( )
```

Returns the linear method used.

5.17.3.92 isPreconditioned()

```
bool Dove::isPreconditioned ( )
```

Returns true if using preconditioning.

5.17.3.93 getLinearToleranceABS()

```
double Dove::getLinearToleranceABS ( )
```

Returns the value of linear tolerance (absolute)

5.17.3.94 getLinearToleranceREL()

```
double Dove::getLinearToleranceREL ( )
```

Returns the value of linear tolerance (relative)

5.17.3.95 getNonlinearToleranceABS()

```
double Dove::getNonlinearToleranceABS ( )
```

Returns the value of nonlinear tolerance (absolute)

5.17.3.96 getNonlinearToleranceREL()

```
double Dove::getNonlinearToleranceREL ( )
```

Returns the value of nonlinear tolerance (relative)

5.17.3.97 getMaxNonlinearIterations()

```
int Dove::getMaxNonlinearIterations ( )
```

Returns the maximum allowable nonlinear iterations.

5.17.3.98 getMaxLinearIterations()

```
int Dove::getMaxLinearIterations ( )
```

Returns the maximum allowable linear iterations.

5.17.3.99 isLinear()

```
bool Dove::isLinear ( )
```

Returns true if the system is being treated as linear.

5.17.3.100 getRestartLevel()

```
int Dove::getRestartLevel ( )
```

Returns the number of iterations allowed before restart for Krylov methods.

5.17.3.101 getRecursionLevel()

```
int Dove::getRecursionLevel ( )
```

Returns the number of recursion levels allow for Krylov preconditioning.

5.17.3.102 isValidName()

```
bool Dove::isValidName (
             std::string name )
```

Returns true if the given name is a variable in the DOVE system.

5.17.3.103 getJacobiMap()

```
std::map<int, double (*) (int i, int j, const Matrix<double> &u, double t, const void *data,
const Dove &dove)>& Dove::getJacobiMap (
             int i )
```

Function to return a reference to the Jacobian Matrix map at the ith row of the matrix.

5.17.3.104 ComputeTimeStep()

```
double Dove::ComputeTimeStep ( )
```

Returns a computed value for the next time step.

5.17.3.105 Eval_Func()

```
double Dove::Eval_Func (
             int i,
             const Matrix< double > & u,
             double t )
```

Evaluate user function i at given u matrix and time t.

5.17.3.106 Eval_Coeff()

```
double Dove::Eval_Coeff (
             int i,
             const Matrix< double > & u,
             double t )
```

Evaluate user time coefficient function i at given u matrix and time t.

5.17.3.107 Eval_Jacobi()

```
double Dove::Eval_Jacobi (
             int i,
             int j,
             const Matrix< double > & u,
             double t )
```

Evaluate user jacobian function for (i,j) at given u matrix and time t.

5.17.3.108 solve_timestep()

```
int Dove::solve_timestep ( )
```

Function to solve a single time step.

5.17.3.109 validate_precond()

```
void Dove::validate_precond ( )
```

Function to validate and set preconditioning pointer.

5.17.3.110 validate_linearsteps()

```
void Dove::validate_linearsteps ( )
```

Function to check and validate the number of linear iterations.

5.17.3.111 validate_method()

```
void Dove::validate_method ( )
```

Function to check and validate the time integration method.

5.17.3.112 update_states()

```
void Dove::update_states ( )
```

Function to update the stateful information.

5.17.3.113 update_timestep()

```
void Dove::update_timestep ( )
```

Function to update the timestep for the simulation.

5.17.3.114 reset_all()

```
void Dove::reset_all ( )
```

Reset all the states.

5.17.3.115 solve_all()

```
int Dove::solve_all ( )
```

Function to solve the system of equations and print results to file (returns 0 on success)

This function will iteratively go through and solve the system for all time steps until either failure occurs or the final time has been reached. Output will be placed into the user's output file or a default output file. This function will assume that the initial conditions have already been set for each variable by the user.

5.17.3.116 solve_FE()

```
int Dove::solve_FE ( )
```

Solver function for explicit-FE method.

This function will solve the Dove system of equations using the standard Forward-Euler method. In this function, DOVE will call the user defined rate functions and use that information at the previous time level to solve for the next time level directly.

unp1[i] = (Rn[i]∗un[i] + dt∗func[i](unp1)) / Rnp1[i]

5.17.3.117 solve_RK4()

```
int Dove::solve_RK4 ( )
```

Solver function for explicit-RK4 method.

This function will solve the Dove system of equations using the Runge-Kutta 4th order method. In this function, D←OVE will call user defined rate functions as necessary and use that information at the previous time level to provide an estimate to the solution at the next time level.

5.17.3.118 solve_RKF()

```
int Dove::solve_RKF ( )
```

Solver function for explicit-RKF method.

This function will solve the Dove system of equations using the Runge-Kutta-Fehlberg method. In this function, D←OVE will call user defined rate runctions as necessary and use that information at the previous time level to provide an estimate to the solution at the next time level.

5.17.4 Member Data Documentation

5.17.4.1 var_names

```
Matrix<std::string> Dove::var_names  [protected]
```

Matrix of variable names (access names by index in numerical order)

5.17.4.2 var_names_hash

```
std::unordered_map<std::string, int> Dove::var_names_hash  [protected]
```

Hash table of variable names and corresponding indices (access index by name)

5.17.4.3 var_steady

```
Matrix<int> Dove::var_steady [protected]
```

Matrix of boolean args used to dictate which variables are considered steady-state (if any)

5.17.4.4 un

```
Matrix<double> Dove::un [protected]
```

Matrix for nth level solution vector.

5.17.4.5 unp1

```
Matrix<double> Dove::unp1 [protected]
```

Matrix for n+1 level solution vector.

5.17.4.6 unm1

```
Matrix<double> Dove::unm1 [protected]
```

Matrix for n-1 level solution vector.

5.17.4.7 dt

```
double Dove::dt [protected]
```

Time step between n and n+1 time levels.

5.17.4.8 dt_old

```
double Dove::dt_old [protected]
```

Time step between n and n-1 time levels.

5.17.4.9 time_end

```
double Dove::time_end [protected]
```

Time on which to end the ODE simulations.

5.17.4.10 time_start

```
double Dove::time_start  [protected]
```

Time on which to start the ODE simulations.

5.17.4.11 time

```
double Dove::time  [protected]
```

Value of current time.

5.17.4.12 time_old

```
double Dove::time_old  [protected]
```

Value of previous time.

5.17.4.13 time_older

```
double Dove::time_older  [protected]
```

Value of older previous time.

5.17.4.14 dtmin

```
double Dove::dtmin  [protected]
```

Minimum allowable time step.

5.17.4.15 dtmin_con

```
double Dove::dtmin_con  [protected]
```

Minimum allowable time step if solution has converged.

5.17.4.16 dtmax

```
double Dove::dtmax  [protected]
```

Maximum allowable time step.

5.17.4.17 tolerance

```
double Dove::tolerance [protected]
```

Residual tolerance desired (or level of accuracy desired)

5.17.4.18 t_count

```
double Dove::t_count [protected]
```

Counter designed to keep track of how often we print results.

5.17.4.19 t_out

```
double Dove::t_out [protected]
```

Direct Dove to only print results after specific time increments.

5.17.4.20 int_type

```
integrate_type Dove::int_type [protected]
```

Type of time integration to use.

5.17.4.21 int_sub

```
integrate_subtype Dove::int_sub [protected]
```

Subtype of time integration scheme to use.

5.17.4.22 timestepper

```
timestep_type Dove::timestepper [protected]
```

Type of time stepper to be used.

5.17.4.23 preconditioner

```
precond_type Dove::preconditioner [protected]
```

Type of preconditioner to use.

5.17.4.24 line_type

```
linesearch_type Dove::line_type [protected]
```

Type of linesearch method to use.

5.17.4.25 Output

```
FILE* Dove::Output [protected]
```

File to where simulation results will be place.

5.17.4.26 num_func

```
int Dove::num_func [protected]
```

Number of functions in the system of ODEs.

5.17.4.27 Converged

```
bool Dove::Converged [protected]
```

Boolean to hold information on whether or not last step converged.

5.17.4.28 DoveOutput

```
bool Dove::DoveOutput [protected]
```

Boolean to determine whether or not to print Dove messages to console.

5.17.4.29 DoveFileOutput

```
bool Dove::DoveFileOutput [protected]
```

Boolean to determine whether or not to print Dove ouput to the file.

5.17.4.30 DoveHeader

```
bool Dove::DoveHeader [protected]
```

Boolean to determine whether or not to print the Dove header to file.

5.17.4.31 Preconditioner

```
bool Dove::Preconditioner  [protected]
```

Boolean to determine whether or not to use a preconditioner.

5.17.4.32 Linear

```
bool Dove::Linear  [protected]
```

Boolean to determine whether or not to treat problem as linear.

5.17.4.33 AllSteadyState

```
bool Dove::AllSteadyState  [protected]
```

Boolean to determine whether or not all variables are steady (true = all steady-state)

5.17.4.34 linmax

```
int Dove::linmax  [protected]
```

Users requested maximum number of linear steps.

5.17.4.35 timesteps

```
int Dove::timesteps  [protected]
```

Running count of number of time steps taken.

5.17.4.36 user_func

```
Matrix<double (*) (int i, const Matrix<double> &u, double t, const void *data, const Dove
&dove)> Dove::user_func  [protected]
```

Matrix object for user defined rate functions.

5.17.4.37 user_coeff

```
Matrix<double (*) (int i, const Matrix<double> &u, double t, const void *data, const Dove
&dove)> Dove::user_coeff  [protected]
```

Matrix object for user defined time coefficients (optional)

5.17.4.38 user_jacobi

```
std::vector< std::map<int, double (*) (int i, int j, const Matrix<double> &u, double t, const
void *data, const Dove &dove)> > Dove::user_jacobi  [protected]
```

A vector of Maps for user defined Jacobian elements (optional)

This structure creates a Sparse Matrix of functions whose sparcity pattern is unknown at creation. Each "vector" index denotes a row in the full matrix. In each row, there is a map of the non-zero elements. Doing the mapping in this way allows for the sparcity of the matrix to easily change while also allowing for relatively fast access to the non-zero elements.

5.17.4.39 user_data

```
const void* Dove::user_data  [protected]
```

Pointer for user defined data structure.

5.17.4.40 newton_dat

```
PJFNK_DATA Dove::newton_dat  [protected]
```

Data structure for the PJFNK iterative method

5.17.4.41 residual

```
int(* Dove::residual) (const Matrix< double > &x, Matrix< double > &F, const void *res_data)
[protected]
```

Function pointer for the residual function of DOVE.

5.17.4.42 precon

```
int(* Dove::precon) (const Matrix< double > &r, Matrix< double > &p, const void *precon_data)
[protected]
```

Function pointer for the preconditioning operation of DOVE.

5.17.4.43 jac_dat

```
NUM_JAC_DATA Dove::jac_dat  [protected]
```

Data structure for making Numerical Jacobian Matrices.

5.17.4.44 Jacobian

```
Matrix<double> Dove::Jacobian [protected]
```

Matrix to hold numerical jacobian.

The documentation for this class was generated from the following file:

- dove.h

5.18 FINCH_DATA Struct Reference

Data structure for the FINCH object.

```
#include <finch.h>
```

Public Attributes

- int d = 0

  Dimension of the problem: 0 = cartesian, 1 = cylindrical, 2 = spherical.
- double dt = 0.0125

  Time step.
- double dt_old = 0.0125

  Previous time step.
- double dt_const = 0.0

  Forced selection of time step.
- double T = 1.0

  Total time.
- double dz = 0.1

  Space step.
- double L = 1.0

  Total space.
- double s = 1.0

  Char quantity (spherical = 1, cylindrical = length, cartesian = area)
- double t = 0.0

  Current Time.
- double t_old = 0.0

  Previous Time.
- double t_out = 0.0

  How often to print output.
- double t_count = 0.0

  Counter to determine when output is to be printed.
- double uT = 0.0

  Total amount of conserved quantity in domain.
- double uT_old = 0.0

  Old Total amount of conserved quantity.
- double uAvg = 0.0

  Average amount of conserved quantity in domain.
- double uAvg_old = 0.0

  Old Average amount of conserved quantity.

- double uIC = 0.0

  Initial condition of Conserved Quantity (if constant)
- double vIC = 1.0

  Initial condition of Velocity (if constant)
- double DIC = 1.0

  Initial condition of Dispersion (if constant)
- double kIC = 1.0

  Initial condition of Reaction (if constant)
- double RIC = 1.0

  Initial condition of the Time Coefficient (if constant)
- double uo = 1.0

  Boundary Value of Conserved Quantity.
- double vo = 1.0

  Boundary Value of Velocity.
- double Do = 1.0

  Boundary Value of Dispersion.
- double ko = 1.0

  Boundary Value of Reaction.
- double Ro = 1.0

  Boundary Value of Time Coefficient.
- double kfn = 1.0

  Film mass transfer coefficient Old.
- double kfnp1 = 1.0

  Film mass transfer coefficient New.
- double lambda_I

  Boundary Coefficient for Implicit Neumann (Calculated at Runtime)
- double lambda_E

  Boundary Coefficient for Explicit Neumann (Calculated at Runtime)
- int LN = 10

  Number of nodes.
- bool CN = true

  True if Crank-Nicholson, false if Implicit, never use explicit.
- bool Update = false

  Flag to check if the system needs updating.
- bool Dirichlet = false

  Flag to indicate use of Dirichlet or Neumann starting boundary.
- bool CheckMass = false

  Flag to indicate whether or not mass is to be checked.
- bool ExplicitFlux = false

  Flag to indicate whether or not to use fully explicit flux limiters.
- bool Iterative = true

  Flag to indicate whether to solve directly, or iteratively.
- bool SteadyState = false

  Flag to determine whether or not to solve the steady-state problem.
- bool NormTrack = true

  Flag to determine whether or not to track the norms during simulation.
- double beta = 0.5

  Scheme type indicator: 0.5=CN & 1.0=Implicit; all else NULL.
- double tol_rel = 1e-6

  Relative Tolerance for Convergence.
- double tol_abs = 1e-6

Absolute Tolerance for Convergence.
- int max_iter = 20

  Maximum number of iterations allowed.
- int total_iter = 0

  Total number of iterations made.
- int nl_method = FINCH_Picard

  Non-linear solution method - default = FINCH_Picard.
- std::vector< double > CL_I

  Left side, implicit coefficients (Calculated at Runtime)
- std::vector< double > CL_E

  Left side, explicit coefficients (Calculated at Runtime)
- std::vector< double > CC_I

  Centered, implicit coefficients (Calculated at Runtime)
- std::vector< double > CC_E

  Centered, explicit coefficients (Calculated at Runtime)
- std::vector< double > CR_I

  Right side, implicit coefficients (Calculated at Runtime)
- std::vector< double > CR_E

  Right side, explicit coefficients (Calculated at Runtime)
- std::vector< double > fL_I

  Left side, implicit fluxes (Calculated at Runtime)
- std::vector< double > fL_E

  Left side, explicit fluxes (Calculated at Runtime)
- std::vector< double > fC_I

  Centered, implicit fluxes (Calculated at Runtime)
- std::vector< double > fC_E

  Centered, explicit fluxes (Calculated at Runtime)
- std::vector< double > fR_I

  Right side, implicit fluxes (Calculated at Runtime)
- std::vector< double > fR_E

  Right side, explicit fluxes (Calculated at Runtime)
- std::vector< double > OI

  Implicit upper diagonal matrix elements (Calculated at Runtime)
- std::vector< double > OE

  Explicit upper diagonal matrix elements (Calculated at Runtime)
- std::vector< double > NI

  Implicit diagonal matrix elements (Calculated at Runtime)
- std::vector< double > NE

  Explicit diagonal matrix elements (Calculated at Runtime)
- std::vector< double > MI

  Implicit lower diagonal matrix elements (Calculated at Runtime)
- std::vector< double > ME

  Explicit lower diagonal matrix elements (Calculated at Runtime)
- std::vector< double > uz_l_I
- std::vector< double > uz_lm1_I
- std::vector< double > uz_lp1_I

  Implicit local slopes (Calculated at Runtime)
- std::vector< double > uz_l_E
- std::vector< double > uz_lm1_E
- std::vector< double > uz_lp1_E

  Explicit local slopes (Calculated at Runtime)

- Matrix< double > unm1

  Conserved Quantity Older.
- Matrix< double > un

  Conserved Quantity Old.
- Matrix< double > unp1

  Conserved Quantity New.
- Matrix< double > u_star

  Conserved Quantity Projected New.
- Matrix< double > ubest

  Best found solution if solving iteratively.
- Matrix< double > vn

  Velocity Old.
- Matrix< double > vnp1

  Velocity New.
- Matrix< double > Dn

  Dispersion Old.
- Matrix< double > Dnp1

  Dispersion New.
- Matrix< double > kn

  Reaction Old.
- Matrix< double > knp1

  Reaction New.
- Matrix< double > Sn

  Forcing Function Old.
- Matrix< double > Snp1

  Forcing Function New.
- Matrix< double > Rn

  Time Coeff Old.
- Matrix< double > Rnp1

  Time Coeff New.
- Matrix< double > Fn

  Flux Limiter Old.
- Matrix< double > Fnp1

  Flux Limiter New.
- Matrix< double > gI

  Implicit Side Boundary Conditions.
- Matrix< double > gE

  Explicit Side Boundary Conditions.
- Matrix< double > res

  Current residual.
- Matrix< double > pres

  Current search direction.
- int(∗ callroutine)(const void ∗user_data)

  Function pointer to executioner (DEFAULT = default_execution)
- int(∗ setic)(const void ∗user_data)

  Function pointer to initial conditions (DEFAULT = default_ic)
- int(∗ settime)(const void ∗user_data)

  Function pointer to set time step (DEFAULT = default_timestep)
- int(∗ setpreprocess)(const void ∗user_data)

  Function pointer to preprocesses (DEFAULT = default_preprocess)
- int(∗ solve)(const void ∗user_data)

Function pointer to the solver (DEFAULT = default_solve)
- int($*$ setparams)(const void $*$user_data)

  Function pointer to set parameters (DEFAULT = default_params)
- int($*$ discretize)(const void $*$user_data)

  Function pointer to discretization (DEFAULT = ospre_discretization)
- int($*$ setbcs)(const void $*$user_data)
- int($*$ evalres)(const Matrix$<$ double $>$ &x, Matrix$<$ double $>$ &res, const void $*$user_data)

  Function pointer to the residual function (DEFAULT = default_res)
- int($*$ evalprecon)(const Matrix$<$ double $>$ &b, Matrix$<$ double $>$ &p, const void $*$user_data)

  Function pointer to the preconditioning function (DEFAULT = default_precon)
- int($*$ setpostprocess)(const void $*$user_data)

  Function pointer to the postprocesses (DEFAULT = default_postprocess)
- int($*$ resettime)(const void $*$user_data)

  Function pointer to reset time (DEFAULT = default_reset)
- PICARD_DATA picard_dat

  Data structure for PICARD method (no need to use this)
- PJFNK_DATA pjfnk_dat

  Data structure for PJFNK method (more rigours method)
- const void $*$ param_data

  User's data structure used to evaluate the parameter function (Must override if setparams is overriden)

5.18.1 Detailed Description

Data structure for the FINCH object.

C-style object that holds data, functions, and other structures necessary to discretize and solve a FINCH problem. All of this information must be overriden or initialized prior to running a FINCH simulation. Many, many default functions are provided to make it easier to incorporate FINCH into other problems. The main function to override will be the setparams function. This will be a function that the user provides to tell the FINCH simulation how the parameters of the problem vary in time and space and whether or not they are coupled the the variable u. All functions are overridable and several can be skipped entirely, or called directly at different times in the execution of a particular routine. This make FINCH extremely flexible to the user.

Note

All parameters and dimensions do not carry any units with them. The user is required to keep track of all their own units in their particular problem and ensure that units will cancel and be consistent in their own physical model.

5.18.2 Member Data Documentation

5.18.2.1 d

```
int FINCH_DATA::d = 0
```

Dimension of the problem: 0 = cartesian, 1 = cylindrical, 2 = spherical.

5.18.2.2 dt

```
double FINCH_DATA::dt = 0.0125
```

Time step.

5.18.2.3 dt_old

```
double FINCH_DATA::dt_old = 0.0125
```

Previous time step.

5.18.2.4 dt_const

```
double FINCH_DATA::dt_const = 0.0
```

Forced selection of time step.

5.18.2.5 T

```
double FINCH_DATA::T = 1.0
```

Total time.

5.18.2.6 dz

```
double FINCH_DATA::dz = 0.1
```

Space step.

5.18.2.7 L

```
double FINCH_DATA::L = 1.0
```

Total space.

5.18.2.8 s

```
double FINCH_DATA::s = 1.0
```

Char quantity (spherical = 1, cylindrical = length, cartesian = area)

5.18.2.9 t

```
double FINCH_DATA::t = 0.0
```

Current Time.

5.18.2.10 t_old

```
double FINCH_DATA::t_old = 0.0
```

Previous Time.

5.18.2.11 t_out

```
double FINCH_DATA::t_out = 0.0
```

How often to print output.

5.18.2.12 t_count

```
double FINCH_DATA::t_count = 0.0
```

Counter to determine when output is to be printed.

5.18.2.13 uT

```
double FINCH_DATA::uT = 0.0
```

Total amount of conserved quantity in domain.

5.18.2.14 uT_old

```
double FINCH_DATA::uT_old = 0.0
```

Old Total amount of conserved quantity.

5.18.2.15 uAvg

```
double FINCH_DATA::uAvg = 0.0
```

Average amount of conserved quantity in domain.

5.18.2.16 uAvg_old

```
double FINCH_DATA::uAvg_old = 0.0
```

Old Average amount of conserved quantity.

5.18.2.17 uIC

```
double FINCH_DATA::uIC = 0.0
```

Initial condition of Conserved Quantity (if constant)

5.18.2.18 vIC

```
double FINCH_DATA::vIC = 1.0
```

Initial condition of Velocity (if constant)

5.18.2.19 DIC

```
double FINCH_DATA::DIC = 1.0
```

Initial condition of Dispersion (if constant)

5.18.2.20 kIC

```
double FINCH_DATA::kIC = 1.0
```

Initial condition of Reaction (if constant)

5.18.2.21 RIC

```
double FINCH_DATA::RIC = 1.0
```

Initial condition of the Time Coefficient (if constant)

5.18.2.22 uo

```
double FINCH_DATA::uo = 1.0
```

Boundary Value of Conserved Quantity.

5.18.2.23 vo

```
double FINCH_DATA::vo = 1.0
```

Boundary Value of Velocity.

5.18.2.24 Do

```
double FINCH_DATA::Do = 1.0
```

Boundary Value of Dispersion.

5.18.2.25 ko

```
double FINCH_DATA::ko = 1.0
```

Boundary Value of Reaction.

5.18.2.26 Ro

```
double FINCH_DATA::Ro = 1.0
```

Boundary Value of Time Coefficient.

5.18.2.27 kfn

```
double FINCH_DATA::kfn = 1.0
```

Film mass transfer coefficient Old.

5.18.2.28 kfnp1

```
double FINCH_DATA::kfnp1 = 1.0
```

Film mass transfer coefficient New.

5.18.2.29 lambda_I

```
double FINCH_DATA::lambda_I
```

Boundary Coefficient for Implicit Neumann (Calculated at Runtime)

5.18.2.30 lambda_E

```
double FINCH_DATA::lambda_E
```

Boundary Coefficient for Explicit Neumann (Calculated at Runtime)

5.18.2.31 LN

```
int FINCH_DATA::LN = 10
```

Number of nodes.

5.18.2.32 CN

```
bool FINCH_DATA::CN = true
```

True if Crank-Nicholson, false if Implicit, never use explicit.

5.18.2.33 Update

```
bool FINCH_DATA::Update = false
```

Flag to check if the system needs updating.

5.18.2.34 Dirichlet

```
bool FINCH_DATA::Dirichlet = false
```

Flag to indicate use of Dirichlet or Neumann starting boundary.

5.18.2.35 CheckMass

```
bool FINCH_DATA::CheckMass = false
```

Flag to indicate whether or not mass is to be checked.

5.18.2.36 ExplicitFlux

```
bool FINCH_DATA::ExplicitFlux = false
```

Flag to indicate whether or not to use fully explicit flux limiters.

5.18.2.37 Iterative

```
bool FINCH_DATA::Iterative = true
```

Flag to indicate whether to solve directly, or iteratively.

5.18.2.38 SteadyState

```
bool FINCH_DATA::SteadyState = false
```

Flag to determine whether or not to solve the steady-state problem.

5.18.2.39 NormTrack

```
bool FINCH_DATA::NormTrack = true
```

Flag to determine whether or not to track the norms during simulation.

5.18.2.40 beta

```
double FINCH_DATA::beta = 0.5
```

Scheme type indicator: 0.5=CN & 1.0=Implicit; all else NULL.

5.18.2.41 tol_rel

```
double FINCH_DATA::tol_rel = 1e-6
```

Relative Tolerance for Convergence.

5.18.2.42 tol_abs

```
double FINCH_DATA::tol_abs = 1e-6
```

Absolute Tolerance for Convergence.

5.18.2.43 max_iter

```
int FINCH_DATA::max_iter = 20
```

Maximum number of iterations allowed.

5.18.2.44 total_iter

```
int FINCH_DATA::total_iter = 0
```

Total number of iterations made.

5.18.2.45 nl_method

```
int FINCH_DATA::nl_method = FINCH_Picard
```

Non-linear solution method - default = FINCH_Picard.

5.18.2.46 CL_I

```
std::vector<double> FINCH_DATA::CL_I
```

Left side, implicit coefficients (Calculated at Runtime)

5.18.2.47 CL_E

```
std::vector<double> FINCH_DATA::CL_E
```

Left side, explicit coefficients (Calculated at Runtime)

5.18.2.48 CC_I

```
std::vector<double> FINCH_DATA::CC_I
```

Centered, implicit coefficients (Calculated at Runtime)

5.18.2.49 CC_E

```
std::vector<double> FINCH_DATA::CC_E
```

Centered, explicit coefficients (Calculated at Runtime)

5.18.2.50 CR_I

```
std::vector<double> FINCH_DATA::CR_I
```

Right side, implicit coefficients (Calculated at Runtime)

5.18.2.51 CR_E

```
std::vector<double> FINCH_DATA::CR_E
```

Right side, explicit coefficients (Calculated at Runtime)

5.18.2.52 fL_I

```
std::vector<double> FINCH_DATA::fL_I
```

Left side, implicit fluxes (Calculated at Runtime)

5.18.2.53 fL_E

```
std::vector<double> FINCH_DATA::fL_E
```

Left side, explicit fluxes (Calculated at Runtime)

5.18.2.54 fC_I

```
std::vector<double> FINCH_DATA::fC_I
```

Centered, implicit fluxes (Calculated at Runtime)

5.18.2.55 fC_E

```
std::vector<double> FINCH_DATA::fC_E
```

Centered, explicit fluxes (Calculated at Runtime)

5.18.2.56 fR_I

```
std::vector<double> FINCH_DATA::fR_I
```

Right side, implicit fluxes (Calculated at Runtime)

5.18.2.57 fR_E

```
std::vector<double> FINCH_DATA::fR_E
```

Right side, explicit fluxes (Calculated at Runtime)

5.18.2.58 OI

```
std::vector<double> FINCH_DATA::OI
```

Implicit upper diagonal matrix elements (Calculated at Runtime)

5.18.2.59 OE

```
std::vector<double> FINCH_DATA::OE
```

Explicit upper diagonal matrix elements (Calculated at Runtime)

5.18.2.60 NI

```
std::vector<double> FINCH_DATA::NI
```

Implicit diagonal matrix elements (Calculated at Runtime)

5.18.2.61 NE

```
std::vector<double> FINCH_DATA::NE
```

Explicit diagonal matrix elements (Calculated at Runtime)

5.18.2.62 MI

```
std::vector<double> FINCH_DATA::MI
```

Implicit lower diagonal matrix elements (Calculated at Runtime)

5.18.2.63 ME

```
std::vector<double> FINCH_DATA::ME
```

Explicit lower diagonal matrix elements (Calculated at Runtime)

5.18.2.64 uz_l_I

```
std::vector<double> FINCH_DATA::uz_l_I
```

5.18.2.65 uz_lm1_I

```
std::vector<double> FINCH_DATA::uz_lm1_I
```

5.18.2.66 uz_lp1_I

```
std::vector<double> FINCH_DATA::uz_lp1_I
```

Implicit local slopes (Calculated at Runtime)

5.18.2.67 uz_l_E

```
std::vector<double> FINCH_DATA::uz_l_E
```

5.18.2.68 uz_lm1_E

```
std::vector<double> FINCH_DATA::uz_lm1_E
```

5.18.2.69 uz_lp1_E

```
std::vector<double> FINCH_DATA::uz_lp1_E
```

Explicit local slopes (Calculated at Runtime)

5.18.2.70 unm1

```
Matrix<double> FINCH_DATA::unm1
```

Conserved Quantity Older.

5.18.2.71 un

```
Matrix<double> FINCH_DATA::un
```

Conserved Quantity Old.

5.18.2.72 unp1

```
Matrix<double> FINCH_DATA::unp1
```

Conserved Quantity New.

5.18.2.73 u_star

```
Matrix<double> FINCH_DATA::u_star
```

Conserved Quantity Projected New.

5.18.2.74 ubest

```
Matrix<double> FINCH_DATA::ubest
```

Best found solution if solving iteratively.

5.18.2.75 vn

```
Matrix<double> FINCH_DATA::vn
```

Velocity Old.

5.18.2.76 vnp1

```
Matrix<double> FINCH_DATA::vnp1
```

Velocity New.

5.18.2.77 Dn

```
Matrix<double> FINCH_DATA::Dn
```

Dispersion Old.

5.18.2.78 Dnp1

```
Matrix<double> FINCH_DATA::Dnp1
```

Dispersion New.

5.18.2.79 kn

`Matrix<double> FINCH_DATA::kn`

Reaction Old.

5.18.2.80 knp1

`Matrix<double> FINCH_DATA::knp1`

Reaction New.

5.18.2.81 Sn

`Matrix<double> FINCH_DATA::Sn`

Forcing Function Old.

5.18.2.82 Snp1

`Matrix<double> FINCH_DATA::Snp1`

Forcing Function New.

5.18.2.83 Rn

`Matrix<double> FINCH_DATA::Rn`

Time Coeff Old.

5.18.2.84 Rnp1

`Matrix<double> FINCH_DATA::Rnp1`

Time Coeff New.

5.18.2.85 Fn

`Matrix<double> FINCH_DATA::Fn`

Flux Limiter Old.

5.18.2.86 Fnp1

```
Matrix<double> FINCH_DATA::Fnp1
```

Flux Limiter New.

5.18.2.87 gI

```
Matrix<double> FINCH_DATA::gI
```

Implicit Side Boundary Conditions.

5.18.2.88 gE

```
Matrix<double> FINCH_DATA::gE
```

Explicit Side Boundary Conditions.

5.18.2.89 res

```
Matrix<double> FINCH_DATA::res
```

Current residual.

5.18.2.90 pres

```
Matrix<double> FINCH_DATA::pres
```

Current search direction.

5.18.2.91 callroutine

```
int(* FINCH_DATA::callroutine) (const void *user_data)
```

Function pointer to executioner (DEFAULT = default_execution)

5.18.2.92 setic

```
int(* FINCH_DATA::setic) (const void *user_data)
```

Function pointer to initial conditions (DEFAULT = default_ic)

5.18.2.93 settime

```
int(* FINCH_DATA::settime) (const void *user_data)
```

Function pointer to set time step (DEFAULT = default_timestep)

5.18.2.94 setpreprocess

```
int(* FINCH_DATA::setpreprocess) (const void *user_data)
```

Function pointer to preprocesses (DEFAULT = default_preprocess)

5.18.2.95 solve

```
int(* FINCH_DATA::solve) (const void *user_data)
```

Function pointer to the solver (DEFAULT = default_solve)

5.18.2.96 setparams

```
int(* FINCH_DATA::setparams) (const void *user_data)
```

Function pointer to set parameters (DEFAULT = default_params)

5.18.2.97 discretize

```
int(* FINCH_DATA::discretize) (const void *user_data)
```

Function pointer to discretization (DEFAULT = ospre_discretization)

5.18.2.98 setbcs

```
int(* FINCH_DATA::setbcs) (const void *user_data)
```

Function pointer to set boundary conditions (DEFAULT = default_bcs)

5.18.2.99 evalres

```
int(* FINCH_DATA::evalres) (const Matrix< double > &x, Matrix< double > &res, const void
*user_data)
```

Function pointer to the residual function (DEFAULT = default_res)

5.18.2.100 evalprecon

```
int(* FINCH_DATA::evalprecon) (const Matrix< double > &b, Matrix< double > &p, const void
*user_data)
```

Function pointer to the preconditioning function (DEFAULT = default_precon)

5.18.2.101 setpostprocess

```
int(* FINCH_DATA::setpostprocess) (const void *user_data)
```

Function pointer to the postprocesses (DEFAULT = default_postprocess)

5.18.2.102 resettime

```
int(* FINCH_DATA::resettime) (const void *user_data)
```

Function pointer to reset time (DEFAULT = default_reset)

5.18.2.103 picard_dat

```
PICARD_DATA FINCH_DATA::picard_dat
```

Data structure for PICARD method (no need to use this)

5.18.2.104 pjfnk_dat

```
PJFNK_DATA FINCH_DATA::pjfnk_dat
```

Data structure for PJFNK method (more rigours method)

5.18.2.105 param_data

```
const void* FINCH_DATA::param_data
```

User's data structure used to evaluate the parameter function (Must override if setparams is overriden)

The documentation for this struct was generated from the following file:

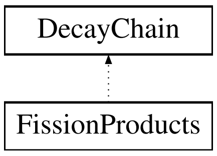
- finch.h

5.19 FissionProducts Class Reference

FissionProducts class object to create decay chains from fission yields.

```
#include <fairy.h>
```

Inheritance diagram for FissionProducts:



Public Member Functions

- FissionProducts ()

  Default Constructor.
- ∼FissionProducts ()

  Default Destructor.
- void DisplayInfo ()

  Display the FAIRY information, initial materials and fractions.
- void DisplayMap ()

  Display the map for the DecayChain object.
- void loadNuclides (yaml_cpp_class &data)

  Function to load the nuclide library into the pointer.
- void unloadNuclides ()

  Delete the pointer to nuclide library to free space.
- int registerInitialNuclide (std::string isotope_name)

  Register an initial nuclide by name (e.g., H-2)
- int registerInitialNuclide (std::string symb, int iso)

  Register an initial nuclide by symbol (e.g., H) and isotope number (e.g., 2)
- int registerInitialNuclide (int atom_num, int iso_num)

  Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)
- int registerInitialNuclide (std::string isotope_name, double ic)

  NOTE: The below functions will register isotopes with their initial conditions as well.
- int registerInitialNuclide (std::string symb, int iso, double ic)

  Register an initial nuclide by symbol (e.g., H) and iso number (e.g., 2)
- int registerInitialNuclide (int atom_num, int iso_num, double ic)

  Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)
- std::string fileFissionProductYields ()

  Function to read in the library and put into the Yaml object.
- void loadFissionProductYields (yaml_cpp_class &data)

  Function to load the fission product library into the Yaml object.
- void unloadFissionProductYields ()

  Function to delete the items in the Yaml object.
- void setFissionType (fiss_type opt)

  Set the fission type for the Fission Products.
- void setTotalMass (double mass)

  Set the total mass of fissible materials (kg)
- void setFissionExtent (double per)

Set the extent of fission parameter (%)
- void setEnergyLevel (double el)

  Set the energy level for neutron source (eV)
- void setThreshold (double val)

  Set the threshold value for half-life (in sec)
- void setUncertainty (int opt)

  Set the uncertainty option (-1,0,1)
- void timeSinceDetonation (double time, double per)

  Automatically sets the threshold value given time (in sec) after detonation and percent conversion rate (default 99%)
- void setTimeUnits (time_units units)

  Set the time units for the simulation.
- void setEndTime (double end)

  Set the end time for the simulation (in choosen units)
- void setTimeSteps (int steps)

  Set the number of time steps to simulate.
- void setVerifySoln (bool opt)

  Set boolean to verify eigen solution.
- void setPrintSparsity (bool opt)

  Set boolean to print sparsity on output.
- void setPrintChain (bool opt)

  Set boolean to print chain info.
- void setPrintResults (bool opt)

  Set boolean to print results of simulation.
- void setConsoleOut (bool opt)

  Set boolean to print output to console.
- double getTotalMoles ()

  Return total moles of weapon.
- double getFissionExtent ()

  Return the % of fission extent.
- int getUncertainty ()

  Return the uncertainty option.
- bool isConsoleOut ()

  Return boolean option for console output.
- std::vector< Isotope > & getWeaponMat ()

  Return reference to vector of isotopes in weapon.
- std::vector< double > & getWeaponFrac ()

  Return reference to vector of molefractions of weapon isotopes.
- int run_simulation (std::string file_name)

  Run a decay simulation from Fission Products.
- int print_yields ()

  Print yield data for weapon or fuel to output file.
- void addIsotopeMaterial (std::string iso, double percent)

  NOTE: For each of the below functions, you must first call the loadNuclides function.
- void addIsotopeMaterial (std::string sym, int iso_num, double percent)

  Add an isotope for the fissible material (by symbol and mass num)
- void addIsotopeMaterial (int atom_num, int iso_num, double percent)

  Add an isotope for the fissible material (by atom num and mass num)
- void checkFractions ()

  Check fractions of materials in fuel/weapone and correct if necessary.
- int evaluateYields ()

  Read yield data and set isotope fractionation based on yields.

- void evaluateEigenSolution ()

    Take the initialized yield info and form the eigenvectors.
- int verifyEigenSoln ()

    Function will verify that the eigenvectors and eigenvalues are correct.
- void calculateFractionation (double t)

    Function to calculate the isotope fractionation given a time t in seconds.
- double returnFractionation (std::string iso_name, double t)

    Return the fractionation of the given nuclide.
- int getNumberNuclides ()

    Return the number of nuclides in the decay chain.
- int getNumberStableNuclides ()

    Return the number of stable nuclides.
- int getIsotopeIndex (std::string iso_name)

    Return the unstable isotope index that corresponds to the given name.
- int getStableIsotopeIndex (std::string iso_name)

    Return the stable isotope index that corresponds to the given name.
- std::vector< int > & getParentList (int i)

    Return the vector list of parents for the ith isotope in the nuclide list.
- std::vector< int > & getStableParentList (int i)

    Return the vector list of parents for the ith stable isotope.
- std::vector< int > & getBranchList (int i, int j)

    Return the vector list of branch fractions for the jth parent of the ith nuclide.
- std::vector< int > & getStableBranchList (int i, int j)

    Return the list of branch fractions for the jth parent of the ith stable nuclide.
- Isotope & getIsotope (int i)

    Return the ith isotope in the nuclide list.
- Isotope & getStableIsotope (int i)

    Return the ith stable isotope.
- Isotope & getIsotope (std::string iso_name)

    Return the isotope (Stable or Unstable) that corresponds to the given name.

Protected Attributes

- fiss_type type

    Type of fission products to be produced.
- std::vector< Isotope > InitialMat

    Initial materials/isotopes to undergoe fission (Conc. in moles)
- std::vector< double > MatFrac

    Material fractionation of the initial material (%)
- std::map< int, double > Yields

    Map of fission yields of material based on atomic mass.
- double total_mass

    Total mass of the weapon or fuel rod (kg)
- double total_moles

    Total moles of fissionable materials (mol)
- double fiss_extent

    Percentage of the starting material that undergoes fission (%)
- double energy_level

    Energy level of neutron source (eV)
- int uncertainty

    Integer option (-1,0,1) to include uncertainty in calculation.
- yaml_cpp_class * fpy_data

    Yaml object to read and store the FPY library files.

Additional Inherited Members

5.19.1 Detailed Description

FissionProducts class object to create decay chains from fission yields.

This object inherits from DecayChain and will formulate decay chains based on starting weapon or fuel materials, as well as the type of fission taking place and the extent of fission for a given mass of the starting material. Fission Product Yields (FPYs) are based on the Evaluated Nuclear Data Format (ENDF) libraries, which contain FPYs for a number of fuel and weapon materials at various energy levels for two different types of fission: (i) neutron-induced and (ii) spontaneous.

5.19.2 Constructor & Destructor Documentation

5.19.2.1 FissionProducts()

```
FissionProducts::FissionProducts ( )
```

Default Constructor.

5.19.2.2 ∼FissionProducts()

```
FissionProducts::∼FissionProducts ( )
```

Default Destructor.

5.19.3 Member Function Documentation

5.19.3.1 DisplayInfo()

```
void FissionProducts::DisplayInfo ( )
```

Display the FAIRY information, initial materials and fractions.

5.19.3.2 DisplayMap()

```
void FissionProducts::DisplayMap ( )
```

Display the map for the DecayChain object.

5.19.3.3 loadNuclides()

```
void FissionProducts::loadNuclides (
             yaml_cpp_class & data )
```

Function to load the nuclide library into the pointer.

5.19.3.4 unloadNuclides()

```
void FissionProducts::unloadNuclides ( )
```

Delete the pointer to nuclide library to free space.

5.19.3.5 registerInitialNuclide() [1/6]

```
int FissionProducts::registerInitialNuclide (
             std::string isotope_name )
```

Register an initial nuclide by name (e.g., H-2)

5.19.3.6 registerInitialNuclide() [2/6]

```
int FissionProducts::registerInitialNuclide (
             std::string symb,
             int iso )
```

Register an initial nuclide by symbol (e.g., H) and isotope number (e.g., 2)

5.19.3.7 registerInitialNuclide() [3/6]

```
int FissionProducts::registerInitialNuclide (
             int atom_num,
             int iso_num )
```

Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)

5.19.3.8 registerInitialNuclide() [4/6]

```
int FissionProducts::registerInitialNuclide (
             std::string isotope_name,
             double ic )
```

NOTE: The below functions will register isotopes with their initial conditions as well.

Register an initial nuclide by name (e.g., H-2)

5.19.3.9 registerInitialNuclide() [5/6]

```
int FissionProducts::registerInitialNuclide (
             std::string symb,
             int iso,
             double ic )
```

Register an initial nuclide by symbol (e.g., H) and iso number (e.g., 2)

5.19.3.10 registerInitialNuclide() [6/6]

```
int FissionProducts::registerInitialNuclide (
             int atom_num,
             int iso_num,
             double ic )
```

Register an initial nuclide by atomic and mass numbers (e.g., H-2 = 1, 2)

5.19.3.11 fileFissionProductYields()

```
std::string FissionProducts::fileFissionProductYields ( )
```

Function to read in the library and put into the Yaml object.

5.19.3.12 loadFissionProductYields()

```
void FissionProducts::loadFissionProductYields (
             yaml_cpp_class & data )
```

Function to load the fission product library into the Yaml object.

5.19.3.13 unloadFissionProductYields()

```
void FissionProducts::unloadFissionProductYields ( )
```

Function to delete the items in the Yaml object.

5.19.3.14 setFissionType()

```
void FissionProducts::setFissionType (
             fiss_type opt )
```

Set the fission type for the Fission Products.

5.19.3.15 setTotalMass()

```
void FissionProducts::setTotalMass (
             double mass )
```

Set the total mass of fissible materials (kg)

5.19.3.16 setFissionExtent()

```
void FissionProducts::setFissionExtent (
             double per )
```

Set the extent of fission parameter (%)

5.19.3.17 setEnergyLevel()

```
void FissionProducts::setEnergyLevel (
             double el )
```

Set the energy level for neutron source (eV)

5.19.3.18 setThreshold()

```
void FissionProducts::setThreshold (
             double val )
```

Set the threshold value for half-life (in sec)

5.19.3.19 setUncertainty()

```
void FissionProducts::setUncertainty (
             int opt )
```

Set the uncertainty option (-1,0,1)

5.19.3.20 timeSinceDetonation()

```
void FissionProducts::timeSinceDetonation (
             double time,
             double per )
```

Automatically sets the threshold value given time (in sec) after detonation and percent conversion rate (default 99%)

5.19.3.21 setTimeUnits()

```
void FissionProducts::setTimeUnits (
            time_units units )
```

Set the time units for the simulation.

5.19.3.22 setEndTime()

```
void FissionProducts::setEndTime (
            double end )
```

Set the end time for the simulation (in choosen units)

5.19.3.23 setTimeSteps()

```
void FissionProducts::setTimeSteps (
            int steps )
```

Set the number of time steps to simulate.

5.19.3.24 setVerifySoln()

```
void FissionProducts::setVerifySoln (
            bool opt )
```

Set boolean to verify eigen solution.

5.19.3.25 setPrintSparsity()

```
void FissionProducts::setPrintSparsity (
            bool opt )
```

Set boolean to print sparsity on output.

5.19.3.26 setPrintChain()

```
void FissionProducts::setPrintChain (
            bool opt )
```

Set boolean to print chain info.

5.19.3.27 setPrintResults()

```
void FissionProducts::setPrintResults (
             bool opt )
```

Set boolean to print results of simulation.

5.19.3.28 setConsoleOut()

```
void FissionProducts::setConsoleOut (
             bool opt )
```

Set boolean to print output to console.

5.19.3.29 getTotalMoles()

```
double FissionProducts::getTotalMoles ( )
```

Return total moles of weapon.

5.19.3.30 getFissionExtent()

```
double FissionProducts::getFissionExtent ( )
```

Return the % of fission extent.

5.19.3.31 getUncertainty()

```
int FissionProducts::getUncertainty ( )
```

Return the uncertainty option.

5.19.3.32 isConsoleOut()

```
bool FissionProducts::isConsoleOut ( )
```

Return boolean option for console output.

5.19.3.33 getWeaponMat()

```
std::vector<Isotope>& FissionProducts::getWeaponMat ( )
```

Return reference to vector of isotopes in weapon.

5.19.3.34 getWeaponFrac()

```
std::vector<double>& FissionProducts::getWeaponFrac ( )
```

Return reference to vector of molefractions of weapon isotopes.

5.19.3.35 run_simulation()

```
int FissionProducts::run_simulation (
             std::string file_name )
```

Run a decay simulation from Fission Products.

5.19.3.36 print_yields()

```
int FissionProducts::print_yields ( )
```

Print yield data for weapon or fuel to output file.

5.19.3.37 addIsotopeMaterial() [1/3]

```
void FissionProducts::addIsotopeMaterial (
             std::string iso,
             double percent )
```

NOTE: For each of the below functions, you must first call the loadNuclides function.

Add an isotope for the fissible material (checks string value)

5.19.3.38 addIsotopeMaterial() [2/3]

```
void FissionProducts::addIsotopeMaterial (
             std::string sym,
             int iso_num,
             double percent )
```

Add an isotope for the fissible material (by symbol and mass num)

5.19.3.39 addIsotopeMaterial() [3/3]

```
void FissionProducts::addIsotopeMaterial (
             int atom_num,
             int iso_num,
             double percent )
```

Add an isotope for the fissible material (by atom num and mass num)

5.19.3.40 checkFractions()

```
void FissionProducts::checkFractions ( )
```

Check fractions of materials in fuel/weapone and correct if necessary.

5.19.3.41 evaluateYields()

```
int FissionProducts::evaluateYields ( )
```

Read yield data and set isotope fractionation based on yields.

5.19.3.42 evaluateEigenSolution()

```
void FissionProducts::evaluateEigenSolution ( )
```

Take the initialized yield info and form the eigenvectors.

5.19.3.43 verifyEigenSoln()

```
int FissionProducts::verifyEigenSoln ( )
```

Function will verify that the eigenvectors and eigenvalues are correct.

5.19.3.44 calculateFractionation()

```
void FissionProducts::calculateFractionation (
             double t )
```

Function to calculate the isotope fractionation given a time t in seconds.

This function must be called after createChains() and after formEigenvectors(). It will use the eigenvector solution to estimate the isotope concentrations for each isotope in the chain at the given time t. Those concentrations are based on values given for the initial concentrations of each isotope and are stored in each isotope object as the current concentration value.

Use an analytical solution based on linear combinations of eigenvectors.

5.19.3.45 returnFractionation()

```
double FissionProducts::returnFractionation (
             std::string iso_name,
             double t )
```

Return the fractionation of the given nuclide.

5.19.3.46 getNumberNuclides()

```
int FissionProducts::getNumberNuclides ( )
```

Return the number of nuclides in the decay chain.

5.19.3.47 getNumberStableNuclides()

```
int FissionProducts::getNumberStableNuclides ( )
```

Return the number of stable nuclides.

5.19.3.48 getIsotopeIndex()

```
int FissionProducts::getIsotopeIndex (
             std::string iso_name )
```

Return the unstable isotope index that corresponds to the given name.

5.19.3.49 getStableIsotopeIndex()

```
int FissionProducts::getStableIsotopeIndex (
             std::string iso_name )
```

Return the stable isotope index that corresponds to the given name.

5.19.3.50 getParentList()

```
std::vector<int>& FissionProducts::getParentList (
             int i )
```

Return the vector list of parents for the ith isotope in the nuclide list.

5.19.3.51 getStableParentList()

```
std::vector<int>& FissionProducts::getStableParentList (
            int i )
```

Return the vector list of parents for the ith stable isotope.

5.19.3.52 getBranchList()

```
std::vector<int>& FissionProducts::getBranchList (
            int i,
            int j )
```

Return the vector list of branch fractions for the jth parent of the ith nuclide.

5.19.3.53 getStableBranchList()

```
std::vector<int>& FissionProducts::getStableBranchList (
            int i,
            int j )
```

Return the list of branch fractions for the jth parent of the ith stable nuclide.

5.19.3.54 getIsotope() [1/2]

```
Isotope& FissionProducts::getIsotope (
            int i )
```

Return the ith isotope in the nuclide list.

5.19.3.55 getStableIsotope()

```
Isotope& FissionProducts::getStableIsotope (
            int i )
```

Return the ith stable isotope.

5.19.3.56 getIsotope() [2/2]

```
Isotope& FissionProducts::getIsotope (
            std::string iso_name )
```

Return the isotope (Stable or Unstable) that corresponds to the given name.

5.19.4 Member Data Documentation

5.19.4.1 type

```
fiss_type FissionProducts::type  [protected]
```

Type of fission products to be produced.

5.19.4.2 InitialMat

```
std::vector<Isotope> FissionProducts::InitialMat  [protected]
```

Initial materials/isotopes to undergoe fission (Conc. in moles)

5.19.4.3 MatFrac

```
std::vector<double> FissionProducts::MatFrac  [protected]
```

Material fractionation of the initial material (%)

5.19.4.4 Yields

```
std::map<int, double> FissionProducts::Yields  [protected]
```

Map of fission yields of material based on atomic mass.

5.19.4.5 total_mass

```
double FissionProducts::total_mass  [protected]
```

Total mass of the weapon or fuel rod (kg)

5.19.4.6 total_moles

```
double FissionProducts::total_moles  [protected]
```

Total moles of fissionable materials (mol)

5.19.4.7 fiss_extent

```
double FissionProducts::fiss_extent  [protected]
```

Percentage of the starting material that undergoes fission (%)

5.19.4.8 energy_level

```
double FissionProducts::energy_level  [protected]
```

Energy level of neutron source (eV)

5.19.4.9 uncertainty

```
int FissionProducts::uncertainty  [protected]
```

Integer option (-1,0,1) to include uncertainty in calculation.

5.19.4.10 fpy_data

```
yaml_cpp_class* FissionProducts::fpy_data  [protected]
```

Yaml object to read and store the FPY library files.

The documentation for this class was generated from the following file:

- fairy.h

5.20 GCR_DATA Struct Reference

Data structure for the implementation of the GCR algorithm for non-symmetric linear systems.

```
#include <lark.h>
```

Public Attributes

- int restart = -1

  Restart parameter for outer iterations - default = 20.
- int maxit = 0

  Maximum allowable outer iterations.
- int iter_outer = 0

  Number of outer iterations taken.
- int iter_inner = 0

  Number of inner iterations taken.
- int total_iter = 0

  Total number of iterations taken.
- bool breakdown = false

  Boolean to determine if a step has failed.
- double alpha

  Inner iteration step size.
- double beta

  Outer iteration step size.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolute tolerance for convergence - default = 1e-6.
- double res

  Absolute residual norm for linear system.
- double relres

  Relative residual norm for linear system.
- double relres_base

  Initial residual norm of the linear system.
- double bestres

  Best found residual norm of the linear system.
- bool Output = true

  True = print messages to the console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual Vector.
- Matrix< double > c_temp

  Temporary c vector to be updated.
- Matrix< double > u_temp

  Temporary u vector to be updated.
- std::vector< Matrix< double > > u

  Vector span for updating x.
- std::vector< Matrix< double > > c

  Vector span for updating r.
- OPTRANS_DATA transpose_dat

  Data structure for Operator Transposition.

5.20.1 Detailed Description

Data structure for the implementation of the GCR algorithm for non-symmetric linear systems.

C-style object used in conjunction with the Generalized Conjugate Residual (GCR) algorithm for solving a non-symmetric linear system of equations. When the linear system in question has a positive-definite-symmetric component to it, then this algorithm is equivalent to GMRESRP. However, it is generally less efficient than GMRESRP and can suffer breakdowns.

5.20.2 Member Data Documentation

5.20.2.1 restart

```
int GCR_DATA::restart = -1
```

Restart parameter for outer iterations - default = 20.

5.20.2.2 maxit

```
int GCR_DATA::maxit = 0
```

Maximum allowable outer iterations.

5.20.2.3 iter_outer

```
int GCR_DATA::iter_outer = 0
```

Number of outer iterations taken.

5.20.2.4 iter_inner

```
int GCR_DATA::iter_inner = 0
```

Number of inner iterations taken.

5.20.2.5 total_iter

```
int GCR_DATA::total_iter = 0
```

Total number of iterations taken.

5.20.2.6 breakdown

```
bool GCR_DATA::breakdown = false
```

Boolean to determine if a step has failed.

5.20.2.7 alpha

```
double GCR_DATA::alpha
```

Inner iteration step size.

5.20.2.8 beta

```
double GCR_DATA::beta
```

Outer iteration step size.

5.20.2.9 tol_rel

```
double GCR_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.20.2.10 tol_abs

```
double GCR_DATA::tol_abs = 1e-6
```

Absolute tolerance for convergence - default = 1e-6.

5.20.2.11 res

```
double GCR_DATA::res
```

Absolute residual norm for linear system.

5.20.2.12 relres

```
double GCR_DATA::relres
```

Relative residual norm for linear system.

5.20.2.13 relres_base

```
double GCR_DATA::relres_base
```

Initial residual norm of the linear system.

5.20.2.14 bestres

```
double GCR_DATA::bestres
```

Best found residual norm of the linear system.

5.20.2.15 Output

```
bool GCR_DATA::Output = true
```

True = print messages to the console.

5.20.2.16 x

```
Matrix<double> GCR_DATA::x
```

Current solution to the linear system.

5.20.2.17 bestx

```
Matrix<double> GCR_DATA::bestx
```

Best found solution to the linear system.

5.20.2.18 r

```
Matrix<double> GCR_DATA::r
```

Residual Vector.

5.20.2.19 c_temp

```
Matrix<double> GCR_DATA::c_temp
```

Temporary c vector to be updated.

5.20.2.20 u_temp

```
Matrix<double> GCR_DATA::u_temp
```

Temporary u vector to be updated.

5.20.2.21 u

```
std::vector<Matrix<double> > GCR_DATA::u
```

Vector span for updating x.

5.20.2.22 c

```
std::vector<Matrix<double> > GCR_DATA::c
```

Vector span for updating r.

5.20.2.23 transpose_dat

```
OPTRANS_DATA GCR_DATA::transpose_dat
```

Data structure for Operator Transposition.

The documentation for this struct was generated from the following file:

- lark.h

5.21 GMRESLP_DATA Struct Reference

Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning.

```
#include <lark.h>
```

Public Attributes

- int restart = -1

  Restart parameter - default = min(vector_size,20)
- int maxit = 0

  Maximum allowable iterations - default = min(vector_size,1000)
- int iter = 0

  Number of iterations needed for convergence.
- int steps = 0

  Total number of gmres iterations and krylov iterations.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolution tolerance for convergence - default = 1e-6.
- double res

  Absolution redisual norm of the linear system.
- double relres

  Relative residual norm of the linear system.
- double relres_base

  Initial residual norm of the linear system.
- double bestres

  Best found residual norm of the linear system.
- bool Output = true

  True = print messages to console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual vector for the linear system.
- ARNOLDI_DATA arnoldi_dat

  Data structure for the kyrlov subspace.

5.21.1 Detailed Description

Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning.

C-style object used in conjunction with Generalized Minimum RESidual Left-Precondtioned (GMRESLP) and Full Orthogonalization Method (FOM) algorithms to iteratively or directly solve a linear system of equations. When using with GMRESLP, you can only check/observe the linear residuals before a restart or after the Arnoldi space is constructed. This is because this object uses Left-side Preconditioning. A faster routine may be GMRESRP, which is able to construct residuals after each Arnoldi iteration.

5.21.2 Member Data Documentation

5.21.2.1 restart

```
int GMRESLP_DATA::restart = -1
```

Restart parameter - default = min(vector_size,20)

5.21.2.2 maxit

```
int GMRESLP_DATA::maxit = 0
```

Maximum allowable iterations - default = min(vector_size,1000)

5.21.2.3 iter

```
int GMRESLP_DATA::iter = 0
```

Number of iterations needed for convergence.

5.21.2.4 steps

```
int GMRESLP_DATA::steps = 0
```

Total number of gmres iterations and krylov iterations.

5.21.2.5 tol_rel

```
double GMRESLP_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.21.2.6 tol_abs

```
double GMRESLP_DATA::tol_abs = 1e-6
```

Absolution tolerance for convergence - default = 1e-6.

5.21.2.7 res

```
double GMRESLP_DATA::res
```

Absolution redisual norm of the linear system.

5.21.2.8 relres

```
double GMRESLP_DATA::relres
```

Relative residual norm of the linear system.

5.21.2.9 relres_base

```
double GMRESLP_DATA::relres_base
```

Initial residual norm of the linear system.

5.21.2.10 bestres

```
double GMRESLP_DATA::bestres
```

Best found residual norm of the linear system.

5.21.2.11 Output

```
bool GMRESLP_DATA::Output = true
```

True = print messages to console.

5.21.2.12 x

```
Matrix<double> GMRESLP_DATA::x
```

Current solution to the linear system.

5.21.2.13 bestx

```
Matrix<double> GMRESLP_DATA::bestx
```

Best found solution to the linear system.

5.21.2.14 r

```
Matrix<double> GMRESLP_DATA::r
```

Residual vector for the linear system.

5.21.2.15 arnoldi_dat

```
ARNOLDI_DATA GMRESLP_DATA::arnoldi_dat
```

Data structure for the kyrlov subspace.

The documentation for this struct was generated from the following file:

- lark.h

5.22 GMRESR_DATA Struct Reference

Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GMRESR)

```
#include <lark.h>
```

Public Attributes

- int gcr_restart = -1
  Number of GCR restarts (default = 20, max = N)
- int gcr_maxit = 0
  Number of GCR iterations.
- int gmres_restart = -1
  Number of GMRES restarts (max = 20)
- int gmres_maxit = 1
  Number of GMRES iterations (max = 5, default = 1)
- int N
  Dimension of the linear system.
- int total_iter
  Total GMRES and GCR iterations.
- int iter_outer
  Total GCR iterations.
- int iter_inner
  Total GMRES iterations.
- bool GCR_Output = true
  True = print GCR messages.
- bool GMRES_Output = false
  True = print GMRES messages.
- double gmres_tol = 0.1
  Tolerance relative to GCR iterations.
- double gcr_rel_tol = 1e-6
  Relative outer residual tolerance.
- double gcr_abs_tol = 1e-6
  Absolute outer residual tolerance.
- Matrix< double > arg
  Argument matrix passed between preconditioner and iterator.
- GCR_DATA gcr_dat
  Data structure for the outer GCR steps.
- GMRESRP_DATA gmres_dat
  Data structure for the inner GMRES steps.

- int(∗ matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void ∗matvec_data)

    User supplied matrix-vector product function.
- int(∗ terminal_precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗precon_data)

    Optional user supplied terminal preconditioner.
- const void ∗ matvec_data

    Data structure for the user's matvec function.
- const void ∗ term_precon

    Data structure for the user's terminal preconditioner.

5.22.1 Detailed Description

Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GMRESR)

C-style object to be used in conjunction with the Generalized Minimum RESidual Recurive (GMRESR) algorithm. Although the name suggests that this method used GMRES recursively, what it is actually doing is nesting GMRE←SRP iterations inside the GCR method to form a preconditioner for GCR. The name GMRESR came from literature (Vorst and Vuik, "GMRESR: A family of nested GMRES methods", 1991).

5.22.2 Member Data Documentation

5.22.2.1 gcr_restart

```
int GMRESR_DATA::gcr_restart = -1
```

Number of GCR restarts (default = 20, max = N)

5.22.2.2 gcr_maxit

```
int GMRESR_DATA::gcr_maxit = 0
```

Number of GCR iterations.

5.22.2.3 gmres_restart

```
int GMRESR_DATA::gmres_restart = -1
```

Number of GMRES restarts (max = 20)

5.22.2.4 gmres_maxit

```
int GMRESR_DATA::gmres_maxit = 1
```

Number of GMRES iterations (max = 5, default = 1)

5.22.2.5 N

```
int GMRESR_DATA::N
```

Dimension of the linear system.

5.22.2.6 total_iter

```
int GMRESR_DATA::total_iter
```

Total GMRES and GCR iterations.

5.22.2.7 iter_outer

```
int GMRESR_DATA::iter_outer
```

Total GCR iterations.

5.22.2.8 iter_inner

```
int GMRESR_DATA::iter_inner
```

Total GMRES iterations.

5.22.2.9 GCR_Output

```
bool GMRESR_DATA::GCR_Output = true
```

True = print GCR messages.

5.22.2.10 GMRES_Output

```
bool GMRESR_DATA::GMRES_Output = false
```

True = print GMRES messages.

5.22.2.11 gmres_tol

```
double GMRESR_DATA::gmres_tol = 0.1
```

Tolerance relative to GCR iterations.

5.22.2.12 gcr_rel_tol

```
double GMRESR_DATA::gcr_rel_tol = 1e-6
```

Relative outer residual tolerance.

5.22.2.13 gcr_abs_tol

```
double GMRESR_DATA::gcr_abs_tol = 1e-6
```

Absolute outer residual tolerance.

5.22.2.14 arg

```
Matrix<double> GMRESR_DATA::arg
```

Argument matrix passed between preconditioner and iterator.

5.22.2.15 gcr_dat

```
GCR_DATA GMRESR_DATA::gcr_dat
```

Data structure for the outer GCR steps.

5.22.2.16 gmres_dat

```
GMRESRP_DATA GMRESR_DATA::gmres_dat
```

Data structure for the inner GMRES steps.

5.22.2.17 matvec

```
int(* GMRESR_DATA::matvec) (const Matrix< double > &x, Matrix< double > &Ax, const void *matvec↩
_data)
```

User supplied matrix-vector product function.

5.22.2.18 terminal_precon

```
int(* GMRESR_DATA::terminal_precon) (const Matrix< double > &r, Matrix< double > &p, const
void *precon_data)
```

Optional user supplied terminal preconditioner.

5.22.2.19 matvec_data

```
const void* GMRESR_DATA::matvec_data
```

Data structure for the user's matvec function.

5.22.2.20 term_precon

```
const void* GMRESR_DATA::term_precon
```

Data structure for the user's terminal preconditioner.

The documentation for this struct was generated from the following file:

- lark.h

5.23 GMRESRP_DATA Struct Reference

Data structure for the Restarted GMRES algorithm with Right Preconditioning.

```
#include <lark.h>
```

Public Attributes

- int restart = -1

  Restart parameter - default = min(20,vector_size)
- int maxit = 0

  Maximum allowable outer iterations.
- int iter_outer = 0

  Total number of outer iterations.
- int iter_inner = 0

  Total number of inner iterations.
- int iter_total = 0

  Total number of overall iterations.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolute tolerance for convergence - default = 1e-6.
- double res

  Absolute residual norm for linear system.
- double relres

  Relative residual norm for linear system.
- double relres_base

  Initial residual norm of the linear system.
- double bestres

  Best found residual norm of the linear system.
- bool Output = true

  True = print messages to console.

- Matrix< double > x

    Current solution to the linear system.
- Matrix< double > bestx

    Best found solution to the linear system.
- Matrix< double > r

    Residual vector for the linear system.
- std::vector< Matrix< double > > Vk

    (N x k) orthonormal vector basis
- std::vector< Matrix< double > > Zk

    (N x k) preconditioned vector set
- std::vector< std::vector< double > > H

    (k+1 x k) upper Hessenberg storage matrix
- std::vector< std::vector< double > > H_bar

    (k+1 x k) Factorized matrix
- std::vector< double > y

    (k x 1) Vector search direction
- std::vector< double > e0

    (k+1 x 1) Normalized vector with residual info
- std::vector< double > e0_bar

    (k+1 x 1) Factorized normal vector
- Matrix< double > w

    (N) x (1) interim result of the matrix_vector multiplication
- Matrix< double > v

    (N) x (1) holding cell for the column entries of Vk and other interims
- Matrix< double > sum

    (N) x (1) running sum of subspace vectors for use in altering w

5.23.1 Detailed Description

Data structure for the Restarted GMRES algorithm with Right Preconditioning.

C-style object used in conjunction with Generalized Minimum RESidual Right Preconditioned (GMRESRP) algorithm to iteratively solve a linear system of equations. Unlike GMRESLP, the GMRESRP method is capable of checking linear residuals at both the inner and outer steps. As a result, this algorithm may terminate earlier than GMRESLP if it has found a suitable solution during one of the inner steps.

5.23.2 Member Data Documentation

5.23.2.1 restart

```
int GMRESRP_DATA::restart = -1
```

Restart parameter - default = min(20,vector_size)

5.23.2.2 maxit

```
int GMRESRP_DATA::maxit = 0
```

Maximum allowable outer iterations.

5.23.2.3 iter_outer

```
int GMRESRP_DATA::iter_outer = 0
```

Total number of outer iterations.

5.23.2.4 iter_inner

```
int GMRESRP_DATA::iter_inner = 0
```

Total number of inner iterations.

5.23.2.5 iter_total

```
int GMRESRP_DATA::iter_total = 0
```

Total number of overall iterations.

5.23.2.6 tol_rel

```
double GMRESRP_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.23.2.7 tol_abs

```
double GMRESRP_DATA::tol_abs = 1e-6
```

Absolute tolerance for convergence - default = 1e-6.

5.23.2.8 res

```
double GMRESRP_DATA::res
```

Absolute residual norm for linear system.

5.23.2.9 relres

```
double GMRESRP_DATA::relres
```

Relative residual norm for linear system.

5.23.2.10 relres_base

```
double GMRESRP_DATA::relres_base
```

Initial residual norm of the linear system.

5.23.2.11 bestres

```
double GMRESRP_DATA::bestres
```

Best found residual norm of the linear system.

5.23.2.12 Output

```
bool GMRESRP_DATA::Output = true
```

True = print messages to console.

5.23.2.13 x

```
Matrix<double> GMRESRP_DATA::x
```

Current solution to the linear system.

5.23.2.14 bestx

```
Matrix<double> GMRESRP_DATA::bestx
```

Best found solution to the linear system.

5.23.2.15 r

```
Matrix<double> GMRESRP_DATA::r
```

Residual vector for the linear system.

5.23.2.16 Vk

`std::vector< Matrix<double> > GMRESRP_DATA::Vk`

(N x k) orthonormal vector basis

5.23.2.17 Zk

`std::vector< Matrix<double> > GMRESRP_DATA::Zk`

(N x k) preconditioned vector set

5.23.2.18 H

`std::vector< std::vector< double > > GMRESRP_DATA::H`

(k+1 x k) upper Hessenberg storage matrix

5.23.2.19 H_bar

`std::vector< std::vector< double > > GMRESRP_DATA::H_bar`

(k+1 x k) Factorized matrix

5.23.2.20 y

`std::vector< double > GMRESRP_DATA::y`

(k x 1) Vector search direction

5.23.2.21 e0

`std::vector< double > GMRESRP_DATA::e0`

(k+1 x 1) Normalized vector with residual info

5.23.2.22 e0_bar

`std::vector< double > GMRESRP_DATA::e0_bar`

(k+1 x 1) Factorized normal vector

5.23.2.23 w

```
Matrix<double> GMRESRP_DATA::w
```

(N) x (1) interim result of the matrix_vector multiplication

5.23.2.24 v

```
Matrix<double> GMRESRP_DATA::v
```

(N) x (1) holding cell for the column entries of Vk and other interims

5.23.2.25 sum

```
Matrix<double> GMRESRP_DATA::sum
```

(N) x (1) running sum of subspace vectors for use in altering w

The documentation for this struct was generated from the following file:

- lark.h

5.24 GPAST_DATA Struct Reference

GPAST Data Structure.

```
#include <magpie.h>
```

Public Attributes

- double x

  Adsorbed mole fraction.
- double y

  Gas phase mole fraction.
- double He

  Henry's Coefficient (mol/kg/kPa)
- double q

  Amount adsorbed for each component (mol/kg)
- std::vector< double > gama_inf

  Infinite dilution activities.
- double qo

  Pure component capacities (mol/kg)
- double PIo

  Pure component spreading pressures (mol/kg)
- std::vector< double > po

  Pure component reference state pressures (kPa)
- double poi

  Reference state pressures solved for using Recover eval GPAST.
- bool present

  If true, then the component is present; if false, then the component is not present.

5.24.1 Detailed Description

GPAST Data Structure.

C-style object holding all parameter information associated with the Generalized Predictive Adsorbed Solution Theory (GPAST) system of equations. Each species in the gas phase will have one of these objects.

5.24.2 Member Data Documentation

5.24.2.1 x

```
double GPAST_DATA::x
```

Adsorbed mole fraction.

5.24.2.2 y

```
double GPAST_DATA::y
```

Gas phase mole fraction.

5.24.2.3 He

```
double GPAST_DATA::He
```

Henry's Coefficient (mol/kg/kPa)

5.24.2.4 q

```
double GPAST_DATA::q
```

Amount adsorbed for each component (mol/kg)

5.24.2.5 gama_inf

```
std::vector<double> GPAST_DATA::gama_inf
```

Infinite dilution activities.

5.24.2.6 qo

```
double GPAST_DATA::qo
```

Pure component capacities (mol/kg)

5.24.2.7 PIo

```
double GPAST_DATA::PIo
```

Pure component spreading pressures (mol/kg)

5.24.2.8 po

```
std::vector<double> GPAST_DATA::po
```

Pure component reference state pressures (kPa)

5.24.2.9 poi

```
double GPAST_DATA::poi
```

Reference state pressures solved for using Recover eval GPAST.

5.24.2.10 present

```
bool GPAST_DATA::present
```

If true, then the component is present; if false, then the component is not present.

The documentation for this struct was generated from the following file:

- magpie.h

5.25 GSTA_DATA Struct Reference

GSTA Data Structure.

```
#include <magpie.h>
```

Public Attributes

- double qmax
  Theoretical maximum capacity of adsorbate-adsorbent pair (mol/kg)
- int m
  Number of parameters in the GSTA isotherm.
- std::vector< double > dHo
  Enthalpies for each site (J/mol)
- std::vector< double > dSo
  Entropies for each site (J/(K∗mol))

5.25.1 Detailed Description

GSTA Data Structure.

C-style object holding all parameter information associated with the Generalized Statistical Thermodynamic Adsorption (GSTA) isotherm model. Each species in the gas phase will have one of these objects.

5.25.2 Member Data Documentation

5.25.2.1 qmax

```
double GSTA_DATA::qmax
```

Theoretical maximum capacity of adsorbate-adsorbent pair (mol/kg)

5.25.2.2 m

```
int GSTA_DATA::m
```

Number of parameters in the GSTA isotherm.

5.25.2.3 dHo

```
std::vector<double> GSTA_DATA::dHo
```

Enthalpies for each site (J/mol)

5.25.2.4 dSo

```
std::vector<double> GSTA_DATA::dSo
```

Entropies for each site (J/(K∗mol))

The documentation for this struct was generated from the following file:

- magpie.h

5.26 GSTA_OPT_DATA Struct Reference

Data structure used in the GSTA optimization routines.

```
#include <gsta_opt.h>
```

Public Attributes

- int total_eval

  Keeps track of the total number of function evaluations.
- int n_par

  Number of parameters being optimized for.
- double qmax

  Maximum theoretical adsorption capacity (M/M) (0 if unknown)
- int iso

  Keeps isotherm that is currently being optimized.
- std::vector< std::vector< double > > Fobj

  Creates a dynamic array to store all Fobj values.
- std::vector< std::vector< double > > q
- std::vector< std::vector< double > > P

  Creates a dynamic array for q and P data pairs.
- std::vector< std::vector< double > > best_par

  Used to store the values of the parameters of best fit.
- std::vector< std::vector< double > > Kno

  Dimensionless parameters determined from best_par.
- std::vector< std::vector< std::vector< double > > > all_pars

  Used to create a ragged array of all parameters.
- std::vector< std::vector< double > > norms

  Used to store the values of all the calculated norms.
- std::vector< double > opt_qmax

  If qmax is unknown, this vector holds it's optimized values.

5.26.1 Detailed Description

Data structure used in the GSTA optimization routines.

C-style structure that keeps track of all infomation during the optimization routine. All solutions and parameters to the GSTA isotherm are held in order to find the best solution with the fewest parameters.

5.26.2 Member Data Documentation

5.26.2.1 total_eval

```
int GSTA_OPT_DATA::total_eval
```

Keeps track of the total number of function evaluations.

5.26.2.2 n_par

```
int GSTA_OPT_DATA::n_par
```

Number of parameters being optimized for.

5.26.2.3 qmax

```
double GSTA_OPT_DATA::qmax
```

Maximum theoretical adsorption capacity (M/M) (0 if unknown)

5.26.2.4 iso

```
int GSTA_OPT_DATA::iso
```

Keeps isotherm that is currently being optimized.

5.26.2.5 Fobj

```
std::vector<std::vector<double> > GSTA_OPT_DATA::Fobj
```

Creates a dynamic array to store all Fobj values.

5.26.2.6 q

```
std::vector<std::vector<double> > GSTA_OPT_DATA::q
```

5.26.2.7 P

```
std::vector<std::vector<double> > GSTA_OPT_DATA::P
```

Creates a dynamic array for q and P data pairs.

5.26.2.8 best_par

```
std::vector<std::vector<double> > GSTA_OPT_DATA::best_par
```

Used to store the values of the parameters of best fit.

5.26.2.9 Kno

```
std::vector<std::vector<double> > GSTA_OPT_DATA::Kno
```

Dimensionless parameters determined from best_par.

5.26.2.10 all_pars

```
std::vector<std::vector<std::vector<double> > > GSTA_OPT_DATA::all_pars
```

Used to create a ragged array of all parameters.

5.26.2.11 norms

```
std::vector<std::vector<double> > GSTA_OPT_DATA::norms
```

Used to store the values of all the calculated norms.

5.26.2.12 opt_qmax

```
std::vector<double> GSTA_OPT_DATA::opt_qmax
```

If qmax is unknown, this vector holds it's optimized values.

The documentation for this struct was generated from the following file:

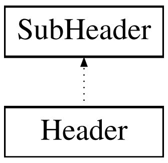
- gsta_opt.h

5.27 Header Class Reference

Object for headers in a yaml document (inherits from SubHeader)

```
#include <yaml_wrapper.h>
```

Inheritance diagram for Header:



Public Member Functions

- Header ()

  Default Constructor.
- ∼Header ()

  Default Destructor.
- Header (const Header &head)

  Copy constructor.
- Header (std::string name)

  Constructor by header name.
- Header (const KeyValueMap &map)

  Constructor by existing map.
- Header (std::string name, const KeyValueMap &map)

  Constructor by name and map.
- Header (std::string key, const SubHeader &sub)

  Constructor by single subheader object.
- Header & operator= (const Header &head)

  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)

  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const

  Return the ValueType at the given key.
- SubHeader & operator() (const std::string key)

  Return the SubHeader reference at the given key.
- SubHeader operator() (const std::string key) const

  Return the SubHeader at the given key.
- std::map< std::string, SubHeader > & getSubMap ()

  Return the reference to the SubHeader Map.
- KeyValueMap & getDataMap ()

  Return the reference to the KeyValueMap.
- SubHeader & getSubHeader (std::string key)

  Return the subheader at the given key.
- std::map< std::string, SubHeader >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, SubHeader >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, SubHeader >::const_iterator begin () const

Returns a const iterator pointing to the begining of the list.

- std::map< std::string, SubHeader >::iterator begin ()

  Returns an iterator pointing to the begining of the list.
- void clear ()

  Clear out the SubMap, KeyValueMap, and other info.
- void resetKeys ()

  Reset the keys of the SubMap to the names of each SubHeader.
- void changeKey (std::string oldKey, std::string newKey)

  Change one of the keys in the map.
- void addPair (std::string key, std::string val)

  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int t)

  Adds a pair object and asserts a type.
- void setName (std::string name)

  Set the name of the Header.
- void setAlias (std::string alias)

  Set the alias of the header, if any.
- void setNameAliasPair (std::string n, std::string a, int s)

  Set the name, alias, and state for the header.
- void setState (int state)

  Set the state of the header, if any.
- void DisplayContents ()

  Display the contents of the header object.
- void addSubKey (std::string key)

  Adds a key to the SubHeader Map.
- void copyAnchor2Alias (std::string alias, SubHeader &ref)

  Find the anchor in the map, and copy to the Header reference given.
- int size ()

  Return the size of the Sub_Map.
- std::string getName ()

  Return the name of the header.
- std::string getAlias ()

  Return the alias of the header.
- int getState ()

  Return the state of the header.
- bool isAlias ()

  Returns true if the header is an alias.
- bool isAnchor ()

  Returns true if the header is an anchor.
- SubHeader & getAnchoredSub (std::string alias)

  Returns reference to the anchored subheader, if any.

Private Attributes

- std::map< std::string, SubHeader > Sub_Map

  Map of the contained subheaders in the main header.

Additional Inherited Members

5.27.1 Detailed Description

Object for headers in a yaml document (inherits from SubHeader)

C++ Object for headers in a yaml document that is built from the SubHeader object already created. The chain of inheritance works in this direction because a Header can have both a map of SubHeaders and a map of KeyValue↩ Pairs. Therefore, the SubHeader object is actually the more generic form of a header.

Since this object inherits from SubHeader, it has access to all it's protected members, including the alias, state, name, and KeyValueMap. Operator overloads and other functions are provided to allow the user to query both the KeyValueMap and SubHeader for specific information. The names of the SubHeaders are also used as it's keys. Make sure all SubHeader keys are unique to this header.

5.27.2 Constructor & Destructor Documentation

5.27.2.1 Header() [1/6]

```
Header::Header ( )
```

Default Constructor.

5.27.2.2 ∼Header()

```
Header::∼Header ( )
```

Default Destructor.

5.27.2.3 Header() [2/6]

```
Header::Header (
            const Header & head )
```

Copy constructor.

5.27.2.4 Header() [3/6]

```
Header::Header (
            std::string name )
```

Constructor by header name.

5.27.2.5 Header() [4/6]

```
Header::Header (
            const KeyValueMap & map )
```

Constructor by existing map.

5.27.2.6 Header() [5/6]

```
Header::Header (
            std::string name,
            const KeyValueMap & map )
```

Constructor by name and map.

5.27.2.7 Header() [6/6]

```
Header::Header (
            std::string key,
            const SubHeader & sub )
```

Constructor by single subheader object.

5.27.3 Member Function Documentation

5.27.3.1 operator=()

```
Header& Header::operator= (
            const Header & head )
```

Equals overload.

5.27.3.2 operator[]() [1/2]

```
ValueTypePair& Header::operator[] (
            const std::string key )
```

Return the ValueType reference at the given key.

5.27.3.3 operator[]() [2/2]

```
ValueTypePair Header::operator[] (
            const std::string key ) const
```

Return the ValueType at the given key.

5.27.3.4 operator()() [1/2]

```
SubHeader& Header::operator() (
            const std::string key )
```

Return the SubHeader reference at the given key.

5.27.3.5 operator()() [2/2]

```
SubHeader Header::operator() (
            const std::string key ) const
```

Return the SubHeader at the given key.

5.27.3.6 getSubMap()

```
std::map<std::string, SubHeader>& Header::getSubMap ( )
```

Return the reference to the SubHeader Map.

5.27.3.7 getDataMap()

```
KeyValueMap& Header::getDataMap ( )
```

Return the reference to the KeyValueMap.

5.27.3.8 getSubHeader()

```
SubHeader& Header::getSubHeader (
            std::string key )
```

Return the subheader at the given key.

5.27.3.9 end() [1/2]

```
std::map<std::string, SubHeader>::const_iterator Header::end ( ) const
```

Returns a const iterator pointing to the end of the list.

5.27.3.10 end() [2/2]

```
std::map<std::string, SubHeader>::iterator Header::end ( )
```

Returns an iterator pointing to the end of the list.

5.27.3.11 begin() [1/2]

```
std::map<std::string, SubHeader>::const_iterator Header::begin ( ) const
```

Returns a const iterator pointing to the begining of the list.

5.27.3.12 begin() [2/2]

```
std::map<std::string, SubHeader>::iterator Header::begin ( )
```

Returns an iterator pointing to the begining of the list.

5.27.3.13 clear()

```
void Header::clear ( )
```

Clear out the SubMap, KeyValueMap, and other info.

5.27.3.14 resetKeys()

```
void Header::resetKeys ( )
```

Reset the keys of the SubMap to the names of each SubHeader.

5.27.3.15 changeKey()

```
void Header::changeKey (
             std::string oldKey,
             std::string newKey )
```

Change one of the keys in the map.

5.27.3.16 addPair() [1/2]

```
void Header::addPair (
             std::string key,
             std::string val )
```

Adds a pair object to the map (with only strings)

5.27.3.17 addPair() [2/2]

```
void Header::addPair (
             std::string key,
             std::string val,
             int t )
```

Adds a pair object and asserts a type.

5.27.3.18 setName()

```
void Header::setName (
             std::string name )
```

Set the name of the Header.

5.27.3.19 setAlias()

```
void Header::setAlias (
             std::string alias )
```

Set the alias of the header, if any.

5.27.3.20 setNameAliasPair()

```
void Header::setNameAliasPair (
             std::string n,
             std::string a,
             int s )
```

Set the name, alias, and state for the header.

5.27.3.21 setState()

```
void Header::setState (
             int state )
```

Set the state of the header, if any.

5.27.3.22 DisplayContents()

```
void Header::DisplayContents ( )
```

Display the contents of the header object.

5.27.3.23 addSubKey()

```
void Header::addSubKey (
             std::string key )
```

Adds a key to the SubHeader Map.

5.27.3.24 copyAnchor2Alias()

```
void Header::copyAnchor2Alias (
             std::string alias,
             SubHeader & ref )
```

Find the anchor in the map, and copy to the Header reference given.

5.27.3.25 size()

```
int Header::size ( )
```

Return the size of the Sub_Map.

5.27.3.26 getName()

```
std::string Header::getName ( )
```

Return the name of the header.

5.27.3.27 getAlias()

```
std::string Header::getAlias ( )
```

Return the alias of the header.

5.27.3.28 getState()

```
int Header::getState ( )
```

Return the state of the header.

5.27.3.29 isAlias()

```
bool Header::isAlias ( )
```

Returns true if the header is an alias.

5.27.3.30 isAnchor()

```
bool Header::isAnchor ( )
```

Returns true if the header is an anchor.

5.27.3.31 getAnchoredSub()

```
SubHeader& Header::getAnchoredSub (
             std::string alias )
```

Returns reference to the anchored subheader, if any.

5.27.4 Member Data Documentation

5.27.4.1 Sub_Map

```
std::map<std::string, SubHeader> Header::Sub_Map  [private]
```

Map of the contained subheaders in the main header.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.28 InfiniteBath Class Reference

InfiniteBath is a class object associated with residuals stemming from an infinite concentration of a species.

```
#include <crow.h>
```

Public Member Functions

- InfiniteBath ()

  Default constructor.
- ∼InfiniteBath ()

  Default destructor.
- void InitializeSolver (Dove &Solver)

  Function to initialize the InfiniteBath object from the Dove object.
- void SetIndex (int index)

  Function to set the index of the primary variable species.
- void SetValue (double val)

  Set the value of the infinite bath parameter.
- void InsertWeight (int i, double w)

  Insert the weight for the indicated species.
- double getValue ()

  Return the value of the infinite bath.
- int getIndex ()

  Function to return the primary variable index.
- std::map< int, double > & getWeightMap ()

  Function to return reference to the weight map object.

Protected Attributes

- double Value

  Value that the variable or set of variables is being held to.
- int main_index

  Variable index for the variable of interest.
- std::map< int, double > weights

  Map of weight coefficients for the infinite bath.
- Dove ∗ SolverInfo

  Pointer to the Dove Object.

5.28.1 Detailed Description

InfiniteBath is a class object associated with residuals stemming from an infinite concentration of a species.

This is a C++ style object designed to store and operate on information associated with material balances associated with the infinite bath representation of materials. In an infinite bath, the concentration level of a certain species, or set of species, is held to a specific constant value and cannot change in time. Thus, the system behaves as if there is an infinite amount of the material in question available to react with other species in the system. As such, this object's rate function will be a simple linear equation and will be a steady- state function when incorporated into the DOVE solver.

5.28.2 Constructor & Destructor Documentation

5.28.2.1 InfiniteBath()

```
InfiniteBath::InfiniteBath ( )
```

Default constructor.

5.28.2.2 ∼InfiniteBath()

```
InfiniteBath::∼InfiniteBath ( )
```

Default destructor.

5.28.3 Member Function Documentation

5.28.3.1 InitializeSolver()

```
void InfiniteBath::InitializeSolver (
             Dove & Solver )
```

Function to initialize the InfiniteBath object from the Dove object.

5.28.3.2 SetIndex()

```
void InfiniteBath::SetIndex (
             int index )
```

Function to set the index of the primary variable species.

5.28.3.3 SetValue()

```
void InfiniteBath::SetValue (
             double val )
```

Set the value of the infinite bath parameter.

5.28.3.4 InsertWeight()

```
void InfiniteBath::InsertWeight (
             int i,
             double w )
```

Insert the weight for the indicated species.

5.28.3.5 getValue()

```
double InfiniteBath::getValue ( )
```

Return the value of the infinite bath.

5.28.3.6 getIndex()

```
int InfiniteBath::getIndex ( )
```

Function to return the primary variable index.

5.28.3.7 getWeightMap()

```
std::map<int, double>& InfiniteBath::getWeightMap ( )
```

Function to return reference to the weight map object.

5.28.4 Member Data Documentation

5.28.4.1 Value

```
double InfiniteBath::Value  [protected]
```

Value that the variable or set of variables is being held to.

5.28.4.2 main_index

```
int InfiniteBath::main_index  [protected]
```

Variable index for the variable of interest.

5.28.4.3 weights

```
std::map<int, double> InfiniteBath::weights  [protected]
```

Map of weight coefficients for the infinite bath.

5.28.4.4 SolverInfo

```
Dove* InfiniteBath::SolverInfo [protected]
```

Pointer to the Dove Object.

The documentation for this class was generated from the following file:

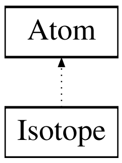
- crow.h

5.29 Isotope Class Reference

Isotope object to hold information and provide decay operations.

```
#include <ibis.h>
```

Inheritance diagram for Isotope:



Public Member Functions

- Isotope ()

  Default constructor.
- ∼Isotope ()

  Default destructor.
- void loadNuclides (yaml_cpp_class &data)

  Function to load the nuclide library into the pointer.
- void unloadNuclides ()

  Delete the pointer to nuclide library to free space.
- void clearChain ()

  Delete the chain for this isotope to free space.
- int registerIsotope (std::string isotope_name)

  Register an isotope given the isotope name (e.g., H-2)
- int registerIsotope (std::string symbol, int iso)

  Register an isotope given an atomic symbol (e.g., H) and isotope number (e.g., 2)
- int registerIsotope (int atom_num, int iso_num)

  Register an isotope given both an atomic and isotope number (e.g., H-2 = 1, 2)
- void DisplayInfo ()

  Print out isotope information to the console.
- void DisplayChain ()

  Print out chain information to the console.
- void createChain ()

  Function to create and fill in the chain of nuclides for this starting isotope.
- void setInitialCondition (double ic)

  Set the value for the initial condition of this nuclide.

- void setConcentration (double c)

  Set the concentration value for the nuclide.
- void setActivity (double a)

  Set the activity value for the nuclide.
- void setUnits2Moles (bool opt)

  Set concentration units to moles if True is passed.
- void setWarnings (bool opt)

  Set the warnings boolean value.
- void setThreshold (double val)

  Set the threshold value for half-life (in sec)
- void updateDecayRate ()

  Increase decay rate by 1% (used to correct issues with same eigenvalues)
- int calculateIonization (std::vector< Atom > &atoms, std::vector< double > &mass_fracs, double density, double potential)

  Calculate the ionization coefficient given a list of atoms and their mass fractions in a particular media.
- int IsotopeNumber ()

  Return the isotope number of the atom (i.e., mass number)
- double DecayRate ()

  Return the decay rate of the isotope.
- double HalfLife (time_units units)

  Return the half-life in the given units.
- time_units HalfLifeUnits ()

  Return the half-life units.
- std::string IsotopeName ()

  Return the name of the isotope.
- bool isStable ()

  Return stability condition.
- bool isIsomericState ()

  Return isomeric condition.
- bool isMolar ()

  Returns True if concentration units are moles.
- int DecayModes ()

  Return the number of decay modes.
- double getInitialCondition ()

  Return the value of the initial condition.
- double getConcentration ()

  Return the concentration value of the nuclide.
- double getActivity ()

  Return the activity value of the nuclide.
- double AtomicWeight ()

  Return the atomic weight of the isotope.
- double MeltingPoint ()

  Returns the melting point.
- double BoilingPoint ()

  Returns the boiling point.
- double ThermalXSection ()

  Returns the thermal cross section.
- double ScatterXSection ()

  Returns the scattering cross section.
- double MassExcess ()

  Returns the mass-excess of the nuclide (in MeV)

- double Jpi ()

  Returns the magnitude of the spin-parity.
- double IonizationCoeff ()

  Returns the average number of ion pairs produced from this nuclide decay.
- decay_mode DecayMode (int i)

  Return the ith decay mode.
- double BranchFraction (int i)

  Return the ith branch fraction.
- std::string ParticleEmitted (int i)

  Return the name of the particle emitted for the ith decay mode.
- int NumberParticlesEmitted (int i)

  Return the number of particles that get emitted.
- std::string Daughter (int i)

  Return the name of the daughter isotope.
- double QValue (int i)

  Return the ith Q-value (in MeV)
- int deltaJ (int i)

  Return the ith spin jump.
- double MeanEnergy (int i)

  Return the ith mean radiation energy (in MeV)
- std::vector< int > DaughterIndices (std::string parent)

  Return a list of indices of the decay modes that this daughter isotope is formed from given the parent isotope's name.
- std::vector< int > EmissionIndices (std::string parent)

  Return a list of indices of the decay modes that this particle emission is formed from given the parent isotope's name.

Protected Member Functions

- int setConstants ()

  Set the decay_modes, branch_ratios, and other info based on load library.
- int calculateMeanEnergies ()

  Calculate the mean radiation energies for the isotope.
- void computeDecayRate ()

  Compute the decay rate (in 1/s) based on the half-life.
- int addPairs (int i, std::string parent)

  Function to add parent/daughter pairs given the parent's name and the current level.
- YamlWrapper & getNuclideLibrary ()

  Return reference to the nuclide library.

Protected Attributes

- std::string IsoName

  Name of the isotope (e.g., H-2)
- std::vector< decay_mode > decay_modes

  List of decay modes the given isotope can undergo.
- std::vector< double > branch_ratios

  Branching ratios for each possible decay mode.
- std::vector< std::string > particle_emitted

  Name of the particle(s) ejected during spec_iso decay.
- std::vector< int > num_particles

Numbers of particles emitted during decay mode.

- std::vector< std::string > daughter

  Name of the daughter isotope formed.
- std::vector< double > Q_values

  Q-value (or maximum radiation energy) of the decay mode (in MeV)
- std::vector< double > mean_radiation_energy

  Mean radiation energy of the decay modes (calculated in MeV)
- std::vector< int > spin_jump

  Magnitude of change in spin from parent to daughter.
- std::vector< std::vector< std::pair< std::string, std::string > > > chain

  List of all parent and daughter/particles pairs in a chain given this isotope.
- double decay_rate

  Rate of decay for the given isotope (1/s)
- double half_life

  Half-life of the isotope (in hl_units)
- time_units hl_units

  Units given for the half-life.
- double hl_threshold

  Half-life value (in seconds) at which 99% of isotope has been converted.
- int isotope_number

  isotope number for the object (i.e., mass number)
- bool Stable

  Boolean is True if isotope is stable.
- bool IsomericState

  Boolean is True if isotope is in an isomeric state.
- double initial_condition

  Value to hold initial condition for this nuclide (moles or atoms)
- double concentration

  Value to hold concentration after a point in time (moles or atoms)
- bool Warnings

  Boolean is True if you want to print warnings to console.
- bool inMoles

  Boolean is True if units of concentration are in moles, False if units in atoms.
- double activity

  Radioactivity of the current nuclide (in disintigrations per second) or (Bq)
- double mass_excess

  Mass excess of the nuclide (in MeV)
- double spin_parity

  Magnitude of the spin parity of the nuclide.
- double ionization_coeff

  Average number of ion pairs produced per decay of this nuclide.
- yaml_cpp_class * nuclides

  Pointer to a yaml object storing the digital library of all nuclides.

Additional Inherited Members

5.29.1 Detailed Description

Isotope object to hold information and provide decay operations.

This is the C++ object to store information and functions associated with the decay of radioactive isotopes. It herits from the Atom object and extends that object to include information such as decay constants, branching ratios, and decay modes. It can be used to determine branching paths and setup systems of equations involving decay products.

5.29.2 Constructor & Destructor Documentation

5.29.2.1 Isotope()

```
Isotope::Isotope ( )
```

Default constructor.

5.29.2.2 ∼Isotope()

```
Isotope::∼Isotope ( )
```

Default destructor.

5.29.3 Member Function Documentation

5.29.3.1 loadNuclides()

```
void Isotope::loadNuclides (
             yaml_cpp_class & data )
```

Function to load the nuclide library into the pointer.

5.29.3.2 unloadNuclides()

```
void Isotope::unloadNuclides ( )
```

Delete the pointer to nuclide library to free space.

5.29.3.3 clearChain()

```
void Isotope::clearChain ( )
```

Delete the chain for this isotope to free space.

5.29.3.4 registerIsotope() [1/3]

```
int Isotope::registerIsotope (
             std::string isotope_name )
```

Register an isotope given the isotope name (e.g., H-2)

5.29.3.5 registerIsotope() [2/3]

```
int Isotope::registerIsotope (
             std::string symbol,
             int iso )
```

Register an isotope given an atomic symbol (e.g., H) and isotope number (e.g., 2)

5.29.3.6 registerIsotope() [3/3]

```
int Isotope::registerIsotope (
             int atom_num,
             int iso_num )
```

Register an isotope given both an atomic and isotope number (e.g., H-2 = 1, 2)

5.29.3.7 DisplayInfo()

```
void Isotope::DisplayInfo ( )
```

Print out isotope information to the console.

5.29.3.8 DisplayChain()

```
void Isotope::DisplayChain ( )
```

Print out chain information to the console.

5.29.3.9 createChain()

```
void Isotope::createChain ( )
```

Function to create and fill in the chain of nuclides for this starting isotope.

5.29.3.10 setInitialCondition()

```
void Isotope::setInitialCondition (
             double ic )
```

Set the value for the initial condition of this nuclide.

5.29.3.11 setConcentration()

```
void Isotope::setConcentration (
             double c )
```

Set the concentration value for the nuclide.

5.29.3.12 setActivity()

```
void Isotope::setActivity (
             double a )
```

Set the activity value for the nuclide.

5.29.3.13 setUnits2Moles()

```
void Isotope::setUnits2Moles (
             bool opt )
```

Set concentration units to moles if True is passed.

5.29.3.14 setWarnings()

```
void Isotope::setWarnings (
             bool opt )
```

Set the warnings boolean value.

5.29.3.15 setThreshold()

```
void Isotope::setThreshold (
             double val )
```

Set the threshold value for half-life (in sec)

5.29.3.16 updateDecayRate()

```
void Isotope::updateDecayRate ( )
```

Increase decay rate by 1% (used to correct issues with same eigenvalues)

5.29.3.17 calculateIonization()

```
int Isotope::calculateIonization (
             std::vector< Atom > & atoms,
             std::vector< double > & mass_fracs,
             double density,
             double potential )
```

Calculate the ionization coefficient given a list of atoms and their mass fractions in a particular media.

This function uses the Linear Energy Transfer function to calculate the average ionization potential of this nuclide decaying in a media made up of the given set of atoms and their respective mass fractions. The result is stored in the ionization_coeff parameter and can be accessed via the IonizationCoeff() return function. As the density and/or mass fraction of the media changes, this function needs to be re-called to update that ionization coefficient.

Parameters

atoms	reference to a list of atoms in the media
mass_fracs	reference to a list of mass fractions for the list of atoms
density	density of the media in g/cm^3
potential	ionization potential of the media in eV

5.29.3.18 IsotopeNumber()

```
int Isotope::IsotopeNumber ( )
```

Return the isotope number of the atom (i.e., mass number)

5.29.3.19 DecayRate()

```
double Isotope::DecayRate ( )
```

Return the decay rate of the isotope.

5.29.3.20 HalfLife()

```
double Isotope::HalfLife (
            time_units units )
```

Return the half-life in the given units.

5.29.3.21 HalfLifeUnits()

```
time_units Isotope::HalfLifeUnits ( )
```

Return the half-life units.

5.29.3.22 IsotopeName()

```
std::string Isotope::IsotopeName ( )
```

Return the name of the isotope.

5.29.3.23 isStable()

```
bool Isotope::isStable ( )
```

Return stability condition.

5.29.3.24 isIsomericState()

```
bool Isotope::isIsomericState ( )
```

Return isomeric condition.

5.29.3.25 isMolar()

```
bool Isotope::isMolar ( )
```

Returns True if concentration units are moles.

5.29.3.26 DecayModes()

```
int Isotope::DecayModes ( )
```

Return the number of decay modes.

5.29.3.27 getInitialCondition()

```
double Isotope::getInitialCondition ( )
```

Return the value of the initial condition.

5.29.3.28 getConcentration()

```
double Isotope::getConcentration ( )
```

Return the concentration value of the nuclide.

5.29.3.29 getActivity()

```
double Isotope::getActivity ( )
```

Return the activity value of the nuclide.

5.29.3.30 AtomicWeight()

```
double Isotope::AtomicWeight ( )
```

Return the atomic weight of the isotope.

5.29.3.31 MeltingPoint()

```
double Isotope::MeltingPoint ( )
```

Returns the melting point.

5.29.3.32 BoilingPoint()

```
double Isotope::BoilingPoint ( )
```

Returns the boiling point.

5.29.3.33 ThermalXSection()

```
double Isotope::ThermalXSection ( )
```

Returns the thermal cross section.

5.29.3.34 ScatterXSection()

```
double Isotope::ScatterXSection ( )
```

Returns the scattering cross section.

5.29.3.35 MassExcess()

```
double Isotope::MassExcess ( )
```

Returns the mass-excess of the nuclide (in MeV)

5.29.3.36 Jpi()

```
double Isotope::Jpi ( )
```

Returns the magnitude of the spin-parity.

5.29.3.37 IonizationCoeff()

```
double Isotope::IonizationCoeff ( )
```

Returns the average number of ion pairs produced from this nuclide decay.

5.29.3.38 DecayMode()

```
decay_mode Isotope::DecayMode (
              int i )
```

Return the ith decay mode.

5.29.3.39 BranchFraction()

```
double Isotope::BranchFraction (
              int i )
```

Return the ith branch fraction.

5.29.3.40 ParticleEmitted()

```
std::string Isotope::ParticleEmitted (
              int i )
```

Return the name of the particle emitted for the ith decay mode.

5.29.3.41 NumberParticlesEmitted()

```
int Isotope::NumberParticlesEmitted (
              int i )
```

Return the number of particles that get emitted.

5.29.3.42 Daughter()

```
std::string Isotope::Daughter (
              int i )
```

Return the name of the daughter isotope.

5.29.3.43 QValue()

```
double Isotope::QValue (
            int i )
```

Return the ith Q-value (in MeV)

5.29.3.44 deltaJ()

```
int Isotope::deltaJ (
            int i )
```

Return the ith spin jump.

5.29.3.45 MeanEnergy()

```
double Isotope::MeanEnergy (
            int i )
```

Return the ith mean radiation energy (in MeV)

5.29.3.46 DaughterIndices()

```
std::vector<int> Isotope::DaughterIndices (
            std::string parent )
```

Return a list of indices of the decay modes that this daughter isotope is formed from given the parent isotope's name.

This function will iterate through the decay modes for the parent isotope we are investigating and return a list of indices that represent the modes of decay that form this isotope from the given parent. If the mode does not form this isotope, then it will have an index of -1. The size of the vector on return will equal the size of the list of decay modes for the parent. If the parent is invalid, then it will return a vector of size 0. If the parent does not form this isotope, then all indices will be -1.

5.29.3.47 EmissionIndices()

```
std::vector<int> Isotope::EmissionIndices (
            std::string parent )
```

Return a list of indices of the decay modes that this particle emission is formed from given the parent isotope's name.

This function will iterate through the decay modes for the parent isotope we are investigating and return a list of indices that represent the modes of decay that form this isotope from the given parent. If the mode does not form this isotope, then it will have an index of -1. The size of the vector on return will equal the size of the list of decay modes for the parent. If the parent is invalid, then it will return a vector of size 0. If the parent does not form this isotope, then all indices will be -1.

5.29.3.48 setConstants()

```
int Isotope::setConstants ( )  [protected]
```

Set the decay_modes, branch_ratios, and other info based on load library.

5.29.3.49 calculateMeanEnergies()

```
int Isotope::calculateMeanEnergies ( )  [protected]
```

Calculate the mean radiation energies for the isotope.

5.29.3.50 computeDecayRate()

```
void Isotope::computeDecayRate ( )  [protected]
```

Compute the decay rate (in 1/s) based on the half-life.

5.29.3.51 addPairs()

```
int Isotope::addPairs (
             int i,
             std::string parent )  [protected]
```

Function to add parent/daughter pairs given the parent's name and the current level.

5.29.3.52 getNuclideLibrary()

```
YamlWrapper& Isotope::getNuclideLibrary ( )  [protected]
```

Return reference to the nuclide library.

5.29.4 Member Data Documentation

5.29.4.1 IsoName

```
std::string Isotope::IsoName  [protected]
```

Name of the isotope (e.g., H-2)

5.29.4.2 decay_modes

```
std::vector<decay_mode> Isotope::decay_modes  [protected]
```

List of decay modes the given isotope can undergo.

5.29.4.3 branch_ratios

```
std::vector<double> Isotope::branch_ratios  [protected]
```

Branching ratios for each possible decay mode.

5.29.4.4 particle_emitted

```
std::vector<std::string> Isotope::particle_emitted  [protected]
```

Name of the particle(s) ejected during spec_iso decay.

5.29.4.5 num_particles

```
std::vector<int> Isotope::num_particles  [protected]
```

Numbers of particles emitted during decay mode.

5.29.4.6 daughter

```
std::vector<std::string> Isotope::daughter  [protected]
```

Name of the daughter isotope formed.

5.29.4.7 Q_values

```
std::vector<double> Isotope::Q_values  [protected]
```

Q-value (or maximum radiation energy) of the decay mode (in MeV)

5.29.4.8 mean_radiation_energy

```
std::vector<double> Isotope::mean_radiation_energy  [protected]
```

Mean radiation energy of the decay modes (calculated in MeV)

5.29.4.9 spin_jump

```
std::vector<int> Isotope::spin_jump  [protected]
```

Magnitude of change in spin from parent to daughter.

5.29.4.10 chain

```
std::vector< std::vector< std::pair<std::string,std::string> > > Isotope::chain  [protected]
```

List of all parent and daughter/particles pairs in a chain given this isotope.

Stores the list of parent and daughter pairs. If the parent produces emitted particles, those go in this list as well. If the parent produces more than one daughter, all those daughters go in the list as well.

chain[i] = contains lists of parent daughter pairs at the ith level of the chain chain[i][j] = contains the jth pair at the ith level chain[i][j].first = contains the name of the parent chain[i][j].second = contains the name of the daughter/particle emitted

Example:

Levels: A A = Principal Parent -----— / \ level 0: B C B = 1st Daughter (pair A/B), C = 1st particle emission (pair A/C) / \ \ level 1: D E F D = 1st Daugher of 1st Daughter, E = 2nd Daughter of 1st Daughter, F = Stable Daughter / | | \ level 2: G H I J So on and so forth...

5.29.4.11 decay_rate

```
double Isotope::decay_rate  [protected]
```

Rate of decay for the given isotope (1/s)

5.29.4.12 half_life

```
double Isotope::half_life  [protected]
```

Half-life of the isotope (in hl_units)

5.29.4.13 hl_units

```
time_units Isotope::hl_units  [protected]
```

Units given for the half-life.

5.29.4.14 hl_threshold

```
double Isotope::hl_threshold  [protected]
```

Half-life value (in seconds) at which 99% of isotope has been converted.

5.29.4.15 isotope_number

```
int Isotope::isotope_number  [protected]
```

isotope number for the object (i.e., mass number)

5.29.4.16 Stable

```
bool Isotope::Stable  [protected]
```

Boolean is True if isotope is stable.

5.29.4.17 IsomericState

```
bool Isotope::IsomericState  [protected]
```

Boolean is True if isotope is in an isomeric state.

5.29.4.18 initial_condition

```
double Isotope::initial_condition  [protected]
```

Value to hold initial condition for this nuclide (moles or atoms)

5.29.4.19 concentration

```
double Isotope::concentration  [protected]
```

Value to hold concentration after a point in time (moles or atoms)

5.29.4.20 Warnings

```
bool Isotope::Warnings  [protected]
```

Boolean is True if you want to print warnings to console.

5.29.4.21 inMoles

```
bool Isotope::inMoles  [protected]
```

Boolean is True if units of concentration are in moles, False if units in atoms.

5.29.4.22 activity

```
double Isotope::activity  [protected]
```

Radioactivity of the current nuclide (in disintigrations per second) or (Bq)

5.29.4.23 mass_excess

```
double Isotope::mass_excess  [protected]
```

Mass excess of the nuclide (in MeV)

5.29.4.24 spin_parity

```
double Isotope::spin_parity  [protected]
```

Magnitude of the spin parity of the nuclide.

5.29.4.25 ionization_coeff

```
double Isotope::ionization_coeff  [protected]
```

Average number of ion pairs produced per decay of this nuclide.

5.29.4.26 nuclides

```
yaml_cpp_class* Isotope::nuclides  [protected]
```

Pointer to a yaml object storing the digital library of all nuclides.

The documentation for this class was generated from the following file:

- ibis.h

5.30 KeyValueMap Class Reference

Key-Value-Type Map object creating a map of the KeyValuePair objects.

```
#include <yaml_wrapper.h>
```

Public Member Functions

- KeyValueMap ()

  Default constructor.
- ∼KeyValueMap ()

  Default destructor.
- KeyValueMap (const std::map< std::string, std::string > &map)

  Construct from a map of strings.
- KeyValueMap (std::string key, std::string value)

  Construct one element in the map.
- KeyValueMap (const KeyValueMap &map)

  Copy constructor.
- KeyValueMap & operator= (const KeyValueMap &map)

  Equals overload.
- ValueTypePair & operator[ ] (const std::string key)

  Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const

  Return the ValueType at the give key.
- std::map< std::string, ValueTypePair > & getMap ()

  Return a reference to the Key_Value map object.
- std::map< std::string, ValueTypePair >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, ValueTypePair >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, ValueTypePair >::const_iterator begin () const

  Returns a const iterator pointing to the beginning of the list.
- std::map< std::string, ValueTypePair >::iterator begin ()

  Returns an iterator pointing to the beginning of the list.
- void clear ()

  Clears the map.
- void addKey (std::string key)

  Adds a key to the object with a default value.
- void editValue4Key (std::string val, std::string key)

  Edits a given value for a pre-existing key.
- void editValue4Key (std::string val, int type, std::string key)

  Edits a value for a pre-existing key and asserts type.
- void addPair (std::string key, ValueTypePair val)

  Adds a pair object to the map.
- void addPair (std::string key, std::string val)

  Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int type)

  Adds a pair object and asserts a type.
- void findType (std::string key)

  Find what data type the value at the key is.

- void assertType (std::string key, int type)

  Assert the given type at the given key.
- void findAllTypes ()

  Find all types for all data in map.
- void DisplayMap ()

  Print out the map to console.
- int size ()

  Returns the size of the map.
- std::string getString (std::string key)

  Retrieve the string at the key.
- bool getBool (std::string key)

  Retrieve the boolean at the key.
- double getDouble (std::string key)

  Retrieve the double at the key.
- int getInt (std::string key)

  Retrieve the int at the key.
- std::string getValue (std::string key)

  Retrieve the value at the key.
- int getType (std::string key)

  Retrieve the type at the key.
- ValueTypePair & getPair (std::string key)

  Retrieve the pair at the key.

Private Attributes

- std::map< std::string, ValueTypePair > Key_Value

  Map of Keys and Values paired with types.

5.30.1 Detailed Description

Key-Value-Type Map object creating a map of the KeyValuePair objects.

C++ Object that creates a map of the KeyValuePair objects. Functions defined here allow the user to iterate through this map, access specific keys in the map, edit values associated with those keys, find the data types for the values in those keys, ect. The keys are used as an access operator for their corresponding value. As such, each key in the map is required to be unique, but the values are allowed to be duplicated.

5.30.2 Constructor & Destructor Documentation

5.30.2.1 KeyValueMap() [1/4]

```
KeyValueMap::KeyValueMap ( )
```

Default constructor.

5.30.2.2 ∼KeyValueMap()

```
KeyValueMap::~KeyValueMap ( )
```

Default destructor.

5.30.2.3 KeyValueMap() [2/4]

```
KeyValueMap::KeyValueMap (
            const std::map< std::string, std::string > & map )
```

Construct from a map of strings.

5.30.2.4 KeyValueMap() [3/4]

```
KeyValueMap::KeyValueMap (
            std::string key,
            std::string value )
```

Construct one element in the map.

5.30.2.5 KeyValueMap() [4/4]

```
KeyValueMap::KeyValueMap (
            const KeyValueMap & map )
```

Copy constructor.

5.30.3 Member Function Documentation

5.30.3.1 operator=()

```
KeyValueMap& KeyValueMap::operator= (
            const KeyValueMap & map )
```

Equals overload.

5.30.3.2 operator[]() [1/2]

```
ValueTypePair& KeyValueMap::operator[] (
            const std::string key )
```

Return the ValueType reference at the given key.

5.30.3.3 operator[]() [2/2]

```
ValueTypePair KeyValueMap::operator[ ] (
             const std::string key ) const
```

Return the ValueType at the give key.

5.30.3.4 getMap()

```
std::map<std::string, ValueTypePair >& KeyValueMap::getMap ( )
```

Return a reference to the Key_Value map object.

5.30.3.5 end() [1/2]

```
std::map<std::string, ValueTypePair>::const_iterator KeyValueMap::end ( ) const
```

Returns a const iterator pointing to the end of the list.

5.30.3.6 end() [2/2]

```
std::map<std::string, ValueTypePair>::iterator KeyValueMap::end ( )
```

Returns an iterator pointing to the end of the list.

5.30.3.7 begin() [1/2]

```
std::map<std::string, ValueTypePair>::const_iterator KeyValueMap::begin ( ) const
```

Returns a const iterator pointing to the beginning of the list.

5.30.3.8 begin() [2/2]

```
std::map<std::string, ValueTypePair>::iterator KeyValueMap::begin ( )
```

Returns an iterator pointing to the beginning of the list.

5.30.3.9 clear()

```
void KeyValueMap::clear ( )
```

Clears the map.

5.30.3.10 addKey()

```
void KeyValueMap::addKey (
             std::string key )
```

Adds a key to the object with a default value.

5.30.3.11 editValue4Key() [1/2]

```
void KeyValueMap::editValue4Key (
             std::string val,
             std::string key )
```

Edits a given value for a pre-existing key.

5.30.3.12 editValue4Key() [2/2]

```
void KeyValueMap::editValue4Key (
             std::string val,
             int type,
             std::string key )
```

Edits a value for a pre-existing key and asserts type.

5.30.3.13 addPair() [1/3]

```
void KeyValueMap::addPair (
             std::string key,
             ValueTypePair val )
```

Adds a pair object to the map.

5.30.3.14 addPair() [2/3]

```
void KeyValueMap::addPair (
             std::string key,
             std::string val )
```

Adds a pair object to the map (with only strings)

5.30.3.15 addPair() [3/3]

```
void KeyValueMap::addPair (
             std::string key,
             std::string val,
             int type )
```

Adds a pair object and asserts a type.

5.30.3.16 findType()

```
void KeyValueMap::findType (
             std::string key )
```

Find what data type the value at the key is.

5.30.3.17 assertType()

```
void KeyValueMap::assertType (
             std::string key,
             int type )
```

Assert the given type at the given key.

5.30.3.18 findAllTypes()

```
void KeyValueMap::findAllTypes ( )
```

Find all types for all data in map.

5.30.3.19 DisplayMap()

```
void KeyValueMap::DisplayMap ( )
```

Print out the map to console.

5.30.3.20 size()

```
int KeyValueMap::size ( )
```

Returns the size of the map.

5.30.3.21 getString()

```
std::string KeyValueMap::getString (
             std::string key )
```

Retrieve the string at the key.

5.30.3.22 getBool()

```
bool KeyValueMap::getBool (
             std::string key )
```

Retrieve the boolean at the key.

5.30.3.23 getDouble()

```
double KeyValueMap::getDouble (
             std::string key )
```

Retrieve the double at the key.

5.30.3.24 getInt()

```
int KeyValueMap::getInt (
             std::string key )
```

Retrieve the int at the key.

5.30.3.25 getValue()

```
std::string KeyValueMap::getValue (
             std::string key )
```

Retrieve the value at the key.

5.30.3.26 getType()

```
int KeyValueMap::getType (
             std::string key )
```

Retrieve the type at the key.

5.30.3.27 getPair()

```
ValueTypePair& KeyValueMap::getPair (
            std::string key )
```

Retrieve the pair at the key.

5.30.4 Member Data Documentation

5.30.4.1 Key_Value

```
std::map<std::string, ValueTypePair > KeyValueMap::Key_Value  [private]
```

Map of Keys and Values paired with types.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.31 KMS_DATA Struct Reference

Data structure for the implemenation of the Krylov Multi-Space (KMS) Method.

```
#include <lark.h>
```

Public Attributes

- int level = 0

  Current level in the recursion.
- int max_level = 0

  Maximum allowable recursion levels (Default = 0 -> GMRES, Max = 5)
- int restart = -1

  Restart parameter for the outer iterates (Default = 20, Max = N)
- int maxit = 0

  Maximum allowable iterations for the outer steps.
- int inner_iter = 0

  Number of inner steps taken.
- int outer_iter = 0

  Number of outer steps taken.
- int total_iter = 0

  Total number of iterations in all steps.
- double outer_reltol = 1e-6

  Relative residual tolerance for outer steps (Default = 1e-6)
- double outer_abstol = 1e-6

  Absolute residual tolerance for outer steps (Default = 1e-6)
- double inner_reltol = 0.1

  Residual tolerance for inner steps made relative to outer steps (Default = 0.1)

- bool Output_outer = true

    True = Print the outer steps residuals.
- bool Output_inner = false

    True = Print the inner steps residuals.
- GMRESRP_DATA gmres_out

    Data structure for the outer steps.
- std::vector< GMRESRP_DATA > gmres_in

    Data structures for each recursion level.
- int(∗ matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void ∗matvec_data)

    User supplied matrix-vector product function.
- int(∗ terminal_precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗precon_data)

    Optional user supplied terminal preconditioner.
- const void ∗ matvec_data

    Data structure for the user's matvec function.
- const void ∗ term_precon

    Data structure for the user's terminal preconditioner.

5.31.1 Detailed Description

Data structure for the implemenation of the Krylov Multi-Space (KMS) Method.

C-style object to be used in conjunction with the Krylov Multi-Space (KMS) Algorithm to iteratively solve non-symmetric, indefinite linear systems. This method was inspired by the Flexible GMRES (FGMRES) and Recursive GMRES (GMRESR) methods proposed by Saad (1993) and Vorst and Vuik (1991), respectively. The idea behind this method is to recursively call FGMRES to solve a linear system with pregressively smaller Krylov Subspaces built by a Right-Preconditioned GMRES algorithm. Thus creating a "V-cycle" of iteration similar to that seen in Multi-Grid algorithms.

5.31.2 Member Data Documentation

5.31.2.1 level

```
int KMS_DATA::level = 0
```

Current level in the recursion.

5.31.2.2 max_level

```
int KMS_DATA::max_level = 0
```

Maximum allowable recursion levels (Default = 0 -> GMRES, Max = 5)

5.31.2.3 restart

```
int KMS_DATA::restart = -1
```

Restart parameter for the outer iterates (Default = 20, Max = N)

5.31.2.4 maxit

```
int KMS_DATA::maxit = 0
```

Maximum allowable iterations for the outer steps.

5.31.2.5 inner_iter

```
int KMS_DATA::inner_iter = 0
```

Number of inner steps taken.

5.31.2.6 outer_iter

```
int KMS_DATA::outer_iter = 0
```

Number of outer steps taken.

5.31.2.7 total_iter

```
int KMS_DATA::total_iter = 0
```

Total number of iterations in all steps.

5.31.2.8 outer_reltol

```
double KMS_DATA::outer_reltol = 1e-6
```

Relative residual tolerance for outer steps (Default = 1e-6)

5.31.2.9 outer_abstol

```
double KMS_DATA::outer_abstol = 1e-6
```

Absolute residual tolerance for outer steps (Default = 1e-6)

5.31.2.10 inner_reltol

```
double KMS_DATA::inner_reltol = 0.1
```

Residual tolerance for inner steps made relative to outer steps (Default = 0.1)

5.31.2.11 Output_outer

```
bool KMS_DATA::Output_outer = true
```

True = Print the outer steps residuals.

5.31.2.12 Output_inner

```
bool KMS_DATA::Output_inner = false
```

True = Print the inner steps residuals.

5.31.2.13 gmres_out

```
GMRESRP_DATA KMS_DATA::gmres_out
```

Data structure for the outer steps.

5.31.2.14 gmres_in

```
std::vector<GMRESRP_DATA> KMS_DATA::gmres_in
```

Data structures for each recursion level.

5.31.2.15 matvec

```
int(* KMS_DATA::matvec) (const Matrix< double > &x, Matrix< double > &Ax, const void *matvec↩
_data)
```

User supplied matrix-vector product function.

5.31.2.16 terminal_precon

```
int(* KMS_DATA::terminal_precon) (const Matrix< double > &r, Matrix< double > &p, const void
*precon_data)
```

Optional user supplied terminal preconditioner.

5.31.2.17 matvec_data

```
const void* KMS_DATA::matvec_data
```

Data structure for the user's matvec function.

5.31.2.18 term_precon

```
const void* KMS_DATA::term_precon
```

Data structure for the user's terminal preconditioner.

The documentation for this struct was generated from the following file:

- lark.h

5.32 LineElement Class Reference

LineElement.

```
#include <mesh.h>
```

Public Member Functions

- LineElement ()
  Default Constructor.
- ∼LineElement ()
  Default Destructor.
- void DisplayInfo ()
  Print out information to the console.
- void AssignNodes (Node &n1, Node &n2)
  Assign nodes for the line segment.
- void AssignIDnumber (unsigned int i)
  Assign the id number for the line segment.
- void calculateLength ()
  Calculate and store length value.
- void findMidpoint ()
  Find and set the midpoint node.
- void determineType ()
  Determine the type of line element.
- void evaluateProperties ()
  Calls functions for length and midpoint.

Private Attributes

- Node ∗ node1

    Pointer to first node.
- Node ∗ node2

    Pointer to second node.
- double length

    Length of the line segment.
- Node midpoint

    Midpoint node for the line segment.
- unsigned int IDnum

    Identification number for the line element.
- element_type SubType

    Type of line segment (BOUDNARY or INTERIOR)

5.32.1 Detailed Description

LineElement.

This class structure creates a C++ object for a line. The line is made up of a reference to two distinct nodes. Based on those nodes, we can tell whether or not the line is on the boundary of a mesh or part of the interior. We will also know the length of the line and the midpoint of the line.

5.32.2 Constructor & Destructor Documentation

5.32.2.1 LineElement()

```
LineElement::LineElement ( )
```

Default Constructor.

5.32.2.2 ∼LineElement()

```
LineElement::∼LineElement ( )
```

Default Destructor.

5.32.3 Member Function Documentation

5.32.3.1 DisplayInfo()

```
void LineElement::DisplayInfo ( )
```

Print out information to the console.

5.32.3.2 AssignNodes()

```
void LineElement::AssignNodes (
            Node & n1,
            Node & n2 )
```

Assign nodes for the line segment.

5.32.3.3 AssignIDnumber()

```
void LineElement::AssignIDnumber (
            unsigned int i )
```

Assign the id number for the line segment.

5.32.3.4 calculateLength()

```
void LineElement::calculateLength ( )
```

Calculate and store length value.

5.32.3.5 findMidpoint()

```
void LineElement::findMidpoint ( )
```

Find and set the midpoint node.

5.32.3.6 determineType()

```
void LineElement::determineType ( )
```

Determine the type of line element.

5.32.3.7 evaluateProperties()

```
void LineElement::evaluateProperties ( )
```

Calls functions for length and midpoint.

5.32.4 Member Data Documentation

5.32.4.1 node1

`Node* LineElement::node1 [private]`

Pointer to first node.

5.32.4.2 node2

`Node* LineElement::node2 [private]`

Pointer to second node.

5.32.4.3 length

`double LineElement::length [private]`

Length of the line segment.

5.32.4.4 midpoint

`Node LineElement::midpoint [private]`

Midpoint node for the line segment.

5.32.4.5 IDnum

`unsigned int LineElement::IDnum [private]`

Identification number for the line element.

5.32.4.6 SubType

`element_type LineElement::SubType [private]`

Type of line segment (BOUDNARY or INTERIOR)

The documentation for this class was generated from the following file:

- mesh.h

5.33 MAGPIE_DATA Struct Reference

MAGPIE Data Structure.

```
#include <magpie.h>
```

Public Attributes

- std::vector< GSTA_DATA > gsta_dat
- std::vector< mSPD_DATA > mspd_dat
- std::vector< GPAST_DATA > gpast_dat
- SYSTEM_DATA sys_dat

5.33.1 Detailed Description

MAGPIE Data Structure.

C-style object holding all information necessary to run a MAGPIE simulation. This is the data structure that will be used in other sub-routines when a mixed gas adsorption simulation needs to be run.

5.33.2 Member Data Documentation

5.33.2.1 gsta_dat

```
std::vector<GSTA_DATA> MAGPIE_DATA::gsta_dat
```

5.33.2.2 mspd_dat

```
std::vector<mSPD_DATA> MAGPIE_DATA::mspd_dat
```

5.33.2.3 gpast_dat

```
std::vector<GPAST_DATA> MAGPIE_DATA::gpast_dat
```

5.33.2.4 sys_dat

```
SYSTEM_DATA MAGPIE_DATA::sys_dat
```

The documentation for this struct was generated from the following file:

- magpie.h

5.34 MassBalance Class Reference

Mass Balance Object.

```
#include <shark.h>
```

Public Member Functions

- MassBalance ()

    Default Constructor.
- ∼MassBalance ()

    Default Destructor.
- void Initialize_Object (MasterSpeciesList &List)

    Function to initialize the MassBalance object from the MasterSpeciesList.
- void Display_Info ()

    Display the mass balance information.
- void Set_Delta (int i, double v)

    Function to set the ith weight (delta) for the mass balance.
- void Set_TotalConcentration (double v)

    Set the total concentration of the mass balance to v (mol/L)
- void Set_Type (int type)

    Set the Mass Balance type to BATCH, CSTR, or PFR.
- void Set_Volume (double v)

    Set the volume of the reactor.
- void Set_FlowRate (double v)

    Set the flow rate for the CSTR or PFR.
- void Set_Area (double v)

    Set the cross sectional area for the PFR.
- void Set_TimeStep (double v)

    Set the time step for the CSTR or PFR.
- void Set_InitialConcentration (double v)

    Set the initial concentration for the mass balance.
- void Set_InletConcentration (double v)

    Set the inlet concentration for the CSTR or PFR.
- void Set_SteadyState (bool ss)

    Set the boolean for Steady-State simulation.
- void Set_ZeroInitialSolids (bool solids)

    Set the boolean for initial solids in solution.
- void Set_Name (std::string name)

    Set the name of the mass balance (i.e., Uranium, Carbonate, etc.)
- double Get_Delta (int i)

    Fetch the ith weight (i.e., delta) value.
- double Sum_Delta ()

    Sums up the delta values and returns the total (should never be zero)
- double Get_TotalConcentration ()

    Fetch the total concentration (mol/L)
- int Get_Type ()

    Fetch the reactor type.
- double Get_Volume ()

    Fetch the reactor volume.

- double Get_FlowRate ()

  Fetch the reactor flow rate.
- double Get_Area ()

  Fetch the reactor cross section area.
- double Get_TimeStep ()

  Fetch the time step.
- double Get_InitialConcentration ()

  Fetch the initial concentration.
- double Get_InletConcentration ()

  Fetch the inlet concentration.
- bool isSteadyState ()

  Fetch the steady-state condition.
- bool isZeroInitialSolids ()

  Fetch the initial solids condition.
- std::string Get_Name ()

  Return name of mass balance object.
- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old)

  Evaluate the residual for the mass balance object given the log(C) concentrations.
- double Eval_IC_Residual (const Matrix< double > &x)

  Evaluate the initial residual for the unsteady mass balance object given the log(C) concentrations.

Protected Attributes

- MasterSpeciesList ∗ List

  Pointer to a master species object.
- std::vector< double > Delta

  Vector of weights (i.e., deltas) used in the mass balance.
- double TotalConcentration

  Total concentration of specific object (mol/L)
- int Type

  Type of mass balance object (default = BATCH)
- double volume

  Volume of the reactor (L)
- double flow_rate

  Volumetric flow rate in reactor (L/hr)
- double xsec_area

  Cross sectional area in PFR configuration (m^2)
- double dt

  Time step for non-batch case (hrs)
- double InitialConcentration

  Concentration initially in the domain (mol/L)
- double InletConcentration

  Concentration in the inlet of the domain (mol/L)
- bool SteadyState

  True if running steady-state simulation.
- bool ZeroInitialSolids

  True if zero solids present for initial condition.

Private Attributes

- std::string Name

  Name designation used in mass balance.

5.34.1 Detailed Description

Mass Balance Object.

C++ style object that holds data and functions associated with mass balances of primary species in a system. The mass balances involve a total concentration (in mol/L) and a vector of weighted contributions to that total concentration from each species in the MasterSpeciesList. This object only considers mass balances in a batch type of system (i.e., not input or output of mass). However, one could inherit from this object to create mass balances for flow systems as well.

5.34.2 Constructor & Destructor Documentation

5.34.2.1 MassBalance()

```
MassBalance::MassBalance ( )
```

Default Constructor.

5.34.2.2 ∼MassBalance()

```
MassBalance::∼MassBalance ( )
```

Default Destructor.

5.34.3 Member Function Documentation

5.34.3.1 Initialize_Object()

```
void MassBalance::Initialize_Object (
            MasterSpeciesList & List )
```

Function to initialize the MassBalance object from the MasterSpeciesList.

5.34.3.2 Display_Info()

```
void MassBalance::Display_Info ( )
```

Display the mass balance information.

5.34.3.3 Set_Delta()

```
void MassBalance::Set_Delta (
            int i,
            double v )
```

Function to set the ith weight (delta) for the mass balance.

This function sets the weight (i.e., delta value) of the ith species in the list to the value of v. That value represents the weighting of that species in the determination of the total mass for the primary species set.

Parameters

i	index of the species in the MasterSpeciesList
v	value of the weigth (or delta) applied to the mass balance

5.34.3.4 Set_TotalConcentration()

```
void MassBalance::Set_TotalConcentration (
             double v )
```

Set the total concentration of the mass balance to v (mol/L)

5.34.3.5 Set_Type()

```
void MassBalance::Set_Type (
             int type )
```

Set the Mass Balance type to BATCH, CSTR, or PFR.

5.34.3.6 Set_Volume()

```
void MassBalance::Set_Volume (
             double v )
```

Set the volume of the reactor.

5.34.3.7 Set_FlowRate()

```
void MassBalance::Set_FlowRate (
             double v )
```

Set the flow rate for the CSTR or PFR.

5.34.3.8 Set_Area()

```
void MassBalance::Set_Area (
             double v )
```

Set the cross sectional area for the PFR.

5.34.3.9 Set_TimeStep()

```
void MassBalance::Set_TimeStep (
             double v )
```

Set the time step for the CSTR or PFR.

5.34.3.10 Set_InitialConcentration()

```
void MassBalance::Set_InitialConcentration (
             double v )
```

Set the initial concentration for the mass balance.

5.34.3.11 Set_InletConcentration()

```
void MassBalance::Set_InletConcentration (
             double v )
```

Set the inlet concentration for the CSTR or PFR.

5.34.3.12 Set_SteadyState()

```
void MassBalance::Set_SteadyState (
             bool ss )
```

Set the boolean for Steady-State simulation.

5.34.3.13 Set_ZeroInitialSolids()

```
void MassBalance::Set_ZeroInitialSolids (
             bool solids )
```

Set the boolean for initial solids in solution.

5.34.3.14 Set_Name()

```
void MassBalance::Set_Name (
             std::string name )
```

Set the name of the mass balance (i.e., Uranium, Carbonate, etc.)

5.34.3.15 Get_Delta()

```
double MassBalance::Get_Delta (
             int i )
```

Fetch the ith weight (i.e., delta) value.

5.34.3.16 Sum_Delta()

```
double MassBalance::Sum_Delta ( )
```

Sums up the delta values and returns the total (should never be zero)

5.34.3.17 Get_TotalConcentration()

```
double MassBalance::Get_TotalConcentration ( )
```

Fetch the total concentration (mol/L)

5.34.3.18 Get_Type()

```
int MassBalance::Get_Type ( )
```

Fetch the reactor type.

5.34.3.19 Get_Volume()

```
double MassBalance::Get_Volume ( )
```

Fetch the reactor volume.

5.34.3.20 Get_FlowRate()

```
double MassBalance::Get_FlowRate ( )
```

Fetch the reactor flow rate.

5.34.3.21 Get_Area()

```
double MassBalance::Get_Area ( )
```

Fetch the reactor cross section area.

5.34.3.22 Get_TimeStep()

```
double MassBalance::Get_TimeStep ( )
```

Fetch the time step.

5.34.3.23 Get_InitialConcentration()

```
double MassBalance::Get_InitialConcentration ( )
```

Fetch the initial concentration.

5.34.3.24 Get_InletConcentration()

```
double MassBalance::Get_InletConcentration ( )
```

Fetch the inlet concentration.

5.34.3.25 isSteadyState()

```
bool MassBalance::isSteadyState ( )
```

Fetch the steady-state condition.

5.34.3.26 isZeroInitialSolids()

```
bool MassBalance::isZeroInitialSolids ( )
```

Fetch the initial solids condition.

5.34.3.27 Get_Name()

```
std::string MassBalance::Get_Name ( )
```

Return name of mass balance object.

5.34.3.28 Eval_Residual()

```
double MassBalance::Eval_Residual (
             const Matrix< double > & x_new,
             const Matrix< double > & x_old )
```

Evaluate the residual for the mass balance object given the log(C) concentrations.

This function calculates and provides the residual for this mass balance object based on the total concentration in the system and the weighted contributions from each species. Concentrations are given as the log(C) values.

Parameters

x_new	matrix of the log(C) concentration values at the current non-linear step
x_old	matrix of the old log(C) concentration values for transient simulations

5.34.3.29 Eval_IC_Residual()

```
double MassBalance::Eval_IC_Residual (
             const Matrix< double > & x )
```

Evaluate the initial residual for the unsteady mass balance object given the log(C) concentrations.

This function calculates and provides the initial residual for this mass balance object based on the initial concentration in the system and the weighted contributions from each species. Concentrations are given as the log(C) values.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.34.4 Member Data Documentation

5.34.4.1 List

```
MasterSpeciesList* MassBalance::List  [protected]
```

Pointer to a master species object.

5.34.4.2 Delta

```
std::vector<double> MassBalance::Delta  [protected]
```

Vector of weights (i.e., deltas) used in the mass balance.

5.34.4.3 TotalConcentration

```
double MassBalance::TotalConcentration  [protected]
```

Total concentration of specific object (mol/L)

5.34.4.4 Type

```
int MassBalance::Type  [protected]
```

Type of mass balance object (default = BATCH)

5.34.4.5 volume

```
double MassBalance::volume  [protected]
```

Volume of the reactor (L)

5.34.4.6 flow_rate

```
double MassBalance::flow_rate  [protected]
```

Volumetric flow rate in reactor (L/hr)

5.34.4.7 xsec_area

```
double MassBalance::xsec_area  [protected]
```

Cross sectional area in PFR configuration (m^2)

5.34.4.8 dt

```
double MassBalance::dt  [protected]
```

Time step for non-batch case (hrs)

5.34.4.9 InitialConcentration

```
double MassBalance::InitialConcentration  [protected]
```

Concentration initially in the domain (mol/L)

5.34.4.10 InletConcentration

```
double MassBalance::InletConcentration  [protected]
```

Concentration in the inlet of the domain (mol/L)

5.34.4.11 SteadyState

```
bool MassBalance::SteadyState  [protected]
```

True if running steady-state simulation.

5.34.4.12 ZeroInitialSolids

```
bool MassBalance::ZeroInitialSolids  [protected]
```

True if zero solids present for initial condition.

5.34.4.13 Name

```
std::string MassBalance::Name  [private]
```

Name designation used in mass balance.

The documentation for this class was generated from the following file:

- shark.h

5.35 MasterSpeciesList Class Reference

Master Species List Object.

```
#include <shark.h>
```

Public Member Functions

- MasterSpeciesList ()

    Default constructor.
- ∼MasterSpeciesList ()

    Default destructor.
- MasterSpeciesList (const MasterSpeciesList &msl)

    Copy Constructor.
- MasterSpeciesList & operator= (const MasterSpeciesList &msl)

    Equals operator.
- void set_list_size (int i)

    Function to initialize the size of the list.
- void set_species (int i, std::string formula)

    Function to register the ith species in the list based on a registered molecular formula (see mola.h)
- void set_species (int i, int charge, double enthalpy, double entropy, double energy, bool HS, bool G, std::string Phase, std::string Name, std::string Formula, std::string lin_formula)

    Function to register the ith species in the list based on custom molecule information (see mola.h)
- void DisplayInfo (int i)

    Function to display information of ith object.

- void DisplayAll ()

  Function to display all information of list.
- void DisplayConcentrations (Matrix< double > &C)

  Function to display the concentrations of species in list.
- void set_alkalinity (double alk)

  Set the alkalinity of the solution (Default = 0 M)
- int list_size ()

  Returns size of list.
- Molecule & get_species (int i)

  Returns a reference to the ith species in master list.
- int get_index (std::string name)

  Returns an integer representing location of the named species in the list.
- double charge (int i)

  Fetch and return charge of ith species in list.
- double alkalinity ()

  Fetch the value of alkalinity of the solution (mol/L)
- std::string speciesName (int i)

  Function to return the name of the ith species.
- double Eval_ChargeResidual (const Matrix< double > &x)

  Calculate charge balance residual for the electroneutrality constraint.

Protected Attributes

- int size

  Size of the list.
- std::vector< Molecule > species

  List of Molecule Objects.
- double residual_alkalinity

  Conc of strong base - conc of strong acid in solution (mol/L)

5.35.1 Detailed Description

Master Species List Object.

C++ style object that holds data and function associated with solving multi-species problems. This object contains a vector of Molecule objects from mola.h and uses those objects to help setup speciation problems that need to be solved. One of the primary functions in this object is the contribution of electroneutrality (Eval_ChargeResidual). However, we only need this constraint if the pH of our aqueous system is unknown.

5.35.2 Constructor & Destructor Documentation

5.35.2.1 MasterSpeciesList() [1/2]

```
MasterSpeciesList::MasterSpeciesList ( )
```

Default constructor.

5.35.2.2 ∼MasterSpeciesList()

```
MasterSpeciesList::∼MasterSpeciesList ( )
```

Default destructor.

5.35.2.3 MasterSpeciesList() [2/2]

```
MasterSpeciesList::MasterSpeciesList (
             const MasterSpeciesList & msl )
```

Copy Constructor.

5.35.3 Member Function Documentation

5.35.3.1 operator=()

```
MasterSpeciesList& MasterSpeciesList::operator= (
             const MasterSpeciesList & msl )
```

Equals operator.

5.35.3.2 set_list_size()

```
void MasterSpeciesList::set_list_size (
             int i )
```

Function to initialize the size of the list.

5.35.3.3 set_species() [1/2]

```
void MasterSpeciesList::set_species (
             int i,
             std::string formula )
```

Function to register the ith species in the list based on a registered molecular formula (see mola.h)

5.35.3.4 set_species() [2/2]

```
void MasterSpeciesList::set_species (
            int i,
            int charge,
            double enthalpy,
            double entropy,
            double energy,
            bool HS,
            bool G,
            std::string Phase,
            std::string Name,
            std::string Formula,
            std::string lin_formula )
```

Function to register the ith species in the list based on custom molecule information (see mola.h)

5.35.3.5 DisplayInfo()

```
void MasterSpeciesList::DisplayInfo (
            int i )
```

Function to display information of ith object.

5.35.3.6 DisplayAll()

```
void MasterSpeciesList::DisplayAll ( )
```

Function to display all information of list.

5.35.3.7 DisplayConcentrations()

```
void MasterSpeciesList::DisplayConcentrations (
            Matrix< double > & C )
```

Function to display the concentrations of species in list.

This function will print to the console the species list in order with each species associated concentration from the matrix C. The concentrations and species list MUST be in the same order and the units of C are assumed to be mol/L.

Parameters

C	matrix of concentrations of species in the list in mol/L

5.35.3.8 set_alkalinity()

```
void MasterSpeciesList::set_alkalinity (
             double alk )
```

Set the alkalinity of the solution (Default = 0 M)

This function is used to set the value of residual alkalinity used in the electroneutrality calculations. Typically, this value will be 0 M (mol/L) if all species in the system are present as variables. However, occasionally, one may want to set the alkalinity of the solution to a constant in order to restrict the pH of the solution.

Parameters

alk	Residual alkalinity in M (mol/L)

5.35.3.9 list_size()

```
int MasterSpeciesList::list_size ( )
```

Returns size of list.

5.35.3.10 get_species()

```
Molecule& MasterSpeciesList::get_species (
             int i )
```

Returns a reference to the ith species in master list.

This function will return a Molecule object for the ith species in the list of molecules. Once returned, the user then can operate on that molecule using the functions define in mola.h.

5.35.3.11 get_index()

```
int MasterSpeciesList::get_index (
             std::string name )
```

Returns an integer representing location of the named species in the list.

5.35.3.12 charge()

```
double MasterSpeciesList::charge (
             int i )
```

Fetch and return charge of ith species in list.

5.35.3.13 alkalinity()

```
double MasterSpeciesList::alkalinity ( )
```

Fetch the value of alkalinity of the solution (mol/L)

5.35.3.14 speciesName()

```
std::string MasterSpeciesList::speciesName (
             int i )
```

Function to return the name of the ith species.

5.35.3.15 Eval_ChargeResidual()

```
double MasterSpeciesList::Eval_ChargeResidual (
             const Matrix< double > & x )
```

Calculate charge balance residual for the electroneutrality constraint.

This function returns the value of the residual for the electroneutrality equation in the system. Electroneutrality is based on the concentrations and charges of each species in the system so the charges of each molecule must be appropriately set. Concentrations of those species are fed into this function via the argument x, but come in as the log(C) values (i.e., x = log(C)).

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.35.4 Member Data Documentation

5.35.4.1 size

```
int MasterSpeciesList::size  [protected]
```

Size of the list.

5.35.4.2 species

```
std::vector<Molecule> MasterSpeciesList::species  [protected]
```

List of Molecule Objects.

5.35.4.3 residual_alkalinity

`double MasterSpeciesList::residual_alkalinity [protected]`

Conc of strong base - conc of strong acid in solution (mol/L)

The documentation for this class was generated from the following file:

- shark.h

5.36 Matrix< T > Class Template Reference

Templated C++ Matrix Class Object (click Matrix to go to function definitions)

`#include <macaw.h>`

Public Member Functions

- Matrix (int rows, int columns)

  Constructor for matrix with given number of rows and columns.
- T & operator() (int i, int j)

  Access operator for the matrix element at row i and column j (e.g., aij = A(i,j))
- T operator() (int i, int j) const

  Constant access operator for the the matrix element at row i and column j.
- Matrix (const Matrix &M)

  Copy constructor for constructing a matrix as a copy of another matrix.
- Matrix & operator= (const Matrix &M)

  Equals operator for setting one matrix equal to another matrix.
- Matrix ()

  Default constructor for creating an empty matrix.
- ∼Matrix ()

  Default destructor for clearing out memory.
- void set_size (int i, int j)

  Function to set/change the size of a matrix to i rows and j columns.
- void zeros ()

  Function to set/change all values in a matrix to zeros.
- void edit (int i, int j, T value)

  Function to set/change the element of a matrix at row i and column j to given value.
- int rows ()

  Function to return the number of rows in a given matrix.
- int columns ()

  Function to return the number of columns in a matrix.
- T determinate ()

  Function to compute the determinate of a matrix and return that value.
- T norm ()

  Function to compute the L2-norm of a matrix and return that value.
- T sum ()

  Function to compute the sum of all elements in a matrix and return that value.
- T inner_product (const Matrix &x)

  Function to compute the inner product between this matrix and matrix x.

- Matrix & cofactor (const Matrix &M)

  Function to convert this matrix to a cofactor matrix of the given matrix M.
- Matrix operator+ (const Matrix &M)

  Operator to add this matrix and matrix M and return the new matrix result.
- Matrix operator- (const Matrix &M)

  Operator to subtract this matrix and matrix M and return the new matrix result.
- Matrix operator∗ (const T)

  Operator to multiply this matrix by a scalar T return the new matrix result.
- Matrix operator/ (const T)

  Operator to divide this matrix by a scalar T and return the new matrix result.
- Matrix operator+ (const T)

  Operator to add this matrix to a scalar T and return the new matrix result.
- Matrix operator- (const T)

  Operator to subtract this matrix to a scalar T and return the new matrix result.
- Matrix operator∗ (const Matrix &M)

  Operator to multiply this matrix and matrix M and return the new matrix result.
- Matrix outer_product (const Matrix &M)

  Operator to perform an outer product between this and M and return result.
- Matrix & transpose (const Matrix &M)
- Matrix & transpose_multiply (const Matrix &MT, const Matrix &v)

  Function to convert this matrix into the result of the given matrix M transposed and multiplied by the other given matrix v.
- Matrix & adjoint (const Matrix &M)

  Function to convert this matrix to the adjoint of the given matrix.
- Matrix & inverse (const Matrix &M)

  Function to convert this matrix to the inverse of the given matrix.
- void Display (const std::string Name)

  Function to display the contents of this matrix given a Name for the matrix.
- Matrix & tridiagonalSolve (const Matrix &A, const Matrix &b)

  Function to solve Ax=b for x if A is symmetric, tridiagonal (this->x)
- Matrix & ladshawSolve (const Matrix &A, const Matrix &d)

  Function to solve Ax=d for x if A is non-symmetric, tridiagonal (this->x)
- Matrix & tridiagonalFill (const T A, const T B, const T C, bool Spherical)

  Function to fill in this matrix with coefficients A, B, and C to form a tridiagonal matrix.
- Matrix & naturalLaplacian3D (int m)

  Function to fill out this matrix with coefficients from a 3D Laplacian function.
- Matrix & sphericalBCFill (int node, const T coeff, T variable)

  Function to fill out a column matrix with spherical specific boundary conditions.
- Matrix & ConstantICFill (const T IC)

  Function to set all values in a column matrix to a given constant.
- Matrix & SolnTransform (const Matrix &A, bool Forward)

  Function to transform the values in a column matrix from cartesian to spherical coordinates.
- T sphericalAvg (double radius, double dr, double bound, bool Dirichlet)

  Function to compute a spatial average of this column matrix in spherical coordinates.
- T IntegralAvg (double radius, double dr, double bound, bool Dirichlet)

  Function to compute a spatial average of this column matrix in spherical coordinates.
- T IntegralTotal (double dr, double bound, bool Dirichlet)

  Function to compute a spatial total of this column matrix in spherical coordinates.
- Matrix & tridiagonalVectorFill (const std::vector< T > &A, const std::vector< T > &B, const std::vector< T > &C)

  Function to fill in this matrix, in tridiagonal fashion, using the vectors of coefficients.

- Matrix & columnVectorFill (const std::vector< T > &A)

  Function to fill in a column matrix with the values of the given vector object.
- Matrix & columnProjection (const Matrix &b, const Matrix &b_old, const double dt, const double dt_old)

  Function to project a column matrix solution in time based on older state vectors.
- Matrix & dirichletBCFill (int node, const T coeff, T variable)

  Function to fill in a column matrix with all zeros except at the given node.
- Matrix & diagonalSolve (const Matrix &D, const Matrix &v)

  Function to solve the system Dx=v for x given that D is diagonal (this->x)
- Matrix & upperTriangularSolve (const Matrix &U, const Matrix &v)

  Function to solve the system Ux=v for x given that U is upper Triagular (this->x)
- Matrix & lowerTriangularSolve (const Matrix &L, const Matrix &v)

  Function to solve the system Lx=v for x given that L is lower Triagular (this->x)
- Matrix & upperHessenberg2Triangular (Matrix &b)

  Function to convert this square matrix to upper Triangular (assuming this is upper Hessenberg)
- Matrix & lowerHessenberg2Triangular (Matrix &b)

  Function to convert this square matrix to lower Triangular (assuming this is lower Hessenberg)
- Matrix & upperHessenbergSolve (const Matrix &H, const Matrix &v)

  Function to solve the system Hx=v for x given that H is upper Hessenberg (this->x)
- Matrix & lowerHessenbergSolve (const Matrix &H, const Matrix &v)

  Function to solve the system Hx=v for x given that H is lower Hessenberg (this->x)
- Matrix & qrSolve (const Matrix &M, const Matrix &b)

  Function to solve the system Mx=b using QR factorization for x given that M is invertable.
- Matrix & columnExtract (int j, const Matrix &M)

  Function to set this column matrix to the jth column of the given matrix M.
- Matrix & rowExtract (int i, const Matrix &M)

  Function to set this row matrix to the ith row of the given matrix M.
- Matrix & columnReplace (int j, const Matrix &v)

  Function to this matrices' jth column with the given column matrix v.
- Matrix & rowReplace (int i, const Matrix &v)

  Function to this matrices' ith row with the given row matrix v.
- void rowShrink ()

  Function to delete the last row of this matrix.
- void columnShrink ()

  Function to delete the last column of this matrix.
- void rowExtend (const Matrix &v)

  Function to add the row matrix v to the end of this matrix.
- void columnExtend (const Matrix &v)

  Function to add the column matrix v to the end of this matrix.

Protected Attributes

- int num_rows

  Number of rows of the matrix.
- int num_cols

  Number of columns of the matrix.
- std::vector< T > Data

  Storage vector for the elements of the matrix.

5.36.1 Detailed Description

template<class T>
class Matrix< T >

Templated C++ Matrix Class Object (click Matrix to go to function definitions)

C++ templated class object containing many different functions, actions, and solver routines associated with Dense Matrices. Operator overloads are also provided to give the user a more natural way of operating matrices on other matrices or scalars. These operator overloads are especially useful for reducing the amount of code needed to be written when working with matrix-based problems.

5.36.2 Constructor & Destructor Documentation

5.36.2.1 Matrix() [1/3]

```
template<class T >
Matrix< T >::Matrix (
              int rows,
              int columns )
```

Constructor for matrix with given number of rows and columns.

5.36.2.2 Matrix() [2/3]

```
template<class T >
Matrix< T >::Matrix (
              const Matrix< T > & M )
```

Copy constructor for constructing a matrix as a copy of another matrix.

5.36.2.3 Matrix() [3/3]

```
template<class T >
Matrix< T >::Matrix ( )
```

Default constructor for creating an empty matrix.

5.36.2.4 ∼Matrix()

```
template<class T >
Matrix< T >::∼Matrix ( )
```

Default destructor for clearing out memory.

5.36.3 Member Function Documentation

5.36.3.1 operator()() [1/2]

```
template<class T >
T & Matrix< T >::operator() (
             int i,
             int j )
```

Access operator for the matrix element at row i and column j (e.g., aij = A(i,j))

5.36.3.2 operator()() [2/2]

```
template<class T >
T Matrix< T >::operator() (
             int i,
             int j ) const
```

Constant access operator for the the matrix element at row i and column j.

5.36.3.3 operator=()

```
template<class T >
Matrix< T > & Matrix< T >::operator= (
             const Matrix< T > & M )
```

Equals operator for setting one matrix equal to another matrix.

5.36.3.4 set_size()

```
template<class T >
void Matrix< T >::set_size (
             int i,
             int j )
```

Function to set/change the size of a matrix to i rows and j columns.

5.36.3.5 zeros()

```
template<class T >
void Matrix< T >::zeros ( )
```

Function to set/change all values in a matrix to zeros.

5.36.3.6 edit()

```
template<class T>
void Matrix< T >::edit (
             int i,
             int j,
             T value )
```

Function to set/change the element of a matrix at row i and column j to given value.

5.36.3.7 rows()

```
template<class T >
int Matrix< T >::rows ( )
```

Function to return the number of rows in a given matrix.

5.36.3.8 columns()

```
template<class T >
int Matrix< T >::columns ( )
```

Function to return the number of columns in a matrix.

5.36.3.9 determinate()

```
template<class T >
T Matrix< T >::determinate ( )
```

Function to compute the determinate of a matrix and return that value.

5.36.3.10 norm()

```
template<class T >
T Matrix< T >::norm ( )
```

Function to compute the L2-norm of a matrix and return that value.

5.36.3.11 sum()

```
template<class T >
T Matrix< T >::sum ( )
```

Function to compute the sum of all elements in a matrix and return that value.

5.36.3.12 inner_product()

```
template<class T >
T Matrix< T >::inner_product (
             const Matrix< T > & x )
```

Function to compute the inner product between this matrix and matrix x.

5.36.3.13 cofactor()

```
template<class T >
Matrix< T > & Matrix< T >::cofactor (
             const Matrix< T > & M )
```

Function to convert this matrix to a cofactor matrix of the given matrix M.

5.36.3.14 operator+() [1/2]

```
template<class T >
Matrix< T > Matrix< T >::operator+ (
             const Matrix< T > & M )
```

Operator to add this matrix and matrix M and return the new matrix result.

5.36.3.15 operator-() [1/2]

```
template<class T >
Matrix< T > Matrix< T >::operator- (
             const Matrix< T > & M )
```

Operator to subtract this matrix and matrix M and return the new matrix result.

5.36.3.16 operator*() [1/2]

```
template<class T>
Matrix< T > Matrix< T >::operator* (
             const T a )
```

Operator to multiply this matrix by a scalar T return the new matrix result.

5.36.3.17 operator/()

```
template<class T>
Matrix< T > Matrix< T >::operator/ (
             const T a )
```

Operator to divide this matrix by a scalar T and return the new matrix result.

5.36.3.18 operator+() [2/2]

```
template<class T>
Matrix< T > Matrix< T >::operator+ (
             const T a )
```

Operator to add this matrix to a scalar T and return the new matrix result.

5.36.3.19 operator-() [2/2]

```
template<class T>
Matrix< T > Matrix< T >::operator- (
             const T a )
```

Operator to subtract this matrix to a scalar T and return the new matrix result.

5.36.3.20 operator*() [2/2]

```
template<class T>
Matrix< T > Matrix< T >::operator* (
             const Matrix< T > & M )
```

Operator to multiply this matrix and matrix M and return the new matrix result.

5.36.3.21 outer_product()

```
template<class T >
Matrix< T > Matrix< T >::outer_product (
             const Matrix< T > & M )
```

Operator to perform an outer product between this and M and return result.

5.36.3.22 transpose()

```
template<class T >
Matrix< T > & Matrix< T >::transpose (
             const Matrix< T > & M )
```

Function to convert this matrix to the transpose of the given matrix M

5.36.3.23 transpose_multiply()

```
template<class T >
Matrix< T > & Matrix< T >::transpose_multiply (
             const Matrix< T > & MT,
             const Matrix< T > & v )
```

Function to convert this matrix into the result of the given matrix M transposed and multiplied by the other given matrix v.

5.36.3.24 adjoint()

```
template<class T >
Matrix< T > & Matrix< T >::adjoint (
             const Matrix< T > & M )
```

Function to convert this matrix to the adjoint of the given matrix.

5.36.3.25 inverse()

```
template<class T >
Matrix< T > & Matrix< T >::inverse (
             const Matrix< T > & M )
```

Function to convert this matrix to the inverse of the given matrix.

5.36.3.26 Display()

```
template<class T >
void Matrix< T >::Display (
             const std::string Name )
```

Function to display the contents of this matrix given a Name for the matrix.

5.36.3.27 tridiagonalSolve()

```
template<class T >
Matrix< T > & Matrix< T >::tridiagonalSolve (
             const Matrix< T > & A,
             const Matrix< T > & b )
```

Function to solve Ax=b for x if A is symmetric, tridiagonal (this->x)

5.36.3.28 ladshawSolve()

```
template<class T >
Matrix< T > & Matrix< T >::ladshawSolve (
             const Matrix< T > & A,
             const Matrix< T > & d )
```

Function to solve Ax=d for x if A is non-symmetric, tridiagonal (this->x)

5.36.3.29 tridiagonalFill()

```
template<class T>
Matrix< T > & Matrix< T >::tridiagonalFill (
             const T A,
             const T B,
             const T C,
             bool Spherical )
```

Function to fill in this matrix with coefficients A, B, and C to form a tridiagonal matrix.

This function fills in the diagonal elements of a square matrix with coefficient B, upper diagonal with C, and lower diagonal with A. The boolean will apply a transformation to those coefficients, if the problem happens to stem from 1-D diffusion in spherical coordinates.

5.36.3.30 naturalLaplacian3D()

```
template<class T >
Matrix< T > & Matrix< T >::naturalLaplacian3D (
             int m )
```

Function to fill out this matrix with coefficients from a 3D Laplacian function.

This function will fill out the coefficients of the matrix with the coefficients that stem from discretizing a 3D Laplacian on a natural grid with 2nd order finite differences.

5.36.3.31 sphericalBCFill()

```
template<class T>
Matrix< T > & Matrix< T >::sphericalBCFill (
             int node,
             const T coeff,
             T variable )
```

Function to fill out a column matrix with spherical specific boundary conditions.

This function will fille out a column matrix with zeros at all nodes expect for the node indicated. That node's value will be the product of the node id with the coeff and variable values given.

5.36.3.32 ConstantICFill()

```
template<class T>
Matrix< T > & Matrix< T >::ConstantICFill (
             const T IC )
```

Function to set all values in a column matrix to a given constant.

5.36.3.33 SolnTransform()

```
template<class T >
Matrix< T > & Matrix< T >::SolnTransform (
            const Matrix< T > & A,
            bool Forward )
```

Function to transform the values in a column matrix from cartesian to spherical coordinates.

5.36.3.34 sphericalAvg()

```
template<class T >
T Matrix< T >::sphericalAvg (
            double radius,
            double dr,
            double bound,
            bool Dirichlet )
```

Function to compute a spatial average of this column matrix in spherical coordinates.

This function is used to compute an average value of a variable, represented in this column matrix, by integrating over the domain of the sphere. (Assumes you have variable value at center node)

Parameters

radius	radius of the sphere
dr	space between each node
bound	value of the variable at the boundary
Dirichlet	True if problem has a Dirichlet BC, False if Neumann

5.36.3.35 IntegralAvg()

```
template<class T >
T Matrix< T >::IntegralAvg (
            double radius,
            double dr,
            double bound,
            bool Dirichlet )
```

Function to compute a spatial average of this column matrix in spherical coordinates.

This function is used to compute an average value of a variable, represented in this column matrix, by integrating over the domain of the sphere. (Assumes you DO NOT have variable value at center node)

Parameters

radius	radius of the sphere
dr	space between each node
bound	value of the variable at the boundary
Dirichlet	True if problem has a Dirichlet BC, False if Neumann

5.36.3.36 IntegralTotal()

```
template<class T >
T Matrix< T >::IntegralTotal (
              double dr,
              double bound,
              bool Dirichlet )
```

Function to compute a spatial total of this column matrix in spherical coordinates.

This function is used to compute an average value of a variable, represented in this column matrix, by integrating over the domain of the sphere. (Assumes you DO NOT have variable value at center node)

Parameters

dr	space between each node
bound	value of the variable at the boundary
Dirichlet	True if problem has a Dirichlet BC, False if Neumann

5.36.3.37 tridiagonalVectorFill()

```
template<class T>
Matrix< T > & Matrix< T >::tridiagonalVectorFill (
              const std::vector< T > & A,
              const std::vector< T > & B,
              const std::vector< T > & C )
```

Function to fill in this matrix, in tridiagonal fashion, using the vectors of coefficients.

5.36.3.38 columnVectorFill()

```
template<class T>
Matrix< T > & Matrix< T >::columnVectorFill (
              const std::vector< T > & A )
```

Function to fill in a column matrix with the values of the given vector object.

5.36.3.39 columnProjection()

```
template<class T >
Matrix< T > & Matrix< T >::columnProjection (
              const Matrix< T > & b,
              const Matrix< T > & b_old,
              const double dt,
              const double dt_old )
```

Function to project a column matrix solution in time based on older state vectors.

This function is used in finch.h to form Matrix u_star. It uses the size of the current step and old step, dt and dt_old respectively, to form an approximation for the next state. The current state and olde state of the variables are passed as b and b_old respectively.

5.36.3.40 dirichletBCFill()

```
template<class T>
Matrix< T > & Matrix< T >::dirichletBCFill (
             int node,
             const T coeff,
             T variable )
```

Function to fill in a column matrix with all zeros except at the given node.

Similar to sphericalBCFill, this function will set the values of all elements in the column matrix to zero except at the given node, where the value is set to the product of coeff and variable. This is often used to set BCs in finch.h or other related files/simulations.

5.36.3.41 diagonalSolve()

```
template<class T >
Matrix< T > & Matrix< T >::diagonalSolve (
             const Matrix< T > & D,
             const Matrix< T > & v )
```

Function to solve the system Dx=v for x given that D is diagonal (this->x)

5.36.3.42 upperTriangularSolve()

```
template<class T >
Matrix< T > & Matrix< T >::upperTriangularSolve (
             const Matrix< T > & U,
             const Matrix< T > & v )
```

Function to solve the system Ux=v for x given that U is upper Triagular (this->x)

5.36.3.43 lowerTriangularSolve()

```
template<class T >
Matrix< T > & Matrix< T >::lowerTriangularSolve (
             const Matrix< T > & L,
             const Matrix< T > & v )
```

Function to solve the system Lx=v for x given that L is lower Triagular (this->x)

5.36.3.44 upperHessenberg2Triangular()

```
template<class T >
Matrix< T > & Matrix< T >::upperHessenberg2Triangular (
             Matrix< T > & b )
```

Function to convert this square matrix to upper Triangular (assuming this is upper Hessenberg)

During this transformation, a column vector (b) is also being transformed to represent the BCs in a linear system. This algorithm uses Givens Rotations to efficiently convert the upper Hessenberg matrix to an upper triangular matrix.

5.36.3.45 lowerHessenberg2Triangular()

```
template<class T >
Matrix< T > & Matrix< T >::lowerHessenberg2Triangular (
             Matrix< T > & b )
```

Function to convert this square matrix to lower Triangular (assuming this is lower Hessenberg)

During this transformation, a column vector (b) is also being transformed to represent the BCs in a linear system. This algorithm uses Givens Rotations to efficiently convert the lower Hessenberg matrix to an lower triangular matrix.

5.36.3.46 upperHessenbergSolve()

```
template<class T >
Matrix< T > & Matrix< T >::upperHessenbergSolve (
             const Matrix< T > & H,
             const Matrix< T > & v )
```

Function to solve the system Hx=v for x given that H is upper Hessenberg (this->x)

5.36.3.47 lowerHessenbergSolve()

```
template<class T >
Matrix< T > & Matrix< T >::lowerHessenbergSolve (
             const Matrix< T > & H,
             const Matrix< T > & v )
```

Function to solve the system Hx=v for x given that H is lower Hessenberg (this->x)

5.36.3.48 qrSolve()

```
template<class T >
Matrix< T > & Matrix< T >::qrSolve (
             const Matrix< T > & M,
             const Matrix< T > & b )
```

Function to solve the system Mx=b using QR factorization for x given that M is invertable.

5.36.3.49 columnExtract()

```
template<class T >
Matrix< T > & Matrix< T >::columnExtract (
             int j,
             const Matrix< T > & M )
```

Function to set this column matrix to the jth column of the given matrix M.

5.36.3.50 rowExtract()

```
template<class T >
Matrix< T > & Matrix< T >::rowExtract (
             int i,
             const Matrix< T > & M )
```

Function to set this row matrix to the ith row of the given matrix M.

5.36.3.51 columnReplace()

```
template<class T >
Matrix< T > & Matrix< T >::columnReplace (
             int j,
             const Matrix< T > & v )
```

Function to this matrices' jth column with the given column matrix v.

5.36.3.52 rowReplace()

```
template<class T >
Matrix< T > & Matrix< T >::rowReplace (
             int i,
             const Matrix< T > & v )
```

Function to this matrices' ith row with the given row matrix v.

5.36.3.53 rowShrink()

```
template<class T >
void Matrix< T >::rowShrink ( )
```

Function to delete the last row of this matrix.

5.36.3.54 columnShrink()

```
template<class T >
void Matrix< T >::columnShrink ( )
```

Function to delete the last column of this matrix.

5.36.3.55 rowExtend()

```
template<class T >
void Matrix< T >::rowExtend (
             const Matrix< T > & v )
```

Function to add the row matrix v to the end of this matrix.

5.36.3.56 columnExtend()

```
template<class T >
void Matrix< T >::columnExtend (
             const Matrix< T > & v )
```

Function to add the column matrix v to the end of this matrix.

5.36.4 Member Data Documentation

5.36.4.1 num_rows

```
template<class T>
int Matrix< T >::num_rows  [protected]
```

Number of rows of the matrix.

5.36.4.2 num_cols

```
template<class T>
int Matrix< T >::num_cols  [protected]
```

Number of columns of the matrix.

5.36.4.3 Data

```
template<class T>
std::vector<T> Matrix< T >::Data  [protected]
```

Storage vector for the elements of the matrix.

The documentation for this class was generated from the following file:

- macaw.h

5.37 MIXED_GAS Struct Reference

Data structure holding information necessary for computing mixed gas properties.

```
#include <egret.h>
```

Public Attributes

- int N

    Given: Total number of gas species.
- bool CheckMolefractions = true

    Given: True = Check Molefractions for errors.
- double total_pressure

    Given: Total gas pressure (kPa)
- double gas_temperature

    Given: Gas temperature (K)
- double velocity

    Given: Gas phase velocity (cm/s)
- double char_length

    Given: Characteristic Length (cm)
- std::vector< double > molefraction

    Given: Gas molefractions of each species (-)
- double total_density

    Calculated: Total gas density (g/cm^3) {use RE3}.
- double total_dyn_vis

    Calculated: Total dynamic viscosity (g/cm/s)
- double kinematic_viscosity

    Calculated: Kinematic viscosity (cm^2/s)
- double total_molecular_weight

    Calculated: Total molecular weight (g/mol)
- double total_specific_heat

    Calculated: Total specific heat (J/g/K)
- double Reynolds

    Calculated: Value of the Reynold's number (-)
- Matrix< double > binary_diffusion

    Calculated: Tensor matrix of binary gas diffusivities (cm^2/s)
- std::vector< PURE_GAS > species_dat

    Vector of the pure gas info of all species.

5.37.1 Detailed Description

Data structure holding information necessary for computing mixed gas properties.

C-style object holding the mixed gas information necessary for performing gas dynamic simulations. This object works in conjunction with the calculate_variables function and uses the kinetic theory of gases to estimate mixed gas properties.

5.37.2 Member Data Documentation

5.37.2.1 N

```
int MIXED_GAS::N
```

Given: Total number of gas species.

5.37.2.2 CheckMolefractions

```
bool MIXED_GAS::CheckMolefractions = true
```

Given: True = Check Molefractions for errors.

5.37.2.3 total_pressure

```
double MIXED_GAS::total_pressure
```

Given: Total gas pressure (kPa)

5.37.2.4 gas_temperature

```
double MIXED_GAS::gas_temperature
```

Given: Gas temperature (K)

5.37.2.5 velocity

```
double MIXED_GAS::velocity
```

Given: Gas phase velocity (cm/s)

5.37.2.6 char_length

```
double MIXED_GAS::char_length
```

Given: Characteristic Length (cm)

5.37.2.7 molefraction

```
std::vector<double> MIXED_GAS::molefraction
```

Given: Gas molefractions of each species (-)

5.37.2.8 total_density

```
double MIXED_GAS::total_density
```

Calculated: Total gas density (g/cm^3) {use RE3}.

5.37.2.9 total_dyn_vis

```
double MIXED_GAS::total_dyn_vis
```

Calculated: Total dynamic viscosity (g/cm/s)

5.37.2.10 kinematic_viscosity

```
double MIXED_GAS::kinematic_viscosity
```

Calculated: Kinematic viscosity (cm^2/s)

5.37.2.11 total_molecular_weight

```
double MIXED_GAS::total_molecular_weight
```

Calculated: Total molecular weight (g/mol)

5.37.2.12 total_specific_heat

```
double MIXED_GAS::total_specific_heat
```

Calculated: Total specific heat (J/g/K)

5.37.2.13 Reynolds

```
double MIXED_GAS::Reynolds
```

Calculated: Value of the Reynold's number (-)

5.37.2.14 binary_diffusion

```
Matrix<double> MIXED_GAS::binary_diffusion
```

Calculated: Tensor matrix of binary gas diffusivities (cm^2/s)

5.37.2.15 species_dat

```
std::vector<PURE_GAS> MIXED_GAS::species_dat
```

Vector of the pure gas info of all species.

The documentation for this struct was generated from the following file:

- egret.h

5.38 Molecule Class Reference

C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

```
#include <mola.h>
```

Public Member Functions

- Molecule ()

  Default Constructor (builds an empty molecule object)
- ∼Molecule ()

  Default Destructor (clears out memory)
- Molecule (int charge, double enthalpy, double entropy, double energy, bool HS, bool G, std::string Phase, std::string Name, std::string Formula, std::string lin_formula)

  Construct any molecule from the available information.
- void Register (int charge, double enthalpy, double entropy, double energy, bool HS, bool G, std::string Phase, std::string Name, std::string Formula, std::string lin_formula)

  Function to register this molecule from the available information.
- void Register (std::string formula)

  Function to register this molecule based on the given formula (if formula is in library)
- void setFormula (std::string form)

  Sets the formula for a molecule.
- void calculateMolarWeight ()

  Forces molecule to calculate its molar weight.
- void calculateMolarVolume ()

  Force molecule to calculate van der Waals volume.
- void calculateMolarArea ()

  Force molecule to calculate van der Waals area.
- void setMolarWeigth (double mw)

  Set the molar weight of species to a constant.
- void setMolarVolume (double v)

  Set the van der Waals volume of the species to a constant.
- void setMolarArea (double a)

  Set the van der Waals area of the species to a constant.
- void editCharge (int c)

  Change the ionic charge of a molecule.
- void editOneOxidationState (int state, std::string Symbol)

  Change oxidation state of one of the given atoms (always first match found)
- void editAllOxidationStates (int state, std::string Symbol)

  Change oxidation state of all of the given atoms.

- void calculateAvgOxiState (std::string Symbol)

  Function to calculate the average oxidation state of the atoms.
- void editEnthalpy (double enthalpy)

  Edit the molecules formation enthalpy (J/mol)
- void editEntropy (double entropy)

  Edit the molecules formation entropy (J/K/mol)
- void editHS (double H, double S)

  Edit both formation enthalpy and entropy.
- void editEnergy (double energy)

  Edit Gibb's formation energy.
- void removeOneAtom (std::string Symbol)

  Removes one atom of the symbol given (always the first atom found)
- void removeAllAtoms (std::string Symbol)

  Removes all atoms of the symbol given.
- int Charge ()

  Return the charge of the molecule.
- double MolarWeight ()

  Return the molar weight of the molecule.
- double MolarVolume ()

  Return the van der Waals volume of the molecule.
- double MolarArea ()

  Return the van der Waals area of the molecule.
- bool HaveHS ()

  Returns true if enthalpy and entropy are known.
- bool HaveEnergy ()

  Returns true if the Gibb's energy is known.
- bool isRegistered ()

  Returns true if the molecule has been registered.
- double Enthalpy ()

  Return the formation enthalpy of the molecule.
- double Entropy ()

  Return the formation entropy of the molecule.
- double Energy ()

  Return the Gibb's formation energy of the molecule.
- std::string MoleculeName ()

  Return the common name of the molecule.
- std::string MolecularFormula ()

  Return the molecular formula of the molecule.
- std::string MoleculePhase ()

  Return the phase of the molecule.
- int MoleculePhaseID ()

  Return the enum phase ID of the molecule.
- std::vector< Atom > & getAtoms ()

  Return reference to the vector of atoms.
- void DisplayInfo ()

  Function to display molecule information.

Protected Attributes

- int charge

    Ionic charge of the molecule - specified.
- double molar_weight

    Molar weight of the molecule (g/mol) - determined from atoms or specified.
- double molar_volume

    van der Waals Volume of the molecule (cubic angstroms) - determined from atoms or specified
- double molar_area

    van der Waals Area of the molecule (square angstroms) - determined from atoms or specified
- double formation_enthalpy

    Enthalpy of formation of the molecule (J/mol) - constant.
- double formation_entropy

    Entropy of formation of the molecule (J/K/mol) - constant.
- double formation_energy

    Gibb's energy of formation (J/mol) - given.
- std::string Phase

    Phase of the molecule (i.e. Solid, Liquid, Aqueous, Gas...)
- int PhaseID

    Phase ID of the molecule (from the enum)
- std::vector< Atom > atoms

    Atoms which make up the molecule - based on Formula.

Private Attributes

- std::string Name

    Name of the Molecule - Common Name (i.e. H2O = Water)
- std::string Formula

    Formula for the molecule - specified (i.e. H2O)
- bool haveG

    True = given Gibb's energy of formation.
- bool haveHS

    True = give enthalpy and entropy of formation.
- bool registered

    True = the object was registered.

5.38.1 Detailed Description

C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

C++ Class Object that stores information and certain operations associated with molecules. Registered molecules are built up from their respective atoms so that the molecule can keep track of information such as molecular weigth and oxidation states. Primarily, this object is used in conjunction with shark.h to formulate the system of equations necessary for solving speciation type problems in aqueous systems. However, this object is generalized enough to be of use in RedOx calculations, reaction formulation, and molecular transformations.

All information for a molecule should be initialized prior to performing operations with or on the object. There are several molecules already defined for construction by the formulas listed at the top of this section.

5.38.2 Constructor & Destructor Documentation

5.38.2.1 Molecule() [1/2]

```
Molecule::Molecule ( )
```

Default Constructor (builds an empty molecule object)

5.38.2.2 ∼Molecule()

```
Molecule::∼Molecule ( )
```

Default Destructor (clears out memory)

5.38.2.3 Molecule() [2/2]

```
Molecule::Molecule (
             int charge,
             double enthalpy,
             double entropy,
             double energy,
             bool HS,
             bool G,
             std::string Phase,
             std::string Name,
             std::string Formula,
             std::string lin_formula )
```

Construct any molecule from the available information.

This constructor will build a user defined custom molecule.

Parameters

charge	the ionic charge of the molecule
enthalpy	the standard formation enthalpy of the molecule (J/mol)
entropy	the standard formation entropy of the molecule (J/K/mol)
energy	the standard Gibb's Free Energy of formation of the molecule (J/mol)
HS	boolean to be set to true if enthalpy and entropy were given
G	boolean to be set to true if the energy was given
Phase	string denoting molecule's phase (i.e., Liquid, Aqueous, Gas, Solid)
Name	string denoting the common name of the molecule (i.e., H2O -> Water)
Formula	string denoting the formula by which the molecule is referened (i.e., Cl - (aq))
lin_formula	string denoting all the atoms in the molecule (i.e., UO2(OH)2 -> UO4H2)

5.38.3 Member Function Documentation

5.38.3.1 Register() [1/2]

```
void Molecule::Register (
            int charge,
            double enthalpy,
            double entropy,
            double energy,
            bool HS,
            bool G,
            std::string Phase,
            std::string Name,
            std::string Formula,
            std::string lin_formula )
```

Function to register this molecule from the available information.

This function will build a user defined custom molecule.

Parameters

charge	the ionic charge of the molecule
enthalpy	the standard formation enthalpy of the molecule (J/mol)
entropy	the standard formation entropy of the molecule (J/K/mol)
energy	the standard Gibb's Free Energy of formation of the molecule (J/mol)
HS	boolean to be set to true if enthalpy and entropy were given
G	boolean to be set to true if the energy was given
Phase	string denoting molecule's phase (i.e., Liquid, Aqueous, Gas, Solid)
Name	string denoting the common name of the molecule (i.e., H2O -> Water)
Formula	string denoting the formula by which the molecule is referened (i.e., Cl - (aq))
lin_formula	string denoting all the atoms in the molecule (i.e., UO2(OH)2 -> UO4H2)

5.38.3.2 Register() [2/2]

```
void Molecule::Register (
            std::string formula )
```

Function to register this molecule based on the given formula (if formula is in library)

This function will create this molecule object from the given formula, but only if that formula is already registered in the library. See the top of this class section for a list of all currently registered formulas.

Note

The formula is checked against a known set of molecules inside of the registration function If the formula is unknown, an error will print to the screen. Unknown molecules should be registered using the full registration function from above. The library can only be added to by a going in and editing the source code of the mola.cpp file. However, this is a relatively simple task.

5.38.3.3 setFormula()

```
void Molecule::setFormula (
             std::string form )
```

Sets the formula for a molecule.

5.38.3.4 calculateMolarWeight()

```
void Molecule::calculateMolarWeight ( )
```

Forces molecule to calculate its molar weight.

5.38.3.5 calculateMolarVolume()

```
void Molecule::calculateMolarVolume ( )
```

Force molecule to calculate van der Waals volume.

5.38.3.6 calculateMolarArea()

```
void Molecule::calculateMolarArea ( )
```

Force molecule to calculate van der Waals area.

5.38.3.7 setMolarWeigth()

```
void Molecule::setMolarWeigth (
             double mw )
```

Set the molar weight of species to a constant.

5.38.3.8 setMolarVolume()

```
void Molecule::setMolarVolume (
             double v )
```

Set the van der Waals volume of the species to a constant.

5.38.3.9 setMolarArea()

```
void Molecule::setMolarArea (
             double a )
```

Set the van der Waals area of the species to a constant.

5.38.3.10 editCharge()

```
void Molecule::editCharge (
             int c )
```

Change the ionic charge of a molecule.

5.38.3.11 editOneOxidationState()

```
void Molecule::editOneOxidationState (
             int state,
             std::string Symbol )
```

Change oxidation state of one of the given atoms (always first match found)

This function will search the list of Atoms that make up the Molecule for the given atomic Symbol. It will change the oxidation state of the first found matching atom with the given state.

5.38.3.12 editAllOxidationStates()

```
void Molecule::editAllOxidationStates (
             int state,
             std::string Symbol )
```

Change oxidation state of all of the given atoms.

This function will search the list of Atoms that make up the Molecule for the given atomic Symbol. It will change the oxidation state of all found matching atoms with the given state.

5.38.3.13 calculateAvgOxiState()

```
void Molecule::calculateAvgOxiState (
             std::string Symbol )
```

Function to calculate the average oxidation state of the atoms.

This function search the atoms in the molecule for the matching atomic Symbol. It then looks at all oxidation states of that atom in the molecule and then sets all the oxidation states of that atom to the average value calculated.

5.38.3.14 editEnthalpy()

```
void Molecule::editEnthalpy (
             double enthalpy )
```

Edit the molecules formation enthalpy (J/mol)

5.38.3.15 editEntropy()

```
void Molecule::editEntropy (
             double entropy )
```

Edit the molecules formation entropy (J/K/mol)

5.38.3.16 editHS()

```
void Molecule::editHS (
             double H,
             double S )
```

Edit both formation enthalpy and entropy.

This function will change or set the values for formation enthalpy (J/mol) and formation entropy (J/K/mol) based on the given values.

Parameters

H	formation enthalpy (J/mol)
S	formation entropy (J/K/mol)

5.38.3.17 editEnergy()

```
void Molecule::editEnergy (
             double energy )
```

Edit Gibb's formation energy.

5.38.3.18 removeOneAtom()

```
void Molecule::removeOneAtom (
             std::string Symbol )
```

Removes one atom of the symbol given (always the first atom found)

5.38.3.19 removeAllAtoms()

```
void Molecule::removeAllAtoms (
             std::string Symbol )
```

Removes all atoms of the symbol given.

5.38.3.20 Charge()

```
int Molecule::Charge ( )
```

Return the charge of the molecule.

5.38.3.21 MolarWeight()

```
double Molecule::MolarWeight ( )
```

Return the molar weight of the molecule.

5.38.3.22 MolarVolume()

```
double Molecule::MolarVolume ( )
```

Return the van der Waals volume of the molecule.

5.38.3.23 MolarArea()

```
double Molecule::MolarArea ( )
```

Return the van der Waals area of the molecule.

5.38.3.24 HaveHS()

```
bool Molecule::HaveHS ( )
```

Returns true if enthalpy and entropy are known.

5.38.3.25 HaveEnergy()

```
bool Molecule::HaveEnergy ( )
```

Returns true if the Gibb's energy is known.

5.38.3.26 isRegistered()

```
bool Molecule::isRegistered ( )
```

Returns true if the molecule has been registered.

5.38.3.27 Enthalpy()

```
double Molecule::Enthalpy ( )
```

Return the formation enthalpy of the molecule.

5.38.3.28 Entropy()

```
double Molecule::Entropy ( )
```

Return the formation entropy of the molecule.

5.38.3.29 Energy()

```
double Molecule::Energy ( )
```

Return the Gibb's formation energy of the molecule.

5.38.3.30 MoleculeName()

```
std::string Molecule::MoleculeName ( )
```

Return the common name of the molecule.

5.38.3.31 MolecularFormula()

```
std::string Molecule::MolecularFormula ( )
```

Return the molecular formula of the molecule.

5.38.3.32 MoleculePhase()

```
std::string Molecule::MoleculePhase ( )
```

Return the phase of the molecule.

5.38.3.33 MoleculePhaseID()

```
int Molecule::MoleculePhaseID ( )
```

Return the enum phase ID of the molecule.

5.38.3.34 getAtoms()

```
std::vector<Atom>& Molecule::getAtoms ( )
```

Return reference to the vector of atoms.

5.38.3.35 DisplayInfo()

```
void Molecule::DisplayInfo ( )
```

Function to display molecule information.

5.38.4 Member Data Documentation

5.38.4.1 charge

```
int Molecule::charge  [protected]
```

Ionic charge of the molecule - specified.

5.38.4.2 molar_weight

```
double Molecule::molar_weight  [protected]
```

Molar weight of the molecule (g/mol) - determined from atoms or specified.

5.38.4.3 molar_volume

```
double Molecule::molar_volume  [protected]
```

van der Waals Volume of the molecule (cubic angstroms) - determined from atoms or specified

5.38.4.4 molar_area

```
double Molecule::molar_area  [protected]
```

van der Waals Area of the molecule (square angstroms) - determined from atoms or specified

5.38.4.5 formation_enthalpy

```
double Molecule::formation_enthalpy  [protected]
```

Enthalpy of formation of the molecule (J/mol) - constant.

5.38.4.6 formation_entropy

```
double Molecule::formation_entropy  [protected]
```

Entropy of formation of the molecule (J/K/mol) - constant.

5.38.4.7 formation_energy

```
double Molecule::formation_energy  [protected]
```

Gibb's energy of formation (J/mol) - given.

5.38.4.8 Phase

```
std::string Molecule::Phase  [protected]
```

Phase of the molecule (i.e. Solid, Liquid, Aqueous, Gas...)

5.38.4.9 PhaseID

```
int Molecule::PhaseID  [protected]
```

Phase ID of the molecule (from the enum)

5.38.4.10 atoms

```
std::vector<Atom> Molecule::atoms  [protected]
```

Atoms which make up the molecule - based on Formula.

5.38.4.11 Name

```
std::string Molecule::Name  [private]
```

Name of the Molecule - Common Name (i.e. H2O = Water)

5.38.4.12 Formula

```
std::string Molecule::Formula  [private]
```

Formula for the molecule - specified (i.e. H2O)

5.38.4.13 haveG

```
bool Molecule::haveG  [private]
```

True = given Gibb's energy of formation.

5.38.4.14 haveHS

```
bool Molecule::haveHS  [private]
```

True = give enthalpy and entropy of formation.

5.38.4.15 registered

```
bool Molecule::registered  [private]
```

True = the object was registered.

The documentation for this class was generated from the following file:

- mola.h

5.39 MONKFISH_DATA Struct Reference

Primary data structure for running MONKFISH.

```
#include <monkfish.h>
```

Public Attributes

- unsigned long int total_steps = 0

    Total number of steps taken by the algorithm (iterations and time steps)
- double time_old = 0.0

    Old value of time in the simulation (hrs)
- double time = 0.0

    Current value of time in the simulation (hrs)
- bool Print2File = true

    True = results to .txt; False = no printing.
- bool Print2Console = true

    True = results to console; False = no printing.
- bool DirichletBC = true

    False = uses film mass transfer for BC, True = Dirichlet BC.
- bool NonLinear = false

    False = Solve directly, True = Solve iteratively.
- bool haveMinMax = false

    True = know min and max fiber density, False = only know avg density (Used in ICs)
- bool MultiScale = true

    True = solve single fiber model at nodes, False = solve equilibrium at nodes.
- int level = 2

    Level of coupling between multiple scales (default = 2)
- double t_counter = 0.0

    Counter for the time output.
- double t_print

    Print output at every t_print time (hrs)
- int NumComp

    Number of species to track.
- double end_time

    Units: hours.
- double total_sorption_old

    Old total adsorption per mass of woven nest (mg/g)
- double total_sorption

    Current total adsorption per mass woven nest (mg/g)
- double single_fiber_density

    Units: g/L.
- double avg_fiber_density

    Units: g/L (Used in ICs)
- double max_fiber_density

    Units: g/L (Used in ICs)
- double min_fiber_density

    Units: g/L (Used in ICs)
- double max_porosity

    Units: -.
- double min_porosity

    Units: -.
- double domain_diameter

    Nominal diameter of the woven fiber ball - Units: cm.
- FILE $*$ Output

    Output file pointer for printing to text file.
- double($*$ eval_eps)(int i, int l, const void $*$user_data)

Function pointer to evaluate the porosity of the woven bundle of fibers.

- double(∗ eval_rho)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the fiber density in the domain.
- double(∗ eval_Dex)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the interparticle diffusivity.
- double(∗ eval_ads)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the adsorption strength for the macro-scale.
- double(∗ eval_Ret)(int i, int l, const void ∗user_data)

  Function pointer to evaluate the retardation coefficient for the macro-scale.
- double(∗ eval_Cex)(int i, const void ∗user_data)

  Function pointer to evaluate the exterior concentration for the domain.
- double(∗ eval_kf)(int i, const void ∗user_data)

  Function pointer to evalutate the film mass transfer coefficient for the macro-scale.
- const void ∗ user_data

  User supplied data function to evaluate the function pointers (Default = MONKFISH_DATA)
- std::vector< FINCH_DATA > finch_dat

  FINCH data structures to solve each species interparticle diffusion equation.
- std::vector< MONKFISH_PARAM > param_dat

  MONKFISH parameter data structure for each species adsorbing.
- std::vector< DOGFISH_DATA > dog_dat

  DOGFISH data structures for each node in the macro-scale problem.

5.39.1 Detailed Description

Primary data structure for running MONKFISH.

C-style object holding simulation information for MONKFISH as well as common system parameters like fiber density, fiber diameter, fiber length, etc. This object also contains function pointers to different parameter evaluation functions that can be changed to suit a particular problem. Default functions will be given, so not every user needs to override these functions. This structure also contains vectors of other objects including FINCH and DOGFISH objects to resolve the diffusion physics at both the macro- and micro-scale.

5.39.2 Member Data Documentation

5.39.2.1 total_steps

```
unsigned long int MONKFISH_DATA::total_steps = 0
```

Total number of steps taken by the algorithm (iterations and time steps)

5.39.2.2 time_old

```
double MONKFISH_DATA::time_old = 0.0
```

Old value of time in the simulation (hrs)

5.39.2.3 time

```
double MONKFISH_DATA::time = 0.0
```

Current value of time in the simulation (hrs)

5.39.2.4 Print2File

```
bool MONKFISH_DATA::Print2File = true
```

True = results to .txt; False = no printing.

5.39.2.5 Print2Console

```
bool MONKFISH_DATA::Print2Console = true
```

True = results to console; False = no printing.

5.39.2.6 DirichletBC

```
bool MONKFISH_DATA::DirichletBC = true
```

False = uses film mass transfer for BC, True = Dirichlet BC.

5.39.2.7 NonLinear

```
bool MONKFISH_DATA::NonLinear = false
```

False = Solve directly, True = Solve iteratively.

5.39.2.8 haveMinMax

```
bool MONKFISH_DATA::haveMinMax = false
```

True = know min and max fiber density, False = only know avg density (Used in ICs)

5.39.2.9 MultiScale

```
bool MONKFISH_DATA::MultiScale = true
```

True = solve single fiber model at nodes, False = solve equilibrium at nodes.

5.39.2.10 level

```
int MONKFISH_DATA::level = 2
```

Level of coupling between multiple scales (default = 2)

5.39.2.11 t_counter

```
double MONKFISH_DATA::t_counter = 0.0
```

Counter for the time output.

5.39.2.12 t_print

```
double MONKFISH_DATA::t_print
```

Print output at every t_print time (hrs)

5.39.2.13 NumComp

```
int MONKFISH_DATA::NumComp
```

Number of species to track.

5.39.2.14 end_time

```
double MONKFISH_DATA::end_time
```

Units: hours.

5.39.2.15 total_sorption_old

```
double MONKFISH_DATA::total_sorption_old
```

Old total adsorption per mass of woven nest (mg/g)

5.39.2.16 total_sorption

```
double MONKFISH_DATA::total_sorption
```

Current total adsorption per mass woven nest (mg/g)

5.39.2.17 single_fiber_density

```
double MONKFISH_DATA::single_fiber_density
```

Units: g/L.

5.39.2.18 avg_fiber_density

```
double MONKFISH_DATA::avg_fiber_density
```

Units: g/L (Used in ICs)

5.39.2.19 max_fiber_density

```
double MONKFISH_DATA::max_fiber_density
```

Units: g/L (Used in ICs)

5.39.2.20 min_fiber_density

```
double MONKFISH_DATA::min_fiber_density
```

Units: g/L (Used in ICs)

5.39.2.21 max_porosity

```
double MONKFISH_DATA::max_porosity
```

Units: -.

5.39.2.22 min_porosity

```
double MONKFISH_DATA::min_porosity
```

Units: -.

5.39.2.23 domain_diameter

```
double MONKFISH_DATA::domain_diameter
```

Nominal diameter of the woven fiber ball - Units: cm.

5.39.2.24 Output

```
FILE* MONKFISH_DATA::Output
```

Output file pointer for printing to text file.

5.39.2.25 eval_eps

```
double(* MONKFISH_DATA::eval_eps) (int i, int l, const void *user_data)
```

Function pointer to evaluate the porosity of the woven bundle of fibers.

5.39.2.26 eval_rho

```
double(* MONKFISH_DATA::eval_rho) (int i, int l, const void *user_data)
```

Function pointer to evaluate the fiber density in the domain.

5.39.2.27 eval_Dex

```
double(* MONKFISH_DATA::eval_Dex) (int i, int l, const void *user_data)
```

Function pointer to evaluate the interparticle diffusivity.

5.39.2.28 eval_ads

```
double(* MONKFISH_DATA::eval_ads) (int i, int l, const void *user_data)
```

Function pointer to evaluate the adsorption strength for the macro-scale.

5.39.2.29 eval_Ret

```
double(* MONKFISH_DATA::eval_Ret) (int i, int l, const void *user_data)
```

Function pointer to evaluate the retardation coefficient for the macro-scale.

5.39.2.30 eval_Cex

```
double(* MONKFISH_DATA::eval_Cex) (int i, const void *user_data)
```

Function pointer to evaluate the exterior concentration for the domain.

5.39.2.31 eval_kf

```
double(* MONKFISH_DATA::eval_kf) (int i, const void *user_data)
```

Function pointer to evalutate the film mass transfer coefficient for the macro-scale.

5.39.2.32 user_data

```
const void* MONKFISH_DATA::user_data
```

User supplied data function to evaluate the function pointers (Default = MONKFISH_DATA)

5.39.2.33 finch_dat

```
std::vector<FINCH_DATA> MONKFISH_DATA::finch_dat
```

FINCH data structures to solve each species interparticle diffusion equation.

5.39.2.34 param_dat

```
std::vector<MONKFISH_PARAM> MONKFISH_DATA::param_dat
```

MONKFISH parameter data structure for each species adsorbing.

5.39.2.35 dog_dat

```
std::vector<DOGFISH_DATA> MONKFISH_DATA::dog_dat
```

DOGFISH data structures for each node in the macro-scale problem.

The documentation for this struct was generated from the following file:

- monkfish.h

5.40 MONKFISH_PARAM Struct Reference

Data structure for species specific information and parameters.

```
#include <monkfish.h>
```

Public Attributes

- double interparticle_diffusion

  Units: cm^2/hr.
- double exterior_concentration

  Units: mol/L.
- double exterior_transfer_coeff

  Units: cm/hr.
- double sorbed_molefraction

  Units: -.
- double initial_sorption

  Units: mg/g.
- double sorption_bc

  Units: mg/g.
- double intraparticle_diffusion

  Units: um^2/hr.
- double film_transfer_coeff

  Units: um/hr.
- Matrix< double > avg_sorption

  Units: mg/g.
- Matrix< double > avg_sorption_old

  Units: mg/g.
- Molecule species

  Species in the liquid phase.

5.40.1 Detailed Description

Data structure for species specific information and parameters.

C-style object to hold information associated with the different species present in the interparticle diffusion problem. Each species may have different diffusivities, mass transfer coefficients, etc. Average adsorption for each species will be held in matrix objects.

5.40.2 Member Data Documentation

5.40.2.1 interparticle_diffusion

```
double MONKFISH_PARAM::interparticle_diffusion
```

Units: cm^2/hr.

5.40.2.2 exterior_concentration

```
double MONKFISH_PARAM::exterior_concentration
```

Units: mol/L.

5.40.2.3 exterior_transfer_coeff

```
double MONKFISH_PARAM::exterior_transfer_coeff
```

Units: cm/hr.

5.40.2.4 sorbed_molefraction

```
double MONKFISH_PARAM::sorbed_molefraction
```

Units: -.

5.40.2.5 initial_sorption

```
double MONKFISH_PARAM::initial_sorption
```

Units: mg/g.

5.40.2.6 sorption_bc

```
double MONKFISH_PARAM::sorption_bc
```

Units: mg/g.

5.40.2.7 intraparticle_diffusion

```
double MONKFISH_PARAM::intraparticle_diffusion
```

Units: um^2/hr.

5.40.2.8 film_transfer_coeff

```
double MONKFISH_PARAM::film_transfer_coeff
```

Units: um/hr.

5.40.2.9 avg_sorption

```
Matrix<double> MONKFISH_PARAM::avg_sorption
```

Units: mg/g.

5.40.2.10 avg_sorption_old

```
Matrix<double> MONKFISH_PARAM::avg_sorption_old
```

Units: mg/g.

5.40.2.11 species

```
Molecule MONKFISH_PARAM::species
```

Species in the liquid phase.

The documentation for this struct was generated from the following file:

- monkfish.h

5.41 mSPD_DATA Struct Reference

MSPD Data Structure.

```
#include <magpie.h>
```

Public Attributes

- double s
  Area shape factor.
- double v
  van der Waals Volume (cm^3/mol)
- double eMax
  Maximum lateral interaction energy (J/mol)
- std::vector< double > eta
  Binary interaction parameter matrix (i,j)
- double gama
  Activity coefficient calculated from mSPD.

5.41.1 Detailed Description

MSPD Data Structure.

C-Style object holding all parameter information associated with the Modified Spreading Pressure Dependent (SPD) activity model. Each species in the gas phase will have one of these objects.

5.41.2 Member Data Documentation

5.41.2.1 s

```
double mSPD_DATA::s
```

Area shape factor.

5.41.2.2 v

```
double mSPD_DATA::v
```

van der Waals Volume (cm^3/mol)

5.41.2.3 eMax

```
double mSPD_DATA::eMax
```

Maximum lateral interaction energy (J/mol)

5.41.2.4 eta

```
std::vector<double> mSPD_DATA::eta
```

Binary interaction parameter matrix (i,j)

5.41.2.5 gama

```
double mSPD_DATA::gama
```

Activity coefficient calculated from mSPD.

The documentation for this struct was generated from the following file:

- magpie.h

5.42 MultiConstReaction Class Reference

MultiConstReaction class is an object associated with rate functions deriving from multiple reactions.

```
#include <crow.h>
```

Public Member Functions

- MultiConstReaction ()

  Default constructor.
- ∼MultiConstReaction ()

  Default destructor.
- void SetNumberReactions (unsigned int i)

  Set the number of reactions for the rate function.
- void InitializeSolver (Dove &Solver)

  Function to initialize each ConstReaction object from the Dove object.
- void SetIndex (int index)

  Function to set the index of the primary variable species (same for all reactions)
- void SetForwardRate (int react, double rate)

  Function to set the forward reaction rate for the indicated reaction.
- void SetReverseRate (int react, double rate)

  Function to set the reverse reaction rate for the indicated reaction.
- void InsertStoichiometry (int react, int i, double v)

  Function to insert stoichiometry for the given reaction.
- int getNumReactions ()

  Return the number of reactions in the object.
- ConstReaction & getReaction (int i)

  Return reference to the ith ConstReaction object.

Protected Attributes

- int num_reactions

  Number of reaction objects.
- std::vector< ConstReaction > reactions

  List of reaction objects associated with MultiConstReaction.

5.42.1 Detailed Description

MultiConstReaction class is an object associated with rate functions deriving from multiple reactions.

This is a C++ style object designed to store and operate on the generic representation of multiple reaction mechanisms. In this object, the reaction parameters are treated as constants and do not change with temperature. This object can be inherited from to add functionality that will compute reaction parameters as a function of system temperature. In addition, this object will contribute user functions (rate functions and jacobians) to DOVE, which will be interfaced through CROW_DATA (see below the class definition).

5.42.2 Constructor & Destructor Documentation

5.42.2.1 MultiConstReaction()

```
MultiConstReaction::MultiConstReaction ( )
```

Default constructor.

5.42.2.2 ∼MultiConstReaction()

```
MultiConstReaction::∼MultiConstReaction ( )
```

Default destructor.

5.42.3 Member Function Documentation

5.42.3.1 SetNumberReactions()

```
void MultiConstReaction::SetNumberReactions (
             unsigned int i )
```

Set the number of reactions for the rate function.

5.42.3.2 InitializeSolver()

```
void MultiConstReaction::InitializeSolver (
             Dove & Solver )
```

Function to initialize each ConstReaction object from the Dove object.

5.42.3.3 SetIndex()

```
void MultiConstReaction::SetIndex (
             int index )
```

Function to set the index of the primary variable species (same for all reactions)

5.42.3.4 SetForwardRate()

```
void MultiConstReaction::SetForwardRate (
             int react,
             double rate )
```

Function to set the forward reaction rate for the indicated reaction.

5.42.3.5 SetReverseRate()

```
void MultiConstReaction::SetReverseRate (
             int react,
             double rate )
```

Function to set the reverse reaction rate for the indicated reaction.

5.42.3.6 InsertStoichiometry()

```
void MultiConstReaction::InsertStoichiometry (
             int react,
             int i,
             double v )
```

Function to insert stoichiometry for the given reaction.

5.42.3.7 getNumReactions()

```
int MultiConstReaction::getNumReactions ( )
```

Return the number of reactions in the object.

5.42.3.8 getReaction()

```
ConstReaction& MultiConstReaction::getReaction (
             int i )
```

Return reference to the ith ConstReaction object.

5.42.4 Member Data Documentation

5.42.4.1 num_reactions

```
int MultiConstReaction::num_reactions  [protected]
```

Number of reaction objects.

5.42.4.2 reactions

```
std::vector<ConstReaction> MultiConstReaction::reactions  [protected]
```

List of reaction objects associated with MultiConstReaction.

The documentation for this class was generated from the following file:

- crow.h

5.43 MultiligandAdsorption Class Reference

Multi-ligand Adsorption Reaction Object.

```
#include <shark.h>
```

Public Member Functions

- MultiligandAdsorption ()

    Default Constructor.
- ∼MultiligandAdsorption ()

    Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int l, std::vector< int > n)

    Function to call the initialization of objects sequentially.
- void modifyDeltas (MassBalance &mbo)

    Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

    Find and set the adsorbed species indices for each reaction object in each ligand object.
- int checkAqueousIndices ()

    Function to check and report errors in the aqueous species indices.
- void setActivityModelInfo (int(∗act)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data), const void ∗act_data)

    Function to set the surface activity model and data pointer.
- int setAqueousIndexAuto ()

    Automatically sets the primary aqueous species index based on reactions for each ligand.
- void setActivityEnum (int act)

    Set the activity enum to the value of act.
- void setMolarFactor (int ligand, int rxn, double m)

    Set the molar factor for the rxn reaction of the ligand ligand to a value of m.
- void setVolumeFactor (int i, double v)

    Set all ith volume factors for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

    Set all ith area factors for the species list (m^2/mol)
- void setSpecificMolality (int ligand, double a)

    Set the specific molality for the ligand (mol/kg)
- void setSurfaceCharge (int ligand, double c)
- void setAdsorbentName (std::string name)

    Set the name of the adsorbent material or particle.
- void setLigandName (int i, std::string name)

    Set the name of the ith ligand.
- void setSpecificArea (double area)

    Set the specific area of the adsorbent.
- void setTotalMass (double mass)

    Set the mass of the adsorbent.
- void setTotalVolume (double volume)

    Set the total volume of the system.
- void setSurfaceChargeBool (bool opt)

    Set the surface charge boolean.
- void setElectricPotential (double a)

    Set the surface electric potential.
- void calculateAreaFactors ()

    Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

    Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

    Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- void calculateElecticPotential (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int rxn, int ligand)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_↩ perm, int rxn, int ligand)

  Calculates the residual for the ith reaction and lth ligand in the system.
- AdsorptionReaction & getAdsorptionObject (int i)

  Return reference to the adsortpion object corresponding to ligand i.
- int getNumberLigands ()

  Get the number of ligands involved with the surface.
- int getActivityEnum ()

  Get the value of the activity enum set by user.
- double getActivity (int i)

  Get the ith activity coefficient from the matrix object.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- double getElectricPotential ()

  Get the value of the electric surface potential (V)
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getLigandName (int i)

  Get the name of the ligand object indexed by i.
- std::string getAdsorbentName ()

  Get the name of the adsorbent.

Protected Attributes

- MasterSpeciesList ∗ List

  Pointer to the MasterSpeciesList object.
- int num_ligands

  Number of different ligands to consider.
- std::string adsorbent_name

  Name of the adsorbent.
- int(∗ surface_activity)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data)

  Pointer to a surface activity model.
- const void ∗ activity_data

Pointer to the data structure needed for surface activities.

- int act_fun

  Enumeration to represent the choosen surface activity function.

- Matrix< double > activities

  List of the activities calculated by the activity model.

- double specific_area

  Specific surface area of the adsorbent (m^2/kg)

- double total_mass

  Total mass of the adsorbent in the system (kg)

- double total_volume

  Total volume of the system (L)

- double ionic_strength

  Ionic Strength of the system used to adjust equilibria constants (mol/L)

- double charge_density

  Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

- double electric_potential

  Electric surface potential of the adsorbent used to adjust equilibria (V)

- bool IncludeSurfCharge

  True = Includes surface charging corrections, False = Does not consider surface charge.

Private Attributes

- std::vector< AdsorptionReaction > ligand_obj

  List of the ligands and reactions they have on the surface.

5.43.1 Detailed Description

Multi-ligand Adsorption Reaction Object.

C++ Object to handle data and functions associated with forumlating multi-ligand adsorption reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure. This object is made from a vector of AdsorptionReaction objects, but differentiate between different ligands that exist on the surface.

5.43.2 Constructor & Destructor Documentation

5.43.2.1 MultiligandAdsorption()

```
MultiligandAdsorption::MultiligandAdsorption ( )
```

Default Constructor.

5.43.2.2 ∼MultiligandAdsorption()

```
MultiligandAdsorption::∼MultiligandAdsorption ( )
```

Default Destructor.

5.43.3 Member Function Documentation

5.43.3.1 Initialize_Object()

```
void MultiligandAdsorption::Initialize_Object (
            MasterSpeciesList & List,
            int l,
            std::vector< int > n )
```

Function to call the initialization of objects sequentially.

Function will initialize each ligand adsorption object.

Parameters

List	reference to MasterSpeciesList object
l	number of ligands on the surface
n	number of reactions for each ligand (ligands must be correctly indexed)

5.43.3.2 modifyDeltas()

```
void MultiligandAdsorption::modifyDeltas (
            MassBalance & mbo )
```

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.43.3.3 setAdsorbIndices()

```
int MultiligandAdsorption::setAdsorbIndices ( )
```

Find and set the adsorbed species indices for each reaction object in each ligand object.

This function searches through the Reaction objects in AdsorptionReaction to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.43.3.4 checkAqueousIndices()

```
int MultiligandAdsorption::checkAqueousIndices ( )
```

Function to check and report errors in the aqueous species indices.

5.43.3.5 setActivityModelInfo()

```
void MultiligandAdsorption::setActivityModelInfo (
             int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void
*data) act,
             const void * act_data )
```

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.43.3.6 setAqueousIndexAuto()

```
int MultiligandAdsorption::setAqueousIndexAuto ( )
```

Automatically sets the primary aqueous species index based on reactions for each ligand.

This function will go through all species and all reactions in each adsorption object and automatically set the primary aqueous species index based on the stoicheometry of the reaction. It will also check and make sure that the primary aqueous index species appears opposite of the adsorbed species in the reactions. Note: This function assumes that the adsorbed indices have already been set.

5.43.3.7 setActivityEnum()

```
void MultiligandAdsorption::setActivityEnum (
             int act )
```

Set the activity enum to the value of act.

5.43.3.8 setMolarFactor()

```
void MultiligandAdsorption::setMolarFactor (
             int ligand,
             int rxn,
             double m )
```

Set the molar factor for the rxn reaction of the ligand ligand to a value of m.

5.43.3.9 setVolumeFactor()

```
void MultiligandAdsorption::setVolumeFactor (
            int i,
            double v )
```

Set all ith volume factors for the species list (cm^3/mol)

5.43.3.10 setAreaFactor()

```
void MultiligandAdsorption::setAreaFactor (
            int i,
            double a )
```

Set all ith area factors for the species list (m^2/mol)

5.43.3.11 setSpecificMolality()

```
void MultiligandAdsorption::setSpecificMolality (
            int ligand,
            double a )
```

Set the specific molality for the ligand (mol/kg)

5.43.3.12 setSurfaceCharge()

```
void MultiligandAdsorption::setSurfaceCharge (
            int ligand,
            double c )
```

5.43.3.13 setAdsorbentName()

```
void MultiligandAdsorption::setAdsorbentName (
            std::string name )
```

Set the name of the adsorbent material or particle.

5.43.3.14 setLigandName()

```
void MultiligandAdsorption::setLigandName (
            int i,
            std::string name )
```

Set the name of the ith ligand.

5.43.3.15 setSpecificArea()

```
void MultiligandAdsorption::setSpecificArea (
            double area )
```

Set the specific area of the adsorbent.

5.43.3.16 setTotalMass()

```
void MultiligandAdsorption::setTotalMass (
            double mass )
```

Set the mass of the adsorbent.

5.43.3.17 setTotalVolume()

```
void MultiligandAdsorption::setTotalVolume (
            double volume )
```

Set the total volume of the system.

5.43.3.18 setSurfaceChargeBool()

```
void MultiligandAdsorption::setSurfaceChargeBool (
            bool opt )
```

Set the surface charge boolean.

5.43.3.19 setElectricPotential()

```
void MultiligandAdsorption::setElectricPotential (
            double a )
```

Set the surface electric potential.

5.43.3.20 calculateAreaFactors()

```
void MultiligandAdsorption::calculateAreaFactors ( )
```

Calculates the area factors used from the van der Waals volumes.

5.43.3.21 calculateEquilibria()

```
void MultiligandAdsorption::calculateEquilibria (
             double T )
```

Calculates all equilibrium parameters as a function of temperature.

5.43.3.22 setChargeDensity()

```
void MultiligandAdsorption::setChargeDensity (
             const Matrix< double > & x )
```

Calculates and sets the current value of charge density.

5.43.3.23 setIonicStrength()

```
void MultiligandAdsorption::setIonicStrength (
             const Matrix< double > & x )
```

Calculates and sets the current value of ionic strength.

5.43.3.24 callSurfaceActivity()

```
int MultiligandAdsorption::callSurfaceActivity (
             const Matrix< double > & x )
```

Calls the activity model and returns an int flag for success or failure.

5.43.3.25 calculateElecticPotential()

```
void MultiligandAdsorption::calculateElecticPotential (
             double sigma,
             double T,
             double I,
             double rel_epsilon )
```

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.43.3.26 calculateEquilibriumCorrection()

```
double MultiligandAdsorption::calculateEquilibriumCorrection (
            double sigma,
            double T,
            double I,
            double rel_epsilon,
            int rxn,
            int ligand )
```

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.43.3.27 Eval_Residual()

```
double MultiligandAdsorption::Eval_Residual (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            double T,
            double rel_perm,
            int rxn,
            int ligand )
```

Calculates the residual for the ith reaction and lth ligand in the system.

This function will provide a system residual for the ith reaction object involved in the lth ligand's Adsorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.43.3.28 getAdsorptionObject()

```
AdsorptionReaction& MultiligandAdsorption::getAdsorptionObject (
             int i )
```

Return reference to the adsortpion object corresponding to ligand i.

5.43.3.29 getNumberLigands()

```
int MultiligandAdsorption::getNumberLigands ( )
```

Get the number of ligands involved with the surface.

5.43.3.30 getActivityEnum()

```
int MultiligandAdsorption::getActivityEnum ( )
```

Get the value of the activity enum set by user.

5.43.3.31 getActivity()

```
double MultiligandAdsorption::getActivity (
             int i )
```

Get the ith activity coefficient from the matrix object.

5.43.3.32 getSpecificArea()

```
double MultiligandAdsorption::getSpecificArea ( )
```

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.43.3.33 getBulkDensity()

```
double MultiligandAdsorption::getBulkDensity ( )
```

Calculate and return bulk density of adsorbent in system (kg/L)

5.43.3.34 getTotalMass()

```
double MultiligandAdsorption::getTotalMass ( )
```

Get the total mass of adsorbent in the system (kg)

5.43.3.35 getTotalVolume()

```
double MultiligandAdsorption::getTotalVolume ( )
```

Get the total volume of the system (L)

5.43.3.36 getChargeDensity()

```
double MultiligandAdsorption::getChargeDensity ( )
```

Get the value of the surface charge density (C/m^2)

5.43.3.37 getIonicStrength()

```
double MultiligandAdsorption::getIonicStrength ( )
```

Get the value of the ionic strength of solution (mol/L)

5.43.3.38 getElectricPotential()

```
double MultiligandAdsorption::getElectricPotential ( )
```

Get the value of the electric surface potential (V)

5.43.3.39 includeSurfaceCharge()

```
bool MultiligandAdsorption::includeSurfaceCharge ( )
```

Returns true if we are considering surface charging during adsorption.

5.43.3.40 getLigandName()

```
std::string MultiligandAdsorption::getLigandName (
             int i )
```

Get the name of the ligand object indexed by i.

5.43.3.41 getAdsorbentName()

```
std::string MultiligandAdsorption::getAdsorbentName ( )
```

Get the name of the adsorbent.

5.43.4 Member Data Documentation

5.43.4.1 List

```
MasterSpeciesList* MultiligandAdsorption::List  [protected]
```

Pointer to the MasterSpeciesList object.

5.43.4.2 num_ligands

```
int MultiligandAdsorption::num_ligands  [protected]
```

Number of different ligands to consider.

5.43.4.3 adsorbent_name

```
std::string MultiligandAdsorption::adsorbent_name  [protected]
```

Name of the adsorbent.

5.43.4.4 surface_activity

```
int(* MultiligandAdsorption::surface_activity) (const Matrix< double > &logq, Matrix< double
> &activity, const void *data)  [protected]
```

Pointer to a surface activity model.

This is a function pointer for a surface activity model. The function must accept the log of the surface concentrations as an argument (logq) and provide the activities for each species (activity). The pointer data is used to pass any additional arguments needed.

Parameters

logq	matrix of the log (base 10) of surface concentrations of all species
activity	matrix of activity coefficients for all surface species (must be overriden)
data	pointer to a data structure needed to calculate activities

5.43.4.5 activity_data

```
const void* MultiligandAdsorption::activity_data  [protected]
```

Pointer to the data structure needed for surface activities.

5.43.4.6 act_fun

```
int MultiligandAdsorption::act_fun  [protected]
```

Enumeration to represent the choosen surface activity function.

5.43.4.7 activities

```
Matrix<double> MultiligandAdsorption::activities  [protected]
```

List of the activities calculated by the activity model.

5.43.4.8 specific_area

```
double MultiligandAdsorption::specific_area  [protected]
```

Specific surface area of the adsorbent (m^2/kg)

5.43.4.9 total_mass

```
double MultiligandAdsorption::total_mass  [protected]
```

Total mass of the adsorbent in the system (kg)

5.43.4.10 total_volume

```
double MultiligandAdsorption::total_volume  [protected]
```

Total volume of the system (L)

5.43.4.11 ionic_strength

`double MultiligandAdsorption::ionic_strength [protected]`

Ionic Strength of the system used to adjust equilibria constants (mol/L)

5.43.4.12 charge_density

`double MultiligandAdsorption::charge_density [protected]`

Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

5.43.4.13 electric_potential

`double MultiligandAdsorption::electric_potential [protected]`

Electric surface potential of the adsorbent used to adjust equilibria (V)

5.43.4.14 IncludeSurfCharge

`bool MultiligandAdsorption::IncludeSurfCharge [protected]`

True = Includes surface charging corrections, False = Does not consider surface charge.

5.43.4.15 ligand_obj

`std::vector<AdsorptionReaction> MultiligandAdsorption::ligand_obj [private]`

List of the ligands and reactions they have on the surface.

The documentation for this class was generated from the following file:

- shark.h

5.44 MultiligandChemisorption Class Reference

Multi-ligand Chemisorption Reaction Object.

`#include <shark.h>`

Public Member Functions

- MultiligandChemisorption ()

  Default Constructor.
- ∼MultiligandChemisorption ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int l, std::vector< int > n)

  Function to call the initialization of objects sequentially.
- void Display_Info ()

  Display the adsorption reaction information.
- void modifyMBEdeltas (MassBalance &mbo)

  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

  Find and set the adsorbed species indices for each reaction object in each ligand object.
- int setLigandIndices ()

  Find and set the ligand species index.
- int setDeltas ()

  Find and set all the delta values for the site balance.
- void setActivityModelInfo (int(∗act)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data), const void ∗act_data)

  Function to set the surface activity model and data pointer.
- void setActivityEnum (int act)

  Set the activity enum to the value of act.
- void setVolumeFactor (int i, double v)

  Set all ith volume factors for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

  Set all ith area factors for the species list (m^2/mol)
- void setSpecificMolality (int ligand, double a)

  Set the specific molality for the ligand (mol/kg)
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent material or particle.
- void setLigandName (int ligand, std::string name)

  Set the name of the ith ligand.
- void setSpecificArea (double area)

  Set the specific area of the adsorbent.
- void setTotalMass (double mass)

  Set the mass of the adsorbent.
- void setTotalVolume (double volume)

  Set the total volume of the system.
- void setSurfaceChargeBool (bool opt)

  Set the surface charge boolean.
- void setElectricPotential (double a)

  Set the surface electric potential.
- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.

- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- void calculateElecticPotential (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int rxn, int ligand)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_RxnResidual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int rxn, int ligand)

  Calculates the residual for the ith reaction and lth ligand in the system.
- double Eval_SiteBalanceResidual (const Matrix< double > &x, int ligand)

  Calculates the residual for the overall site balance for a given ligand.
- ChemisorptionReaction & getChemisorptionObject (int ligand)

  Return reference to the adsortpion object corresponding to the ligand.
- int getNumberLigands ()

  Get the number of ligands involved with the surface.
- int getActivityEnum ()

  Get the value of the activity enum set by user.
- double getActivity (int i)

  Get the ith activity coefficient from the matrix object.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getBulkDensity ()

  Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

  Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

  Get the total volume of the system (L)
- double getChargeDensity ()

  Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

  Get the value of the ionic strength of solution (mol/L)
- double getElectricPotential ()

  Get the value of the electric surface potential (V)
- bool includeSurfaceCharge ()

  Returns true if we are considering surface charging during adsorption.
- std::string getLigandName (int ligand)

  Get the name of the ligand object indexed by ligand.
- std::string getAdsorbentName ()

  Get the name of the adsorbent.

Protected Attributes

- MasterSpeciesList * List

  Pointer to the MasterSpeciesList object.
- int num_ligands

  Number of different ligands to consider.
- std::string adsorbent_name

  Name of the adsorbent.
- int(* surface_activity)(const Matrix< double > &logq, Matrix< double > &activity, const void *data)

  Pointer to a surface activity model.

- const void ∗ activity_data

  Pointer to the data structure needed for surface activities.
- int act_fun

  Enumeration to represent the choosen surface activity function.
- Matrix< double > activities

  List of the activities calculated by the activity model.
- double specific_area

  Specific surface area of the adsorbent (m^2/kg)
- double total_mass

  Total mass of the adsorbent in the system (kg)
- double total_volume

  Total volume of the system (L)
- double ionic_strength

  Ionic Strength of the system used to adjust equilibria constants (mol/L)
- double charge_density

  Surface charge density of the adsorbent used to adjust equilbria (C/m^2)
- double electric_potential

  Electric surface potential of the adsorbent used to adjust equilibria (V)
- bool IncludeSurfCharge

  True = Includes surface charging corrections, False = Does not consider surface charge.

Private Attributes

- std::vector< ChemisorptionReaction > ligand_obj

  List of the ligands and reactions they have on the surface.

5.44.1 Detailed Description

Multi-ligand Chemisorption Reaction Object.

C++ Object to handle data and functions associated with forumlating multi-ligand chemisorption reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure. This object is made from a vector of ChemisorptionReaction objects, but differentiate between different ligands that exist on the surface. It is based largely off of the original Multiligand Adsorption object, but will include an explict way to handle the site balances associated with each ligand.

5.44.2 Constructor & Destructor Documentation

5.44.2.1 MultiligandChemisorption()

```
MultiligandChemisorption::MultiligandChemisorption ( )
```

Default Constructor.

5.44.2.2 ∼MultiligandChemisorption()

```
MultiligandChemisorption::∼MultiligandChemisorption ( )
```

Default Destructor.

5.44.3 Member Function Documentation

5.44.3.1 Initialize_Object()

```
void MultiligandChemisorption::Initialize_Object (
            MasterSpeciesList & List,
            int l,
            std::vector< int > n )
```

Function to call the initialization of objects sequentially.

Function will initialize each ligand adsorption object.

Parameters

List	reference to MasterSpeciesList object
l	number of ligands on the surface
n	number of reactions for each ligand (ligands must be correctly indexed)

5.44.3.2 Display_Info()

```
void MultiligandChemisorption::Display_Info ( )
```

Display the adsorption reaction information.

5.44.3.3 modifyMBEdeltas()

```
void MultiligandChemisorption::modifyMBEdeltas (
            MassBalance & mbo )
```

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.44.3.4 setAdsorbIndices()

```
int MultiligandChemisorption::setAdsorbIndices ( )
```

Find and set the adsorbed species indices for each reaction object in each ligand object.

This function searches through the Reaction objects in ChemisorptionReaction to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.44.3.5 setLigandIndices()

```
int MultiligandChemisorption::setLigandIndices ( )
```

Find and set the ligand species index.

This function searches through the Reaction objects in ChemisorptionReaction to find the ligand species and its index to set that information in the ligand_index structure. Function will return 0 if successful and -1 on a failure.

5.44.3.6 setDeltas()

```
int MultiligandChemisorption::setDeltas ( )
```

Find and set all the delta values for the site balance.

This function searches through all reaction object instances for the stoicheometry of the ligand in each adsorption reaction. That stoicheometry serves as the basis for determining the site balance. NOTE: the delta for the ligand is set automatically in the setLigandIndex() function, so we can ignore that species. In addition, this function must be called after setLigandIndex() and setAdsorbIndices() are called and after the stoicheometry of each reaction has been determined.

5.44.3.7 setActivityModelInfo()

```
void MultiligandChemisorption::setActivityModelInfo (
             int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void
*data) act,
             const void * act_data )
```

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.44.3.8 setActivityEnum()

```
void MultiligandChemisorption::setActivityEnum (
             int act )
```

Set the activity enum to the value of act.

5.44.3.9 setVolumeFactor()

```
void MultiligandChemisorption::setVolumeFactor (
             int i,
             double v )
```

Set all ith volume factors for the species list (cm^3/mol)

5.44.3.10 setAreaFactor()

```
void MultiligandChemisorption::setAreaFactor (
             int i,
             double a )
```

Set all ith area factors for the species list (m^2/mol)

5.44.3.11 setSpecificMolality()

```
void MultiligandChemisorption::setSpecificMolality (
             int ligand,
             double a )
```

Set the specific molality for the ligand (mol/kg)

5.44.3.12 setAdsorbentName()

```
void MultiligandChemisorption::setAdsorbentName (
             std::string name )
```

Set the name of the adsorbent material or particle.

5.44.3.13 setLigandName()

```
void MultiligandChemisorption::setLigandName (
             int ligand,
             std::string name )
```

Set the name of the ith ligand.

5.44.3.14 setSpecificArea()

```
void MultiligandChemisorption::setSpecificArea (
             double area )
```

Set the specific area of the adsorbent.

5.44.3.15 setTotalMass()

```
void MultiligandChemisorption::setTotalMass (
            double mass )
```

Set the mass of the adsorbent.

5.44.3.16 setTotalVolume()

```
void MultiligandChemisorption::setTotalVolume (
            double volume )
```

Set the total volume of the system.

5.44.3.17 setSurfaceChargeBool()

```
void MultiligandChemisorption::setSurfaceChargeBool (
            bool opt )
```

Set the surface charge boolean.

5.44.3.18 setElectricPotential()

```
void MultiligandChemisorption::setElectricPotential (
            double a )
```

Set the surface electric potential.

5.44.3.19 calculateAreaFactors()

```
void MultiligandChemisorption::calculateAreaFactors ( )
```

Calculates the area factors used from the van der Waals volumes.

5.44.3.20 calculateEquilibria()

```
void MultiligandChemisorption::calculateEquilibria (
            double T )
```

Calculates all equilibrium parameters as a function of temperature.

5.44.3.21 setChargeDensity()

```
void MultiligandChemisorption::setChargeDensity (
            const Matrix< double > & x )
```

Calculates and sets the current value of charge density.

5.44.3.22 setIonicStrength()

```
void MultiligandChemisorption::setIonicStrength (
            const Matrix< double > & x )
```

Calculates and sets the current value of ionic strength.

5.44.3.23 callSurfaceActivity()

```
int MultiligandChemisorption::callSurfaceActivity (
            const Matrix< double > & x )
```

Calls the activity model and returns an int flag for success or failure.

5.44.3.24 calculateElecticPotential()

```
void MultiligandChemisorption::calculateElecticPotential (
            double sigma,
            double T,
            double I,
            double rel_epsilon )
```

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.44.3.25 calculateEquilibriumCorrection()

```
double MultiligandChemisorption::calculateEquilibriumCorrection (
            double sigma,
            double T,
            double I,
            double rel_epsilon,
            int rxn,
            int ligand )
```

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.44.3.26 Eval_RxnResidual()

```
double MultiligandChemisorption::Eval_RxnResidual (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            double T,
            double rel_perm,
            int rxn,
            int ligand )
```

Calculates the residual for the ith reaction and lth ligand in the system.

This function will provide a system residual for the ith reaction object involved in the lth ligand's Adsorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
rxn	index of the reaction of interest for the adsorption object
ligand	index of the ligand of interest for the adsorption object

5.44.3.27 Eval_SiteBalanceResidual()

```
double MultiligandChemisorption::Eval_SiteBalanceResidual (
              const Matrix< double > & x,
              int ligand )
```

Calculates the residual for the overall site balance for a given ligand.

This function will provide a system residual for the site/ligand balance for the Chemisorption Reaction object. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
ligand	index of the ligand of interest of the chemisorption object

5.44.3.28 getChemisorptionObject()

```
ChemisorptionReaction& MultiligandChemisorption::getChemisorptionObject (
              int ligand )
```

Return reference to the adsortpion object corresponding to the ligand.

5.44.3.29 getNumberLigands()

```
int MultiligandChemisorption::getNumberLigands ( )
```

Get the number of ligands involved with the surface.

5.44.3.30 getActivityEnum()

```
int MultiligandChemisorption::getActivityEnum ( )
```

Get the value of the activity enum set by user.

5.44.3.31 getActivity()

```
double MultiligandChemisorption::getActivity (
              int i )
```

Get the ith activity coefficient from the matrix object.

5.44.3.32 getSpecificArea()

```
double MultiligandChemisorption::getSpecificArea ( )
```

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.44.3.33 getBulkDensity()

```
double MultiligandChemisorption::getBulkDensity ( )
```

Calculate and return bulk density of adsorbent in system (kg/L)

5.44.3.34 getTotalMass()

```
double MultiligandChemisorption::getTotalMass ( )
```

Get the total mass of adsorbent in the system (kg)

5.44.3.35 getTotalVolume()

```
double MultiligandChemisorption::getTotalVolume ( )
```

Get the total volume of the system (L)

5.44.3.36 getChargeDensity()

```
double MultiligandChemisorption::getChargeDensity ( )
```

Get the value of the surface charge density (C/m^2)

5.44.3.37 getIonicStrength()

```
double MultiligandChemisorption::getIonicStrength ( )
```

Get the value of the ionic strength of solution (mol/L)

5.44.3.38 getElectricPotential()

```
double MultiligandChemisorption::getElectricPotential ( )
```

Get the value of the electric surface potential (V)

5.44.3.39 includeSurfaceCharge()

```
bool MultiligandChemisorption::includeSurfaceCharge ( )
```

Returns true if we are considering surface charging during adsorption.

5.44.3.40 getLigandName()

```
std::string MultiligandChemisorption::getLigandName (
             int ligand )
```

Get the name of the ligand object indexed by ligand.

5.44.3.41 getAdsorbentName()

```
std::string MultiligandChemisorption::getAdsorbentName ( )
```

Get the name of the adsorbent.

5.44.4 Member Data Documentation

5.44.4.1 List

```
MasterSpeciesList* MultiligandChemisorption::List  [protected]
```

Pointer to the MasterSpeciesList object.

5.44.4.2 num_ligands

```
int MultiligandChemisorption::num_ligands  [protected]
```

Number of different ligands to consider.

5.44.4.3 adsorbent_name

```
std::string MultiligandChemisorption::adsorbent_name  [protected]
```

Name of the adsorbent.

5.44.4.4 surface_activity

```
int(* MultiligandChemisorption::surface_activity) (const Matrix< double > &logq, Matrix<
double > &activity, const void *data)  [protected]
```

Pointer to a surface activity model.

This is a function pointer for a surface activity model. The function must accept the log of the surface concentrations as an argument (logq) and provide the activities for each species (activity). The pointer data is used to pass any additional arguments needed.

Parameters

logq	matrix of the log (base 10) of surface concentrations of all species
activity	matrix of activity coefficients for all surface species (must be overriden)
data	pointer to a data structure needed to calculate activities

5.44.4.5 activity_data

```
const void* MultiligandChemisorption::activity_data  [protected]
```

Pointer to the data structure needed for surface activities.

5.44.4.6 act_fun

```
int MultiligandChemisorption::act_fun  [protected]
```

Enumeration to represent the choosen surface activity function.

5.44.4.7 activities

```
Matrix<double> MultiligandChemisorption::activities  [protected]
```

List of the activities calculated by the activity model.

5.44.4.8 specific_area

```
double MultiligandChemisorption::specific_area  [protected]
```

Specific surface area of the adsorbent (m^2/kg)

5.44.4.9 total_mass

```
double MultiligandChemisorption::total_mass  [protected]
```

Total mass of the adsorbent in the system (kg)

5.44.4.10 total_volume

```
double MultiligandChemisorption::total_volume  [protected]
```

Total volume of the system (L)

5.44.4.11 ionic_strength

```
double MultiligandChemisorption::ionic_strength [protected]
```

Ionic Strength of the system used to adjust equilibria constants (mol/L)

5.44.4.12 charge_density

```
double MultiligandChemisorption::charge_density [protected]
```

Surface charge density of the adsorbent used to adjust equilbria (C/m^2)

5.44.4.13 electric_potential

```
double MultiligandChemisorption::electric_potential [protected]
```

Electric surface potential of the adsorbent used to adjust equilibria (V)

5.44.4.14 IncludeSurfCharge

```
bool MultiligandChemisorption::IncludeSurfCharge [protected]
```

True = Includes surface charging corrections, False = Does not consider surface charge.

5.44.4.15 ligand_obj

```
std::vector<ChemisorptionReaction> MultiligandChemisorption::ligand_obj [private]
```

List of the ligands and reactions they have on the surface.

The documentation for this class was generated from the following file:

- shark.h

5.45 Node Class Reference

Node object.

```
#include <mesh.h>
```

Public Member Functions

- Node ()

  Default constructor.
- ∼Node ()

  Default destructor.
- void DisplayInfo ()

  Display the information associated with this node.
- void AssignCoordinates (double x, double y, double z)

  Assign the coordinates for the node.
- void AssignX (double x)

  Assign the x-coordinate for the node.
- void AssignY (double y)

  Assign the y-coordinate for the node.
- void AssignZ (double z)

  Assign the z-coordinate for the node.
- void AssignIDnumber (unsigned int i)

  Assign the ID number for the node.
- void AssignSubType (element_type type)

  Assign the node_type for the node.
- bool isSameLocation (Node &node)

  Returns true if nodes are at same location (witin tolerance)
- bool isSameType (Node &node)

  Returns true if nodes are of same type.
- bool isSameNode (Node &node)

  Returns true if nodes have same ID number (could indicate error)
- double distance (Node &node)

  Returns the distance between two given nodes.
- double angle (Node &n1, Node &n2)

  Returns angle between n1 and n2 with respect to this node (radians)
- double angle_degrees (Node &n1, Node &n2)

  Returns angle between n1 and n2 with respect to this node (degrees)
- double getX ()

  Returns x value of node.
- double getY ()

  Returns y value of node.
- double getZ ()

  Returns z value of node.
- element_type getType ()

  Returns the type of node.
- Vector3D operator- (const Node &node)

  Returns vector formed by difference between nodes.

Private Attributes

- Vector3D coordinates

  x, y, z location of the node in space
- unsigned int IDnum

  Identification number for the node.
- element_type SubType

  Sub-type for the node.
- double distance_tolerance

  Tolerance used to determine if two nodes are in same location.

5.45.1 Detailed Description

Node object.

This class structure creates a C++ object for a node in a mesh. The node will have an identifying ID number and a sub_type (either BOUNDARY or INTERIOR) to add in identification of different aspects the node has in the overall mesh. All nodes are considered to be points in 3D space, whether or not the final mesh is 3D. As such, every node will have a coordinate vector (x, y, z) associated with it.

5.45.2 Constructor & Destructor Documentation

5.45.2.1 Node()

```
Node::Node ( )
```

Default constructor.

5.45.2.2 ∼Node()

```
Node::∼Node ( )
```

Default destructor.

5.45.3 Member Function Documentation

5.45.3.1 DisplayInfo()

```
void Node::DisplayInfo ( )
```

Display the information associated with this node.

5.45.3.2 AssignCoordinates()

```
void Node::AssignCoordinates (
            double x,
            double y,
            double z )
```

Assign the coordinates for the node.

5.45.3.3 AssignX()

```
void Node::AssignX (
             double x )
```

Assign the x-coordinate for the node.

5.45.3.4 AssignY()

```
void Node::AssignY (
             double y )
```

Assign the y-coordinate for the node.

5.45.3.5 AssignZ()

```
void Node::AssignZ (
             double z )
```

Assign the z-coordinate for the node.

5.45.3.6 AssignIDnumber()

```
void Node::AssignIDnumber (
             unsigned int i )
```

Assign the ID number for the node.

5.45.3.7 AssignSubType()

```
void Node::AssignSubType (
             element_type type )
```

Assign the node_type for the node.

5.45.3.8 isSameLocation()

```
bool Node::isSameLocation (
             Node & node )
```

Returns true if nodes are at same location (witin tolerance)

5.45.3.9 isSameType()

```
bool Node::isSameType (
             Node & node )
```

Returns true if nodes are of same type.

5.45.3.10 isSameNode()

```
bool Node::isSameNode (
             Node & node )
```

Returns true if nodes have same ID number (could indicate error)

5.45.3.11 distance()

```
double Node::distance (
             Node & node )
```

Returns the distance between two given nodes.

5.45.3.12 angle()

```
double Node::angle (
             Node & n1,
             Node & n2 )
```

Returns angle between n1 and n2 with respect to this node (radians)

5.45.3.13 angle_degrees()

```
double Node::angle_degrees (
             Node & n1,
             Node & n2 )
```

Returns angle between n1 and n2 with respect to this node (degrees)

5.45.3.14 getX()

```
double Node::getX ( )
```

Returns x value of node.

5.45.3.15 getY()

```
double Node::getY ( )
```

Returns y value of node.

5.45.3.16 getZ()

```
double Node::getZ ( )
```

Returns z value of node.

5.45.3.17 getType()

```
element_type Node::getType ( )
```

Returns the type of node.

5.45.3.18 operator-()

```
Vector3D Node::operator- (
             const Node & node )
```

Returns vector formed by difference between nodes.

5.45.4 Member Data Documentation

5.45.4.1 coordinates

```
Vector3D Node::coordinates  [private]
```

x, y, z location of the node in space

5.45.4.2 IDnum

```
unsigned int Node::IDnum  [private]
```

Identification number for the node.

5.45.4.3 SubType

```
element_type Node::SubType  [private]
```

Sub-type for the node.

5.45.4.4 distance_tolerance

```
double Node::distance_tolerance  [private]
```

Tolerance used to determine if two nodes are in same location.

The documentation for this class was generated from the following file:

- mesh.h

5.46 NodeSet Class Reference

NodeSet Object.

```
#include <mesh.h>
```

Public Member Functions

- NodeSet ()

  Default constructor.
- ∼NodeSet ()

  Default destructor.
- void InitializeNodeSet (unsigned int size)

  Create a list of nodes of particular size.
- void RegisterNode (unsigned int idnum, double x, double y, double z, element_type type)

  Retister node with specific info.
- void RegisterNode (Node &node)

  Register node from existing node.
- void AddNode (double x, double y, double z, element_type type)

  Add node to the list (dynamic)
- void AssignDistanceTolerances (double tol)

  Assign all node tolerances to a specific value.

Private Attributes

- std::vector< Node > node_set

5.46.1 Detailed Description

NodeSet Object.

This class structure creates a C++ object for a set of nodes. The set of nodes will be a listing of all nodes in a domain. Each node must have a unique ID number and unique locations within the larger structure. Subsets of the full list are used to construct LineElements, SurfaceElements, and VolumeElements of the Mesh object.

5.46.2 Constructor & Destructor Documentation

5.46.2.1 NodeSet()

```
NodeSet::NodeSet ( )
```

Default constructor.

5.46.2.2 ∼NodeSet()

```
NodeSet::∼NodeSet ( )
```

Default destructor.

5.46.3 Member Function Documentation

5.46.3.1 InitializeNodeSet()

```
void NodeSet::InitializeNodeSet (
             unsigned int size )
```

Create a list of nodes of particular size.

5.46.3.2 RegisterNode() [1/2]

```
void NodeSet::RegisterNode (
             unsigned int idnum,
             double x,
             double y,
             double z,
             element_type type )
```

Retister node with specific info.

5.46.3.3 RegisterNode() [2/2]

```
void NodeSet::RegisterNode (
             Node & node )
```

Register node from existing node.

5.46.3.4 AddNode()

```
void NodeSet::AddNode (
            double x,
            double y,
            double z,
            element_type type )
```

Add node to the list (dynamic)

5.46.3.5 AssignDistanceTolerances()

```
void NodeSet::AssignDistanceTolerances (
            double tol )
```

Assign all node tolerances to a specific value.

5.46.4 Member Data Documentation

5.46.4.1 node_set

```
std::vector<Node> NodeSet::node_set  [private]
```

The documentation for this class was generated from the following file:

- mesh.h

5.47 NUM_JAC_DATA Struct Reference

Data structure to form a numerical jacobian matrix with finite differences.

```
#include <lark.h>
```

Public Attributes

- double eps = sqrt(DBL_EPSILON)

  Perturbation value.
- Matrix< double > Fx

  Vector of function evaluations at x.
- Matrix< double > Fxp

  Vector of function evaluations at x+eps.
- Matrix< double > dxj

  Vector of perturbed x values.

5.47.1 Detailed Description

Data structure to form a numerical jacobian matrix with finite differences.

C-style object to be used in conjunction with the Numerical Jacobian algorithm. This algorithm will used double-precision finite-differences to formulate an approximate Jacobian matrix at the given variable state for the given residual/non-linear function.

5.47.2 Member Data Documentation

5.47.2.1 eps

```
double NUM_JAC_DATA::eps = sqrt(DBL_EPSILON)
```

Perturbation value.

5.47.2.2 Fx

```
Matrix<double> NUM_JAC_DATA::Fx
```

Vector of function evaluations at x.

5.47.2.3 Fxp

```
Matrix<double> NUM_JAC_DATA::Fxp
```

Vector of function evaluations at x+eps.

5.47.2.4 dxj

```
Matrix<double> NUM_JAC_DATA::dxj
```

Vector of perturbed x values.

The documentation for this struct was generated from the following file:

- lark.h

5.48 OPTRANS_DATA Struct Reference

Data structure for implementation of linear operator transposition.

```
#include <lark.h>
```

Public Attributes

- Matrix< double > Ii

  The ith column vector of the identity operator.
- Matrix< double > Ai

  The ith column vector of the user's linear operator.

5.48.1 Detailed Description

Data structure for implementation of linear operator transposition.

C-style object used in conjunction with the Operator Transpose algorithm to form an action of A^T∗r when A is only available as a linear operator and not a matrix. This is a sub-routine required by GCR and GMRESR to stabilize the outer iterations.

5.48.2 Member Data Documentation

5.48.2.1 Ii

```
Matrix<double> OPTRANS_DATA::Ii
```

The ith column vector of the identity operator.

5.48.2.2 Ai

```
Matrix<double> OPTRANS_DATA::Ai
```

The ith column vector of the user's linear operator.

The documentation for this struct was generated from the following file:

- lark.h

5.49 PCG_DATA Struct Reference

Data structure for implementation of the PCG algorithms for symmetric linear systems.

```
#include <lark.h>
```

Public Attributes

- int maxit = 0

  Maximum allowable iterations - default = min(vector_size,1000)
- int iter = 0

  Actual number of iterations taken.
- double alpha

  Step size for new solution.
- double beta

  Step size for new search direction.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolution tolerance for convergence - default = 1e-6.
- double res

  Absolute residual norm.
- double relres

  Relative residual norm.
- double relres_base

  Initial residual norm.
- double bestres

  Best found residual norm.
- bool Output = true

  True = print messages to console.
- Matrix< double > x

  Current solution to the linear system.
- Matrix< double > bestx

  Best found solution to the linear system.
- Matrix< double > r

  Residual vector for the linear system.
- Matrix< double > r_old

  Previous residual vector.
- Matrix< double > z

  Preconditioned residual vector (result of precon function)
- Matrix< double > z_old

  Previous preconditioned residual vector.
- Matrix< double > p

  Search direction.
- Matrix< double > Ap

  Result of matrix-vector multiplication.

5.49.1 Detailed Description

Data structure for implementation of the PCG algorithms for symmetric linear systems.

C-style object used in conjunction with the Preconditioned Conjugate Gradient (PCG) algorithm to iteratively solve a symmetric linear system of equations. This algorithm is optimal if your linear system is symmetric, but will not work at all if your system is asymmetric. For asymmetric systems, use one of the other linear methods.

5.49.2 Member Data Documentation

5.49.2.1 maxit

```
int PCG_DATA::maxit = 0
```

Maximum allowable iterations - default = min(vector_size,1000)

5.49.2.2 iter

```
int PCG_DATA::iter = 0
```

Actual number of iterations taken.

5.49.2.3 alpha

```
double PCG_DATA::alpha
```

Step size for new solution.

5.49.2.4 beta

```
double PCG_DATA::beta
```

Step size for new search direction.

5.49.2.5 tol_rel

```
double PCG_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.49.2.6 tol_abs

```
double PCG_DATA::tol_abs = 1e-6
```

Absolution tolerance for convergence - default = 1e-6.

5.49.2.7 res

```
double PCG_DATA::res
```

Absolute residual norm.

5.49.2.8 relres

```
double PCG_DATA::relres
```

Relative residual norm.

5.49.2.9 relres_base

```
double PCG_DATA::relres_base
```

Initial residual norm.

5.49.2.10 bestres

```
double PCG_DATA::bestres
```

Best found residual norm.

5.49.2.11 Output

```
bool PCG_DATA::Output = true
```

True = print messages to console.

5.49.2.12 x

```
Matrix<double> PCG_DATA::x
```

Current solution to the linear system.

5.49.2.13 bestx

```
Matrix<double> PCG_DATA::bestx
```

Best found solution to the linear system.

5.49.2.14 r

```
Matrix<double> PCG_DATA::r
```

Residual vector for the linear system.

5.49.2.15 r_old

```
Matrix<double> PCG_DATA::r_old
```

Previous residual vector.

5.49.2.16 z

```
Matrix<double> PCG_DATA::z
```

Preconditioned residual vector (result of precon function)

5.49.2.17 z_old

```
Matrix<double> PCG_DATA::z_old
```

Previous preconditioned residual vector.

5.49.2.18 p

```
Matrix<double> PCG_DATA::p
```

Search direction.

5.49.2.19 Ap

```
Matrix<double> PCG_DATA::Ap
```

Result of matrix-vector multiplication.

The documentation for this struct was generated from the following file:

- lark.h

5.50 PeriodicTable Class Reference

Class object that store a digitial copy of all Atom objects.

```
#include <eel.h>
```

Public Member Functions

- PeriodicTable ()
  Default Constructor - Build Perodic Table.
- ∼PeriodicTable ()
  Default Destructor - Destroy the table.
- PeriodicTable (int ∗n, int N)
  Construct a partial table from a list of atomic numbers.
- PeriodicTable (std::vector< std::string > &Symbol)
  Construct a partial table from a vector of atom symbols.
- PeriodicTable (std::vector< int > &n)
  Construct a partial table from a vector of atomic numbers.
- void DisplayTable ()
  Displays the periodic table via symbols.

Protected Attributes

- std::vector< Atom > Table
  Storage vector for all atoms in the table.

Private Attributes

- int number_elements
  Number of atom objects being stored.

5.50.1 Detailed Description

Class object that store a digitial copy of all Atom objects.

C++ class object to hold digitally registered Atom objects. All registered atoms (Hydrogen to Ununoctium) are stored as in a vector. Currently, this object is unused, but could be modified to be explorable and used as a constant referece for all atoms in the table.

5.50.2 Constructor & Destructor Documentation

5.50.2.1 PeriodicTable() [1/4]

```
PeriodicTable::PeriodicTable ( )
```

Default Constructor - Build Perodic Table.

5.50.2.2 ∼PeriodicTable()

```
PeriodicTable::∼PeriodicTable ( )
```

Default Destructor - Destroy the table.

5.50.2.3 PeriodicTable() [2/4]

```
PeriodicTable::PeriodicTable (
            int * n,
            int N )
```

Construct a partial table from a list of atomic numbers.

5.50.2.4 PeriodicTable() [3/4]

```
PeriodicTable::PeriodicTable (
            std::vector< std::string > & Symbol )
```

Construct a partial table from a vector of atom symbols.

5.50.2.5 PeriodicTable() [4/4]

```
PeriodicTable::PeriodicTable (
            std::vector< int > & n )
```

Construct a partial table from a vector of atomic numbers.

5.50.3 Member Function Documentation

5.50.3.1 DisplayTable()

```
void PeriodicTable::DisplayTable ( )
```

Displays the periodic table via symbols.

5.50.4 Member Data Documentation

5.50.4.1 Table

`std::vector<Atom> PeriodicTable::Table [protected]`

Storage vector for all atoms in the table.

5.50.4.2 number_elements

`int PeriodicTable::number_elements [private]`

Number of atom objects being stored.

The documentation for this class was generated from the following file:

- eel.h

5.51 PICARD_DATA Struct Reference

Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems.

`#include <lark.h>`

Public Attributes

- int maxit = 0

  Maximum allowable iterations - default = min(3∗vec_size,1000)
- int iter = 0

  Actual number of iterations.
- double tol_rel = 1e-6

  Relative tolerance for convergence - default = 1e-6.
- double tol_abs = 1e-6

  Absolution tolerance for convergence - default = 1e-6.
- double res

  Residual norm of the iterate.
- double relres

  Relative residual norm of the iterate.
- double relres_base

  Initial residual norm.
- double bestres

  Best found residual norm.
- bool Output = true

  True = print messages to console.
- Matrix< double > x0

  Previous iterate solution vector.
- Matrix< double > bestx

  Best found solution vector.
- Matrix< double > r

  Residual of the non-linear system.

5.51.1 Detailed Description

Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems.

C-style object used in conjunction with the Picard algorithm for solving a non-linear system of equations. This is an extradorinarily simple iterative method by which a weak or loose form of the non-linear system is solved based on an initial guess. User must supplied a residual function for the non-linear system and a function representing the weak solution. Generally, this method is less efficient than Newton methods, but is significantly cheaper.

5.51.2 Member Data Documentation

5.51.2.1 maxit

```
int PICARD_DATA::maxit = 0
```

Maximum allowable iterations - default = min(3∗vec_size,1000)

5.51.2.2 iter

```
int PICARD_DATA::iter = 0
```

Actual number of iterations.

5.51.2.3 tol_rel

```
double PICARD_DATA::tol_rel = 1e-6
```

Relative tolerance for convergence - default = 1e-6.

5.51.2.4 tol_abs

```
double PICARD_DATA::tol_abs = 1e-6
```

Absolution tolerance for convergence - default = 1e-6.

5.51.2.5 res

```
double PICARD_DATA::res
```

Residual norm of the iterate.

5.51.2.6 relres

```
double PICARD_DATA::relres
```

Relative residual norm of the iterate.

5.51.2.7 relres_base

```
double PICARD_DATA::relres_base
```

Initial residual norm.

5.51.2.8 bestres

```
double PICARD_DATA::bestres
```

Best found residual norm.

5.51.2.9 Output

```
bool PICARD_DATA::Output = true
```

True = print messages to console.

5.51.2.10 x0

```
Matrix<double> PICARD_DATA::x0
```

Previous iterate solution vector.

5.51.2.11 bestx

```
Matrix<double> PICARD_DATA::bestx
```

Best found solution vector.

5.51.2.12 r

```
Matrix<double> PICARD_DATA::r
```

Residual of the non-linear system.

The documentation for this struct was generated from the following file:

- lark.h

5.52 PJFNK_DATA Struct Reference

Data structure for the implementation of the PJFNK algorithm for non-linear systems.

```
#include <lark.h>
```

Public Attributes

- int nl_iter = 0

  Number of non-linear iterations.
- int l_iter = 0

  Number of linear iterations.
- int fun_call = 0

  Actual number of function calls made.
- int nl_maxit = 0

  Maximum allowable non-linear steps.
- int l_maxit = 0

  Maximum allowable linear steps.
- int l_restart = -1

  Number of inner linear steps before restarting (for GMRES, GCR, KMS, etc)
- int linear_solver = -1

  Flag to denote which linear solver to use - default = PJFNK Chooses.
- double nl_tol_abs = 1e-6

  Absolute Convergence tolerance for non-linear system - default = 1e-6.
- double nl_tol_rel = 1e-6

  Relative Convergence tol for the non-linear system - default = 1e-6.
- double lin_tol_rel = 1e-6

  Relative tolerance of the linear solver - default = 1e-6.
- double lin_tol_abs = 1e-6

  Absolute tolerance of the linear solver - default = 1e-6.
- double nl_res

  Absolute redidual norm for the non-linear system.
- double nl_relres

  Relative residual for the non-linear system.
- double nl_res_base

  Initial residual norm for the non-linear system.
- double nl_bestres

  Best found residual norm.
- double eps =sqrt(DBL_EPSILON)

  Value of epsilon used jacvec - default = sqrt(DBL_EPSILON)
- bool NL_Output = true

  True = print PJFNK messages to console.
- bool L_Output = false

  True = print Linear messages to console.
- bool LineSearch = false

  True = use Backtracking Linesearch for global convergence.
- bool Bounce = false

  True = allow Linesearch to go outside local well, False = Strict local convergence.
- bool Converged = false

  True = solution has converged, False = solution has not converged.

- Matrix< double > F

    Stored fuction evaluation at x (also the residual)
- Matrix< double > Fv

    Stored function evaluation at x+eps∗v.
- Matrix< double > v

    Stored vector of x+eps∗v.
- Matrix< double > x

    Current solution vector for the non-linear system.
- Matrix< double > bestx

    Best found solution vector to the non-linear system.
- GMRESLP_DATA gmreslp_dat

    Data structure for the GMRESLP method.
- PCG_DATA pcg_dat

    Data structure for the PCG method.
- BiCGSTAB_DATA bicgstab_dat

    Data structure for the BiCGSTAB method.
- CGS_DATA cgs_dat

    Data structure for the CGS method.
- GMRESRP_DATA gmresrp_dat

    Data structure for the GMRESRP method.
- GCR_DATA gcr_dat

    Data structure for the GCR method.
- GMRESR_DATA gmresr_dat

    Data structure for the GMRESR method.
- KMS_DATA kms_dat

    Data structure for the KMS method.
- QR_DATA qr_dat

    Data structure for the QR solve method.
- BACKTRACK_DATA backtrack_dat

    Data structure for the Backtracking Linesearch algorithm.
- const void ∗ res_data

    Data structure pointer for user's residual data.
- const void ∗ precon_data

    Data structure pointer for user's preconditioning data.
- int(∗ funeval)(const Matrix< double > &x, Matrix< double > &F, const void ∗res_data)

    Function pointer for the user's function F(x) using there data.
- int(∗ precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗precon_data)

    Function pointer for the user's preconditioning function for the linear system.

5.52.1 Detailed Description

Data structure for the implementation of the PJFNK algorithm for non-linear systems.

C-style object to be used in conjunction with the Preconditioned Jacobian-Free Newton-Krylov (PJFNK) method for solving a non-linear system of equations. You can use any of the Krylov methods listed in the krylov_method enum to solve the linear sub-problem. When FOM is specified as the Krylov method, this algorithm becomes equivalent to an exact Newton method. If no Krylov method is specified, then the algorithm will try to pick a method based on the problem size and availability of preconditioning.

5.52.2 Member Data Documentation

5.52.2.1 nl_iter

```
int PJFNK_DATA::nl_iter = 0
```

Number of non-linear iterations.

5.52.2.2 l_iter

```
int PJFNK_DATA::l_iter = 0
```

Number of linear iterations.

5.52.2.3 fun_call

```
int PJFNK_DATA::fun_call = 0
```

Actual number of function calls made.

5.52.2.4 nl_maxit

```
int PJFNK_DATA::nl_maxit = 0
```

Maximum allowable non-linear steps.

5.52.2.5 l_maxit

```
int PJFNK_DATA::l_maxit = 0
```

Maximum allowable linear steps.

5.52.2.6 l_restart

```
int PJFNK_DATA::l_restart = -1
```

Number of inner linear steps before restarting (for GMRES, GCR, KMS, etc)

5.52.2.7 linear_solver

```
int PJFNK_DATA::linear_solver = -1
```

Flag to denote which linear solver to use - default = PJFNK Chooses.

5.52.2.8 nl_tol_abs

```
double PJFNK_DATA::nl_tol_abs = 1e-6
```

Absolute Convergence tolerance for non-linear system - default = 1e-6.

5.52.2.9 nl_tol_rel

```
double PJFNK_DATA::nl_tol_rel = 1e-6
```

Relative Convergence tol for the non-linear system - default = 1e-6.

5.52.2.10 lin_tol_rel

```
double PJFNK_DATA::lin_tol_rel = 1e-6
```

Relative tolerance of the linear solver - default = 1e-6.

5.52.2.11 lin_tol_abs

```
double PJFNK_DATA::lin_tol_abs = 1e-6
```

Absolute tolerance of the linear solver - default = 1e-6.

5.52.2.12 nl_res

```
double PJFNK_DATA::nl_res
```

Absolute redidual norm for the non-linear system.

5.52.2.13 nl_relres

```
double PJFNK_DATA::nl_relres
```

Relative residual for the non-linear system.

5.52.2.14 nl_res_base

```
double PJFNK_DATA::nl_res_base
```

Initial residual norm for the non-linear system.

5.52.2.15 nl_bestres

```
double PJFNK_DATA::nl_bestres
```

Best found residual norm.

5.52.2.16 eps

```
double PJFNK_DATA::eps =sqrt(DBL_EPSILON)
```

Value of epsilon used jacvec - default = sqrt(DBL_EPSILON)

5.52.2.17 NL_Output

```
bool PJFNK_DATA::NL_Output = true
```

True = print PJFNK messages to console.

5.52.2.18 L_Output

```
bool PJFNK_DATA::L_Output = false
```

True = print Linear messages to console.

5.52.2.19 LineSearch

```
bool PJFNK_DATA::LineSearch = false
```

True = use Backtracking Linesearch for global convergence.

5.52.2.20 Bounce

```
bool PJFNK_DATA::Bounce = false
```

True = allow Linesearch to go outside local well, False = Strict local convergence.

5.52.2.21 Converged

```
bool PJFNK_DATA::Converged = false
```

True = solution has converged, False = solution has not converged.

5.52.2.22 F

```
Matrix<double> PJFNK_DATA::F
```

Stored fuction evaluation at x (also the residual)

5.52.2.23 Fv

```
Matrix<double> PJFNK_DATA::Fv
```

Stored function evaluation at x+eps∗v.

5.52.2.24 v

```
Matrix<double> PJFNK_DATA::v
```

Stored vector of x+eps∗v.

5.52.2.25 x

```
Matrix<double> PJFNK_DATA::x
```

Current solution vector for the non-linear system.

5.52.2.26 bestx

```
Matrix<double> PJFNK_DATA::bestx
```

Best found solution vector to the non-linear system.

5.52.2.27 gmreslp_dat

```
GMRESLP_DATA PJFNK_DATA::gmreslp_dat
```

Data structure for the GMRESLP method.

5.52.2.28 pcg_dat

`PCG_DATA PJFNK_DATA::pcg_dat`

Data structure for the PCG method.

5.52.2.29 bicgstab_dat

`BiCGSTAB_DATA PJFNK_DATA::bicgstab_dat`

Data structure for the BiCGSTAB method.

5.52.2.30 cgs_dat

`CGS_DATA PJFNK_DATA::cgs_dat`

Data structure for the CGS method.

5.52.2.31 gmresrp_dat

`GMRESRP_DATA PJFNK_DATA::gmresrp_dat`

Data structure for the GMRESRP method.

5.52.2.32 gcr_dat

`GCR_DATA PJFNK_DATA::gcr_dat`

Data structure for the GCR method.

5.52.2.33 gmresr_dat

`GMRESR_DATA PJFNK_DATA::gmresr_dat`

Data structure for the GMRESR method.

5.52.2.34 kms_dat

`KMS_DATA PJFNK_DATA::kms_dat`

Data structure for the KMS method.

5.52.2.35 qr_dat

```
QR_DATA PJFNK_DATA::qr_dat
```

Data structure for the QR solve method.

5.52.2.36 backtrack_dat

```
BACKTRACK_DATA PJFNK_DATA::backtrack_dat
```

Data structure for the Backtracking Linesearch algorithm.

5.52.2.37 res_data

```
const void* PJFNK_DATA::res_data
```

Data structure pointer for user's residual data.

5.52.2.38 precon_data

```
const void* PJFNK_DATA::precon_data
```

Data structure pointer for user's preconditioning data.

5.52.2.39 funeval

```
int(* PJFNK_DATA::funeval) (const Matrix< double > &x, Matrix< double > &F, const void *res←
_data)
```

Function pointer for the user's function F(x) using there data.

5.52.2.40 precon

```
int(* PJFNK_DATA::precon) (const Matrix< double > &r, Matrix< double > &p, const void *precon←
_data)
```

Function pointer for the user's preconditioning function for the linear system.

The documentation for this struct was generated from the following file:

- lark.h

5.53 PURE_GAS Struct Reference

Data structure holding all the parameters for each pure gas spieces.

```
#include <egret.h>
```

Public Attributes

- double molecular_weight

  Given: molecular weights (g/mol)
- double Sutherland_Temp

  Given: Sutherland's Reference Temperature (K)
- double Sutherland_Const

  Given: Sutherland's Constant (K)
- double Sutherland_Viscosity

  Given: Sutherland's Reference Viscosity (g/cm/s)
- double specific_heat

  Given: Specific heat of the gas (J/g/K)
- double molecular_diffusion

  Calculated: molecular diffusivities (cm^2/s)
- double dynamic_viscosity

  Calculated: dynamic viscosities (g/cm/s)
- double density

  Calculated: gas densities (g/cm^3) {use RE3}.
- double Schmidt

  Calculated: Value of the Schmidt number (-)

5.53.1 Detailed Description

Data structure holding all the parameters for each pure gas spieces.

C-style object that holds the constants and parameters associated with each pure gas species in the overall mixture. This information is used in conjunction with the kinetic theory of gases to produce approximations to many different gas properties needed in simulating gas dynamics, mobility of a gas through porous media, as well as some kinetic adsorption parameters such as diffusivities.

5.53.2 Member Data Documentation

5.53.2.1 molecular_weight

```
double PURE_GAS::molecular_weight
```

Given: molecular weights (g/mol)

5.53.2.2 Sutherland_Temp

```
double PURE_GAS::Sutherland_Temp
```

Given: Sutherland's Reference Temperature (K)

5.53.2.3 Sutherland_Const

```
double PURE_GAS::Sutherland_Const
```

Given: Sutherland's Constant (K)

5.53.2.4 Sutherland_Viscosity

```
double PURE_GAS::Sutherland_Viscosity
```

Given: Sutherland's Reference Viscosity (g/cm/s)

5.53.2.5 specific_heat

```
double PURE_GAS::specific_heat
```

Given: Specific heat of the gas (J/g/K)

5.53.2.6 molecular_diffusion

```
double PURE_GAS::molecular_diffusion
```

Calculated: molecular diffusivities (cm^2/s)

5.53.2.7 dynamic_viscosity

```
double PURE_GAS::dynamic_viscosity
```

Calculated: dynamic viscosities (g/cm/s)

5.53.2.8 density

```
double PURE_GAS::density
```

Calculated: gas densities (g/cm^3) {use RE3}.

5.53.2.9 Schmidt

```
double PURE_GAS::Schmidt
```

Calculated: Value of the Schmidt number (-)

The documentation for this struct was generated from the following file:

- egret.h

5.54 QR_DATA Struct Reference

Data structure for the implementation of a QR solver given some invertable linear operator.

```
#include <lark.h>
```

Public Attributes

- Matrix< double > ek

  Unit vector used to extract columns from the linear operator.
- Matrix< double > Ro

  Upper triangular matrix formed from factoring the linear operator.
- Matrix< double > x

  Solution to the linear system.

5.54.1 Detailed Description

Data structure for the implementation of a QR solver given some invertable linear operator.

C-style object to be used in conjuction with a QR solver for invertable linear operators. This method will extract columns from the linear operator and use Householder Reflections to factor the operator into an upper triangular matrix and a unitary reflection matrix. It is generally less efficient to use this method for sparse systems, but is more stable and occassionally more efficient for dense systems.

5.54.2 Member Data Documentation

5.54.2.1 ek

```
Matrix<double> QR_DATA::ek
```

Unit vector used to extract columns from the linear operator.

5.54.2.2 Ro

```
Matrix<double> QR_DATA::Ro
```

Upper triangular matrix formed from factoring the linear operator.

5.54.2.3 x

```
Matrix<double> QR_DATA::x
```

Solution to the linear system.

The documentation for this struct was generated from the following file:

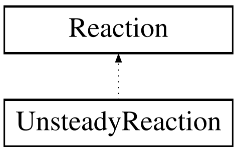
- lark.h

5.55 Reaction Class Reference

Reaction Object.

```
#include <shark.h>
```

Inheritance diagram for Reaction:



Public Member Functions

- Reaction ()

  Default constructor.
- ∼Reaction ()

  Default destructor.
- void Initialize_Object (MasterSpeciesList &List)

  Function to initialize the Reaction object from the MasterSpeciesList.
- void Display_Info ()

  Display the reaction information.
- void Set_Stoichiometric (int i, double v)

  Set the ith stoichiometric value.
- void Set_Equilibrium (double logK)

  Set the equilibrium constant in log(K) units.
- void Set_Enthalpy (double H)

  Set the enthalpy of the reaction (J/mol)
- void Set_Entropy (double S)

  Set the entropy of the reaction (J/K/mol)
- void Set_EnthalpyANDEntropy (double H, double S)

Set both the enthalpy and entropy (J/mol) & (J/K/mol)
- void Set_Energy (double G)

  Set the Gibb's free energy of reaction (J/mol)
- void checkSpeciesEnergies ()

  Function to check MasterList Reference for species energy info.
- void calculateEnergies ()
- void calculateEquilibrium (double T)

  Function to calculate the equilibrium constant based on temperature in K.
- bool haveEquilibrium ()

  Function to return true if equilibrium constant is given or can be calculated.
- double Get_Stoichiometric (int i)

  Fetch the ith stoichiometric value.
- double Get_Equilibrium ()

  Fetch the equilibrium constant (logK)
- double Get_Enthalpy ()

  Fetch the enthalpy of the reaction (J/mol)
- double Get_Entropy ()

  Fetch the entropy of the reaction (J/K/mol)
- double Get_Energy ()

  Fetch the energy of the reaction (J/mol)
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama)

Protected Attributes

- MasterSpeciesList ∗ List

  Pointer to a master species object.
- std::vector< double > Stoichiometric

  Vector of stoichiometric constants corresponding to species list.
- double Equilibrium

  Equilibrium constant for the reaction (logK)
- double enthalpy

  Reaction enthalpy (J/mol)
- double entropy

  Reaction entropy (J/K/mol)
- double energy

  Gibb's Free energy of reaction (J/mol)
- bool CanCalcHS

  True if all molecular info is avaiable to calculate dH and dS.
- bool CanCalcG

  True if all molecular info is available to calculate dG.
- bool HaveHS

  True if dH and dS is given, or can be calculated.
- bool HaveG

  True if dG is given, or can be calculated.
- bool HaveEquil

  True as long as Equilibrium is given, or can be calculated.

5.55.1 Detailed Description

Reaction Object.

C++ style object that holds data and functions associated with standard chemical reactions...
i.e., aA + bB <=> cC + dD
These reactions are assumed steady state and are characterized by stoichiometry coefficients and equilibrium/stability constants. Types of reactions that these are valid for would be acid/base reactions, metal-ligand complexation reactions, oxidation-reduction reactions, Henry's Law phase changes, and more. Reactions that this may not be suitable for include mechanisms, adsorption, and precipitation. Those types of reactions would be better handled by more specific objects that inherit from this object.
If all species in the reaction are registered and known species in mola.h AND have known formation energies, then the equilibrium constants for that particular reaction will be calculated based on the species involved in the reaction. However, if using some custom molecule objects, then the reaction equilibrium may not be able to be automatically formed by the routine. In this case, you would need to also supply the equilibrium constant for the particular reaction.

5.55.2 Constructor & Destructor Documentation

5.55.2.1 Reaction()

```
Reaction::Reaction ( )
```

Default constructor.

5.55.2.2 ∼Reaction()

```
Reaction::∼Reaction ( )
```

Default destructor.

5.55.3 Member Function Documentation

5.55.3.1 Initialize_Object()

```
void Reaction::Initialize_Object (
            MasterSpeciesList & List )
```

Function to initialize the Reaction object from the MasterSpeciesList.

5.55.3.2 Display_Info()

```
void Reaction::Display_Info ( )
```

Display the reaction information.

5.55.3.3 Set_Stoichiometric()

```
void Reaction::Set_Stoichiometric (
            int i,
            double v )
```

Set the ith stoichiometric value.

This function will set the stoichiometric constant of the ith species in the master list to the given value of v. All values of v are set to zero unless overriden by this function.

Parameters

i	index of the species in the MasterSpeciesList
v	value of the stoichiometric constant for that species in the reaction

5.55.3.4 Set_Equilibrium()

```
void Reaction::Set_Equilibrium (
             double logK )
```

Set the equilibrium constant in log(K) units.

5.55.3.5 Set_Enthalpy()

```
void Reaction::Set_Enthalpy (
             double H )
```

Set the enthalpy of the reaction (J/mol)

5.55.3.6 Set_Entropy()

```
void Reaction::Set_Entropy (
             double S )
```

Set the entropy of the reaction (J/K/mol)

5.55.3.7 Set_EnthalpyANDEntropy()

```
void Reaction::Set_EnthalpyANDEntropy (
             double H,
             double S )
```

Set both the enthalpy and entropy (J/mol) & (J/K/mol)

5.55.3.8 Set_Energy()

```
void Reaction::Set_Energy (
             double G )
```

Set the Gibb's free energy of reaction (J/mol)

5.55.3.9 checkSpeciesEnergies()

```
void Reaction::checkSpeciesEnergies ( )
```

Function to check MasterList Reference for species energy info.

This function will go through the stoichiometry of this reaction and check the molecules in the MasterSpeciesList that correspond to the species present in this reaction for the existance of their formation energies. Based on the states of those energies, it will note internally whether or not it can determine the equilibrium constants based soley on individual species information. If it cannot, then the user must provide either the reaction energies to form the equilibrium constant or the equilibrium constant itself. Function to calculate and set the energy of the reaction

5.55.3.10 calculateEnergies()

```
void Reaction::calculateEnergies ( )
```

If the energies of the reaction can be determined from the individual species in the reaction, then this function uses that information. Otherwise, it sets the energies equal to the constants given to the object by the user.

5.55.3.11 calculateEquilibrium()

```
void Reaction::calculateEquilibrium (
             double T )
```

Function to calculate the equilibrium constant based on temperature in K.

5.55.3.12 haveEquilibrium()

```
bool Reaction::haveEquilibrium ( )
```

Function to return true if equilibrium constant is given or can be calculated.

5.55.3.13 Get_Stoichiometric()

```
double Reaction::Get_Stoichiometric (
             int i )
```

Fetch the ith stoichiometric value.

5.55.3.14 Get_Equilibrium()

```
double Reaction::Get_Equilibrium ( )
```

Fetch the equilibrium constant (logK)

5.55.3.15 Get_Enthalpy()

```
double Reaction::Get_Enthalpy ( )
```

Fetch the enthalpy of the reaction (J/mol)

5.55.3.16 Get_Entropy()

```
double Reaction::Get_Entropy ( )
```

Fetch the entropy of the reaction (J/K/mol)

5.55.3.17 Get_Energy()

```
double Reaction::Get_Energy ( )
```

Fetch the energy of the reaction (J/mol)

Evaluate a residual for the reaction given variable x=log(C) and activity coefficients gama

5.55.3.18 Eval_Residual()

```
double Reaction::Eval_Residual (
             const Matrix< double > & x,
             const Matrix< double > & gama )
```

This function will calculate the reaction residual from this object's stoichiometry, equilibrium constant, log(C) concentrations, and activity coefficients.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step

5.55.4 Member Data Documentation

5.55.4.1 List

```
MasterSpeciesList* Reaction::List  [protected]
```

Pointer to a master species object.

5.55.4.2 Stoichiometric

```
std::vector<double> Reaction::Stoichiometric  [protected]
```

Vector of stoichiometric constants corresponding to species list.

5.55.4.3 Equilibrium

```
double Reaction::Equilibrium  [protected]
```

Equilibrium constant for the reaction (logK)

5.55.4.4 enthalpy

```
double Reaction::enthalpy  [protected]
```

Reaction enthalpy (J/mol)

5.55.4.5 entropy

```
double Reaction::entropy  [protected]
```

Reaction entropy (J/K/mol)

5.55.4.6 energy

```
double Reaction::energy  [protected]
```

Gibb's Free energy of reaction (J/mol)

5.55.4.7 CanCalcHS

```
bool Reaction::CanCalcHS  [protected]
```

True if all molecular info is avaiable to calculate dH and dS.

5.55.4.8 CanCalcG

```
bool Reaction::CanCalcG  [protected]
```

True if all molecular info is available to calculate dG.

5.55.4.9 HaveHS

```
bool Reaction::HaveHS  [protected]
```

True if dH and dS is given, or can be calculated.

5.55.4.10 HaveG

```
bool Reaction::HaveG  [protected]
```

True if dG is given, or can be calculated.

5.55.4.11 HaveEquil

```
bool Reaction::HaveEquil  [protected]
```

True as long as Equilibrium is given, or can be calculated.

The documentation for this class was generated from the following file:

- shark.h

5.56 SCOPSOWL_DATA Struct Reference

Primary data structure for SCOPSOWL simulations.

```
#include <scopsowl.h>
```

Public Attributes

- unsigned long int total_steps

  Running total of all calculation steps.
- int coord_macro

  Coordinate system for large pellet.
- int coord_micro

  Coordinate system for small crystal (if any)
- int level = 2

  Level of coupling between the different scales (default = 2)
- double sim_time

  Stopping time for the simulation (hrs)
- double t_old

  Old time of the simulations (hrs)
- double t

  Current time of the simulations (hrs)
- double t_counter = 0.0

  Counter for the time output.
- double t_print

Print output at every t_print time (hrs)

- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- bool SurfDiff = true

  True = includes SKUA simulation if Heterogeneous; False = only uses MAGPIE.
- bool Heterogeneous = true

  True = pellet is made of binder and crystals, False = all one phase.
- double gas_velocity

  Superficial Gas Velocity arount pellet (cm/s)
- double total_pressure

  Gas phase total pressure (kPa)
- double gas_temperature

  Gas phase temperature (K)
- double pellet_radius

  Nominal radius of the pellet - macroscale domain (cm)
- double crystal_radius

  Nominal radius of the crystal - microscale domain (um)
- double char_macro

  Characteristic size for macro scale (cm or cm^2) - only if pellet is not spherical.
- double char_micro

  Characteristic size for micro scale (um or um^2) - only if crystal is not spherical.
- double binder_fraction

  Volume of binder per total volume of pellet (-)
- double binder_porosity

  Volume of pores per volume of binder (-)
- double binder_poresize

  Nominal radius of the binder pores (cm)
- double pellet_density

  Mass of the pellet per volume of pellet (kg/L)
- bool DirichletBC = false

  True = Dirichlet BC; False = Neumann BC.
- bool NonLinear = true

  True = Non-linear solver; False = Linear solver.
- std::vector< double > y

  Outside mole fractions of each component (-)
- std::vector< double > tempy

  Temporary place holder for gas mole fractions in other locations (-)
- FILE ∗ OutputFile

  Output file pointer to the output file for postprocesses.
- double(∗ eval_ads)(int i, int l, const void ∗user_data)

  Function pointer for evaluating adsorption (mol/kg)
- double(∗ eval_retard)(int i, int l, const void ∗user_data)

  Function pointer for evaluating retardation (-)
- double(∗ eval_diff)(int i, int l, const void ∗user_data)

  Function pointer for evaluating pore diffusion (cm^2/hr)
- double(∗ eval_surfDiff)(int i, int l, const void ∗user_data)

  Function pointer for evaluating surface diffusion (um^2/hr)
- double(∗ eval_kf)(int i, const void ∗user_data)

  Function pointer for evaluating film mass transfer (cm/hr)

- const void ∗ user_data

  Data structure for users info to calculate parameters.
- MIXED_GAS ∗ gas_dat

  Pointer to the MIXED_GAS data structure (may or may not be used)
- MAGPIE_DATA magpie_dat

  Data structure for a magpie problem (to be used if not using skua)
- std::vector< FINCH_DATA > finch_dat

  Data structure for pore adsorption kinetics for all species (u in mol/L)
- std::vector< SCOPSOWL_PARAM_DATA > param_dat

  Data structure for parameter info for all species.
- std::vector< SKUA_DATA > skua_dat

  Data structure holding a skua object for all nodes (each skua has an object for each species)

5.56.1 Detailed Description

Primary data structure for SCOPSOWL simulations.

C-style object holding necessary information to run a SCOPSOWL simulation. SCOPSOWL is a multi-scale problem involving PDE solution for the macro-scale adsorbent pellet and the micro-scale adsorbent crystals. As such, each SCOPSOWL simulation involves multiple SKUA simulations at the nodes in the macro-scale domain. Alternatively, if the user wishes to specify that the adsorbent is homogeneous, then you can run SCOPSOWL as a single-scale problem. Additionally, you can simplfy the model by assuming that the micro-scale diffusion is very fast, and therefore replace each SKUA simulation with a simpler MAGPIE evaluation. Details on running SCOPSOWL with the various options will be discussed in the SCOPSOWL_SCENARIOS function.

5.56.2 Member Data Documentation

5.56.2.1 total_steps

```
unsigned long int SCOPSOWL_DATA::total_steps
```

Running total of all calculation steps.

5.56.2.2 coord_macro

```
int SCOPSOWL_DATA::coord_macro
```

Coordinate system for large pellet.

5.56.2.3 coord_micro

```
int SCOPSOWL_DATA::coord_micro
```

Coordinate system for small crystal (if any)

5.56.2.4 level

```
int SCOPSOWL_DATA::level = 2
```

Level of coupling between the different scales (default = 2)

5.56.2.5 sim_time

```
double SCOPSOWL_DATA::sim_time
```

Stopping time for the simulation (hrs)

5.56.2.6 t_old

```
double SCOPSOWL_DATA::t_old
```

Old time of the simulations (hrs)

5.56.2.7 t

```
double SCOPSOWL_DATA::t
```

Current time of the simulations (hrs)

5.56.2.8 t_counter

```
double SCOPSOWL_DATA::t_counter = 0.0
```

Counter for the time output.

5.56.2.9 t_print

```
double SCOPSOWL_DATA::t_print
```

Print output at every t_print time (hrs)

5.56.2.10 Print2File

```
bool SCOPSOWL_DATA::Print2File = true
```

True = results to .txt; False = no printing.

5.56.2.11 Print2Console

```
bool SCOPSOWL_DATA::Print2Console = true
```

True = results to console; False = no printing.

5.56.2.12 SurfDiff

```
bool SCOPSOWL_DATA::SurfDiff = true
```

True = includes SKUA simulation if Heterogeneous; False = only uses MAGPIE.

5.56.2.13 Heterogeneous

```
bool SCOPSOWL_DATA::Heterogeneous = true
```

True = pellet is made of binder and crystals, False = all one phase.

5.56.2.14 gas_velocity

```
double SCOPSOWL_DATA::gas_velocity
```

Superficial Gas Velocity arount pellet (cm/s)

5.56.2.15 total_pressure

```
double SCOPSOWL_DATA::total_pressure
```

Gas phase total pressure (kPa)

5.56.2.16 gas_temperature

```
double SCOPSOWL_DATA::gas_temperature
```

Gas phase temperature (K)

5.56.2.17 pellet_radius

```
double SCOPSOWL_DATA::pellet_radius
```

Nominal radius of the pellet - macroscale domain (cm)

5.56.2.18 crystal_radius

```
double SCOPSOWL_DATA::crystal_radius
```

Nominal radius of the crystal - microscale domain (um)

5.56.2.19 char_macro

```
double SCOPSOWL_DATA::char_macro
```

Characteristic size for macro scale (cm or cm^2) - only if pellet is not spherical.

5.56.2.20 char_micro

```
double SCOPSOWL_DATA::char_micro
```

Characteristic size for micro scale (um or um^2) - only if crystal is not spherical.

5.56.2.21 binder_fraction

```
double SCOPSOWL_DATA::binder_fraction
```

Volume of binder per total volume of pellet (-)

5.56.2.22 binder_porosity

```
double SCOPSOWL_DATA::binder_porosity
```

Volume of pores per volume of binder (-)

5.56.2.23 binder_poresize

```
double SCOPSOWL_DATA::binder_poresize
```

Nominal radius of the binder pores (cm)

5.56.2.24 pellet_density

```
double SCOPSOWL_DATA::pellet_density
```

Mass of the pellet per volume of pellet (kg/L)

5.56.2.25 DirichletBC

```
bool SCOPSOWL_DATA::DirichletBC = false
```

True = Dirichlet BC; False = Neumann BC.

5.56.2.26 NonLinear

```
bool SCOPSOWL_DATA::NonLinear = true
```

True = Non-linear solver; False = Linear solver.

5.56.2.27 y

```
std::vector<double> SCOPSOWL_DATA::y
```

Outside mole fractions of each component (-)

5.56.2.28 tempy

```
std::vector<double> SCOPSOWL_DATA::tempy
```

Temporary place holder for gas mole fractions in other locations (-)

5.56.2.29 OutputFile

```
FILE* SCOPSOWL_DATA::OutputFile
```

Output file pointer to the output file for postprocesses.

5.56.2.30 eval_ads

```
double(* SCOPSOWL_DATA::eval_ads) (int i, int l, const void *user_data)
```

Function pointer for evaluating adsorption (mol/kg)

5.56.2.31 eval_retard

```
double(* SCOPSOWL_DATA::eval_retard) (int i, int l, const void *user_data)
```

Function pointer for evaluating retardation (-)

5.56.2.32 eval_diff

```
double(* SCOPSOWL_DATA::eval_diff) (int i, int l, const void *user_data)
```

Function pointer for evaluating pore diffusion (cm^2/hr)

5.56.2.33 eval_surfDiff

```
double(* SCOPSOWL_DATA::eval_surfDiff) (int i, int l, const void *user_data)
```

Function pointer for evaluating surface diffusion (um^2/hr)

5.56.2.34 eval_kf

```
double(* SCOPSOWL_DATA::eval_kf) (int i, const void *user_data)
```

Function pointer for evaluating film mass transfer (cm/hr)

5.56.2.35 user_data

```
const void* SCOPSOWL_DATA::user_data
```

Data structure for users info to calculate parameters.

5.56.2.36 gas_dat

```
MIXED_GAS* SCOPSOWL_DATA::gas_dat
```

Pointer to the MIXED_GAS data structure (may or may not be used)

5.56.2.37 magpie_dat

```
MAGPIE_DATA SCOPSOWL_DATA::magpie_dat
```

Data structure for a magpie problem (to be used if not using skua)

5.56.2.38 finch_dat

```
std::vector<FINCH_DATA> SCOPSOWL_DATA::finch_dat
```

Data structure for pore adsorption kinetics for all species (u in mol/L)

5.56.2.39 param_dat

```
std::vector<SCOPSOWL_PARAM_DATA> SCOPSOWL_DATA::param_dat
```

Data structure for parameter info for all species.

5.56.2.40 skua_dat

```
std::vector<SKUA_DATA> SCOPSOWL_DATA::skua_dat
```

Data structure holding a skua object for all nodes (each skua has an object for each species)

The documentation for this struct was generated from the following file:

- scopsowl.h

5.57 SCOPSOWL_OPT_DATA Struct Reference

Data structure for the SCOPSOWL optmization routine.

```
#include <scopsowl_opt.h>
```

Public Attributes

- int num_curves

  Number of adsorption curves to analyze.
- int evaluation

  Number of times the eval function has been called for a single curve.
- unsigned long int total_eval

  Total number of evaluations needed for completion.
- int current_points

  Number of points in the current curve.
- int num_params = 1

  Number of adjustable parameters for the current curve (currently only supports 1)
- int diffusion_type

  Flag to identify type of diffusion function to use.
- int adsorb_index

  Component index for adsorbable species.
- int max_guess_iter = 20

  Maximum allowed guess iterations (default = 20)
- bool Optimize

  True = run optimization, False = run a comparison.
- bool Rough

  True = use only a rough estimate, False = run full optimization.
- double current_temp

  Temperature for current curve.
- double current_press

  Partial pressure for current curve.

- double current_equil

  Equilibrium data point for the current curve.
- double simulation_equil

  Equilibrium simulation point for the current curve.
- double max_bias

  Positive maximum bias plausible for fitting.
- double min_bias

  Negative minimum bias plausible for fitting.
- double e_norm

  Euclidean norm of current fit.
- double f_bias

  Function bias of current fit.
- double e_norm_old

  Euclidean norm of the previous fit.
- double f_bias_old

  Function bias of the previous fit.
- double param_guess

  Parameter guess for the surface/crystal diffusivity.
- double param_guess_old

  Parameter guess for the previous curve.
- double rel_tol_norm = 0.01

  Tolerance for convergence of the guess norm.
- double abs_tol_bias = 1.0

  Tolerance for convergence of the guess bias.
- std::vector< double > y_base

  Gas phase mole fractions in absense of adsorbing species.
- std::vector< double > q_data

  Amount adsorbed at a particular point in current curve.
- std::vector< double > q_sim

  Amount adsorbed based on the simulation.
- std::vector< double > t

  Time points in the current curve.
- FILE ∗ ParamFile

  Output file for parameter results.
- FILE ∗ CompareFile

  Output file for comparison of results.
- SCOPSOWL_DATA owl_dat

  Data structure for the SCOPSOWL simulation.

5.57.1 Detailed Description

Data structure for the SCOPSOWL optmization routine.

C-style object holding information about the optimization routine as well as the standard SCOPSOwl_DATA structure for SCOPSOWL simulations.

5.57.2 Member Data Documentation

5.57.2.1 num_curves

```
int SCOPSOWL_OPT_DATA::num_curves
```

Number of adsorption curves to analyze.

5.57.2.2 evaluation

```
int SCOPSOWL_OPT_DATA::evaluation
```

Number of times the eval function has been called for a single curve.

5.57.2.3 total_eval

```
unsigned long int SCOPSOWL_OPT_DATA::total_eval
```

Total number of evaluations needed for completion.

5.57.2.4 current_points

```
int SCOPSOWL_OPT_DATA::current_points
```

Number of points in the current curve.

5.57.2.5 num_params

```
int SCOPSOWL_OPT_DATA::num_params = 1
```

Number of adjustable parameters for the current curve (currently only supports 1)

5.57.2.6 diffusion_type

```
int SCOPSOWL_OPT_DATA::diffusion_type
```

Flag to identify type of diffusion function to use.

5.57.2.7 adsorb_index

```
int SCOPSOWL_OPT_DATA::adsorb_index
```

Component index for adsorbable species.

5.57.2.8 max_guess_iter

```
int SCOPSOWL_OPT_DATA::max_guess_iter = 20
```

Maximum allowed guess iterations (default = 20)

5.57.2.9 Optimize

```
bool SCOPSOWL_OPT_DATA::Optimize
```

True = run optimization, False = run a comparison.

5.57.2.10 Rough

```
bool SCOPSOWL_OPT_DATA::Rough
```

True = use only a rough estimate, False = run full optimization.

5.57.2.11 current_temp

```
double SCOPSOWL_OPT_DATA::current_temp
```

Temperature for current curve.

5.57.2.12 current_press

```
double SCOPSOWL_OPT_DATA::current_press
```

Partial pressure for current curve.

5.57.2.13 current_equil

```
double SCOPSOWL_OPT_DATA::current_equil
```

Equilibrium data point for the current curve.

5.57.2.14 simulation_equil

```
double SCOPSOWL_OPT_DATA::simulation_equil
```

Equilibrium simulation point for the current curve.

5.57.2.15 max_bias

```
double SCOPSOWL_OPT_DATA::max_bias
```

Positive maximum bias plausible for fitting.

5.57.2.16 min_bias

```
double SCOPSOWL_OPT_DATA::min_bias
```

Negative minimum bias plausible for fitting.

5.57.2.17 e_norm

```
double SCOPSOWL_OPT_DATA::e_norm
```

Euclidean norm of current fit.

5.57.2.18 f_bias

```
double SCOPSOWL_OPT_DATA::f_bias
```

Function bias of current fit.

5.57.2.19 e_norm_old

```
double SCOPSOWL_OPT_DATA::e_norm_old
```

Euclidean norm of the previous fit.

5.57.2.20 f_bias_old

```
double SCOPSOWL_OPT_DATA::f_bias_old
```

Function bias of the previous fit.

5.57.2.21 param_guess

```
double SCOPSOWL_OPT_DATA::param_guess
```

Parameter guess for the surface/crystal diffusivity.

5.57.2.22 param_guess_old

```
double SCOPSOWL_OPT_DATA::param_guess_old
```

Parameter guess for the previous curve.

5.57.2.23 rel_tol_norm

```
double SCOPSOWL_OPT_DATA::rel_tol_norm = 0.01
```

Tolerance for convergence of the guess norm.

5.57.2.24 abs_tol_bias

```
double SCOPSOWL_OPT_DATA::abs_tol_bias = 1.0
```

Tolerance for convergence of the guess bias.

5.57.2.25 y_base

```
std::vector<double> SCOPSOWL_OPT_DATA::y_base
```

Gas phase mole fractions in absense of adsorbing species.

5.57.2.26 q_data

```
std::vector<double> SCOPSOWL_OPT_DATA::q_data
```

Amount adsorbed at a particular point in current curve.

5.57.2.27 q_sim

```
std::vector<double> SCOPSOWL_OPT_DATA::q_sim
```

Amount adsorbed based on the simulation.

5.57.2.28 t

```
std::vector<double> SCOPSOWL_OPT_DATA::t
```

Time points in the current curve.

5.57.2.29 ParamFile

```
FILE* SCOPSOWL_OPT_DATA::ParamFile
```

Output file for parameter results.

5.57.2.30 CompareFile

```
FILE* SCOPSOWL_OPT_DATA::CompareFile
```

Output file for comparison of results.

5.57.2.31 owl_dat

```
SCOPSOWL_DATA SCOPSOWL_OPT_DATA::owl_dat
```

Data structure for the SCOPSOWL simulation.

The documentation for this struct was generated from the following file:

- scopsowl_opt.h

5.58 SCOPSOWL_PARAM_DATA Struct Reference

Data structure for the species' parameters in SCOPSOWL.

```
#include <scopsowl.h>
```

Public Attributes

- Matrix< double > qAvg

    Average adsorbed amount for a species at each node (mol/kg)
- Matrix< double > qAvg_old

    Old Average adsorbed amount for a species at each node (mol/kg)
- Matrix< double > Qst

    Heat of adsorption for all nodes (J/mol)
- Matrix< double > Qst_old

    Old Heat of adsorption for all nodes (J/mol)
- Matrix< double > dq_dc

    Storage vector for current adsorption slope/strength (dq/dc) (L/kg)
- double xIC

    Initial conditions for adsorbed molefractions.
- double qIntegralAvg

    Integral average of adsorption over the entire pellet (mol/kg)
- double qIntegralAvg_old

    Old Integral average of adsorption over the entire pellet (mol/kg)
- double QstAvg

Integral average heat of adsorption (J/mol)
- double QstAvg_old

  Old integral average heat of adsorption (J/mol)
- double qo

  Boundary value of adsorption if using Dirichlet BCs (mol/kg)
- double Qsto

  Boundary value of adsorption heat if using Dirichlet BCs (J/mol)
- double dq_dco

  Boundary value of adsorption slope for Dirichelt BCs (L/kg)
- double pore_diffusion

  Value for constant pore diffusion (cm^2/hr)
- double film_transfer

  Value for constant film mass transfer (cm/hr)
- double activation_energy

  Activation energy for surface diffusion (J/mol)
- double ref_diffusion

  Reference state surface diffusivity (um^2/hr)
- double ref_temperature

  Reference temperature for empirical adjustments (K)
- double affinity

  Affinity parameter used in empirical adjustments (-)
- double ref_pressure
- bool Adsorbable

  True = species can adsorb; False = species cannot adsorb.
- std::string speciesName

  String to hold the name of each species.

5.58.1 Detailed Description

Data structure for the species' parameters in SCOPSOWL.

C-style object that holds information on all species for a particular SCOPSOWL simulation. Initial conditions, kinetic parameters, and interim matrix objects are stored here for use in various SCOSPSOWL functions.

5.58.2 Member Data Documentation

5.58.2.1 qAvg

```
Matrix<double> SCOPSOWL_PARAM_DATA::qAvg
```

Average adsorbed amount for a species at each node (mol/kg)

5.58.2.2 qAvg_old

```
Matrix<double> SCOPSOWL_PARAM_DATA::qAvg_old
```

Old Average adsorbed amount for a species at each node (mol/kg)

5.58.2.3 Qst

```
Matrix<double> SCOPSOWL_PARAM_DATA::Qst
```

Heat of adsorption for all nodes (J/mol)

5.58.2.4 Qst_old

```
Matrix<double> SCOPSOWL_PARAM_DATA::Qst_old
```

Old Heat of adsorption for all nodes (J/mol)

5.58.2.5 dq_dc

```
Matrix<double> SCOPSOWL_PARAM_DATA::dq_dc
```

Storage vector for current adsorption slope/strength (dq/dc) (L/kg)

5.58.2.6 xIC

```
double SCOPSOWL_PARAM_DATA::xIC
```

Initial conditions for adsorbed molefractions.

5.58.2.7 qIntegralAvg

```
double SCOPSOWL_PARAM_DATA::qIntegralAvg
```

Integral average of adsorption over the entire pellet (mol/kg)

5.58.2.8 qIntegralAvg_old

```
double SCOPSOWL_PARAM_DATA::qIntegralAvg_old
```

Old Integral average of adsorption over the entire pellet (mol/kg)

5.58.2.9 QstAvg

```
double SCOPSOWL_PARAM_DATA::QstAvg
```

Integral average heat of adsorption (J/mol)

5.58.2.10 QstAvg_old

`double SCOPSOWL_PARAM_DATA::QstAvg_old`

Old integral average heat of adsorption (J/mol)

5.58.2.11 qo

`double SCOPSOWL_PARAM_DATA::qo`

Boundary value of adsorption if using Dirichlet BCs (mol/kg)

5.58.2.12 Qsto

`double SCOPSOWL_PARAM_DATA::Qsto`

Boundary value of adsorption heat if using Dirichlet BCs (J/mol)

5.58.2.13 dq_dco

`double SCOPSOWL_PARAM_DATA::dq_dco`

Boundary value of adsorption slope for Dirichelt BCs (L/kg)

5.58.2.14 pore_diffusion

`double SCOPSOWL_PARAM_DATA::pore_diffusion`

Value for constant pore diffusion (cm^2/hr)

5.58.2.15 film_transfer

`double SCOPSOWL_PARAM_DATA::film_transfer`

Value for constant film mass transfer (cm/hr)

5.58.2.16 activation_energy

`double SCOPSOWL_PARAM_DATA::activation_energy`

Activation energy for surface diffusion (J/mol)

5.58.2.17 ref_diffusion

```
double SCOPSOWL_PARAM_DATA::ref_diffusion
```

Reference state surface diffusivity (um^2/hr)

5.58.2.18 ref_temperature

```
double SCOPSOWL_PARAM_DATA::ref_temperature
```

Reference temperature for empirical adjustments (K)

5.58.2.19 affinity

```
double SCOPSOWL_PARAM_DATA::affinity
```

Affinity parameter used in empirical adjustments (-)

5.58.2.20 ref_pressure

```
double SCOPSOWL_PARAM_DATA::ref_pressure
```

5.58.2.21 Adsorbable

```
bool SCOPSOWL_PARAM_DATA::Adsorbable
```

True = species can adsorb; False = species cannot adsorb.

5.58.2.22 speciesName

```
std::string SCOPSOWL_PARAM_DATA::speciesName
```

String to hold the name of each species.

The documentation for this struct was generated from the following file:

- scopsowl.h

5.59 SHARK_DATA Struct Reference

Data structure for SHARK simulations.

```
#include <shark.h>
```

Public Attributes

- MasterSpeciesList MasterList

  Master List of species object.
- std::vector< Reaction > ReactionList

  Equilibrium reaction objects.
- std::vector< MassBalance > MassBalanceList

  Mass balance objects.
- std::vector< UnsteadyReaction > UnsteadyList

  Unsteady Reaction objects.
- std::vector< AdsorptionReaction > AdsorptionList

  Equilibrium Adsorption Reaction Objects.
- std::vector< UnsteadyAdsorption > UnsteadyAdsList

  Unsteady Adsorption Reaction Objects.
- std::vector< MultiligandAdsorption > MultiAdsList

  Multiligand Adsorptioin Objects.
- std::vector< ChemisorptionReaction > ChemisorptionList

  Chemisorption Reaction objects.
- std::vector< MultiligandChemisorption > MultiChemList

  Multiligand Chemisorption Reaction Objects.
- std::vector< double(∗)(const Matrix< double > &x, SHARK_DATA ∗shark_dat, const void ∗data) > OtherList

  Array of Other Residual functions to be defined by user.
- int numvar

  Total number of functions and species.
- int num_ssr

  Number of steady-state reactions.
- int num_mbe

  Number of mass balance equations.
- int num_usr = 0

  Number of unsteady-state reactions.
- int num_ssao = 0

  Number of steady-state adsorption objects.
- int num_usao = 0

  Number of unsteady adsorption objects.
- int num_multi_ssao = 0

  Number of multiligand steady-state adsorption objects.
- int num_sschem = 0

  Number of steady-state chemisorption objects.
- int num_multi_sschem = 0

  Number of multiligand steady-state chemisorption objects.
- std::vector< int > num_ssar

  List of the numbers of reactions in each adsorption object.
- std::vector< int > num_usar

  List of the numbers of reactions in each unsteady adsorption object.
- std::vector< int > num_sschem_rxns

  List of the numbers of reactions in each steady-state chemisorption object.
- std::vector< std::vector< int > > num_multi_ssar

  List of all multiligand objects -> List of ligands and rxns of that ligand.
- std::vector< std::vector< int > > num_multichem_rxns

  List of all multiligand chemisorption objects -> List of num rxns for that ligand.
- std::vector< std::string > ss_ads_names

List of the steady-state adsorbent object names.
- std::vector< std::string > us_ads_names

  List of the unsteady adsorption object names.
- std::vector< std::string > ss_chem_names

  List of the steady-state chemisorption object names.
- std::vector< std::vector< std::string > > ssmulti_names

  List of the names of the ligands in each multiligand object.
- std::vector< std::vector< std::string > > ssmultichem_names

  List of the names of the ligands in each multiligand chemisorption object.
- int num_other = 0

  Number of other functions to be used (default is always 0)
- int act_fun = IDEAL

  Flag denoting the activity function to use (default is IDEAL)
- int reactor_type = BATCH

  Flag denoting the type of reactor considered for the system (default is BATCH)
- int totalsteps = 0

  Total number of iterations.
- int totalcalls = 0

  Total number of residual function calls.
- int timesteps = 0

  Number of time steps taken to complete simulation.
- int pH_index = -1

  Contains the index of the pH variable (set internally)
- int pOH_index = -1

  Contains the index of the pOH variable (set internally)
- double simulationtime = 0.0

  Time to simulate unsteady reactions for (default = 0.0 hrs)
- double dt = 0.1

  Time step size (hrs)
- double dt_min = sqrt(DBL_EPSILON)

  Minimum allowable step size.
- double dt_max = 744.0

  Maximum allowable step size (∼1 month in time)
- double t_out = 0.0

  Time increment by which file output is made (default = print all time steps)
- double t_count = 0.0

  Running count of time increments.
- double time = 0.0

  Current value of time (starts from t = 0.0 hrs)
- double time_old = 0.0

  Previous value of time (start from t = 0.0 hrs)
- double pH = 7.0

  Value of pH if needed (default = 7)
- double pH_step = 0.5

  Value by which to increment pH when doing a speciation curve (default = 0.5)
- double pH_end = 14.0

  Value at which we stop doing the speciation curve as a function of pH (default = 14.0)
- double start_temp = 277.15

  Value of the starting temperature used for Temperature Curves (default = 277.15 K)
- double end_temp = 323.15

  Value of the ending temperature used for Temperature Curves (default = 323.15 K)

- double temp_step = 10.0

  Size of the step changes to use for Temperature Curves (default = 10.0 K);.
- double volume = 1.0

  Volume of the domain in liters (default = 1 L)
- double flow_rate = 1.0

  Flow rate in the reactor in L/hr (default = 1 L/hr)
- double xsec_area = 1.0

  Cross sectional area of the reactor in m^2 (default = 1 m^2)
- double Norm = 0.0

  Current value of euclidean norm in solution.
- double dielectric_const = 78.325

  Dielectric constant used in many activity models (default: water = 78.325 (1/K))
- double relative_permittivity = 80.1

  Relative permittivity of the medium (default: water = 80.1 (-))
- double temperature = 298.15

  Solution temperature (default = 25 oC or 298.15 K)
- double ionic_strength = 0.0

  Solution ionic strength in Molar (calculated internally)
- bool steadystate = true

  True = solve steady problem; False = solve transient problem.
- bool ZeroInitialSolids = false

  True = no solids or adsorption initially in the reactor.
- bool TimeAdaptivity = false

  True = solve using variable time step.
- bool const_pH = false

  True = set pH to a constant; False = solve for pH.
- bool SpeciationCurve = false

  True = runs a series of constant pH steady-state problems to produce curves.
- bool TemperatureCurve = false

  True = runs a series of constant temperature steady-state problmes to produce curves.
- bool Console_Output = true

  True = display output to console.
- bool File_Output = false

  True = write output to a file.
- bool Contains_pH = false

  True = system contains pH as a variable (set internally)
- bool Contains_pOH = false

  True = system contains pOH as a variable (set internally)
- bool Converged = false

  True = system converged within tolerance.
- bool LocalMin = true

  True = allow the system to settle for a local minimum if tolerance not reached.
- Matrix< double > X_old

  Solution vector for old time step - log(C)
- Matrix< double > X_new

  Solution vector for current time step - log(C)
- Matrix< double > Conc_old

  Concentration vector for old time step - 10^x.
- Matrix< double > Conc_new

  Concentration vector for current time step - 10^x.
- Matrix< double > activity_new

Activity matrix for current time step.

- Matrix< double > activity_old

  Activity matrix from prior time step.
- int(∗ EvalActivity)(const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Function pointer to evaluate activity coefficients.
- int(∗ Residual)(const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Function pointer to evaluate all residuals in the system.
- int(∗ lin_precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗data)

  Function pointer to form a linear preconditioning operation for the Jacobian.
- PJFNK_DATA Newton_data

  Data structure for the Newton-Krylov solver (see lark.h)
- const void ∗ activity_data

  User defined data structure for an activity model.
- const void ∗ residual_data

  User defined data structure for the residual function.
- const void ∗ precon_data

  User defined data structure for preconditioning.
- const void ∗ other_data

  User define data structure used for user defined residuals.
- FILE ∗ OutputFile

  Output File pointer.
- yaml_cpp_class yaml_object

  yaml object to read and access digitized yaml documents (see yaml_wrapper.h)

5.59.1 Detailed Description

Data structure for SHARK simulations.

C-style object holding data and function pointers associated with solving aqueous speciation and reaction kinetics. This object couples all other objects available in shark.h in order to provide residual calculations for each individual function that makes up the overall system model. Those residuals are brought together inside the residual function and fed into the lark.h PJFNK solver routine. That solver then attempts to find a solution to all non-linear variables simultaneously. Any function or data pointers in this structure can be overriden to change how you interface with and solve the problem. Users may also provide a set of custom residual functions through the "OtherList" vector object. Those residual function must all have the same format.

5.59.2 Member Data Documentation

5.59.2.1 MasterList

```
MasterSpeciesList SHARK_DATA::MasterList
```

Master List of species object.

5.59.2.2 ReactionList

```
std::vector<Reaction> SHARK_DATA::ReactionList
```

Equilibrium reaction objects.

5.59.2.3 MassBalanceList

```
std::vector<MassBalance> SHARK_DATA::MassBalanceList
```

Mass balance objects.

5.59.2.4 UnsteadyList

```
std::vector<UnsteadyReaction> SHARK_DATA::UnsteadyList
```

Unsteady Reaction objects.

5.59.2.5 AdsorptionList

```
std::vector<AdsorptionReaction> SHARK_DATA::AdsorptionList
```

Equilibrium Adsorption Reaction Objects.

5.59.2.6 UnsteadyAdsList

```
std::vector<UnsteadyAdsorption> SHARK_DATA::UnsteadyAdsList
```

Unsteady Adsorption Reaction Objects.

5.59.2.7 MultiAdsList

```
std::vector<MultiligandAdsorption> SHARK_DATA::MultiAdsList
```

Multiligand Adsorptioin Objects.

5.59.2.8 ChemisorptionList

```
std::vector<ChemisorptionReaction> SHARK_DATA::ChemisorptionList
```

Chemisorption Reaction objects.

5.59.2.9 MultiChemList

```
std::vector<MultiligandChemisorption> SHARK_DATA::MultiChemList
```

Multiligand Chemisorption Reaction Objects.

5.59.2.10 OtherList

```
std::vector< double (*) (const Matrix<double> &x, SHARK_DATA *shark_dat, const void *data) >
SHARK_DATA::OtherList
```

Array of Other Residual functions to be defined by user.

This list of function pointers can be declared and set up by the user in order to add to or change the behavior of the SHARK system. Each one must be declared setup individually by the user. They will be called by the shark←
_residual function when needed. Alternatively, the user is free to provide their own shark_residual function for the overall system.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
shark_dat	pointer to the SHARK_DATA data structure
data	pointer to a user defined data structure that is used to evaluate this residual

5.59.2.11 numvar

```
int SHARK_DATA::numvar
```

Total number of functions and species.

5.59.2.12 num_ssr

```
int SHARK_DATA::num_ssr
```

Number of steady-state reactions.

5.59.2.13 num_mbe

```
int SHARK_DATA::num_mbe
```

Number of mass balance equations.

5.59.2.14 num_usr

```
int SHARK_DATA::num_usr = 0
```

Number of unsteady-state reactions.

5.59.2.15 num_ssao

```
int SHARK_DATA::num_ssao = 0
```

Number of steady-state adsorption objects.

5.59.2.16 num_usao

```
int SHARK_DATA::num_usao = 0
```

Number of unsteady adsorption objects.

5.59.2.17 num_multi_ssao

```
int SHARK_DATA::num_multi_ssao = 0
```

Number of multiligand steady-state adsorption objects.

5.59.2.18 num_sschem

```
int SHARK_DATA::num_sschem = 0
```

Number of steady-state chemisorption objects.

5.59.2.19 num_multi_sschem

```
int SHARK_DATA::num_multi_sschem = 0
```

Number of multiligand steady-state chemisorption objects.

5.59.2.20 num_ssar

```
std::vector<int> SHARK_DATA::num_ssar
```

List of the numbers of reactions in each adsorption object.

5.59.2.21 num_usar

```
std::vector<int> SHARK_DATA::num_usar
```

List of the numbers of reactions in each unsteady adsorption object.

5.59.2.22 num_sschem_rxns

```
std::vector<int> SHARK_DATA::num_sschem_rxns
```

List of the numbers of reactions in each steady-state chemisorption object.

5.59.2.23 num_multi_ssar

```
std::vector< std::vector<int> > SHARK_DATA::num_multi_ssar
```

List of all multiligand objects -> List of ligands and rxns of that ligand.

5.59.2.24 num_multichem_rxns

```
std::vector< std::vector<int> > SHARK_DATA::num_multichem_rxns
```

List of all multiligand chemisorption objects -> List of num rxns for that ligand.

5.59.2.25 ss_ads_names

```
std::vector<std::string> SHARK_DATA::ss_ads_names
```

List of the steady-state adsorbent object names.

5.59.2.26 us_ads_names

```
std::vector<std::string> SHARK_DATA::us_ads_names
```

List of the unsteady adsorption object names.

5.59.2.27 ss_chem_names

```
std::vector<std::string> SHARK_DATA::ss_chem_names
```

List of the steady-state chemisorption object names.

5.59.2.28 ssmulti_names

```
std::vector< std::vector<std::string> > SHARK_DATA::ssmulti_names
```

List of the names of the ligands in each multiligand object.

5.59.2.29 ssmultichem_names

```
std::vector< std::vector<std::string> > SHARK_DATA::ssmultichem_names
```

List of the names of the ligands in each multiligand chemisorption object.

5.59.2.30 num_other

```
int SHARK_DATA::num_other = 0
```

Number of other functions to be used (default is always 0)

5.59.2.31 act_fun

```
int SHARK_DATA::act_fun = IDEAL
```

Flag denoting the activity function to use (default is IDEAL)

5.59.2.32 reactor_type

```
int SHARK_DATA::reactor_type = BATCH
```

Flag denoting the type of reactor considered for the system (default is BATCH)

5.59.2.33 totalsteps

```
int SHARK_DATA::totalsteps = 0
```

Total number of iterations.

5.59.2.34 totalcalls

```
int SHARK_DATA::totalcalls = 0
```

Total number of residual function calls.

5.59.2.35 timesteps

```
int SHARK_DATA::timesteps = 0
```

Number of time steps taken to complete simulation.

5.59.2.36 pH_index

```
int SHARK_DATA::pH_index = -1
```

Contains the index of the pH variable (set internally)

5.59.2.37 pOH_index

```
int SHARK_DATA::pOH_index = -1
```

Contains the index of the pOH variable (set internally)

5.59.2.38 simulationtime

```
double SHARK_DATA::simulationtime = 0.0
```

Time to simulate unsteady reactions for (default = 0.0 hrs)

5.59.2.39 dt

```
double SHARK_DATA::dt = 0.1
```

Time step size (hrs)

5.59.2.40 dt_min

```
double SHARK_DATA::dt_min = sqrt(DBL_EPSILON)
```

Minimum allowable step size.

5.59.2.41 dt_max

```
double SHARK_DATA::dt_max = 744.0
```

Maximum allowable step size (∼1 month in time)

5.59.2.42 t_out

```
double SHARK_DATA::t_out = 0.0
```

Time increment by which file output is made (default = print all time steps)

5.59.2.43 t_count

```
double SHARK_DATA::t_count = 0.0
```

Running count of time increments.

5.59.2.44 time

```
double SHARK_DATA::time = 0.0
```

Current value of time (starts from t = 0.0 hrs)

5.59.2.45 time_old

```
double SHARK_DATA::time_old = 0.0
```

Previous value of time (start from t = 0.0 hrs)

5.59.2.46 pH

```
double SHARK_DATA::pH = 7.0
```

Value of pH if needed (default = 7)

5.59.2.47 pH_step

```
double SHARK_DATA::pH_step = 0.5
```

Value by which to increment pH when doing a speciation curve (default = 0.5)

5.59.2.48 pH_end

```
double SHARK_DATA::pH_end = 14.0
```

Value at which we stop doing the speciation curve as a function of pH (default = 14.0)

5.59.2.49 start_temp

```
double SHARK_DATA::start_temp = 277.15
```

Value of the starting temperature used for Temperature Curves (default = 277.15 K)

5.59.2.50 end_temp

```
double SHARK_DATA::end_temp = 323.15
```

Value of the ending temperature used for Temperature Curves (default = 323.15 K)

5.59.2.51 temp_step

```
double SHARK_DATA::temp_step = 10.0
```

Size of the step changes to use for Temperature Curves (default = 10.0 K);.

5.59.2.52 volume

```
double SHARK_DATA::volume = 1.0
```

Volume of the domain in liters (default = 1 L)

5.59.2.53 flow_rate

```
double SHARK_DATA::flow_rate = 1.0
```

Flow rate in the reactor in L/hr (default = 1 L/hr)

5.59.2.54 xsec_area

```
double SHARK_DATA::xsec_area = 1.0
```

Cross sectional area of the reactor in m^2 (default = 1 m^2)

5.59.2.55 Norm

```
double SHARK_DATA::Norm = 0.0
```

Current value of euclidean norm in solution.

5.59.2.56 dielectric_const

```
double SHARK_DATA::dielectric_const = 78.325
```

Dielectric constant used in many activity models (default: water = 78.325 (1/K))

5.59.2.57 relative_permittivity

```
double SHARK_DATA::relative_permittivity = 80.1
```

Relative permittivity of the medium (default: water = 80.1 (-))

5.59.2.58 temperature

```
double SHARK_DATA::temperature = 298.15
```

Solution temperature (default = 25 oC or 298.15 K)

5.59.2.59 ionic_strength

```
double SHARK_DATA::ionic_strength = 0.0
```

Solution ionic strength in Molar (calculated internally)

5.59.2.60 steadystate

```
bool SHARK_DATA::steadystate = true
```

True = solve steady problem; False = solve transient problem.

5.59.2.61 ZeroInitialSolids

```
bool SHARK_DATA::ZeroInitialSolids = false
```

True = no solids or adsorption initially in the reactor.

5.59.2.62 TimeAdaptivity

```
bool SHARK_DATA::TimeAdaptivity = false
```

True = solve using variable time step.

5.59.2.63 const_pH

```
bool SHARK_DATA::const_pH = false
```

True = set pH to a constant; False = solve for pH.

5.59.2.64 SpeciationCurve

```
bool SHARK_DATA::SpeciationCurve = false
```

True = runs a series of constant pH steady-state problems to produce curves.

5.59.2.65 TemperatureCurve

```
bool SHARK_DATA::TemperatureCurve = false
```

True = runs a series of constant temperature steady-state problmes to produce curves.

5.59.2.66 Console_Output

```
bool SHARK_DATA::Console_Output = true
```

True = display output to console.

5.59.2.67 File_Output

```
bool SHARK_DATA::File_Output = false
```

True = write output to a file.

5.59.2.68 Contains_pH

```
bool SHARK_DATA::Contains_pH = false
```

True = system contains pH as a variable (set internally)

5.59.2.69 Contains_pOH

```
bool SHARK_DATA::Contains_pOH = false
```

True = system contains pOH as a variable (set internally)

5.59.2.70 Converged

```
bool SHARK_DATA::Converged = false
```

True = system converged within tolerance.

5.59.2.71 LocalMin

```
bool SHARK_DATA::LocalMin = true
```

True = allow the system to settle for a local minimum if tolerance not reached.

5.59.2.72 X_old

```
Matrix<double> SHARK_DATA::X_old
```

Solution vector for old time step - log(C)

5.59.2.73 X_new

```
Matrix<double> SHARK_DATA::X_new
```

Solution vector for current time step - log(C)

5.59.2.74 Conc_old

```
Matrix<double> SHARK_DATA::Conc_old
```

Concentration vector for old time step - 10^x.

5.59.2.75 Conc_new

```
Matrix<double> SHARK_DATA::Conc_new
```

Concentration vector for current time step - 10^x.

5.59.2.76 activity_new

```
Matrix<double> SHARK_DATA::activity_new
```

Activity matrix for current time step.

5.59.2.77 activity_old

```
Matrix<double> SHARK_DATA::activity_old
```

Activity matrix from prior time step.

5.59.2.78 EvalActivity

```
int(* SHARK_DATA::EvalActivity) (const Matrix< double > &x, Matrix< double > &F, const void
*data)
```

Function pointer to evaluate activity coefficients.

This function pointer is called within the shark_residual function to calculate and modify the activity_new matrix entries. When using the SHARK default options, this function pointer will be automatically set to a cooresponding activity function for the list of valid functions from the valid_act enum. User may override this function pointer if they desire. Must be overriden after calling the setup function.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

5.59.2.79 Residual

```
int(* SHARK_DATA::Residual) (const Matrix< double > &x, Matrix< double > &F, const void *data)
```

Function pointer to evaluate all residuals in the system.

This function will be fed into the PJFNK solver (see lark.h) to solve the non-linear system of equations. By default, this pointer will be the shark_residual function (see below). However, the user may override the function and provide their own residuals for the PJFNK solver to operate on.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of residuals that are to be altered from the functions in the system
data	pointer to a data structure needed to evaluate the activity model

5.59.2.80 lin_precon

```
int(* SHARK_DATA::lin_precon) (const Matrix< double > &r, Matrix< double > &p, const void
*data)
```

Function pointer to form a linear preconditioning operation for the Jacobian.

This function will be fed into the linear solver used for each non-linear step in PJFNK (see lark.h). By default, we cannot provide any linear preconditioner, because we do not know the form or sparcity of the Jacobian before hand. It will be the user's responsibility to form their own preconditioner until we can figure out a generic way to precondition the system.

5.59.2.81 Newton_data

```
PJFNK_DATA SHARK_DATA::Newton_data
```

Data structure for the Newton-Krylov solver (see lark.h)

5.59.2.82 activity_data

```
const void* SHARK_DATA::activity_data
```

User defined data structure for an activity model.

5.59.2.83 residual_data

```
const void* SHARK_DATA::residual_data
```

User defined data structure for the residual function.

5.59.2.84 precon_data

```
const void* SHARK_DATA::precon_data
```

User defined data structure for preconditioning.

5.59.2.85 other_data

```
const void* SHARK_DATA::other_data
```

User define data structure used for user defined residuals.

5.59.2.86 OutputFile

```
FILE* SHARK_DATA::OutputFile
```

Output File pointer.

5.59.2.87 yaml_object

`yaml_cpp_class SHARK_DATA::yaml_object`

yaml object to read and access digitized yaml documents (see yaml_wrapper.h)

The documentation for this struct was generated from the following file:

- shark.h

5.60 SKUA_DATA Struct Reference

Data structure for all simulation information in SKUA.

`#include <skua.h>`

Public Attributes

- unsigned long int total_steps

  Running total of all calculation steps.
- int coord

  Used to determine the coordinates of the problem.
- double sim_time

  Stopping time for the simulation (hrs)
- double t_old

  Old time of the simulations (hrs)
- double t

  Current time of the simulations (hrs)
- double t_counter = 0.0

  Counts for print times for output (hrs)
- double t_print

  Prints out every t_print time (hrs)
- double qTn

  Old total amounts adsorbed (mol/kg)
- double qTnp1

  New total amounts adsorbed (mol/kg)
- bool Print2File = true

  True = results to .txt; False = no printing.
- bool Print2Console = true

  True = results to console; False = no printing.
- double gas_velocity

  Superficial Gas Velocity arount pellet (cm/s)
- double pellet_radius

  Nominal radius of the pellet/crystal (um)
- double char_measure

  Length or Area if in Cylindrical or Cartesian coordinates (um or um^2)
- bool DirichletBC = true

  True = Dirichlet BC; False = Neumann BC.
- bool NonLinear = true

  True = Non-linear solver; False = Linear solver.

- std::vector< double > y

    Outside mole fractions of each component (-)
- FILE ∗ OutputFile

    Output file pointer to the output file.
- double(∗ eval_diff)(int i, int l, const void ∗user_data)

    Function pointer for evaluating surface diffusivity.
- double(∗ eval_kf)(int i, const void ∗user_data)

    Function pointer for evaluating film mass transfer.
- const void ∗ user_data

    Data structure for user's information needed in parameter functions.
- MAGPIE_DATA magpie_dat

    Data structure for adsorption equilibria (see magpie.h)
- MIXED_GAS ∗ gas_dat

    Pointer to the MIXED_GAS data structure (see egret.h)
- std::vector< FINCH_DATA > finch_dat

    Data structure for adsorption kinetics (see finch.h)
- std::vector< SKUA_PARAM > param_dat

    Data structure for SKUA specific parameters.

5.60.1 Detailed Description

Data structure for all simulation information in SKUA.

C-style object holding all data, functions, and other objects needed to successfully run a SKUA simulation. This object holds system information, such as boundary condition type, adsorbent size, and total adsorption, and also contains structure for EGRET (egret.h), FINCH (finch.h), and MAGPIE (magpie.h) calculations. Function pointers for evaluation of the surface diffusivity and film mass transfer coefficients can be overriden by the user to change the behavior of the SKUA simulation. However, defaults are also provided for these functions.

5.60.2 Member Data Documentation

5.60.2.1 total_steps

```
unsigned long int SKUA_DATA::total_steps
```

Running total of all calculation steps.

5.60.2.2 coord

```
int SKUA_DATA::coord
```

Used to determine the coordinates of the problem.

5.60.2.3 sim_time

```
double SKUA_DATA::sim_time
```

Stopping time for the simulation (hrs)

5.60.2.4 t_old

```
double SKUA_DATA::t_old
```

Old time of the simulations (hrs)

5.60.2.5 t

```
double SKUA_DATA::t
```

Current time of the simulations (hrs)

5.60.2.6 t_counter

```
double SKUA_DATA::t_counter = 0.0
```

Counts for print times for output (hrs)

5.60.2.7 t_print

```
double SKUA_DATA::t_print
```

Prints out every t_print time (hrs)

5.60.2.8 qTn

```
double SKUA_DATA::qTn
```

Old total amounts adsorbed (mol/kg)

5.60.2.9 qTnp1

```
double SKUA_DATA::qTnp1
```

New total amounts adsorbed (mol/kg)

5.60.2.10 Print2File

```
bool SKUA_DATA::Print2File = true
```

True = results to .txt; False = no printing.

5.60.2.11 Print2Console

```
bool SKUA_DATA::Print2Console = true
```

True = results to console; False = no printing.

5.60.2.12 gas_velocity

```
double SKUA_DATA::gas_velocity
```

Superficial Gas Velocity arount pellet (cm/s)

5.60.2.13 pellet_radius

```
double SKUA_DATA::pellet_radius
```

Nominal radius of the pellet/crystal (um)

5.60.2.14 char_measure

```
double SKUA_DATA::char_measure
```

Length or Area if in Cylindrical or Cartesian coordinates (um or um^2)

5.60.2.15 DirichletBC

```
bool SKUA_DATA::DirichletBC = true
```

True = Dirichlet BC; False = Neumann BC.

5.60.2.16 NonLinear

```
bool SKUA_DATA::NonLinear = true
```

True = Non-linear solver; False = Linear solver.

5.60.2.17 y

```
std::vector<double> SKUA_DATA::y
```

Outside mole fractions of each component (-)

5.60.2.18 OutputFile

```
FILE* SKUA_DATA::OutputFile
```

Output file pointer to the output file.

5.60.2.19 eval_diff

```
double(* SKUA_DATA::eval_diff) (int i, int l, const void *user_data)
```

Function pointer for evaluating surface diffusivity.

5.60.2.20 eval_kf

```
double(* SKUA_DATA::eval_kf) (int i, const void *user_data)
```

Function pointer for evaluating film mass transfer.

5.60.2.21 user_data

```
const void* SKUA_DATA::user_data
```

Data structure for user's information needed in parameter functions.

5.60.2.22 magpie_dat

```
MAGPIE_DATA SKUA_DATA::magpie_dat
```

Data structure for adsorption equilibria (see magpie.h)

5.60.2.23 gas_dat

```
MIXED_GAS* SKUA_DATA::gas_dat
```

Pointer to the MIXED_GAS data structure (see egret.h)

5.60.2.24 finch_dat

```
std::vector<FINCH_DATA> SKUA_DATA::finch_dat
```

Data structure for adsorption kinetics (see finch.h)

5.60.2.25 param_dat

```
std::vector<SKUA_PARAM> SKUA_DATA::param_dat
```

Data structure for SKUA specific parameters.

The documentation for this struct was generated from the following file:

- skua.h

5.61 SKUA_OPT_DATA Struct Reference

Data structure for the SKUA Optimization Routine.

```
#include <skua_opt.h>
```

Public Attributes

- int num_curves

  Number of adsorption curves to analyze.
- int evaluation

  Number of times the eval function has been called for a single curve.
- unsigned long int total_eval

  Total number of evaluations needed for completion.
- int current_points

  Number of points in the current curve.
- int num_params = 1

  Number of adjustable parameters for the current curve.
- int diffusion_type

  Flag to identify type of diffusion function to use.
- int adsorb_index

  Component index for adsorbable species.
- int max_guess_iter = 20

  Maximum allowed guess iterations (default = 20)
- bool Optimize

  True = run optimization, False = run a comparison.
- bool Rough

  True = use only a rough estimate, False = run full optimization.
- double current_temp

  Temperature for current curve.
- double current_press

  Partial pressure for current curve.

- double current_equil

    Equilibrium data point for the current curve.
- double simulation_equil

    Equilibrium simulation point for the current curve.
- double max_bias

    Positive maximum bias plausible for fitting.
- double min_bias

    Negative minimum bias plausible for fitting.
- double e_norm

    Euclidean norm of current fit.
- double f_bias

    Function bias of current fit.
- double e_norm_old

    Euclidean norm of the previous fit.
- double f_bias_old

    Function bias of the previous fit.
- double param_guess

    Parameter guess for the surface/crystal diffusivity.
- double param_guess_old

    Parameter guess for the previous curve.
- double rel_tol_norm = 0.1

    Tolerance for convergence of the guess norm.
- double abs_tol_bias = 0.1

    Tolerance for convergence of the guess bias.
- std::vector< double > y_base

    Gas phase mole fractions in absense of adsorbing species.
- std::vector< double > q_data

    Amount adsorbed at a particular point in current curve.
- std::vector< double > q_sim

    Amount adsorbed based on the simulation.
- std::vector< double > t

    Time points in the current curve.
- FILE ∗ ParamFile

    Output file for parameter results.
- FILE ∗ CompareFile

    Output file for comparison of results.
- SKUA_DATA skua_dat

    Data structure for the SKUA simulation.

5.61.1 Detailed Description

Data structure for the SKUA Optimization Routine.

C-style object holding data and pointers necessary for running a SKUA optimization. It contains information about the type of optimization requested, the current status of the optimization, the data being compared against, and the SKUA_DATA object for the evaluation of a SKUA simulation. The pointers in the structure are for the two output files produced by the routine: (i) parameter results and (ii) model comparison results.

5.61.2 Member Data Documentation

5.61.2.1 num_curves

```
int SKUA_OPT_DATA::num_curves
```

Number of adsorption curves to analyze.

5.61.2.2 evaluation

```
int SKUA_OPT_DATA::evaluation
```

Number of times the eval function has been called for a single curve.

5.61.2.3 total_eval

```
unsigned long int SKUA_OPT_DATA::total_eval
```

Total number of evaluations needed for completion.

5.61.2.4 current_points

```
int SKUA_OPT_DATA::current_points
```

Number of points in the current curve.

5.61.2.5 num_params

```
int SKUA_OPT_DATA::num_params = 1
```

Number of adjustable parameters for the current curve.

5.61.2.6 diffusion_type

```
int SKUA_OPT_DATA::diffusion_type
```

Flag to identify type of diffusion function to use.

5.61.2.7 adsorb_index

```
int SKUA_OPT_DATA::adsorb_index
```

Component index for adsorbable species.

5.61.2.8 max_guess_iter

```
int SKUA_OPT_DATA::max_guess_iter = 20
```

Maximum allowed guess iterations (default = 20)

5.61.2.9 Optimize

```
bool SKUA_OPT_DATA::Optimize
```

True = run optimization, False = run a comparison.

5.61.2.10 Rough

```
bool SKUA_OPT_DATA::Rough
```

True = use only a rough estimate, False = run full optimization.

5.61.2.11 current_temp

```
double SKUA_OPT_DATA::current_temp
```

Temperature for current curve.

5.61.2.12 current_press

```
double SKUA_OPT_DATA::current_press
```

Partial pressure for current curve.

5.61.2.13 current_equil

```
double SKUA_OPT_DATA::current_equil
```

Equilibrium data point for the current curve.

5.61.2.14 simulation_equil

```
double SKUA_OPT_DATA::simulation_equil
```

Equilibrium simulation point for the current curve.

5.61.2.15 max_bias

```
double SKUA_OPT_DATA::max_bias
```

Positive maximum bias plausible for fitting.

5.61.2.16 min_bias

```
double SKUA_OPT_DATA::min_bias
```

Negative minimum bias plausible for fitting.

5.61.2.17 e_norm

```
double SKUA_OPT_DATA::e_norm
```

Euclidean norm of current fit.

5.61.2.18 f_bias

```
double SKUA_OPT_DATA::f_bias
```

Function bias of current fit.

5.61.2.19 e_norm_old

```
double SKUA_OPT_DATA::e_norm_old
```

Euclidean norm of the previous fit.

5.61.2.20 f_bias_old

```
double SKUA_OPT_DATA::f_bias_old
```

Function bias of the previous fit.

5.61.2.21 param_guess

```
double SKUA_OPT_DATA::param_guess
```

Parameter guess for the surface/crystal diffusivity.

5.61.2.22 param_guess_old

```
double SKUA_OPT_DATA::param_guess_old
```

Parameter guess for the previous curve.

5.61.2.23 rel_tol_norm

```
double SKUA_OPT_DATA::rel_tol_norm = 0.1
```

Tolerance for convergence of the guess norm.

5.61.2.24 abs_tol_bias

```
double SKUA_OPT_DATA::abs_tol_bias = 0.1
```

Tolerance for convergence of the guess bias.

5.61.2.25 y_base

```
std::vector<double> SKUA_OPT_DATA::y_base
```

Gas phase mole fractions in absense of adsorbing species.

5.61.2.26 q_data

```
std::vector<double> SKUA_OPT_DATA::q_data
```

Amount adsorbed at a particular point in current curve.

5.61.2.27 q_sim

```
std::vector<double> SKUA_OPT_DATA::q_sim
```

Amount adsorbed based on the simulation.

5.61.2.28 t

```
std::vector<double> SKUA_OPT_DATA::t
```

Time points in the current curve.

5.61.2.29 ParamFile

```
FILE* SKUA_OPT_DATA::ParamFile
```

Output file for parameter results.

5.61.2.30 CompareFile

```
FILE* SKUA_OPT_DATA::CompareFile
```

Output file for comparison of results.

5.61.2.31 skua_dat

```
SKUA_DATA SKUA_OPT_DATA::skua_dat
```

Data structure for the SKUA simulation.

The documentation for this struct was generated from the following file:

- skua_opt.h

5.62 SKUA_PARAM Struct Reference

Data structure for species' parameters in SKUA.

```
#include <skua.h>
```

Public Attributes

- double activation_energy
- double ref_diffusion
- double ref_temperature
- double affinity
- double ref_pressure
- double film_transfer
- double xIC
- double y_eff
- double Qstn
- double Qstnp1
- double xn
- double xnp1
- bool Adsorbable
- std::string speciesName

5.62.1 Detailed Description

Data structure for species' parameters in SKUA.

C-style object holding data and parameters associated with the gas/solid species in the overall SKUA system. These parameters are used in to modify surface diffusivity with temperature, establish film mass transfer coefficients, formulate the initial conditions, and store solution results for heat of adsorption and adsorbed mole fractions. One of these objects will be created for each species in the gas system.

5.62.2 Member Data Documentation

5.62.2.1 activation_energy

```
double SKUA_PARAM::activation_energy
```

5.62.2.2 ref_diffusion

```
double SKUA_PARAM::ref_diffusion
```

5.62.2.3 ref_temperature

```
double SKUA_PARAM::ref_temperature
```

5.62.2.4 affinity

```
double SKUA_PARAM::affinity
```

5.62.2.5 ref_pressure

```
double SKUA_PARAM::ref_pressure
```

5.62.2.6 film_transfer

```
double SKUA_PARAM::film_transfer
```

5.62.2.7 xIC

```
double SKUA_PARAM::xIC
```

5.62.2.8 y_eff

```
double SKUA_PARAM::y_eff
```

5.62.2.9 Qstn

```
double SKUA_PARAM::Qstn
```

5.62.2.10 Qstnp1

```
double SKUA_PARAM::Qstnp1
```

5.62.2.11 xn

```
double SKUA_PARAM::xn
```

5.62.2.12 xnp1

```
double SKUA_PARAM::xnp1
```

5.62.2.13 Adsorbable

```
bool SKUA_PARAM::Adsorbable
```

5.62.2.14 speciesName

```
std::string SKUA_PARAM::speciesName
```

The documentation for this struct was generated from the following file:

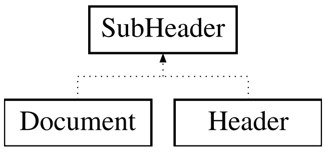
- skua.h

5.63 SubHeader Class Reference

Object for the Lowest level of Header for the yaml_wrapper.

```
#include <yaml_wrapper.h>
```

Inheritance diagram for SubHeader:



Public Member Functions

- SubHeader ()

    Default Constructor.
- ∼SubHeader ()

    Default Destructor.
- SubHeader (const SubHeader &subheader)

    Copy constructor.
- SubHeader (const KeyValueMap &map)

    Construction by existing map.
- SubHeader (std::string name)

    Construction by name only.
- SubHeader (std::string name, const KeyValueMap &map)

    Construction by name and map.
- SubHeader & operator= (const SubHeader &sub)

    Equals overload.
- ValueTypePair & operator[ ] (const std::string key)

    Return the ValueType reference at the given key.
- ValueTypePair operator[ ] (const std::string key) const

    Return the ValueType at the give key.
- KeyValueMap & getMap ()

    Returns reference to the KeyValueMap object.
- void clear ()

    Empty out data contents.
- void addPair (std::string key, std::string val)

    Adds a pair object to the map (with only strings)
- void addPair (std::string key, std::string val, int type)

    Adds a pair object and asserts a type.
- void setName (std::string name)

    Sets the name of the subheader.
- void setAlias (std::string alias)

    Set the alias without type specification.
- void setAlias (std::string alias, int state)

    Sets the alias and state of the subheader.
- void setNameAliasPair (std::string name, std::string alias, int state)

    Sets the name and alias of the subheader.
- void setState (int state)

Sets the state of the subheader.
- void DisplayContents ()

  Display the contents of the subheader.
- std::string getName ()

  Return the name of the subheader.
- std::string getAlias ()

  Return the alias of the subheader, if one exists.
- bool isAlias ()

  Returns true if subheader is an alias.
- bool isAnchor ()

  Returns true if subheader is an anchor.
- int getState ()

  Returns the state of the subheader.

Protected Attributes

- KeyValueMap Data_Map

  A Map of Keys and Values.
- std::string name

  Name of the subheader.
- std::string alias

  Name of the alias for the subheader.
- int state

  State of the header.

5.63.1 Detailed Description

Object for the Lowest level of Header for the yaml_wrapper.

C++ Object for sub-headers in a yaml document. This object contains a KeyValueMap that holds a set of key-value pairs for data listed under a sub-header in yaml files. It is the lowest allowable recursion of headers in a yaml document and so is the base class for Header and Document, which themselves can contain KeyValueMaps as well as maps for other header-like objects.

SubHeaders are recognized by a unique name and/or alias while being put together in other higher document structures. Additionally, each header or sub-header will have a state to denote whether the object is a yaml alias, anchor, or niether. This is used in the yaml documents to ensure that aliases for anchors have the correct data moved over into the new structures.

5.63.2 Constructor & Destructor Documentation

5.63.2.1 SubHeader() [1/5]

```
SubHeader::SubHeader ( )
```

Default Constructor.

5.63.2.2 ∼SubHeader()

```
SubHeader::∼SubHeader ( )
```

Default Destructor.

5.63.2.3 SubHeader() [2/5]

```
SubHeader::SubHeader (
            const SubHeader & subheader )
```

Copy constructor.

5.63.2.4 SubHeader() [3/5]

```
SubHeader::SubHeader (
            const KeyValueMap & map )
```

Construction by existing map.

5.63.2.5 SubHeader() [4/5]

```
SubHeader::SubHeader (
            std::string name )
```

Construction by name only.

5.63.2.6 SubHeader() [5/5]

```
SubHeader::SubHeader (
            std::string name,
            const KeyValueMap & map )
```

Construction by name and map.

5.63.3 Member Function Documentation

5.63.3.1 operator=()

```
SubHeader& SubHeader::operator= (
            const SubHeader & sub )
```

Equals overload.

5.63.3.2 operator[]() [1/2]

```
ValueTypePair& SubHeader::operator[] (
            const std::string key )
```

Return the ValueType reference at the given key.

5.63.3.3 operator[]() [2/2]

```
ValueTypePair SubHeader::operator[] (
            const std::string key ) const
```

Return the ValueType at the give key.

5.63.3.4 getMap()

```
KeyValueMap& SubHeader::getMap ( )
```

Returns reference to the KeyValueMap object.

5.63.3.5 clear()

```
void SubHeader::clear ( )
```

Empty out data contents.

5.63.3.6 addPair() [1/2]

```
void SubHeader::addPair (
            std::string key,
            std::string val )
```

Adds a pair object to the map (with only strings)

5.63.3.7 addPair() [2/2]

```
void SubHeader::addPair (
            std::string key,
            std::string val,
            int type )
```

Adds a pair object and asserts a type.

5.63.3.8 setName()

```
void SubHeader::setName (
             std::string name )
```

Sets the name of the subheader.

5.63.3.9 setAlias() [1/2]

```
void SubHeader::setAlias (
             std::string alias )
```

Set the alias without type specification.

5.63.3.10 setAlias() [2/2]

```
void SubHeader::setAlias (
             std::string alias,
             int state )
```

Sets the alias and state of the subheader.

5.63.3.11 setNameAliasPair()

```
void SubHeader::setNameAliasPair (
             std::string name,
             std::string alias,
             int state )
```

Sets the name and alias of the subheader.

5.63.3.12 setState()

```
void SubHeader::setState (
             int state )
```

Sets the state of the subheader.

5.63.3.13 DisplayContents()

```
void SubHeader::DisplayContents ( )
```

Display the contents of the subheader.

5.63.3.14 getName()

```
std::string SubHeader::getName ( )
```

Return the name of the subheader.

5.63.3.15 getAlias()

```
std::string SubHeader::getAlias ( )
```

Return the alias of the subheader, if one exists.

5.63.3.16 isAlias()

```
bool SubHeader::isAlias ( )
```

Returns true if subheader is an alias.

5.63.3.17 isAnchor()

```
bool SubHeader::isAnchor ( )
```

Returns true if subheader is an anchor.

5.63.3.18 getState()

```
int SubHeader::getState ( )
```

Returns the state of the subheader.

5.63.4 Member Data Documentation

5.63.4.1 Data_Map

```
KeyValueMap SubHeader::Data_Map [protected]
```

A Map of Keys and Values.

5.63.4.2 name

```
std::string SubHeader::name [protected]
```

Name of the subheader.

5.63.4.3 alias

```
std::string SubHeader::alias [protected]
```

Name of the alias for the subheader.

5.63.4.4 state

```
int SubHeader::state [protected]
```

State of the header.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.64 SurfaceElement Class Reference

SurfaceElement Object.

```
#include <mesh.h>
```

Public Member Functions

- SurfaceElement ()

  Default Constructor.
- ∼SurfaceElement ()

  Default Destructor.
- void DisplayInfo ()

  Print out information to the console.
- void AssignNodes (Node &n1, Node &n2, Node &n3)

  Assign nodes for the surface element.
- void AssignIDnumber (unsigned int i)

  Assign the id number for the surface element.
- void calculateArea ()

  Calculate and store area value.
- void findCentroid ()

  Find and set the centroid node.
- void determineType ()

  Determine the type of surface element.
- void evaluateProperties ()

  Calls functions for area and centroid.

Private Attributes

- Node ∗ node1

  Pointer to first node.
- Node ∗ node2

  Pointer to second node.
- Node ∗ node3

  Pointer to third node.
- double area

  Area of the surface element.
- Node centroid

  Centroid node for the surface element.
- unsigned int IDnum

  Identification number for the surface element.
- element_type SubType

  Type of surface element (BOUDNARY or INTERIOR)

5.64.1 Detailed Description

SurfaceElement Object.

This class structure creates a C++ object for a surface element. The surface is made up of a reference to three distinct nodes. Based on those nodes, we can formulate a set of line segments that encapsulates the surface element. The surface element will also have an identification number, an area, and a centroid (similar to the midpoint of a line segment).

5.64.2 Constructor & Destructor Documentation

5.64.2.1 SurfaceElement()

```
SurfaceElement::SurfaceElement ( )
```

Default Constructor.

5.64.2.2 ∼SurfaceElement()

```
SurfaceElement::∼SurfaceElement ( )
```

Default Destructor.

5.64.3 Member Function Documentation

5.64.3.1 DisplayInfo()

```
void SurfaceElement::DisplayInfo ( )
```

Print out information to the console.

5.64.3.2 AssignNodes()

```
void SurfaceElement::AssignNodes (
             Node & n1,
             Node & n2,
             Node & n3 )
```

Assign nodes for the surface element.

5.64.3.3 AssignIDnumber()

```
void SurfaceElement::AssignIDnumber (
             unsigned int i )
```

Assign the id number for the surface element.

5.64.3.4 calculateArea()

```
void SurfaceElement::calculateArea ( )
```

Calculate and store area value.

5.64.3.5 findCentroid()

```
void SurfaceElement::findCentroid ( )
```

Find and set the centroid node.

5.64.3.6 determineType()

```
void SurfaceElement::determineType ( )
```

Determine the type of surface element.

5.64.3.7 evaluateProperties()

```
void SurfaceElement::evaluateProperties ( )
```

Calls functions for area and centroid.

5.64.4 Member Data Documentation

5.64.4.1 node1

```
Node* SurfaceElement::node1  [private]
```

Pointer to first node.

5.64.4.2 node2

```
Node* SurfaceElement::node2  [private]
```

Pointer to second node.

5.64.4.3 node3

```
Node* SurfaceElement::node3  [private]
```

Pointer to third node.

5.64.4.4 area

```
double SurfaceElement::area  [private]
```

Area of the surface element.

5.64.4.5 centroid

```
Node SurfaceElement::centroid  [private]
```

Centroid node for the surface element.

5.64.4.6 IDnum

```
unsigned int SurfaceElement::IDnum  [private]
```

Identification number for the surface element.

5.64.4.7 SubType

```
element_type SurfaceElement::SubType  [private]
```

Type of surface element (BOUDNARY or INTERIOR)

The documentation for this class was generated from the following file:

- mesh.h

5.65 SYSTEM_DATA Struct Reference

System Data Structure.

```
#include <magpie.h>
```

Public Attributes

- double T
  System Temperature (K)
- double PT
  Total Pressure (kPa)
- double qT
  Total Amount adsorbed (mol/kg)
- double PI
  Total Lumped Spreading Pressure (mol/kg)
- double pi
  Actual Spreading pressure (J/m^2)
- double As
  Specific surface area of adsorbent (m^2/kg)
- int N
  Total Number of Components.
- int I
- int J
- int K
  Special indices used to keep track of sub-systems.
- unsigned long int total_eval
  Counter to keep track of total number of non-linear steps.
- double avg_norm
  Used to store all norms from evaluations then average at end of run.
- double max_norm
  Used to store the maximum e.norm calculated from non-linear iterations.

- int Sys

    Number of sub-systems to solve.
- int Par

    Number of binary parameters to solve for.
- bool Recover

    If Recover == false, standard GPAST using y's as knowns.
- bool Carrier

    If there is an inert carrier gas, Carrier == true.
- bool Ideal

    If the behavior of the system is determined to be ideal, then Ideal == true.
- bool Output

    Boolean to suppress output if desired (true = display, false = no display.

5.65.1 Detailed Description

System Data Structure.

C-style object holding all the data associated with the overall system to be modeled.

5.65.2 Member Data Documentation

5.65.2.1 T

```
double SYSTEM_DATA::T
```

System Temperature (K)

5.65.2.2 PT

```
double SYSTEM_DATA::PT
```

Total Pressure (kPa)

5.65.2.3 qT

```
double SYSTEM_DATA::qT
```

Total Amount adsorbed (mol/kg)

5.65.2.4 PI

```
double SYSTEM_DATA::PI
```

Total Lumped Spreading Pressure (mol/kg)

5.65.2.5 pi

```
double SYSTEM_DATA::pi
```

Actual Spreading pressure (J/m^2)

5.65.2.6 As

```
double SYSTEM_DATA::As
```

Specific surface area of adsorbent (m^2/kg)

5.65.2.7 N

```
int SYSTEM_DATA::N
```

Total Number of Components.

5.65.2.8 I

```
int SYSTEM_DATA::I
```

5.65.2.9 J

```
int SYSTEM_DATA::J
```

5.65.2.10 K

```
int SYSTEM_DATA::K
```

Special indices used to keep track of sub-systems.

5.65.2.11 total_eval

```
unsigned long int SYSTEM_DATA::total_eval
```

Counter to keep track of total number of non-linear steps.

5.65.2.12 avg_norm

```
double SYSTEM_DATA::avg_norm
```

Used to store all norms from evaluations then average at end of run.

5.65.2.13 max_norm

```
double SYSTEM_DATA::max_norm
```

Used to store the maximum e.norm calculated from non-linear iterations.

5.65.2.14 Sys

```
int SYSTEM_DATA::Sys
```

Number of sub-systems to solve.

5.65.2.15 Par

```
int SYSTEM_DATA::Par
```

Number of binary parameters to solve for.

5.65.2.16 Recover

```
bool SYSTEM_DATA::Recover
```

If Recover == false, standard GPAST using y's as knowns.

5.65.2.17 Carrier

```
bool SYSTEM_DATA::Carrier
```

If there is an inert carrier gas, Carrier == true.

5.65.2.18 Ideal

```
bool SYSTEM_DATA::Ideal
```

If the behavior of the system is determined to be ideal, then Ideal == true.

5.65.2.19 Output

```
bool SYSTEM_DATA::Output
```

Boolean to suppress output if desired (true = display, false = no display.

The documentation for this struct was generated from the following file:

- magpie.h

5.66 TRAJECTORY_DATA Struct Reference

```
#include <Trajectory.h>
```

Public Attributes

- double mu_0 = 12.57e-7

  permeability of free space, H/m
- double rho_f = 1000.0

  Fluid density, Kg/m3.
- double eta = 0.001
- double Hamaker = 1.3e-21
- double Temp = 298
- double k = 1.38e-23
- double Rs
- double L
- double porosity
- double V_separator
- double a
- double V_wire
- double L_wire
- double A_separator
- double A_wire
- double B0
- double H0
- double Ms = 0.6
- double b
- double chi_p
- double rho_p = 8000.0
- double Q_in
- double V0
- double Y_initial = 20.0
- double dt
- double M

- double mp
- double beta
- double q_bar
- double sigma_v
- double sigma_vz
- double sigma_z
- double sigma_n
- double sigma_m
- double n_rand
- double m_rand
- double s_rand
- double t_rand
- Matrix< double > POL
- Matrix< double > H
- Matrix< double > dX
- Matrix< double > dY
- Matrix< double > Vr
- Matrix< double > Vt
- Matrix< double > X
- Matrix< double > Y
- Matrix< int > Cap

5.66.1 Member Data Documentation

5.66.1.1 mu_0

```
double TRAJECTORY_DATA::mu_0 = 12.57e-7
```

permeability of free space, H/m

5.66.1.2 rho_f

```
double TRAJECTORY_DATA::rho_f = 1000.0
```

Fluid density, Kg/m3.

5.66.1.3 eta

```
double TRAJECTORY_DATA::eta = 0.001
```

5.66.1.4 Hamaker

```
double TRAJECTORY_DATA::Hamaker = 1.3e-21
```

5.66.1.5 Temp

```
double TRAJECTORY_DATA::Temp = 298
```

5.66.1.6 k

```
double TRAJECTORY_DATA::k = 1.38e-23
```

5.66.1.7 Rs

```
double TRAJECTORY_DATA::Rs
```

5.66.1.8 L

```
double TRAJECTORY_DATA::L
```

5.66.1.9 porosity

```
double TRAJECTORY_DATA::porosity
```

5.66.1.10 V_separator

```
double TRAJECTORY_DATA::V_separator
```

5.66.1.11 a

```
double TRAJECTORY_DATA::a
```

5.66.1.12 V_wire

```
double TRAJECTORY_DATA::V_wire
```

5.66.1.13 L_wire

```
double TRAJECTORY_DATA::L_wire
```

5.66.1.14 A_separator

```
double TRAJECTORY_DATA::A_separator
```

5.66.1.15 A_wire

```
double TRAJECTORY_DATA::A_wire
```

5.66.1.16 B0

```
double TRAJECTORY_DATA::B0
```

5.66.1.17 H0

```
double TRAJECTORY_DATA::H0
```

5.66.1.18 Ms

```
double TRAJECTORY_DATA::Ms = 0.6
```

5.66.1.19 b

```
double TRAJECTORY_DATA::b
```

5.66.1.20 chi_p

```
double TRAJECTORY_DATA::chi_p
```

5.66.1.21 rho_p

```
double TRAJECTORY_DATA::rho_p = 8000.0
```

5.66.1.22 Q_in

```
double TRAJECTORY_DATA::Q_in
```

5.66.1.23 V0

```
double TRAJECTORY_DATA::V0
```

5.66.1.24 Y_initial

```
double TRAJECTORY_DATA::Y_initial = 20.0
```

5.66.1.25 dt

```
double TRAJECTORY_DATA::dt
```

5.66.1.26 M

```
double TRAJECTORY_DATA::M
```

5.66.1.27 mp

```
double TRAJECTORY_DATA::mp
```

5.66.1.28 beta

```
double TRAJECTORY_DATA::beta
```

5.66.1.29 q_bar

```
double TRAJECTORY_DATA::q_bar
```

5.66.1.30 sigma_v

```
double TRAJECTORY_DATA::sigma_v
```

5.66.1.31 sigma_vz

```
double TRAJECTORY_DATA::sigma_vz
```

5.66.1.32 sigma_z

```
double TRAJECTORY_DATA::sigma_z
```

5.66.1.33 sigma_n

```
double TRAJECTORY_DATA::sigma_n
```

5.66.1.34 sigma_m

```
double TRAJECTORY_DATA::sigma_m
```

5.66.1.35 n_rand

```
double TRAJECTORY_DATA::n_rand
```

5.66.1.36 m_rand

```
double TRAJECTORY_DATA::m_rand
```

5.66.1.37 s_rand

```
double TRAJECTORY_DATA::s_rand
```

5.66.1.38 t_rand

```
double TRAJECTORY_DATA::t_rand
```

5.66.1.39 POL

```
Matrix<double> TRAJECTORY_DATA::POL
```

5.66.1.40 H

```
Matrix<double> TRAJECTORY_DATA::H
```

5.66.1.41 dX

Matrix<double> TRAJECTORY_DATA::dX

5.66.1.42 dY

Matrix<double> TRAJECTORY_DATA::dY

5.66.1.43 Vr

Matrix<double> TRAJECTORY_DATA::Vr

5.66.1.44 Vt

Matrix<double> TRAJECTORY_DATA::Vt

5.66.1.45 X

Matrix<double> TRAJECTORY_DATA::X

5.66.1.46 Y

Matrix<double> TRAJECTORY_DATA::Y

5.66.1.47 Cap

Matrix<int> TRAJECTORY_DATA::Cap

The documentation for this struct was generated from the following file:

- Trajectory.h

5.67 UI_DATA Struct Reference

Data structure holding the UI arguments.

```
#include <ui.h>
```

Public Attributes

- ValueTypePair value_type

  Data pair for input, tells what the input is and it's type.
- std::vector< std::string > user_input

  What is read in from the console at any point.
- std::vector< std::string > input_files

  A vector of input file names and directories given by user.
- std::string path

  Path to where input files are located.
- int count = 0

  Number of times a questing has been asked.
- int max = 3

  Maximum allowable recursions of a question.
- int option

  Current option choosen by the user.
- bool Path = false

  True if user gives path as an option.
- bool Files = false

  True if user gives input files as an option.
- bool MissingArg = true

  True if an input argument is missing; False if everything is ok.
- bool BasicUI = true

  True if using Basic UI; False if using Advanced UI.
- int argc

  Number of console arguments given on input.

5.67.1 Detailed Description

Data structure holding the UI arguments.

C-Style object for interfacing with users request upon execution of the program. User input is stored in objects below and a series of booleans is used to determine how and what to execute.

5.67.2 Member Data Documentation

5.67.2.1 value_type

```
ValueTypePair UI_DATA::value_type
```

Data pair for input, tells what the input is and it's type.

5.67.2.2 user_input

```
std::vector<std::string> UI_DATA::user_input
```

What is read in from the console at any point.

5.67.2.3 input_files

```
std::vector<std::string> UI_DATA::input_files
```

A vector of input file names and directories given by user.

5.67.2.4 path

```
std::string UI_DATA::path
```

Path to where input files are located.

5.67.2.5 count

```
int UI_DATA::count = 0
```

Number of times a questing has been asked.

5.67.2.6 max

```
int UI_DATA::max = 3
```

Maximum allowable recursions of a question.

5.67.2.7 option

```
int UI_DATA::option
```

Current option choosen by the user.

5.67.2.8 Path

```
bool UI_DATA::Path = false
```

True if user gives path as an option.

5.67.2.9 Files

```
bool UI_DATA::Files = false
```

True if user gives input files as an option.

5.67.2.10 MissingArg

```
bool UI_DATA::MissingArg = true
```

True if an input argument is missing; False if everything is ok.

5.67.2.11 BasicUI

```
bool UI_DATA::BasicUI = true
```

True if using Basic UI; False if using Advanced UI.

5.67.2.12 argc

```
int UI_DATA::argc
```

Number of console arguments given on input.

The documentation for this struct was generated from the following file:

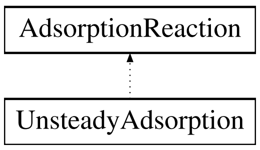
- ui.h

5.68 UnsteadyAdsorption Class Reference

Unsteady Adsorption Reaction Object.

```
#include <shark.h>
```

Inheritance diagram for UnsteadyAdsorption:



Public Member Functions

- UnsteadyAdsorption ()

  Default Constructor.
- ∼UnsteadyAdsorption ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List, int n)

  Function to call the initialization of objects sequentially.
- void Display_Info ()

  Display the adsorption reaction information (PLACE HOLDER)
- void modifyDeltas (MassBalance &mbo)

  Modify the Deltas in the MassBalance Object.
- int setAdsorbIndices ()

  Find and set the adsorbed species indices for each reaction object.
- int checkAqueousIndices ()

  Function to check and report errors in the aqueous species indices.
- void setActivityModelInfo (int(∗act)(const Matrix< double > &logq, Matrix< double > &activity, const void ∗data), const void ∗act_data)

  Function to set the surface activity model and data pointer.
- void setAqueousIndex (int rxn_i, int species_i)

  Set the primary aqueous species index for the ith reaction.
- int setAqueousIndexAuto ()

  Automatically sets the primary aqueous species index based on reactions.
- void setActivityEnum (int act)

  Set the surface activity enum value.
- void setMolarFactor (int rxn_i, double m)

  Set the molar factor for the ith reaction (mol/mol)
- void setVolumeFactor (int i, double v)

  Set the ith volume factor for the species list (cm^3/mol)
- void setAreaFactor (int i, double a)

  Set the ith area factor for the species list (m^2/mol)
- void setSpecificArea (double a)

  Set the specific area for the adsorbent (m^2/kg)
- void setSpecificMolality (double a)

  Set the specific molality for the adsorbent (mol/kg)
- void setSurfaceCharge (double c)

  Set the surface charge of the uncomplexed ligands.
- void setTotalMass (double m)

  Set the total mass of the adsorbent (kg)
- void setTotalVolume (double v)

  Set the total volume of the system (L)
- void setAreaBasisBool (bool opt)

  Set the basis boolean directly.
- void setSurfaceChargeBool (bool opt)

  Set the boolean for inclusion of surface charging.
- void setBasis (std::string option)

  Set the basis of the adsorption problem from the given string arg.
- void setAdsorbentName (std::string name)

  Set the name of the adsorbent to the given string.
- void updateActivities ()

  Set the old activities as the new activities before doing next time step.

- void calculateAreaFactors ()

  Calculates the area factors used from the van der Waals volumes.
- void calculateEquilibria (double T)

  Calculates all equilibrium parameters as a function of temperature.
- void calculateRates (double T)

  Calculates all reaction rate parameters as a function of temperature.
- void setChargeDensity (const Matrix< double > &x)

  Calculates and sets the current value of charge density.
- void setIonicStrength (const Matrix< double > &x)

  Calculates and sets the current value of ionic strength.
- int callSurfaceActivity (const Matrix< double > &x)

  Calls the activity model and returns an int flag for success or failure.
- double calculateActiveFraction (const Matrix< double > &x)

  Calculates the fraction of the surface that is active and available.
- double calculateSurfaceChargeDensity (const Matrix< double > &x)

  Function to calculate the surface charge density based on concentrations.
- double calculatePsi (double sigma, double T, double I, double rel_epsilon)

  Function calculates the Psi (electric surface potential) given a set of arguments.
- double calculateAqueousChargeExchange (int i)

  Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.
- double calculateEquilibriumCorrection (double sigma, double T, double I, double rel_epsilon, int i)

  Function to calculate the correction term for the equilibrium parameter.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_↩ perm, int i)

  Calculates the residual for the ith reaction in the system.
- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old, const Matrix< double > &gama_new, const Matrix< double > &gama_old, double T, double rel_perm, int i)

  Calculates the unsteady residual for the ith reaction in the system.
- double Eval_ReactionRate (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int i)

  Function to calculate the explicit or implicit rate of reaction.
- double Eval_IC_Residual (const Matrix< double > &x, int i)

  Calculate the unsteady residual for initial conditions.
- double Explicit_Eval (const Matrix< double > &x, const Matrix< double > &gama, double T, double rel_perm, int i)

  Return an approximate explicit solution to our unsteady adsorption variable (mol/kg)
- UnsteadyReaction & getReaction (int i)

  Return reference to the ith reaction object in the adsorption object.
- double getMolarFactor (int i)

  Get the ith reaction's molar factor for adsorption (mol/mol)
- double getVolumeFactor (int i)

  Get the ith volume factor (species not involved return zeros) (cm^3/mol)
- double getAreaFactor (int i)

  Get the ith area factor (species not involved return zeros) (m^2/mol)
- double getActivity (int i)

  Get the ith activity factor for the surface species.
- double getOldActivity (int i)

  Get the ith old activity factor for the surface species.
- double getSpecificArea ()

  Get the specific area of the adsorbent (m^2/kg) or (mol/kg)
- double getSpecificMolality ()

Get the specific molality of the adsorbent (mol/kg)
- double getSurfaceCharge ()

    Get the surface charge of the adsorbent.
- double getBulkDensity ()

    Calculate and return bulk density of adsorbent in system (kg/L)
- double getTotalMass ()

    Get the total mass of adsorbent in the system (kg)
- double getTotalVolume ()

    Get the total volume of the system (L)
- double getChargeDensity ()

    Get the value of the surface charge density (C/m^2)
- double getIonicStrength ()

    Get the value of the ionic strength of solution (mol/L)
- int getNumberRxns ()

    Get the number of reactions involved in the adsorption object.
- int getAdsorbIndex (int i)

    Get the index of the adsorbed species in the ith reaction.
- int getAqueousIndex (int i)

    Get the index of the primary aqueous species in the ith reaction.
- int getActivityEnum ()

    Return the enum representing the choosen activity function.
- bool isAreaBasis ()

    Returns true if we are in the Area Basis, False if in Molar Basis.
- bool includeSurfaceCharge ()

    Returns true if we are considering surface charging during adsorption.
- std::string getAdsorbentName ()

    Returns the name of the adsorbent as a string.

Protected Attributes

- Matrix< double > activities_old

    List of the old activities calculated by the activity model.

Private Attributes

- std::vector< UnsteadyReaction > ads_rxn

    List of reactions involved with adsorption.

Additional Inherited Members

5.68.1 Detailed Description

Unsteady Adsorption Reaction Object.

C++ Object to handle data and functions associated with forumlating unsteady adsorption reactions in a aqueous mixture. Each unique surface in a system will require an instance of this structure.

5.68.2 Constructor & Destructor Documentation

5.68.2.1 UnsteadyAdsorption()

```
UnsteadyAdsorption::UnsteadyAdsorption ( )
```

Default Constructor.

5.68.2.2 ∼UnsteadyAdsorption()

```
UnsteadyAdsorption::∼UnsteadyAdsorption ( )
```

Default Destructor.

5.68.3 Member Function Documentation

5.68.3.1 Initialize_Object()

```
void UnsteadyAdsorption::Initialize_Object (
            MasterSpeciesList & List,
            int n )
```

Function to call the initialization of objects sequentially.

5.68.3.2 Display_Info()

```
void UnsteadyAdsorption::Display_Info ( )
```

Display the adsorption reaction information (PLACE HOLDER)

5.68.3.3 modifyDeltas()

```
void UnsteadyAdsorption::modifyDeltas (
            MassBalance & mbo )
```

Modify the Deltas in the MassBalance Object.

This function will take a mass balance object as an argument and modify the deltas in that object to correct for how adsorption affects that particular mass balance. Since adsorption can effect multiple mass balances, this function must be called for each mass balance in the system.

Parameters

mbo	reference to the MassBalance Object the adsorption is acting on

5.68.3.4 setAdsorbIndices()

```
int UnsteadyAdsorption::setAdsorbIndices ( )
```

Find and set the adsorbed species indices for each reaction object.

This function searches through the Reaction objects in UnsteadyAdsorption to find the solid species and their indices to set that information in the adsorb_index structure. That information will be used later to approximate maximum capacities and equilibrium parameters for use in a modified extended Langmuir type expression. Function will return 0 if successful and -1 on a failure.

5.68.3.5 checkAqueousIndices()

```
int UnsteadyAdsorption::checkAqueousIndices ( )
```

Function to check and report errors in the aqueous species indices.

5.68.3.6 setActivityModelInfo()

```
void UnsteadyAdsorption::setActivityModelInfo (
            int(*)(const Matrix< double > &logq, Matrix< double > &activity, const void
*data) act,
            const void * act_data )
```

Function to set the surface activity model and data pointer.

This function will setup the surface activity model based on the given pointer arguments. If no arguments are given, or are given as NULL, then the activity model will default to ideal solution assumption.

5.68.3.7 setAqueousIndex()

```
void UnsteadyAdsorption::setAqueousIndex (
            int rxn_i,
            int species_i )
```

Set the primary aqueous species index for the ith reaction.

5.68.3.8 setAqueousIndexAuto()

```
int UnsteadyAdsorption::setAqueousIndexAuto ( )
```

Automatically sets the primary aqueous species index based on reactions.

This function will go through all species and all reactions in the adsorption object and automatically set the primary aqueous species index based on the stoicheometry of the reaction. It will also check and make sure that the primary aqueous index species appears opposite of the adsorbed species in the reactions. Note: This function assumes that the adsorbed indices have already been set.

5.68.3.9 setActivityEnum()

```
void UnsteadyAdsorption::setActivityEnum (
            int act )
```

Set the surface activity enum value.

5.68.3.10 setMolarFactor()

```
void UnsteadyAdsorption::setMolarFactor (
            int rxn_i,
            double m )
```

Set the molar factor for the ith reaction (mol/mol)

5.68.3.11 setVolumeFactor()

```
void UnsteadyAdsorption::setVolumeFactor (
            int i,
            double v )
```

Set the ith volume factor for the species list (cm^3/mol)

5.68.3.12 setAreaFactor()

```
void UnsteadyAdsorption::setAreaFactor (
            int i,
            double a )
```

Set the ith area factor for the species list (m^2/mol)

5.68.3.13 setSpecificArea()

```
void UnsteadyAdsorption::setSpecificArea (
            double a )
```

Set the specific area for the adsorbent (m^2/kg)

5.68.3.14 setSpecificMolality()

```
void UnsteadyAdsorption::setSpecificMolality (
            double a )
```

Set the specific molality for the adsorbent (mol/kg)

5.68.3.15 setSurfaceCharge()

```
void UnsteadyAdsorption::setSurfaceCharge (
             double c )
```

Set the surface charge of the uncomplexed ligands.

5.68.3.16 setTotalMass()

```
void UnsteadyAdsorption::setTotalMass (
             double m )
```

Set the total mass of the adsorbent (kg)

5.68.3.17 setTotalVolume()

```
void UnsteadyAdsorption::setTotalVolume (
             double v )
```

Set the total volume of the system (L)

5.68.3.18 setAreaBasisBool()

```
void UnsteadyAdsorption::setAreaBasisBool (
             bool opt )
```

Set the basis boolean directly.

5.68.3.19 setSurfaceChargeBool()

```
void UnsteadyAdsorption::setSurfaceChargeBool (
             bool opt )
```

Set the boolean for inclusion of surface charging.

5.68.3.20 setBasis()

```
void UnsteadyAdsorption::setBasis (
             std::string option )
```

Set the basis of the adsorption problem from the given string arg.

5.68.3.21 setAdsorbentName()

```
void UnsteadyAdsorption::setAdsorbentName (
             std::string name )
```

Set the name of the adsorbent to the given string.

5.68.3.22 updateActivities()

```
void UnsteadyAdsorption::updateActivities ( )
```

Set the old activities as the new activities before doing next time step.

5.68.3.23 calculateAreaFactors()

```
void UnsteadyAdsorption::calculateAreaFactors ( )
```

Calculates the area factors used from the van der Waals volumes.

5.68.3.24 calculateEquilibria()

```
void UnsteadyAdsorption::calculateEquilibria (
             double T )
```

Calculates all equilibrium parameters as a function of temperature.

5.68.3.25 calculateRates()

```
void UnsteadyAdsorption::calculateRates (
             double T )
```

Calculates all reaction rate parameters as a function of temperature.

5.68.3.26 setChargeDensity()

```
void UnsteadyAdsorption::setChargeDensity (
             const Matrix< double > & x )
```

Calculates and sets the current value of charge density.

5.68.3.27 setIonicStrength()

```
void UnsteadyAdsorption::setIonicStrength (
             const Matrix< double > & x )
```

Calculates and sets the current value of ionic strength.

5.68.3.28 callSurfaceActivity()

```
int UnsteadyAdsorption::callSurfaceActivity (
             const Matrix< double > & x )
```

Calls the activity model and returns an int flag for success or failure.

5.68.3.29 calculateActiveFraction()

```
double UnsteadyAdsorption::calculateActiveFraction (
             const Matrix< double > & x )
```

Calculates the fraction of the surface that is active and available.

5.68.3.30 calculateSurfaceChargeDensity()

```
double UnsteadyAdsorption::calculateSurfaceChargeDensity (
             const Matrix< double > & x )
```

Function to calculate the surface charge density based on concentrations.

This function is used to calculate the surface charge density of the adsorbed species based on the charges and concentrations of the adsorbed species. The calculation is used to correct the adsorption equilibria constant based on a localized surface charge balance. This requires that you know the molality of the uncomplexed ligand species on the surface, as well as the specific surface area for the adsorbent.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.68.3.31 calculatePsi()

```
double UnsteadyAdsorption::calculatePsi (
             double sigma,
             double T,
             double I,
             double rel_epsilon )
```

Function calculates the Psi (electric surface potential) given a set of arguments.

This function will calculate the electric surface potential of the adsorbent under the current conditions of charge density, temperature, ionic strength, and relative permittivity.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)

5.68.3.32 calculateAqueousChargeExchange()

```
double UnsteadyAdsorption::calculateAqueousChargeExchange (
            int i )
```

Function to calculate the net exchange of charges of the aqeous species involved in a given reaction.

This function will look at all aqueous species involved in the ith adsorption reaction and sum up their stoicheometries and charges to see what the net change in charge is caused by the adsorption of charged species in solution. It is then used to adjust or correct the equilibrium constant for the given adsorption reaction.

Parameters

i	index of the reaction of interest for the adsorption object

5.68.3.33 calculateEquilibriumCorrection()

```
double UnsteadyAdsorption::calculateEquilibriumCorrection (
            double sigma,
            double T,
            double I,
            double rel_epsilon,
            int i )
```

Function to calculate the correction term for the equilibrium parameter.

This function calculates the correction term that gets applied to the equilibrium parameter to correct for surface charge and charge accumulation/depletion effects. It will call the psi approximation and charge exchange functions, therefore it needs to have those functions arguments passed to it as well.

Parameters

sigma	charge density of the surface (C/m^2)
T	temperature of the system in question (K)
I	ionic strength of the medium the surface is in (mol/L)
rel_epsilon	relative permittivity of the medium (Unitless)
i	index of the reaction of interest for the adsorption object

5.68.3.34 Eval_Residual() [1/2]

```
double UnsteadyAdsorption::Eval_Residual (
             const Matrix< double > & x,
             const Matrix< double > & gama,
             double T,
             double rel_perm,
             int i )
```

Calculates the residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.68.3.35 Eval_Residual() [2/2]

```
double UnsteadyAdsorption::Eval_Residual (
             const Matrix< double > & x_new,
             const Matrix< double > & x_old,
             const Matrix< double > & gama_new,
             const Matrix< double > & gama_old,
             double T,
             double rel_perm,
             int i )
```

Calculates the unsteady residual for the ith reaction in the system.

This function will provide a system residual for the ith reaction object involved in the Unsteady Adsorption Reaction. The residual is fed into the SHARK solver to find the solution to solid and aqueous phase concentrations simultaneously. This function will also adjust the equilibrium parameter for the reaction

Parameters

x_new	matrix of the current log(C) concentration values at the current non-linear step
gama_new	matrix of current activity coefficients for each species at the current non-linear step
x_old	matrix of the old log(C) concentration values at the current non-linear step
gama_old	matrix of old activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.68.3.36 Eval_ReactionRate()

```
double UnsteadyAdsorption::Eval_ReactionRate (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            double T,
            double rel_perm,
            int i )
```

Function to calculate the explicit or implicit rate of reaction.

This function will calculate the rate/extent of the unsteady adsorption reaction given the log(C) concentrations and aqueous activities, as well as temperature and permittivity. The temperature and permittivity are used to make surface charge corrections to the equilibria and rate constants.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.68.3.37 Eval_IC_Residual()

```
double UnsteadyAdsorption::Eval_IC_Residual (
            const Matrix< double > & x,
            int i )
```

Calculate the unsteady residual for initial conditions.

Setting the intial conditions for all variables in the system requires a speciation calculation. However, we want the unsteady variables to be set to their respective initial conditions. Using this residual function imposes an equality constraint on those non-linear, unsteady variables allowing the rest of the speciation problem to be solved via PJFNK iterations.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
i	index of the reaction of interest for the adsorption object

5.68.3.38 Explicit_Eval()

```
double UnsteadyAdsorption::Explicit_Eval (
            const Matrix< double > & x,
            const Matrix< double > & gama,
            double T,
            double rel_perm,
            int i )
```

Return an approximate explicit solution to our unsteady adsorption variable (mol/kg)

This function will approximate the concentration of the unsteady variables based on an explicit time discretization. The purpose of this function is to try to provide the PJFNK method with a good initial guess for the values of the non-linear, unsteady variables. If we do not provide a good initial guess to these variables, then the PJFNK method may not converge to the correct solution, because the unsteady problem is the most difficult to solve.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step
T	temperature of the system in question (K)
rel_perm	relative permittivity of the media (unitless)
i	index of the reaction of interest for the adsorption object

5.68.3.39 getReaction()

```
UnsteadyReaction& UnsteadyAdsorption::getReaction (
             int i )
```

Return reference to the ith reaction object in the adsorption object.

5.68.3.40 getMolarFactor()

```
double UnsteadyAdsorption::getMolarFactor (
             int i )
```

Get the ith reaction's molar factor for adsorption (mol/mol)

5.68.3.41 getVolumeFactor()

```
double UnsteadyAdsorption::getVolumeFactor (
             int i )
```

Get the ith volume factor (species not involved return zeros) (cm^3/mol)

5.68.3.42 getAreaFactor()

```
double UnsteadyAdsorption::getAreaFactor (
             int i )
```

Get the ith area factor (species not involved return zeros) (m^2/mol)

5.68.3.43 getActivity()

```
double UnsteadyAdsorption::getActivity (
             int i )
```

Get the ith activity factor for the surface species.

5.68.3.44 getOldActivity()

```
double UnsteadyAdsorption::getOldActivity (
             int i )
```

Get the ith old activity factor for the surface species.

5.68.3.45 getSpecificArea()

```
double UnsteadyAdsorption::getSpecificArea ( )
```

Get the specific area of the adsorbent (m^2/kg) or (mol/kg)

5.68.3.46 getSpecificMolality()

```
double UnsteadyAdsorption::getSpecificMolality ( )
```

Get the specific molality of the adsorbent (mol/kg)

5.68.3.47 getSurfaceCharge()

```
double UnsteadyAdsorption::getSurfaceCharge ( )
```

Get the surface charge of the adsorbent.

5.68.3.48 getBulkDensity()

```
double UnsteadyAdsorption::getBulkDensity ( )
```

Calculate and return bulk density of adsorbent in system (kg/L)

5.68.3.49 getTotalMass()

```
double UnsteadyAdsorption::getTotalMass ( )
```

Get the total mass of adsorbent in the system (kg)

5.68.3.50 getTotalVolume()

```
double UnsteadyAdsorption::getTotalVolume ( )
```

Get the total volume of the system (L)

5.68.3.51 getChargeDensity()

```
double UnsteadyAdsorption::getChargeDensity ( )
```

Get the value of the surface charge density (C/m^2)

5.68.3.52 getIonicStrength()

```
double UnsteadyAdsorption::getIonicStrength ( )
```

Get the value of the ionic strength of solution (mol/L)

5.68.3.53 getNumberRxns()

```
int UnsteadyAdsorption::getNumberRxns ( )
```

Get the number of reactions involved in the adsorption object.

5.68.3.54 getAdsorbIndex()

```
int UnsteadyAdsorption::getAdsorbIndex (
             int i )
```

Get the index of the adsorbed species in the ith reaction.

5.68.3.55 getAqueousIndex()

```
int UnsteadyAdsorption::getAqueousIndex (
             int i )
```

Get the index of the primary aqueous species in the ith reaction.

5.68.3.56 getActivityEnum()

```
int UnsteadyAdsorption::getActivityEnum ( )
```

Return the enum representing the choosen activity function.

5.68.3.57 isAreaBasis()

```
bool UnsteadyAdsorption::isAreaBasis ( )
```

Returns true if we are in the Area Basis, False if in Molar Basis.

5.68.3.58 includeSurfaceCharge()

```
bool UnsteadyAdsorption::includeSurfaceCharge ( )
```

Returns true if we are considering surface charging during adsorption.

5.68.3.59 getAdsorbentName()

```
std::string UnsteadyAdsorption::getAdsorbentName ( )
```

Returns the name of the adsorbent as a string.

5.68.4 Member Data Documentation

5.68.4.1 activities_old

```
Matrix<double> UnsteadyAdsorption::activities_old  [protected]
```

List of the old activities calculated by the activity model.

5.68.4.2 ads_rxn

```
std::vector<UnsteadyReaction> UnsteadyAdsorption::ads_rxn  [private]
```

List of reactions involved with adsorption.

The documentation for this class was generated from the following file:

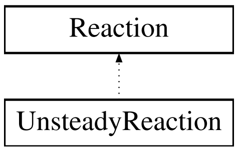
- shark.h

5.69 UnsteadyReaction Class Reference

Unsteady Reaction Object (inherits from Reaction)

```
#include <shark.h>
```

Inheritance diagram for UnsteadyReaction:



Public Member Functions

- UnsteadyReaction ()

  Default Constructor.
- ∼UnsteadyReaction ()

  Default Destructor.
- void Initialize_Object (MasterSpeciesList &List)

  Function to initialize the UnsteadyReaction object from the MasterSpeciesList.
- void Display_Info ()

  Display the unsteady reaction information.
- void Set_Species_Index (int i)

  Set the Unsteady species index by number.
- void Set_Species_Index (std::string formula)

  Set the Unsteady species index by formula.
- void Set_Stoichiometric (int i, double v)

  Set the ith stoichiometric value (see Reaction object)
- void Set_Equilibrium (double v)

  Set the equilibrium constant (logK) (see Reaction object)
- void Set_Enthalpy (double H)

  Set the enthalpy of the reaction (J/mol) (see Reaction object)
- void Set_Entropy (double S)

  Set the entropy of the reaction (J/K/mol) (see Reaction object)
- void Set_EnthalpyANDEntropy (double H, double S)

  Set both the enthalpy and entropy (J/mol) & (J/K/mol) (see Reaction object)
- void Set_Energy (double G)

  Set the Gibb's free energy of reaction (J/mol) (see Reaction object)
- void Set_InitialValue (double ic)

  Set the initial value of the unsteady variable.
- void Set_MaximumValue (double max)

  Set the maximum value of the unsteady variable to a given value max (mol/L)
- void Set_Forward (double forward)

  Set the forward rate for the reaction (mol/L/hr)
- void Set_Reverse (double reverse)

  Set the reverse rate for the reaction (mol/L/hr)
- void Set_ForwardRef (double Fref)

  Set the forward reference rate (mol/L/hr)
- void Set_ReverseRef (double Rref)

Set the reverse reference rate (mol/L/hr)
- void Set_ActivationEnergy (double E)

  Set the activation energy for the reaction (J/mol)
- void Set_Affinity (double b)

  Set the temperature affinity parameter for the reaction.
- void Set_TimeStep (double dt)

  Set the time step for the current simulation.
- void checkSpeciesEnergies ()

  Function to check MasterSpeciesList for species energy info (see Reaction object)
- void calculateEnergies ()

  Function to calculate the energy of the reaction (see Reaction object)
- void calculateEquilibrium (double T)

  Function to calculate the equilibrium constant (see Reaction object)
- void calculateRate (double T)

  Function to calculate the rate constant based on given temperature.
- bool haveEquilibrium ()

  True if equilibrium constant is given or can be calculated (see Reaction object)
- bool haveRate ()

  Function to return true if you have the forward or reverse rate calculated.
- bool haveForwardRef ()

  Function to return true if you have the forward reference rate.
- bool haveReverseRef ()

  Function to return true if you have the reverse reference rate.
- bool haveForward ()

  Function to return true if you have the forward rate.
- bool haveReverse ()

  Function to return true if you have the reverse rate.
- int Get_Species_Index ()

  Fetch the index of the Unsteady species.
- double Get_Stoichiometric (int i)

  Fetch the ith stoichiometric value.
- double Get_Equilibrium ()

  Fetch the equilibrium constant (logK)
- double Get_Enthalpy ()

  Fetch the enthalpy of the reaction.
- double Get_Entropy ()

  Fetch the entropy of the reaction.
- double Get_Energy ()

  Fetch the energy of the reaction.
- double Get_InitialValue ()

  Fetch the initial value of the variable.
- double Get_MaximumValue ()

  Fetch the maximum value of the variable.
- double Get_Forward ()

  Fetch the forward rate.
- double Get_Reverse ()

  Fetch the reverse rate.
- double Get_ForwardRef ()

  Fetch the forward reference rate.
- double Get_ReverseRef ()

  Fetch the reverse reference rate.

- double Get_ActivationEnergy ()

  Fetch the activation energy for the reaction.
- double Get_Affinity ()

  Fetch the temperature affinity for the reaction.
- double Get_TimeStep ()

  Fetch the time step.
- double Eval_ReactionRate (const Matrix< double > &x, const Matrix< double > &gama)

  Calculate reation rate (dC/dt) from concentrations and activities.
- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old, const Matrix< double > &gama_new, const Matrix< double > &gama_old)

  Calculate the unsteady residual for the reaction using and implicit time discretization.
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama)

  Calculate the steady-state residual for this reaction (see Reaction object)
- double Eval_IC_Residual (const Matrix< double > &x)

  Calculate the unsteady residual for initial conditions.
- double Explicit_Eval (const Matrix< double > &x, const Matrix< double > &gama)

  Return an approximate explicit solution to our unsteady variable (mol/L)

Protected Attributes

- double initial_value

  Initial value given at t=0 (in mol/L)
- double max_value

  Maximum value plausible (in mol/L)
- double forward_rate

  Forward reaction rate constant (in (mol/L)^ n/hr)
- double reverse_rate

  Reverse reaction rate constant (in (mol/L)^ n/hr)
- double forward_ref_rate

  Forward reference rate constant (in (mol/L)^ n/hr)
- double reverse_ref_rate

  Reverse reference rate constant (in (mol/L)^ n/hr)
- double activation_energy

  Activation or barrier energy for the reaction (J/mol)
- double temperature_affinity

  Temperature affinity parameter (dimensionless)
- double time_step

  Time step size for current step.
- bool HaveForward

  True if can calculate, or was given the forward rate.
- bool HaveReverse

  True if can calculate, or was given the reverse rate.
- bool HaveForRef

  True if given the forward reference rate.
- bool HaveRevRef

  True if given the reverse reference rate.
- int species_index

  Index in MasterList of Unsteady Species.

Additional Inherited Members

5.69.1 Detailed Description

Unsteady Reaction Object (inherits from Reaction)

C++ style object that holds data and functions associated with unsteady chemical reactions...
i.e., aA + bB <– reverse : forward –> cC + dD
This is essentially the same as the steady reaction, but we now have a forward and reverse reaction rate to deal with. It should be noted that this is a very simple kinetic reaction model based on splitting an overall equilibrium reaction into an overall forward and reverse reaction model. Therefore, it is not expected that this representation of the reaction will provide high accuracy results for reaction kinetics, but should at least provide an overall idea of the process occuring.

5.69.2 Constructor & Destructor Documentation

5.69.2.1 UnsteadyReaction()

```
UnsteadyReaction::UnsteadyReaction ( )
```

Default Constructor.

5.69.2.2 ∼UnsteadyReaction()

```
UnsteadyReaction::∼UnsteadyReaction ( )
```

Default Destructor.

5.69.3 Member Function Documentation

5.69.3.1 Initialize_Object()

```
void UnsteadyReaction::Initialize_Object (
            MasterSpeciesList & List )
```

Function to initialize the UnsteadyReaction object from the MasterSpeciesList.

5.69.3.2 Display_Info()

```
void UnsteadyReaction::Display_Info ( )
```

Display the unsteady reaction information.

5.69.3.3 Set_Species_Index() [1/2]

```
void UnsteadyReaction::Set_Species_Index (
            int i )
```

Set the Unsteady species index by number.

This function will set the unsteady species index by the index i given. That given index must correspond to the index of the species in the MasterSpeciesList that is being considered as the unsteady species.

Parameters

i	index of the unsteady species in the MasterSpeciesList

5.69.3.4 Set_Species_Index() [2/2]

```
void UnsteadyReaction::Set_Species_Index (
             std::string formula )
```

Set the Unsteady species index by formula.

This function will check the MasterSpeciesList for the molecule object that has the given formula, then set the unsteady species index based on the index of that species in the master list.

Parameters

formula	molecular formula of the unsteady species (see mola.h for standard formatting)

5.69.3.5 Set_Stoichiometric()

```
void UnsteadyReaction::Set_Stoichiometric (
             int i,
             double v )
```

Set the ith stoichiometric value (see Reaction object)

5.69.3.6 Set_Equilibrium()

```
void UnsteadyReaction::Set_Equilibrium (
             double v )
```

Set the equilibrium constant (logK) (see Reaction object)

5.69.3.7 Set_Enthalpy()

```
void UnsteadyReaction::Set_Enthalpy (
             double H )
```

Set the enthalpy of the reaction (J/mol) (see Reaction object)

5.69.3.8 Set_Entropy()

```
void UnsteadyReaction::Set_Entropy (
             double S )
```

Set the entropy of the reaction (J/K/mol) (see Reaction object)

5.69.3.9 Set_EnthalpyANDEntropy()

```
void UnsteadyReaction::Set_EnthalpyANDEntropy (
             double H,
             double S )
```

Set both the enthalpy and entropy (J/mol) & (J/K/mol) (see Reaction object)

5.69.3.10 Set_Energy()

```
void UnsteadyReaction::Set_Energy (
             double G )
```

Set the Gibb's free energy of reaction (J/mol) (see Reaction object)

5.69.3.11 Set_InitialValue()

```
void UnsteadyReaction::Set_InitialValue (
             double ic )
```

Set the initial value of the unsteady variable.

This function sets the initial concentration value for the unsteady species to the given value ic (mol/L). Only unsteady species need to be given an initial value. All other species initial values for the overall system is setup based on a speciation calculation performed while holding the unsteady variables constant at their respective initial values.

Parameters

ic	initial concentration value for the unsteady object (mol/L)

5.69.3.12 Set_MaximumValue()

```
void UnsteadyReaction::Set_MaximumValue (
             double max )
```

Set the maximum value of the unsteady variable to a given value max (mol/L)

This function will be called internally to help bound the unsteady variable to reasonable maximum values. That maximum is usually based on the mass balances for the current non-linear iteration.

Parameters

max	maximum allowable value for the unsteady variable (mol/L)

5.69.3.13 Set_Forward()

```
void UnsteadyReaction::Set_Forward (
            double forward )
```

Set the forward rate for the reaction (mol/L/hr)

5.69.3.14 Set_Reverse()

```
void UnsteadyReaction::Set_Reverse (
            double reverse )
```

Set the reverse rate for the reaction (mol/L/hr)

5.69.3.15 Set_ForwardRef()

```
void UnsteadyReaction::Set_ForwardRef (
            double Fref )
```

Set the forward reference rate (mol/L/hr)

Unlike just setting the forward rate, this function sets a reference forward rate of the reaction that can be used to correct the overall forward rate based on system temperature and Arrhenius Rate Equation constants.

Parameters

Fref	forward reference rate constant (mol/L/hr)

5.69.3.16 Set_ReverseRef()

```
void UnsteadyReaction::Set_ReverseRef (
            double Rref )
```

Set the reverse reference rate (mol/L/hr)

Unlike just setting the reverse rate, this function sets a reference reverse rate of the reaction that can be used to correct the overall reverse rate based on system temperature and Arrhenius Rate Equation constants.

Parameters

Rref	reverse reference rate constant (mol/L/hr)

5.69.3.17 Set_ActivationEnergy()

```
void UnsteadyReaction::Set_ActivationEnergy (
            double E )
```

Set the activation energy for the reaction (J/mol)

This function will set the activation energy for the reaction to the given value of E. Note that we will only set one value for activation energy, even though there are rates for forward and reverse reactions. This is because we use the ratio of the rates and the equilibrium constant to establish the other rate. Therefore, we only need either the forward or reverse rate and the equilibrium constant to set all the rates.

Parameters

E	activation energy for the forward or reverse rate, depending on which was given

5.69.3.18 Set_Affinity()

```
void UnsteadyReaction::Set_Affinity (
            double b )
```

Set the temperature affinity parameter for the reaction.

This function will set the temperature affinity for the reaction to the given value of b. Note that we will only set one value for temperature affinity, even though there are rates for forward and reverse reactions. This is because we use the ratio of the rates and the equilibrium constant to establish the other rate. Therefore, we only need either the forward or reverse rate and the equilibrium constant to set all the rates.

Parameters

b	temperature affinity for the forward or reverse rate, depending on which was given

5.69.3.19 Set_TimeStep()

```
void UnsteadyReaction::Set_TimeStep (
            double dt )
```

Set the time step for the current simulation.

5.69.3.20 checkSpeciesEnergies()

```
void UnsteadyReaction::checkSpeciesEnergies ( )
```

Function to check MasterSpeciesList for species energy info (see Reaction object)

5.69.3.21 calculateEnergies()

```
void UnsteadyReaction::calculateEnergies ( )
```

Function to calculate the energy of the reaction (see Reaction object)

5.69.3.22 calculateEquilibrium()

```
void UnsteadyReaction::calculateEquilibrium (
             double T )
```

Function to calculate the equilibrium constant (see Reaction object)

5.69.3.23 calculateRate()

```
void UnsteadyReaction::calculateRate (
             double T )
```

Function to calculate the rate constant based on given temperature.

This function will calculate and set either the forward or reverse rate for the unsteady reaction based on what information was given. If the forward rate information was given, then it sets the reverse rate and visa versa. If nothing was set correctly, an error will occur.

Parameters

T	temperature of the system in Kelvin

5.69.3.24 haveEquilibrium()

```
bool UnsteadyReaction::haveEquilibrium ( )
```

True if equilibrium constant is given or can be calculated (see Reaction object)

5.69.3.25 haveRate()

```
bool UnsteadyReaction::haveRate ( )
```

Function to return true if you have the forward or reverse rate calculated.

5.69.3.26 haveForwardRef()

```
bool UnsteadyReaction::haveForwardRef ( )
```

Function to return true if you have the forward reference rate.

5.69.3.27 haveReverseRef()

```
bool UnsteadyReaction::haveReverseRef ( )
```

Function to return true if you have the reverse reference rate.

5.69.3.28 haveForward()

```
bool UnsteadyReaction::haveForward ( )
```

Function to return true if you have the forward rate.

5.69.3.29 haveReverse()

```
bool UnsteadyReaction::haveReverse ( )
```

Function to return true if you have the reverse rate.

5.69.3.30 Get_Species_Index()

```
int UnsteadyReaction::Get_Species_Index ( )
```

Fetch the index of the Unsteady species.

5.69.3.31 Get_Stoichiometric()

```
double UnsteadyReaction::Get_Stoichiometric (
             int i )
```

Fetch the ith stoichiometric value.

5.69.3.32 Get_Equilibrium()

```
double UnsteadyReaction::Get_Equilibrium ( )
```

Fetch the equilibrium constant (logK)

5.69.3.33 Get_Enthalpy()

```
double UnsteadyReaction::Get_Enthalpy ( )
```

Fetch the enthalpy of the reaction.

5.69.3.34 Get_Entropy()

```
double UnsteadyReaction::Get_Entropy ( )
```

Fetch the entropy of the reaction.

5.69.3.35 Get_Energy()

```
double UnsteadyReaction::Get_Energy ( )
```

Fetch the energy of the reaction.

5.69.3.36 Get_InitialValue()

```
double UnsteadyReaction::Get_InitialValue ( )
```

Fetch the initial value of the variable.

5.69.3.37 Get_MaximumValue()

```
double UnsteadyReaction::Get_MaximumValue ( )
```

Fetch the maximum value of the variable.

5.69.3.38 Get_Forward()

```
double UnsteadyReaction::Get_Forward ( )
```

Fetch the forward rate.

5.69.3.39 Get_Reverse()

```
double UnsteadyReaction::Get_Reverse ( )
```

Fetch the reverse rate.

5.69.3.40 Get_ForwardRef()

```
double UnsteadyReaction::Get_ForwardRef ( )
```

Fetch the forward reference rate.

5.69.3.41 Get_ReverseRef()

```
double UnsteadyReaction::Get_ReverseRef ( )
```

Fetch the reverse reference rate.

5.69.3.42 Get_ActivationEnergy()

```
double UnsteadyReaction::Get_ActivationEnergy ( )
```

Fetch the activation energy for the reaction.

5.69.3.43 Get_Affinity()

```
double UnsteadyReaction::Get_Affinity ( )
```

Fetch the temperature affinity for the reaction.

5.69.3.44 Get_TimeStep()

```
double UnsteadyReaction::Get_TimeStep ( )
```

Fetch the time step.

5.69.3.45 Eval_ReactionRate()

```
double UnsteadyReaction::Eval_ReactionRate (
             const Matrix< double > & x,
             const Matrix< double > & gama )
```

Calculate reation rate (dC/dt) from concentrations and activities.

This function calculates the right hand side of the unsteady reaction equation based on the available rates, the current values of the non-linear variables (x=log(C)), and the activity coefficients (gama).

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step

5.69.3.46 Eval_Residual() [1/2]

```
double UnsteadyReaction::Eval_Residual (
            const Matrix< double > & x_new,
            const Matrix< double > & x_old,
            const Matrix< double > & gama_new,
            const Matrix< double > & gama_old )
```

Calculate the unsteady residual for the reaction using and implicit time discretization.

This function uses the current time step and states of the non-linear variables and activities to form the residual contribution of the unsteady reaction. The time dependent functions are discretized using an implicit finite difference for best stability.

Parameters

x_new	matrix of the log(C) concentration values at the current non-linear step
gama_new	matrix of activity coefficients for each species at the current non-linear step
x_old	matrix of the log(C) concentration values at the previous non-linear step
gama_old	matrix of activity coefficients for each species at the previous non-linear step

5.69.3.47 Eval_Residual() [2/2]

```
double UnsteadyReaction::Eval_Residual (
            const Matrix< double > & x,
            const Matrix< double > & gama )
```

Calculate the steady-state residual for this reaction (see Reaction object)

5.69.3.48 Eval_IC_Residual()

```
double UnsteadyReaction::Eval_IC_Residual (
            const Matrix< double > & x )
```

Calculate the unsteady residual for initial conditions.

Setting the intial conditions for all variables in the system requires a speciation calculation. However, we want the unsteady variables to be set to their respective initial conditions. Using this residual function imposes an equality constraint on those non-linear, unsteady variables allowing the rest of the speciation problem to be solved via PJFNK iterations.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step

5.69.3.49 Explicit_Eval()

```
double UnsteadyReaction::Explicit_Eval (
            const Matrix< double > & x,
            const Matrix< double > & gama )
```

Return an approximate explicit solution to our unsteady variable (mol/L)

This function will approximate the concentration of the unsteady variables based on an explicit time discretization. The purpose of this function is to try to provide the PJFNK method with a good initial guess for the values of the non-linear, unsteady variables. If we do not provide a good initial guess to these variables, then the PJFNK method may not converge to the correct solution, because the unsteady problem is the most difficult to solve.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
gama	matrix of activity coefficients for each species at the current non-linear step

5.69.4 Member Data Documentation

5.69.4.1 initial_value

```
double UnsteadyReaction::initial_value  [protected]
```

Initial value given at t=0 (in mol/L)

5.69.4.2 max_value

```
double UnsteadyReaction::max_value  [protected]
```

Maximum value plausible (in mol/L)

5.69.4.3 forward_rate

```
double UnsteadyReaction::forward_rate  [protected]
```

Forward reaction rate constant (in (mol/L)^n/hr)

5.69.4.4 reverse_rate

```
double UnsteadyReaction::reverse_rate  [protected]
```

Reverse reaction rate constant (in (mol/L)^n/hr)

5.69.4.5 forward_ref_rate

```
double UnsteadyReaction::forward_ref_rate  [protected]
```

Forward reference rate constant (in (mol/L)^n/hr)

5.69.4.6 reverse_ref_rate

```
double UnsteadyReaction::reverse_ref_rate  [protected]
```

Reverse reference rate constant (in (mol/L)^n/hr)

5.69.4.7 activation_energy

```
double UnsteadyReaction::activation_energy  [protected]
```

Activation or barrier energy for the reaction (J/mol)

5.69.4.8 temperature_affinity

```
double UnsteadyReaction::temperature_affinity  [protected]
```

Temperature affinity parameter (dimensionless)

5.69.4.9 time_step

```
double UnsteadyReaction::time_step  [protected]
```

Time step size for current step.

5.69.4.10 HaveForward

```
bool UnsteadyReaction::HaveForward  [protected]
```

True if can calculate, or was given the forward rate.

5.69.4.11 HaveReverse

```
bool UnsteadyReaction::HaveReverse  [protected]
```

True if can calculate, or was given the reverse rate.

5.69.4.12 HaveForRef

```
bool UnsteadyReaction::HaveForRef  [protected]
```

True if given the forward reference rate.

5.69.4.13 HaveRevRef

```
bool UnsteadyReaction::HaveRevRef  [protected]
```

True if given the reverse reference rate.

5.69.4.14 species_index

```
int UnsteadyReaction::species_index  [protected]
```

Index in MasterList of Unsteady Species.

The documentation for this class was generated from the following file:

- shark.h

5.70 ValueTypePair Class Reference

Value-Type Pair object to recognize data type of a string that was read.

```
#include <yaml_wrapper.h>
```

Public Member Functions

- ValueTypePair ()

    Default constructor.
- ∼ValueTypePair ()

    Default destructor.
- ValueTypePair (const std::pair< std::string, int > &vt)

    Constructor by pair.
- ValueTypePair (std::string value, int type)

    Construction by string and int.
- ValueTypePair (const ValueTypePair &vt)

    Copy constructor.
- ValueTypePair & operator= (const ValueTypePair &vt)

    Equals operator overload.
- void editValue (std::string value)

    Edits value to pair with UNKOWN type.
- void editPair (std::string value, int type)

    Creates a paired Value-Type from the given args.
- void findType ()

    Determines the data type of the object.
- void assertType (int type)

    Forces a specific data type.
- void DisplayPair ()

    Display the pair information.
- std::string getString ()

    Returns the value of the pair as a string.
- bool getBool ()

    Returns the value of the pair as a bool.
- double getDouble ()

    Returns the value of the pair as a double.
- int getInt ()

    Returns the value of the pair as an int.
- std::string getValue ()

    Returns the value of the pair as it was given.
- int getType ()

    Returns the type of the pair.
- std::pair< std::string, int > & getPair ()

    Returns reference to the actual pair object.

Private Attributes

- std::pair< std::string, int > Value_Type

    pair object holding the Value and Type info
- int type

    Type of the value.

5.70.1 Detailed Description

Value-Type Pair object to recognize data type of a string that was read.

C++ Object that creates a pair between a read in value as a string and an enum denoting what the data type of that string is. This object is primarily used in the other yaml_wrapper objects, but can also be used for any string that you want to parse to identify it's type. The supported types are denoted in the data_type enum and can be determined automatically by the findType() function or can be specified by the assertType() function.

5.70.2 Constructor & Destructor Documentation

5.70.2.1 ValueTypePair() [1/4]

```
ValueTypePair::ValueTypePair ( )
```

Default constructor.

5.70.2.2 ∼ValueTypePair()

```
ValueTypePair::∼ValueTypePair ( )
```

Default destructor.

5.70.2.3 ValueTypePair() [2/4]

```
ValueTypePair::ValueTypePair (
            const std::pair< std::string, int > & vt )
```

Constructor by pair.

5.70.2.4 ValueTypePair() [3/4]

```
ValueTypePair::ValueTypePair (
            std::string value,
            int type )
```

Construction by string and int.

5.70.2.5 ValueTypePair() [4/4]

```
ValueTypePair::ValueTypePair (
            const ValueTypePair & vt )
```

Copy constructor.

5.70.3 Member Function Documentation

5.70.3.1 operator=()

```
ValueTypePair& ValueTypePair::operator= (
             const ValueTypePair & vt )
```

Equals operator overload.

5.70.3.2 editValue()

```
void ValueTypePair::editValue (
             std::string value )
```

Edits value to pair with UNKOWN type.

5.70.3.3 editPair()

```
void ValueTypePair::editPair (
             std::string value,
             int type )
```

Creates a paired Value-Type from the given args.

5.70.3.4 findType()

```
void ValueTypePair::findType ( )
```

Determines the data type of the object.

5.70.3.5 assertType()

```
void ValueTypePair::assertType (
             int type )
```

Forces a specific data type.

5.70.3.6 DisplayPair()

```
void ValueTypePair::DisplayPair ( )
```

Display the pair information.

5.70.3.7 getString()

```
std::string ValueTypePair::getString ( )
```

Returns the value of the pair as a string.

5.70.3.8 getBool()

```
bool ValueTypePair::getBool ( )
```

Returns the value of the pair as a bool.

5.70.3.9 getDouble()

```
double ValueTypePair::getDouble ( )
```

Returns the value of the pair as a double.

5.70.3.10 getInt()

```
int ValueTypePair::getInt ( )
```

Returns the value of the pair as an int.

5.70.3.11 getValue()

```
std::string ValueTypePair::getValue ( )
```

Returns the value of the pair as it was given.

5.70.3.12 getType()

```
int ValueTypePair::getType ( )
```

Returns the type of the pair.

5.70.3.13 getPair()

```
std::pair<std::string,int>& ValueTypePair::getPair ( )
```

Returns reference to the actual pair object.

5.70.4 Member Data Documentation

5.70.4.1 Value_Type

```
std::pair<std::string,int> ValueTypePair::Value_Type  [private]
```

pair object holding the Value and Type info

5.70.4.2 type

```
int ValueTypePair::type  [private]
```

Type of the value.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.71 Vector3D Class Reference

3D Vector Object

```
#include <mesh.h>
```

Public Member Functions

- Vector3D ()
  Default Constructor.
- ∼Vector3D ()
  Default Destructor.
- Vector3D (double x, double y, double z)
  Construction of RVector with each component.
- Vector3D (const Vector3D &v)
  Copy constructor for the vector.
- double & operator() (int i)
  Access to reference of a component of the vector.
- double operator() (int i) const
  Access to a component of the vector.
- double norm ()
  Calculation of the 2-Norm of the vector.
- double dot_product (const Vector3D &v)
  Perform the dot product between two vectors.
- double angleRAD (Vector3D &v)
  Returns the angle between the two vectors in radians.
- double angleDEG (Vector3D &v)
  Returns the angle between the two vectors in degrees.

- void edit (int i, double value)

  Editing a single value in the vector.
- void set_vector (double x, double y, double z)

  Editing all values in a vector.
- Vector3D & operator= (const Vector3D &v)

  Vector assignment.
- Vector3D operator+ (const Vector3D &v)

  Vector Addition.
- Vector3D operator- (const Vector3D &v)

  Vector Subtraction.
- double operator$*$ (const Vector3D &v)

  Vector dot product (short hand = this'$*$v)
- Vector3D operator$*$ (const double a)

  Vector-scalar multiplication.
- Vector3D operator/ (const double a)

  Vector-scalar division.
- Vector3D cross_product (const Vector3D &v)

  Vector cross product.

Protected Attributes

- Matrix$<$ double $>$ vector

  Matrix object to store vector data.

5.71.1 Detailed Description

3D Vector Object

This class structure creates a C++ object for a vector in 3D space. Built using the MACAW matrix object, this object contains functions and data associated with working with vectors in 3D space.

5.71.2 Constructor & Destructor Documentation

5.71.2.1 Vector3D() [1/3]

```
Vector3D::Vector3D ( )
```

Default Constructor.

5.71.2.2 ∼Vector3D()

```
Vector3D::∼Vector3D ( )
```

Default Destructor.

5.71.2.3 Vector3D() [2/3]

```
Vector3D::Vector3D (
             double x,
             double y,
             double z )
```

Construction of RVector with each component.

5.71.2.4 Vector3D() [3/3]

```
Vector3D::Vector3D (
             const Vector3D & v )
```

Copy constructor for the vector.

5.71.3 Member Function Documentation

5.71.3.1 operator()() [1/2]

```
double& Vector3D::operator() (
             int i )
```

Access to reference of a component of the vector.

5.71.3.2 operator()() [2/2]

```
double Vector3D::operator() (
             int i ) const
```

Access to a component of the vector.

5.71.3.3 norm()

```
double Vector3D::norm ( )
```

Calculation of the 2-Norm of the vector.

5.71.3.4 dot_product()

```
double Vector3D::dot_product (
             const Vector3D & v )
```

Perform the dot product between two vectors.

5.71.3.5 angleRAD()

```
double Vector3D::angleRAD (
             Vector3D & v )
```

Returns the angle between the two vectors in radians.

5.71.3.6 angleDEG()

```
double Vector3D::angleDEG (
             Vector3D & v )
```

Returns the angle between the two vectors in degrees.

5.71.3.7 edit()

```
void Vector3D::edit (
             int i,
             double value )
```

Editing a single value in the vector.

5.71.3.8 set_vector()

```
void Vector3D::set_vector (
             double x,
             double y,
             double z )
```

Editing all values in a vector.

5.71.3.9 operator=()

```
Vector3D& Vector3D::operator= (
             const Vector3D & v )
```

Vector assignment.

5.71.3.10 operator+()

```
Vector3D Vector3D::operator+ (
             const Vector3D & v )
```

Vector Addition.

5.71.3.11 operator-()

```
Vector3D Vector3D::operator- (
             const Vector3D & v )
```

Vector Subtraction.

5.71.3.12 operator*() [1/2]

```
double Vector3D::operator* (
             const Vector3D & v )
```

Vector dot product (short hand = this'*v)

5.71.3.13 operator*() [2/2]

```
Vector3D Vector3D::operator* (
             const double a )
```

Vector-scalar multiplication.

5.71.3.14 operator/()

```
Vector3D Vector3D::operator/ (
             const double a )
```

Vector-scalar division.

5.71.3.15 cross_product()

```
Vector3D Vector3D::cross_product (
             const Vector3D & v )
```

Vector cross product.

5.71.4 Member Data Documentation

5.71.4.1 vector

```
Matrix<double> Vector3D::vector  [protected]
```

Matrix object to store vector data.

The documentation for this class was generated from the following file:

- mesh.h

5.72 VolumeElement Class Reference

VolumeElement Object.

```
#include <mesh.h>
```

Public Member Functions

- VolumeElement ()

  Default Constructor.
- ∼VolumeElement ()

  Default Destructor.
- void DisplayInfo ()

  Print out information to the console.
- void AssignNodes (Node &n1, Node &n2, Node &n3, Node &n4)

  Assign nodes for the volume element.
- void AssignIDnumber (unsigned int i)

  Assign the id number for the volume element.
- void calculateVolume ()

  Calculate and store volume value.
- void findCentroid ()

  Find and set the centroid node.
- void evaluateProperties ()

  Calls functions for volume and centroid.

Private Attributes

- Node ∗ node1

  Pointer to first node.
- Node ∗ node2

  Pointer to second node.
- Node ∗ node3

  Pointer to third node.
- Node ∗ node4

  Pointer to fourth node.
- double volume

  Area of the volume element.
- Node centroid

  Centroid node for the volume element.
- unsigned int IDnum

  Identification number for the volume element.

5.72.1 Detailed Description

VolumeElement Object.

This class structure creates a C++ object for a volume element. The volume is made up of a reference to four distinct nodes. Based on those nodes, we can formulate a set of surfaces that encapsulates the volume element. The volume element will also have an identification number, a volume, and a centroid.

5.72.2 Constructor & Destructor Documentation

5.72.2.1 VolumeElement()

```
VolumeElement::VolumeElement ( )
```

Default Constructor.

5.72.2.2 ∼VolumeElement()

```
VolumeElement::∼VolumeElement ( )
```

Default Destructor.

5.72.3 Member Function Documentation

5.72.3.1 DisplayInfo()

```
void VolumeElement::DisplayInfo ( )
```

Print out information to the console.

5.72.3.2 AssignNodes()

```
void VolumeElement::AssignNodes (
             Node & n1,
             Node & n2,
             Node & n3,
             Node & n4 )
```

Assign nodes for the volume element.

5.72.3.3 AssignIDnumber()

```
void VolumeElement::AssignIDnumber (
             unsigned int i )
```

Assign the id number for the volume element.

5.72.3.4 calculateVolume()

```
void VolumeElement::calculateVolume ( )
```

Calculate and store volume value.

5.72.3.5 findCentroid()

```
void VolumeElement::findCentroid ( )
```

Find and set the centroid node.

5.72.3.6 evaluateProperties()

```
void VolumeElement::evaluateProperties ( )
```

Calls functions for volume and centroid.

5.72.4 Member Data Documentation

5.72.4.1 node1

```
Node* VolumeElement::node1  [private]
```

Pointer to first node.

5.72.4.2 node2

```
Node* VolumeElement::node2  [private]
```

Pointer to second node.

5.72.4.3 node3

```
Node* VolumeElement::node3  [private]
```

Pointer to third node.

5.72.4.4 node4

```
Node* VolumeElement::node4  [private]
```

Pointer to fourth node.

5.72.4.5 volume

```
double VolumeElement::volume  [private]
```

Area of the volume element.

5.72.4.6 centroid

```
Node VolumeElement::centroid  [private]
```

Centroid node for the volume element.

5.72.4.7 IDnum

```
unsigned int VolumeElement::IDnum  [private]
```

Identification number for the volume element.

The documentation for this class was generated from the following file:

- mesh.h

5.73 yaml_cpp_class Class Reference

Primary object used when reading and digitally storing yaml files.

```
#include <yaml_wrapper.h>
```

Public Member Functions

- yaml_cpp_class ()

  Default constructor.
- ∼yaml_cpp_class ()

  Default destructor.
- int setInputFile (const char ∗file)

  Set the input file to be read.
- int readInputFile ()

  Reads through input file and stores into YamlWrapper.
- int cleanup ()

  Deletes yaml_c objects and closes the input file.
- int executeYamlRead (const char ∗file)

  Runs the full execution of initialization, reading, and cleaning.
- YamlWrapper & getYamlWrapper ()

  Returns reference to the YamlWrapper Object.
- void DisplayContents ()

  Print out the contents of the read to the console window.
- void DeleteContents ()

  Delete the data in the yaml_wrapper to free up space.

Private Attributes

- YamlWrapper yaml_wrapper

  YamlWrapper object where digital file is stored.
- FILE ∗ input_file

  Function pointer to the yaml formatted file.
- const char ∗ file_name

  Name of the file to be parsed and read.
- yaml_parser_t token_parser

  C-YAML parser object for token based parsing.
- yaml_token_t current_token

  C-YAML token object for the current token in the file.
- yaml_token_t previous_token

  C-YAML token object for the previous token in the file.

5.73.1 Detailed Description

Primary object used when reading and digitally storing yaml files.

C++ Object that holds the YamlWrapper object and the C-YAML objects necessary for reading and parsing a yaml formatted file. This is the primary object that users are expected to work with when using yaml_wrapper.h to read input files. It contains functions necessary to setup a read instance, read and parse the input, place the parsed input results into the digital YamlWrapper object, and allow the user to query that object.

The two main functions that the typical user will need are: (i) executeYamlRead() and (ii) getYamlWrapper. Make sure that the read function was called prior to querying the YamlWrapper structure. Do not call the cleanup() function if using executeYamlRead(). That function will be automattically called after the read is complete.

5.73.2 Constructor & Destructor Documentation

5.73.2.1 yaml_cpp_class()

```
yaml_cpp_class::yaml_cpp_class ( )
```

Default constructor.

5.73.2.2 ∼yaml_cpp_class()

```
yaml_cpp_class::∼yaml_cpp_class ( )
```

Default destructor.

5.73.3 Member Function Documentation

5.73.3.1 setInputFile()

```
int yaml_cpp_class::setInputFile (
             const char * file )
```

Set the input file to be read.

5.73.3.2 readInputFile()

```
int yaml_cpp_class::readInputFile ( )
```

Reads through input file and stores into YamlWrapper.

5.73.3.3 cleanup()

```
int yaml_cpp_class::cleanup ( )
```

Deletes yaml_c objects and closes the input file.

5.73.3.4 executeYamlRead()

```
int yaml_cpp_class::executeYamlRead (
             const char * file )
```

Runs the full execution of initialization, reading, and cleaning.

5.73.3.5 getYamlWrapper()

```
YamlWrapper& yaml_cpp_class::getYamlWrapper ( )
```

Returns reference to the YamlWrapper Object.

5.73.3.6 DisplayContents()

```
void yaml_cpp_class::DisplayContents ( )
```

Print out the contents of the read to the console window.

5.73.3.7 DeleteContents()

```
void yaml_cpp_class::DeleteContents ( )
```

Delete the data in the yaml_wrapper to free up space.

5.73.4 Member Data Documentation

5.73.4.1 yaml_wrapper

```
YamlWrapper yaml_cpp_class::yaml_wrapper  [private]
```

YamlWrapper object where digital file is stored.

5.73.4.2 input_file

```
FILE* yaml_cpp_class::input_file  [private]
```

Function pointer to the yaml formatted file.

5.73.4.3 file_name

```
const char* yaml_cpp_class::file_name  [private]
```

Name of the file to be parsed and read.

5.73.4.4 token_parser

```
yaml_parser_t yaml_cpp_class::token_parser  [private]
```

C-YAML parser object for token based parsing.

5.73.4.5 current_token

```
yaml_token_t yaml_cpp_class::current_token  [private]
```

C-YAML token object for the current token in the file.

5.73.4.6 previous_token

```
yaml_token_t yaml_cpp_class::previous_token  [private]
```

C-YAML token object for the previous token in the file.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

5.74 YamlWrapper Class Reference

Object for the entire yaml file holding all documents, header, sub-headers, keys, and values.

```
#include <yaml_wrapper.h>
```

Public Member Functions

- YamlWrapper ()

  Default constructor.
- ∼YamlWrapper ()

  Default destructor.
- YamlWrapper (const YamlWrapper &yaml)

  Copy constructor.
- YamlWrapper (std::string key, const Document &doc)

  Constructor by a single document.
- YamlWrapper & operator= (const YamlWrapper &yaml)

  Equals overload.
- Document & operator() (const std::string key)

  Return the Document reference at the given key.
- Document operator() (const std::string key) const

  Return the Document at the given key.
- std::map< std::string, Document > & getDocMap ()

  Return reference to the document map.
- Document & getDocument (std::string key)

  Return reference to the document at the key.
- std::map< std::string, Document >::const_iterator end () const

  Returns a const iterator pointing to the end of the list.
- std::map< std::string, Document >::iterator end ()

  Returns an iterator pointing to the end of the list.
- std::map< std::string, Document >::const_iterator begin () const

  Returns a const iterator pointing to the begining of the list.
- std::map< std::string, Document >::iterator begin ()

  Returns an iterator pointing to the begining of the list.
- void clear ()

  Clear out the yaml object.
- void resetKeys ()

  Resets all the keys in DocumentMap to match document names.
- void changeKey (std::string oldKey, std::string newKey)

  Change a given oldKey in the map to the newKey given.
- void revalidateAllKeys ()

  Resets and validates all keys in the structure.
- void DisplayContents ()

  Display the contents of the wrapper.
- void addDocKey (std::string key)

  Add a key to the document map.
- void copyAnchor2Alias (std::string alias, Document &ref)

  Find the anchor in the map, and copy to the Document reference given.
- int size ()

  Return the size of the document map.

- Document & getAnchoredDoc (std::string alias)

  Return the reference to the document that is anchored with the given alias.
- Document & getDocFromHeadAlias (std::string alias)

  Return reference to the document that contains the header with the given alias.
- Document & getDocFromSubAlias (std::string alias)

  Return reference to the document that contains the subheader with the given alias.

Private Attributes

- std::map< std::string, Document > Doc_Map

  Map of the documents contained within the wrapper.

5.74.1 Detailed Description

Object for the entire yaml file holding all documents, header, sub-headers, keys, and values.

C++ Object for the yaml file. This object holds a map of all Documents in the yaml file. Each document holds a map of Key-values and Headers. The Headers hold maps of Key-values and SubHeaders, and each SubHeader can hold more Key-values.

This object is used to represent a digital and queryable structure for all information contained within a yaml file. There are some limitations to what can be held here, and those limitations are based on the limitations in the C-YAML Library token parser. The main limitation is that the deepest level of allowable recursion in the file is Sub↩ Header. Meaning that you are not allowed to have Sub-SubHeaders underneath a SubHeader object. This imposes a hard limit to number of nested lists that can be in a single Document object.

When using yaml_cpp_class, this object will generally be what you query into to get the information from your yaml input files. From this object, functions and operators are provided to give you the capability of querying down into any allowable level of the file by the keys that are were used in the file. Be sure that you are querying the correct objects by the correct keys, otherwise errors and exceptions will be thrown.

5.74.2 Constructor & Destructor Documentation

5.74.2.1 YamlWrapper() [1/3]

```
YamlWrapper::YamlWrapper ( )
```

Default constructor.

5.74.2.2 ∼YamlWrapper()

```
YamlWrapper::∼YamlWrapper ( )
```

Default destructor.

5.74.2.3 YamlWrapper() [2/3]

```
YamlWrapper::YamlWrapper (
             const YamlWrapper & yaml )
```

Copy constructor.

5.74.2.4 YamlWrapper() [3/3]

```
YamlWrapper::YamlWrapper (
             std::string key,
             const Document & doc )
```

Constructor by a single document.

5.74.3 Member Function Documentation

5.74.3.1 operator=()

```
YamlWrapper& YamlWrapper::operator= (
             const YamlWrapper & yaml )
```

Equals overload.

5.74.3.2 operator()() [1/2]

```
Document& YamlWrapper::operator() (
             const std::string key )
```

Return the Document reference at the given key.

5.74.3.3 operator()() [2/2]

```
Document YamlWrapper::operator() (
             const std::string key ) const
```

Return the Document at the given key.

5.74.3.4 getDocMap()

```
std::map<std::string, Document>& YamlWrapper::getDocMap ( )
```

Return reference to the document map.

5.74.3.5 getDocument()

```
Document& YamlWrapper::getDocument (
              std::string key )
```

Return reference to the document at the key.

5.74.3.6 end() [1/2]

```
std::map<std::string, Document>::const_iterator YamlWrapper::end ( ) const
```

Returns a const iterator pointing to the end of the list.

5.74.3.7 end() [2/2]

```
std::map<std::string, Document>::iterator YamlWrapper::end ( )
```

Returns an iterator pointing to the end of the list.

5.74.3.8 begin() [1/2]

```
std::map<std::string, Document>::const_iterator YamlWrapper::begin ( ) const
```

Returns a const iterator pointing to the begining of the list.

5.74.3.9 begin() [2/2]

```
std::map<std::string, Document>::iterator YamlWrapper::begin ( )
```

Returns an iterator pointing to the begining of the list.

5.74.3.10 clear()

```
void YamlWrapper::clear ( )
```

Clear out the yaml object.

5.74.3.11 resetKeys()

```
void YamlWrapper::resetKeys ( )
```

Resets all the keys in DocumentMap to match document names.

5.74.3.12 changeKey()

```
void YamlWrapper::changeKey (
             std::string oldKey,
             std::string newKey )
```

Change a given oldKey in the map to the newKey given.

5.74.3.13 revalidateAllKeys()

```
void YamlWrapper::revalidateAllKeys ( )
```

Resets and validates all keys in the structure.

5.74.3.14 DisplayContents()

```
void YamlWrapper::DisplayContents ( )
```

Display the contents of the wrapper.

5.74.3.15 addDocKey()

```
void YamlWrapper::addDocKey (
             std::string key )
```

Add a key to the document map.

5.74.3.16 copyAnchor2Alias()

```
void YamlWrapper::copyAnchor2Alias (
             std::string alias,
             Document & ref )
```

Find the anchor in the map, and copy to the Document reference given.

5.74.3.17 size()

```
int YamlWrapper::size ( )
```

Return the size of the document map.

5.74.3.18 getAnchoredDoc()

```
Document& YamlWrapper::getAnchoredDoc (
            std::string alias )
```

Return the reference to the document that is anchored with the given alias.

5.74.3.19 getDocFromHeadAlias()

```
Document& YamlWrapper::getDocFromHeadAlias (
            std::string alias )
```

Return reference to the document that contains the header with the given alias.

5.74.3.20 getDocFromSubAlias()

```
Document& YamlWrapper::getDocFromSubAlias (
            std::string alias )
```

Return reference to the document that contains the subheader with the given alias.

5.74.4 Member Data Documentation

5.74.4.1 Doc_Map

```
std::map<std::string, Document> YamlWrapper::Doc_Map  [private]
```

Map of the documents contained within the wrapper.

The documentation for this class was generated from the following file:

- yaml_wrapper.h

6 File Documentation

6.1 cardinal.h File Reference

Cloud-rise And Radioactivity Distribution Invoked from Nuclear Arms Launch.

```
#include "kea.h"
#include "crane.h"
```

Classes

- class Cardinal

    C++ object for coupling cloud rise and activity distribution.

Functions

- int CARDINAL_SCENARIO (const char ∗yaml_input, const char ∗atmosphere_data, const char ∗data_path)
- int cardinal_simulation (const char ∗yaml_input, const char ∗atm_data, const char ∗nuc_path, int unc_opt, const char ∗cloud_rise_out, const char ∗cloud_growth_out, const char ∗nuc_out)

    C-style interface for use by python 3.5 (or newer)

6.1.1 Detailed Description

Cloud-rise And Radioactivity Distribution Invoked from Nuclear Arms Launch.

cardinal.cpp

This file contains a C++ object for coupling cloud rise simulations and activity-size distributions. It will link together libraries for nuclide half-lifes and decay modes with libraries for fission product yields from different nuclear materials. User must provide the common path to both the nuclide data and fission product data. User must also provide input files to control the cloud rise simulation and provide atomspheric data.

Author

Austin Ladshaw

Date

02/22/2019

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for research in the area of radioactive particle decay and transport. Copyright (c) 2019, all rights reserved.

6.1.2 Function Documentation

6.1.2.1 CARDINAL_SCENARIO()

```
int CARDINAL_SCENARIO (
            const char * yaml_input,
            const char * atmosphere_data,
            const char * data_path )
```

6.1.2.2 cardinal_simulation()

```
int cardinal_simulation (
             const char * yaml_input,
             const char * atm_data,
             const char * nuc_path,
             int unc_opt,
             const char * cloud_rise_out,
             const char * cloud_growth_out,
             const char * nuc_out )
```

C-style interface for use by python 3.5 (or newer)

6.2 crane.h File Reference

Cloud Rise After Nuclear Explosion.

```
#include "dove.h"
#include "mola.h"
```

Classes

- class Crane

  CRANE object to hold data and functions associated with Cloud Rise.

Functions

- double rate_cloud_rise (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled cloud rise residual.
- double rate_cloud_alt (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled cloud altitude residual.
- double rate_x_water_vapor (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled water ratio residual.
- double rate_temperature (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled temperature residual.
- double rate_w_water_conds (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled water fraction residual.
- double rate_energy (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled energy residual.
- double rate_cloud_mass (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled cloud mass residual.
- double rate_s_soil (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled soil ratio residual.
- double rate_entrained_mass (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Function to provide a coupled entrained mass residual.
- int CRANE_SCENARIO (const char ∗yaml_input, const char ∗atmosphere_data)

  CRANE Executable given an input file.
- int CRANE_TESTS ()

  Test function for CRANE.

6.2.1 Detailed Description

Cloud Rise After Nuclear Explosion.

This file creates objects and subroutines for solving the systems of equations for the mass, energy, and temperature of debris clouds caused by nuclear detonations. The equations solved and methods used come from the DEfense Land Fallout Interpretative Code (DELFIC) developed by U.S. DOD in the 1960s to 1970s. The original DELFIC software was written in Fortran77 and the source code is not available to the public. This software is a recreation of the Cloud Rise Module from DELFIC based on the reports made publically available. In this software, we are only interested in estimating the cloud rise and the shape of the nuclear debris cloud post- detonation of a nuclear weapon. This software does not perform any transport of the resulting fallout cloud of debris. Transport will be handled by a different code for modeling systems of PDEs. The cloud rise simulation performed here will become the initial conditions for a transport model that is to be developed later.

References for DELFIC

H.G. Normet, "DELFIC: Department of Defense Fallout Prediction System - Volume I - Fundamentals," U.S. DOD, DNA-001-76-C-0010, DNA 5159F-1, December 1979.

H.G. Normet, "DELFIC: Department of Defense Fallout Prediction System - Volume II - User's Manual," U.S. DOD, DNA-001-76-C-0010, DNA 5159F-2, December 1979.

H.G. Normet and S. Woolf, "Department of Defense Land Fallout Prediction System - Volume III - Cloud Rise," U.S. DOD, DASA01-69-C-0077, DASA-1800-III, September 1970.

References for Default Atmospheric Profile

Engineering ToolBox, (2001). [online] Available at: https://www.engineeringtoolbox.com Accessed on June 13, 2018.

C.J. Brasefield, "Winds at Altitudes up to 80 Kilometers," J. of Geophysical Research, 59, 233-237, 1954.

References for Default Soil Characteristics

J.B. Hanni, E. Pressly, J.V. Crum, K.B.C. Minister, D.Tran, "Liquidus temperature measurements for modeling oxide glass systems relevant to nuclear waste vitrification," J. Mater. Res., 20, 3346-3357, 2005.

Q. Rao, G.F. Piepel, P. Hrma, J.V. Crum, "Liquidus temperatures of HLW glasses with zirconium- containing primary crystalline phases," J. Non-Crystalline Solids, 220, 17-29, 1997.

Author

Austin Ladshaw

Date

05/30/2018

Copyright

6.2.2 Function Documentation

6.2.2.1 rate_cloud_rise()

```
double rate_cloud_rise (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled cloud rise residual.

6.2.2.2 rate_cloud_alt()

```
double rate_cloud_alt (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled cloud altitude residual.

6.2.2.3 rate_x_water_vapor()

```
double rate_x_water_vapor (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled water ratio residual.

6.2.2.4 rate_temperature()

```
double rate_temperature (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled temperature residual.

6.2.2.5 rate_w_water_conds()

```
double rate_w_water_conds (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled water fraction residual.

6.2.2.6 rate_energy()

```
double rate_energy (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled energy residual.

6.2.2.7 rate_cloud_mass()

```
double rate_cloud_mass (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled cloud mass residual.

6.2.2.8 rate_s_soil()

```
double rate_s_soil (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Function to provide a coupled soil ratio residual.

6.2.2.9 rate_entrained_mass()

```
double rate_entrained_mass (
            int i,
            const Matrix< double > & u,
            double t,
            const void * data,
            const Dove & dove )
```

Function to provide a coupled entrained mass residual.

6.2.2.10 CRANE_SCENARIO()

```
int CRANE_SCENARIO (
            const char * yaml_input,
            const char * atmosphere_data )
```

CRANE Executable given an input file.

Main CRANE executable from the ecosystem cli. User must provide a yaml-style input file directing all simulation, solver, and runtime options. The second file is optional and will contain meteorlogical data as a function of hieght above mean sea level.

Note

The atmosphere file must be in a specific format. Each line of the file must read off the following, in order, using the noted units: (1) altitude [m], (2) temperature [K], (3) pressure [Pa], and (4) relative humidity (%). Then, start a new line for each change in altitude. Wind data is optional and is given in the Yaml Input File instead. This is because wind readings do not often correspond to the same altitudes as pressure and temperature readings. Also note that the first line of the atmosphere file is NOT read in by the program. It is treated as just a header.

Example Yaml Input File

Simulation_Conditions:

bomb_yield: 50 #kT burst_height: 0 #m above ground ground_level: 500 #m above mean sea level particle_bins: 10 #number of particle size distributions tight_coupling: true #use tight or loose coupling for variables shear_↩
correction: true #use a correction for shear velocity (requires wind profile)

...

ODE_Options:

file_output: true #print results to a file console_output: false #print messages to the console window after each step integration_method: bdf2 #choices: be, bdf2, fe, cn, rk4, rkf time_stepper: adaptive #choices: constant, adaptive, fehlberg, ratebased preconditioner: sgs #choices: jacobi, ugs, lgs, sgs tolerance: 0.001 #explicit solver tolerance dtmin: 1e-8 #minimum allowable time step dtmax: 0.1 #maximum allowable time step converged_dtmin: 0.001 #minimum allowbable time step after convergence time_out: 1.0 #number of seconds between each print-to-file action end_time: 1000.0 #number of seconds until simulation forced to end

...

Solver_Options:

linear_method: qr #choices: gmreslp, pcg, bicgstab, cgs, fom, gmresrp, gcr, gmresr, kms, gmres, qr line_search: bt #choices: none, bt, abt linear: false #treat system as linear (default = false) precondition: false #use a preconditioner (default = false) nl_out: false #print non-linear residuals to console lin_out: false #print linear residuals to console max_nl_iter: 50 #maximum allowable non-linear iterations max_lin_iter: 200 #maximum allowable linear iterations restart_limit: 20 #number of allowable vector spans before restart recursion_limit: 2 #number of allowable recurives calls for preconditioning nl_abs_tol: 1e-6 #Absolute tolerance for non-linear iterations nl_rel_tol: 1e-6 #Relative tolerance for non-linear iterations lin_abs_tol: 1e-6 #Absolute tolerance for linear iterations lin_rel_tol: 1e-4 #↩
Relative tolerance for linear iterations

...

Wind_Profile:

#user provides lists of velocity components at various altitude values #name of each list is the altitude in m #under each list is vx and vy in m/s at corresponding altitude #NOTE: This entire document for wind is optional (a default can be used)

- 216: vx: -5.14 vy: 6.13
- 1548: vx: -5.494 vy: 11.78
- 3097: vx: 0.8582 vy: 4.924
- 5688: vx: 5.13 vy: 14.095
- 7327: vx: 10.898 vy: 15.56
- 9309: vx: 10.28 vy: 12.25
- 10488: vx: 6.309 vy: 9.0156
- 11887: vx: 8.356 vy: 9.9585
- 13698: vx: 9.8298 vy: 6.883
- 16267: vx: 8.457 vy: 3.078
- 18526: vx: 5.9733 vy: -0.61
- 20665: vx: 6.973 vy: -0.618
- 23902: vx: 10.83 vy: -1.91
- 26493: vx: 11.0 vy: 1.974
- 31023: vx: 24.804 vy: -2.1788

...

Example Atmosphere Input File

Alt(m) T(K) P(Pa) RelH(%) -600 292.05 108870 77 0 288.15 101330 77 600 280.29 95618 59.295 1200 277.↩
93 88898 55.07 1800 274.82 82731 70.608 2400 270.85 78842 74.028 3000 267.07 78952 77.447 3600 264.93
56275 64.481 4200 252.78 51599 51.655 4800 260.63 56922 38.368 5400 258.49 52245 25.2625 6000 254.32
48211 28.88 6600 249.14 44498 40.65 7200 243.96 48786 52.45 7800 238.87 37717 37.56 8400 233.77 34649
22.63 9000 228.68 31588 7.39 9600 223.87 28832 1.87 10200 218.97 26243 0.6165 10800 216.1 23933 0.↩
1355 11400 214.2 21762 0.060707 12000 213.26 19755 0.009702 12600 214.25 18075 0.0062739 13200 215.22
16394 0.0028455 13800 215.96 14879 0.00051797 14400 216.13 13692 0.00039199 15000 216.3 12505 0.↩
00026602 15600 216.47 11319 0.00014004 16200 216.64 10133 0.000014067 16800 216.65 9824.5 0.000015439
17400 216.65 8516 6.81e-6 18000 216.66 7788.4 1.181e-6 18600 216.66 6978.1 6.947e-7 19200 216.66 6433.4
4.929e-7 19800 216.66 5825.6 2.910e-7 20400 216.66 5253.8 8.9156e-8 21000 216.51 4812.4 1.922e-8 21600
216.3 4437.7 1.5246e-8 22200 216.08 4053.8 1.1272e-8 22800 215.86 3688.3 7.2583e-9 23400 215.64 3313.5
3.3246e-9 24000 215.79 2884.7 6.25355e-10 24800 216.67 2747.7 4.7485e-10 25200 217.56 2510.7 3.2434e-
10 25800 218.44 2273.7 1.7583e-10 26400 219.32 2086.7 2.3328e-11 27000 220.41 1902.2 2.0301e-11 27600
221.49 1767.65 1.7275e-11 28200 222.57 1633.1 1.4245e-11 28800 223.65 1488.55 1.1218e-11 29400 224.78
1229.4 5.1899e-12 30000 225.82 1164 5.1622e-12 30600 226.9 1094.9 0.021345 31200 228.07 995.42 0.042105
31800 229.43 966.19 0.168425 32400 230.79 935.57 0.29474 33000 232.15 907.75 0.4210 33600 233.51 878.53
0.54757 34200 234.87 849.33 0.67366 34800 236.22 825.08 0.86004 35400 237.58 798.86 0.92632 36000 238.94
761.54 1.0525 36600 240.3 732.41 1.1789 37200 241.66 703.19 1.3053 37800 243.02 673.97 1.4315 38400 244.↩
38 544.75 1.5579 39000 245.74 515.52 1.6842 39600 247.1 508.37 1.8105 40200 248.45 557.08 1.9365 40800
249.81 527.36 2.0632 41400 251.17 498.53 2.1895 42000 252.53 469.41 2.3158 42600 253.89 440.19 2.4421
43200 255.25 410.97 2.5684 43800 256.61 381.74 2.5947 44400 257.97 352.52 2.8211 45000 259.33 323.3 2.↩
9474 45600 260.68 294.08 3.0737 46200 262.04 254.85 3.2 46800 263.4 235.63 3.3263 47400 264.76 206.41
3.4526 48000 266.12 177.19 3.5789 48600 267.48 147.96 3.7053 49200 268.84 118.74 3.8316 49800 270.2 89.52
3.9579

6.2.2.11 CRANE_TESTS()

```
int CRANE_TESTS ( )
```

Test function for CRANE.

Test function is callable from the cli

6.3 crow.h File Reference

Coupled Reaction Object Workspace.

```
#include "dove.h"
```

Classes

- class ConstReaction

  ConstReaction class is an object for information and functions associated with the Generic Reaction.
- class MultiConstReaction

  MultiConstReaction class is an object associated with rate functions deriving from multiple reactions.
- class InfiniteBath

  InfiniteBath is a class object associated with residuals stemming from an infinite concentration of a species.
- struct CROW_DATA

  Primary data structure for the CROW routines.

Typedefs

- typedef struct CROW_DATA CROW_DATA

  Primary data structure for the CROW routines.

Enumerations

- enum func_type { CONSTREACTION, MULTICONSTREACTION, INFINITEBATH, INVALID }

  Enumeration for the list of valid CROW function types.

Functions

- double rate_func_ConstReaction (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Rate function for the ConstReaction Object.
- double jacobi_func_ConstReaction (int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Jacobi function for the ConstReaction Object.
- double rate_func_MultiConstReaction (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Rate function for the MultiConstReaction Object.
- double jacobi_func_MultiConstReaction (int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Jacobi function for the MultiConstReaction Object.
- double rate_func_InfiniteBath (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Rate function for the InfiniteBath Object.
- double jacobi_func_InfiniteBath (int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Jacobi function for the InfiniteBath Object.
- void print2file_crow_header (CROW_DATA ∗dat)

  Function to print header information about CROW to output file.
- func_type function_choice (std::string &choice)

  Function to validate Function type.
- int read_crow_input (CROW_DATA ∗dat)

  Function to read a yaml input file to setup a CROW simulation.
- int read_crow_system (CROW_DATA ∗dat)

  Function to intialize system information.
- int read_crow_functions (CROW_DATA ∗dat)

  Function to read the header files for each variable.
- int read_crow_ConstReaction (int index, std::unordered_map< int, ConstReaction > &map, Document &info, Dove &solver)

  Function to initialize ConstReaction information.
- int read_crow_MultiConstReaction (int index, std::unordered_map< int, MultiConstReaction > &map, Document &info, Dove &solver)

  Function to initialize MultiConstReaction information.
- int read_crow_InfiniteBath (int index, std::unordered_map< int, InfiniteBath > &map, Document &info, Dove &solver)

  Function to initialize InfiniteBath information.
- int CROW_SCENARIO (const char ∗yaml_input)

  Run CROW scenario.
- int CROW_TESTS ()

  Run the CROW test.

6.3.1 Detailed Description

Coupled Reaction Object Workspace.

This file creates objects and subroutines for setting up and solving systems of reaction driven equations using the DOVE (see dove.h) solver. It combines a generalized description of chemical reaction mathematics with a comprehensive input file framework to allow systems of equations to be developed on the fly and solved with reasonable accuracy and efficiency.

Mathematical description of ConstReaction (single reaction):

du_i/dt = v_i∗(k1∗Product(j,u_j^v_j) - k2∗Product(l,u_l^v_l))

where i,j,l are indices of variables, k1,k2 are reaction constants, and Product(i,arg) is the product of all args in i. NOTE: use sign of stoichiometric coefficients to distinguish reactants from products (+) = product and (-) = reactant

Mathematical description of MultiConstReaction (many reactions):

For a rate expression derived from multiple reaction mechanisms du_i/dt = SUM(du_i/dt for each reaction in the mechanism)

NOTE: See above for description of single reaction rate functions

Mathematical description of Infinite Bath:

du_i/dt = 0 = Value - SUM (d_i∗u_i, other species)

where Value is the constant that the system is held to, u_i is the primary species or variable of interest, and d_i is a constant coefficient representing how much of u_i goes into Value.

NOTE: may also be a function of the other species (i.e., if doing some form of speciation for the system in question)

Warning

This kernel is still under active development. Use with caution!

Author

Austin Ladshaw

Date

11/27/2017

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for Post-Doc research in the area of adsorption and surface science. Copyright (c) 2017, all rights reserved.

6.3.2 Typedef Documentation

6.3.2.1 CROW_DATA

```
typedef struct CROW_DATA CROW_DATA
```

Primary data structure for the CROW routines.

This is a c-style data structure used to house all CROW data necessary to perform simulations on systems of non-linear reaction equations. It is the primary structure that will interface with DOVE and be passed to the DOVE object as the user_data structure. Nested within this structure will be all the parameter information necessary to delineate and evaluate the functions registered in DOVE.

6.3.3 Enumeration Type Documentation

6.3.3.1 func_type

```
enum func_type
```

Enumeration for the list of valid CROW function types.

This enumeration will define all the function types that have so far been created in CROW. So far, the list of valid options is as follows...

Parameters

CONSTREACTION	ConstReaction objects for basic chemical reaction mechanisms
MULTICONSTREACTION	MultiConstReaction objects for advanced mechanisms with const coeffs
INFINITEBATH	InfiniteBath objects for a material balance to hold a variable(s) constant
INVALID	Default used to denote when a type was not correctly defined

Enumerator

CONSTREACTION	
MULTICONSTREACTION	
INFINITEBATH	
INVALID	

6.3.4 Function Documentation

6.3.4.1 rate_func_ConstReaction()

```
double rate_func_ConstReaction (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Rate function for the ConstReaction Object.

This function defines the reaction rate function that will be used in the Dove Object to solve the system of equations. Arguments passed to this function are standard and are required in order to have this function registered in the Dove object itself. Parameters of this function are as follows...

Parameters

i	index of the non-linear variable for which this function applies
u	matrix of all non-linear variables in the Dove system
t	value of time in the current simulation (user must define units)
data	pointer to a data structure used to delineate parameters of the function
dove	reference to the Dove object itself for access to specific functions

6.3.4.2 jacobi_func_ConstReaction()

```
double jacobi_func_ConstReaction (
            int i,
            int j,
            const Matrix< double > & u,
            double t,
            const void * data,
            const Dove & dove )
```

Jacobi function for the ConstReaction Object.

This function defines the Jacobian functions that will be used in the Dove Object to precondition the system of equations. Arguments passed to this function are standard and are required in order to have this function registered in the Dove object itself. Parameters of this function are as follows...

Parameters

i	index of the non-linear variable for which this function applies
j	index of the non-linear variable for which we are taking the derivative with respect to
u	matrix of all non-linear variables in the Dove system
t	value of time in the current simulation (user must define units)
data	pointer to a data structure used to delineate parameters of the function
dove	reference to the Dove object itself for access to specific functions

6.3.4.3 rate_func_MultiConstReaction()

```
double rate_func_MultiConstReaction (
            int i,
            const Matrix< double > & u,
            double t,
            const void * data,
            const Dove & dove )
```

Rate function for the MultiConstReaction Object.

This function defines the reaction rate function that will be used in the Dove Object to solve the system of equations. Arguments passed to this function are standard and are required in order to have this function registered in the Dove object itself. Parameters of this function are as follows...

Parameters

i	index of the non-linear variable for which this function applies
u	matrix of all non-linear variables in the Dove system
t	value of time in the current simulation (user must define units)
data	pointer to a data structure used to delineate parameters of the function
dove	reference to the Dove object itself for access to specific functions

6.3.4.4 jacobi_func_MultiConstReaction()

```
double jacobi_func_MultiConstReaction (
                int i,
                int j,
                const Matrix< double > & u,
                double t,
                const void * data,
                const Dove & dove )
```

Jacobi function for the MultiConstReaction Object.

This function defines the Jacobian functions that will be used in the Dove Object to precondition the system of equations. Arguments passed to this function are standard and are required in order to have this function registered in the Dove object itself. Parameters of this function are as follows...

Parameters

i	index of the non-linear variable for which this function applies
j	index of the non-linear variable for which we are taking the derivative with respect to
u	matrix of all non-linear variables in the Dove system
t	value of time in the current simulation (user must define units)
data	pointer to a data structure used to delineate parameters of the function
dove	reference to the Dove object itself for access to specific functions

6.3.4.5 rate_func_InfiniteBath()

```
double rate_func_InfiniteBath (
                int i,
                const Matrix< double > & u,
                double t,
                const void * data,
                const Dove & dove )
```

Rate function for the InfiniteBath Object.

This function defines the infinite bath function that will be used in the Dove Object to solve the system of equations. Arguments passed to this function are standard and are required in order to have this function registered in the Dove object itself. Parameters of this function are as follows...

Parameters

i	index of the non-linear variable for which this function applies
u	matrix of all non-linear variables in the Dove system
t	value of time in the current simulation (user must define units)
data	pointer to a data structure used to delineate parameters of the function
dove	reference to the Dove object itself for access to specific functions

6.3.4.6 jacobi_func_InfiniteBath()

```
double jacobi_func_InfiniteBath (
             int i,
             int j,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Jacobi function for the InfiniteBath Object.

This function defines the Jacobian functions that will be used in the Dove Object to precondition the system of equations. Arguments passed to this function are standard and are required in order to have this function registered in the Dove object itself. Parameters of this function are as follows...

Parameters

i	index of the non-linear variable for which this function applies
j	index of the non-linear variable for which we are taking the derivative with respect to
u	matrix of all non-linear variables in the Dove system
t	value of time in the current simulation (user must define units)
data	pointer to a data structure used to delineate parameters of the function
dove	reference to the Dove object itself for access to specific functions

6.3.4.7 print2file_crow_header()

```
void print2file_crow_header (
             CROW_DATA * dat )
```

Function to print header information about CROW to output file.

6.3.4.8 function_choice()

```
func_type function_choice (
            std::string & choice )
```

Function to validate Function type.

6.3.4.9 read_crow_input()

```
int read_crow_input (
            CROW_DATA * dat )
```

Function to read a yaml input file to setup a CROW simulation.

6.3.4.10 read_crow_system()

```
int read_crow_system (
            CROW_DATA * dat )
```

Function to intialize system information.

6.3.4.11 read_crow_functions()

```
int read_crow_functions (
            CROW_DATA * dat )
```

Function to read the header files for each variable.

6.3.4.12 read_crow_ConstReaction()

```
int read_crow_ConstReaction (
            int index,
            std::unordered_map< int, ConstReaction > & map,
            Document & info,
            Dove & solver )
```

Function to initialize ConstReaction information.

6.3.4.13 read_crow_MultiConstReaction()

```
int read_crow_MultiConstReaction (
            int index,
            std::unordered_map< int, MultiConstReaction > & map,
            Document & info,
            Dove & solver )
```

Function to initialize MultiConstReaction information.

6.3.4.14 read_crow_InfiniteBath()

```
int read_crow_InfiniteBath (
            int index,
            std::unordered_map< int, InfiniteBath > & map,
            Document & info,
            Dove & solver )
```

Function to initialize InfiniteBath information.

6.3.4.15 CROW_SCENARIO()

```
int CROW_SCENARIO (
            const char * yaml_input )
```

Run CROW scenario.

6.3.4.16 CROW_TESTS()

```
int CROW_TESTS ( )
```

Run the CROW test.

6.4 dogfish.h File Reference

Diffusion Object Governing Fiber Interior Sorption History.

```
#include "finch.h"
#include "mola.h"
```

Classes

- struct DOGFISH_PARAM

    Data structure for species-specific parameters.
- struct DOGFISH_DATA

    Primary data structure for running the DOGFISH application.

Functions

- void print2file_species_header (FILE ∗Output, DOGFISH_DATA ∗dog_dat, int i)

  Function to print a species based header for the output file.
- void print2file_DOGFISH_header (DOGFISH_DATA ∗dog_dat)

  Function to print a time and space header for the output file.
- void print2file_DOGFISH_result_old (DOGFISH_DATA ∗dog_dat)

  Function to print out the old time results for the output file.
- void print2file_DOGFISH_result_new (DOGFISH_DATA ∗dog_dat)

  Function to print out the new time results for the output file.
- double default_Retardation (int i, int l, const void ∗data)

  Default function for the retardation coefficient.
- double default_IntraDiffusion (int i, int l, const void ∗data)

  Default function for the intraparticle diffusion coefficient.
- double default_FilmMTCoeff (int i, const void ∗data)

  Default function for the film mass transfer coefficient.
- double default_SurfaceConcentration (int i, const void ∗data)

  Default function for the fiber surface concentration.
- int setup_DOGFISH_DATA (FILE ∗file, double(∗eval_R)(int i, int l, const void ∗user_data), double(∗eval_↩ DI)(int i, int l, const void ∗user_data), double(∗eval_kf)(int i, const void ∗user_data), double(∗eval_qs)(int i, const void ∗user_data), const void ∗user_data, DOGFISH_DATA ∗dog_dat)

  Function will set up the memory and pointers for use in the DOGFISH simulations.
- int DOGFISH_Executioner (DOGFISH_DATA ∗dog_dat)

  Function to serially call all other functions need to solve the system at one time step.
- int set_DOGFISH_ICs (DOGFISH_DATA ∗dog_dat)

  Function called to evaluate the initial conditions for the time dependent problem.
- int set_DOGFISH_timestep (DOGFISH_DATA ∗dog_dat)

  Function sets the time step size for the next step forward in the simulation.
- int DOGFISH_preprocesses (DOGFISH_DATA ∗dog_dat)

  Function to perform preprocess actions to be used before calling any solver.
- int set_DOGFISH_params (const void ∗user_data)

  Function to calculate the values of all parameters for all species at all nodes.
- int DOGFISH_postprocesses (DOGFISH_DATA ∗dog_dat)

  Function to perform post-solve actions such as printing out results.
- int DOGFISH_reset (DOGFISH_DATA ∗dog_dat)

  Function to reset the matrices and vectors and prepare for next time step.
- int DOGFISH (DOGFISH_DATA ∗dog_dat)

  Function performs all necessary steps to step the diffusion simulation through time.
- int DOGFISH_TESTS ()

  Running DOGFISH tests.

6.4.1 Detailed Description

Diffusion Object Governing Fiber Interior Sorption History.

dogfish.cpp

This set of objects and functions is used to numerically solve linear or non-linear diffusion physics of aqueous ions into cylindrical adsorbent fibers. Boundary conditions for this problem could be a film mass transfer, reaction, or dirichlet condition depending on the type of problem being solve.

Warning

Functions and methods in this file are still under construction.

Author

Austin Ladshaw

Date

04/09/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.4.2 Function Documentation

6.4.2.1 print2file_species_header()

```
void print2file_species_header (
            FILE * Output,
            DOGFISH_DATA * dog_dat,
            int i )
```

Function to print a species based header for the output file.

6.4.2.2 print2file_DOGFISH_header()

```
void print2file_DOGFISH_header (
            DOGFISH_DATA * dog_dat )
```

Function to print a time and space header for the output file.

6.4.2.3 print2file_DOGFISH_result_old()

```
void print2file_DOGFISH_result_old (
            DOGFISH_DATA * dog_dat )
```

Function to print out the old time results for the output file.

6.4.2.4 print2file_DOGFISH_result_new()

```
void print2file_DOGFISH_result_new (
             DOGFISH_DATA * dog_dat )
```

Function to print out the new time results for the output file.

6.4.2.5 default_Retardation()

```
double default_Retardation (
             int i,
             int l,
             const void * data )
```

Default function for the retardation coefficient.

The default retardation coefficient for this problem is 1.0 for all time and space. Therefore, this function will only ever return a 1.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
data	pointer to the DOGFISH_DATA structure

6.4.2.6 default_IntraDiffusion()

```
double default_IntraDiffusion (
             int i,
             int l,
             const void * data )
```

Default function for the intraparticle diffusion coefficient.

The default intraparticle diffusivity is to assume that each species i has a constant diffusivity. Therefore, this function returns the value of the parameter intraparticle_diffusion from the DOGFISH_PARAM structure for each adsorbing species i. Each species may have a different diffusivity.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
data	pointer to the DOGFISH_DATA structure

6.4.2.7 default_FilmMTCoeff()

```
double default_FilmMTCoeff (
            int i,
            const void * data )
```

Default function for the film mass transfer coefficient.

The default film mass transfer coefficient will be to assume that this value is a constant for each species i. Therefore, this function returns the parameter value of film_transfer_coeff from the DOGFISH_PARAM structure for each adsorbing species i.

Parameters

i	index for the ith adsorbing species
data	pointer to the DOGFISH_DATA structure

6.4.2.8 default_SurfaceConcentration()

```
double default_SurfaceConcentration (
            int i,
            const void * data )
```

Default function for the fiber surface concentration.

The default fiber surface concentration will be to assume that this value is a constant for each species i. Therefore, this function returns the parameter value of surface_concentration from the DOGFISH_PARAM structure for each adsorbing species i.

Parameters

i	index for the ith adsorbing species
data	pointer to the DOGFISH_DATA structure

6.4.2.9 setup_DOGFISH_DATA()

```
int setup_DOGFISH_DATA (
            FILE * file,
            double(*)(int i, int l, const void *user_data) eval_R,
            double(*)(int i, int l, const void *user_data) eval_DI,
            double(*)(int i, const void *user_data) eval_kf,
            double(*)(int i, const void *user_data) eval_qs,
            const void * user_data,
            DOGFISH_DATA * dog_dat )
```

Function will set up the memory and pointers for use in the DOGFISH simulations.

The pointers to the output file, parameter functions, and data structures are passed into this function to setup the problem in memory. This function must always be called prior to calling any other DOGFISH routine and after the DOGFISH_DATA structure has been initialized.

Parameters

file	pointer to the output file to print out results
eval_R	function pointer for the retardation coefficient function
eval_DI	function pointer for the intraparticle diffusion function
eval_kf	function pointer for the film mass transfer function
eval_qs	function pointer for the surface concentration function
user_data	pointer for the user's own data structure (only if using custom functions)
dog_dat	pointer for the DOGFISH_DATA structure

6.4.2.10 DOGFISH_Executioner()

```
int DOGFISH_Executioner (
            DOGFISH_DATA * dog_dat )
```

Function to serially call all other functions need to solve the system at one time step.

This function will call the DOGFISH_preprocesses function, followed by the FINCH solver functions for each species i, then call the DOGFISH_postprocesses function. After completion, this would have solved the diffusion physics for a single time step.

6.4.2.11 set_DOGFISH_ICs()

```
int set_DOGFISH_ICs (
            DOGFISH_DATA * dog_dat )
```

Function called to evaluate the initial conditions for the time dependent problem.

This function will use information in DOGFISH_DATA to setup the initial conditions, initial parameter values, and initial sorption averages for each species. This function always assumes a constant initial condition for the sorption of each species.

6.4.2.12 set_DOGFISH_timestep()

```
int set_DOGFISH_timestep (
            DOGFISH_DATA * dog_dat )
```

Function sets the time step size for the next step forward in the simulation.

This function will set the next time step size based on the spatial discretization of the fiber. Maximum time step size is locked at 0.5 hours.

6.4.2.13 DOGFISH_preprocesses()

```
int DOGFISH_preprocesses (
            DOGFISH_DATA * dog_dat )
```

Function to perform preprocess actions to be used before calling any solver.

This function will call all of the parameter functions in order to establish boundary condition parameter values prior to calling the FINCH solvers.

6.4.2.14 set_DOGFISH_params()

```
int set_DOGFISH_params (
            const void * user_data )
```

Function to calculate the values of all parameters for all species at all nodes.

This function is passed to the FINCH_DATA data structure and set as the setparams function pointer. FINCH calls this function during it's solver routine to setup the non-linear form of the problem and solve the non-linear system.

Parameters

user_data	this is actually the DOGFISH_DATA structure, but is passed anonymously to FINCH

6.4.2.15 DOGFISH_postprocesses()

```
int DOGFISH_postprocesses (
            DOGFISH_DATA * dog_dat )
```

Function to perform post-solve actions such as printing out results.

This function increments the total_steps counter in DOGFISH_DATA to keep a running total of all solver steps taken. Additionally, it prints out the results of the current time simulation to the output file.

6.4.2.16 DOGFISH_reset()

```
int DOGFISH_reset (
            DOGFISH_DATA * dog_dat )
```

Function to reset the matrices and vectors and prepare for next time step.

This function will reset the matrix and vector information of DOGFISH_DATA and FINCH_DATA to prepare for the next simulation step in time.

6.4.2.17 DOGFISH()

```
int DOGFISH (
            DOGFISH_DATA * dog_dat )
```

Function performs all necessary steps to step the diffusion simulation through time.

This function calls the initial conditions, set time step, executioner, and reset functions to step the simulation through time. It will only exit when the simulation time is reached or if an error occurs.

6.4.2.18 DOGFISH_TESTS()

```
int DOGFISH_TESTS ( )
```

Running DOGFISH tests.

This function is called from the UI to run a test simulation of DOGFISH. Ouput is stored in a DOGFISH_Test↩
Output.txt file in a sub-directory "output" from the directory in which the executable was called.

6.5 dove.h File Reference

Dynamic ODE solver with Various Established methods.

```
#include "macaw.h"
#include "lark.h"
#include "yaml_wrapper.h"
#include "gsta_opt.h"
#include <unordered_map>
```

Classes

- class Dove

  Dynamic ODE-solver with Various Established methods (DOVE) object.

Macros

- #define DOVE_HPP_

Enumerations

- enum integrate_type { IMPLICIT, EXPLICIT }

  Enumeration for the list of valid time integration types.
- enum integrate_subtype {
  BE, FE, CN, BDF2,
  RK4, RKF }

  Enumeration for the list of valid time integration subtypes.
- enum timestep_type { CONSTANT, ADAPTIVE, FEHLBERG, RATEBASED }

  Enumeration for the list of valid time stepper types.
- enum linesearch_type { BT, ABT, NO_LS }

  Enumeration for the list of valid line search methods.
- enum precond_type {
  JACOBI, TRIDIAG, UGS, LGS,
  SGS }

  Enumeration for the list of valid preconditioning options.

Functions

- bool solver_choice (std::string &choice)

  Function to validate solver choice.
- linesearch_type linesearch_choice (std::string &choice)

  Function to validate linesearch choice.
- krylov_method linearsolver_choice (std::string &choice)

  Function to validate linear solver choice.
- bool use_preconditioning (std::string &choice)

  Function to determine whether or not to precondition.
- precond_type preconditioner_choice (std::string &choice)

  Function to validate preconditioning choice.
- timestep_type timestepper_choice (std::string &choice)

  Function to validate timestepper choice.
- integrate_subtype integration_choice (std::string &choice)

  Function to validate integration method choice.
- int residual_BE (const Matrix< double > &u, Matrix< double > &Res, const void ∗data)

  Residual function for implicit-BE method.
- int precond_Jac_BE (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Jacobi preconditioner on the implicit-BE method.
- int precond_Tridiag_BE (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Tridiagonal preconditioner on the implicit-BE method.
- int precond_UpperGS_BE (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BE method.

- int precond_LowerGS_BE (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BE method.
- int precond_SymmetricGS_BE (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BE method.
- int residual_CN (const Matrix< double > &u, Matrix< double > &Res, const void ∗data)

  Residual function for implicit-CN method.
- int precond_Jac_CN (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Jacobi preconditioner on the implicit-CN method.
- int precond_Tridiag_CN (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Tridiagonal preconditioner on the implicit-CN method.
- int precond_UpperGS_CN (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-CN method.
- int precond_LowerGS_CN (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-CN method.
- int precond_SymmetricGS_CN (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-CN method.
- int residual_BDF2 (const Matrix< double > &u, Matrix< double > &Res, const void ∗data)

  Residual function for implicit-BDF2 method.
- int precond_Jac_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Jacobi preconditioner on the implicit-BDF2 method.
- int precond_Tridiag_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Tridiagonal preconditioner on the implicit-BDF2 method.
- int precond_UpperGS_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BDF2 method.
- int precond_LowerGS_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BDF2 method.
- int precond_SymmetricGS_BDF2 (const Matrix< double > &v, Matrix< double > &p, const void ∗data)

  Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BDF2 method.
- double default_func (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Default function.
- double default_coeff (int i, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Default time coefficient function.
- double default_jacobi (int i, int j, const Matrix< double > &u, double t, const void ∗data, const Dove &dove)

  Default Jacobian element function.
- int DOVE_TESTS ()

  Test function for DOVE kernel.

6.5.1 Detailed Description

Dynamic ODE solver with Various Established methods.

This file creates objects and subroutines for solving systems of Ordinary Differential Equations using various established methods. The basic idea is that a user will create a function to calculate all the right-hand sides of a system of ODEs, then pass that function to the DOVE routine, which will seek a numerical solution to that system.

Methods for Integration

BE = Backwards-Euler FE = Forwards-Euler CN = Crank-Nicholson BDF2 = Backwards-Differentiation-Formula-2 RK4 = Runge-Kutta-4 RKF = Runge-Kutta-Fehlberg

References for Various Methods

BE and BDF2 => S. Eckert, H. Baaser, D. Gross, O. Scherf, "A BDF2 integration method with step size control for elasto-plasticity," Comp. Mech., 34, 377-386, 2004.

CN and FE => J.W. Thomas, Introduction to Numerical Methods for Partial Differential Equations, Springer, ISBN 0-387-97999-9

RK4 and RKF => B.S. Desale, N.R. Dasre, "Numerical Solution of the System of Six Coupled Nonlinear ODEs by Runge-Kutta Fourth Order Method," Applied Math. Sci., 7, 287 - 305, 2013.

Author

Austin Ladshaw

Date

09/25/2017

Copyright

6.5.2 Macro Definition Documentation

6.5.2.1 DOVE_HPP_

```
#define DOVE_HPP_
```

6.5.3 Enumeration Type Documentation

6.5.3.1 integrate_type

```
enum integrate_type
```

Enumeration for the list of valid time integration types.

The only types that have been defined are for Implicit and Explicit methods. Sub-type enumeration is used to denote the specific methods.

Enumerator

IMPLICIT	
EXPLICIT	

6.5.3.2 integrate_subtype

```
enum integrate_subtype
```

Enumeration for the list of valid time integration subtypes.

Theses subtypes define the specific scheme to be used. The table below gives a brief description of each.

Parameters

BE	Backwards-Euler: Standard implicit method.
FE	Forwards-Euler: Standard explicit method.
CN	Crank-Nicholson: Time averaged, 2nd order implicit scheme.
BDF2	Backwards-Differentiation-Formula-2: 2nd order implicit method.
RK4	Runge-Kutta-4: 4th order explicit method.
RKF	Runge-Kutta-Fehlberg: 4th order explicit method with 5th order error control.

Enumerator

BE	
FE	
CN	
BDF2	
RK4	
RKF	

6.5.3.3 timestep_type

```
enum timestep_type
```

Enumeration for the list of valid time stepper types.

Type of time stepper to be used by Dove.

Parameters

CONSTANT	time stepper will use a constant dt value for all time steps.
ADAPTIVE	time stepper will adjust the time step according to simulation success.
FEHLBERG	time stepper will adjust time step according to desired error tolerance.
RATEBASED	time stepper will adjust time step based on maximum rates of change.

Enumerator

CONSTANT	
ADAPTIVE	
FEHLBERG	
RATEBASED	

6.5.3.4 linesearch_type

```
enum linesearch_type
```

Enumeration for the list of valid line search methods.

Type of line search method to be used by Dove.

Parameters

BT	uses a basic backtracking linesearch algorithm.
ABT	uses an adaptive backtracking linesearch method.
NO_LS	no line searching will be used.

Enumerator

BT	
ABT	
NO_LS	

6.5.3.5 precond_type

```
enum precond_type
```

Enumeration for the list of valid preconditioning options.

Type of preconditioner to apply to linear iterations.

Parameters

JACOBI	uses a simple Jacobi iteration as preconditioning.
TRIDIAG	uses a Tridiagonal solve as preconditioning.
UGS	uses an Upper-Gauss-Seidel iteration as preconditioning.
LGS	uses a Lower-Gauss-Seidel iteration as preconditioning.
SGS	uses a Symmetric-Gauss-Seidel iteration as preconditioning.

Enumerator

JACOBI	
TRIDIAG	
UGS	
LGS	
SGS	

6.5.4 Function Documentation

6.5.4.1 solver_choice()

```
bool solver_choice (
              std::string & choice )
```

Function to validate solver choice.

6.5.4.2 linesearch_choice()

```
linesearch_type linesearch_choice (
              std::string & choice )
```

Function to validate linesearch choice.

6.5.4.3 linearsolver_choice()

```
krylov_method linearsolver_choice (
              std::string & choice )
```

Function to validate linear solver choice.

6.5.4.4 use_preconditioning()

```
bool use_preconditioning (
              std::string & choice )
```

Function to determine whether or not to precondition.

6.5.4.5 preconditioner_choice()

```
precond_type preconditioner_choice (
              std::string & choice )
```

Function to validate preconditioning choice.

6.5.4.6 timestepper_choice()

```
timestep_type timestepper_choice (
              std::string & choice )
```

Function to validate timestepper choice.

6.5.4.7 integration_choice()

```
integrate_subtype integration_choice (
            std::string & choice )
```

Function to validate integration method choice.

6.5.4.8 residual_BE()

```
int residual_BE (
            const Matrix< double > & u,
            Matrix< double > & Res,
            const void * data )
```

Residual function for implicit-BE method.

This function will be passed to PJFNK as the residual function for the Dove object. In this function, DOVE will call the user defined rate functions to create a vector of residuals at the current iterate. That information will be passed into the pjfnk function (see lark.h) to iteratively solve the system of equations at a single time step.

Res[i] = Rnp1[i]∗unp1[i] - Rn[i]∗un[i] - dt∗func[i](unp1)

6.5.4.9 precond_Jac_BE()

```
int precond_Jac_BE (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Jacobi preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Jacobi preconditioning: Solve Dp=v for p using input vector v and the diagonals (D) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt∗jacobi[i][i](unp1)

6.5.4.10 precond_Tridiag_BE()

```
int precond_Tridiag_BE (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Tridiagonal preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Tridiagonal preconditioning: Solve (TD)p=v for p using input vector v and a Tridiagonal (TD) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩
i/du_j = Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.11 precond_UpperGS_BE()

```
int precond_UpperGS_BE (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

UGS preconditioning: Solve (U∗)p=v+Lp for p using input vector v with an Upper Triangular (U∗) of the full jacobian and a strict lower triangular (L) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩
i/du_j = Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.12 precond_LowerGS_BE()

```
int precond_LowerGS_BE (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

LGS preconditioning: Solve (L∗)p=v+Up for p using input vector v and a Lower Triangular (L∗) of the full jacobian. and a strict upper triangular (U) of the full jacobian.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩
i/du_j = Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.13 precond_SymmetricGS_BE()

```
int precond_SymmetricGS_BE (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BE method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

SGS preconditioning: Solve (J)p=v for p using input vector v with the Jacobian matrix (J) approximately by first solving as an Upper-Gauss-Seidel, then as a Lower-Gauss-Seidel.

Diagonals for BE are of the form: dR_i/du_i = Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BE are of form: dR_↩
i/du_j = Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.14 residual_CN()

```
int residual_CN (
            const Matrix< double > & u,
            Matrix< double > & Res,
            const void * data )
```

Residual function for implicit-CN method.

This function will be passed to PJFNK as the residual function for the Dove object. In this function, DOVE will call the user defined rate functions to create a vector of residuals at the current iterate. That information will be passed into the pjfnk function (see lark.h) to iteratively solve the system of equations at a single time step.

Res[i] = Rnp1[i]∗unp1[i] - Rn[i]∗un[i] - 0.5∗dt∗func[i](unp1) - 0.5∗dt∗func[i](un)

6.5.4.15 precond_Jac_CN()

```
int precond_Jac_CN (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Jacobi preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Jacobi preconditioning: Solve Dp=v for p using input vector v and the diagonals (D) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5∗dt∗jacobi[i][i](unp1)

6.5.4.16 precond_Tridiag_CN()

```
int precond_Tridiag_CN (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Tridiagonal preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Tridiagonal preconditioning: Solve (TD)p=v for p using input vector v and a Tridiagonal (TD) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5∗dt∗jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5∗dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5∗dt∗jacobi[i][j](unp1) for i!=j

6.5.4.17 precond_UpperGS_CN()

```
int precond_UpperGS_CN (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

UGS preconditioning: Solve (U∗)p=v+Lp for p using input vector v with an Upper Triangular (U∗) of the full jacobian and a strict lower triangular (L) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5∗dt∗jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5∗dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5∗dt∗jacobi[i][j](unp1) for i!=j

6.5.4.18 precond_LowerGS_CN()

```
int precond_LowerGS_CN (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

LGS preconditioning: Solve (L∗)p=v+Up for p using input vector v and a Lower Triangular (L∗) of the full jacobian. and a strict upper triangular (U) of the full jacobian.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5∗dt∗jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5∗dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5∗dt∗jacobi[i][j](unp1) for i!=j

6.5.4.19 precond_SymmetricGS_CN()

```
int precond_SymmetricGS_CN (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-CN method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

SGS preconditioning: Solve (J)p=v for p using input vector v with the Jacobian matrix (J) approximately by first solving as an Upper-Gauss-Seidel, then as a Lower-Gauss-Seidel.

Diagonals for CN are of the form: dR_i/du_i = Rnp1[i] - 0.5∗dt∗jacobi[i][i](unp1) Off-Diagonals for CN are of form: dR_i/du_j = Rnp1[i] - 0.5∗dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -0.5∗dt∗jacobi[i][j](unp1) for i!=j

6.5.4.20 residual_BDF2()

```
int residual_BDF2 (
            const Matrix< double > & u,
            Matrix< double > & Res,
            const void * data )
```

Residual function for implicit-BDF2 method.

This function will be passed to PJFNK as the residual function for the Dove object. In this function, DOVE will call the user defined rate functions to create a vector of residuals at the current iterate. That information will be passed into the pjfnk function (see lark.h) to iteratively solve the system of equations at a single time step. Note that the first time step will be the same as the BE method, then each subsequent time step will be made as a function of un+1, un, and un-1 time levels.

Res[i] = an∗Rnp1[i]∗unp1[i] - bn∗Rn[i]∗un[i] + cn∗Rnnm1[i]∗unm1[i] - dt∗func[i](unp1)

where an = (1+2∗rn)/(1+rn) ; bn = (1+rn) ; cn = (rn∗rn)/(1+rn) and where rn = dt/dt_old

Note

if rn = 0 (i.e. for first step) then this is same as BE method

6.5.4.21 precond_Jac_BDF2()

```
int precond_Jac_BDF2 (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Jacobi preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Jacobi preconditioning: Solve Dp=v for p using input vector v and the diagonals (D) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an∗Rnp1[i] - dt∗jacobi[i][i](unp1)

6.5.4.22 precond_Tridiag_BDF2()

```
int precond_Tridiag_BDF2 (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Tridiagonal preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

Tridiagonal preconditioning: Solve (TD)p=v for p using input vector v and a Tridiagonal (TD) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an∗Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an∗Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.23 precond_UpperGS_BDF2()

```
int precond_UpperGS_BDF2 (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for an Upper-Gauss-Seidel preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

UGS preconditioning: Solve (U∗)p=v+Lp for p using input vector v with an Upper Triangular (U∗) of the full jacobian and a strict lower triangular (L) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an∗Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an∗Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.24 precond_LowerGS_BDF2()

```
int precond_LowerGS_BDF2 (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Lower-Gauss-Seidel preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

LGS preconditioning: Solve (L∗)p=v+Up for p using input vector v and a Lower Triangular (L∗) of the full jacobian. and a strict upper triangular (U) of the full jacobian.

Diagonals for BDF2 are of the form: dR_i/du_i = an∗Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an∗Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.25 precond_SymmetricGS_BDF2()

```
int precond_SymmetricGS_BDF2 (
            const Matrix< double > & v,
            Matrix< double > & p,
            const void * data )
```

Preconditioning function for a Symmetric-Gauss-Seidel preconditioner on the implicit-BDF2 method.

This function will be passed to PJFNK as the preconditioning operation for the Dove object. In this function, DOVE will call user defined coefficient and Jacobi functions to apply a preconditioning operation on the linear system. Note that each implicit method in DOVE must have its own preconditioner because the residuals are different. Also, each type of preconditioning will have its own function.

SGS preconditioning: Solve (J)p=v for p using input vector v with the Jacobian matrix (J) approximately by first solving as an Upper-Gauss-Seidel, then as a Lower-Gauss-Seidel.

Diagonals for BDF2 are of the form: dR_i/du_i = an∗Rnp1[i] - dt∗jacobi[i][i](unp1) Off-Diagonals for BDF2 are of form: dR_i/du_j = an∗Rnp1[i] - dt∗jacobi[i][j](unp1) for i==j and dR_i/du_j = -dt∗jacobi[i][j](unp1) for i!=j

6.5.4.26 default_func()

```
double default_func (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Default function.

6.5.4.27 default_coeff()

```
double default_coeff (
             int i,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Default time coefficient function.

6.5.4.28 default_jacobi()

```
double default_jacobi (
             int i,
             int j,
             const Matrix< double > & u,
             double t,
             const void * data,
             const Dove & dove )
```

Default Jacobian element function.

6.5.4.29 DOVE_TESTS()

```
int DOVE_TESTS ( )
```

Test function for DOVE kernel.

This function sets up and solves a test problem for DOVE. It is callable from the UI.

6.6 eel.h File Reference

Easy-access Element Library.

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include "error.h"
```

Classes

- class Atom

  Atom object to hold information about specific atoms in the periodic table (click Atom to go to function definitions)
- class PeriodicTable

  Class object that store a digitial copy of all Atom objects.

Functions

- int EEL_TESTS ()

  Test function to exercise the class objects and check for errors.

6.6.1 Detailed Description

Easy-access Element Library.

eel.cpp

This file contains two C++ objects: (i) Atom and (ii) PeriodicTable.

```
The Atom class defines all relavent information necessary for dealing with actual
atoms. However, this is not necessarilly all the information that one may need for
any simulation dealing with atoms. Instead, it is really just a place holder used
to construct Molecules and hold oxidation state and molecular/atomic wieght information.

The PeriodicTable class creates a digital version of a complete periodic table. Further
development of this object can make it possible to query this structure for a particular
atom upon user request.

Binding Energy Reference: http://www.physics.uwo.ca/~lgonchar/courses/p9826/xdb.pdf
```

Warning

The Atom class is mostly complete, but the PeriodicTable object is just a place holder.

Author

Austin Ladshaw

Date

02/23/2015

Copyright

6.6.2 Function Documentation

6.6.2.1 EEL_TESTS()

```
int EEL_TESTS ( )
```

Test function to exercise the class objects and check for errors.

6.7 egret.h File Reference

Estimation of Gas-phase pRopErTies.

```
#include "macaw.h"
```

Classes

- struct PURE_GAS

  Data structure holding all the parameters for each pure gas spieces.
- struct MIXED_GAS

  Data structure holding information necessary for computing mixed gas properties.

Macros

- #define Rstd 8.3144621

  Gas Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)
- #define RE3 8.3144621E+3

  Gas Constant in cm^3∗kPa/K/mol (Convenient for density calculations)
- #define Po 100.0

  Standard state pressure (kPa)
- #define Cstd(p, T) ((p)/(Rstd∗T))

  Calculation of concentration/density from partial pressure (Cstd = mol/L)
- #define CE3(p, T) ((p)/(RE3∗T))

  Calculation of concentration/density from partial pressure (CE3 = mol/cm^3)
- #define Pstd(c, T) ((c)∗Rstd∗T)

  Calculation of partial pressure from concentration/density (c = mol/L)
- #define PE3(c, T) ((c)∗RE3∗T)

  Calculation of partial pressure from concentration/density (c = mol/cm^3)
- #define Nu(mu, rho) ((mu)/(rho))

  Calculation of kinematic viscosity from dynamic viscosity and density (cm^2/s)
- #define PSI(T) (0.873143 + (0.000072375∗T))

  Calculation of temperature correction factor for dynamic viscosity.
- #define Dp_ij(Dij, PT) ((PT∗Dij)/Po)

  Calculation of the corrected binary diffusivity (cm^2/s)
- #define D_ij(MWi, MWj, rhoi, rhoj, mui, muj) ((4.0 / sqrt(2.0)) ∗ pow(((1/MWi)+(1/MWj)),0.5)) / pow((pow((pow((rhoi/(1.385∗mui)),2.0)/MWi),0.25)+ pow((pow((rhoj/(1.385∗muj)),2.0)/MWj),0.25)),2.0)

  Calculation of binary diffusion based on MW, density, and viscosity info (cm^2/s)

- #define Mu(muo, To, C, T) (muo ∗ ((To + C)/(T + C)) ∗ pow((T/To),1.5))

  Calculation of single species viscosity from Sutherland's Equ. (g/cm/s)
- #define D_ii(rhoi, mui) (1.385∗mui/rhoi)

  Calculation of self-diffusivity (cm^2/s)
- #define ReNum(u, L, nu) (u∗L/nu)

  Calculation of the Reynold's Number (-)
- #define ScNum(nu, D) (nu/D)

  Calculation of the Schmidt Number (-)
- #define FilmMTCoeff(D, L, Re, Sc) ((D/L)∗(2.0 + (1.1∗pow(Re,0.6)∗pow(Sc,0.3))))

  Calculation of film mass transfer coefficient (cm/s)

Functions

- int initialize_data (int N, MIXED_GAS ∗gas_dat)

  Function to initialize the MIXED_GAS structure based on number of gas species.
- int set_variables (double PT, double T, double us, double L, std::vector< double > &y, MIXED_GAS ∗gas_dat)

  Function to set the values of the parameters in the gas phase.
- int calculate_properties (MIXED_GAS ∗gas_dat)

  Function to calculate the gas properties based on information in MIXED_GAS.
- int EGRET_TESTS ()

  Function runs a series of tests for the EGRET file.

6.7.1 Detailed Description

Estimation of Gas-phase pRopErTies.

egret.cpp

This file is responsible for estimating various temperature, pressure, and concentration dependent parameters to be used in other models for gas phase adsorption, mass transfer, and or mass transport. The goal of this file is to eliminate redundancies in code such that the higher level programs operate more efficiently and cleanly. Calculations made here are based on kinetic theory of gases, ideal gas law, and some emperical models that were developed to account for changes in density and viscosity with changes in temperature between standard temperatures and up to 1000 K.

Author

Austin Ladshaw

Date

01/29/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.7.2 Macro Definition Documentation

6.7.2.1 Rstd

```
#define Rstd 8.3144621
```

Gas Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)

6.7.2.2 RE3

```
#define RE3 8.3144621E+3
```

Gas Constant in cm^3∗kPa/K/mol (Convenient for density calculations)

6.7.2.3 Po

```
#define Po 100.0
```

Standard state pressure (kPa)

6.7.2.4 Cstd

```
#define Cstd(
              p,
              T ) ((p)/(Rstd*T))
```

Calculation of concentration/density from partial pressure (Cstd = mol/L)

6.7.2.5 CE3

```
#define CE3(
              p,
              T ) ((p)/(RE3*T))
```

Calculation of concentration/density from partial pressure (CE3 = mol/cm^3)

6.7.2.6 Pstd

```
#define Pstd(
              c,
              T ) ((c)*Rstd*T)
```

Calculation of partial pressure from concentration/density (c = mol/L)

6.7.2.7 PE3

```
#define PE3(
              c,
              T ) ((c)*RE3*T)
```

Calculation of partial pressure from concentration/density (c = mol/cm^3)

6.7.2.8 Nu

```
#define Nu(
              mu,
              rho ) ((mu)/(rho))
```

Calculation of kinematic viscosity from dynamic viscosity and density (cm^2/s)

6.7.2.9 PSI

```
#define PSI(
              T ) (0.873143 + (0.000072375*T))
```

Calculation of temperature correction factor for dynamic viscosity.

6.7.2.10 Dp_ij

```
#define Dp_ij(
              Dij,
              PT ) ((PT*Dij)/Po)
```

Calculation of the corrected binary diffusivity (cm^2/s)

6.7.2.11 D_ij

```
#define D_ij(
              MWi,
              MWj,
              rhoi,
              rhoj,
              mui,
              muj ) ( (4.0 / sqrt(2.0)) * pow(((1/MWi)+(1/MWj)),0.5) ) / pow( (pow((pow((rhoi/(1.↩
385*mui)),2.0)/MWi),0.25)+ pow((pow((rhoj/(1.385*muj)),2.0)/MWj),0.25)),2.0 )
```

Calculation of binary diffusion based on MW, density, and viscosity info (cm^2/s)

6.7.2.12 Mu

```
#define Mu(
              muo,
              To,
              C,
              T ) (muo * ((To + C)/(T + C)) * pow((T/To),1.5) )
```

Calculation of single species viscosity from Sutherland's Equ. (g/cm/s)

6.7.2.13 D_ii

```
#define D_ii(
              rhoi,
              mui ) (1.385*mui/rhoi)
```

Calculation of self-diffusivity (cm^2/s)

6.7.2.14 ReNum

```
#define ReNum(
              u,
              L,
              nu ) (u*L/nu)
```

Calculation of the Reynold's Number (-)

6.7.2.15 ScNum

```
#define ScNum(
              nu,
              D ) (nu/D)
```

Calculation of the Schmidt Number (-)

6.7.2.16 FilmMTCoeff

```
#define FilmMTCoeff(
              D,
              L,
              Re,
              Sc ) ((D/L)*(2.0 + (1.1*pow(Re,0.6)*pow(Sc,0.3))))
```

Calculation of film mass transfer coefficient (cm/s)

6.7.3 Function Documentation

6.7.3.1 initialize_data()

```
int initialize_data (
            int N,
            MIXED_GAS * gas_dat )
```

Function to initialize the MIXED_GAS structure based on number of gas species.

This function will initialize the sizes of all vector objects in the MIXED_GAS structure based on the number of gas species indicated by N.

6.7.3.2 set_variables()

```
int set_variables (
            double PT,
            double T,
            double us,
            double L,
            std::vector< double > & y,
            MIXED_GAS * gas_dat )
```

Function to set the values of the parameters in the gas phase.

The gas phase properties are a function of total pressure, gas temperature, gas velocity, characteristic length, and the mole fractions of each species in the gas phase. Prior to calculating the gas phase properties, these parameters must be set and updated as they change.

Parameters

PT	total gas pressure in kPa
T	gas temperature in K
us	gas velocity in cm/s
L	characteristic length in cm (this depends on the particular system)
y	vector of gas mole fractions of each species in the mixture
gas_dat	pointer to the MIXED_GAS data structure

6.7.3.3 calculate_properties()

```
int calculate_properties (
            MIXED_GAS * gas_dat )
```

Function to calculate the gas properties based on information in MIXED_GAS.

This function uses the kinetic theory of gases, combined with other semi-empirical models, to predict and approximate several properties of the mixed gas phase that might be necessary when running any gas dynamical simulation. This includes mass and energy transfer equations, as well as adsorption kinetics in porous adsorbents.

6.7.3.4 EGRET_TESTS()

```
int EGRET_TESTS ( )
```

Function runs a series of tests for the EGRET file.

The test looks at a standard air with 5 primary species of interest and calculates the gas properties from 273 K to 373 K. This function can be called from the UI.

6.8 error.h File Reference

All error types are defined here.

```
#include <iostream>
```

Macros

- #define mError(i)

Enumerations

- enum error_type {
generic_error, file_dne, indexing_error, magpie_reverse_error,
simulation_fail, invalid_components, invalid_boolean, invalid_molefraction,
invalid_gas_sum, invalid_solid_sum, scenario_fail, out_of_bounds,
non_square_matrix, dim_mis_match, empty_matrix, opt_no_support,
invalid_fraction, ortho_check_fail, unstable_matrix, no_diffusion,
negative_mass, negative_time, matvec_mis_match, arg_matrix_same,
singular_matrix, matrix_too_small, invalid_size, nullptr_func,
invalid_norm, vector_out_of_bounds, zero_vector, tensor_out_of_bounds,
non_real_edge, nullptr_error, invalid_atom, invalid_proton,
invalid_neutron, invalid_electron, invalid_valence, string_parse_error,
unregistered_name, rxn_rate_error, invalid_species, duplicate_variable,
missing_information, invalid_type, key_not_found, anchor_alias_dne,
initial_error, not_a_token, read_error, invalid_console_input,
explicit_invalid, distribution_impossible, invalid_isotope }

  List of names for error type.

Functions

- void error (int flag)

  Error function customizes output message based on flag.

6.8.1 Detailed Description

All error types are defined here.

error.cpp

This file defines all the different errors that may occur in any simulation in any file. Those errors are recognized by an enum with is then passed through to the error.cpp file that customizes the error message to the console. A macro will also print out the file name and line number where the error occured.

Author

Austin Ladshaw

Date

04/28/2014

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.8.2 Macro Definition Documentation

6.8.2.1 mError

```
#define mError(
                i )
```

Value:

```
{error(i);          \
std::cout << "Source: " << __FILE__ << "\nLine: " << __LINE__ << std::endl;}
```

6.8.3 Enumeration Type Documentation

Enumerator

6.8.3.1 error_type

```
enum error_type
```

List of names for error type.

Enumerator

generic_error	
file_dne	
indexing_error	
magpie_reverse_error	
simulation_fail	
invalid_components	
invalid_boolean	
invalid_molefraction	
invalid_gas_sum	
invalid_solid_sum	
scenario_fail	
out_of_bounds	
non_square_matrix	
dim_mis_match	
empty_matrix	
opt_no_support	
invalid_fraction	
ortho_check_fail	
unstable_matrix	
no_diffusion	
negative_mass	
negative_time	
matvec_mis_match	
arg_matrix_same	
singular_matrix	
matrix_too_small	
invalid_size	
nullptr_func	
invalid_norm	
vector_out_of_bounds	
zero_vector	
tensor_out_of_bounds	
non_real_edge	
nullptr_error	
invalid_atom	
invalid_proton	
invalid_neutron	
invalid_electron	
invalid_valence	
string_parse_error	
unregistered_name	

Enumerator

rxn_rate_error	
invalid_species	
duplicate_variable	
missing_information	
invalid_type	
key_not_found	
anchor_alias_dne	
initial_error	
not_a_token	
read_error	
invalid_console_input	
explicit_invalid	
distribution_impossible	
invalid_isotope	

6.8.4 Function Documentation

6.8.4.1 error()

```
void error (
            int flag )
```

Error function customizes output message based on flag.

This error function is reference in the error.cpp file, but is not called by any other file. Instead, all other files call the mError(i) macro that expands into this error function call plus prints out the file name and line number where the error occured.

6.9 fairy.h File Reference

Fission-products from Atomic Incident and their Respective Yields.

```
#include "ibis.h"
```

Classes

- class FissionProducts

  FissionProducts class object to create decay chains from fission yields.

Enumerations

- enum fiss_type { neutron, spontaneous, explosion }

  Enumeration for Fission Product Yield Type.

Functions

- fiss_type fisstype_choice (std::string &choice)

  Function to determine the fission type based on given string.
- int FAIRY_TESTS ()

  Test function for FAIRY.

6.9.1 Detailed Description

Fission-products from Atomic Incident and their Respective Yields.

fairy.cpp

This file contains a C++ object for determining fission products and their yields from some nuclear event based on: (i) type of fission, either neutron-induced or spontaneous, (ii) energy level of neutron source or bomb yield, (iii) extent of fission, and (iv) initial mass and composition of fuel or bomb materials. Data for fission products comes from ENDF-6 data libraries that were read with a python script and output into a yaml format (see 'scripts/fission-product-yields').

Author

Austin Ladshaw

Date

12/07/2018

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for research in the area of radioactive particle decay and transport. Copyright (c) 2019, all rights reserved.

6.9.2 Enumeration Type Documentation

6.9.2.1 fiss_type

```
enum fiss_type
```

Enumeration for Fission Product Yield Type.

Enumerator

neutron	
spontaneous	
explosion	

6.9.3 Function Documentation

6.9.3.1 fisstype_choice()

```
fiss_type fisstype_choice (
            std::string & choice )
```

Function to determine the fission type based on given string.

6.9.3.2 FAIRY_TESTS()

```
int FAIRY_TESTS ( )
```

Test function for FAIRY.

6.10 finch.h File Reference

Flux-limiting Implicit Non-oscillatory Conservative High-resolution scheme.

```
#include "macaw.h"
#include "lark.h"
#include "yaml_wrapper.h"
```

Classes

- struct FINCH_DATA

  Data structure for the FINCH object.

Enumerations

- enum finch_solve_type { FINCH_Picard, LARK_Picard, LARK_PJFNK }

  List of enum options to define the solver type in FINCH.
- enum finch_coord_type { Cartesian, Cylindrical, Spherical }

  List of enum options to define the coordinate system in FINCH.

Functions

- double max (std::vector< double > &values)

  Function returns the maximum in a list of values.
- double min (std::vector< double > &values)

  Function returns the minimum in a list of values.
- double minmod (std::vector< double > &values)

  Function returns the result of the minmod function acting on a list of values.
- int uTotal (FINCH_DATA ∗dat)

  Function integrates the conserved quantity to return it's total in the domain.
- int uAverage (FINCH_DATA ∗dat)

  Function integrates the conserved quantity to reture it's average in the domain.
- int check_Mass (FINCH_DATA ∗dat)

  Function checks the unp1 vector for negative values and will adjust if needed.
- int l_direct (FINCH_DATA ∗dat)

  Function solves the discretized FINCH problem directly by assuming it is linear.
- int lark_picard_step (const Matrix< double > &x, Matrix< double > &G, const void ∗data)

  Function to perform the necessary LARK Picard iterative method (not typically used)
- int nl_picard (FINCH_DATA ∗dat)

  Function to solve the discretized FINCH problem iteratively by assuming it is non-linear.
- int setup_FINCH_DATA (int(∗user_callroutine)(const void ∗user_data), int(∗user_setic)(const void ∗user_←data), int(∗user_timestep)(const void ∗user_data), int(∗user_preprocess)(const void ∗user_data), int(∗user←_solve)(const void ∗user_data), int(∗user_setparams)(const void ∗user_data), int(∗user_discretize)(const void ∗user_data), int(∗user_bcs)(const void ∗user_data), int(∗user_res)(const Matrix< double > &x, Matrix< double > &res, const void ∗user_data), int(∗user_precon)(const Matrix< double > &b, Matrix< double > &p, const void ∗user_data), int(∗user_postprocess)(const void ∗user_data), int(∗user_reset)(const void ∗user_data), FINCH_DATA ∗dat, const void ∗param_data)

  Function to setup memory and set user defined functions into the FINCH object.
- void print2file_dim_header (FILE ∗Output, FINCH_DATA ∗dat)

  Function will print out a dimension header for FINCH output.
- void print2file_time_header (FILE ∗Output, FINCH_DATA ∗dat)

  Function will print out a time header for FINCH output.
- void print2file_result_old (FILE ∗Output, FINCH_DATA ∗dat)

  Function will print out the old results to the variable u.
- void print2file_result_new (FILE ∗Output, FINCH_DATA ∗dat)

  Function will print out the new results to the variable u.
- void print2file_newline (FILE ∗Output, FINCH_DATA ∗dat)

  Function will force print out a blank line.
- void print2file_tab (FILE ∗Output, FINCH_DATA ∗dat)

  Function will force print out a tab.
- int default_execution (const void ∗user_data)

  Default executioner function for FINCH.
- int default_ic (const void ∗user_data)

  Default initial conditions function for FINCH.
- int default_timestep (const void ∗user_data)

  Default time step function for FINCH.
- int default_preprocess (const void ∗user_data)

  Default preprocesses function for FINCH.
- int default_solve (const void ∗user_data)

  Default solve function for FINCH.
- int default_params (const void ∗user_data)

  Default params function for FINCH.

- int minmod_discretization (const void ∗user_data)

  Minmod Discretization function for FINCH.
- int vanAlbada_discretization (const void ∗user_data)

  Van Albada Discretization function for FINCH.
- int ospre_discretization (const void ∗user_data)

  Ospre Discretization function for FINCH.
- int default_bcs (const void ∗user_data)

  Default boundary conditions function for FINCH.
- int default_res (const Matrix< double > &x, Matrix< double > &res, const void ∗user_data)

  Default residual function for FINCH.
- int default_precon (const Matrix< double > &b, Matrix< double > &p, const void ∗user_data)

  Default preconditioning function for FINCH.
- int default_postprocess (const void ∗user_data)
- int default_reset (const void ∗user_data)

  Default reset function for FINCH.
- int FINCH_TESTS ()

  Function runs a particular FINCH test.

6.10.1 Detailed Description

Flux-limiting Implicit Non-oscillatory Conservative High-resolution scheme.

finch.cpp

This is a conservative finite differences scheme based on the Kurganov and Tadmoor (2000) MUSCL scheme for non-linear conservation laws. It can solve 1-D conservation law problems in three different coordinate systems: (i) Cartesian - axial, (ii) Cylindrical - radial, and (iii) Spherical - radial. It is the backbone algorithm behind all 1-D PDE problems in the ecosystem software.

The form of the general conservation law problem that FINCH solves is...

z^d∗R∗du/dt = d/dz(z^d∗D∗du/dz) - d/dz(z^d∗v∗u) - z^d∗k∗u + z^d∗S

where R, D, v, k, and S are the parameters of the problem and d, z, and u are the coordinates, spatial dimension, and conserved quantities, respectively. The parameter R is a retardation coefficient, D is a diffusion coefficient, v is a velocity, k is a reaction coefficient, and S is a forcing function or source/sink term.

FINCH supports the use of both Dirichlet and Neuman boundary conditions as the input/inlet condition and uses the No Flux (or Natural) boundary condition for the output/outlet of the domain. For radial problems, the outlet is always taken to the the center of the cylindrical or spherical particle. This enforces the symmetry of the problem. For axial problems, the outlet is determined by the sign of the velocity term and is therefore choosen by the routine based on the actual flow direction in the domain.

Parameters of the problem can be coupled to the variable u and also be functions of space and time. The coupling of the parameters with the variable forces the problem to become non-linear, which requires iteration to solve. The default iterative method is a built-in Picard's method. This method is equivalent to an inexact Newton method, because we use the Linear Solve of this system as a weak approximation to the non-linear solve. Generally, this method is sufficient and is the most efficient. However, if a problem is particularly difficult to solve, then we can call some of the non-linear solvers developed in LARK. If PJFNK is used, then the Linear Solve for the FINCH problem is used as the Preconditioner for the Linear Solve in PJFNK.

This algorithm comes packaged with three different slope limiter functions to stabilize the velocity term for highly advectively dominate problems. The available slope limiters are: (i) minmod, (ii) van Albada, and (iii) ospre. By default, the FINCH setup function will set the slope limiter to ospre, because this method provides a reasonable compromise between accuracy and efficiency.

Slope Limiter Stats:

minmod -> Highest Accuracy, Lowest Efficiency
van Albada -> Lowest Accuracy, Highest Efficiency
ospre -> Average Accuracy, Average Efficiency

Author

Austin Ladshaw

Date

01/29/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.10.2 Enumeration Type Documentation

6.10.2.1 finch_solve_type

`enum finch_solve_type`

List of enum options to define the solver type in FINCH.

Enumerator

FINCH_Picard	
LARK_Picard	
LARK_PJFNK	

6.10.2.2 finch_coord_type

`enum finch_coord_type`

List of enum options to define the coordinate system in FINCH.

Enumerator

Cartesian	
Cylindrical	
Spherical	

6.10.3 Function Documentation

6.10.3.1 max()

```
double max (
            std::vector< double > & values )
```

Function returns the maximum in a list of values.

6.10.3.2 min()

```
double min (
            std::vector< double > & values )
```

Function returns the minimum in a list of values.

6.10.3.3 minmod()

```
double minmod (
            std::vector< double > & values )
```

Function returns the result of the minmod function acting on a list of values.

6.10.3.4 uTotal()

```
int uTotal (
            FINCH_DATA * dat )
```

Function integrates the conserved quantity to return it's total in the domain.

6.10.3.5 uAverage()

```
int uAverage (
            FINCH_DATA * dat )
```

Function integrates the conserved quantity to reture it's average in the domain.

6.10.3.6 check_Mass()

```
int check_Mass (
            FINCH_DATA * dat )
```

Function checks the unp1 vector for negative values and will adjust if needed.

This function can be turned off or on in the FINCH_DATA structure. Typically, you will want to leave this on so that the routine does not return negative values for u. However, if you want to get negative values of u, then turn this option off.

6.10.3.7 l_direct()

```
int l_direct (
            FINCH_DATA * dat )
```

Function solves the discretized FINCH problem directly by assuming it is linear.

6.10.3.8 lark_picard_step()

```
int lark_picard_step (
            const Matrix< double > & x,
            Matrix< double > & G,
            const void * data )
```

Function to perform the necessary LARK Picard iterative method (not typically used)

6.10.3.9 nl_picard()

```
int nl_picard (
            FINCH_DATA * dat )
```

Function to solve the discretized FINCH problem iteratively by assuming it is non-linear.

Note

> If the problem is actually linear, then this will solve it in one iteration. So it may be best to always assume the problem is non-linear.

6.10.3.10 setup_FINCH_DATA()

```
int setup_FINCH_DATA (
            int(∗)(const void ∗user_data) user_callroutine,
            int(∗)(const void ∗user_data) user_setic,
            int(∗)(const void ∗user_data) user_timestep,
            int(∗)(const void ∗user_data) user_preprocess,
            int(∗)(const void ∗user_data) user_solve,
            int(∗)(const void ∗user_data) user_setparams,
            int(∗)(const void ∗user_data) user_discretize,
            int(∗)(const void ∗user_data) user_bcs,
            int(∗)(const Matrix< double > &x, Matrix< double > &res, const void ∗user_data)
user_res,
            int(∗)(const Matrix< double > &b, Matrix< double > &p, const void ∗user_data)
user_precon,
            int(∗)(const void ∗user_data) user_postprocess,
            int(∗)(const void ∗user_data) user_reset,
            FINCH_DATA ∗ dat,
            const void ∗ param_data )
```

Function to setup memory and set user defined functions into the FINCH object.

This function MUST be called prior to running any FINCH based simulation. However, you are only every required to provide this function with the FINCH_DATA pointer. It is recommended, however, that you do provide the user←_setparams and param_data pointers, as these will likely vary significantly from problem to problem.

After the problem is setup in memory, you do not technically have to have FINCH call all of it's own functions. You can write your own executioner, initial conditions, and other functions and decided how and when everything is called. Then just call the solve function in FINCH_DATA when you want to use the FINCH solver. This is how FINCH is used in SKUA, SCOPSOWL, DOGFISH, and MONKFISH.

Parameters

user_callroutine	function pointer the the call routine function
user_setic	function pointer to set initial conditions for problem
user_timestep	function pointer to set the next time step
user_preprocess	function pointer to setup a preprocess operation
user_solve	function pointer to solve the system of equations
user_setparams	function pointer to set the parameters in the problem (always override this)
user_discretize	function pointer to select discretization scheme for the problem
user_bcs	function pointer to evaluate boundary conditions for the problem
user_res	function pointer to evaluate non-linear residuals for the problem
user_precon	function pointer to perform a linear preconditioning operation
user_postprocess	function pointer to setup a postprocess operation
user_reset	function pointer to reset stateful data for next simulation
dat	pointer to the FINCH_DATA structure
param_data	user supplied pointer to a data structure needed in user_setparams

6.10.3.11 print2file_dim_header()

```
void print2file_dim_header (
            FILE ∗ Output,
            FINCH_DATA ∗ dat )
```

Function will print out a dimension header for FINCH output.

6.10.3.12 print2file_time_header()

```
void print2file_time_header (
             FILE * Output,
             FINCH_DATA * dat )
```

Function will print out a time header for FINCH output.

6.10.3.13 print2file_result_old()

```
void print2file_result_old (
             FILE * Output,
             FINCH_DATA * dat )
```

Function will print out the old results to the variable u.

6.10.3.14 print2file_result_new()

```
void print2file_result_new (
             FILE * Output,
             FINCH_DATA * dat )
```

Function will print out the new results to the variable u.

6.10.3.15 print2file_newline()

```
void print2file_newline (
             FILE * Output,
             FINCH_DATA * dat )
```

Function will force print out a blank line.

6.10.3.16 print2file_tab()

```
void print2file_tab (
             FILE * Output,
             FINCH_DATA * dat )
```

Function will force print out a tab.

6.10.3.17 default_execution()

```
int default_execution (
            const void * user_data )
```

Default executioner function for FINCH.

The default executioner function for FINCH assumes the user_data parameter is the FINCH_DATA structure and calls the preprocesses, solve, postprocesses, checkMass, uTotal, and uAverage functions in that order.

6.10.3.18 default_ic()

```
int default_ic (
            const void * user_data )
```

Default initial conditions function for FINCH.

The default initial condition function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the initial values of all system parameters according to the given constants in that structure.

6.10.3.19 default_timestep()

```
int default_timestep (
            const void * user_data )
```

Default time step function for FINCH.

The default time step function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the time step to 1/2 the mesh size or bases the time step off of the CFL condition if the problem is not being solved iteratively and involves an advective portion.

6.10.3.20 default_preprocess()

```
int default_preprocess (
            const void * user_data )
```

Default preprocesses function for FINCH.

The default preprocesses function for FINCH assumes the user_data parameter is the FINCH_DATA structure and does nothing.

6.10.3.21 default_solve()

```
int default_solve (
            const void * user_data )
```

Default solve function for FINCH.

The default solve function for FINCH assumes the user_data parameter is the FINCH_DATA structure and calls the corresponding solution method depending on the users conditions.

6.10.3.22 default_params()

```
int default_params (
            const void * user_data )
```

Default params function for FINCH.

The default params function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the values of all parameters at all nodes equal to the values of those parameters at the boundaries.

6.10.3.23 minmod_discretization()

```
int minmod_discretization (
            const void * user_data )
```

Minmod Discretization function for FINCH.

The minmod discretization function for FINCH assumes the user_data parameter is the FINCH_DATA structure and discretizes the time and space portion of the problem with 2nd order finite differences and uses the minmod slope limiter function to stabilize the advective physics.

6.10.3.24 vanAlbada_discretization()

```
int vanAlbada_discretization (
            const void * user_data )
```

Van Albada Discretization function for FINCH.

The van Albada discretization function for FINCH assumes the user_data parameter is the FINCH_DATA structure and discretizes the time and space portion of the problem with 2nd order finite differences and uses the van Albada slope limiter function to stabilize the advective physics.

6.10.3.25 ospre_discretization()

```
int ospre_discretization (
            const void * user_data )
```

Ospre Discretization function for FINCH.

The ospre discretization function for FINCH assumes the user_data parameter is the FINCH_DATA structure and discretizes the time and space portion of the problem with 2nd order finite differences and uses the ospre slope limiter function to stabilize the advective physics. This is the default discretization function.

6.10.3.26 default_bcs()

```
int default_bcs (
            const void * user_data )
```

Default boundary conditions function for FINCH.

The default boundary conditions function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets the boundary conditions according to the type of problem requested. The input BCs will always be either Neumann or Dirichlet and the output BC will always be a zero flux Neumann BC.

6.10.3.27 default_res()

```
int default_res (
            const Matrix< double > & x,
            Matrix< double > & res,
            const void * user_data )
```

Default residual function for FINCH.

The default residual function for FINCH assumes the user_data parameter is the FINCH_DATA structure and calls the setparams function (passing the param_data structure), the discretization function, and the set BCs functions, in that order. It then forms the implicit and explicit side residuals that go into the iterative solver.

6.10.3.28 default_precon()

```
int default_precon (
            const Matrix< double > & b,
            Matrix< double > & p,
            const void * user_data )
```

Default preconditioning function for FINCH.

The default preconditioning function for FINCH assumes the user_data parameter is the FINCH_DATA structure and performs a tridiagonal linear solve using a Modified Thomas Algorithm. This preconditioner will solve the linear problem exactly if there is no advective portion of the physics. Additionally, this preconditioner is also used as the basis for forming the default FINCH non-linear iterations and is sufficient for solving most problems.

6.10.3.29 default_postprocess()

```
int default_postprocess (
            const void * user_data )
```

The default postprocesses function for FINCH assumes the user_data parameter is the FINCH_DATA structure and does nothing.

6.10.3.30 default_reset()

```
int default_reset (
            const void * user_data )
```

Default reset function for FINCH.

The default reset function for FINCH assumes the user_data parameter is the FINCH_DATA structure and sets all old state parameters and variables to the new state.

6.10.3.31 FINCH_TESTS()

```
int FINCH_TESTS ( )
```

Function runs a particular FINCH test.

The FINCH_TESTS function is used to exercise and test out the FINCH algorithms for correctness, efficiency, and accuracy. This test should never report a failure.

6.11 flock.h File Reference

FundamentaL Off-gas Collection of Kernels.

```
#include "macaw.h"
#include "egret.h"
#include "finch.h"
#include "lark.h"
#include "skua.h"
#include "scopsowl.h"
#include "gsta_opt.h"
#include "magpie.h"
#include "skua_opt.h"
#include "scopsowl_opt.h"
#include "yaml_wrapper.h"
#include "dove.h"
#include "crow.h"
#include "mesh.h"
#include "crane.h"
#include "ibis.h"
#include "fairy.h"
#include "kea.h"
#include "cardinal.h"
```

6.11.1 Detailed Description

FundamentaL Off-gas Collection of Kernels.

This is just a .h file that holds all the includes necessary to develop and run simulations for adsorption and/or mass/energy transfer problems for gaseous systems. Include this file into any other project or source code that needs the methods below.

Files Included in FLOCK

macaw.h egret.h finch.h lark.h skua.h scopsowl.h gsta_opt.h magpie.h skua_opt.h scopsowl_opt.h yaml_wrapper.h dove.h

Author

Austin Ladshaw

Date

04/28/2014

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.12 gsta_opt.h File Reference

Generalized Statistical Thermodynamic Adsorption (GSTA) Optimization Routine.

```
#include "lmcurve.h"
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include "error.h"
```

Classes

- struct GSTA_OPT_DATA

  Data structure used in the GSTA optimization routines.

Macros

- #define Po 100.0

  Standard State Pressure - Units: kPa.
- #define R 8.3144621

  Gas Constant - Units: J/(K∗mol) = kB ∗ Na.
- #define Na 6.0221413E+23

  Avagadro's Number - Units: molecules/mol.

Functions

- int roundIt (double d)

  Function rounds a double to an integer.
- int twoFifths (int m)

  Function returns the rounded two-fifths result of int m.
- int orderMag (double x)

  Function returns the order of magnitude for the parameter x.
- int minValue (std::vector< int > &array)

  Function returns the minimum integer in an array of integers.
- int minIndex (std::vector< double > &array)

  Function returns the index of the minimum integer in an array of integers.
- int avgPar (std::vector< int > &array)

  Function returns the average integer value in an array of integers.
- double avgValue (std::vector< double > &array)

  Function returns an average in an array of doubles.
- double weightedAvg (double ∗enorm, double ∗x, int n)

  Function returns a weighted average in an array.
- double rSq (double ∗x, double ∗y, double slope, double vint, int m_dat)

Function calculates the Coefficient of Determination (R Squared) for the temperature regression.
- bool isSmooth (double ∗par, void ∗data)

    Function looks at the list of parameters to check if they are smoothly changing.
- void orthoLinReg (double ∗x, double ∗y, double ∗par, int m_dat, int n_par)

    Function performs an Orthogonal Linear Regression on a set of data.
- void eduGuess (double ∗P, double ∗q, double ∗par, int k, int m_dat, void ∗data)

    Function will formed an educated guess for the next set of parameters in the GSTA analysis.
- double gstaFunc (double p, const double ∗K, double qmax, int n_par)

    Function evaluates the result of the GSTA isotherm model.
- double gstaObjFunc (double ∗t, double ∗y, double ∗par, int m_dat, void ∗data)

    Function to evaulate the GSTA objective function value.
- void eval_GSTA (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

    Function to evaluate the GSTA model and feed into the lmfit routine.
- int gsta_optimize (const char ∗fileName)

    Function to perform the GSTA optimization routine.

6.12.1 Detailed Description

Generalized Statistical Thermodynamic Adsorption (GSTA) Optimization Routine.

gsta_opt.cpp

Optimization routine developed for the GSTA isotherm and data analysis. This algorithm was the primary subject of a publication made in Fluid Phase Equilibria. Please refer to the below paper for technical information about the algorithms.

Reference: Ladshaw, Yiacoumi, Tsouris, and DePaoli, Fluid Phase Equilibria, 388, 169-181, 2015.

The GSTA model was first introduced by Llano-Restrepo and Mosquera (2009). Please refer to the below reference for theoretical information about the model.

Reference: Llano-Restrepo and Mosquera, Fluid Phase Equilibria, 283, 73-88, 2009.

Author

Austin Ladshaw

Date

12/17/2013

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.12.2 Macro Definition Documentation

6.12.2.1 Po

```
#define Po 100.0
```

Standard State Pressure - Units: kPa.

6.12.2.2 R

```
#define R 8.3144621
```

Gas Constant - Units: J/(K∗mol) = kB ∗ Na.

6.12.2.3 Na

```
#define Na 6.0221413E+23
```

Avagadro's Number - Units: molecules/mol.

6.12.3 Function Documentation

6.12.3.1 roundIt()

```
int roundIt (
            double d )
```

Function rounds a double to an integer.

This function returns a rounded value of d. Rounding up for any decimal larger than 0.5 and down for all else.

6.12.3.2 twoFifths()

```
int twoFifths (
            int m )
```

Function returns the rounded two-fifths result of int m.

This function is used to determine what the maximum number of parameters should be based on the number of data points m. It is designed to prevent the algorithms from "over fitting" the data.

6.12.3.3 orderMag()

```
int orderMag (
            double x )
```

Function returns the order of magnitude for the parameter x.

This function is used to help create initial guesses for the new GSTA parameters that are being optimized for. In order to make sure that those parameters are considered relavent in the optimization routine, we need to make the initial guesses to be around the same order of magnitude of the other GSTA parameters.

6.12.3.4 minValue()

```
int minValue (
            std::vector< int > & array )
```

Function returns the minimum integer in an array of integers.

This function is used to determine the minimum number of GSTA parameters that were required to adequately fit the isotherm data.

6.12.3.5 minIndex()

```
int minIndex (
            std::vector< double > & array )
```

Function returns the index of the minimum integer in an array of integers.

This function identifies the index of the minimum number of parameters needed for the GSTA model to fit the data. This index is common for all vectors in the GSTA_OPT_DATA structure and is used to identify the most suitable solution.

6.12.3.6 avgPar()

```
int avgPar (
            std::vector< int > & array )
```

Function returns the average integer value in an array of integers.

This function is used to identify the average number of parameters that all the data fitting needed for each GSTA analysis.

6.12.3.7 avgValue()

```
double avgValue (
            std::vector< double > & array )
```

Function returns an average in an array of doubles.

6.12.3.8 weightedAvg()

```
double weightedAvg (
            double * enorm,
            double * x,
            int n )
```

Function returns a weighted average in an array.

This averaging scheme is used to approximate the qmax parameter for the GSTA isotherm model, if that value is unknown. The weighting is based on the euclidean norms of all the fits of the data. Smaller norms are more heavily weighted since they represent a better fit of the data. Once averaging is complete and we have an estimate for qmax, the entire algorithm is re-run holding that qmax constant.

Parameters

enorm	array of euclidean norms from the fitting of the data
x	array of optimum qmax values to be averaged
n	the number of enorm and x values in the array

6.12.3.9 rSq()

```
double rSq (
            double * x,
            double * y,
            double slope,
            double vint,
            int m_dat )
```

Function calculates the Coefficient of Determination (R Squared) for the temperature regression.

This function is used to determine the "fittness" of the linear regression performed on the temperature independent parameters of the GSTA isotherm. A good linear regression should return a value between 1.0 and 0.9.

Parameters

x	observations in the x-axis
y	observations in the y-axis
slope	slope of the linear regression
vint	intercept of the linear regression
m_dat	number of data points used in the linear regression

6.12.3.10 isSmooth()

```
bool isSmooth (
            double * par,
            void * data )
```

Function looks at the list of parameters to check if they are smoothly changing.

This function takes the parameter array par and GSTA_OPT_DATA structure and checks to see if those parameters are changing smoothly. If they are erratic or non-smooth, then it could be an indication of "over fitting" of the data.

6.12.3.11 orthoLinReg()

```
void orthoLinReg (
            double * x,
            double * y,
            double * par,
            int m_dat,
            int n_par )
```

Function performs an Orthogonal Linear Regression on a set of data.

This function takes an array of x and y observations and performs an orthogonal linear regression on that information to find optimum parameters for slope and intercept.

Parameters

x	array of x-axis observations
y	array of y-axis observations
par	array of parameter results after regression
m_dat	number of data points or observations
n_par	number of parameters to seek (if n_par != 1 or 2, then par[0] = intercept and par[1] = slope)

6.12.3.12 eduGuess()

```
void eduGuess (
             double * P,
             double * q,
             double * par,
             int k,
             int m_dat,
             void * data )
```

Function will formed an educated guess for the next set of parameters in the GSTA analysis.

This function takes partial pressure and adsorption observations, P and q, and tries to give a decent initial guess to what the GSTA parameters, par, will be for the next iteration.

Parameters

P	partial pressure observations in the data (kPa)
q	adsorption observations in the data (any units)
par	parameter array for the GSTA isotherm
k	index of the current number of parameters being considered
m_dat	number of pressure-adsorption observations in the isotherm
data	pointer to the GSTA_OPT_DATA data structure

6.12.3.13 gstaFunc()

```
double gstaFunc (
             double p,
             const double * K,
             double qmax,
             int n_par )
```

Function evaluates the result of the GSTA isotherm model.

This function will evaluate the GSTA model and return the adsorbed amount given the current partial pressure p and the equilibrium parameters K.

Parameters

p	current partial pressure (kPa)
K	array of equilibrium parameters (1/kPa^n)
qmax	the theorectical maximum capacity for the isotherm
n_par	the number of equilibrium parameters

6.12.3.14 gstaObjFunc()

```
double gstaObjFunc (
            double * t,
            double * y,
            double * par,
            int m_dat,
            void * data )
```

Function to evaulate the GSTA objective function value.

The objective function seeks to penalize the relative fittness of the model based on the number of parameters it took to minimize the euclidean norms. By penalizing the fittness of the model in this fashion, we can find the best solution to the system that required the least number of equilibrium parameters.

6.12.3.15 eval_GSTA()

```
void eval_GSTA (
            const double * par,
            int m_dat,
            const void * data,
            double * fvec,
            int * info )
```

Function to evaluate the GSTA model and feed into the lmfit routine.

This function will formulate the residuals that go into the Levenberg-Marquardt's Algorithm for non-linear least squares regression. The form of this function is specific to how we interface with the lmfit routines.

6.12.3.16 gsta_optimize()

```
int gsta_optimize (
            const char * fileName )
```

Function to perform the GSTA optimization routine.

This function is callable from the UI and is used to find the optimum parameters of the GSTA isotherm model given a particular set of isotherm data for single-component adsorption equilibria.

Parameters

fileName	name of the input file that holds the isotherm data

Note

The input file for the GSTA optimization routine is a text file holding the necessary information and data needed to run the routine. That input file has a very specific format that is detailed below.
Number of Isotherm Curves
Theoretical Maximum Adsorption Capacity (if unknown, provide 0)
Temperature of the ith Isotherm (K)
Number of Data points for the ith Isotherm
Partial Pressure (kPa) [tab] Corresponding Adsorbed Amount (any units)

(2nd Line down is repeated for all isotherms you are optimizing on...)

Example:

2
21.0
298.15
4
0.000165483 2.77
0.000306379 2.75
0.00044922 5.00
0.000939259 10.40
313.15
4
0.000589636 2.75
0.001063584 3.70
0.001351836 4.2
0.001543464 4.6
The above example would be for 2 sets of isotherms at 298.15 and 313.15 K, respectively. Maximum adsorption capacity is given as 21 (which in this has units of wt%). Each isotherm has 4 data points, which are given in a list as p (kPa) and q (wt%) pairs. Units of adsorption don't matter as long as they are consistent. If you give maximum capacity in mol/kg, then the q's in the lists must also be in mol/kg.

6.13 ibis.h File Reference

Implicit Branching Isotope System.

```
#include "eel.h"
#include "yaml_wrapper.h"
#include "dove.h"
#include "gsta_opt.h"
#include <unordered_map>
```

Classes

- class Isotope
  Isotope object to hold information and provide decay operations.
- class DecayChain
  DecayChain object to hold and store a set of unique isotopes in a branched decay chain.

Enumerations

- enum decay_mode {
  alpha, beta_min, beta_plus, stable,
  spon_fiss, iso_trans, neutron_em, proton_em,
  beta_min_neutron_em, beta_plus_neutron_em, beta_plus_alpha, beta_plus_beta_plus,
  beta_min_beta_min, beta_min_2neutron_em, beta_min_alpha, proton_em_proton_em,
  neutron_em_neutron_em, beta_min_3neutron_em, beta_min_4neutron_em, beta_plus_2proton_em,
  beta_plus_3proton_em, specific_isotope, beta_plus_proton_em, undefined }
  Enumeration for the list of valid decay modes.
- enum time_units {
  seconds, minutes, hours, days,
  years }
  Enumeration for the list of valid units of half-life.

Functions

- double time_conversion (time_units end_unit, double start_value, time_units start_unit)

    Function to convert from a starting unit and value to and ending unit and value (returns converted value)
- time_units timeunits_choice (std::string &choice)

    Function to determine what type of time units are used based on given string argument.
- decay_mode decaymode_choice (std::string &choice)

    Function to determine the decay mode based on given string.
- std::string decaymode_string (decay_mode mode)

    Function to return a string for the name of a decay mode given the enum.
- std::string timeunits_string (time_units units)

    Function to return a string for the units.
- double EmpFermi (int z, double W)

    Empirical formulation of the Fermi Function.
- double EmpShape (int L, double W0, double W)

    Empirical formulation of the Shape Function.
- double DistEnergy (int z, int L, double W0, double W)

    Distribution of beta energy function.
- double MeanEnergy_Stepwise (int z, int L, double E0)

    Stepwise integration for average energy.
- double aux_beta_minus (double beta, double eta, double tau)

    Auxillary Function for beta-.
- double aux_beta_plus (double beta, double eta, double tau)

    Auxillary Function for beta+.
- double mean_excitation_energy (int atom_num)

    Mean Excitation energy (in eV)
- double stopping_power_beta_minus (double mean_energy, int Zj, double Aj, double delta)

    Stopping power for beta minus.
- double stopping_power_beta_plus (double mean_energy, int Zj, double Aj, double delta)

    Stopping power for beta plus.
- double stopping_power_nonbeta (double mean_energy, int Zj, double Aj, double delta, double charge)

    Stopping power for non-beta.
- double mean_path_beta (double mean_energy, double density)

    Mean path length for beta (cm)
- double mean_path_nonbeta (double mean_energy)

    Mean path length for non-beta (cm)
- int IBIS_SCENARIO (const char ∗yaml_input)

    IBIS Executable given an input file.
- int IBIS_TESTS ()

6.13.1 Detailed Description

Implicit Branching Isotope System.

ibis.cpp

This file contains a C++ object for creating and utilizing isotopes in a branching isotope decay chain. The object inherits from Class Atom (see eel.h), which creates individual atoms from names, symbols, or atomic numbers, then adds the ability to decay those atoms to different isotopes through various decay modes. Added to the Atom object are parameters for decay rates (half-lifes), branching ratios, and decay modes (alpha, beta, etc). The intent of this system is to determine how radioactive decay occurs when given a single or multiple starting isotopes. The branching system of isotopes is solved analytically using an eigenvector-eigenvalue solution coupled with a sorting algorithm to guerentee a lower triangular coefficient matrix. In addition, we also solve for the accumulation of the stable isotopes by integrating the unstable solution through time. This method is based on similar methods from the following sources:

Decay Solution Methods:

[1] M. Amaku, P.R. Pascholati, V.R. Vanin, Comp. Phys. Com. 181, 21-23, 2010. [2] D. Yuan, W. Kernan, J. Appl. Phys. 101, 094907, 2007.

Also implemented in the Isotope class are methods to estimate the mean radiation energy of beta emitters and the ionization rate coefficient for nuclides of any decay modes. The mean radiation energy uses approximations for the Fermi Function and Shape Function of the beta spectra. Methods could potentially be improved upon by updating those two functions. The ionization coefficient is calculated using the Linear Energy Transfer (LET) model coupled with theoretical models for the Stopping Power of the emissions based on decay type. References are as follows:

LET and Stopping Power Functions:

[1] Y.H. Kim, S. Yiacoumi, C. Tsouris, J. Env. Radio. 143, 91-99, 2015. [2] H. Huizenga, P.R.M. Storchi, Phys. Med. Biol. 34, 1371-1396, 1989. [3] ICRU Report 37, "Stopping Powers for Electrons and Positrons," 1984. [4] W.A. Tome, J.R. Palta, Med. Phys. 5, 758-772, 1998. [5] Q.H. Mohammad, H.A. Maghdid, Research Reviews: J. Pure Appl. Phys. 5, 22-28, 2017.

Mean Energy, Fermi Functions, and Beta Spectra Shape Functions:

[1] X. Mougeot, Phys. Rev. C, 91, 055504, 2015. [2] V. Venkataramaiah, K. Gopala, A. Basavaraju, S.S. Suryanarayana, H. Sanjeeviah, J. Phys. G: Nucl. Phys. 11, 359-364, 1985. [3] P.J. Mohr, B.N. Taylor, D.B. Newell, Reviews of Mod. Phys. 84, 1527-1605, 2012.

Author

Austin Ladshaw

Date

09/04/2018

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for research in the area of radioactive particle decay. Copyright (c) 2018, all rights reserved.

6.13.2 Enumeration Type Documentation

6.13.2.1 decay_mode

```
enum decay_mode
```

Enumeration for the list of valid decay modes.

List of valid types of radioactive decay. The type of decay dictates how the Isotope object will transform the given isotope.

Enumerator

alpha	
beta_min	
beta_plus	
stable	
spon_fiss	
iso_trans	
neutron_em	
proton_em	
beta_min_neutron_em	
beta_plus_neutron_em	
beta_plus_alpha	
beta_plus_beta_plus	
beta_min_beta_min	
beta_min_2neutron_em	
beta_min_alpha	
proton_em_proton_em	
neutron_em_neutron_em	
beta_min_3neutron_em	
beta_min_4neutron_em	
beta_plus_2proton_em	
beta_plus_3proton_em	
specific_isotope	
beta_plus_proton_em	
undefined	

6.13.2.2 time_units

```
enum time_units
```

Enumeration for the list of valid units of half-life.

List of valid units for half-lifes for better readability of code.

Enumerator

seconds	
minutes	
hours	
days	
years	

6.13.3 Function Documentation

6.13.3.1 time_conversion()

```
double time_conversion (
             time_units end_unit,
             double start_value,
             time_units start_unit )
```

Function to convert from a starting unit and value to and ending unit and value (returns converted value)

6.13.3.2 timeunits_choice()

```
time_units timeunits_choice (
             std::string & choice )
```

Function to determine what type of time units are used based on given string argument.

6.13.3.3 decaymode_choice()

```
decay_mode decaymode_choice (
             std::string & choice )
```

Function to determine the decay mode based on given string.

6.13.3.4 decaymode_string()

```
std::string decaymode_string (
             decay_mode mode )
```

Function to return a string for the name of a decay mode given the enum.

6.13.3.5 timeunits_string()

```
std::string timeunits_string (
             time_units units )
```

Function to return a string for the units.

6.13.3.6 EmpFermi()

```
double EmpFermi (
             int z,
             double W )
```

Empirical formulation of the Fermi Function.

6.13.3.7 EmpShape()

```
double EmpShape (
             int L,
             double W0,
             double W )
```

Empirical formulation of the Shape Function.

6.13.3.8 DistEnergy()

```
double DistEnergy (
             int z,
             int L,
             double W0,
             double W )
```

Distribution of beta energy function.

6.13.3.9 MeanEnergy_Stepwise()

```
double MeanEnergy_Stepwise (
             int z,
             int L,
             double E0 )
```

Stepwise integration for average energy.

6.13.3.10 aux_beta_minus()

```
double aux_beta_minus (
             double beta,
             double eta,
             double tau )
```

Auxillary Function for beta-.

6.13.3.11 aux_beta_plus()

```
double aux_beta_plus (
             double beta,
             double eta,
             double tau )
```

Auxillary Function for beta+.

6.13.3.12 mean_excitation_energy()

```
double mean_excitation_energy (
             int atom_num )
```

Mean Excitation energy (in eV)

6.13.3.13 stopping_power_beta_minus()

```
double stopping_power_beta_minus (
             double mean_energy,
             int Zj,
             double Aj,
             double delta )
```

Stopping power for beta minus.

6.13.3.14 stopping_power_beta_plus()

```
double stopping_power_beta_plus (
             double mean_energy,
             int Zj,
             double Aj,
             double delta )
```

Stopping power for beta plus.

6.13.3.15 stopping_power_nonbeta()

```
double stopping_power_nonbeta (
             double mean_energy,
             int Zj,
             double Aj,
             double delta,
             double charge )
```

Stopping power for non-beta.

6.13.3.16 mean_path_beta()

```
double mean_path_beta (
             double mean_energy,
             double density )
```

Mean path length for beta (cm)

6.13.3.17 mean_path_nonbeta()

```
double mean_path_nonbeta (
            double mean_energy )
```

Mean path length for non-beta (cm)

6.13.3.18 IBIS_SCENARIO()

```
int IBIS_SCENARIO (
            const char * yaml_input )
```

IBIS Executable given an input file.

Main IBIS executable from the ecosystem cli. User must provide a yaml-style input file directing all simulation, isotope, and runtime options. See example yaml-style input file below.

Input file has a Runtime block and an Isotopes block. The Runtime block is used to direct the type of simulation to be run, what time units to use, and what information to print to the output file. If print_results is false, then the simulations are not carried out for the isotope fractionation. If verify is false, then IBIS will skip the verification of the eigen solution.

The Isotopes block is where you provide the inital conditions for a starting set of isotopes. Each sub-block in Isotopes must be the name of an isotope registered in the Nuclide library. Currently, initial conditions for isotopes can only be given in moles of isotopes.

Yaml Input File Example

Runtime:

time_units: seconds end_time: 3600 time_steps: 10

verify: true print_sparsity: true print_chain: true print_results: false ...

Isotopes:

- U-235: initial_cond: 90
- U-238: initial_cond: 10 ...Function to test the implementation of Isotope

6.13.3.19 IBIS_TESTS()

```
int IBIS_TESTS ( )
```

6.14 kea.h File Reference

Kernel for Estimating Activity-distribution.

```
#include "fairy.h"
#include "mola.h"
```

Classes

- class ActivityDistribution

  C++ Object for determining the activity-size distribution.

Enumerations

- enum asd_model { freiling, freiling_tompkins, mod_freiling, mod_freiling_tompkins }

  Enumeration for the list of valid activity-size distribution methods.

Functions

- asd_model activitymodel_choice (std::string &choice)

  Function to determine the activity-size distribution model type.
- int KEA_TESTS ()

  Test function for KEA.

6.14.1 Detailed Description

Kernel for Estimating Activity-distribution.

kea.cpp

This file contains a C++ object for determining the distribution of radioactivity particles and activity onto debris particles in specific size classes. It is directly coupled with FAIRY to determine yields of nuclides from a specific nuclear event, then will be integrated into CRANE to establish the distribution of nuclides in the debris cloud. For the sake of modularity, this kernel will not be coupled with CRANE and instead CRANE will integrate this kernel into its source. Thus, the activity-distribution in KEA will be determined from information that is anticipated to be passed to the functions and objects developed here. That will allow for independent testing of this kernel and allow for changes to how the activity-distribution is determined to be made on the fly.

References for Activity-Distribution

E.C. Freiling, "Radionuclide Fractionation in Bomb Debris," Science, 1991-1998, 1961.

J.T. McGahan, E.J. Kownaki, "Sensitivity of Fallout Predictions to Initial Conditions and Model Assumptions," Defense Nuclear Agency, DNA-3439F, 1974.

H.G. Norment, "DELFIC: Department of Defense Fallout Prediction System: Volume I - Fundamentals," Defense Nuclear Agency, DNA-5159F, 1979.

R.C. Tompkins, "DELFIC: Department of Defense Fallout Prediction System: Volume V - Pacticle Activity," US Army Nuclear Defense Laboratory, DASA-1800-V, 1968.

D.A. Hooper, V.J. Jodoin, "Revision of the DELFIC Particle Activity Module," Oak Ridge National Laboratory, OR↩NL/TM-2010/220, 2010.

Reference for Induced-Soil Activity

T.H. Jones, "A Prediction System for the Neutron-Induced Activity Contribution to Fallout Exposure Rates," U.S. Naval Radiological Defense Laboratory, USNRDL-TR-1056, 1966.

Reference for Neutron Absorption and Scattering Cross Sections (in EEL)

V.F. Sears, "Neutron Scattering Lengths and Cross Sections," Neutron News, 3, 26-37, 1992.

Author

Austin Ladshaw

Date

02/07/2019

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for research in the area of radioactive particle decay and transport. Copyright (c) 2019, all rights reserved.

6.14.2 Enumeration Type Documentation

6.14.2.1 asd_model

```
enum asd_model
```

Enumeration for the list of valid activity-size distribution methods.

List of valid models for activity-size distributions.

Enumerator

freiling	
freiling_tompkins	
mod_freiling	
mod_freiling_tompkins	

6.14.3 Function Documentation

6.14.3.1 activitymodel_choice()

```
asd_model activitymodel_choice (
            std::string & choice )
```

Function to determine the activity-size distribution model type.

6.14.3.2 KEA_TESTS()

```
int KEA_TESTS ( )
```

Test function for KEA.

6.15 lark.h File Reference

Linear Algebra Residual Kernels.

```
#include "macaw.h"
#include <float.h>
```

Classes

- struct ARNOLDI_DATA

  Data structure for the construction of the Krylov subspaces for a linear system.
- struct GMRESLP_DATA

  Data structure for implementation of the Restarted GMRES algorithm with Left Preconditioning.
- struct GMRESRP_DATA

  Data structure for the Restarted GMRES algorithm with Right Preconditioning.
- struct PCG_DATA

  Data structure for implementation of the PCG algorithms for symmetric linear systems.
- struct BiCGSTAB_DATA

  Data structure for the implementation of the BiCGSTAB algorithm for non-symmetric linear systems.
- struct CGS_DATA

  Data structure for the implementation of the CGS algorithm for non-symmetric linear systems.
- struct OPTRANS_DATA

  Data structure for implementation of linear operator transposition.
- struct GCR_DATA

  Data structure for the implementation of the GCR algorithm for non-symmetric linear systems.
- struct GMRESR_DATA

  Data structure for the implementation of GCR with Nested GMRES preconditioning (i.e., GMRESR)
- struct KMS_DATA

  Data structure for the implemenation of the Krylov Multi-Space (KMS) Method.
- struct QR_DATA

  Data structure for the implementation of a QR solver given some invertable linear operator.
- struct PICARD_DATA

  Data structure for the implementation of a Picard or Fixed-Point iteration for non-linear systems.
- struct BACKTRACK_DATA

  Data structure for the implementation of Backtracking Linesearch.
- struct PJFNK_DATA

  Data structure for the implementation of the PJFNK algorithm for non-linear systems.
- struct NUM_JAC_DATA

  Data structure to form a numerical jacobian matrix with finite differences.

Macros

- #define MIN_TOL 1e-15

  Minimum Allowable Tolerance for linear and non-linear problems.

Enumerations

- enum krylov_method {
GMRESLP, PCG, BiCGSTAB, CGS,
FOM, GMRESRP, GCR, GMRESR,
KMS, QR }

    Enum of definitions for linear solver types in PJFNK.

Functions

- int update_arnoldi_solution (Matrix< double > &x, Matrix< double > &x0, ARNOLDI_DATA *arnoldi_dat)

    Function to update the linear vector x based on the Arnoldi Krylov subspace.
- int arnoldi (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &r0, ARNOLDI_DATA *arnoldi_dat, const void *matvec_data, const void *precon_data)

    Function to factor a linear operator into an orthonormal basis and upper Hessenberg matrix.
- int gmresLeftPreconditioned (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &b, GMRESLP_DATA *gmreslp_dat, const void *matvec_data, const void *precon_data)

    Function to iteratively solve a non-symmetric, indefinite linear system with GMRESLP.
- int fom (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &b, GMRESLP_DATA *gmreslp_dat, const void *matvec_data, const void *precon_data)

    Function to directly solve a non-symmetric, indefinite linear system with FOM.
- int gmresRightPreconditioned (int(*matvec)(const Matrix< double > &v, Matrix< double > &w, const void *data), int(*precon)(const Matrix< double > &b, Matrix< double > &p, const void *data), Matrix< double > &b, GMRESRP_DATA *gmresrp_dat, const void *matvec_data, const void *precon_data)

    Function to iteratively solve a non-symmetric, indefinite linear system with GMRESRP.
- int pcg (int(*matvec)(const Matrix< double > &p, Matrix< double > &Ap, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &z, const void *data), Matrix< double > &b, PCG_DATA *pcg_dat, const void *matvec_data, const void *precon_data)

    Function to iteratively solve a symmetric, definite linear system with PCG.
- int bicgstab (int(*matvec)(const Matrix< double > &p, Matrix< double > &Ap, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &z, const void *data), Matrix< double > &b, BiCGSTAB_DATA *bicg_dat, const void *matvec_data, const void *precon_data)

    Function to iteratively solve a non-symmetric, definite linear system with BiCGSTAB.
- int cgs (int(*matvec)(const Matrix< double > &p, Matrix< double > &Ap, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &z, const void *data), Matrix< double > &b, CGS_DATA *cgs_dat, const void *matvec_data, const void *precon_data)

    Function to iteratively solve a non-symmetric, definite linear system with CGS.
- int operatorTranspose (int(*matvec)(const Matrix< double > &v, Matrix< double > &Av, const void *data), Matrix< double > &r, Matrix< double > &u, OPTRANS_DATA *transpose_dat, const void *matvec_data)

    *Function that is used to perform transposition of a linear operator and results in a new vector A^T*r=u.*
- int gcr (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &Mr, const void *data), Matrix< double > &b, GCR_DATA *gcr_dat, const void *matvec_data, const void *precon_data)

    Function to iteratively solve a non-symmetric, definite linear system with GCR.
- int gmresrPreconditioner (const Matrix< double > &r, Matrix< double > &Mr, const void *data)

    Function used in conjunction with GMRESR to apply GMRESRP iterations as a preconditioner.
- int gmresr (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), int(*terminal_precon)(const Matrix< double > &r, Matrix< double > &Mr, const void *data), Matrix< double > &b, GMRESR_DATA *gmresr_dat, const void *matvec_data, const void *term_precon_data)

    Function to iteratively solve a non-symmetric, indefinite linear system with GMRESR.

- int kmsPreconditioner (const Matrix< double > &r, Matrix< double > &Mr, const void *data)

  Preconditioner function for the Krylov Multi-Space.
- int krylovMultiSpace (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), int(*terminal_precon)(const Matrix< double > &r, Matrix< double > &Mr, const void *data), Matrix< double > &b, KMS_DATA *kms_dat, const void *matvec_data, const void *term_precon_data)

  Function to iteratively solve a non-symmetric, indefinite linear system with KMS.
- int QRsolve (int(*matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void *data), Matrix< double > &b, QR_DATA *qr_dat, const void *matvec_data)

  Function to solve a dense linear operator system using QR factorization.
- int picard (int(*res)(const Matrix< double > &x, Matrix< double > &r, const void *data), int(*evalx)(const Matrix< double > &x0, Matrix< double > &x, const void *data), Matrix< double > &x, PICARD_DATA *picard_dat, const void *res_data, const void *evalx_data)

  Function to iteratively solve a non-linear system using the Picard or Fixed-Point method.
- int jacvec (const Matrix< double > &v, Matrix< double > &Jv, const void *data)

  Function to form a linear operator of a Jacobian matrix used along with the PJFNK method.
- int backtrackLineSearch (int(*feval)(const Matrix< double > &x, Matrix< double > &F, const void *data), Matrix< double > &Fkp1, Matrix< double > &xkp1, Matrix< double > &pk, double normFk, BACKTRAC↩ K_DATA *backtrack_dat, const void *feval_data)

  Function to perform a Backtracking Line Search operation to smooth out convergence of PJFNK.
- int pjfnk (int(*res)(const Matrix< double > &x, Matrix< double > &F, const void *data), int(*precon)(const Matrix< double > &r, Matrix< double > &p, const void *data), Matrix< double > &x, PJFNK_DATA *pjfnk↩ _dat, const void *res_data, const void *precon_data)

  Function to perform the PJFNK algorithm to solve a non-linear system of equations.
- int NumericalJacobian (int(*Func)(const Matrix< double > &x, Matrix< double > &F, const void *user_↩ data), const Matrix< double > &x, Matrix< double > &J, int Nx, int Nf, NUM_JAC_DATA *jac_dat, const void *user_data)

  Function to form a full numerical Jacobian matrix from a given non-linear function.
- int LARK_TESTS ()

  Function that runs a variety of tests on all the functions in LARK.

6.15.1 Detailed Description

Linear Algebra Residual Kernels.

lark.cpp

The functions contained within are designed to solve generic linear and non-linear square systems of equations given a function argument and data from the user. Optionally, the user can also provide a function to return a preconditioning result that will be applied to the system.

Having the user define how the preconditioning is carried out provides two major advantages: (1) we do not need to store and large, sparse preconditioning matrices and instead only store the preconditioned vector result and (2) this allows the user to use any kind of preconditioner they see fit for their problem.

The Arnoldi function is typically not called by the user, but can be if desired. It accepts the function arguments and a residual vector to form an orthonormal basis of the Krylov subspace using the Modified Gram-Schmidt process (aka Arnoldi Iteration). This function is called by GMRES to iteratively solve a linear system of equations. Note that you can use this function to directly solve the linear system as long as that system is not too large. Construction of the basis is expensive, which is why this is used as a sub-function of an iterative method.

The Restarted GMRES function will accept function arguments for a linear system and attempt to solve said system iteratively by constructing an orthonormal basis from the Krylov function. Note that this GMRES function does support restarting and will use restarting by default if the linear system is too large.

Also included is a GMRES algorithm without restarting. This will directly solve the linear system within residual tolerance using a Full Orthogonal basis set of that system. It is equivalent to calling the Krylov method with the k parameter equal to N (i.e. the number of equations). This method is nick-named the Full Othogonalization Method (FOM), although the true FOM algorithm in literature is slightly different.

The PJFNK function will accept function arguments for a square, non-linear system of equations and attempt to solve it iteratively using both the GMRES and Krylov functions with Newton's method to convert the non-linear system into a linear system.

Also built here is a PCG implementation for solving symmetric linear systems. Can also be called by PJFNK if we know that the linear system (i.e. the Jacobian) is symmetric. This algorithm is significantly more efficient than GMRES, but is only valid if the system of equations is symmetric.

Other linear solvers implemented in this work are the BiCGSTAB and CGS algorithms for non-symmetric, positive definite matrices. These algorithms are significantly more computationally efficient than GMRES or FOM. However, they can both break down if the linear system is poorly conditioned. In general, you only want to use these methods if you have preconditioning available and your linear system is very, very large. Otherwise, you will be better suited to using GMRES or FOM.

There is also an implementation of the Generalized Conjugate Residual (GCR) method with and without restarting. This is a GMRES-like method that should give the exact solution within N iterations, where N is the original size of the matrix. Built ontop of the GCR method is a GMRESR (or GMRES Recursive) algorithm that uses GCR as the base method and performs GMRESRP iterations as a preconditioner at each iteration of GCR. This is the only linear solver that has built-in preconditioning. As a result, it may be slower than other algorithms for simple problems, but generally will have much better convergence behavior and will almost always give better residual reduction, even for hard to solve problems.

We have also developed a novel/experimental iterative method based on the idea of recursively preconditioning a Krylov Subspace with more Krylov Subspaces. We have called with algorithm the Krylov Multi-Space (KMS) method. This algorithm is based on publications from Vorst and Vuik (1991) and Saad (1993). The idea is too use the FGMRES algorithm developed by Saad (1993) and precondition it with more FGMRES steps, i.e., nesting the iterations as Vorst and Vuik (1991) had proposed. In this way, we have created a generalized Krylov Subspace method that has it's own variable preconditioner that can be adjusted depending on the user's desired complexity and convergence rate. If the levels of recursion requested is zero, then this algorithm is exactly equal to GMRES with right preconditioning. If the level is one, then it is FGMRES with a GMRES preconditioner. However, we allow the levels of recursion to reach up to 5, thus allowing us to precondition the preconitioners with more GMRES steps. This can result is significantly faster convergence rates, but is typically only necessary for very large or difficult to solve problems.

NOTE: There are three GMRES implementations: (i) gmresLP, (ii) fom, and (iii) gmresRP. GMRESLP is a restarted GMRES implementation that is left preconditioned and only checks the residual on the outer loops. This may be less efficient than GMRESRP, which can check both outer and inner loop residuals. However, GMRESRP has to use right preconditioning, which also slightly changes the convergence behavior of the linear system. GMRES with left preconditioning and without restarting will just build the full subspace by default, thus solving the system exactly, but may require too much memory. You can do a GMRESRP unrestarted by specifying that the restart parameter be equal to the size of the problem.

Basic Implementation Details:

Linear Solvers -> Solve Ax=b for x
Non-Linear Solvers -> Solve F(x)=0 for x
All implementations require system size to be 2 or greater

Author

Austin Ladshaw

Date

10/14/2014

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.15.2 Macro Definition Documentation

6.15.2.1 MIN_TOL

```
#define MIN_TOL 1e-15
```

Minimum Allowable Tolerance for linear and non-linear problems.

6.15.3 Enumeration Type Documentation

6.15.3.1 krylov_method

```
enum krylov_method
```

Enum of definitions for linear solver types in PJFNK.

Enum delineates the available Krylov Subspace methods that can be used to solve the linear sub-problem at each non-linear iteration in a Newton method.

Enumerator

GMRESLP	
PCG	
BiCGSTAB	
CGS	
FOM	
GMRESRP	
GCR	
GMRESR	
KMS	
QR	

6.15.4 Function Documentation

6.15.4.1 update_arnoldi_solution()

```
int update_arnoldi_solution (
             Matrix< double > & x,
             Matrix< double > & x0,
             ARNOLDI_DATA * arnoldi_dat )
```

Function to update the linear vector x based on the Arnoldi Krylov subspace.

This function will update a solution vector x based on the previous solution x0 given the orthonormal basis and upper Hessenberg matrix formed in the Arnoldi algorithm. Updating is automatically called by the GMRESLP function. It is expected that the Arnoldi algorithm has already been called prior to calling this function.

Parameters

x	matrix that will hold the new updated solution to the linear system
x0	matrix that holds the previous solution to the linear system
arnoldi_dat	pointer to the ARNOLDI_DATA data structure

6.15.4.2 arnoldi()

```
int arnoldi (
             int(*)(const Matrix< double > &v, Matrix< double > &w, const void *data) matvec,
             int(*)(const Matrix< double > &b, Matrix< double > &p, const void *data) precon,
             Matrix< double > & r0,
             ARNOLDI_DATA * arnoldi_dat,
             const void * matvec_data,
             const void * precon_data )
```

Function to factor a linear operator into an orthonormal basis and upper Hessenberg matrix.

This function performs the Arnoldi algorithm to factor a linear operator into an orthonormal basis and upper Hessenberg matrix. Each orthonormal vector is formed using a Modified Gram-Schmidt procedure. When used in conjunction with GMRESLP, user may supply a preconditioning operator to improve convergence of the linear system. However, this function can be used by itself to factor the user's linear operator.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
r0	user supplied vector to serve as the first basis vector in the orthonormal basis
arnoldi_dat	pointer to the ARNOLDI_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

--—

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

--—

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

--—

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

--—

6.15.4.3 gmresLeftPreconditioned()

```
int gmresLeftPreconditioned (
            int(*)(const Matrix< double > &v, Matrix< double > &w, const void *data) matvec,
            int(*)(const Matrix< double > &b, Matrix< double > &p, const void *data) precon,
            Matrix< double > & b,
            GMRESLP_DATA * gmreslp_dat,
            const void * matvec_data,
            const void * precon_data )
```

Function to iteratively solve a non-symmetric, indefinite linear system with GMRESLP.

This function iteratively solves a non-symmetric, indefinite linear system using the Generalized Minimum RE←Sidual method with Left Preconditioning (GMRESLP). It calls the Arnoldi algorithm to factor a linear operator into an orthonormal basis and upper Hessenberg matrix, then uses that factorization to form an approximation to the linear system. Because this algorithm uses left-side preconditioning, it can only check the linear residuals at the outer iterations.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmreslp_dat	pointer to the GMRESLP_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (*matvec) (const Matrix<double>& v, Matrix<double> &Av, const void *data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (*precon) (const Matrix<double>& b, Matrix<double> &Mb, const void *data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.15.4.4 fom()

```
int fom (
            int(*)(const Matrix< double > &v, Matrix< double > &w, const void *data) matvec,
            int(*)(const Matrix< double > &b, Matrix< double > &p, const void *data) precon,
            Matrix< double > & b,
            GMRESLP_DATA * gmreslp_dat,
            const void * matvec_data,
            const void * precon_data )
```

Function to directly solve a non-symmetric, indefinite linear system with FOM.

This function directly solves a non-symmetric, indefinite linear system using the Full Orthogonalization Method (F↩
OM). This algorithm is exactly equivalent to GMRESLP without restarting. Therefore, it uses the GMRESLP_DATA structure and calls the GMRESLP algorithm without using restarts. As a result, it never checks linear residuals. However, this should give the exact solution upon completion, assuming the linear operator is not singular.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmreslp_dat	pointer to the GMRESLP_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.15.4.5 gmresRightPreconditioned()

```
int gmresRightPreconditioned (
            int(*)(const Matrix< double > &v, Matrix< double > &w, const void *data) matvec,
            int(*)(const Matrix< double > &b, Matrix< double > &p, const void *data) precon,
            Matrix< double > & b,
            GMRESRP_DATA * gmresrp_dat,
            const void * matvec_data,
            const void * precon_data )
```

Function to iteratively solve a non-symmetric, indefinite linear system with GMRESRP.

This function iteratively solves a non-symmetric, indefinite linear system using the Generalized Minimum RESidual method with Right Preconditioning (GMRESRP). Because this algorithm uses right preconditioning, it is able to check the linear residuals at both the outer and inner iterations. This may be much for efficient compared to G↩
MRESLP. In order to check inner residuals, this algorithm has to perform it's own internal Modified Gram-Schmidt procedure and will not call the Arnoldi algorithm.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmresrp_dat	pointer to the GMRESRP_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.15.4.6 pcg()

```
int pcg (
            int(∗)(const Matrix< double > &p, Matrix< double > &Ap, const void ∗data) matvec,
            int(∗)(const Matrix< double > &r, Matrix< double > &z, const void ∗data) precon,
            Matrix< double > & b,
            PCG_DATA ∗ pcg_dat,
            const void ∗ matvec_data,
            const void ∗ precon_data )
```

Function to iteratively solve a symmetric, definite linear system with PCG.

This function iteratively solves a symmetric, definite linear system using the Preconditioned Conjugate Gradient (PCG) method. The PCG algorithm is optimal in terms of efficiency and residual reduction, but only if the linear system is symmetric. PCG will fail if the linear operator is non-symmetric!

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
pcg_dat	pointer to the PCG_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.15.4.7 bicgstab()

```
int bicgstab (
            int(∗)(const Matrix< double > &p, Matrix< double > &Ap, const void ∗data) matvec,
            int(∗)(const Matrix< double > &r, Matrix< double > &z, const void ∗data) precon,
            Matrix< double > & b,
            BiCGSTAB_DATA ∗ bicg_dat,
            const void ∗ matvec_data,
            const void ∗ precon_data )
```

Function to iteratively solve a non-symmetric, definite linear system with BiCGSTAB.

This function iteratively solves a non-symmetric, definite linear system using the Bi-Conjugate Gradient STABilized (BiCGSTAB) method. This is a highly efficient algorithm for solving non-symmetric problems, but will occassionally breakdown and fail. Most common failures are caused by poor preconditioning. Works very well for grid-based linear systems.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
bicg_dat	pointer to the BiCGSTAB_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

--—

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

--—

6.15.4.8 cgs()

```
int cgs (
            int(∗)(const Matrix< double > &p, Matrix< double > &Ap, const void ∗data) matvec,
            int(∗)(const Matrix< double > &r, Matrix< double > &z, const void ∗data) precon,
            Matrix< double > & b,
            CGS_DATA ∗ cgs_dat,
            const void ∗ matvec_data,
            const void ∗ precon_data )
```

Function to iteratively solve a non-symmetric, definite linear system with CGS.

This function iteratively solves a non-symmetric, definite linear system using the Conjugate Gradient Squared (CGS) method. This is an extremely efficient algorithm for solving non-symmetric problems, but will often breakdown and fail. Most common failures are caused by poor or no preconditioning. Works very will for grid-based linear systems.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
cgs_dat	pointer to the CGS_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

--—

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

--—

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

--—

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

--—

6.15.4.9 operatorTranspose()

```
int operatorTranspose (
            int(*)(const Matrix< double > &v, Matrix< double > &Av, const void *data) matvec,
            Matrix< double > & r,
            Matrix< double > & u,
            OPTRANS_DATA * transpose_dat,
            const void * matvec_data )
```

Function that is used to perform transposition of a linear operator and results in a new vector A^T∗r=u.

This function takes a user supplied linear operator and forms the result of that operator transposed and multiplied by a given vector r (A^T∗r=u). Transposition is accomplised by reordering the transpose operator and multiplying the non-transposed operator by a complete set of orthonormal vectors. The end result gives the ith component of the vector u for each operation (u_i = r^T∗A∗i). Here, i is a vector made from the ith column of the identity matrix. If the linear system if sufficiently large, then this operation may take some time.

Parameters

matvec	user supplied linear operator given as an int function
r	vector to be multiplied by the transpose of the operator
u	vector to store the result of the operator transposition (u=A^T∗r)
transpose_dat	pointer to the OPTRANS_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)
--—
This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.
--—

6.15.4.10 gcr()

```
int gcr (
            int(*)(const Matrix< double > &x, Matrix< double > &Ax, const void *data) matvec,
            int(*)(const Matrix< double > &r, Matrix< double > &Mr, const void *data) precon,
            Matrix< double > & b,
            GCR_DATA * gcr_dat,
            const void * matvec_data,
            const void * precon_data )
```

Function to iteratively solve a non-symmetric, definite linear system with GCR.

This function iteratively solves a non-symmetric, definite linear system using the Generalized Conjugate Residual (GCR) method. Similar to GMRESRP, this algorithm will construct a growing orthonormal basis set that will eventually form the exact solution to the linear system. However, this algorithm is less efficient than GMRESRP and can suffer breakdowns if the linear system is indefinite.

Parameters

matvec	user supplied linear operator given as an int function
precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gcr_dat	pointer to the GCR_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

---——

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

---——

int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

--—

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

--—

6.15.4.11 gmresrPreconditioner()

```
int gmresrPreconditioner (
            const Matrix< double > & r,
            Matrix< double > & Mr,
            const void * data )
```

Function used in conjunction with GMRESR to apply GMRESRP iterations as a preconditioner.

This function is required to take the form of the user supplied preconditioning functions for other iterative methods. However, it cannot be used in conjunction with any other Krylov method. It is only called by the GMRESR function when the preconditioner needs to be applied.

Parameters

r	vector supplied to the preconditioner to operate on
Mr	vector to hold the result of the preconditioning operation
data	void pointer to the GMRESR_DATA data structure

6.15.4.12 gmresr()

```
int gmresr (
            int (*) (const Matrix< double > &x, Matrix< double > &Ax, const void *data) matvec,
            int (*) (const Matrix< double > &r, Matrix< double > &Mr, const void *data) terminal↩
_precon,
            Matrix< double > & b,
            GMRESR_DATA * gmresr_dat,
            const void * matvec_data,
            const void * term_precon_data )
```

Function to iteratively solve a non-symmetric, indefinite linear system with GMRESR.

This function iteratively solves a non-symmetric, indefinite linear system using the Generalized Minimum RE↩ Sidual Recursive (GMRESR) method. This algorithm actually uses GCR at the outer iterations, but stabilizes GCR with GMRESRP inner iterations to implicitly form a variable preconditioner to the linear system. As such, this is one of only two methods that inherently includes preconditioning (the other is KMS), without any user supplied preconditioning operator. However, this algorithms is signficantly more computationally expensive than GCR or GMRESRP separately. It should only be used for solving very large or very hard to solve linear systems.

Parameters

matvec	user supplied linear operator given as an int function
terminal_precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
gmresr_dat	pointer to the GMRESR_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
term_precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (∗terminal_precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.15.4.13 kmsPreconditioner()

```
int kmsPreconditioner (
            const Matrix< double > & r,
            Matrix< double > & Mr,
            const void * data )
```

Preconditioner function for the Krylov Multi-Space.

This function is required to take the form of the user supplied preconditioning functions for other iterative methods. However, it cannot be used in conjunction with any other Krylov method. It is only called by the KMS function when the preconditioner needs to be applied.

Parameters

r	vector supplied to the preconditioner to operate on
Mr	vector to hold the result of the preconditioning operation
data	void pointer to the KMS_DATA data structure

6.15.4.14 krylovMultiSpace()

```
int krylovMultiSpace (
            int(*)(const Matrix< double > &x, Matrix< double > &Ax, const void *data) matvec,
            int(*)(const Matrix< double > &r, Matrix< double > &Mr, const void *data) terminal←
_precon,
            Matrix< double > & b,
            KMS_DATA * kms_dat,
            const void * matvec_data,
            const void * term_precon_data )
```

Function to iteratively solve a non-symmetric, indefinite linear system with KMS.

This function iteratively solves a non-symmetric, indefinite linear system using the Krylov Multi-Space (KMS) method. This algorithm uses GMRESRP at both outer and inner iterations to implicitly form a variable preconditioner to the linear system. As such, this is one of only two methods that inherently includes preconditioning, without any user supplied preconditioning operator (the other being GMRESR). The advantage to this method over GMRESR is that this method is GMRES at its core, and will therefore never breakdown or need to be stabilized. Additionally, you can call this method and set it's max_level parameter (see KMS_DATA) to 0, which will make this algorithm exactly equal to GMRESRP. If the max_level is set to 1, then this algorithm is exactly FGMRES (Saad, 1993) with the GMRES algorithm as a preconditioner. However, you can set max_level higher to precondition the preconditioners with more preconditioners. Thus creating a method with any desired complexity or rate of convergence.

Parameters

matvec	user supplied linear operator given as an int function
terminal_precon	user supplied preconditioning operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
kms_dat	pointer to the KMS_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator
term_precon_data	user supplied void pointer to a data structure needed for the precondtioning operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)

This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.

----

int (∗terminal_precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the above linear operator function and should have all the same properties. The only difference is that this function must form

an approximate matrix inversion on the original linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.
---—

6.15.4.15 QRsolve()

```
int QRsolve (
            int(*)(const Matrix< double > &x, Matrix< double > &Ax, const void *data) matvec,
            Matrix< double > & b,
            QR_DATA * qr_dat,
            const void * matvec_data )
```

Function to solve a dense linear operator system using QR factorization.

This function is used to solve a dense linear system using QR factorization. It should only be used if iterative methods are unstable or if the linear system is very dense. There will likely be memory limitations to using this method, since it is assumed that the matrix/operator is dense. This method may also be less efficient because it has to extract the matrix elements from the linear operator. So if the linear operator is large, then the setup cost for this method is high.

Factorization is carried out using Householder Reflections. Each reflection matrix is iteratively applied to the operator and the vector b to convert the linear system to upper triangular. Then, the system is solved using backwards substitution.

Parameters

matvec	user supplied linear operator given as an int function
b	matrix of boundary conditions in the linear system Ax=b
qr_dat	pointer to the QR_DATA data structure
matvec_data	user supplied void pointer to a data structure needed for the linear operator

Note

int (∗matvec) (const Matrix<double>& v, Matrix<double> &Av, const void ∗data)
---—
This is a user supplied function for a linear operator. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix v that will act on the linear operator a modified the matrix entries of Av to form the result of a matrix-vector product. Void pointer data is used to pass any user data structure that the function may need in order to perform the linear operation.
---—

6.15.4.16 picard()

```
int picard (
            int(*)(const Matrix< double > &x, Matrix< double > &r, const void *data) res,
            int(*)(const Matrix< double > &x0, Matrix< double > &x, const void *data) evalx,
            Matrix< double > & x,
            PICARD_DATA * picard_dat,
            const void * res_data,
            const void * evalx_data )
```

Function to iteratively solve a non-linear system using the Picard or Fixed-Point method.

This function iteratively solves a non-linear system using the Picard method. User supplies a residual function and a weak solution form function. The weak form function is used to approximate the next solution vector for the non-linear system and the residual function is used to determine convergence. User also supplies an initial guess to the non-linear system as a matix x, which will also be used to store the solution. This algorithm is very simple and may not be sufficient to solve complex non-linear systems.

Parameters

res	user supplied function for the non-linear residuals of the system
evalx	user supplied function for the weak form to estimate the next solution
x	user supplied matrix holding the initial guess to the non-linear system
picard_dat	pointer to the PICARD_DATA data structure
res_data	user supplied void pointer to a data structure used for residual evaluations
evalx_data	user supplied void pointer to a data structure used for evaluation of weak form

Note

int (∗res) (const Matrix<double>& x, Matrix<double> &F, const void ∗data)
--—
This is a user supplied function for the non-linear residuals. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix x representing the current non-linear variables. Those variables are used to evaluate the users functions and return the residuals in the matrix F. The void pointer data is a data structure provided by the user to hold information the function may need in order to form the residuals.
--—
int (∗evalx) (const Matrix<double>& x0, Matrix<double> &x, const void ∗data)
--—
This is a user supplied function to approximate the next solution vector x based on the previous solution vector x0. The x0 matrix is passed to this function and must be used to edit the entries of x based on the weak form of the problem. The user is free to define any weak form approximation. Void pointer data is the users data structure that may be used to pass additional information into this function in order to evaluate the weak form.
Example Residual: F(x) = x^2 + x - 1 Goal is to make this function equal zero
Example Weak Form: x = 1 - x0^2 Rearrage residual to form a weak solution
--—

6.15.4.17 jacvec()

```
int jacvec (
            const Matrix< double > & v,
            Matrix< double > & Jv,
            const void * data )
```

Function to form a linear operator of a Jacobian matrix used along with the PJFNK method.

This function is used in conjunction with the PJFNK routine to form a linear operator that a Krylov method can operate on. This linear operator is formed from the current residual vector of the non-linear iteration in PJFNK using a finite difference approximation.
Jacobian Linear Operator: J∗v = (F(x_k + eps∗v) - F(x_k)) / eps

Parameters

v	vector to be multiplied by the Jacobian matrix
Jv	storage vector for the result of the Jacobi-vector product
data	void pointer to the PJFNK_DATA data structure holding solver information

6.15.4.18 backtrackLineSearch()

```
int backtrackLineSearch (
            int(*)(const Matrix< double > &x, Matrix< double > &F, const void *data) feval,
            Matrix< double > & Fkp1,
            Matrix< double > & xkp1,
            Matrix< double > & pk,
            double normFk,
            BACKTRACK_DATA * backtrack_dat,
            const void * feval_data )
```

Function to perform a Backtracking Line Search operation to smooth out convergence of PJFNK.

This function performs a simple backtracking line search operation on the residuals from the PJFNK method. The step size of the non-linear iteration is checked against a level of tolerance for residual reduction, then adjusted down if necessary. This method always starts out with the maximum allowable step size. If the largest step size is fine, then the algorithm does nothing. Otherwise, it iteratively adjusts the step size down, until a suitable step is found. In the case that no suitable step is found, this algorithm will report failure to the PJFNK method and PJFNK will decide whether to continue trying to find a global minimum or report that it is stuck in a local minimum.

Parameters

feval	user supplied residual function for the non-linear system
Fkp1	vector holding the residuals for the next non-linear step
xkp1	vector holding the solution for the next non-linear step
pk	vector holding the current non-linear search direction
normFk	value of the current non-linear residual
backtrack_dat	pointer to the BACKTRACK_DATA data structure
feval_data	user supplied void pointer to the data structure needed for residual evaluation

Note

int (∗feval) (const Matrix<double>& x, Matrix<double> &F, const void ∗data)

--—

This is a user supplied function for the non-linear residuals. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix x representing the current non-linear variables. Those variables are used to evaluate the users functions and return the residuals in the matrix F. The void pointer data is a data structure provided by the user to hold information the function may need in order to form the residuals.

--—

6.15.4.19 pjfnk()

```
int pjfnk (
            int(*)(const Matrix< double > &x, Matrix< double > &F, const void *data) res,
            int(*)(const Matrix< double > &r, Matrix< double > &p, const void *data) precon,
            Matrix< double > & x,
            PJFNK_DATA * pjfnk_dat,
            const void * res_data,
            const void * precon_data )
```

Function to perform the PJFNK algorithm to solve a non-linear system of equations.

This function solves a non-linear system of equations using the Preconditioned Jacobian- Free Newton-Krylov (P↩JFNK) algorithm. Each non-linear step of this method results in a linear sub-problem that is solved iteratively with one of the Krylov methods in the krylov_method enum. User must supplied a residual function that computes the non-linear residuals of the system given the current state of the variables x. Additionally, the user must also supplied an initial guess to the non-linear system. Optionally, the user may supply a preconditioning function for the linear sub-problem.
Basic Steps: (i) Calc F(x_k), (ii) Solve J(x_k)∗s_k=-F(x_k) for s_k, (iii) Form x_kp1 = x_k + s_k

Parameters

res	user supplied residual function for the non-linear system
precon	user supplied preconditioning function for the linear sub-problems
x	user supplied initial guess and storage location of the solution
pjfnk_dat	pointer to the PJFNK_DATA data structure
res_data	user supplied void pointer to data structure used in residual function
precon_data	user supplied void pointer to data structure used in preconditioning function

Note

int (∗res) (const Matrix<double>& x, Matrix<double> &F, const void ∗data)

This is a user supplied function for the non-linear residuals. User's function must return an int of 0 upon success and anything else denotes a failure. The function accepts a matrix x representing the current non-linear variables. Those variables are used to evaluate the users functions and return the residuals in the matrix F. The void pointer data is a data structure provided by the user to hold information the function may need in order to form the residuals.
----
int (∗precon) (const Matrix<double>& b, Matrix<double> &Mb, const void ∗data)

This is a user supplied function for a preconditioning operator. It has the same form as the linear operators from the Krylov methods and should have all the same properties. The only difference is that this function must form an approximate matrix inversion on the jacvec linear operator and modify the entries of Mb to represent the result of that approximate matrix inversion. The matrix b is given as the vector that this operator is acting on and the void pointer data is for any user data structure that the operator may need.

6.15.4.20 NumericalJacobian()

```
int NumericalJacobian (
            int(*)(const Matrix< double > &x, Matrix< double > &F, const void *user_data)
Func,
            const Matrix< double > & x,
            Matrix< double > & J,
            int Nx,
            int Nf,
            NUM_JAC_DATA * jac_dat,
            const void * user_data )
```

Function to form a full numerical Jacobian matrix from a given non-linear function.

This function uses finite differences to form a full rank Jacobian matrix for a user supplied non-linear function. The Jacobian matrix will be formed at the current state of the non-linear variables x and stored in a full matrix J. Integers Nx and Nf are used to determine the size of the Jacobian matrix.

Parameters

Func	user supplied function for evaluation of the non-linear system
x	matrix holding the current value of the non-linear variables
J	matrix that will store the numerical Jacobian result
Nx	number of non-linear variables in the system
Nf	number of non-linear functions in the system
jac_dat	pointer to the NUM_JAC_DATA data structure
user_data	user supplied void pointer to a data structure used in the non-linear function

6.15.4.21 LARK_TESTS()

```
int LARK_TESTS ( )
```

Function that runs a variety of tests on all the functions in LARK.

This function runs a variety of tests on the linear and non-linear methods developed in LARK. It can be called from the UI.

6.16 macaw.h File Reference

MAtrix CAlculation Workspace.

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include <exception>
#include "error.h"
```

Classes

- class Matrix< T >

  Templated C++ Matrix Class Object (click Matrix to go to function definitions)

Macros

- #define M_PI 3.14159265358979323846264338327950288

  Value of PI with double precision.

Functions

- int MACAW_TESTS ()

  Function to run the MACAW tests.

6.16.1 Detailed Description

MAtrix CAlculation Workspace.

macaw.cpp

This is a small C++ library that faciltates the use and construction of real matrices using vector objects. The Matrix class is templated so that users are able to work with matrices of any type including, but not limited to: (i) doubles, (ii) ints, (iii) floats, and (iv) even other matrices! Routines and functions are defined for Dense matrix operations. As a result, we typically only use Column Matrices (or Vectors) when doing any actual simulations. However, the development of this class was integral to the development and testing of the Sparse matrix operators in lark.h.

While the primary goal of this object was to define how to operate on real matrices, we could extend this idea to complex matrices as well. For this, we could develop objects that represent imaginary and complex numbers and then create a Matrix of those objects. For this reason, the matrix operations here are all templated to abstract away the specificity of the type of matrix being operated on.

Author

Austin Ladshaw

Date

01/07/2014

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.16.2 Macro Definition Documentation

6.16.2.1 M_PI

```
#define M_PI 3.14159265358979323846264338327950288
```

Value of PI with double precision.

6.16.3 Function Documentation

6.16.3.1 MACAW_TESTS()

```
int MACAW_TESTS ( )
```

Function to run the MACAW tests.

This function is callable from the UI and is used to run several algorithm tests for the Matrix objects. This test should never report any errors.

6.17 magpie.h File Reference

Multicomponent Adsorption Generalized Procedure for Isothermal Equilibria.

```
#include "lmcurve.h"
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <time.h>
#include <float.h>
#include <string>
#include "error.h"
#include "lark.h"
```

Classes

- struct GSTA_DATA

  GSTA Data Structure.
- struct mSPD_DATA

  MSPD Data Structure.
- struct GPAST_DATA

  GPAST Data Structure.
- struct SYSTEM_DATA

  System Data Structure.
- struct MAGPIE_DATA

  MAGPIE Data Structure.

Macros

- #define DBL_EPSILON 2.2204460492503131e-016

  Machine precision value used for approximating gradients.
- #define Z 10.0

  Surface coordination number used in the MSPD activity model.
- #define A 3.13E+09

  Corresponding van der Waals standard area for our coordination number (cm^2/mol)
- #define V 18.92

  Corresponding van der Waals standard volume for our coordination number (cm^3/mol)
- #define Po 100.0

  Standard State Pressure - Units: kPa.
- #define R 8.3144621

  Gas Constant - Units: J/(K∗mol) = kB ∗ Na.
- #define Na 6.0221413E+23

  Avagadro's Number - Units: molecules/mol.
- #define kB 1.3806488E-23

  Boltzmann's Constant - Units: J/K.
- #define shapeFactor(v_i) (((Z - 2) ∗ v_i) / (Z ∗ V)) + (2 / Z)

  This macro replaces all instances of shapeFactor(#) with the following single line calculation.
- #define lnKo(H, S, T) -(H / (R ∗ T)) + (S / R)

  This macro calculates the natural log of the dimensionless isotherm parameter.
- #define He(qm, K1, m) (qm ∗ K1) / (m ∗ Po)

  This macro calculates the Henry's Coefficient for the ith component.

Functions

- double qo (double po, const void ∗data, int i)

  Function computes the result of the GSTA isotherm for the ith species.
- double dq_dp (double p, const void ∗data, int i)

  Function computes the derivative of the GSTA model with respect to partial pressure.
- double q_p (double p, const void ∗data, int i)

  Function computes the ratio between the adsorbed amount and partial pressure for the GSTA isotherm.
- double PI (double po, const void ∗data, int i)

  Function computes the spreading pressure integral of the ith species.
- double Qst (double po, const void ∗data, int i)

  Function computes the heat of adsorption based on the ith species GSTA parameters.
- double eMax (const void ∗data, int i)

  Function to approximate the maximum lateral energy term for the ith species.
- double lnact_mSPD (const double ∗par, const void ∗data, int i, volatile double PI)

  Function to evaluate the MSPD activity coefficient for the ith species.
- double grad_mSPD (const double ∗par, const void ∗data, int i)

  Function to approximate the derivative of the MSPD activity model with spreading pressure.
- double qT (const double ∗par, const void ∗data)

  Function to calculate the total adsorbed amount (mol/kg) for the mixed surface phase.
- void initialGuess_mSPD (double ∗par, const void ∗data)

  Function to provide an initial guess to the unknown parameters being solved for in GPAST.
- void eval_po_PI (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function used with lmfit to evaluate the reference state pressure of a species based on spreading pressure.
- void eval_po_qo (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

Function used with lmfit to evaluate the reference state pressure of a species based on that species isotherm.

- void eval_po (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function used with lmfit to evaluate the reference state pressure of a species based on a sub-system.
- void eval_eta (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function used with lmfit to evaluate the binary interaction parameters for each unique species pair.
- void eval_GPAST (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function used with lmfit to solve the GPAST system of equations.
- int MAGPIE (const void ∗data)

  Function to call all sub-routines to solve a MAGPIE/GPAST problem at a given temperature and pressure.
- int MAGPIE_SCENARIOS (const char ∗inputFileName, const char ∗sceneFileName)

  Function to perform a series of MAGPIE simulations based on given input files.

6.17.1 Detailed Description

Multicomponent Adsorption Generalized Procedure for Isothermal Equilibria.

magpie.cpp

This file contains all functions and routines associated with predicting isothermal adsorption equilibria from only single component isotherm information. The basis of the model is the Adsorbed Solution Theory developed by Myers and Prausnitz (1965). Added to that base model is a procedure by which we can predict the non-idealities present at the surface phase by solving a closed system of equations involving the activity model.

For more details on this procedure, check out our publication in AIChE where we give a fully feature explaination of our Generalized Predictive Adsorbed Solution Theory (GPAST).

Reference: Ladshaw, A., Yiacoumi, S., and Tsouris, C., "A generalized procedure for the prediction of multicomponent adsorption equilibria", AIChE J., vol. 61, No. 8, p. 2600-2610, 2015.

MAGPIE represents a special case of the more general GPAST procedure, wherin the isotherm for each species is respresent by the GSTA isotherm (see gsta_opt.h) and the activity model for non-ideality at the adsorbent surface is a Modified Spreading Pressure Dependent (MSPD) model. See the above paper reference for more details.

Author

Austin Ladshaw

Date

12/17/2013

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.17.2 Macro Definition Documentation

6.17.2.1 DBL_EPSILON

```
#define DBL_EPSILON 2.2204460492503131e-016
```

Machine precision value used for approximating gradients.

6.17.2.2 Z

```
#define Z 10.0
```

Surface coordination number used in the MSPD activity model.

6.17.2.3 A

```
#define A 3.13E+09
```

Corresponding van der Waals standard area for our coordination number (cm^2/mol)

6.17.2.4 V

```
#define V 18.92
```

Corresponding van der Waals standard volume for our coordination number (cm^3/mol)

6.17.2.5 Po

```
#define Po 100.0
```

Standard State Pressure - Units: kPa.

6.17.2.6 R

```
#define R 8.3144621
```

Gas Constant - Units: J/(K∗mol) = kB ∗ Na.

6.17.2.7 Na

```
#define Na 6.0221413E+23
```

Avagadro's Number - Units: molecules/mol.

6.17.2.8 kB

```
#define kB 1.3806488E-23
```

Boltzmann's Constant - Units: J/K.

6.17.2.9 shapeFactor

```
#define shapeFactor(
              v_i ) ( ( (Z - 2) * v_i ) / ( Z * V ) ) + ( 2 / Z )
```

This macro replaces all instances of shapeFactor(#) with the following single line calculation.

6.17.2.10 lnKo

```
#define lnKo(
              H,
              S,
              T ) -( H / ( R * T ) ) + ( S / R )
```

This macro calculates the natural log of the dimensionless isotherm parameter.

6.17.2.11 He

```
#define He(
              qm,
              K1,
              m ) ( qm * K1 ) / ( m * Po )
```

This macro calculates the Henry's Coefficient for the ith component.

6.17.3 Function Documentation

6.17.3.1 qo()

```
double qo (
             double po,
             const void * data,
             int i )
```

Function computes the result of the GSTA isotherm for the ith species.

This function just computes the result of the GSTA isotherm model for the ith species given the partial pressure po.

Parameters

po	partial pressure in kPa at which to evaluate the GSTA model
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.17.3.2 dq_dp()

```
double dq_dp (
             double p,
             const void * data,
             int i )
```

Function computes the derivative of the GSTA model with respect to partial pressure.

This function just computes the result of the derivative of GSTA isotherm model for the ith species at the given the partial pressure p.

Parameters

p	partial pressure in kPa at which to evaluate the GSTA model
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.17.3.3 q_p()

```
double q_p (
             double p,
             const void * data,
             int i )
```

Function computes the ratio between the adsorbed amount and partial pressure for the GSTA isotherm.

This function just computes the ratio between the adsorbed amount q (mol/kg) and the partial pressure p (kPa) at the given partial pressure. If p == 0, then this function returns the Henry's Law constant for the isotherm of the ith species.

Parameters

p	partial pressure in kPa at which to evaluate the GSTA model
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.17.3.4 PI()

```
double PI (
            double po,
            const void * data,
            int i )
```

Function computes the spreading pressure integral of the ith species.

This function uses an analytical solution to the spreading pressure integral with the GSTA isotherm to evaluate and return the value computed by that integral equation.

Parameters

po	partial pressure in kPa at which to evaluate the lumped spreading pressure
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.17.3.5 Qst()

```
double Qst (
            double po,
            const void * data,
            int i )
```

Function computes the heat of adsorption based on the ith species GSTA parameters.

This function computes the isosteric heat of adsorption (J/mol) for the GSTA parameters of the ith species.

Parameters

po	partial pressure in kPa at which to evaluate the heat of adsorption
data	void pointer to the MAGPIE_DATA data structure
i	index of the gas species for which the GSTA model is being evaluated

6.17.3.6 eMax()

```
double eMax (
            const void * data,
            int i )
```

Function to approximate the maximum lateral energy term for the ith species.

The function attempts to approximate the maximum lateral energy term for the ith species. This is not a true maximum, but a cheaper estimate. Value being computed is used to shift the geometric mean and formulate the average cross-lateral energy term between species i and j.

6.17.3.7 lnact_mSPD()

```
double lnact_mSPD (
             const double * par,
             const void * data,
             int i,
             volatile double PI )
```

Function to evaluate the MSPD activity coefficient for the ith species.

This function will return the natural log of the ith species activity coefficient using the Modified Spreading Pressure Dependent (MSPD) activity model. The par argument holds the variable values being solved for by GPAST and their contents will change depending on whether we are doing a forward or reverse evaluation. This function should not be called by the user and will only be called when needed in the GPAST routine.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure
i	ith species that we want to calculate the activity coefficient for
PI	lumped spreading pressure term used in gradient estimations

6.17.3.8 grad_mSPD()

```
double grad_mSPD (
             const double * par,
             const void * data,
             int i )
```

Function to approximate the derivative of the MSPD activity model with spreading pressure.

This function returns a 2nd order, finite different approximation of the derivative of the MSPD activity model with the spreading pressure. The par argument will either hold the current iterates estimate of spreading pressure or should be passed as null. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure
i	ith species for which we will approximate the activty model gradient

6.17.3.9 qT()

```
double qT (
             const double * par,
             const void * data )
```

Function to calculate the total adsorbed amount (mol/kg) for the mixed surface phase.

This function will uses the obtained system parameters from par and estimate the total amount of gases adsorbed to the surface in mol/kg. The user does not need to call this function, since this result will be stored in the SYST←
EM_DATA structure.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure

6.17.3.10 initialGuess_mSPD()

```
void initialGuess_mSPD (
            double * par,
            const void * data )
```

Function to provide an initial guess to the unknown parameters being solved for in GPAST.

This function intends to provide an initial guess for the unknown values being solved for in the GPAST system. Depending on what type of solve is requested, this algorithm will provide a guess for the adsorbed or gas phase composition.

Parameters

par	list of parameters representing variables to be solved for in GPAST
data	void pointer for the MAGPIE_DATA data structure

6.17.3.11 eval_po_PI()

```
void eval_po_PI (
            const double * par,
            int m_dat,
            const void * data,
            double * fvec,
            int * info )
```

Function used with lmfit to evaluate the reference state pressure of a species based on spreading pressure.

This function is used inside of the MSPD activity model to calculate the reference state pressure of a particular species at a given spreading pressure for the system. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.17.3.12 eval_po_qo()

```
void eval_po_qo (
             const double * par,
             int m_dat,
             const void * data,
             double * fvec,
             int * info )
```

Function used with lmfit to evaluate the reference state pressure of a species based on that species isotherm.

This function is used to evaluate the partial pressure or reference state pressure for a particular species given single-component adsorbed amount. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.17.3.13 eval_po()

```
void eval_po (
             const double * par,
             int m_dat,
             const void * data,
             double * fvec,
             int * info )
```

Function used with lmfit to evaluate the reference state pressure of a species based on a sub-system.

This function is used to approximate reference state pressures based on the spreading pressure of a sub-system in GPAST. The sub-system will be one of the unique binary systems that exist in the overall mixed gas system. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.17.3.14 eval_eta()

```
void eval_eta (
             const double * par,
             int m_dat,
             const void * data,
             double * fvec,
             int * info )
```

Function used with lmfit to evaluate the binary interaction parameters for each unique species pair.

This function is used to estimate the binary interaction parameters for all species pairs in a given sub-system. Those parameters are then stored for later used when evaluating the activity coefficients for the overall mixture. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.17.3.15 eval_GPAST()

```
void eval_GPAST (
             const double * par,
             int m_dat,
             const void * data,
             double * fvec,
             int * info )
```

Function used with lmfit to solve the GPAST system of equations.

This function is used after having calculated and stored all necessary information to solve a closed form GPAST system of equations. User does not need to call this function. GPAST will call automatically when needed.

Parameters

par	list of parameters representing variables to be solved for in GPAST
m_dat	number of functions/variables in the GPAST system of equations
data	void pointer for the MAGPIE_DATA data structure
fvec	list of residuals formed by the functions in GPAST
info	integer flag variable used in the lmfit routine

6.17.3.16 MAGPIE()

```
int MAGPIE (
             const void * data )
```

Function to call all sub-routines to solve a MAGPIE/GPAST problem at a given temperature and pressure.

This is the function that a typical user will want to incorporate into their own codes when evaluating adsorption of a gas mixture. Prior to calling this function, all required structures and information in the MAGPIE_DATA structure must have been properly initialized. After this function has completed it's operations, it will return an integer used to denote a success or failure of the routine. Integers 0, 1, 2, and 3 all denote success. Anything else is considered a failure.

To setup the MAGPIE_DATA structure correctly, you must reserve space for all vector objects based on the number of gas species in the mixture. In general, you only need to reserve space for the adsorbing species. However, you can also reserve space for non-adsorbing species, but you MUST give a gas/adsorbed mole fraction of the non-adsorbing species 0.0 so that the routine knows to ignore them (very important)!

After setting up the memory for the vector objects, you can intialize information specific to the simulation you want to request. The number of species (N), total pressure (PT) and gas temperature (T) must always be given. You can neglect the non-idealities of the surface phase by setting the Ideal bool to true. This will result in faster calculations, because MAGPIE will just revert down to the Ideal Adsorbed Solution Theory (IAST).

The Recover bool will denote whether we are doing a forward or reverse GPAST evaluation. Forward evaluation is for solving for the composition of the adsorbed phase given the composition of the gas phase (Recover = false). Reverse evaluation is for solve for the composition of the gas phase given the composition of the adsorbed phase (Recover = true).

For a reverse evaluation (Recover = true) you will also need to stipulate whether or not there is a carrier gas (Carrier = true or false). A carrier gas is considered any non-adsorbing species that may be present in the gas phase and contributing to the total pressure in the system.

The parameters that must be initialized for all species include all GSTA_DATA parameters and the van der Waals volume parameter (v) in the mSPD_DATA structure. For non-adsorbing species, you can ignore these parameters, but need to set the sites (m) from GSTA_DATA to 1. GPAST cannot run any evaluations without these parameters being set properly AND set in the same order for all species (i.e., make sure that gpast_dat[i].qmax corresponds to mspd_dat[i].v and so on).

Lastly, you need to give either the gas phase or adsorbed phase mole fractions, depending on whether you are going to run a forward or reverse evaluation, respectively. For a forward evaluation, provide the gas mole fractions (y) in GPAST_DATA for each species (non-adsorbing species should have this value set to 0.0). For a reverse evaluation, provide the adsorbed mole fractions (x) in GPAST_DATA for each species, as well as the total adsorbed amount (qT) in SYSTEM_DATA. Again, non-adsorbing species should have their respective phase mole fractions set to 0.0 to exclude them from the simulation. Additionally, if there are non-adsorbing species present, then the Carrier bool in SYSTEM_DATA must be set to true.

Parameters

data	void pointer for the MAGPIE_DATA data structure holding all necessary information

6.17.3.17 MAGPIE_SCENARIOS()

```
int MAGPIE_SCENARIOS (
            const char * inputFileName,
            const char * sceneFileName )
```

Function to perform a series of MAGPIE simulations based on given input files.

This function is callable from the UI and is used to perform a series of isothermal equilibria evaluations using the MAGPIE routines. There are two input files that must be provided: (i) inputFileName - containing parameter information for the species and (ii) sceneFileName - containing information for each MAGPIE simulation. Each of these files have a specific structure (see below). NOTE: this may change in future versions.

inputFileName Text File Structure:

Integer for Number of Adsorbing Species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat above for all n sites in species i)
(repeat above for all species i)

Example Input File:

5
17.1
5.8797
1
-20351.9 -81.8369
16.2
5.14934
1
-16662.7 -74.4766
19.7
9.27339
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
13.25
4.59144
1
-13418.5 -84.888
18.0
10.0348
1
-20640.4 -72.6119
(The above input file gives the parameter information for 5 adsorbing species)

sceneFileName Text File Structure:

Integer Flag to mark Forward (0) or { Reverse (1) evaluations }
Number of Simulations to Run
Total Pressure (kPa) [tab] Temperature (K) { [tab] Total Adsorption (mol/kg) [tab] Carrier Gas Flag (0=false, 1=true) }
Gas/Adsorbed Mole Fractions for each species in the order given in prior file (tab separated)
(repeat above for all simulations desired)
NOTE: only provide the Total Adsorption and Carrier Flag if doing Reverse evaluations!

Example Scenario File 1:

0
4
0.65 303.15

0.364 0.318 0.318
3.25 303.15
0.371 0.32 0.309
6.85 303.15
0.388 0.299 0.313
13.42 303.15
0.349 0.326 0.325
(The above scenario file is for 4 forward evaluations/simulations for a 3-adsorbing species system)

Example Scenario File 2:

1
4
0.65 303.15 5.4 0
0.364 0.318 0.318
3.25 303.15 7.7 0
0.371 0.32 0.309
6.85 303.15 9.8 0
0.388 0.299 0.313
13.42 303.15 10.4 0
0.349 0.326 0.325
(The above scenario file is for 4 reverse evaluations/simulations for a 3-adsorbing species system and no carrier gas)

6.18 mesh.h File Reference

Mesh Objects and Associated Sub-Objects.

```
#include "macaw.h"
```

Classes

- class Vector3D

  3D Vector Object
- class Node

  Node object.
- class LineElement

  LineElement.
- class SurfaceElement

  SurfaceElement Object.
- class VolumeElement

  VolumeElement Object.
- class NodeSet

  NodeSet Object.

Enumerations

- enum element_type { BOUNDARY, INTERIOR }

  Enumeration for the list of valid element types.

Functions

- int MESH_TESTS ()

    Test function for MESH kernel.

6.18.1 Detailed Description

Mesh Objects and Associated Sub-Objects.

This kernel allows for creation of mesh objects from constitutient sub-objects such as Nodes and Elements (Line, Surface, or Volume). Mesh objects can be used to establish physical-chemcial simulations in multi-dimensional space.

Warning

This kernel is still under active development. Use with caution!

Author

Austin Ladshaw

Date

04/09/2018

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for Post-Doc research in the area of adsorption and surface science. Copyright (c) 2018, all rights reserved.

6.18.2 Enumeration Type Documentation

6.18.2.1 element_type

```
enum element_type
```

Enumeration for the list of valid element types.

The only types that have been defined are for Boudary and Interior elements.

Enumerator

BOUNDARY	
INTERIOR	

6.18.3 Function Documentation

6.18.3.1 MESH_TESTS()

```
int MESH_TESTS ( )
```

Test function for MESH kernel.

This function runs tests on the mesh objects.

6.19 mola.h File Reference

Molecule Object Library from Atoms.

```
#include <ctype.h>
#include "eel.h"
```

Classes

- class Molecule

  C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

Macros

- #define M_PI 3.14159
- #define SphereVolume(r) ((4.0/3.0)∗M_PI∗r∗r∗r)
- #define SphereArea(r) (4.0∗M_PI∗r∗r)

Enumerations

- enum valid_phase {
  SOLID, LIQUID, AQUEOUS, GAS,
  PLASMA, ADSORBED, OTHER }

Functions

- int MOLA_TESTS ()

  Function to run the MOLA tests.

6.19.1 Detailed Description

Molecule Object Library from Atoms.

mola.cpp

This file contains a C++ Class for creating Molecule objects from the Atom objects that were defined in eel.↩ h. Molecules can be created and registered from basic information or can be registered from a growing list of pre-registered molecules that are accessible by name/formula.

Registered Molecules are are known and defined prior to runtime. They have a charge, energy characteristics, phase, name, and formula that they are recongized by. The formula is used to create the atoms that they are made from. If some information is incomplete, it must be specified as to what information is missing (i.e. denote whether the formation energies are known).

Formation energies are used to determine stability/dissociation/acidity equilibrium constants during runtime. If the formation energies are unknown, then the equilibrium constants must be given to a reaction object on when it is initialized.

The molecule formula's are given as strings which are parsed in the constructor to determine what atoms from the EEL files will be registered and used. Note, you will be able to build molecules from an input file, but the library molecules here are ready to be used in applications and require no more input other that the molecule's formula.

List of Currently Registered Molecules

Ag (s)
Ag + (aq)
AgBr (s)
AgCl (s)
AgI (s)
Ag2S (s)
AgOH (aq)
Ag(OH)2 - (aq)
AgCl (aq)
AgCl2 - (aq)
Al (s)
Al 3+ (aq)
AlOH 2+ (aq)
Al(OH)2 + (aq)
Al(OH)3 (aq)
Al(OH)4 - (aq)
Al2O3 (s)
AlOOH (s)
Al(OH)3 (s)
Al2Si2(OH)4 (s)
As (s)
AsO4 3- (aq)
Ba 2+ (aq)
BaSO4 (s)
BaCO3 (s)
Be 2+ (aq)
Be(OH)2 (s)
Be3(OH)3 3+ (aq)
B(OH)4 - (aq)
Br2 (l)
Br2 (aq)
Br - (aq)

Cr2O3 (s)
CrO4 2- (aq)
Cr2O7 2- (aq)
Cu (s)
Cu + (aq)
Cu 2+ (aq)
CuOH + (aq)
Cu(OH)2 (aq)
Cu(OH)3 - (aq)
Cu(OH)4 2- (aq)
CuS (s)
Cu2S (s)
CuO (s)
CuCO3Cu(OH)2 (s)
(CuCO3)2Cu(OH)2 (s)
F2 (g)
F - (aq)
Fe (s)
Fe 2+ (aq)
FeOH + (aq)
Fe(OH)2 (aq)
Fe(OH)3 - (aq)
Fe 3+ (aq)
FeOH 2+ (aq)
Fe(OH)2 + (aq)
Fe(OH)3 (aq)
Fe(OH)4 - (aq)
Fe2(OH)2 4+ (aq)
FeS2 (s)
FeO (s)
Fe(OH)2 (s)
Fe2O3 (s)
Fe3O4 (s)
FeOOH (s)
Fe(OH)3 (s)
FeCO3 (s)
Fe2SiO4 (s)
H2O (l)
H + (aq)
H2CO3 (aq)
HCO3 - (aq)
HNO3 (aq)
HCl (aq)
H3AsO4 (aq)
H2AsO4 - (aq)
HAsO4 2- (aq)
H2AsO3 - (aq)
H3BO3 (aq)
HBrO (aq)
HCOOH (aq)
HCOO - (aq)
HCN (aq)
HClO (aq)
HCoO2 - (aq)
HCrO4 - (aq)
HCuO2 - (aq)
HF (aq)
HF2 - (aq)

Mg5Al2Si3O10(OH)8 (s)
Mg (s)
Mg 2+ (aq)
MgOH + (aq)
Mg(OH)2 (s)
Mn (s)
Mn 2+ (aq)
Mn(OH)2 (s)
Mn3O4 (s)
MnOOH (s)
MnO2 (s)
MnCO3 (s)
MnS (s)
MnSiO3 (s)
NaHCO3 (aq)
NaCO3 - (aq)
Na + (aq)
NaCl (aq)
NaOH (aq)
NO3 - (aq)
NH3 (aq)
NaAlSiO3O8 (s)
NH2CH2COOH (aq)
NH2CH2COO - (aq)
N2 (g)
N2O (g)
NH3 (g)
NH4 + (aq)
NO2 - (aq)
Ni 2+ (aq)
NiOH + (aq)
Ni(OH)2 (aq)
Ni(OH)3 - (aq)
NiO (s)
NiS (s)
OH - (aq)
O2 (g)
O2 (aq)
O3 (g)
P (s)
PO4 3- (aq)
Pb (s)
Pb 2+ (aq)
PbOH + (aq)
Pb(OH)2 (aq)
Pb(OH)3 - (aq)
Pb(OH)4 2- (aq)
Pb(OH)2 (s)
PbO (s)
PbO2 (s)
Pb3O4 (s)
PbS (s)
PbSO4 (s)
PbCO3 (s)
S (s)
SO2 (g)
SO3 (g)
S 2- (aq)

VO2(CO3)2 3- (aq)
Zn (s)
Zn 2+ (aq)
ZnOH + (aq)
Zn(OH)2 (aq)
Zn(OH)3 - (aq)
Zn(OH)4 2- (aq)
Zn(OH)2 (s)
ZnCl + (aq)
ZnCl2 (aq)
ZnCl3 - (aq)
ZnCl4 2- (aq)
ZnCO3 (s)
Those registered molecules follow a strict naming convention by which they can be recognized (see below)...

Naming Convention

Plus (+) and minus (-) charges are denoted by the numeric value of the charge followed by a + or - sign, respectively (e.g. UO2(CO3)3 4- (aq))

The phase is always denoted last and will be marked as (l) for liquid, (s) for solid, (aq) for aqueous, and (g) for gas (see above).

When registering a molecule that is not in the library, you must also provide a linear formula during construction or registration. This is needed so that the string parsing is easier to handle when the molecule subsequently registers the necessary atoms. (e.g. UO2(CO3)3 = UO2C3O9 or UO11C3).

Author

Austin Ladshaw

Date

02/24/2014

Copyright

6.19.2 Macro Definition Documentation

6.19.2.1 M_PI

```
#define M_PI 3.14159
```

6.19.2.2 SphereVolume

```
#define SphereVolume(
            r ) ((4.0/3.0)*M_PI*r*r*r)
```

6.19.2.3 SphereArea

```
#define SphereArea(
                r ) (4.0*M_PI*r*r)
```

6.19.3 Enumeration Type Documentation

6.19.3.1 valid_phase

```
enum valid_phase
```

Enumerator

SOLID	
LIQUID	
AQUEOUS	
GAS	
PLASMA	
ADSORBED	
OTHER	

6.19.4 Function Documentation

6.19.4.1 MOLA_TESTS()

```
int MOLA_TESTS ( )
```

Function to run the MOLA tests.

This function is callable from the UI and is used to run several algorithm tests for the Molecule objects. This test should never report any errors.

6.20 monkfish.h File Reference

Multi-fiber wOven Nest Kernel For Interparticle Sorption History.

```
#include "dogfish.h"
```

Classes

- struct MONKFISH_PARAM
  Data structure for species specific information and parameters.
- struct MONKFISH_DATA
  Primary data structure for running MONKFISH.

Functions

- double default_porosity (int i, int l, const void ∗user_data)

    Default porosity function for MONKFISH.
- double default_density (int i, int l, const void ∗user_data)

    Default density function for MONKFISH.
- double default_interparticle_diffusion (int i, int l, const void ∗user_data)

    Default interparticle diffusion function.
- double default_monk_adsorption (int i, int l, const void ∗user_data)

    Default adsorption strength function.
- double default_monk_equilibrium (int i, int l, const void ∗user_data)

    Default equilibirium adsorption function in mg/g.
- double default_monkfish_retardation (int i, int l, const void ∗user_data)

    Default retardation coefficient function.
- double default_exterior_concentration (int i, const void ∗user_data)

    Default exterior concentratio function.
- double default_film_transfer (int i, const void ∗user_data)

    Default film mass transfer function.
- int setup_MONKFISH_DATA (FILE ∗file, double(∗eval_porosity)(int i, int l, const void ∗user_data), double(∗eval_density)(int i, int l, const void ∗user_data), double(∗eval_ext_diff)(int i, int l, const void ∗user↩_data), double(∗eval_adsorb)(int i, int l, const void ∗user_data), double(∗eval_retard)(int i, int l, const void ∗user_data), double(∗eval_ext_conc)(int i, const void ∗user_data), double(∗eval_ext_film)(int i, const void ∗user_data), double(∗dog_diffusion)(int i, int l, const void ∗user_data), double(∗dog_ext_film)(int i, const void ∗user_data), double(∗dog_surf_conc)(int i, const void ∗user_data), const void ∗user_data, MONKFISH_D↩ATA ∗monk_dat)

    Setup function to allocate memory and setup function pointers for the MONKFISH simulation.
- int MONKFISH_TESTS ()

    Function to run tests on the MONKFISH algorithms.

6.20.1 Detailed Description

Multi-fiber wOven Nest Kernel For Interparticle Sorption History.

monkfish.cpp

This file contains structures and functions associated with modeling the sorption characteristics of woven fiber bundles used to recover uranium from seawater. It is coupled with the DOGFISH kernel that determines the sorption of individual fibers. This kernel will resolve the interparticle diffusion between bundles of individual fibers in a woven ball-like domain.

Warning

Functions and methods in this file are still under construction.

Author

Austin Ladshaw

Date

04/14/2015

Copyright

6.20.2 Function Documentation

6.20.2.1 default_porosity()

```
double default_porosity (
            int i,
            int l,
            const void * user_data )
```

Default porosity function for MONKFISH.

This function assumes a linear relationship between the maximum porosity at the center of the woven fibers and the minimum porosity at the edge of the woven fiber bundle.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.20.2.2 default_density()

```
double default_density (
            int i,
            int l,
            const void * user_data )
```

Default density function for MONKFISH.

This function calls the porosity function and uses the single fiber density to provide an estimate of the bulk fiber density locally in the woven fiber bundle.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.20.2.3 default_interparticle_diffusion()

```
double default_interparticle_diffusion (
            int i,
            int l,
            const void * user_data )
```

Default interparticle diffusion function.

This function assumes that the interparticle diffusivity is a contant and returns that diffusivity multiplied by the domain porosity to form the effective diffusion coefficient in the domain.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.20.2.4 default_monk_adsorption()

```
double default_monk_adsorption (
             int i,
             int l,
             const void * user_data )
```

Default adsorption strength function.

This function will either use the default equilibrium function or the DOGFISH simulation result to produce the approximate adsorption strength using perturbation theory.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.20.2.5 default_monk_equilibrium()

```
double default_monk_equilibrium (
             int i,
             int l,
             const void * user_data )
```

Default equilibirium adsorption function in mg/g.

This function uses the exterior species' concentration (mol/L), the species' molecular weight (g/mol), and the bulk fiber density (g/L) to calculate the adsorption equilibrium in mg/g. It assumes that the exterior concentration represents the moles of species per liter of solution that is being sorbed.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.20.2.6 default_monkfish_retardation()

```
double default_monkfish_retardation (
            int i,
            int l,
            const void * user_data )
```

Default retardation coefficient function.

This function calls the porosity, density, and adsorption functions to evaluate the retardation coefficient of the diffusing material.

Parameters

i	index for the ith adsorbing species
l	index for the lth node in the domain
user_data	pointer to the MONKFISH_DATA structure

6.20.2.7 default_exterior_concentration()

```
double default_exterior_concentration (
            int i,
            const void * user_data )
```

Default exterior concentratio function.

This function assumes that the exterior concentration for sorption is just equal to the value of exterior_concentration given in MONKFISH_PARAM.

Parameters

i	index for the ith adsorbing species
user_data	pointer to the MONKFISH_DATA structure

6.20.2.8 default_film_transfer()

```
double default_film_transfer (
            int i,
            const void * user_data )
```

Default film mass transfer function.

This function assumes that the film mass transfer coefficient is just equal to the value of the film_transfer_coeff in MONKFISH_PARAM.

Parameters

i	index for the ith adsorbing species
user_data	pointer to the MONKFISH_DATA structure

6.20.2.9 setup_MONKFISH_DATA()

```
int setup_MONKFISH_DATA (
            FILE * file,
            double(*)(int i, int l, const void *user_data) eval_porosity,
            double(*)(int i, int l, const void *user_data) eval_density,
            double(*)(int i, int l, const void *user_data) eval_ext_diff,
            double(*)(int i, int l, const void *user_data) eval_adsorb,
            double(*)(int i, int l, const void *user_data) eval_retard,
            double(*)(int i, const void *user_data) eval_ext_conc,
            double(*)(int i, const void *user_data) eval_ext_film,
            double(*)(int i, int l, const void *user_data) dog_diffusion,
            double(*)(int i, const void *user_data) dog_ext_film,
            double(*)(int i, const void *user_data) dog_surf_conc,
            const void * user_data,
            MONKFISH_DATA * monk_dat )
```

Setup function to allocate memory and setup function pointers for the MONKFISH simulation.

This function will allocate memory and setup the MONKFISH problem. To specify use of the default functions in MONKFISH, pass NULL args for all function pointers and the user_data data structure. Otherwise, pass in your own custom arguments. The MONKFISH_DATA pointer must always be passed to this function.

Parameters

file	pointer to the output file to print out results
eval_porosity	function pointer for the bulk domain porosity function
eval_density	function pointer for the bulk domain density function
eval_ext_diff	function pointer for the interparticle diffusion function
eval_adsorb	function pointer for the adsorption strength function
eval_retard	function pointer for the retardation coefficient function
eval_ext_conc	function pointer for the external concentration function
eval_ext_film	function pointer for the external film mass transfer function
dog_diffusion	function pointer for the DOGFISH diffusion function (see dogfish.h)
dog_ext_film	function pointer for the DOGFISH film mass transfer (see dogfish.h)
dog_surf_conc	function pointer for the DOGFISH surface concentration (see dogfish.h)
user_data	pointer for the user's own data structure (only if using custom functions)
monk_dat	pointer for the MONKFISH_DATA structure

6.20.2.10 MONKFISH_TESTS()

```
int MONKFISH_TESTS ( )
```

Function to run tests on the MONKFISH algorithms.

This function currently does nothing and is not callable from the UI.

6.21 sandbox.h File Reference

Coding Test Area.

```
#include "flock.h"
#include "school.h"
```

Functions

- int RUN_SANDBOX ()

  Function to run the methods implemented in the Sandbox.
- int blah ()

  Function to provide a C-style linkage and function call for C++ functions and objects.
- double obj_func (double ∗list, int len, double ∗args)

6.21.1 Detailed Description

Coding Test Area.

sandbox.cpp

This file contains a series of simple tests for routines used in other files and algorithms. Before any code or methods are used, they are tested here to make sure that they are useful. The tests in the sandbox are callable from the UI to make it easier to alter existing sandbox code and run tests on new proposed methods or algorithms.

Warning

Functions and methods in this file are not meant to be used anywhere else.

Author

Austin Ladshaw

Date

04/11/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.21.2 Function Documentation

6.21.2.1 RUN_SANDBOX()

```
int RUN_SANDBOX ( )
```

Function to run the methods implemented in the Sandbox.

This function is callable from the UI and is used to observe results from the tests of newly developed algorithms. Edit header and source files here to test out your own routines or functions. Then you can run those functions by rebuilding the Ecosystem executable and running the sandbox tests.

6.21.2.2 blah()

```
int blah ( )
```

Function to provide a C-style linkage and function call for C++ functions and objects.

6.21.2.3 obj_func()

```
double obj_func (
            double * list,
            int len,
            double * args )
```

6.22 school.h File Reference

Seawater Codes from a Highly Object-Oriented Library.

```
#include "eel.h"
#include "mola.h"
#include "shark.h"
#include "dogfish.h"
#include "monkfish.h"
#include "yaml_wrapper.h"
#include "mesh.h"
```

6.22.1 Detailed Description

Seawater Codes from a Highly Object-Oriented Library.

This file contains include statements for all files used in the aqueous adsorption problems, primarily targeted at Seawater simulations. Include this file into any other project or source code that needs the methods below.

Files Included in SCHOOL

eel.h mola.h shark.h dogfish.h monkfish.h yaml_wrapper.h

Note

(1) shark.h also includes methods from macaw.h and lark.h
(2) dogfish.h also includes methods from finch.h

Author

Austin Ladshaw

Date

02/23/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.23 scopsowl.h File Reference

Simultaneously Coupled Objects for Pore and Surface diffusion Operations With Linear systems.

```
#include "egret.h"
#include "skua.h"
```

Classes

- struct SCOPSOWL_PARAM_DATA

  Data structure for the species' parameters in SCOPSOWL.
- struct SCOPSOWL_DATA

  Primary data structure for SCOPSOWL simulations.

Macros

- #define SCOPSOWL_HPP_
- #define Dp(Dm, ep) (ep∗ep∗Dm)

  Estimate of Pore Diffusivity (cm^2/s)
- #define Dk(rp, T, MW) (9700.0∗rp∗pow((T/MW),0.5))

  Estimate of Knudsen Diffusivity (cm^2/s)
- #define avgDp(Dp, Dk) (pow(((1/Dp)+(1/Dk)),-1.0))

  Estimate of Average Pore Diffusion (cm^2/s)

Functions

- void print2file_species_header (FILE ∗Output, SCOPSOWL_DATA ∗owl_dat, int i)

  Function to print out the main header for the output file.
- void print2file_SCOPSOWL_time_header (FILE ∗Output, SCOPSOWL_DATA ∗owl_dat, int i)

  Function to print out the time and space header for the output file.
- void print2file_SCOPSOWL_header (SCOPSOWL_DATA ∗owl_dat)

  Function to call the species and time header functions.
- void print2file_SCOPSOWL_result_old (SCOPSOWL_DATA ∗owl_dat)

  Function to print out the old time results to the output file.
- void print2file_SCOPSOWL_result_new (SCOPSOWL_DATA ∗owl_dat)

  Function to print out the new time results to the output file.
- double default_adsorption (int i, int l, const void ∗user_data)

  Default function for evaluating adsorption and adsorption strength.
- double default_retardation (int i, int l, const void ∗user_data)

  Default function for evaluating retardation coefficient.
- double default_pore_diffusion (int i, int l, const void ∗user_data)

  Default function for evaluating pore diffusivity.
- double default_surf_diffusion (int i, int l, const void ∗user_data)

  Default function for evaluating surface diffusion for HOMOGENEOUS pellets.
- double zero_surf_diffusion (int i, int l, const void ∗user_data)

  Zero function for evaluating no surface diffusion in HOMOGENEOUS pellets.
- double default_effective_diffusion (int i, int l, const void ∗user_data)

  Default function for evaluating effective diffusivity for HOMOGENEOUS pellets.

- double const_pore_diffusion (int i, int l, const void ∗user_data)

  Constant pore diffusion function for homogeneous or heterogeneous pellets.
- double default_filmMassTransfer (int i, const void ∗user_data)

  Default function for evaluating the film mass transfer coefficient.
- double const_filmMassTransfer (int i, const void ∗user_data)

  Constant film mass transfer coefficient function.
- int setup_SCOPSOWL_DATA (FILE ∗file, double(∗eval_sorption)(int i, int l, const void ∗user_data), double(∗eval_retardation)(int i, int l, const void ∗user_data), double(∗eval_pore_diff)(int i, int l, const void ∗user_data), double(∗eval_filmMT)(int i, const void ∗user_data), double(∗eval_surface_diff)(int i, int l, const void ∗user_data), const void ∗user_data, MIXED_GAS ∗gas_data, SCOPSOWL_DATA ∗owl_data)

  Setup function to allocate memory and setup function pointers for the SCOPSOWL simulation.
- int SCOPSOWL_Executioner (SCOPSOWL_DATA ∗owl_dat)

  SCOPSOWL executioner function to solve a time step.
- int set_SCOPSOWL_ICs (SCOPSOWL_DATA ∗owl_dat)

  Function to set the initial conditions for a SCOPSOWL simulation.
- int set_SCOPSOWL_timestep (SCOPSOWL_DATA ∗owl_dat)

  Function to set the timestep of the SCOPSOWL simulation.
- int SCOPSOWL_preprocesses (SCOPSOWL_DATA ∗owl_dat)

  Function to perform all preprocess SCOPSOWL operations.
- int set_SCOPSOWL_params (const void ∗user_data)

  Function to set the values of all non-linear system parameters during simulation.
- int SCOPSOWL_postprocesses (SCOPSOWL_DATA ∗owl_dat)

  Function to perform all postprocess SCOPSOWL operations.
- int SCOPSOWL_reset (SCOPSOWL_DATA ∗owl_dat)

  Function to reset all stateful information to prepare for next simulation.
- int SCOPSOWL (SCOPSOWL_DATA ∗owl_dat)

  Function to progress the SCOPSOWL simulation through time till complete.
- int SCOPSOWL_SCENARIOS (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate)

  Function to perform a SCOPSOWL simulation based on a set of parameters given in input files.
- int SCOPSOWL_TESTS ()

  Function to run a SCOPSOWL test simulation.

6.23.1 Detailed Description

Simultaneously Coupled Objects for Pore and Surface diffusion Operations With Linear systems.

scopsowl.cpp

This file contains structures and functions associated with modeling adsorption in commercial, bi-porous adsorbents such as zeolites and mordenites. The pore diffusion and mass transfer equations are coupled with adsorption and surface diffusion through smaller crystals embedded in a binder matrix. However, you can also direct this simulation to treat the adsorbent as homogeneous (instead of heterogeneous) in order to model an even greater variety of gaseous adsorption kinetic problems. This object is coupled with either MAGPIE, SKUA, or BOTH depending on the type of simulation requested.

Author

Austin Ladshaw

Date

01/29/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.23.2 Macro Definition Documentation

6.23.2.1 SCOPSOWL_HPP_

```
#define SCOPSOWL_HPP_
```

6.23.2.2 Dp

```
#define Dp(
              Dm,
              ep ) (ep*ep*Dm)
```

Estimate of Pore Diffusivity (cm^2/s)

6.23.2.3 Dk

```
#define Dk(
              rp,
              T,
              MW ) (9700.0*rp*pow((T/MW),0.5))
```

Estimate of Knudsen Diffusivity (cm^2/s)

6.23.2.4 avgDp

```
#define avgDp(
              Dp,
              Dk ) (pow(((1/Dp)+(1/Dk)),-1.0))
```

Estimate of Average Pore Diffusion (cm^2/s)

6.23.3 Function Documentation

6.23.3.1 print2file_species_header()

```
void print2file_species_header (
             FILE * Output,
             SCOPSOWL_DATA * owl_dat,
             int i )
```

Function to print out the main header for the output file.

6.23.3.2 print2file_SCOPSOWL_time_header()

```
void print2file_SCOPSOWL_time_header (
            FILE * Output,
            SCOPSOWL_DATA * owl_dat,
            int i )
```

Function to print out the time and space header for the output file.

6.23.3.3 print2file_SCOPSOWL_header()

```
void print2file_SCOPSOWL_header (
            SCOPSOWL_DATA * owl_dat )
```

Function to call the species and time header functions.

6.23.3.4 print2file_SCOPSOWL_result_old()

```
void print2file_SCOPSOWL_result_old (
            SCOPSOWL_DATA * owl_dat )
```

Function to print out the old time results to the output file.

6.23.3.5 print2file_SCOPSOWL_result_new()

```
void print2file_SCOPSOWL_result_new (
            SCOPSOWL_DATA * owl_dat )
```

Function to print out the new time results to the output file.

6.23.3.6 default_adsorption()

```
double default_adsorption (
            int i,
            int l,
            const void * user_data )
```

Default function for evaluating adsorption and adsorption strength.

This function is called in the preprocesses and postprocesses to estimate the strength of adsorption in the macro-scale problem from perturbations. It will use perturbations in either the MAGPIE simulation or SKUA simulation, depending on the type of problem the user is solving.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.7 default_retardation()

```
double default_retardation (
            int i,
            int l,
            const void * user_data )
```

Default function for evaluating retardation coefficient.

This function is called in the preprocesses and postprocesses to estimate the retardation coefficient for the simulation. It is recalculated at every time step to keep track of all changing conditions in the simulation.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.8 default_pore_diffusion()

```
double default_pore_diffusion (
            int i,
            int l,
            const void * user_data )
```

Default function for evaluating pore diffusivity.

This function is called during the evaluation of non-linear residuals to more accurately represent non-linearities in the pore diffusion behavior. The pore diffusion is calculated based on kinetic theory of gases (see egret.h) and is adjusted according to the Knudsen Diffusion model and the porosity of the binder material.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.9 default_surf_diffusion()

```
double default_surf_diffusion (
            int i,
            int l,
            const void * user_data )
```

Default function for evaluating surface diffusion for HOMOGENEOUS pellets.

This function is ONLY used if the pellet is determined to be homogeneous. Otherwise, this is replaced by the surface diffusion function for the SKUA simulation. The diffusivity is calculated based on the Arrhenius rate expression and then adjusted by the outside partial pressure of the adsorbing species.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.10 zero_surf_diffusion()

```
double zero_surf_diffusion (
             int i,
             int l,
             const void * user_data )
```

Zero function for evaluating no surface diffusion in HOMOGENEOUS pellets.

This function is ONLY used if the pellet is determined to be homogeneous and we want to specify that there is no surface diffusion.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.11 default_effective_diffusion()

```
double default_effective_diffusion (
             int i,
             int l,
             const void * user_data )
```

Default function for evaluating effective diffusivity for HOMOGENEOUS pellets.

This function is ONLY used if the pellet is determined to be homogeneous. Otherwise, this is replaced by the pore diffusion function. The effective diffusivity is determined by the combination of pore diffusivity and surface diffusivity with adsorption strength in an homogeneous pellet.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.12 const_pore_diffusion()

```
double const_pore_diffusion (
            int i,
            int l,
            const void * user_data )
```

Constant pore diffusion function for homogeneous or heterogeneous pellets.

This function should be used if the user wants to specify a constant pore diffusivity. The value of pore diffusion is then set equal to the value of pore_diffusion in the SCOPSOWL_PARAM_DATA structure.

Parameters

i	index for the ith species in the system
l	index for the lth node in the macro-scale domain
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.13 default_filmMassTransfer()

```
double default_filmMassTransfer (
            int i,
            const void * user_data )
```

Default function for evaluating the film mass transfer coefficient.

This function is called during the setup of the boundary conditions and is used to estimate the film mass transfer coefficient for the macro-scale problem. The coefficient is calculated according to the kinetic theory of gases (see egret.h).

Parameters

i	index for the ith species in the system
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.14 const_filmMassTransfer()

```
double const_filmMassTransfer (
            int i,
            const void * user_data )
```

Constant film mass transfer coefficient function.

This function is used when the user wants to specify a constant value for film mass transfer. The value of that coefficient is then set equal to the value of film_transfer in the SCOPSOWL_PARAM_DATA structure.

Parameters

i	index for the ith species in the system
user_data	pointer for the SCOSPOWL_DATA structure

6.23.3.15 setup_SCOPSOWL_DATA()

```
int setup_SCOPSOWL_DATA (
            FILE * file,
            double(*)(int i, int l, const void *user_data) eval_sorption,
            double(*)(int i, int l, const void *user_data) eval_retardation,
            double(*)(int i, int l, const void *user_data) eval_pore_diff,
            double(*)(int i, const void *user_data) eval_filmMT,
            double(*)(int i, int l, const void *user_data) eval_surface_diff,
            const void * user_data,
            MIXED_GAS * gas_data,
            SCOPSOWL_DATA * owl_data )
```

Setup function to allocate memory and setup function pointers for the SCOPSOWL simulation.

This function sets up the memory and function pointers used in SCOPSOWL simulations. User can provide NULL in place of functions for the function pointers and the setup will automatically use just the default settings. However, the user is required to pass the necessary data structure pointers for MIXED_GAS and SCOPSOWL_DATA.

Parameters

file	pointer to the output file to print out results
eval_sorption	pointer to the adsorption evaluation function
eval_retardation	pointer to the retardation evaluation function
eval_pore_diff	pointer to the pore diffusion function
eval_filmMT	pointer to the film mass transfer function
eval_surface_diff	pointer to the surface diffusion function (required)
user_data	pointer to the user's data structure used for the parameter functions
gas_data	pointer to the MIXED_GAS structure used to evaluate kinetic gas theory
owl_data	pointer to the SCOPSOWL_DATA structure

6.23.3.16 SCOPSOWL_Executioner()

```
int SCOPSOWL_Executioner (
            SCOPSOWL_DATA * owl_dat )
```

SCOPSOWL executioner function to solve a time step.

This function will call the preprocess, solver, and postprocess functions to evaluate a single time step in a simulation. All simulation conditions must be set prior to calling this function. This function will typically be the one called from other simulations that will involve a SCOPSOWL evaluation to resolve kinetic coupling.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.17 set_SCOPSOWL_ICs()

```
int set_SCOPSOWL_ICs (
            SCOPSOWL_DATA * owl_dat )
```

Function to set the initial conditions for a SCOPSOWL simulation.

This function will setup the initial conditions of the simulation based on the initial temperature, pressure, and adsorbed molefractions. It assumes that the initial conditions are constant throughout the domain of the problem. This function should only be called once during a simulation.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.18 set_SCOPSOWL_timestep()

```
int set_SCOPSOWL_timestep (
            SCOPSOWL_DATA * owl_dat )
```

Function to set the timestep of the SCOPSOWL simulation.

This function is used to set the next time step to be used in the SCOPSOWL simulation. A constant time step based on the size of the pellet discretization will be used. Users may want to use a custom time step to ensure that coupled-multi-scale systems are all in sync.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.19 SCOPSOWL_preprocesses()

```
int SCOPSOWL_preprocesses (
            SCOPSOWL_DATA * owl_dat )
```

Function to perform all preprocess SCOPSOWL operations.

This function will update the boundary conditions and simulation conditions based on the current temperature, pressure, and gas phase composition, which may all vary in time. Since this function is called by the SCOPSOW↩
L_Executioner, it does not need to be called explicitly by the user.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.20 set_SCOPSOWL_params()

```
int set_SCOPSOWL_params (
            const void * user_data )
```

Function to set the values of all non-linear system parameters during simulation.

This is the function override for the FINCH setparams function (see finch.h). It will update the values of non-linear parameters in the residuals so that all variables in a species' system are fully coupled.

Parameters

user_data	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.21 SCOPSOWL_postprocesses()

```
int SCOPSOWL_postprocesses (
            SCOPSOWL_DATA * owl_dat )
```

Function to perform all postprocess SCOPSOWL operations.

This function will update the retardation coefficients based on newly obtained simulation results for the current time step and calculate the average and total amount of adsorption of each species in the domain. Additionally, this function will call the print functions to store simulation results in the output file.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.22 SCOPSOWL_reset()

```
int SCOPSOWL_reset (
            SCOPSOWL_DATA * owl_dat )
```

Function to reset all stateful information to prepare for next simulation.

This function will update the stateful information used in SCOPSOWL to prepare the system for the next time step in the simulation. However, because updating the states erases the old state, the user must be absolutely sure that the simulation is ready to be updated. For just running standard simulations, this is not an issue, but in coupling with other simulations it is very important.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.23 SCOPSOWL()

```
int SCOPSOWL (
            SCOPSOWL_DATA * owl_dat )
```

Function to progress the SCOPSOWL simulation through time till complete.

This function will call the initial conditions, then progressively call the executioner, time step, and reset functions to propagate the simulation in time. As such, this function is primarily used when running a SCOPSOWL simulation by itself and not when coupling it to an other problem.

Parameters

owl_dat	pointer to the SCOPSOWL_DATA structure (must be initialized)

6.23.3.24 SCOPSOWL_SCENARIOS()

```
int SCOPSOWL_SCENARIOS (
            const char * scene,
            const char * sorbent,
            const char * comp,
            const char * sorbate )
```

Function to perform a SCOPSOWL simulation based on a set of parameters given in input files.

This is the primary function to be called when running a stand-alone SCOPSOWL simulation. Parameters and system information for the simulation are given in a series of input files that come in as character arrays. These inputs are all required to call this function.

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File

Note

Each input file has a particular format that must be strictly adhered to in order for the simulation to be carried out correctly. The format for each input file, and an example, is provided below...

Scenario Input Format

System Temperature (K) [tab] Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Simulation Time (hrs) [tab] Print Out Time (hrs)
BC Type (0 = Neumann, 1 = Dirichlet)
Number of Gas Species
Initial Total Adsorption (mol/kg)
Name of ith Species [tab] Adsorbable? (0 = false, 1 = true) [tab] Gas Phase Molefraction [tab] Initial Sorbed Molefraction
(repeat above for all species)

Example Scenario Input

353.15 101.35 0.36
4.0 0.05
0
5
0.0
N2 0 0.7634 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0191 0.0
Above example is for a 5-component mixture of N2, O2, Ar, CO2, and H2O, but we are only considering the H2O as adsorbable.

Adsorbent Input File

Heterogeneous Pellet? (0 = false, 1 = true) [tab] Surface Diffusion Included? (0 = false, 1 = true)
Macro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (cm) [tab] Pellet Density (kg/L) [tab] Porosity (vol. void / vol. binder) [tab] Pore Radius (cm)
(Below is only needed if pellet is Heterogeneous)
Micro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
Crystal Radius (um) [tab] Binder Fraction (vol. binder / vol. pellet)

Example Adsorbent Input

1 1
2
0.118 1.69 0.272 3.5E-6
2
2.0 0.175
Above example is for a heterogeneous adsorbent with surface diffusion. The pellet and crystals are both considered spherical. Pellet radius is 0.118 cm, density is 1.69 kg/L, porosity is 0.272, and pore size is 3.5e-6 cm. The pellet is made up of 17.5 % binder material and contains crystals roughly 2.0 um in radius.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846

0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is a continuation of the Scenario Input example wherein each grouping represents parameters that are associated with N2, O2, Ar, CO2, and H2O, respectively. The order is VERY important!

Adsorbate Input File

{ Type of Surface Diffusion Function (0 = constant, 1 = simple Darken, 2 = theoretical Darken) }
(NOTE: The above option is only given IF the pellet was specified as Heterogeneous!)
Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0
0.8814 0.0
267.999 0.0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
1.28 540.1
374.99 0.01
3.01
1.27
2
-46597.5 -53.6994
-125024 -221.073
Above example would be for a simulation involving two adsorbable species using a constant surface diffusion function. Each adsorbable species has it's own set of kinetic and equilibrium parameters that must be given in the same order as the species appeared in the Scenario Input. Note: we do not need to supply this information for non-adsorbable species.

6.23.3.25 SCOPSOWL_TESTS()

```
int SCOPSOWL_TESTS ( )
```

Function to run a SCOPSOWL test simulation.

This function runs a test of the SCOPSOWL physics and prints out results to a text file. It is callable from the UI.

6.24 scopsowl_opt.h File Reference

Optimization Routine for Surface Diffusivities in SCOPSOWL.

```
#include "scopsowl.h"
```

Classes

- struct SCOPSOWL_OPT_DATA

  Data structure for the SCOPSOWL optmization routine.

Functions

- int SCOPSOWL_OPT_set_y (SCOPSOWL_OPT_DATA ∗owl_opt)

  Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.
- int initial_guess_SCOPSOWL (SCOPSOWL_OPT_DATA ∗owl_opt)

  Function to set up an initial guess for the surface diffusivity parameter in SCOPSOWL.
- void eval_SCOPSOWL_Uptake (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.
- int SCOPSOWL_OPTIMIZE (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate, const char ∗data)

  Function called to perform the optimization routine given a specific set of information and data.

6.24.1 Detailed Description

Optimization Routine for Surface Diffusivities in SCOPSOWL.

scopsowl_opt.cpp

This file contains structures and functions associated with performing non-linear least-squares optimization of the SCOPSOWL simulation results against actual kinetic adsorption data. The optimization routine here allows you to run data comparisons and optimizations in three forms: (i) Rough optimizations - cheaper operations, but less accurate, (ii) Exact optmizations - much more expensive, but greater accuracy, and (iii) data/model comparisons - no optimization, just using system parameters to compare simulation results agains a set of data.

Depending on the level of optimization desired, this routine could take several minutes or several hours. The optimization/comparisons are printed out in two files: (i) a parameter file, which contains the simulation partial pressures and temperatures and the optimized diffusivities with the euclidean norm of the fitting and (ii) a comparison file that shows the model value and data value at each time step for each kinetic curve.

The optimized diffusion parameters are given for each individual kinetic data curve. Each data curve will have a different pairing of partial pressure and temperature. Because of this, you will get a list of different diffusivities for each data curve. To get the optimum kinetic parameters from this list of diffusivities, you must fit the diffusion parameter values to the following diffusion function model...
D_opt = D_ref ∗ exp(-E / (R∗T)) ∗ pow(p , (T_ref/T) - B)
where D_ref is the Reference Diffusivity (um^2/hr), E is the activation energy for adsorption (J/mol), R is the gas law constant (J/K/mol), T is the system temperature (K), p is the partial pressure of the adsorbing species (kPa), T_ref is the Reference Temperature (K), and B is the Affinity constant. This algorithm does not automatically produce these parameters for you, but gives you everything you need to produce them yourself.
This routine allows you to optimize multiple kinetic curves at one time. However, all data must be for the same adsorbent-adsorbate system. In other words, the adsorbent and adsorbate pair must be the same for each kinetic curve analyzed. Also, each experiment must have been done in a thin bed or continuous flow system where the adsorbents were exposed to a nearly constant outside partial pressure for all time steps and the gas velocity of that system is assumed constant for all experiments. This experimental setup is very typical for studying adsorption kinetics for gas-solid systems.

Author

Austin Ladshaw

Date

05/14/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.24.2 Function Documentation

6.24.2.1 SCOPSOWL_OPT_set_y()

```
int SCOPSOWL_OPT_set_y (
            SCOPSOWL_OPT_DATA * owl_opt )
```

Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.

This function takes the current mole fraction of the adsorbing gas and calculates the gas mole fractions of the other gases in the sytem based on the standard inlet gas composition given in the scenario file.

6.24.2.2 initial_guess_SCOPSOWL()

```
int initial_guess_SCOPSOWL (
            SCOPSOWL_OPT_DATA * owl_opt )
```

Function to set up an initial guess for the surface diffusivity parameter in SCOPSOWL.

This function performs the Rough optimization on the surface diffusivity based on the idea of reducing or eliminating function bias between data and simulation. A positive function bias means that the simulation curve is "higher" than the data curve and a negative function bias means that the simulation curve is "lower" than the data curve. We use this information to incrementally adjust the rate of surface diffusion until this bias is near zero. When bias is near zero, the simulation is nearly optimized, but further refinement may be necessary to find the true minimum solution.

6.24.2.3 eval_SCOPSOWL_Uptake()

```
void eval_SCOPSOWL_Uptake (
            const double * par,
            int m_dat,
            const void * data,
            double * fvec,
            int * info )
```

Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.

This function will run the SCOPSOWL simulation at a given value of surface diffusivity and produce residuals that feed into the Levenberg-Marquardt's algorithm for non-linear least-squares regression. The form of this function is specific to the format required by the lmfit routine.

Parameters

par	array of parameters that are to be optimized
m_dat	number of data points or functions to evaluate
data	user supplied data structure holding information necessary to form the residuals
fvec	array of residuals computed at the current parameter values
info	integer pointer denoting whether or not the user requests to end a particular simulation

6.24.2.4 SCOPSOWL_OPTIMIZE()

```
int SCOPSOWL_OPTIMIZE (
            const char * scene,
            const char * sorbent,
            const char * comp,
            const char * sorbate,
            const char * data )
```

Function called to perform the optimization routine given a specific set of information and data.

This is the function that is callable by the UI. The user must provide 5 input files to the routine in order to establish simulation conditions, adsborbent properties, component properties, adsorbate equilibrium parameters, and the set of data that we are comparing the simulations to. Each input file has a very specific structure and order to the information that it contains. The structure here is DIFFERENT than the structure for just running standard SCOP↵
SOWL simulations (see scopsowl.h).

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File
data	Kinetic Adsorption Data File

Note

Much of the structure of these input files are "similar" to that of the input files used in SCOPSOWL_SCENA↵
RIOS (see scopsowl.h), but with some notable differences. Below gives the format for each input file with an example. Make sure your input files follow this format before calling this routine from the UI.

Scenario Input File

Optimization? (0 = false, 1 = true) [tab] Rough Optimization? (0 = false, 1 = true)
Surf. Diff. (0 = constant, 1 = simple Darken, 2 = theoretical Darken) [tab] BC Type (0 = Neumann, 1 = Dirichlet)
Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Number of Gaseous Species
Initial Adsorption Total (mol/kg)
Name [tab] Adsorbable? (0 = false, 1 = true) [tab] Inlet Gas Mole Fraction [tab] Initial Adsorbed Mole Fraction
(NOTE: The above line is repeated for all species in gas phase. Also, this algorithm only allows you to consider one adsorbable gas component. Inlet gas mole fractions must be non-zero for all non-adsorbing gases and must sum

to 1.)

Example Scenario Input

1 0
0 0
101.35 0.36
5
0.0
N2 0 0.7825 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0 0.0
Above example is for running optimizations on data collected with a gas stream at 0.36 cm/s with 5 gas species in the mixture, only H2O of which is adsorbing. The "base line" or "inlet gas" without H2O has a composition of N2 at 0.7825, O2 at 0.2081, Ar at 0.009, and CO2 at 0.0004.

Adsorbent Input File

Heterogeneous Pellet? (0 = false, 1 = true)
Macro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (cm) [tab] Pellet Density (kg/L) [tab] Porosity (vol. void / vol. binder) [tab] Pore Radius (cm)
(Below is only needed if pellet is Heterogeneous)
Micro Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (cm) (i.e., cylinder length) }
Crystal Radius (um) [tab] Binder Fraction (vol. binder / vol. pellet)

Example Adsorbent Input

1
2
0.118 1.69 0.272 3.5E-6
2
2.0 0.175
Above example is nearly identical to the file given in the SCOPSOWL_SCENARIO example (see scopsowl.h). However, here we do not give an integer flag denoting whether or not we are considering surface diffusion as a mechanism. This is because we automatically assume that surface diffusion is a mechanism in the system, since that is the unknown parameter that we are performing the optimizations for.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846
0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is exactly the same as in the SCOPSOWL_SCENARIO example (see scopsowl.h). There is no difference in the input file formats for this input. Keep in mind that the order is VERY important! All species information must be in the same order that the species appeared in the Scenario input file.

Adsorbate Input File

Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0 0
0 0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
Above example gives the equilibrium parameters associated with the H2O-MS3A single component adsorption system. Note that the kinetic parameters (Ref. Diff., Act. Energy, Ref. Temp., and Affinity) were all given a value of zero. These values are irrelavent if we are running an optimization because they will be replaced with a single estimate for the diffusivity that is being optimization for. However, if we wanted to run this routine with comparisons and not do any optmization, then you would need to provide non-zero values for these parameters (at least for Ref. Diff.).

Data Input File

Number of Kinetic Data Curves
Number of data points in the ith curve
Temperature (K) [tab] Partial Pressure (kPa) [tab] Equilibrium Adsorption (mol/kg) all of ith curve
Time point 1 (hrs) [tab] Adsorption 1 (mol/kg) of ith curve
Time point 1 (hrs) [tab] Adsorption 2 (mol/kg) of ith curve
... (Repeat for all time-adsorption data points)
(Repeat above for all curves i)

Example Data Input

40
2990
298.15 0.000310922 2.9
0 0
0.166666667 0.001834419
0.333611111 0.004880247
0.5 0.008306803
...
2789
298.15 0.00055189 5
0 0
0.166944444 0.003350185
0.333611111 0.007418267
0.5 0.009930906
0.666666667 0.014597236
0.833611111 0.021377373
....
Above is a partial example for a data set of 40 kinetic curves. The first curve contains 2990 data points and has temperature of 298.15 K, partial pressure of 0.000310922 kPa, and an equilibrium adsorption of 2.9. Each first time point should start from 0 hours and each initial adsorption should correspond to the value of initial adsorption indicated in the Scenario input file. Then, this structure is repeated for all adsorptio curves.

6.25 shark.h File Reference

Speciation-object Hierarchy for Adsorption Reactions and Kinetics.

```
#include "mola.h"
#include "macaw.h"
#include "lark.h"
#include "yaml_wrapper.h"
#include "dogfish.h"
```

Classes

- class MasterSpeciesList

  Master Species List Object.
- class Reaction

  Reaction Object.
- class MassBalance

  Mass Balance Object.
- class UnsteadyReaction

  Unsteady Reaction Object (inherits from Reaction)
- class AdsorptionReaction

  Adsorption Reaction Object.
- class UnsteadyAdsorption

  Unsteady Adsorption Reaction Object.
- class MultiligandAdsorption

  Multi-ligand Adsorption Reaction Object.
- class ChemisorptionReaction

Chemisorption Reaction Object.

- class MultiligandChemisorption

    Multi-ligand Chemisorption Reaction Object.
- struct SHARK_DATA

    Data structure for SHARK simulations.

Macros

- #define Rstd 8.3144621

    Gas Law Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)
- #define Na 6.0221413E+23

    Avagadro's Number - Units: molecules/mol.
- #define kB 1.3806488E-23

    Boltzmann's Constant - Units: J/K or C∗V/K.
- #define e 1.6021766208E-19

    Elementary Electric Charge - Units: C.
- #define Faraday 96485.33289

    Faraday's Constant - C/mol.
- #define VolumeSTD 15.17

    Standard Segment Volume - cm^3/mol.
- #define AreaSTD 2.5E5

    Standard Segment Area - m^2/mol.
- #define CoordSTD 10

    Standard Coordination Number.
- #define LengthFactor(z, r, s) (((z/2.0)∗(r-s)) - (r-1.0))

    Calculation of the Length Factor Parameter in UNIQUAC.
- #define VacuumPermittivity 8.8541878176E-12

    Vacuum Permittivity Constant - F/m or C/V/m.
- #define WaterRelPerm 80.1

    Approximate Relative Permittivity for water - Unitless.
- #define AbsPerm(Rel) (Rel∗VacuumPermittivity)

    Calculation of Absolute Permittivity of a medium - F/m or C/V/m.

Typedefs

- typedef struct SHARK_DATA SHARK_DATA

    Data structure for SHARK simulations.

Enumerations

- enum valid_mb { BATCH, CSTR, PFR }

    Enumeration for the list of valid activity models for non-ideal solutions.
- enum valid_act {
  IDEAL, DAVIES, DEBYE_HUCKEL, SIT,
  PITZER }

    Enumeration for the list of valid activity models for non-ideal solutions.
- enum valid_surf_act { IDEAL_ADS, FLORY_HUGGINS, UNIQUAC_ACT }

    Enumeration for the list of valid surface activity models for non-ideal adsorption.

Functions

- void print2file_shark_info (SHARK_DATA ∗shark_dat)

  Function to print out simulation conditions and options to the output file.
- void print2file_shark_header (SHARK_DATA ∗shark_dat)

  Function to print out the head of species and time stamps to the output file.
- void print2file_shark_results_new (SHARK_DATA ∗shark_dat)

  Function to print out the simulation results for the current time step.
- void print2file_shark_results_old (SHARK_DATA ∗shark_dat)

  Function to print out the simulation results for the previous time step.
- double calculate_ionic_strength (const Matrix< double > &x, MasterSpeciesList &MasterList)

  Function to calculate the ionic strength of the solution.
- int FloryHuggins (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for simple non-ideal adsorption (for adsorption reaction object)
- int FloryHuggins_unsteady (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for simple non-ideal adsorption (for unsteady adsorption object)
- int FloryHuggins_multiligand (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for simple non-ideal adsorption (for multiligand adsorption object)
- int FloryHuggins_chemi (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for simple non-ideal adsorption (for chemisorption reaction object)
- int FloryHuggins_multichemi (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for simple non-ideal adsorption (for multiligand chemisorption object)
- int UNIQUAC (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for adsorption reaction object)
- int UNIQUAC_unsteady (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for unsteady adsorption object)
- int UNIQUAC_multiligand (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand adsorption object)
- int UNIQUAC_chemi (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for chemisorption reaction object)
- int UNIQUAC_multichemi (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand chemisorption object)
- int ideal_solution (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Activity function for Ideal Solution.
- int Davies_equation (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Activity function for Davies Equation.
- int DebyeHuckel_equation (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

  Activity function for Debye-Huckel Equation.
- int surf_act_choice (const std::string &input)

  First test of SIT Model.
- int act_choice (const std::string &input)

  Function takes a given string and returns a flag denoting which activity model was choosen.
- int reactor_choice (const std::string &input)

  Function takes a give string and returns a flag denoting which type of reactor was choosen for the system.
- bool linesearch_choice (const std::string &input)

  Function returns a bool to determine the form of line search requested.
- int linearsolve_choice (const std::string &input)

  Function returns the linear solver flag for the PJFNK method.
- int Convert2LogConcentration (const Matrix< double > &x, Matrix< double > &logx)

  Function to convert the given values of variables (x) to the log of those variables (logx)
- int Convert2Concentration (const Matrix< double > &logx, Matrix< double > &x)

Function to convert the given log values of variables (logx) to the values of those variables (x)

- int read_scenario (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for the scenario document.
- int read_multiligand_scenario (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object to setup memory space for multiligand objects.
- int read_multichemi_scenario (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object to setup memory space for multiligand chemisorption objects.
- int read_options (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for the solver options document.
- int read_species (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for the master species document.
- int read_massbalance (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for the mass balance document.
- int read_equilrxn (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for the equilibrium reaction document.
- int read_unsteadyrxn (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for the unsteady reaction document.
- int read_adsorbobjects (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for each Adsorption Object.
- int read_unsteadyadsorbobjects (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for each Unsteady Adsorption Object.
- int read_multiligandobjects (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for each MultiligandAdsorption Object.
- int read_chemisorbobjects (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for each Chemisorption Object.
- int read_multichemiobjects (SHARK_DATA ∗shark_dat)

  Function to go through the yaml object for each MultiligandChemisorption Object.
- int setup_SHARK_DATA (FILE ∗file, int(∗residual)(const Matrix< double > &x, Matrix< double > &res, const void ∗data), int(∗activity)(const Matrix< double > &x, Matrix< double > &gama, const void ∗data), int(∗precond)(const Matrix< double > &r, Matrix< double > &p, const void ∗data), SHARK_DATA ∗dat, const void ∗activity_data, const void ∗residual_data, const void ∗precon_data, const void ∗other_data)

  Function to setup the memory and pointers for the SHARK_DATA structure for the current simulation.
- int shark_add_customResidual (int i, double(∗other_res)(const Matrix< double > &x, SHARK_DATA ∗shark_dat, const void ∗other_data), SHARK_DATA ∗shark_dat)

  Function to add user defined custom residual functions to the OtherList vector object in SHARK_DATA.
- int shark_parameter_check (SHARK_DATA ∗shark_dat)

  Function to check the Reaction and UnsteadyReaction objects for missing info.
- int shark_energy_calculations (SHARK_DATA ∗shark_dat)

  Function to calculate all Reaction and UnsteadyReaction energies.
- int shark_temperature_calculations (SHARK_DATA ∗shark_dat)

  Function to calculate all Reaction and UnsteadyReaction parameters as a function of temperature.
- int shark_pH_finder (SHARK_DATA ∗shark_dat)

  Function will search MasterSpeciesList for existance of H + (aq) and OH - (aq) molecules.
- int shark_guess (SHARK_DATA ∗shark_dat)

  Function provides a rough initial guess for the values of all non-linear variables.
- int shark_initial_conditions (SHARK_DATA ∗shark_dat)

  Function to establish the initial conditions of the shark simulation.
- int shark_executioner (SHARK_DATA ∗shark_dat)

  Function to execute a shark simulation at a single time step or pH value.
- int shark_timestep_const (SHARK_DATA ∗shark_dat)

  Function to set up all time steps in the simulation to a specified constant.

- int shark_timestep_adapt (SHARK_DATA ∗shark_dat)

    Function to set up all time steps in the simulation based on success or failure to converge.
- int shark_preprocesses (SHARK_DATA ∗shark_dat)

    Function to call other functions for calculation of parameters and setting of time steps.
- int shark_solver (SHARK_DATA ∗shark_dat)

    Function to call the PJFNK solver routine given the current SHARK_DATA information.
- int shark_postprocesses (SHARK_DATA ∗shark_dat)

    Function to convert PJFNK solutions to concentration values and print to the output file.
- int shark_reset (SHARK_DATA ∗shark_dat)

    Function to reset the values of all stateful information in SHARK_DATA.
- int shark_residual (const Matrix< double > &x, Matrix< double > &F, const void ∗data)

    Default residual function for shark evaluations.
- int SHARK (SHARK_DATA ∗shark_dat)

    Function to call all above functions to perform a shark simulation.
- int SHARK_SCENARIO (const char ∗yaml_input)

    Function to perform a shark simulation based on the conditions in a yaml formatted input file.
- int SHARK_TESTS ()

    Function to perform a series of shark calculation tests.
- int SHARK_TESTS_OLD ()

    Function to perform a series of shark calculation tests (older version)

6.25.1 Detailed Description

Speciation-object Hierarchy for Adsorption Reactions and Kinetics.

shark.cpp

This file contains structures and functions associated with solving speciation and kinetic problems in aqueous systems. The primary aim for the development of these algorithms was to solve speciation and adsorption problems for the recovery of uranium resources from seawater. Seawater is an extradorinarily complex medium in which to work, which is why these algorithms are being constructed in a piece-wise, object-oriented fashion. This allows us to displace much of the complexity of the problem by breaking it down into smaller, more managable pieces.

Each piece of SHARK contributes to a residual function when solving the overall speciation, reaction, kinetic chemical problem. These residuals are then fed into the PJFNK solver function in lark.h. The variables of the system are the log(C) concentration values of each species in the system. We solve for log(C) concentrations, rather than just C, because the PJFNK method is an unbounded solution algorithm. So to prevent the algorithm from producing negative values for concentration, we reformulate all residuals in terms of the log(C) values. In this way, regardless of the value found for log(C), the concentration C will always be greater than 0.

Currenty, SHARK supports standard aqueous speciation problems with simple kinetic models based on an unsteady form of the standard reaction stoichiometry. As more methods and algorithms are completed, the SHARK simulations will be capable of doing much, much more.

Warning

Much of this is still underconstruction and many methods or interfaces may change. Use with caution.

Author

Austin Ladshaw

Date

05/27/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.25.2 Macro Definition Documentation

6.25.2.1 Rstd

```
#define Rstd 8.3144621
```

Gas Law Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)

6.25.2.2 Na

```
#define Na 6.0221413E+23
```

Avagadro's Number - Units: molecules/mol.

6.25.2.3 kB

```
#define kB 1.3806488E-23
```

Boltzmann's Constant - Units: J/K or C∗V/K.

6.25.2.4 e

```
#define e 1.6021766208E-19
```

Elementary Electric Charge - Units: C.

6.25.2.5 Faraday

```
#define Faraday 96485.33289
```

Faraday's Constant - C/mol.

6.25.2.6 VolumeSTD

```
#define VolumeSTD 15.17
```

Standard Segment Volume - cm^3/mol.

6.25.2.7 AreaSTD

```
#define AreaSTD 2.5E5
```

Standard Segment Area - m^2/mol.

6.25.2.8 CoordSTD

```
#define CoordSTD 10
```

Standard Coordination Number.

6.25.2.9 LengthFactor

```
#define LengthFactor(
              z,
              r,
              s ) (((z/2.0)*(r-s)) - (r-1.0))
```

Calculation of the Length Factor Parameter in UNIQUAC.

6.25.2.10 VacuumPermittivity

```
#define VacuumPermittivity 8.8541878176E-12
```

Vacuum Permittivity Constant - F/m or C/V/m.

6.25.2.11 WaterRelPerm

```
#define WaterRelPerm 80.1
```

Approximate Relative Permittivity for water - Unitless.

6.25.2.12 AbsPerm

```
#define AbsPerm(
              Rel ) (Rel*VacuumPermittivity)
```

Calculation of Absolute Permittivity of a medium - F/m or C/V/m.

6.25.3 Typedef Documentation

6.25.3.1 SHARK_DATA

```
typedef struct SHARK_DATA SHARK_DATA
```

Data structure for SHARK simulations.

C-style object holding data and function pointers associated with solving aqueous speciation and reaction kinetics. This object couples all other objects available in shark.h in order to provide residual calculations for each individual function that makes up the overall system model. Those residuals are brought together inside the residual function and fed into the lark.h PJFNK solver routine. That solver then attempts to find a solution to all non-linear variables simultaneously. Any function or data pointers in this structure can be overriden to change how you interface with and solve the problem. Users may also provide a set of custom residual functions through the "OtherList" vector object. Those residual function must all have the same format.

6.25.4 Enumeration Type Documentation

6.25.4.1 valid_mb

```
enum valid_mb
```

Enumeration for the list of valid activity models for non-ideal solutions.

Note

The SIT and PITZER models are not currently supported.

Enumerator

BATCH	
CSTR	
PFR	

6.25.4.2 valid_act

```
enum valid_act
```

Enumeration for the list of valid activity models for non-ideal solutions.

Note

The SIT and PITZER models are not currently supported.

Enumerator

IDEAL	
DAVIES	
DEBYE_HUCKEL	
SIT	
PITZER	

6.25.4.3 valid_surf_act

```
enum valid_surf_act
```

Enumeration for the list of valid surface activity models for non-ideal adsorption.

Note

We had to create an IDEAL_ADS option to replace the IDEAL enum already in use for non-ideal solution or aqueous phases. (ADS => adsorption)

Enumerator

IDEAL_ADS	
FLORY_HUGGINS	
UNIQUAC_ACT	

6.25.5 Function Documentation

6.25.5.1 print2file_shark_info()

```
void print2file_shark_info (
            SHARK_DATA * shark_dat )
```

Function to print out simulation conditions and options to the output file.

6.25.5.2 print2file_shark_header()

```
void print2file_shark_header (
            SHARK_DATA * shark_dat )
```

Function to print out the head of species and time stamps to the output file.

6.25.5.3 print2file_shark_results_new()

```
void print2file_shark_results_new (
            SHARK_DATA * shark_dat )
```

Function to print out the simulation results for the current time step.

6.25.5.4 print2file_shark_results_old()

```
void print2file_shark_results_old (
             SHARK_DATA * shark_dat )
```

Function to print out the simulation results for the previous time step.

6.25.5.5 calculate_ionic_strength()

```
double calculate_ionic_strength (
             const Matrix< double > & x,
             MasterSpeciesList & MasterList )
```

Function to calculate the ionic strength of the solution.

This function calculates the ionic strength of a system given the concentrations of the species present in solution, as well as any other relavent information from SHARK_DATA such as charge.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
MasterList	reference to the MasterSpeciesList object holding species information

6.25.5.6 FloryHuggins()

```
int FloryHuggins (
             const Matrix< double > & x,
             Matrix< double > & F,
             const void * data )
```

Surface Activity function for simple non-ideal adsorption (for adsorption reaction object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. NOTE: Only for AdsorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the AdsorptionReaction object holding parameter information

6.25.5.7 FloryHuggins_unsteady()

```
int FloryHuggins_unsteady (
             const Matrix< double > & x,
             Matrix< double > & F,
             const void * data )
```

Surface Activity function for simple non-ideal adsorption (for unsteady adsorption object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. NOTE: Only for UnsteadyAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the UnsteadyAdsorption object holding parameter information

6.25.5.8 FloryHuggins_multiligand()

```
int FloryHuggins_multiligand (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for simple non-ideal adsorption (for multiligand adsorption object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. NOTE: Only for MultiligandAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandAdsorption object holding parameter information

6.25.5.9 FloryHuggins_chemi()

```
int FloryHuggins_chemi (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for simple non-ideal adsorption (for chemisorption reaction object)

This is a simple surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. NOTE: Only for ChemisorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the ChemisorptionReaction object holding parameter information

6.25.5.10 FloryHuggins_multichemi()

```
int FloryHuggins_multichemi (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for simple non-ideal adsorption (for multiligand chemisorption object)

This is a simple surface activity model to be used with the Chemisorption objects to evaluate the non-ideal behavoir of the surface phase. The model's only parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Chemisorption Object itself as the const void ∗data structure. NOTE: Only for MultiligandChemisorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandChemisorption object holding parameter information

6.25.5.11 UNIQUAC()

```
int UNIQUAC (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for adsorption reaction object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for AdsorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the AdsorptionReaction object holding parameter information

6.25.5.12 UNIQUAC_unsteady()

```
int UNIQUAC_unsteady (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for unsteady adsorption object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for UnsteadyAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the UnsteadyAdsorption object holding parameter information

6.25.5.13 UNIQUAC_multiligand()

```
int UNIQUAC_multiligand (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand adsorption object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for MultiligandAdsorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandAdsorption object holding parameter information

6.25.5.14 UNIQUAC_chemi()

```
int UNIQUAC_chemi (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for chemisorption reaction object)

This is a more complex surface activity model to be used with the Adsorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Adsorption Object itself as the const void ∗data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for ChemisorptionReaction!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the ChemisorptionReaction object holding parameter information

6.25.5.15 UNIQUAC_multichemi()

```
int UNIQUAC_multichemi (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Surface Activity function for the UNIQUAC model for non-ideal adsorption (for multiligand chemisorption object)

This is a more complex surface activity model to be used with the Chemisorption objects to evaluate the non-ideal behavoir of the surface phase. The model's primary parameters are the shape factors in adsorption and the relative concentrations of each surface species. Therefore, we will pass the Chemisorption Object itself as the const void ∗data structure. However, future development may require some additional parameters, which will be accessed later. NOTE: Only for MultiligandChemisorption!

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to the MultiligandChemisorption object holding parameter information

6.25.5.16 ideal_solution()

```
int ideal_solution (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Activity function for Ideal Solution.

This is one of the default activity models available. It assumes the system behaves ideally and sets the activity coefficients to 1 for all species.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

6.25.5.17 Davies_equation()

```
int Davies_equation (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Activity function for Davies Equation.

This is one of the default activity models available. It uses the Davies semi-empirical model to calculate average activities of each species in solution. This model is typically valid for systems involving high ionic strengths upto 0.5 M (mol/L).

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

6.25.5.18 DebyeHuckel_equation()

```
int DebyeHuckel_equation (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Activity function for Debye-Huckel Equation.

This is one of the default activity models available. It uses the Debye-Huckel limiting model to calculate average activities of each species in solution. This model is typically valid for systems involving low ionic strengths and is only good for solutions between 0 and 0.01 M.

Parameters

x	matrix of the log(C) concentration values at the current non-linear step
F	matrix of activity coefficients that are to be altered by this function
data	pointer to a data structure needed to evaluate the activity model

6.25.5.19 surf_act_choice()

```
int surf_act_choice (
            const std::string & input )
```

First test of SIT Model.

Function takes a given string and returns a flag denoting which surface activity model was choosen This function returns an integer flag that will be one of the valid surface activity model flags from the valid_surf_act enum. If the input string was not recognized, then it defaults to returning the IDEAL_ADS flag.

Parameters

input	string for the name of the surface activity model

6.25.5.20 act_choice()

```
int act_choice (
            const std::string & input )
```

Function takes a given string and returns a flag denoting which activity model was choosen.

This function returns an integer flag that will be one of the valid activity model flags from the valid_act enum. If the input string was not recognized, then it defaults to returning the IDEAL flag.

Parameters

input	string for the name of the activity model

6.25.5.21 reactor_choice()

```
int reactor_choice (
            const std::string & input )
```

Function takes a give string and returns a flag denoting which type of reactor was choosen for the system.

This function returns an integer flag that will be one of the valid reactor type flags from the valid_mb enum. If the input string was not recognized, then it defaults to returning the BATCH flag.

Parameters

input	string for the name of the activity model

6.25.5.22 linesearch_choice()

```
bool linesearch_choice (
            const std::string & input )
```

Function returns a bool to determine the form of line search requested.

This function returns true if the user requests a bouncing line search algorithm and false if the user wants a standard line search. If the input string is unrecognized, then it returns false.

Parameters

input	string for the line search method option

6.25.5.23 linearsolve_choice()

```
int linearsolve_choice (
            const std::string & input )
```

Function returns the linear solver flag for the PJFNK method.

This function takes in a string argument and returns the integer flag for the appropriate linear solver in PJFNK. If the input string was unrecognized, then it returns the GMRESRP flag.

Parameters

input	string for the linear solver method option

6.25.5.24 Convert2LogConcentration()

```
int Convert2LogConcentration (
            const Matrix< double > & x,
            Matrix< double > & logx )
```

Function to convert the given values of variables (x) to the log of those variables (logx)

This function returns an integer flag to denote success of failure. It takes a constant matrix argument x and replaces the elements of the matrix logx with the base 10 log of those x values. This is used mainly to convert a set of concentrations (x) to their respective log(C) values (logx).

Parameters

x	matrix of values to take the base 10 log of
logx	matrix whose entries are to be changed to base 10 log(x)

6.25.5.25 Convert2Concentration()

```
int Convert2Concentration (
            const Matrix< double > & logx,
            Matrix< double > & x )
```

Function to convert the given log values of variables (logx) to the values of those variables (x)

This function returns an integer flag to denote success of failure. It takes a constant matrix argument logx and replaces the elements of the matrix x with 10^logx. This is used mainly to convert a set of log(C) values (logx) to their respective concentration values (x).

Parameters

logx	matrix of values to apply as the power of 10 (i.e., 10^logx)
x	matrix whose entries are to be changed to the result of 10^logx

6.25.5.26 read_scenario()

```
int read_scenario (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for the scenario document.

This function checks the yaml object for the expected keys and values of the scenario document to setup the shark simulation for the input given in the input file.

6.25.5.27 read_multiligand_scenario()

```
int read_multiligand_scenario (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object to setup memory space for multiligand objects.

This function checks the yaml object for the expected keys and values of the multiligand scenario documents to setup the shark simulation for the input given in the input file.

6.25.5.28 read_multichemi_scenario()

```
int read_multichemi_scenario (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object to setup memory space for multiligand chemisorption objects.

This function checks the yaml object for the expected keys and values of the multiligand chemisorption scenario documents to setup the shark simulation for the input given in the input file.

6.25.5.29 read_options()

```
int read_options (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for the solver options document.

This function checks the yaml object for the expected keys and values of the solver options document to setup the shark simulation for the input given in the input file.

6.25.5.30 read_species()

```
int read_species (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for the master species document.

This function checks the yaml object for the expected keys and values of the master species document to setup the shark simulation for the input given in the input file.

6.25.5.31 read_massbalance()

```
int read_massbalance (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for the mass balance document.

This function checks the yaml object for the expected keys and values of the mass balance document to setup the shark simulation for the input given in the input file.

6.25.5.32 read_equilrxn()

```
int read_equilrxn (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for the equilibrium reaction document.

This function checks the yaml object for the expected keys and values of the equilibrium reaction document to setup the shark simulation for the input given in the input file.

6.25.5.33 read_unsteadyrxn()

```
int read_unsteadyrxn (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for the unsteady reaction document.

This function checks the yaml object for the expected keys and values of the unsteady reaction document to setup the shark simulation for the input given in the input file.

6.25.5.34 read_adsorbobjects()

```
int read_adsorbobjects (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for each Adsorption Object.

This function checks the yaml object for the expected keys and values of the adsorption object documents to setup the shark simulation for the input given in the input file.

Note

Each adsorption object will have its own document header by the name of that object

6.25.5.35 read_unsteadyadsorbobjects()

```
int read_unsteadyadsorbobjects (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for each Unsteady Adsorption Object.

This function checks the yaml object for the expected keys and values of the unsteady adsorption object documents to setup the shark simulation for the input given in the input file.

Note

Each unsteady adsorption object will have its own document header by the name of that object

6.25.5.36 read_multiligandobjects()

```
int read_multiligandobjects (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for each MultiligandAdsorption Object.

This function checks the yaml object for the expected keys and values of the multiligand object documents to setup the shark simulation for the input given in the input file.

Note

Each ligand object will have its own document header by the name of that object

6.25.5.37 read_chemisorbobjects()

```
int read_chemisorbobjects (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for each Chemisorption Object.

This function checks the yaml object for the expected keys and values of the chemisorption object documents to setup the shark simulation for the input given in the input file.

Note

Each adsorption object will have its own document header by the name of that object

6.25.5.38 read_multichemiobjects()

```
int read_multichemiobjects (
            SHARK_DATA * shark_dat )
```

Function to go through the yaml object for each MultiligandChemisorption Object.

This function checks the yaml object for the expected keys and values of the multiligand chemisorption object documents to setup the shark simulation for the input given in the input file.

Note

Each ligand object will have its own document header by the name of that object

6.25.5.39 setup_SHARK_DATA()

```
int setup_SHARK_DATA (
            FILE * file,
            int(*)(const Matrix< double > &x, Matrix< double > &res, const void *data) residual,
            int(*)(const Matrix< double > &x, Matrix< double > &gama, const void *data)
activity,
            int(*)(const Matrix< double > &r, Matrix< double > &p, const void *data) precond,
            SHARK_DATA * dat,
            const void * activity_data,
            const void * residual_data,
            const void * precon_data,
            const void * other_data )
```

Function to setup the memory and pointers for the SHARK_DATA structure for the current simulation.

This function will be called after reading the scenario file and is used to setup the memory and other pointers for the user requested simulation. This function must be called before running a simulation or trying to read in the remander of the yaml formatted input file. Options may be overriden manually after calling this function.

Parameters

file	pointer for the output file where shark results will be stored
residual	pointer to the residual function that will be fed into the PJFNK solver
activity	pointer to the activity function that will determine the activity coefficients
precond	pointer to the linear preconditioning operation to be applied to the Jacobian
dat	pointer to the SHARK_DATA data structure
activity_data	optional pointer for data needed in activity functions
residual_data	optional pointer for data needed in residual functions
precon_data	optional pointer for data needed in preconditioning functions
other_data	optional pointer for data needed in the evaluation of user defined residual functions

6.25.5.40 shark_add_customResidual()

```
int shark_add_customResidual (
            int i,
            double(*)(const Matrix< double > &x, SHARK_DATA *shark_dat, const void *other_↩
data) other_res,
            SHARK_DATA * shark_dat )
```

Function to add user defined custom residual functions to the OtherList vector object in SHARK_DATA.

This function will need to be used if the user wants to include custom residuals into the system via the OtherList object in SHARK_DATA. For each i residual you want to add, you must call this function passing your residual function and the SHARK_DATA structure pointer. The order that those functions are executed in are determined by the integer i.

Parameters

i	index that the other_res function will appear at in the OtherList object
other_res	function pointer for the user's custom residual function
shark_dat	pointer to the SHARK_DATA data structure

6.25.5.41 shark_parameter_check()

```
int shark_parameter_check (
            SHARK_DATA * shark_dat )
```

Function to check the Reaction and UnsteadyReaction objects for missing info.

This function checks the Reaction and UnsteadyReaction objects for missing information. If information is missing, this function will return an error that will cause the program to force quit.

6.25.5.42 shark_energy_calculations()

```
int shark_energy_calculations (
            SHARK_DATA * shark_dat )
```

Function to calculate all Reaction and UnsteadyReaction energies.

This function will call the calculate energy functions for Reaction and UnsteadyReaction objects.

6.25.5.43 shark_temperature_calculations()

```
int shark_temperature_calculations (
            SHARK_DATA * shark_dat )
```

Function to calculate all Reaction and UnsteadyReaction parameters as a function of temperature.

This function will call all temperature dependent functions in Reaction and UnsteadyReaction to calculate equilibrium and reaction rate parameters as a function of system temperature.

6.25.5.44 shark_pH_finder()

```
int shark_pH_finder (
            SHARK_DATA * shark_dat )
```

Function will search MasterSpeciesList for existance of H + (aq) and OH - (aq) molecules.

This function searches all molecules in the MasterSpeciesList object for the H + (aq) and OH - (aq) molecules. If they are found, then it sets the pH_index and pOH_index of the SHARK_DATA structure and indicates that the system contains these variables.

6.25.5.45 shark_guess()

```
int shark_guess (
            SHARK_DATA * shark_dat )
```

Function provides a rough initial guess for the values of all non-linear variables.

This function constructs an rough initial guess for the values of all non-linear variables in the system. The guess is based primarily off of trying to satisfy all mass balance constraints, initial conditions, and pH constraints if any apply.

6.25.5.46 shark_initial_conditions()

```
int shark_initial_conditions (
            SHARK_DATA * shark_dat )
```

Function to establish the initial conditions of the shark simulation.

This function will establish the initial conditions for a transient problem by solving the speciation of the system while holding the transient/unsteady variables constant at their respective initial values. However, if the system we are trying to solve is steady, then this function just calls the shark_guess function.

6.25.5.47 shark_executioner()

```
int shark_executioner (
            SHARK_DATA * shark_dat )
```

Function to execute a shark simulation at a single time step or pH value.

This function calls the preprocess, solver, and postprocess functions in order. If a particular solve did not converge, then it will retry the solver routine until it runs out of tries or attains convergence.

6.25.5.48 shark_timestep_const()

```
int shark_timestep_const (
            SHARK_DATA * shark_dat )
```

Function to set up all time steps in the simulation to a specified constant.

This function will set all time steps for the current simulation to a constant that is specified in the input file. The time step will not be changed unless the simulation fails, then it will be reduced in order to try to get the system to converge.

6.25.5.49 shark_timestep_adapt()

```
int shark_timestep_adapt (
            SHARK_DATA * shark_dat )
```

Function to set up all time steps in the simulation based on success or failure to converge.

This function will set all time steps for the current simulation based on some factor multiple of the prior time step used and whether or not the previous solution step was successful. If the previous step converged, then the new time step will be 1.5x the old time step. If it failed, then the simulation will be retried with a new time step of 0.5x the old time step.

6.25.5.50 shark_preprocesses()

```
int shark_preprocesses (
            SHARK_DATA * shark_dat )
```

Function to call other functions for calculation of parameters and setting of time steps.

This function will call the shark_temperature_calculations function and the appropriate time step function. If the user requests a constant time step, it will call the shark_timestep_const function. Otherwise, it calls the shark_↩
timestep_adapt function.

6.25.5.51 shark_solver()

```
int shark_solver (
            SHARK_DATA * shark_dat )
```

Function to call the PJFNK solver routine given the current SHARK_DATA information.

This function will perform the necessary steps before and after calling the PJFNK solver routine. Based on the simulation flags, the solver function will perform an intial guess for unsteady variables, call the PJFNK method, and the printout a console message about the performance. If a terminal failure occurs during the solver, it will print out the current state of residuals, variables, and the Jacobian matrix to the console. Analyzing this information could provide clues as to why failure occured.

6.25.5.52 shark_postprocesses()

```
int shark_postprocesses (
            SHARK_DATA * shark_dat )
```

Function to convert PJFNK solutions to concentration values and print to the output file.

This function will convert the non-linear variables to their respective concentration values, then print the solve information out to the output file.

6.25.5.53 shark_reset()

```
int shark_reset (
            SHARK_DATA * shark_dat )
```

Function to reset the values of all stateful information in SHARK_DATA.

This function will reset all stateful matrix data in the SHARK_DATA structure in preparation of the next time step simulation.

6.25.5.54 shark_residual()

```
int shark_residual (
            const Matrix< double > & x,
            Matrix< double > & F,
            const void * data )
```

Default residual function for shark evaluations.

This function calls each individual object's residual function to formulate the overall residual function used in the P↩
JFNK solver routine. It will also call the activity function. The order in which these function calls occurs is as follows: (i) activities, (ii) Reaction, (iii) UnsteadyReaction, (iv) MassBalance, (v) OtherList, and (vi) MasterSpeciesList. If a constant pH is specified, then the MasterSpeciesList residual call is replaced with a constraint on the H + (aq) variable (if one exists).

6.25.5.55 SHARK()

```
int SHARK (
            SHARK_DATA * shark_dat )
```

Function to call all above functions to perform a shark simulation.

This function is called after reading in all inputs, setting all constants, and calling the setup function. It will call all the necessary functions and subroutines iteratively until the desired simulation is complete.

6.25.5.56 SHARK_SCENARIO()

```
int SHARK_SCENARIO (
            const char * yaml_input )
```

Function to perform a shark simulation based on the conditions in a yaml formatted input file.

This is the primary function used to run shark simulations from the UI. It requires that ths user provide one input file that is formatted with yaml keys, symbols, and spacing so that it can be recognized by the parser. This style of input file is much easier to use and understand than the input files used for SCOPSOWL or SKUA. Below shows an example of a typical input file. Note that the # symbol is used in the input file to comment out lines of text that the parser does not need to read.

Example Yaml Input for SHARK

#This will serve as a test input file for shark to demo how to structure the document
#In practice, this section should be listed first, but it doesn't really matter
#DO NOT USE TABS IN THESE INPUT FILES
#— Starts a document ... Ends a document
#All keys must be proceeded by a :
#All lists/header must be preceeded by a -
#Spacing of the keys will indicate which list/header they belong to
Scenario:
—

- vars_fun:
  numvar: 25
  num_ssr: 15
  num_mbe: 7
  num_usr: 2
  num_other: 0 #Not required or used in current version

- sys_data:
  act_fun: davies
  const_pH: false
  pH: 7 #Only required if we are specifying a const_pH
  temp: 298.15 #Units must be in Kelvin
  dielec: 78.325 #Units must be in (1/Kelvin)
  rel_perm: 80.1 #Unitless number
  res_alk: 0 #Units must be in mol/L (Residual Alkalinity)
  volume: 1.0 #Units must be in L

- run_time:
  steady: false #NOTE: All time must be represented in hours
  specs_curve: false #Only needed if steady = true, and will default to false
  dt: 0.001 #Only required if steady = false
  time_adapt: true #Only needed if steady = false, and will default to false
  sim_time: 96.0 #Only required if steady = false
  t_out: 0.01 #Only required if steady = false
  ...

#The following header is entirely optional, but is used to set solver options
SolverOptions:
—
line_search: true #Default = true, and is recommended to be true
search_type: standard
linear_solve: gmresrp #Note: FOM will be fastest for small problems
restart: 25 #Note: restart only used if using GMRES or GCR type solvers
nl_maxit: 50
nl_abstol: 1e-5
nl_reltol: 1e-8
lin_reltol: 1e-10 #Min Tol = 1e-15
lin_abstol: 1e-10 #Min Tol = 1e-15
nl_print: true
l_print: true
...

#After the Scenario read, shark will call the setup_function, then read info below
MasterSpecies:
—
#Header names are specific
#Keys are chosen by user, but must span numbers 0 through numvar-1
#Keys will denote the ordering of the variables
#Note: Currently, the number of reg molecules is very limited

- reg:
  0: Cl - (aq)
  1: NaHCO3 (aq)
  2: NaCO3 - (aq)
  3: Na + (aq)
  4: HNO3 (aq)
  5: NO3 - (aq)
  6: H2CO3 (aq)
  7: HCO3 - (aq)
  8: CO3 2- (aq)
  9: UO2 2+ (aq)
  10: UO2NO3 + (aq)
  11: UO2(NO3)2 (aq)
  12: UO2OH + (aq)
  13: UO2(OH)3 - (aq)
  14: (UO2)2(OH)2 2+ (aq)
  15: (UO2)3(OH)5 + (aq)
  16: UO2CO3 (aq)
  17: UO2(CO3)2 2- (aq)
  18: UO2(CO3)3 4- (aq)
  19: H2O (l)
  20: OH - (aq)
  21: H + (aq)
  #Keys for the sub-headers must follow same rules as keys from above

- unreg:

  - 22:
    formula: A(OH)2 (aq)
    charge: 0
    enthalpy: 0
    entropy: 0
    have_HS: false
    energy: 0
    have_G: false
    phase: Aqueous
    name: Amidoxime
    lin_form: none

  - 23:
    formula: UO2AO2 (aq)
    charge: 0
    enthalpy: 0
    entropy: 0
    have_HS: false
    energy: 0
    have_G: false
    phase: Aqueous
    name: Uranyl-amidoximate
    lin_form: none

  - 24:
    formula: UO2CO3AO2 2- (aq)
    charge: -2
    enthalpy: 0
    entropy: 0
    have_HS: false
    energy: 0
    have_G: false
    phase: Aqueous
    name: Uranyl-carbonate-amidoximate
    lin_form: none
    ...
    #NOTE: Total concentrations must be given in mol/L
    MassBalance:
    —
    #Header names under MassBalance are choosen by the user
    #All other keys will be checked

- water:
  total_conc: 1

  - delta:
    "H2O (l)": 1

- carbonate:
  total_conc: 0.0004175

  - delta:
    "NaHCO3 (aq)": 1
    "NaCO3 - (aq)": 1
    "H2CO3 (aq)": 1
    "HCO3 - (aq)": 1
    "CO3 2- (aq)": 1
    "UO2CO3 (aq)": 1
    "UO2(CO3)2 2- (aq)": 2
    "UO2(CO3)3 4- (aq)": 3
    "UO2CO3AO2 2- (aq)": 1
    #Other mass balances skipped for demo purposes...
    ...
    #Document for equilibrium or steady reactions
    EquilRxn:
    —
    #Headers under EquilRxn separate out each reaction object
    #Keys for these headers only factor into the order of the equations
    #Stoichiometry follows the convention that products are pos(+) and reactants are neg(-)
    #Note: logK is only required if any species in stoichiometry is unregistered
    #Example: below represents - {H2O (l)} –> {H + (aq)} + {OH - (aq)}
    #Note: a valid reaction statement requires at least 1 stoichiometry args
    #Note: You can also provide reaction energies: enthalpy, entropy, and energy

- rxn00:
  logK: -14

  - stoichiometry:
    "H2O (l)": -1
    "OH - (aq)": 1
    "H + (aq)": 1

- rxn01:
  logK: -6.35

  - stoichiometry:
    "H2CO3 (aq)": -1
    "HCO3 - (aq)": 1
    "H + (aq)": 1
    #Other reactions skipped for demo purposes...
    ...
    #Document for unsteady reactions
    UnsteadyRxn:
    —
    #Same basic standards for this doc as the EquilRxn
    #Main difference is the inclusion of rate information
    #You are required to give at least 1 rate
    #You are also required to denote which variable is unsteady
    #You must give the initial concentration for the variable in mol/L
    #Rate units are in (L/mol)^n/hr
    #Note: we also have keys for forward_ref, reverse_ref,
    #activation_energy, and temp_affinity.
    #These are optional if forward and/or reverse are given
    #Note: You can also provide reaction energies: enthalpy, entropy, and energy

- rxn00:
  unsteady_var: UO2AO2 (aq)
  initial_condition: 0
  logK: -1.35
  forward: 4.5e+6
  reverse: 1.00742e+8

  - stoichiometry:
    "UO2 2+ (aq)": -1
    "A(OH)2 (aq)": -1
    "UO2AO2 (aq)": 1
    "H + (aq)": 2

- rxn01:
  unsteady_var: UO2CO3AO2 2- (aq)
  initial_condition: 0
  logK: 3.45
  forward: 2.55e+15
  reverse: 9.04774e+11

  - stoichiometry:
    "UO2 2+ (aq)": -1
    "CO3 2- (aq)": -1
    "A(OH)2 (aq)": -1
    "UO2CO3AO2 2- (aq)": 1
    "H + (aq)": 2
    ...

    Note

    It may be advantageous to look at some other shark input file examples. More input files are provided in the input_files/SHARK directory of the ecosystem project folder. Please refer to your own source file location for more input file examples for SHARK.

6.25.5.57 SHARK_TESTS()

```
int SHARK_TESTS ( )
```

Function to perform a series of shark calculation tests.

This function sets up and solves a test problem for shark. It is callable from the UI.

6.25.5.58 SHARK_TESTS_OLD()

```
int SHARK_TESTS_OLD ( )
```

Function to perform a series of shark calculation tests (older version)

This function sets up and solves a test problem for shark. It is NOT callable from the UI.

6.26 skua.h File Reference

Surface Kinetics for Uptake by Adsorption.

```
#include "finch.h"
#include "magpie.h"
#include "egret.h"
```

Classes

- struct SKUA_PARAM

    Data structure for species' parameters in SKUA.
- struct SKUA_DATA

    Data structure for all simulation information in SKUA.

Macros

- #define SKUA_HPP_
- #define D_inf(Dref, Tref, B, p, T) (Dref ∗ pow(p+sqrt(DBL_EPSILON),(Tref/T)-B))

    Empirical correction of diffusivity (um^2/hr)
- #define D_o(Diff, E, T) (Diff ∗ exp(-E/(Rstd∗T)))

    Arrhenius Rate Expression for Diffusivity (um^2/hr)
- #define D_c(Diff, phi) (Diff ∗ (1.0/((1.0+1.1E-6)-phi)))

    Approximate Darken Diffusivity Equation (um^2/hr)

Functions

- void print2file_species_header (FILE ∗Output, SKUA_DATA ∗skua_dat, int i)

    Function to print out the species' headers to output file.
- void print2file_SKUA_time_header (FILE ∗Output, SKUA_DATA ∗skua_dat, int i)

    Function to print out time and space headers to output file.
- void print2file_SKUA_header (SKUA_DATA ∗skua_dat)

    Function calls the other header functions to establish output file structure.
- void print2file_SKUA_results_old (SKUA_DATA ∗skua_dat)

    Function to print out the old time step simulation results to the output file.
- void print2file_SKUA_results_new (SKUA_DATA ∗skua_dat)

    Function to print out the new time step simulation results to the output file.
- double default_Dc (int i, int l, const void ∗data)

    Default function for surface diffusivity.
- double default_kf (int i, const void ∗data)

    Default function for film mass transfer coefficent.
- double const_Dc (int i, int l, const void ∗data)

    Constant surface diffusivity function.
- double simple_darken_Dc (int i, int l, const void ∗data)

    Simple Darken model for surface diffusivity.
- double theoretical_darken_Dc (int i, int l, const void ∗data)

    Theoretical Darken model for surface diffusivity.
- double empirical_kf (int i, const void ∗data)

Empirical function for film mass transfer coefficent.

- double const_kf (int i, const void ∗data)

    Constant function for film mass transfer coefficent.

- int molefractionCheck (SKUA_DATA ∗skua_dat)

    Function to check mole fractions in gas and solid phases for errors.

- int setup_SKUA_DATA (FILE ∗file, double(∗eval_Dc)(int i, int l, const void ∗user_data), double(∗eval_Kf)(int i, const void ∗user_data), const void ∗user_data, MIXED_GAS ∗gas_data, SKUA_DATA ∗skua_dat)

    Function to setup the function pointers and vector objects in memory to setup the SKUA simulation.

- int SKUA_Executioner (SKUA_DATA ∗skua_dat)

    Function to execute preprocesses, solvers, and postprocesses for a SKUA simulation.

- int set_SKUA_ICs (SKUA_DATA ∗skua_dat)

    Function to establish the initial conditions of adsorption in the adsorbent.

- int set_SKUA_timestep (SKUA_DATA ∗skua_dat)

    Function to establish the time step for the current simulation.

- int SKUA_preprocesses (SKUA_DATA ∗skua_dat)

    Function to perform the necessary preprocess operations before a solve.

- int set_SKUA_params (const void ∗user_data)

    Function to call the diffusivity function during the solve.

- int SKUA_postprocesses (SKUA_DATA ∗skua_dat)

    Function to perform the necessary postprocess operations after a solve.

- int SKUA_reset (SKUA_DATA ∗skua_dat)

    Function to reset the stateful information in SKUA after a simulation.

- int SKUA (SKUA_DATA ∗skua_dat)

    Function to iteratively call all execution steps to evolve a simulation through time.

- int SKUA_SCENARIOS (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate)

    Function callable from the UI to perform a SKUA simulation based on user supplied input files.

- int SKUA_TESTS ()

    Function to perform a test of the SKUA functions and routines.

6.26.1 Detailed Description

Surface Kinetics for Uptake by Adsorption.

skua.cpp

This file contains structures and functions associated with solving the surface diffusion partial differential equations for adsorption kinetics in spherical and/or cylindrical adsorbents. For this system, it is assumed that the pore size is so small that all molecules are confined to movement exclusively on the surface area of the adsorbent. The total amount of adsorption for each species is drive by the MAGPIE model for non-ideal mixed gas adsorption. Spatial and temporal varience in adsorption is caused by a combination of different kinetics between adsorbing species and different adsorption affinities for the surface.

The function for surface diffusion involves four parameters, although not all of these parameters are required to be used. Surface diffusion theoretically varies with temperature according to the Arrhenius rate expression, but we also add in an empirical correction term to account for variations in diffusivity with the partial pressure of the species in the gas phase.
D_surf = D_ref ∗ exp(-E / (R∗T)) ∗ pow(p , (T_ref/T) - B)
D_ref is the Reference Diffusivity (um^2/hr), E is the activation energy for adsorption (J/mol), R is the gas law constant (J/K/mol), T is the system temperature (K), p is the partial pressure of the adsorbing species (kPa), T_ref is the Reference Temperature (K), and B is the Affinity constant.

Author

Austin Ladshaw

Date

01/26/2015

Copyright

6.26.2 Macro Definition Documentation

6.26.2.1 SKUA_HPP_

```
#define SKUA_HPP_
```

6.26.2.2 D_inf

```
#define D_inf(
              Dref,
              Tref,
              B,
              p,
              T ) ( Dref * pow(p+sqrt(DBL_EPSILON),(Tref/T)-B) )
```

Empirical correction of diffusivity (um^2/hr)

6.26.2.3 D_o

```
#define D_o(
              Diff,
              E,
              T ) ( Diff * exp(-E/(Rstd*T)) )
```

Arrhenius Rate Expression for Diffusivity (um^2/hr)

6.26.2.4 D_c

```
#define D_c(
              Diff,
              phi ) ( Diff * (1.0/((1.0+1.1E-6)-phi) ) )
```

Approximate Darken Diffusivity Equation (um^2/hr)

6.26.3 Function Documentation

6.26.3.1 print2file_species_header()

```
void print2file_species_header (
            FILE * Output,
            SKUA_DATA * skua_dat,
            int i )
```

Function to print out the species' headers to output file.

6.26.3.2 print2file_SKUA_time_header()

```
void print2file_SKUA_time_header (
            FILE * Output,
            SKUA_DATA * skua_dat,
            int i )
```

Function to print out time and space headers to output file.

6.26.3.3 print2file_SKUA_header()

```
void print2file_SKUA_header (
            SKUA_DATA * skua_dat )
```

Function calls the other header functions to establish output file structure.

6.26.3.4 print2file_SKUA_results_old()

```
void print2file_SKUA_results_old (
            SKUA_DATA * skua_dat )
```

Function to print out the old time step simulation results to the output file.

6.26.3.5 print2file_SKUA_results_new()

```
void print2file_SKUA_results_new (
            SKUA_DATA * skua_dat )
```

Function to print out the new time step simulation results to the output file.

6.26.3.6 default_Dc()

```
double default_Dc (
            int i,
            int l,
            const void * data )
```

Default function for surface diffusivity.

This is the default function provided by SKUA for the calculation of the surface diffusivity parameter. The diffusivity is calculated based on the Arrhenius rate expression, then corrected for using the empirical correction term with the outside partial pressure of the gas species.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.7 default_kf()

```
double default_kf (
            int i,
            const void * data )
```

Default function for film mass transfer coefficent.

This is the default function provided by SKUA for the calculation of the film mass transfer parameter. By default, we are usually going to couple the SKUA model with a pore diffusion model (see scopsowl.h). Therefore, the film mass transfer coefficient would be zero, because we would only consider a Dirichlet boundary condition for this sub-problem.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.8 const_Dc()

```
double const_Dc (
            int i,
            int l,
            const void * data )
```

Constant surface diffusivity function.

This function allows the user to specify just a single constant value for surface diffusivity. The value of diffusivity applied at all nodes will be the ref_diffusion parameter in SKUA_PARAM.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.9 simple_darken_Dc()

```
double simple_darken_Dc (
            int i,
            int l,
            const void * data )
```

Simple Darken model for surface diffusivity.

This function uses an approximation to Darken's model for surface diffusion. The approximation is exact if the isotherm for adsorption takes the form of the Langmuir model, but is only approximate if the isotherm is heterogeneous. Forming the approximation in this manner is significantly cheaper than forming the true Darken model expression for the GSTA isotherm.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.10 theoretical_darken_Dc()

```
double theoretical_darken_Dc (
            int i,
            int l,
            const void * data )
```

Theoretical Darken model for surface diffusivity.

This function uses the full theoretical expression of the Darken's diffusion model to calculate the surface diffusivity. This calculation involves formulating the reference state pressures for the adsorbed amount at every node, then calculating derivatives of the adsorption isotherm for each species. It is more accurate than the simple Darken model function, but costs significantly more computational time.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
l	index of the node in the spatial discretization that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.11 empirical_kf()

```
double empirical_kf (
            int i,
            const void * data )
```

Empirical function for film mass transfer coefficent.

This function provides an empirical estimate of the mass transfer coefficient using the gas velocity, molecular diffusivities, and dimensionless numbers (see egret.h). It is used as the default film mass transfer function IF the boundary condition is specified to be a Neumann type boundary by the user.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.12 const_kf()

```
double const_kf (
            int i,
            const void * data )
```

Constant function for film mass transfer coefficent.

This function allows the user to specify a constant value for the film mass transfer coefficient. The value of the film mass transfer coefficient will be the value of film_transfer given in the SKUA_PARAM data structure.

Parameters

i	index of the gas/adsorbed phase species that this function acts on
data	pointer to the SKUA_DATA structure

6.26.3.13 molefractionCheck()

```
int molefractionCheck (
            SKUA_DATA * skua_dat )
```

Function to check mole fractions in gas and solid phases for errors.

This function is called after reading input and before calling the primary solution routines. It will force and error and quit the program if their are inconsistencies in the mole fractions it was given. All mole fractions must sum to 1, otherwise there is missing information.

6.26.3.14 setup_SKUA_DATA()

```
int setup_SKUA_DATA (
            FILE * file,
            double(*)(int i, int l, const void *user_data) eval_Dc,
            double(*)(int i, const void *user_data) eval_Kf,
            const void * user_data,
            MIXED_GAS * gas_data,
            SKUA_DATA * skua_dat )
```

Function to setup the function pointers and vector objects in memory to setup the SKUA simulation.

This function is called to setup the SKUA problem in memory and set function pointers to either defaults or user specified functions. It must be called prior to calling any other SKUA function and will report an error if the object was not setup properly.

Parameters

file	pointer to the output file for SKUA simulations
eval_Dc	pointer to the function to evaluate the surface diffusivity
eval_Kf	pointer to the function to evaluate the film mass transfer coefficient
user_data	pointer to a user defined data structure used in the calculation the the parameters
gas_data	pointer to the MIXED_GAS data structure for egret.h calculations
skua_dat	pointer to the SKUA_DATA data structure

6.26.3.15 SKUA_Executioner()

```
int SKUA_Executioner (
            SKUA_DATA * skua_dat )
```

Function to execute preprocesses, solvers, and postprocesses for a SKUA simulation.

This function calls the preprocess, solver, and postprocess functions to complete a single time step in a SKUA simulation. User's will want to call this function whenever a time step simulation result is needed. This is used primarily when coupling with other models (see scopsowl.h).

6.26.3.16 set_SKUA_ICs()

```
int set_SKUA_ICs (
            SKUA_DATA * skua_dat )
```

Function to establish the initial conditions of adsorption in the adsorbent.

This function needs to be called before doing any simulation or execution of a time step, but only once per simulation. It sets the value of adsorption for each adsorbable species to the specified initial values given via qT and xIC in SKUA_DATA.

6.26.3.17 set_SKUA_timestep()

```
int set_SKUA_timestep (
            SKUA_DATA * skua_dat )
```

Function to establish the time step for the current simulation.

This function is called to set a time step value for a particular simulation step. By default, the time step is set to (1/4)x space step size. If you need to change the step size, you must do so manually.

6.26.3.18 SKUA_preprocesses()

```
int SKUA_preprocesses (
            SKUA_DATA * skua_dat )
```

Function to perform the necessary preprocess operations before a solve.

This function performs preprocess operations prior to calling the solver routine. Those preprocesses include establishing boundary conditions and performing a MAGPIE simulation for the adsorption on the surface (see magpie.h).

6.26.3.19 set_SKUA_params()

```
int set_SKUA_params (
            const void * user_data )
```

Function to call the diffusivity function during the solve.

This is the function passed into FINCH to be called during the FINCH solver (see finch.h). It will call the diffusion functions set by the user in the setup function above. This is not overridable.

6.26.3.20 SKUA_postprocesses()

```
int SKUA_postprocesses (
            SKUA_DATA * skua_dat )
```

Function to perform the necessary postprocess operations after a solve.

This function performs postprocess operations after a solve was completed successfully. Those operations include estimating average total adsorption, average adsorbed mole fractions, and heat of adsorption for each species. Results are then printed to the output file.

6.26.3.21 SKUA_reset()

```
int SKUA_reset (
            SKUA_DATA * skua_dat )
```

Function to reset the stateful information in SKUA after a simulation.

This function sets all the old state data to the newly formed state data. It needs to be called after a successful execution of the simulation step and before calling for the next time step to be solved. Do not call out of turn, otherwise information will be lost.

6.26.3.22 SKUA()

```
int SKUA (
            SKUA_DATA * skua_dat )
```

Function to iteratively call all execution steps to evolve a simulation through time.

This function is used in conjunction with the scenario call from the UI to numerically solve the adsorption kinetics problem in time. It will call the initial conditions function once, then iteratively call the reset, time step, and executioner functions for SKUA to push the simulation forward in time. This function will be called from the SKUA_SCENARIOS function.

6.26.3.23 SKUA_SCENARIOS()

```
int SKUA_SCENARIOS (
            const char * scene,
            const char * sorbent,
            const char * comp,
            const char * sorbate )
```

Function callable from the UI to perform a SKUA simulation based on user supplied input files.

This is the primary function to be called when running a stand-alone SKUA simulation. Parameters and system information for the simulation are given in a series of input files that come in as character arrays. These inputs are all required to call this function.

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File

Note

Each input file has a particular format that must be strictly adhered to in order for the simulation to be carried out correctly. The format for each input file, and an example, is provided below...

Scenario Input Format

System Temperature (K) [tab] Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Simulation Time (hrs) [tab] Print Out Time (hrs)
BC Type (0 = Neumann, 1 = Dirichlet)
Number of Gas Species
Initial Total Adsorption (mol/kg)
Name of ith Species [tab] Adsorbable? (0 = false, 1 = true) [tab] Gas Phase Molefraction [tab] Initial Sorbed Molefraction
(repeat above for all species)

Example Scenario Input

353.15 101.35 0.36
4.0 0.05
0
5
0.0
N2 0 0.7634 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0191 0.0
Above example is for a 5-component mixture of N2, O2, Ar, CO2, and H2O, but we are only considering the H2O as adsorbable.

Adsorbent Input File

Domain Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (um) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (um)

Example Adsorbent Input

1 6.0
2.0
Above example is for a cylindrical adsorbent with a length of 5 um and radius of 2 um.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846
0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is a continuation of the Scenario Input example wherein each grouping represents parameters that are associated with N2, O2, Ar, CO2, and H2O, respectively. The order is VERY important!

Adsorbate Input File

Type of Surface Diffusion Function (0 = constant, 1 = simple Darken, 2 = theoretical Darken)
Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0
0.8814 0.0
267.999 0.0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
1.28 540.1
374.99 0.01
3.01
1.27
2
-46597.5 -53.6994
-125024 -221.073
Above example would be for a simulation involving two adsorbable species using a constant surface diffusion function. Each adsorbable species has it's own set of kinetic and equilibrium parameters that must be given in the same order as the species appeared in the Scenario Input. Note: we do not need to supply this information for non-adsorbable species.

6.26.3.24 SKUA_TESTS()

```
int SKUA_TESTS ( )
```

Function to perform a test of the SKUA functions and routines.

This function is callable from the UI and will perform a test simulation of the SKUA system of equations. Results from that test are output into a sub-directory called output and named SKUA_Test_Output.txt.

6.27 skua_opt.h File Reference

Optimization Routine for the SKUA Model.

```
#include "skua.h"
```

Classes

- struct SKUA_OPT_DATA

  Data structure for the SKUA Optimization Routine.

Functions

- int SKUA_OPT_set_y (SKUA_OPT_DATA ∗skua_opt)

  Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.
- int initial_guess_SKUA (SKUA_OPT_DATA ∗skua_opt)

  Function to set up an initial guess for the surface diffusivity parameter in SKUA.
- void eval_SKUA_Uptake (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)

  Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.
- int SKUA_OPTIMIZE (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate, const char ∗data)

  Function called to perform the optimization routine given a specific set of information and data.

6.27.1 Detailed Description

Optimization Routine for the SKUA Model.

skua_opt.cpp

This file contains structures and functions associated with performing non-linear least-squares optimization of the SKUA simulation results against actual kinetic adsorption data. The optimization routine here allows you to run data comparisons and optimizations in three forms: (i) Rough optimizations - cheaper operations, but less accurate, (ii) Exact optmizations - much more expensive, but greater accuracy, and (iii) data/model comparisons - no optimization, just using system parameters to compare simulation results agains a set of data.

Depending on the level of optimization desired, this routine could take several minutes or several hours. The optimization/comparisons are printed out in two files: (i) a parameter file, which contains the simulation partial pressures and temperatures and the optimized diffusivities with the euclidean norm of the fitting and (ii) a comparison file that shows the model value and data value at each time step for each kinetic curve.

The optimized diffusion parameters are given for each individual kinetic data curve. Each data curve will have a different pairing of partial pressure and temperature. Because of this, you will get a list of different diffusivities for each data curve. To get the optimum kinetic parameters from this list of diffusivities, you must fit the diffusion parameter values to the following diffusion function model...
D_opt = D_ref ∗ exp(-E / (R∗T)) ∗ pow(p , (T_ref/T) - B)
where D_ref is the Reference Diffusivity (um^2/hr), E is the activation energy for adsorption (J/mol), R is the gas law constant (J/K/mol), T is the system temperature (K), p is the partial pressure of the adsorbing species (kPa), T_ref is the Reference Temperature (K), and B is the Affinity constant. This algorithm does not automatically produce these parameters for you, but gives you everything you need to produce them yourself.
This routine allows you to optimize multiple kinetic curves at one time. However, all data must be for the same adsorbent-adsorbate system. In other words, the adsorbent and adsorbate pair must be the same for each kinetic curve analyzed. Also, each experiment must have been done in a thin bed or continuous flow system where the adsorbents were exposed to a nearly constant outside partial pressure for all time steps and the gas velocity of that system is assumed constant for all experiments. This experimental setup is very typical for studying adsorption kinetics for gas-solid systems.

Author

Austin Ladshaw

Date

05/11/2015

Copyright

6.27.2 Function Documentation

6.27.2.1 SKUA_OPT_set_y()

```
int SKUA_OPT_set_y (
            SKUA_OPT_DATA * skua_opt )
```

Function to set the rest of the gas phase mole fractions based on current mole fraction of adsorbing gas.

This function takes the current mole fraction of the adsorbing gas and calculates the gas mole fractions of the other gases in the sytem based on the standard inlet gas composition given in the scenario file.

6.27.2.2 initial_guess_SKUA()

```
int initial_guess_SKUA (
            SKUA_OPT_DATA * skua_opt )
```

Function to set up an initial guess for the surface diffusivity parameter in SKUA.

This function performs the Rough optimization on the surface diffusivity based on the idea of reducing or eliminating function bias between data and simulation. A positive function bias means that the simulation curve is "higher" than the data curve and a negative function bias means that the simulation curve is "lower" than the data curve. We use this information to incrementally adjust the rate of surface diffusion until this bias is near zero. When bias is near zero, the simulation is nearly optimized, but further refinement may be necessary to find the true minimum solution.

6.27.2.3 eval_SKUA_Uptake()

```
void eval_SKUA_Uptake (
             const double * par,
             int m_dat,
             const void * data,
             double * fvec,
             int * info )
```

Function that works in conjunction with the lmfit routine to minimize the euclidean norm between function and data.

This function will run the SKUA simulation at a given value of surface diffusivity and produce residuals that feed into the Levenberg-Marquardt's algorithm for non-linear least-squares regression. The form of this function is specific to the format required by the lmfit routine.

Parameters

par	array of parameters that are to be optimized
m_dat	number of data points or functions to evaluate
data	user supplied data structure holding information necessary to form the residuals
fvec	array of residuals computed at the current parameter values
info	integer pointer denoting whether or not the user requests to end a particular simulation

6.27.2.4 SKUA_OPTIMIZE()

```
int SKUA_OPTIMIZE (
             const char * scene,
             const char * sorbent,
             const char * comp,
             const char * sorbate,
             const char * data )
```

Function called to perform the optimization routine given a specific set of information and data.

This is the function that is callable by the UI. The user must provide 5 input files to the routine in order to establish simulation conditions, adsborbent properties, component properties, adsorbate equilibrium parameters, and the set of data that we are comparing the simulations to. Each input file has a very specific structure and order to the information that it contains. The structure here is DIFFERENT than the structure for just running standard SKUA simulations (see skua.h).

Parameters

scene	Sceneario Input File
sorbent	Adsorbent Input File
comp	Component Input File
sorbate	Adsorbate Input File
data	Kinetic Adsorption Data File

Note

Much of the structure of these input files are "similar" to that of the input files used in SKUA_SCENARIOS (see skua.h), but with some notable differences. Below gives the format for each input file with an example. Make sure your input files follow this format before calling this routine from the UI.

Scenario Input File

Optimization? (0 = false, 1 = true) [tab] Rough Optimization? (0 = false, 1 = true)
Surf. Diff. (0 = constant, 1 = simple Darken, 2 = theoretical Darken) [tab] BC Type (0 = Neumann, 1 = Dirichlet)
Total Pressure (kPa) [tab] Gas Velocity (cm/s)
Number of Gaseous Species
Initial Adsorption Total (mol/kg)
Name [tab] Adsorbable? (0 = false, 1 = true) [tab] Inlet Gas Mole Fraction [tab] Initial Adsorbed Mole Fraction
(NOTE: The above line is repeated for all species in gas phase. Also, this algorithm only allows you to consider one adsorbable gas component. Inlet gas mole fractions must be non-zero for all non-adsorbing gases and must sum to 1.)

Example Scenario Input

1 0
0 0
101.35 0.36
5
0.0
N2 0 0.7825 0.0
O2 0 0.2081 0.0
Ar 0 0.009 0.0
CO2 0 0.0004 0.0
H2O 1 0.0 0.0
Above example is for running optimizations on data collected with a gas stream at 0.36 cm/s with 5 gas species in the mixture, only H2O of which is adsorbing. The "base line" or "inlet gas" without H2O has a composition of N2 at 0.7825, O2 at 0.2081, Ar at 0.009, and CO2 at 0.0004.

Adsorbent Input File

Domain Coord. (2 = spherical, 1 = cylindrical) { [tab] Char. Length (um) (i.e., cylinder length) }
(NOTE: Char. Length is only needed if problem is not spherical)
Pellet Radius (um)

Example Adsorbent Input

1 6.0
2.0
Above example is for a cylindrical adsorbent with a length of 5 um and radius of 2 um.

Component Input File

Molar Weight of ith species (g/mol) [tab] Specific Heat of ith species (J/g/K)
Sutherland Viscosity (g/cm/s) [tab] Sutherland Temperature (K) [tab] Sutherland Constant (K) of ith species
(repeat above for all species in same order they appeared in the Scenario Input File)

Example Component Input

28.016 1.04
0.0001781 300.55 111.0
32.0 0.919
0.0002018 292.25 127.0
39.948 0.522
0.0002125 273.11 144.4
44.009 0.846
0.000148 293.15 240.0
18.0 1.97
0.0001043 298.16 784.72
Above example is exactly the same as in the SCOPSOWL_SCENARIO example (see scopsowl.h). There is no difference in the input file formats for this input. Keep in mind that the order is VERY important! All species information must be in the same order that the species appeared in the Scenario input file.

Adsorbate Input File

Reference Diffusivity (um^2/hr) [tab] Activation Energy (J/mol) of ith adsorbable species
Reference Temperature (K) [tab] Affinity Constant (-) of ith adsorbable species
van der Waals Volume (cm^3/mol) of ith species
GSTA adsorption capacity (mol/kg) of ith species
Number of GSTA parameters of ith species
Enthalpy (J/mol) of nth site [tab] Entropy of nth site (J/K/mol) of ith species
(repeat enthalpy and entropy for all n sites in species i)
(repeat above for all species i)

Example Adsorbate Input

0 0
0 0
13.91
11.67
4
-46597.5 -53.6994
-125024 -221.073
-193619 -356.728
-272228 -567.459
Above example gives the equilibrium parameters associated with the H2O-MS3A single component adsorption system. Note that the kinetic parameters (Ref. Diff., Act. Energy, Ref. Temp., and Affinity) were all given a value of zero. These values are irrelavent if we are running an optimization because they will be replaced with a single estimate for the diffusivity that is being optimization for. However, if we wanted to run this routine with comparisons and not do any optmization, then you would need to provide non-zero values for these parameters (at least for Ref. Diff.).

Data Input File

Number of Kinetic Data Curves
Number of data points in the ith curve
Temperature (K) [tab] Partial Pressure (kPa) [tab] Equilibrium Adsorption (mol/kg) all of ith curve
Time point 1 (hrs) [tab] Adsorption 1 (mol/kg) of ith curve
Time point 1 (hrs) [tab] Adsorption 2 (mol/kg) of ith curve
... (Repeat for all time-adsorption data points)
(Repeat above for all curves i)

Example Data Input

40
2990
298.15 0.000310922 2.9
0 0
0.166666667 0.001834419
0.333611111 0.004880247
0.5 0.008306803
...
2789
298.15 0.00055189 5
0 0
0.166944444 0.003350185
0.333611111 0.007418267
0.5 0.009930906
0.666666667 0.014597236
0.833611111 0.021377373
....
Above is a partial example for a data set of 40 kinetic curves. The first curve contains 2990 data points and has temperature of 298.15 K, partial pressure of 0.000310922 kPa, and an equilibrium adsorption of 2.9. Each first time point should start from 0 hours and each initial adsorption should correspond to the value of initial adsorption indicated in the Scenario input file. Then, this structure is repeated for all adsorptio curves.

6.28 Trajectory.h File Reference

Single Particle Trajectory Analysis for Magnetic Filtration.

```
#include "macaw.h"
#include <random>
#include <chrono>
```

Classes

- struct TRAJECTORY_DATA

Functions

- double Magnetic_R (const Matrix< double > &dX, const Matrix< double > &dY, int i, double b, double mu_0, double chi_p, double M, double H0, double a)
- double Magnetic_T (const Matrix< double > &dX, const Matrix< double > &dY, int i, double b, double mu_0, double chi_p, double M, double H0, double a)
- double Grav_R (const Matrix< double > &dX, int i, double b, double rho_p, double rho_f)
- double Grav_T (const Matrix< double > &dX, int i, double b, double rho_p, double rho_f)
- double Van_R (const Matrix< double > &dX, const Matrix< double > &dY, int i, double Hamaker, double b, double a)
- double V_RAD (const Matrix< double > &dX, const Matrix< double > &dY, int i, double V0, double rho_f, double a, double eta)
- double V_THETA (const Matrix< double > &dX, const Matrix< double > &dY, int i, double V0, double rho_f, double a, double eta)
- double Brown_RAD (double n_rand, double m_rand, double sigma_n, double sigma_m)

- double Brown_THETA (double s_rand, double t_rand, double sigma_n, double sigma_m)
- int POLAR (Matrix< double > &POL, const Matrix< double > &dX, const Matrix< double > &dY, const void ∗data, int i)
- double In_PVel_Rad (const Matrix< double > &POL)
- double In_PVel_Theta (const Matrix< double > &POL)
- int In_P_Velocity (const Matrix< double > &POL, Matrix< double > &Vr, Matrix< double > &Vt)
- double PVel_Rad (const Matrix< double > &POL, const Matrix< double > &Vr, int i, double mp, double beta, double t, double sigma_v, double rand_n)
- double PVel_Theta (const Matrix< double > &POL, const Matrix< double > &Vt, int i, double mp, double beta, double t, double sigma_v, double rand_s)
- int P_Velocity (const Matrix< double > &POL, Matrix< double > &Vr, Matrix< double > &Vt, int i, const void ∗data)
- double RADIAL_FORCE (const Matrix< double > &POL, const Matrix< double > &Vr, int i, double beta, double mp, double dt, double a)
- double TANGENTIAL_FORCE (const Matrix< double > &POL, const Matrix< double > &Vt, const Matrix< double > &dY, int i, double beta, double mp, double dt, double a)
- double Capture_Force (const Matrix< double > &POL, const Matrix< double > &Vr, int i, double beta, double mp, double dt, double a)
- int CARTESIAN (const Matrix< double > &POL, const Matrix< double > &Vr, const Matrix< double > &Vt, Matrix< double > &H, const Matrix< double > &dY, int i, const void ∗data)
- int DISPLACEMENT (Matrix< double > &dX, Matrix< double > &dY, const Matrix< double > &H, int i)
- int LOCATION (const Matrix< double > &dY, const Matrix< double > &dX, Matrix< double > &X, Matrix< double > &Y, int i)
- double Removal_Efficiency (double Sum_Cap, const void ∗data)
- int Trajectory_SetupConstants (TRAJECTORY_DATA ∗dat)
- int Number_Generator (TRAJECTORY_DATA ∗dat)
- int Run_Trajectory ()

  Run_Trajectory function.

6.28.1 Detailed Description

Single Particle Trajectory Analysis for Magnetic Filtration.

Trajectory.cpp

Alex, Please provide details here... and elsewhere in the file.

Author

Alex Wiechert

Date

08/25/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Alex Wiechert for PhD research in the area of environmental surface science. Copyright (c) 2015, all rights reserved.

6.28.2 Function Documentation

6.28.2.1 Magnetic_R()

```
double Magnetic_R (
            const Matrix< double > & dX,
            const Matrix< double > & dY,
            int i,
            double b,
            double mu_0,
            double chi_p,
            double M,
            double H0,
            double a )
```

6.28.2.2 Magnetic_T()

```
double Magnetic_T (
            const Matrix< double > & dX,
            const Matrix< double > & dY,
            int i,
            double b,
            double mu_0,
            double chi_p,
            double M,
            double H0,
            double a )
```

6.28.2.3 Grav_R()

```
double Grav_R (
            const Matrix< double > & dX,
            int i,
            double b,
            double rho_p,
            double rho_f )
```

6.28.2.4 Grav_T()

```
double Grav_T (
            const Matrix< double > & dX,
            int i,
            double b,
            double rho_p,
            double rho_f )
```

6.28.2.5 Van_R()

```
double Van_R (
            const Matrix< double > & dX,
            const Matrix< double > & dY,
            int i,
            double Hamaker,
            double b,
            double a )
```

6.28.2.6 V_RAD()

```
double V_RAD (
            const Matrix< double > & dX,
            const Matrix< double > & dY,
            int i,
            double V0,
            double rho_f,
            double a,
            double eta )
```

6.28.2.7 V_THETA()

```
double V_THETA (
            const Matrix< double > & dX,
            const Matrix< double > & dY,
            int i,
            double V0,
            double rho_f,
            double a,
            double eta )
```

6.28.2.8 Brown_RAD()

```
double Brown_RAD (
            double n_rand,
            double m_rand,
            double sigma_n,
            double sigma_m )
```

6.28.2.9 Brown_THETA()

```
double Brown_THETA (
            double s_rand,
            double t_rand,
            double sigma_n,
            double sigma_m )
```

6.28.2.10 POLAR()

```
int POLAR (
            Matrix< double > & POL,
            const Matrix< double > & dX,
            const Matrix< double > & dY,
            const void * data,
            int i )
```

6.28.2.11 In_PVel_Rad()

```
double In_PVel_Rad (
            const Matrix< double > & POL )
```

6.28.2.12 In_PVel_Theta()

```
double In_PVel_Theta (
            const Matrix< double > & POL )
```

6.28.2.13 In_P_Velocity()

```
int In_P_Velocity (
            const Matrix< double > & POL,
            Matrix< double > & Vr,
            Matrix< double > & Vt )
```

6.28.2.14 PVel_Rad()

```
double PVel_Rad (
            const Matrix< double > & POL,
            const Matrix< double > & Vr,
            int i,
            double mp,
            double beta,
            double t,
            double sigma_v,
            double rand_n )
```

6.28.2.15 PVel_Theta()

```
double PVel_Theta (
            const Matrix< double > & POL,
            const Matrix< double > & Vt,
            int i,
            double mp,
            double beta,
            double t,
            double sigma_v,
            double rand_s )
```

6.28.2.16 P_Velocity()

```
int P_Velocity (
            const Matrix< double > & POL,
            Matrix< double > & Vr,
            Matrix< double > & Vt,
            int i,
            const void * data )
```

6.28.2.17 RADIAL_FORCE()

```
double RADIAL_FORCE (
            const Matrix< double > & POL,
            const Matrix< double > & Vr,
            int i,
            double beta,
            double mp,
            double dt,
            double a )
```

6.28.2.18 TANGENTIAL_FORCE()

```
double TANGENTIAL_FORCE (
            const Matrix< double > & POL,
            const Matrix< double > & Vt,
            const Matrix< double > & dY,
            int i,
            double beta,
            double mp,
            double dt,
            double a )
```

6.28.2.19 Capture_Force()

```
double Capture_Force (
            const Matrix< double > & POL,
            const Matrix< double > & Vr,
            int i,
            double beta,
            double mp,
            double dt,
            double a )
```

6.28.2.20 CARTESIAN()

```
int CARTESIAN (
            const Matrix< double > & POL,
            const Matrix< double > & Vr,
            const Matrix< double > & Vt,
            Matrix< double > & H,
            const Matrix< double > & dY,
            int i,
            const void * data )
```

6.28.2.21 DISPLACEMENT()

```
int DISPLACEMENT (
            Matrix< double > & dX,
            Matrix< double > & dY,
            const Matrix< double > & H,
            int i )
```

6.28.2.22 LOCATION()

```
int LOCATION (
            const Matrix< double > & dY,
            const Matrix< double > & dX,
            Matrix< double > & X,
            Matrix< double > & Y,
            int i )
```

6.28.2.23 Removal_Efficiency()

```
double Removal_Efficiency (
            double Sum_Cap,
            const void * data )
```

6.28.2.24 Trajectory_SetupConstants()

```
int Trajectory_SetupConstants (
            TRAJECTORY_DATA * dat )
```

6.28.2.25 Number_Generator()

```
int Number_Generator (
            TRAJECTORY_DATA * dat )
```

6.28.2.26 Run_Trajectory()

```
int Run_Trajectory ( )
```

Run_Trajectory function.

Function to run the Trajectory project.

6.29 ui.h File Reference

User Interface for Ecosystem.

```
#include <fstream>
#include <string>
#include <iostream>
#include "error.h"
#include "yaml_wrapper.h"
#include "flock.h"
#include "school.h"
#include "sandbox.h"
#include "Trajectory.h"
```

Classes

- struct UI_DATA

  Data structure holding the UI arguments.

Macros

- #define UI_HPP_
- #define ECO_VERSION "1.0.0"

  Macro expansion for executable current version number.
- #define ECO_EXECUTABLE "eco"

  Macro expansion for executable current name.

Enumerations

- enum valid_options {
  TEST, EXECUTE, EXIT, CONTINUE,
  HELP, dogfish, eel, egret,
  finch, lark, macaw, mola,
  monkfish, sandbox, scopsowl, shark,
  skua, gsta_opt, magpie, scops_opt,
  skua_opt, trajectory, dove, crow,
  mesh, crane, ibis, fairy,
  kea, cardinal }

  Valid options available upon execution of the code.

Functions

- void aui_help ()

    Function to display help for Advanced User Interface.
- void bui_help ()

    Function to display help for Basic User Interface.
- bool exit (const std::string &input)

    Function returns true if user requests exit.
- bool help (const std::string &input)

    Function returns trun if the user requests help.
- bool version (const std::string &input)

    Function returns true if user requests to know the executable version.
- bool test (const std::string &input)

    Function returns true if user requests to run a test.
- bool exec (const std::string &input)

    Function returns true if the user requests to run a simulation/executable.
- bool path (const std::string &input)

    Function returns true if the user indicates that input files share a common path.
- bool input (const std::string &input)

    Function returns true if the user indicates that the next arguments are input files.
- bool valid_test_string (const std::string &input, UI_DATA ∗ui_dat)

    Function returns true if the user gave a valid test option.
- bool valid_exec_string (const std::string &input, UI_DATA ∗ui_dat)

    Function returns true if the user gave a valid execution option.
- int number_files (UI_DATA ∗ui_dat)

    Function returns the number of expected input files for the user's run option.
- bool valid_addon_options (UI_DATA ∗ui_dat)

    Function returns true if the user has choosen a valid additional runtime option.
- void display_help (UI_DATA ∗ui_dat)

    Function to call the appropriate help menu based on type of interface.
- void display_version (UI_DATA ∗ui_dat)

    Function to display ecosystem version information to the console.
- int invalid_input (int count, int max)

    Function returns a CONTINUE or EXIT when invalid input is given.
- bool valid_input_main (UI_DATA ∗ui_dat)

    Function returns true if user gave valid input in Basic UI.
- bool valid_input_tests (UI_DATA ∗ui_dat)

    Function returns true if user gave a valid test function to run.
- bool valid_input_execute (UI_DATA ∗ui_dat)

    Function returns true if user gave a valid executable function to run.
- int test_loop (UI_DATA ∗ui_dat)

    Function that loops the Basic UI until a valid test option was selected.
- int exec_loop (UI_DATA ∗ui_dat)

    Function that loops the Basic UI until a valid executable option was selected.
- int run_test (UI_DATA ∗ui_dat)

    Function will call the user requested test function.
- int run_exec (UI_DATA ∗ui_dat)

    Function will call the user requested executable function.
- int run_executable (int argc, const char ∗argv[ ])

    Function called by the main and runs both user interfaces for the program.

6.29.1 Detailed Description

User Interface for Ecosystem.

ui.cpp

These routines define how the user will interface with the software

Author

Austin Ladshaw

Date

08/25/2015

Copyright

This software was designed and built at the Georgia Institute of Technology by Austin Ladshaw for PhD research in the area of adsorption and surface science. Copyright (c) 2015, all rights reserved.

6.29.2 Macro Definition Documentation

6.29.2.1 UI_HPP_

```
#define UI_HPP_
```

6.29.2.2 ECO_VERSION

```
#define ECO_VERSION "1.0.0"
```

Macro expansion for executable current version number.

6.29.2.3 ECO_EXECUTABLE

```
#define ECO_EXECUTABLE "eco"
```

Macro expansion for executable current name.

6.29.3 Enumeration Type Documentation

6.29.3.1 valid_options

```
enum valid_options
```

Valid options available upon execution of the code.

Enumeration of valid options for executing the ecosystem code. More options become available as the code updates. Some options that appear here may not be viewable in the "help" screen of the executable. Those options are hidden, but are still valid entries.

Enumerator

TEST	
EXECUTE	
EXIT	
CONTINUE	
HELP	
dogfish	
eel	
egret	
finch	
lark	
macaw	
mola	
monkfish	
sandbox	
scopsowl	
shark	
skua	
gsta_opt	
magpie	
scops_opt	
skua_opt	
trajectory	
dove	
crow	
mesh	
crane	
ibis	
fairy	
kea	
cardinal	

6.29.4 Function Documentation

6.29.4.1 aui_help()

```
void aui_help ( )
```

Function to display help for Advanced User Interface.

The Advanved User Interface help screen is accessed by including run option -h or –help when executing the program from command line.

6.29.4.2 bui_help()

```
void bui_help ( )
```

Function to display help for Basic User Interface.

The Basic User Interface help screen is accessed by running the executable, then typing "help" at any point during the console prompts. Exception to this occurs when the console prompts you to provide input files for your choosen routine. In this circumstance, the executable always assumes that what the user types in will be an input file.

6.29.4.3 exit()

```
bool exit (
            const std::string & input )
```

Function returns true if user requests exit.

This function will check the input string for "exit" or "quit" and terminate the executable. Only checked if using the Basic User Interface.

Parameters

input	input string user gives to the console

6.29.4.4 help()

```
bool help (
            const std::string & input )
```

Function returns trun if the user requests help.

This function will check the input string for "help", "-h", or "--help" and will tell the executable to display the help menu. The help menu that gets displayed depends on how the executable was run to begin with.

Parameters

input	input string user gives to the console

6.29.4.5 version()

```
bool version (
            const std::string & input )
```

Function returns true if user requests to know the executable version.

This function will check the input string for "version", "-v", or "--version" and will tell the executable to display version information about the executable.

Parameters

input	input string user gives to the console

6.29.4.6 test()

```
bool test (
            const std::string & input )
```

Function returns true if user requests to run a test.

This function will check the input string for "-t" or "--test" and determine whether or not the user requests to run an ecosystem test function.

Parameters

input	input string user gives to the console

6.29.4.7 exec()

```
bool exec (
             const std::string & input )
```

Function returns true if the user requests to run a simulation/executable.

This function will check the input string for "-e" or "--execute" and determine whether or not the user requests to run an ecosystem executable function.

Parameters

input	input string the user gives to the console

6.29.4.8 path()

```
bool path (
             const std::string & input )
```

Function returns true if the user indicates that input files share a common path.

This function will check the input string for "-p" or "--path" and determine whether or not the user will give a common path to all input files needed for the specified simulation. Only used in Advanced User Interface.

Parameters

input	input string the user gives to the console

6.29.4.9 input()

```
bool input (
             const std::string & input )
```

Function returns true if the user indicates that the next arguments are input files.

This function will check the input string for "-i" or "--input" and determine whether or not the user's next arguments are input files for a specific simulation. Only used in Advanced User Interface.

Parameters

input	input string the user gives to the console

6.29.4.10 valid_test_string()

```
bool valid_test_string (
            const std::string & input,
            UI_DATA * ui_dat )
```

Function returns true if the user gave a valid test option.

This function will check the input string given by the user and determine whether that string denotes a valid test. Then, it will mark the option variable in ui_dat with the appropriate option from the valid_options enum.

Parameters

input	input string the user gives to the console
ui_dat	pointer to the data structure for the ui object

6.29.4.11 valid_exec_string()

```
bool valid_exec_string (
            const std::string & input,
            UI_DATA * ui_dat )
```

Function returns true if the user gave a valid execution option.

This function will check the input string given by the user and determine whether that string denotes a valid execution option. Then, it will mark the option variable in ui_dat with the appropriate option from the valid_options enum.

Parameters

input	input string the user gives to the console
ui_dat	pointer to the data structure for the ui object

6.29.4.12 number_files()

```
int number_files (
            UI_DATA * ui_dat )
```

Function returns the number of expected input files for the user's run option.

This function will check the option variable in the ui_dat structure to determine the number of input files that is expected to be given. Running different executable functions in ecosystem may require various number of input files.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.13 valid_addon_options()

```
bool valid_addon_options (
            UI_DATA * ui_dat )
```

Function returns true if the user has choosen a valid additional runtime option.

This function will check all additional input options in the user_input variable of ui_dat to determine if the user requests any additional options during runtime. Valid additional options are -p or –path and -i or –input.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.14 display_help()

```
void display_help (
            UI_DATA * ui_dat )
```

Function to call the appropriate help menu based on type of interface.

This function looks at the ui_dat structure and the user's OS files to determine what help menu to display and how to display it. There are two different types of help menus that can be displayed: (i) Advanced Help and (ii) Basic Help. Additionally, this function checks the OS file system for the existence of installed help files. If it finds those files, then it instructs the command terminal to read the contents of those files with the "less" command. Otherwise, it will just print the appropriate help menu to the console window.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.15 display_version()

```
void display_version (
            UI_DATA * ui_dat )
```

Function to display ecosystem version information to the console.

This function will check the ui_dat structure to see which type of interface the user is using, then print out the version information for the executable being run.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.16 invalid_input()

```
int invalid_input (
            int count,
            int max )
```

Function returns a CONTINUE or EXIT when invalid input is given.

This function looks at the current count and the max iterations and determines whether or not to force the executable to terminate. If the user provides too many incorrect options during the Basic User Interface, then the executable will force quit.

Parameters

count	number of times the user has provided a bad option
max	maximum allowable bad options before force quit

6.29.4.17 valid_input_main()

```
bool valid_input_main (
            UI_DATA * ui_dat )
```

Function returns true if user gave valid input in Basic UI.

This function is only called if the user is running the Basic UI. It checks the given console argument stored in user_input of ui_dat for a valid option. If no valid option is given, then this function returns false.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.18 valid_input_tests()

```
bool valid_input_tests (
            UI_DATA * ui_dat )
```

Function returns true if user gave a valid test function to run.

This function checks the user_input argument of ui_dat for a valid test option. If no valid test was given, then this function returns false.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.19 valid_input_execute()

```
bool valid_input_execute (
            UI_DATA * ui_dat )
```

Function returns true if user gave a valid executable function to run.

This function checks the user_input argument of ui_dat for a valid executable option. If no valid executable was given, then this function returns false.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.20 test_loop()

```
int test_loop (
            UI_DATA * ui_dat )
```

Function that loops the Basic UI until a valid test option was selected.

This function loops the Basic UI menu for running a test until a valid test is selected by the user. If a valid test is not selected, and the maximum number of loops has been reached, then this function will cause the program to force quit.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.21 exec_loop()

```
int exec_loop (
            UI_DATA * ui_dat )
```

Function that loops the Basic UI until a valid executable option was selected.

This function loops the Basic UI menu for running an executable until a valid executable is selected by the user. If a valid executable is not selected, and the maximum number of loops has been reached, then this function will cause the program to force quit.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.22 run_test()

```
int run_test (
            UI_DATA * ui_dat )
```

Function will call the user requested test function.

This function checks the option variable of the ui_dat structure and runs the corresponding test function.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.23 run_exec()

```
int run_exec (
            UI_DATA * ui_dat )
```

Function will call the user requested executable function.

This function checks the option variable of the ui_dat structure and runs the corresponding executable function.

Parameters

ui_dat	pointer to the data structure for the ui object

6.29.4.24 run_executable()

```
int run_executable (
            int argc,
            const char * argv[] )
```

Function called by the main and runs both user interfaces for the program.

This function is called in the main.cpp file and passes the console arguments given at run time.

Parameters

argc	number of arguments provided by the user at the time of execution
argv	list of C-strings that was provided by the user at the time of execution

6.30 yaml_wrapper.h File Reference

C++ Wrapper for the C-YAML Library.

```
#include "yaml.h"
#include "error.h"
#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <stdexcept>
```

Classes

- class ValueTypePair

  Value-Type Pair object to recognize data type of a string that was read.
- class KeyValueMap

  Key-Value-Type Map object creating a map of the KeyValuePair objects.
- class SubHeader

  Object for the Lowest level of Header for the yaml_wrapper.
- class Header

  Object for headers in a yaml document (inherits from SubHeader)
- class Document

  Object for the various documents in the yaml file.
- class YamlWrapper

  Object for the entire yaml file holding all documents, header, sub-headers, keys, and values.
- class yaml_cpp_class

  Primary object used when reading and digitally storing yaml files.

Typedefs

- typedef enum data_type data_type

  Enum for valid data types in ValueTypePair.
- typedef enum header_state header_state

  Enum for state of the headers in the yaml_wrapper.

Enumerations

- enum data_type {
  STRING, BOOLEAN, DOUBLE, INT,
  UNKNOWN }
- enum header_state { ANCHOR, ALIAS, NONE }

Functions

- yaml_cpp_class ∗ New_YAML ()

    Set of C-style functions to be used from python 3.5 (or higher)
- void YAML_DeleteContents (yaml_cpp_class ∗obj)

    Delete the contents of the obj.
- int YAML_executeYamlRead (yaml_cpp_class ∗obj, const char ∗file)

    Use yaml object to read a file.
- void YAML_DisplayContents (yaml_cpp_class ∗obj)

    Display the contents of the yaml object.
- int YAML_DocumentKeys_Size (yaml_cpp_class ∗obj)

    Return buffer size for all Document Keys.
- void YAML_DocumentKeys (yaml_cpp_class ∗obj, char ∗keys)

    Modify char∗ to gain the list of keys.
- int YAML_HeaderKeys_Size (yaml_cpp_class ∗obj, const char ∗doc)

    Return buffer size for all Header keys in a doc.
- void YAML_HeaderKeys (yaml_cpp_class ∗obj, const char ∗doc, char ∗keys)
- int YAML_SubHeaderKeys_Size (yaml_cpp_class ∗obj, const char ∗doc, const char ∗head)

    Return buffer size for all Header keys in a doc.
- void YAML_SubHeaderKeys (yaml_cpp_class ∗obj, const char ∗doc, const char ∗head, char ∗keys)

    Modify char∗ to gain list of all keys in doc.
- int YAML_DocumentData_Size (yaml_cpp_class ∗obj, const char ∗doc)

    Return buffer size for all key-value pairs in the document map.
- void YAML_DocumentData (yaml_cpp_class ∗obj, const char ∗doc, char ∗key_values)

    Modify char∗ to gain list of all key-value pairs in document separated by ∗ symbols.
- int YAML_HeaderData_Size (yaml_cpp_class ∗obj, const char ∗doc, const char ∗head)

    Return buffer size for all key-value pairs in the header map.
- void YAML_HeaderData (yaml_cpp_class ∗obj, const char ∗doc, const char ∗head, char ∗key_values)

    Modify char∗ to gain list of all key-value pairs in header separated by ∗ symbols.
- int YAML_SubHeaderData_Size (yaml_cpp_class ∗obj, const char ∗doc, const char ∗head, const char ∗subhead)

    Return buffer size for all key-value pairs in the subheader map.
- void YAML_SubHeaderData (yaml_cpp_class ∗obj, const char ∗doc, const char ∗head, const char ∗subhead, char ∗key_values)

    Modify char∗ to gain list of all key-value pairs in subheader separated by ∗ symbols.
- std::string allLower (const std::string &input)

    Function to return an all lower case string based on the passed argument.
- bool isEven (int n)

    Function to return true if the given argument is an even number.
- int YAML_WRAPPER_TESTS ()

    Function to run tests on all the objects that yaml_cpp_class.
- int YAML_CPP_TEST (const char ∗file)

    Function to run a test read for the yaml_cpp_class on a given file.

6.30.1 Detailed Description

C++ Wrapper for the C-YAML Library.

yaml_wrapper.cpp

This file holds objects, structures, and functions associatied with using the C-YAML library. A C++ wrapper has been created for the Kirill Simonov (2006) LibYAML library to more easily store and query information in yaml style input files. The wrapper uses the C-YAML parser to identify the file structure and store the read in information from that document into an object using C++ maps. Those maps are hold information in a series of Key-Value pairs as well as lists of Key-Value pairs. This allows the user to create well organized input files to change the behavior of simulations.

The yaml_wrapper is restricted to the same limitations in the C-YAML source code in terms of how the documents are allowed to be structured for TOKEN based parsing. C-YAML only recognizes specific tokens and will only allow a certain level of Sub-Header mapping. Therefore, this wrapper has the same limitations. Below is an example of acceptable formatting for a C-YAML document.

#Test input file for YAML and SHARK
TestDoc1: &hat
—

- scenario:
  numvar: 25
  act_fun: DAVIES
  steadystate: FALSE
  t_out: 1
  pH: 0

- testblock:
  another: block

  - subblock:
    sub: block
    ...
    TestDoc2: ∗hat
    —

- masterspecies:
  "Cl - (aq)": 0
  "Na + (aq)": 1
  "H2O (l)": 2
  3: NaCl (aq)
  ...
  TestDoc3:
  —
  apple: red
  pear: green

- array: #Block
  banana: yellow
  #List 1 in array

  - list1: &a #also a block
    a: 1 #key : value
    b: 2
    c: 3
    #List 2 in array
  - list2: ∗a
    a: 4
    b: 5
    c: 6
    ...
    TestDoc4:
    —

- anchor: &anchor
  stuff: to do

- alias: ∗anchor
  add: to stuff

- list:

  - anchored: &list_anchor
    info: blah
    atta: boy

  - aliased: ∗list_anchor
    info: bruh
    atta: ber
    ...
    #WARNING: MAKE SURE FILE DOES NOT CONTAIN TABS!!!

TestDoc5:
—

- grab: ∗anchor
  add2: more adds

  - listcopy: ∗a

- block: {1: 2, 3: 4}
  still: in block
  ...

  Note

  You can view the actual yaml example file in the input_files/SHARK/test_input.yml sub-directory of the project folder.

  Author

  Austin Ladshaw

Date

07/29/2015

Copyright

6.30.2 Typedef Documentation

6.30.2.1 data_type

```
typedef enum data_type data_type
```

Enum for valid data types in ValueTypePair.

6.30.2.2 header_state

```
typedef enum header_state header_state
```

Enum for state of the headers in the yaml_wrapper.

6.30.3 Enumeration Type Documentation

6.30.3.1 data_type

```
enum data_type
```

Enumerator

STRING	
BOOLEAN	
DOUBLE	
INT	
UNKNOWN	

6.30.3.2 header_state

```
enum header_state
```

Enumerator

ANCHOR	
ALIAS	
NONE	

6.30.4 Function Documentation

6.30.4.1 New_YAML()

```
yaml_cpp_class* New_YAML ( )
```

Set of C-style functions to be used from python 3.5 (or higher)

Create instance of yaml C++ object

6.30.4.2 YAML_DeleteContents()

```
void YAML_DeleteContents (
             yaml_cpp_class * obj )
```

Delete the contents of the obj.

6.30.4.3 YAML_executeYamlRead()

```
int YAML_executeYamlRead (
             yaml_cpp_class * obj,
             const char * file )
```

Use yaml object to read a file.

6.30.4.4 YAML_DisplayContents()

```
void YAML_DisplayContents (
            yaml_cpp_class * obj )
```

Display the contents of the yaml object.

6.30.4.5 YAML_DocumentKeys_Size()

```
int YAML_DocumentKeys_Size (
            yaml_cpp_class * obj )
```

Return buffer size for all Document Keys.

6.30.4.6 YAML_DocumentKeys()

```
void YAML_DocumentKeys (
            yaml_cpp_class * obj,
            char * keys )
```

Modify char* to gain the list of keys.

6.30.4.7 YAML_HeaderKeys_Size()

```
int YAML_HeaderKeys_Size (
            yaml_cpp_class * obj,
            const char * doc )
```

Return buffer size for all Header keys in a doc.

6.30.4.8 YAML_HeaderKeys()

```
void YAML_HeaderKeys (
            yaml_cpp_class * obj,
            const char * doc,
            char * keys )
```

Modify char* to gain list of all keys in doc

6.30.4.9 YAML_SubHeaderKeys_Size()

```
int YAML_SubHeaderKeys_Size (
            yaml_cpp_class * obj,
            const char * doc,
            const char * head )
```

Return buffer size for all Header keys in a doc.

6.30.4.10 YAML_SubHeaderKeys()

```
void YAML_SubHeaderKeys (
             yaml_cpp_class * obj,
             const char * doc,
             const char * head,
             char * keys )
```

Modify char∗ to gain list of all keys in doc.

6.30.4.11 YAML_DocumentData_Size()

```
int YAML_DocumentData_Size (
             yaml_cpp_class * obj,
             const char * doc )
```

Return buffer size for all key-value pairs in the document map.

6.30.4.12 YAML_DocumentData()

```
void YAML_DocumentData (
             yaml_cpp_class * obj,
             const char * doc,
             char * key_values )
```

Modify char∗ to gain list of all key-value pairs in document separated by ∗ symbols.

6.30.4.13 YAML_HeaderData_Size()

```
int YAML_HeaderData_Size (
             yaml_cpp_class * obj,
             const char * doc,
             const char * head )
```

Return buffer size for all key-value pairs in the header map.

6.30.4.14 YAML_HeaderData()

```
void YAML_HeaderData (
             yaml_cpp_class * obj,
             const char * doc,
             const char * head,
             char * key_values )
```

Modify char∗ to gain list of all key-value pairs in header separated by ∗ symbols.

6.30.4.15 YAML_SubHeaderData_Size()

```
int YAML_SubHeaderData_Size (
             yaml_cpp_class * obj,
             const char * doc,
             const char * head,
             const char * subhead )
```

Return buffer size for all key-value pairs in the subheader map.

6.30.4.16 YAML_SubHeaderData()

```
void YAML_SubHeaderData (
             yaml_cpp_class * obj,
             const char * doc,
             const char * head,
             const char * subhead,
             char * key_values )
```

Modify char∗ to gain list of all key-value pairs in subheader separated by ∗ symbols.

6.30.4.17 allLower()

```
std::string allLower (
             const std::string & input )
```

Function to return an all lower case string based on the passed argument.

This function will copy the input paramter and convert that copy to all lower case. The copy is then returned and can be checked against valid or allowed strings.

Parameters

input	string to copy and convert to lower case

6.30.4.18 isEven()

```
bool isEven (
             int n )
```

Function to return true if the given argument is an even number.

6.30.4.19 YAML_WRAPPER_TESTS()

```
int YAML_WRAPPER_TESTS ( )
```

Function to run tests on all the objects that yaml_cpp_class.

This test function is currently NOT callable from the UI.

6.30.4.20 YAML_CPP_TEST()

```
int YAML_CPP_TEST (
            const char * file )
```

Function to run a test read for the yaml_cpp_class on a given file.

This test/executable function is currently NOT callable from the UI.

Index