# Ecosystem

## Version 0.0 beta

Generated by Doxygen 1.8.3.1

# Contents

# Chapter 1

# Hierarchical Index

## 1.1 Class Hierarchy

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

# Chapter 2

# Class Index

## 2.1 Class List

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

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all files with brief descriptions:

# Chapter 4

# Class Documentation

## 4.1 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*

### 4.1.1 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.

### 4.1.2 Member Data Documentation

#### 4.1.2.1 int ARNOLDI_DATA::k

Desired size of the Krylov subspace.

#### 4.1.2.2 int ARNOLDI_DATA::iter

Actual size of the Krylov subspace.

#### 4.1.2.3 double ARNOLDI_DATA::beta

Normalization parameter.

#### 4.1.2.4 double ARNOLDI_DATA::hp1

Additional row element of H (separate storage for holding)

#### 4.1.2.5 bool ARNOLDI_DATA::Output = true

True = print messages to console.

#### 4.1.2.6 std::vector< Matrix<double> > ARNOLDI_DATA::Vk

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

#### 4.1.2.7 Matrix<double> ARNOLDI_DATA::Hkp1

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

#### 4.1.2.8 Matrix<double> ARNOLDI_DATA::yk

(k) x (1) vector search direction

#### 4.1.2.9 Matrix<double> ARNOLDI_DATA::e1

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

#### 4.1.2.10 Matrix<double> ARNOLDI_DATA::w

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

**4.1.2.11  Matrix$<$double$>$ ARNOLDI_DATA::v**

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

**4.1.2.12  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

## 4.2  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>
```

**Public Member Functions**

- Atom ()

  *Default Constructor.*
- ∼Atom ()

  *Default Destructor.*
- Atom (std::string Name)

  *Constructor by Atom Name.*
- Atom (int number)

  *Constructor by Atomic number.*
- 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 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.*

- 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.*

## Private Attributes

- 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.*

### 4.2.1 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 Ununoctium (118).

### 4.2.2 Constructor & Destructor Documentation

#### 4.2.2.1 Atom::Atom ( )

Default Constructor.

#### 4.2.2.2 Atom::∼Atom ( )

Default Destructor.

#### 4.2.2.3 Atom::Atom ( std::string *Name* )

Constructor by Atom Name.

#### 4.2.2.4 Atom::Atom ( int *number* )

Constructor by Atomic number.

### 4.2.3 Member Function Documentation

#### 4.2.3.1 void Atom::Register ( std::string *Symbol* )

Register an atom object by symbol.

#### 4.2.3.2 void Atom::Register ( int *number* )

Register an atom object by number.

#### 4.2.3.3 void Atom::editAtomicWeight ( double *AW* )

Manually changes the atomic weight.

#### 4.2.3.4 void Atom::editOxidationState ( int *state* )

Manually changes the oxidation state.

#### 4.2.3.5 void Atom::editProtons ( int *proton* )

Manually changes the number of protons.

#### 4.2.3.6 void Atom::editNeutrons ( int *neutron* )

Manually changes the number of neutrons.

**4.2.3.7    void Atom::editElectrons (  int *electron*  )**

Manually changes the number of electrons.

**4.2.3.8    void Atom::editValence (  int *val*  )**

Manually changes the number of valence electrons.

**4.2.3.9    void Atom::removeProton (    )**

Manually removes 1 proton and adjusts weight.

**4.2.3.10    void Atom::removeNeutron (    )**

Manually removes 1 neutron and adjusts weight.

**4.2.3.11    void Atom::removeElectron (    )**

Manually removes 1 electron from valence.

**4.2.3.12    double Atom::AtomicWeight (    )**

Returns the current atomic weight (g/mol)

**4.2.3.13    int Atom::OxidationState (    )**

Returns the current oxidation state.

**4.2.3.14    int Atom::Protons (    )**

Returns the current number of protons.

**4.2.3.15    int Atom::Neutrons (    )**

Returns the current number of neutrons.

**4.2.3.16    int Atom::Electrons (    )**

Returns the current number of electrons.

**4.2.3.17    int Atom::BondingElectrons (    )**

Returns the number of electrons available for bonding.

**4.2.3.18    std::string Atom::AtomName (    )**

Returns the name of the atom.

**4.2.3.19    std::string Atom::AtomSymbol (   )**

Returns the symbol of the atom.

**4.2.3.20    std::string Atom::AtomCategory (   )**

Returns the category of the atom.

**4.2.3.21    std::string Atom::AtomState (   )**

Returns the state of the atom.

**4.2.3.22    int Atom::AtomicNumber (   )**

Returns the atomic number of the atom.

**4.2.3.23    void Atom::DisplayInfo (   )**

Displays Atom information to console.

### 4.2.4    Member Data Documentation

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

Holds the atomic weight of the atom.

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

Holds the oxidation state of the atom.

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

Holds the number of protons in the atom.

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

Holds the number of neutrons in the atom.

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

Holds the number of electrons in the atom.

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

Holds the number of valence electrons in the atom.

**4.2.4.7    std::string Atom::Name** `[private]`

Holds the name of the atom.

**4.2.4.8   std::string Atom::Symbol**  `[private]`

Holds the atomic symbol for the atom.

**4.2.4.9   std::string Atom::Category**  `[private]`

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

**4.2.4.10   std::string Atom::NaturalState**  `[private]`

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

**4.2.4.11   int Atom::atomic_number**  `[private]`

Holds the atomic number of the atom.

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

- eel.h

## 4.3   BACKTRACK_DATA Struct Reference

Data structure for the implementation of Backtracking Linesearch.

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

**Public Attributes**

- 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.*

### 4.3.1   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.

### 4.3.2 Member Data Documentation

#### 4.3.2.1 double BACKTRACK_DATA::alpha = 1e-4

Scaling parameter for determination of search step size.

#### 4.3.2.2 double BACKTRACK_DATA::rho = 0.1

Scaling parameter for to change step size by.

#### 4.3.2.3 double BACKTRACK_DATA::lambdaMin =DBL_EPSILON

Smallest allowable step length.

#### 4.3.2.4 double BACKTRACK_DATA::normFkp1

New residual norm of the Newton step.

#### 4.3.2.5 bool BACKTRACK_DATA::constRho = false

True = use a constant value for rho.

#### 4.3.2.6 Matrix<double> BACKTRACK_DATA::Fk

Old residual vector of the Newton step.

#### 4.3.2.7 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

## 4.4 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.*

### 4.4.1   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.

### 4.4.2 Member Data Documentation

#### 4.4.2.1 int BiCGSTAB_DATA::maxit = 0

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

#### 4.4.2.2 int BiCGSTAB_DATA::iter = 0

Actual number of iterations.

#### 4.4.2.3 bool BiCGSTAB_DATA::breakdown

Boolean to determine if the method broke down.

#### 4.4.2.4 double BiCGSTAB_DATA::alpha

Step size parameter for next solution.

#### 4.4.2.5 double BiCGSTAB_DATA::beta

Step size parameter for search direction.

#### 4.4.2.6 double BiCGSTAB_DATA::rho

Scaling parameter for alpha and beta.

#### 4.4.2.7 double BiCGSTAB_DATA::rho_old

Previous scaling parameter for alpha and beta.

#### 4.4.2.8 double BiCGSTAB_DATA::omega

Scaling parameter and additional step length.

#### 4.4.2.9 double BiCGSTAB_DATA::omega_old

Previous scaling parameter and step length.

#### 4.4.2.10 double BiCGSTAB_DATA::tol_rel = 1e-6

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

#### 4.4.2.11 double BiCGSTAB_DATA::tol_abs = 1e-6

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

#### 4.4.2.12 double BiCGSTAB_DATA::res

Absolute residual norm.

**4.4.2.13   double BiCGSTAB_DATA::relres**

Relative residual norm.

**4.4.2.14   double BiCGSTAB_DATA::relres_base**

Initial residual norm.

**4.4.2.15   double BiCGSTAB_DATA::bestres**

Best found residual norm.

**4.4.2.16   bool BiCGSTAB_DATA::Output = true**

True = print messages to console.

**4.4.2.17   Matrix$<$double$>$ BiCGSTAB_DATA::x**

Current solution to the linear system.

**4.4.2.18   Matrix$<$double$>$ BiCGSTAB_DATA::bestx**

Best found solution to the linear system.

**4.4.2.19   Matrix$<$double$>$ BiCGSTAB_DATA::r**

Residual vector for the linear system.

**4.4.2.20   Matrix$<$double$>$ BiCGSTAB_DATA::r0**

Initial residual vector.

**4.4.2.21   Matrix$<$double$>$ BiCGSTAB_DATA::v**

Search direction for p.

**4.4.2.22   Matrix$<$double$>$ BiCGSTAB_DATA::p**

Search direction for updating.

**4.4.2.23   Matrix$<$double$>$ BiCGSTAB_DATA::y**

Preconditioned search direction.

**4.4.2.24   Matrix$<$double$>$ BiCGSTAB_DATA::s**

Residual updating vector.

**4.4.2.25  Matrix$<$double$>$ BiCGSTAB_DATA::z**

Preconditioned residual updating vector.

**4.4.2.26  Matrix$<$double$>$ BiCGSTAB_DATA::t**

Search direction for resdidual updates.

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

- lark.h

## 4.5  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.*

### 4.5.1 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.

### 4.5.2 Member Data Documentation

#### 4.5.2.1 int CGS_DATA::maxit = 0

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

#### 4.5.2.2 int CGS_DATA::iter = 0

Actual number of iterations.

#### 4.5.2.3 bool CGS_DATA::breakdown

Boolean to determine if the method broke down.

#### 4.5.2.4 double CGS_DATA::alpha

Step size parameter for next solution.

#### 4.5.2.5 double CGS_DATA::beta

Step size parameter for search direction.

#### 4.5.2.6 double CGS_DATA::rho

Scaling parameter for alpha and beta.

**4.5.2.7    double CGS_DATA::sigma**

Scaling parameter and additional step length.

**4.5.2.8    double CGS_DATA::tol_rel = 1e-6**

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

**4.5.2.9    double CGS_DATA::tol_abs = 1e-6**

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

**4.5.2.10    double CGS_DATA::res**

Absolute residual norm.

**4.5.2.11    double CGS_DATA::relres**

Relative residual norm.

**4.5.2.12    double CGS_DATA::relres_base**

Initial residual norm.

**4.5.2.13    double CGS_DATA::bestres**

Best found residual norm.

**4.5.2.14    bool CGS_DATA::Output = true**

True = print messages to console.

**4.5.2.15    Matrix<double> CGS_DATA::x**

Current solution to the linear system.

**4.5.2.16    Matrix<double> CGS_DATA::bestx**

Best found solution to the linear system.

**4.5.2.17    Matrix<double> CGS_DATA::r**

Residual vector for the linear system.

**4.5.2.18    Matrix<double> CGS_DATA::r0**

Initial residual vector.

**4.5.2.19 Matrix$<$double$>$ CGS_DATA::u**

Search direction for v.

**4.5.2.20 Matrix$<$double$>$ CGS_DATA::w**

Updates sigma and u.

**4.5.2.21 Matrix$<$double$>$ CGS_DATA::v**

Search direction for x.

**4.5.2.22 Matrix$<$double$>$ CGS_DATA::p**

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

**4.5.2.23 Matrix$<$double$>$ CGS_DATA::c**

Holds the matvec result between A and p.

**4.5.2.24 Matrix$<$double$>$ CGS_DATA::z**

Full search direction for x.

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

- lark.h

## 4.6 Document Class Reference

```
#include <yaml_wrapper.h>
```

Inheritance diagram for Document:

```
SubHeader
    ▲
    ┊
Document
```
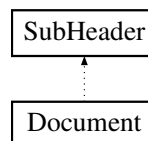
**Public Member Functions**

- Document ()
- ∼Document ()
- Document (const Document &doc)
- Document (std::string name)
- Document (const KeyValueMap &map)
- Document (std::string name, const KeyValueMap &map)
- Document (std::string key, const Header &head)
- Document & operator= (const Document &doc)
- ValueTypePair & operator[] (const std::string key)

- ValueTypePair operator[] (const std::string key) const
- Header & operator() (const std::string key)
- Header operator() (const std::string key) const
- std::map< std::string, Header > & getHeadMap ()
- KeyValueMap & getDataMap ()
- Header & getHeader (std::string key)
- std::map< std::string, Header >
  ::const_iterator end () const
- std::map< std::string, Header >
  ::iterator end ()
- std::map< std::string, Header >
  ::const_iterator begin () const
- std::map< std::string, Header >
  ::iterator begin ()
- void clear ()
- void resetKeys ()
- void changeKey (std::string oldKey, std::string newKey)
- void revalidateAllKeys ()
- void addPair (std::string key, std::string val)
- void addPair (std::string key, std::string val, int t)
- void setName (std::string name)
- void setAlias (std::string alias)
- void setNameAliasPair (std::string n, std::string a, int s)
- void setState (int state)
- void DisplayContents ()
- void addHeadKey (std::string key)
- void copyAnchor2Alias (std::string alias, Header &ref)
- int size ()
- std::string getName ()
- std::string getAlias ()
- int getState ()
- bool isAlias ()
- bool isAnchor ()
- Header & getAnchoredHeader (std::string alias)
- Header & getHeadFromSubAlias (std::string alias)

## Private Attributes

- std::map< std::string, Header > Head_Map

## Additional Inherited Members

## 4.6.1 Constructor & Destructor Documentation

### 4.6.1.1 Document::Document ( )

### 4.6.1.2 Document::∼Document ( )

### 4.6.1.3 Document::Document ( const Document & *doc* )

### 4.6.1.4 Document::Document ( std::string *name* )

### 4.6.1.5 Document::Document ( const KeyValueMap & *map* )

**4.6.1.6** **Document::Document ( std::string** *name,* **const KeyValueMap &** *map* **)**

**4.6.1.7** **Document::Document ( std::string** *key,* **const Header &** *head* **)**

**4.6.2** **Member Function Documentation**

**4.6.2.1** **Document& Document::operator= ( const Document &** *doc* **)**

**4.6.2.2** **ValueTypePair& Document::operator[] ( const std::string** *key* **)**

**4.6.2.3** **ValueTypePair Document::operator[] ( const std::string** *key* **) const**

**4.6.2.4** **Header& Document::operator() ( const std::string** *key* **)**

**4.6.2.5** **Header Document::operator() ( const std::string** *key* **) const**

**4.6.2.6** **std::map<std::string, Header>& Document::getHeadMap (  )**

**4.6.2.7** **KeyValueMap& Document::getDataMap (  )**

**4.6.2.8** **Header& Document::getHeader ( std::string** *key* **)**

**4.6.2.9** **std::map<std::string, Header>::const_iterator Document::end (  ) const**

**4.6.2.10** **std::map<std::string, Header>::iterator Document::end (  )**

**4.6.2.11** **std::map<std::string, Header>::const_iterator Document::begin (  ) const**

**4.6.2.12** **std::map<std::string, Header>::iterator Document::begin (  )**

**4.6.2.13** **void Document::clear (  )**

**4.6.2.14** **void Document::resetKeys (  )**

**4.6.2.15** **void Document::changeKey ( std::string** *oldKey,* **std::string** *newKey* **)**

**4.6.2.16** **void Document::revalidateAllKeys (  )**

**4.6.2.17** **void Document::addPair ( std::string** *key,* **std::string** *val* **)**

**4.6.2.18** **void Document::addPair ( std::string** *key,* **std::string** *val,* **int** *t* **)**

**4.6.2.19** **void Document::setName ( std::string** *name* **)**

**4.6.2.20** **void Document::setAlias ( std::string** *alias* **)**

**4.6.2.21** **void Document::setNameAliasPair ( std::string** *n,* **std::string** *a,* **int** *s* **)**

**4.6.2.22** **void Document::setState ( int** *state* **)**

**4.6.2.23** **void Document::DisplayContents (  )**

**4.6.2.24** **void Document::addHeadKey ( std::string** *key* **)**

**4.6.2.25** **void Document::copyAnchor2Alias ( std::string** *alias,* **Header &** *ref* **)**

**4.6.2.26  int Document::size ( )**

**4.6.2.27  std::string Document::getName ( )**

**4.6.2.28  std::string Document::getAlias ( )**

**4.6.2.29  int Document::getState ( )**

**4.6.2.30  bool Document::isAlias ( )**

**4.6.2.31  bool Document::isAnchor ( )**

**4.6.2.32  Header& Document::getAnchoredHeader ( std::string *alias* )**

**4.6.2.33  Header& Document::getHeadFromSubAlias ( std::string *alias* )**

**4.6.3  Member Data Documentation**

**4.6.3.1  std::map<std::string, Header> Document::Head_Map** `[private]`

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

- yaml_wrapper.h

## 4.7  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.*
- 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.*

### 4.7.1 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.

### 4.7.2 Member Data Documentation

#### 4.7.2.1 unsigned long int DOGFISH_DATA::total_steps = 0

Total number of solver steps taken.

#### 4.7.2.2 double DOGFISH_DATA::time_old = 0.0

Old value of time (hrs)

#### 4.7.2.3 double DOGFISH_DATA::time = 0.0

Current value of time (hrs)

#### 4.7.2.4 bool DOGFISH_DATA::Print2File = true

True = results to .txt; False = no printing.

**4.7.2.5   bool DOGFISH_DATA::Print2Console = true**

True = results to console; False = no printing.

**4.7.2.6   bool DOGFISH_DATA::DirichletBC = false**

False = uses film mass transfer for BC, True = Dirichlet BC.

**4.7.2.7   bool DOGFISH_DATA::NonLinear = false**

False = Solve directly, True = Solve iteratively.

**4.7.2.8   double DOGFISH_DATA::t_counter = 0.0**

Counter for the time output.

**4.7.2.9   double DOGFISH_DATA::t_print**

Print output at every t_print time (hrs)

**4.7.2.10   int DOGFISH_DATA::NumComp**

Number of species to track.

**4.7.2.11   double DOGFISH_DATA::end_time**

Units: hours.

**4.7.2.12   double DOGFISH_DATA::total_sorption_old**

Per mass or volume of single fiber.

**4.7.2.13   double DOGFISH_DATA::total_sorption**

Per mass or volume of single fiber.

**4.7.2.14   double DOGFISH_DATA::fiber_length**

Units: um.

**4.7.2.15   double DOGFISH_DATA::fiber_diameter**

Units: um.

**4.7.2.16   FILE∗ DOGFISH_DATA::OutputFile**

Output file pointer to the output file for postprocesses and results.

**4.7.2.17   double(∗ DOGFISH␣DATA::eval␣R)(int i, int l, const void ∗data)**

Function pointer to evaluate retardation coefficient.

**4.7.2.18   double(∗ DOGFISH␣DATA::eval␣DI)(int i, int l, const void ∗data)**

Function pointer to evaluate intraparticle diffusivity.

**4.7.2.19   double(∗ DOGFISH␣DATA::eval_kf)(int i, const void ∗data)**

Function pointer to evaluate film mass transfer coefficient.

**4.7.2.20   double(∗ DOGFISH␣DATA::eval_qs)(int i, const void ∗data)**

Function pointer to evaluate fiber surface concentration.

**4.7.2.21   const void∗ DOGFISH␣DATA::user␣data**

Data structure for users info to calculate all parameters.

**4.7.2.22   std::vector<FINCH_DATA> DOGFISH␣DATA::finch␣dat**

Data structure for FINCH_DATA objects.

**4.7.2.23   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

## 4.8   DOGFISH␣PARAM Struct Reference

Data structure for species-specific parameters.

```
#include <dogfish.h>
```

**Public Attributes**

- double intraparticle_diffusion
    *Units: um$^\wedge$2/hr.*
- double film_transfer_coeff
    *Units: um/hr.*
- double surface_concentration
    *Units: mg/g.*
- double initial_sorption
    *Units: mg/g.*
- double sorbed_molefraction
    *Molefraction of sorbed species.*
- Molecule species
    *Adsorbed species Molecule Object.*

### 4.8.1 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.

### 4.8.2 Member Data Documentation

#### 4.8.2.1 double DOGFISH_PARAM::intraparticle_diffusion

Units: $um^{\wedge}2$/hr.

#### 4.8.2.2 double DOGFISH_PARAM::film_transfer_coeff

Units: um/hr.

#### 4.8.2.3 double DOGFISH_PARAM::surface_concentration

Units: mg/g.

#### 4.8.2.4 double DOGFISH_PARAM::initial_sorption

Units: mg/g.

#### 4.8.2.5 double DOGFISH_PARAM::sorbed_molefraction

Molefraction of sorbed species.

#### 4.8.2.6 Molecule DOGFISH_PARAM::species

Adsorbed species Molecule Object.

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

- dogfish.h

## 4.9 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 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 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)*

### 4.9.1  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.

### 4.9.2 Member Data Documentation

#### 4.9.2.1 int FINCH_DATA::d = 0

Dimension of the problem: 0 = cartesian, 1 = cylindrical, 2 = spherical.

#### 4.9.2.2 double FINCH_DATA::dt = 0.0125

Time step.

#### 4.9.2.3 double FINCH_DATA::dt_old = 0.0125

Previous time step.

#### 4.9.2.4 double FINCH_DATA::T = 1.0

Total time.

#### 4.9.2.5 double FINCH_DATA::dz = 0.1

Space step.

#### 4.9.2.6 double FINCH_DATA::L = 1.0

Total space.

#### 4.9.2.7 double FINCH_DATA::s = 1.0

Char quantity (spherical = 1, cylindrical = length, cartesian = area)

#### 4.9.2.8 double FINCH_DATA::t = 0.0

Current Time.

#### 4.9.2.9 double FINCH_DATA::t_old = 0.0

Previous Time.

#### 4.9.2.10 double FINCH_DATA::uT = 0.0

Total amount of conserved quantity in domain.

#### 4.9.2.11 double FINCH_DATA::uT_old = 0.0

Old Total amount of conserved quantity.

**4.9.2.12 double FINCH_DATA::uAvg = 0.0**

Average amount of conserved quantity in domain.

**4.9.2.13 double FINCH_DATA::uAvg_old = 0.0**

Old Average amount of conserved quantity.

**4.9.2.14 double FINCH_DATA::uIC = 0.0**

Initial condition of Conserved Quantity (if constant)

**4.9.2.15 double FINCH_DATA::vIC = 1.0**

Initial condition of Velocity (if constant)

**4.9.2.16 double FINCH_DATA::DIC = 1.0**

Initial condition of Dispersion (if constant)

**4.9.2.17 double FINCH_DATA::kIC = 1.0**

Initial condition of Reaction (if constant)

**4.9.2.18 double FINCH_DATA::RIC = 1.0**

Initial condition of the Time Coefficient (if constant)

**4.9.2.19 double FINCH_DATA::uo = 1.0**

Boundary Value of Conserved Quantity.

**4.9.2.20 double FINCH_DATA::vo = 1.0**

Boundary Value of Velocity.

**4.9.2.21 double FINCH_DATA::Do = 1.0**

Boundary Value of Dispersion.

**4.9.2.22 double FINCH_DATA::ko = 1.0**

Boundary Value of Reaction.

**4.9.2.23 double FINCH_DATA::Ro = 1.0**

Boundary Value of Time Coefficient.

**4.9.2.24 double FINCH_DATA::kfn = 1.0**

Film mass transfer coefficient Old.

**4.9.2.25 double FINCH_DATA::kfnp1 = 1.0**

Film mass transfer coefficient New.

**4.9.2.26 double FINCH_DATA::lambda_I**

Boundary Coefficient for Implicit Neumann (Calculated at Runtime)

**4.9.2.27 double FINCH_DATA::lambda_E**

Boundary Coefficient for Explicit Neumann (Calculated at Runtime)

**4.9.2.28 int FINCH_DATA::LN = 10**

Number of nodes.

**4.9.2.29 bool FINCH_DATA::CN = true**

True if Crank-Nicholson, false if Implicit, never use explicit.

**4.9.2.30 bool FINCH_DATA::Update = false**

Flag to check if the system needs updating.

**4.9.2.31 bool FINCH_DATA::Dirichlet = false**

Flag to indicate use of Dirichlet or Neumann starting boundary.

**4.9.2.32 bool FINCH_DATA::CheckMass = false**

Flag to indicate whether or not mass is to be checked.

**4.9.2.33 bool FINCH_DATA::ExplicitFlux = false**

Flag to indicate whether or not to use fully explicit flux limiters.

**4.9.2.34 bool FINCH_DATA::Iterative = true**

Flag to indicate whether to solve directly, or iteratively.

**4.9.2.35 bool FINCH_DATA::SteadyState = false**

Flag to determine whether or not to solve the steady-state problem.

**4.9.2.36 bool FINCH_DATA::NormTrack = true**

Flag to determine whether or not to track the norms during simulation.

**4.9.2.37 double FINCH_DATA::beta = 0.5**

Scheme type indicator: 0.5=CN & 1.0=Implicit; all else NULL.

**4.9.2.38 double FINCH_DATA::tol_rel = 1e-6**

Relative Tolerance for Convergence.

**4.9.2.39 double FINCH_DATA::tol_abs = 1e-6**

Absolute Tolerance for Convergence.

**4.9.2.40 int FINCH_DATA::max_iter = 20**

Maximum number of iterations allowed.

**4.9.2.41 int FINCH_DATA::total_iter = 0**

Total number of iterations made.

**4.9.2.42 int FINCH_DATA::nl_method = FINCH_Picard**

Non-linear solution method - default = FINCH_Picard.

**4.9.2.43 std::vector<double> FINCH_DATA::CL_I**

Left side, implicit coefficients (Calculated at Runtime)

**4.9.2.44 std::vector<double> FINCH_DATA::CL_E**

Left side, explicit coefficients (Calculated at Runtime)

**4.9.2.45 std::vector<double> FINCH_DATA::CC_I**

Centered, implicit coefficients (Calculated at Runtime)

**4.9.2.46 std::vector<double> FINCH_DATA::CC_E**

Centered, explicit coefficients (Calculated at Runtime)

**4.9.2.47 std::vector<double> FINCH_DATA::CR_I**

Right side, implicit coefficients (Calculated at Runtime)

**4.9.2.48   std::vector<double> FINCH_DATA::CR_E**

Right side, explicit coefficients (Calculated at Runtime)

**4.9.2.49   std::vector<double> FINCH_DATA::fL_I**

Left side, implicit fluxes (Calculated at Runtime)

**4.9.2.50   std::vector<double> FINCH_DATA::fL_E**

Left side, explicit fluxes (Calculated at Runtime)

**4.9.2.51   std::vector<double> FINCH_DATA::fC_I**

Centered, implicit fluxes (Calculated at Runtime)

**4.9.2.52   std::vector<double> FINCH_DATA::fC_E**

Centered, explicit fluxes (Calculated at Runtime)

**4.9.2.53   std::vector<double> FINCH_DATA::fR_I**

Right side, implicit fluxes (Calculated at Runtime)

**4.9.2.54   std::vector<double> FINCH_DATA::fR_E**

Right side, explicit fluxes (Calculated at Runtime)

**4.9.2.55   std::vector<double> FINCH_DATA::OI**

Implicit upper diagonal matrix elements (Calculated at Runtime)

**4.9.2.56   std::vector<double> FINCH_DATA::OE**

Explicit upper diagonal matrix elements (Calculated at Runtime)

**4.9.2.57   std::vector<double> FINCH_DATA::NI**

Implicit diagonal matrix elements (Calculated at Runtime)

**4.9.2.58   std::vector<double> FINCH_DATA::NE**

Explicit diagonal matrix elements (Calculated at Runtime)

**4.9.2.59   std::vector<double> FINCH_DATA::MI**

Implicit lower diagonal matrix elements (Calculated at Runtime)

**4.9.2.60   std::vector<double> FINCH_DATA::ME**

Explicit lower diagonal matrix elements (Calculated at Runtime)

**4.9.2.61   std::vector<double> FINCH_DATA::uz_l_I**

**4.9.2.62   std::vector<double> FINCH_DATA::uz_lm1_I**

**4.9.2.63   std::vector<double> FINCH_DATA::uz_lp1_I**

Implicit local slopes (Calculated at Runtime)

**4.9.2.64   std::vector<double> FINCH_DATA::uz_l_E**

**4.9.2.65   std::vector<double> FINCH_DATA::uz_lm1_E**

**4.9.2.66   std::vector<double> FINCH_DATA::uz_lp1_E**

Explicit local slopes (Calculated at Runtime)

**4.9.2.67   Matrix<double> FINCH_DATA::unm1**

Conserved Quantity Older.

**4.9.2.68   Matrix<double> FINCH_DATA::un**

Conserved Quantity Old.

**4.9.2.69   Matrix<double> FINCH_DATA::unp1**

Conserved Quantity New.

**4.9.2.70   Matrix<double> FINCH_DATA::u_star**

Conserved Quantity Projected New.

**4.9.2.71   Matrix<double> FINCH_DATA::ubest**

Best found solution if solving iteratively.

**4.9.2.72   Matrix<double> FINCH_DATA::vn**

Velocity Old.

**4.9.2.73   Matrix<double> FINCH_DATA::vnp1**

Velocity New.

**4.9.2.74   Matrix<double> FINCH_DATA::Dn**

Dispersion Old.

**4.9.2.75   Matrix<double> FINCH_DATA::Dnp1**

Dispersion New.

**4.9.2.76   Matrix<double> FINCH_DATA::kn**

Reaction Old.

**4.9.2.77   Matrix<double> FINCH_DATA::knp1**

Reaction New.

**4.9.2.78   Matrix<double> FINCH_DATA::Sn**

Forcing Function Old.

**4.9.2.79   Matrix<double> FINCH_DATA::Snp1**

Forcing Function New.

**4.9.2.80   Matrix<double> FINCH_DATA::Rn**

Time Coeff Old.

**4.9.2.81   Matrix<double> FINCH_DATA::Rnp1**

Time Coeff New.

**4.9.2.82   Matrix<double> FINCH_DATA::Fn**

Flux Limiter Old.

**4.9.2.83   Matrix<double> FINCH_DATA::Fnp1**

Flux Limiter New.

**4.9.2.84   Matrix<double> FINCH_DATA::gI**

Implicit Side Boundary Conditions.

**4.9.2.85   Matrix<double> FINCH_DATA::gE**

Explicit Side Boundary Conditions.

**4.9.2.86   Matrix<double> FINCH_DATA::res**

Current residual.

**4.9.2.87  Matrix<double> FINCH_DATA::pres**

Current search direction.

**4.9.2.88  int(∗ FINCH_DATA::callroutine)(const void ∗user_data)**

Function pointer to executioner (DEFAULT = default_execution)

**4.9.2.89  int(∗ FINCH_DATA::setic)(const void ∗user_data)**

Function pointer to initial conditions (DEFAULT = default_ic)

**4.9.2.90  int(∗ FINCH_DATA::settime)(const void ∗user_data)**

Function pointer to set time step (DEFAULT = default_timestep)

**4.9.2.91  int(∗ FINCH_DATA::setpreprocess)(const void ∗user_data)**

Function pointer to preprocesses (DEFAULT = default_preprocess)

**4.9.2.92  int(∗ FINCH_DATA::solve)(const void ∗user_data)**

Function pointer to the solver (DEFAULT = default_solve)

**4.9.2.93  int(∗ FINCH_DATA::setparams)(const void ∗user_data)**

Function pointer to set parameters (DEFAULT = default_params)

**4.9.2.94  int(∗ FINCH_DATA::discretize)(const void ∗user_data)**

Function pointer to discretization (DEFAULT = ospre_discretization)

**4.9.2.95  int(∗ FINCH_DATA::setbcs)(const void ∗user_data)**

Function pointer to set boundary conditions (DEFAULT = default_bcs)

**4.9.2.96  int(∗ FINCH_DATA::evalres)(const Matrix< double > &x, Matrix< double > &res, const void ∗user_data)**

Function pointer to the residual function (DEFAULT = default_res)

**4.9.2.97  int(∗ FINCH_DATA::evalprecon)(const Matrix< double > &b, Matrix< double > &p, const void ∗user_data)**

Function pointer to the preconditioning function (DEFAULT = default_precon)

**4.9.2.98  int(∗ FINCH_DATA::setpostprocess)(const void ∗user_data)**

Function pointer to the postprocesses (DEFAULT = default_postprocess)

**4.9.2.99 int(∗ FINCH_DATA::resettime)(const void ∗user_data)**

Function pointer to reset time (DEFAULT = default_reset)

**4.9.2.100 PICARD_DATA FINCH_DATA::picard_dat**

Data structure for PICARD method (no need to use this)

**4.9.2.101 PJFNK_DATA FINCH_DATA::pjfnk_dat**

Data structure for PJFNK method (more rigours method)

**4.9.2.102 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

## 4.10 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 = 50.*
- 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.*

### 4.10.1    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.

### 4.10.2    Member Data Documentation

#### 4.10.2.1    int GCR_DATA::restart = -1

Restart parameter for outer iterations - default = 50.

#### 4.10.2.2    int GCR_DATA::maxit = 0

Maximum allowable outer iterations.

#### 4.10.2.3    int GCR_DATA::iter_outer = 0

Number of outer iterations taken.

#### 4.10.2.4    int GCR_DATA::iter_inner = 0

Number of inner iterations taken.

**4.10.2.5 int GCR_DATA::total_iter = 0**

Total number of iterations taken.

**4.10.2.6 bool GCR_DATA::breakdown = false**

Boolean to determine if a step has failed.

**4.10.2.7 double GCR_DATA::alpha**

Inner iteration step size.

**4.10.2.8 double GCR_DATA::beta**

Outer iteration step size.

**4.10.2.9 double GCR_DATA::tol_rel = 1e-6**

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

**4.10.2.10 double GCR_DATA::tol_abs = 1e-6**

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

**4.10.2.11 double GCR_DATA::res**

Absolute residual norm for linear system.

**4.10.2.12 double GCR_DATA::relres**

Relative residual norm for linear system.

**4.10.2.13 double GCR_DATA::relres_base**

Initial residual norm of the linear system.

**4.10.2.14 double GCR_DATA::bestres**

Best found residual norm of the linear system.

**4.10.2.15 bool GCR_DATA::Output = true**

True = print messages to the console.

**4.10.2.16 Matrix<double> GCR_DATA::x**

Current solution to the linear system.

**4.10.2.17 Matrix⟨double⟩ GCR_DATA::bestx**

Best found solution to the linear system.

**4.10.2.18 Matrix⟨double⟩ GCR_DATA::r**

Residual Vector.

**4.10.2.19 Matrix⟨double⟩ GCR_DATA::c_temp**

Temporary c vector to be updated.

**4.10.2.20 Matrix⟨double⟩ GCR_DATA::u_temp**

Temporary u vector to be updated.

**4.10.2.21 std::vector⟨Matrix⟨double⟩ ⟩ GCR_DATA::u**

Vector span for updating x.

**4.10.2.22 std::vector⟨Matrix⟨double⟩ ⟩ GCR_DATA::c**

Vector span for updating r.

**4.10.2.23 OPTRANS_DATA GCR_DATA::transpose_dat**

Data structure for Operator Transposition.

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

- lark.h

## 4.11 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,50)*
- 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.*

### 4.11.1 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.

### 4.11.2 Member Data Documentation

#### 4.11.2.1 int GMRESLP_DATA::restart = -1

Restart parameter - default = min(vector_size,50)

#### 4.11.2.2 int GMRESLP_DATA::maxit = 0

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

#### 4.11.2.3 int GMRESLP_DATA::iter = 0

Number of iterations needed for convergence.

#### 4.11.2.4 int GMRESLP_DATA::steps = 0

Total number of gmres iterations and krylov iterations.

#### 4.11.2.5 double GMRESLP_DATA::tol_rel = 1e-6

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

**4.11.2.6 double GMRESLP_DATA::tol_abs = 1e-6**

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

**4.11.2.7 double GMRESLP_DATA::res**

Absolution redisual norm of the linear system.

**4.11.2.8 double GMRESLP_DATA::relres**

Relative residual norm of the linear system.

**4.11.2.9 double GMRESLP_DATA::relres_base**

Initial residual norm of the linear system.

**4.11.2.10 double GMRESLP_DATA::bestres**

Best found residual norm of the linear system.

**4.11.2.11 bool GMRESLP_DATA::Output = true**

True = print messages to console.

**4.11.2.12 Matrix<double> GMRESLP_DATA::x**

Current solution to the linear system.

**4.11.2.13 Matrix<double> GMRESLP_DATA::bestx**

Best found solution to the linear system.

**4.11.2.14 Matrix<double> GMRESLP_DATA::r**

Residual vector for the linear system.

**4.11.2.15 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

## 4.12 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 = 50, 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.*

### 4.12.1 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).

### 4.12.2 Member Data Documentation

#### 4.12.2.1 int GMRESR_DATA::gcr_restart = -1

Number of GCR restarts (default = 50, max = N)

#### 4.12.2.2 int GMRESR_DATA::gcr_maxit = 0

Number of GCR iterations.

#### 4.12.2.3 int GMRESR_DATA::gmres_restart = -1

Number of GMRES restarts (max = 20)

#### 4.12.2.4 int GMRESR_DATA::gmres_maxit = 1

Number of GMRES iterations (max = 5, default = 1)

#### 4.12.2.5 int GMRESR_DATA::N

Dimension of the linear system.

#### 4.12.2.6 int GMRESR_DATA::total_iter

Total GMRES and GCR iterations.

#### 4.12.2.7 int GMRESR_DATA::iter_outer

Total GCR iterations.

#### 4.12.2.8 int GMRESR_DATA::iter_inner

Total GMRES iterations.

#### 4.12.2.9 bool GMRESR_DATA::GCR_Output = true

True = print GCR messages.

#### 4.12.2.10 bool GMRESR_DATA::GMRES_Output = false

True = print GMRES messages.

#### 4.12.2.11 double GMRESR_DATA::gmres_tol = 0.1

Tolerance relative to GCR iterations.

#### 4.12.2.12 double GMRESR_DATA::gcr_rel_tol = 1e-6

Relative outer residual tolerance.

**4.12.2.13   double GMRESR_DATA::gcr_abs_tol = 1e-6**

Absolute outer residual tolerance.

**4.12.2.14   Matrix<double> GMRESR_DATA::arg**

Argument matrix passed between preconditioner and iterator.

**4.12.2.15   GCR_DATA GMRESR_DATA::gcr_dat**

Data structure for the outer GCR steps.

**4.12.2.16   GMRESRP_DATA GMRESR_DATA::gmres_dat**

Data structure for the inner GMRES steps.

**4.12.2.17   int(∗ GMRESR_DATA::matvec)(const Matrix< double > &x, Matrix< double > &Ax, const void ∗matvec_data)**

User supplied matrix-vector product function.

**4.12.2.18   int(∗ GMRESR_DATA::terminal_precon)(const Matrix< double > &r, Matrix< double > &p, const void ∗precon_data)**

Optional user supplied terminal preconditioner.

**4.12.2.19   const void∗ GMRESR_DATA::matvec_data**

Data structure for the user's matvec function.

**4.12.2.20   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

# 4.13   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(50,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< 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*

### 4.13.1 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.

### 4.13.2 Member Data Documentation

#### 4.13.2.1 int GMRESRP_DATA::restart = -1

Restart parameter - default = min(50,vector_size)

#### 4.13.2.2 int GMRESRP_DATA::maxit = 0

Maximum allowable outer iterations.

#### 4.13.2.3 int GMRESRP_DATA::iter_outer = 0

Total number of outer iterations.

#### 4.13.2.4 int GMRESRP_DATA::iter_inner = 0

Total number of inner iterations.

#### 4.13.2.5 int GMRESRP_DATA::iter_total = 0

Total number of overall iterations.

#### 4.13.2.6 double GMRESRP_DATA::tol_rel = 1e-6

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

#### 4.13.2.7 double GMRESRP_DATA::tol_abs = 1e-6

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

#### 4.13.2.8 double GMRESRP_DATA::res

Absolute residual norm for linear system.

#### 4.13.2.9 double GMRESRP_DATA::relres

Relative residual norm for linear system.

#### 4.13.2.10 double GMRESRP_DATA::relres_base

Initial residual norm of the linear system.

#### 4.13.2.11 double GMRESRP_DATA::bestres

Best found residual norm of the linear system.

#### 4.13.2.12 bool GMRESRP_DATA::Output = true

True = print messages to console.

**4.13.2.13 Matrix⟨double⟩ GMRESRP_DATA::x**

Current solution to the linear system.

**4.13.2.14 Matrix⟨double⟩ GMRESRP_DATA::bestx**

Best found solution to the linear system.

**4.13.2.15 Matrix⟨double⟩ GMRESRP_DATA::r**

Residual vector for the linear system.

**4.13.2.16 std::vector⟨ Matrix⟨double⟩ ⟩ GMRESRP_DATA::Vk**

(N x k) orthonormal vector basis

**4.13.2.17 std::vector⟨ std::vector⟨ double ⟩ ⟩ GMRESRP_DATA::H**

(k+1 x k) upper Hessenberg storage matrix

**4.13.2.18 std::vector⟨ std::vector⟨ double ⟩ ⟩ GMRESRP_DATA::H_bar**

(k+1 x k) Factorized matrix

**4.13.2.19 std::vector⟨ double ⟩ GMRESRP_DATA::y**

(k x 1) Vector search direction

**4.13.2.20 std::vector⟨ double ⟩ GMRESRP_DATA::e0**

(k+1 x 1) Normalized vector with residual info

**4.13.2.21 std::vector⟨ double ⟩ GMRESRP_DATA::e0_bar**

(k+1 x 1) Factorized normal vector

**4.13.2.22 Matrix⟨double⟩ GMRESRP_DATA::w**

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

**4.13.2.23 Matrix⟨double⟩ GMRESRP_DATA::v**

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

**4.13.2.24 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

## 4.14 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.*

### 4.14.1 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.

### 4.14.2 Member Data Documentation

#### 4.14.2.1 double GPAST_DATA::x

Adsorbed mole fraction.

#### 4.14.2.2 double GPAST_DATA::y

Gas phase mole fraction.

#### 4.14.2.3 double GPAST_DATA::He

Henry's Coefficient (mol/kg/kPa)

**4.14.2.4 double GPAST_DATA::q**

Amount adsorbed for each component (mol/kg)

**4.14.2.5 std::vector$<$double$>$ GPAST_DATA::gama_inf**

Infinite dilution activities.

**4.14.2.6 double GPAST_DATA::qo**

Pure component capacities (mol/kg)

**4.14.2.7 double GPAST_DATA::Plo**

Pure component spreading pressures (mol/kg)

**4.14.2.8 std::vector$<$double$>$ GPAST_DATA::po**

Pure component reference state pressures (kPa)

**4.14.2.9 double GPAST_DATA::poi**

Reference state pressures solved for using Recover eval GPAST.

**4.14.2.10 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

## 4.15 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))*

### 4.15.1 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.

### 4.15.2 Member Data Documentation

#### 4.15.2.1 double GSTA_DATA::qmax

Theoretical maximum capacity of adsorbate-adsorbent pair (mol/kg)

#### 4.15.2.2 int GSTA_DATA::m

Number of parameters in the GSTA isotherm.

#### 4.15.2.3 std::vector<double> GSTA_DATA::dHo

Enthalpies for each site (J/mol)

#### 4.15.2.4 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

## 4.16 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.*

### 4.16.1 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.

### 4.16.2 Member Data Documentation

#### 4.16.2.1 int GSTA_OPT_DATA::total_eval

Keeps track of the total number of function evaluations.

#### 4.16.2.2 int GSTA_OPT_DATA::n_par

Number of parameters being optimized for.

#### 4.16.2.3 double GSTA_OPT_DATA::qmax

Maximum theoretical adsorption capacity (M/M) (0 if unknown)

#### 4.16.2.4 int GSTA_OPT_DATA::iso

Keeps isotherm that is currently being optimized.

#### 4.16.2.5 std::vector<std::vector<double>> GSTA_OPT_DATA::Fobj

Creates a dynamic array to store all Fobj values.

#### 4.16.2.6 std::vector<std::vector<double>> GSTA_OPT_DATA::q

#### 4.16.2.7 std::vector<std::vector<double>> GSTA_OPT_DATA::P

Creates a dynamic array for q and P data pairs.

**4.16.2.8  std::vector<std::vector<double> > GSTA_OPT_DATA::best_par**

Used to store the values of the parameters of best fit.

**4.16.2.9  std::vector<std::vector<double> > GSTA_OPT_DATA::Kno**

Dimensionless parameters determined from best_par.

**4.16.2.10  std::vector<std::vector<std::vector<double> > > GSTA_OPT_DATA::all_pars**

Used to create a ragged array of all parameters.

**4.16.2.11  std::vector<std::vector<double> > GSTA_OPT_DATA::norms**

Used to store the values of all the calculated norms.

**4.16.2.12  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

## 4.17  Header Class Reference

```
#include <yaml_wrapper.h>
```

Inheritance diagram for Header:
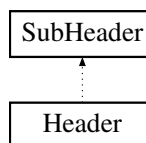


**Public Member Functions**

- Header ()
- ∼Header ()
- Header (const Header &head)
- Header (std::string name)
- Header (const KeyValueMap &map)
- Header (std::string name, const KeyValueMap &map)
- Header (std::string key, const SubHeader &sub)
- Header & operator= (const Header &head)
- ValueTypePair & operator[] (const std::string key)
- ValueTypePair operator[] (const std::string key) const
- SubHeader & operator() (const std::string key)
- SubHeader operator() (const std::string key) const
- std::map< std::string,
  SubHeader > & getSubMap ()

- KeyValueMap & getDataMap ()
- SubHeader & getSubHeader (std::string key)
- std::map< std::string,
  SubHeader >::const_iterator end () const
- std::map< std::string,
  SubHeader >::iterator end ()
- std::map< std::string,
  SubHeader >::const_iterator begin () const
- std::map< std::string,
  SubHeader >::iterator begin ()
- void clear ()
- void resetKeys ()
- void changeKey (std::string oldKey, std::string newKey)
- void addPair (std::string key, std::string val)
- void addPair (std::string key, std::string val, int t)
- void setName (std::string name)
- void setAlias (std::string alias)
- void setNameAliasPair (std::string n, std::string a, int s)
- void setState (int state)
- void DisplayContents ()
- void addSubKey (std::string key)
- void copyAnchor2Alias (std::string alias, SubHeader &ref)
- int size ()
- std::string getName ()
- std::string getAlias ()
- int getState ()
- bool isAlias ()
- bool isAnchor ()
- SubHeader & getAnchoredSub (std::string alias)

**Private Attributes**

- std::map< std::string, SubHeader > Sub_Map

**Additional Inherited Members**

**4.17.1 Constructor & Destructor Documentation**

**4.17.1.1 Header::Header (  )**

**4.17.1.2 Header::∼Header (  )**

**4.17.1.3 Header::Header ( const Header & *head* )**

**4.17.1.4 Header::Header ( std::string *name* )**

**4.17.1.5 Header::Header ( const KeyValueMap & *map* )**

**4.17.1.6 Header::Header ( std::string *name,* const KeyValueMap & *map* )**

**4.17.1.7 Header::Header ( std::string *key,* const SubHeader & *sub* )**

**4.17.2 Member Function Documentation**

**4.17.2.1** **Header& Header::operator= (** const **Header** & *head* **)**

**4.17.2.2** **ValueTypePair& Header::operator[] (** const std::string *key* **)**

**4.17.2.3** **ValueTypePair Header::operator[] (** const std::string *key* **) const**

**4.17.2.4** **SubHeader& Header::operator() (** const std::string *key* **)**

**4.17.2.5** **SubHeader Header::operator() (** const std::string *key* **) const**

**4.17.2.6** std::map⟨std::string, **SubHeader**⟩& Header::getSubMap **(** **)**

**4.17.2.7** **KeyValueMap&** Header::getDataMap **(** **)**

**4.17.2.8** **SubHeader&** Header::getSubHeader **(** std::string *key* **)**

**4.17.2.9** std::map⟨std::string, **SubHeader**⟩::const_iterator Header::end **(** **) const**

**4.17.2.10** std::map⟨std::string, **SubHeader**⟩::iterator Header::end **(** **)**

**4.17.2.11** std::map⟨std::string, **SubHeader**⟩::const_iterator Header::begin **(** **) const**

**4.17.2.12** std::map⟨std::string, **SubHeader**⟩::iterator Header::begin **(** **)**

**4.17.2.13** void Header::clear **(** **)**

**4.17.2.14** void Header::resetKeys **(** **)**

**4.17.2.15** void Header::changeKey **(** std::string *oldKey,* std::string *newKey* **)**

**4.17.2.16** void Header::addPair **(** std::string *key,* std::string *val* **)**

**4.17.2.17** void Header::addPair **(** std::string *key,* std::string *val,* int *t* **)**

**4.17.2.18** void Header::setName **(** std::string *name* **)**

**4.17.2.19** void Header::setAlias **(** std::string *alias* **)**

**4.17.2.20** void Header::setNameAliasPair **(** std::string *n,* std::string *a,* int *s* **)**

**4.17.2.21** void Header::setState **(** int *state* **)**

**4.17.2.22** void Header::DisplayContents **(** **)**

**4.17.2.23** void Header::addSubKey **(** std::string *key* **)**

**4.17.2.24** void Header::copyAnchor2Alias **(** std::string *alias,* **SubHeader** & *ref* **)**

**4.17.2.25** int Header::size **(** **)**

**4.17.2.26** std::string Header::getName **(** **)**

**4.17.2.27** std::string Header::getAlias **(** **)**

**4.17.2.28** int Header::getState **(** **)**

**4.17.2.29   bool Header::isAlias (   )**

**4.17.2.30   bool Header::isAnchor (   )**

**4.17.2.31   SubHeader& Header::getAnchoredSub (  std::string *alias* )**

### 4.17.3   Member Data Documentation

**4.17.3.1   std::map$<$std::string, SubHeader$>$ Header::Sub_Map**   `[private]`

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

- yaml_wrapper.h

## 4.18   KeyValueMap Class Reference

```
#include <yaml_wrapper.h>
```

**Public Member Functions**

- KeyValueMap ()
- ∼KeyValueMap ()
- KeyValueMap (const std::map$<$ std::string, std::string $>$ &map)
- KeyValueMap (std::string key, std::string value)
- KeyValueMap (const KeyValueMap &map)
- KeyValueMap & operator= (const KeyValueMap &map)
- ValueTypePair & operator[] (const std::string key)
- ValueTypePair operator[] (const std::string key) const
- std::map$<$ std::string,
  ValueTypePair $>$ & getMap ()
- std::map$<$ std::string,
  ValueTypePair $>$
  ::const_iterator end () const
- std::map$<$ std::string,
  ValueTypePair $>$::iterator end ()
- std::map$<$ std::string,
  ValueTypePair $>$
  ::const_iterator begin () const
- std::map$<$ std::string,
  ValueTypePair $>$::iterator begin ()
- void clear ()
- void addKey (std::string key)
- void editValue4Key (std::string val, std::string key)
- void editValue4Key (std::string val, int type, std::string key)
- void addPair (std::string key, ValueTypePair val)
- void addPair (std::string key, std::string val)
- void addPair (std::string key, std::string val, int type)
- void findType (std::string key)
- void assertType (std::string key, int type)
- void findAllTypes ()
- void DisplayMap ()
- int size ()
- std::string getString (std::string key)

- bool getBool (std::string key)
- double getDouble (std::string key)
- int getInt (std::string key)
- std::string getValue (std::string key)
- int getType (std::string key)
- ValueTypePair & getPair (std::string key)

## Private Attributes

- std::map< std::string,
  ValueTypePair > Key_Value

### 4.18.1 Constructor & Destructor Documentation

#### 4.18.1.1 KeyValueMap::KeyValueMap ( )

#### 4.18.1.2 KeyValueMap::∼KeyValueMap ( )

#### 4.18.1.3 KeyValueMap::KeyValueMap ( const std::map< std::string, std::string > & *map* )

#### 4.18.1.4 KeyValueMap::KeyValueMap ( std::string *key,* std::string *value* )

#### 4.18.1.5 KeyValueMap::KeyValueMap ( const **KeyValueMap** & *map* )

### 4.18.2 Member Function Documentation

#### 4.18.2.1 KeyValueMap& KeyValueMap::operator= ( const **KeyValueMap** & *map* )

#### 4.18.2.2 ValueTypePair& KeyValueMap::operator[] ( const std::string *key* )

#### 4.18.2.3 ValueTypePair KeyValueMap::operator[] ( const std::string *key* ) const

#### 4.18.2.4 std::map<std::string, ValueTypePair >& KeyValueMap::getMap ( )

#### 4.18.2.5 std::map<std::string, ValueTypePair>::const_iterator KeyValueMap::end ( ) const

#### 4.18.2.6 std::map<std::string, ValueTypePair>::iterator KeyValueMap::end ( )

#### 4.18.2.7 std::map<std::string, ValueTypePair>::const_iterator KeyValueMap::begin ( ) const

#### 4.18.2.8 std::map<std::string, ValueTypePair>::iterator KeyValueMap::begin ( )

#### 4.18.2.9 void KeyValueMap::clear ( )

#### 4.18.2.10 void KeyValueMap::addKey ( std::string *key* )

#### 4.18.2.11 void KeyValueMap::editValue4Key ( std::string *val,* std::string *key* )

#### 4.18.2.12 void KeyValueMap::editValue4Key ( std::string *val,* int *type,* std::string *key* )

#### 4.18.2.13 void KeyValueMap::addPair ( std::string *key,* **ValueTypePair** *val* )

#### 4.18.2.14 void KeyValueMap::addPair ( std::string *key,* std::string *val* )

**4.18.2.15 void KeyValueMap::addPair ( std::string *key,* std::string *val,* int *type* )**

**4.18.2.16 void KeyValueMap::findType ( std::string *key* )**

**4.18.2.17 void KeyValueMap::assertType ( std::string *key,* int *type* )**

**4.18.2.18 void KeyValueMap::findAllTypes ( )**

**4.18.2.19 void KeyValueMap::DisplayMap ( )**

**4.18.2.20 int KeyValueMap::size ( )**

**4.18.2.21 std::string KeyValueMap::getString ( std::string *key* )**

**4.18.2.22 bool KeyValueMap::getBool ( std::string *key* )**

**4.18.2.23 double KeyValueMap::getDouble ( std::string *key* )**

**4.18.2.24 int KeyValueMap::getInt ( std::string *key* )**

**4.18.2.25 std::string KeyValueMap::getValue ( std::string *key* )**

**4.18.2.26 int KeyValueMap::getType ( std::string *key* )**

**4.18.2.27 ValueTypePair& KeyValueMap::getPair ( std::string *key* )**

**4.18.3 Member Data Documentation**

**4.18.3.1 std::map<std::string, ValueTypePair > KeyValueMap::Key_Value** `[private]`

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

- yaml_wrapper.h

# 4.19 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

**4.19.1 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.

**4.19.2 Member Data Documentation**

**4.19.2.1 std::vector<GSTA_DATA> MAGPIE_DATA::gsta_dat**

**4.19.2.2 std::vector<mSPD_DATA> MAGPIE_DATA::mspd_dat**

**4.19.2.3 std::vector<GPAST_DATA> MAGPIE_DATA::gpast_dat**

**4.19.2.4 SYSTEM_DATA MAGPIE_DATA::sys_dat**

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

- magpie.h

## 4.20 MassBalance Class Reference

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

**Public Member Functions**

- MassBalance ()
- ∼MassBalance ()
- void Initialize_List (MasterSpeciesList &List)
- void Display_Info ()
- void Set_Delta (int i, double v)
- void Set_TotalConcentration (double v)
- void Set_Name (std::string name)
- double Get_Delta (int i)
- double Sum_Delta ()
- double Get_TotalConcentration ()
- std::string Get_Name ()
- double Eval_Residual (const Matrix< double > &x)

**Protected Attributes**

- MasterSpeciesList ∗ List
- std::vector< double > Delta
- double TotalConcentration

**Private Attributes**

- std::string Name

**4.20.1 Constructor & Destructor Documentation**

**4.20.1.1 MassBalance::MassBalance ( )**

**4.20.1.2 MassBalance::∼MassBalance ( )**

**4.20.2 Member Function Documentation**

**4.20.2.1**  **void MassBalance::Initialize␣List (  MasterSpeciesList &  *List*  )**

**4.20.2.2**  **void MassBalance::Display␣Info (   )**

**4.20.2.3**  **void MassBalance::Set␣Delta (  int  *i,*  double  *v*  )**

**4.20.2.4**  **void MassBalance::Set␣TotalConcentration (  double  *v*  )**

**4.20.2.5**  **void MassBalance::Set␣Name (  std::string  *name*  )**

**4.20.2.6**  **double MassBalance::Get␣Delta (  int  *i*  )**

**4.20.2.7**  **double MassBalance::Sum␣Delta (   )**

**4.20.2.8**  **double MassBalance::Get␣TotalConcentration (   )**

**4.20.2.9**  **std::string MassBalance::Get␣Name (   )**

**4.20.2.10**   **double MassBalance::Eval␣Residual (  const  Matrix$<$ double $>$ &  *x*  )**

**4.20.3**   **Member Data Documentation**

**4.20.3.1**  **MasterSpeciesList**∗ **MassBalance::List**   `[protected]`

**4.20.3.2**  **std::vector$<$double$>$ MassBalance::Delta**   `[protected]`

**4.20.3.3**  **double MassBalance::TotalConcentration**   `[protected]`

**4.20.3.4**  **std::string MassBalance::Name**   `[private]`

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

- shark.h

## 4.21   **MasterSpeciesList Class Reference**

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

Inheritance diagram for MasterSpeciesList:
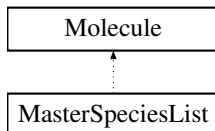


**Public Member Functions**

- MasterSpeciesList ()
- ∼MasterSpeciesList ()
- MasterSpeciesList (const MasterSpeciesList &msl)
- MasterSpeciesList & operator= (const MasterSpeciesList &msl)
- void set_list_size (int i)
- void set_species (int i, std::string formula)

- 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)
- void DisplayInfo (int i)
- void DisplayAll ()
- void DisplayConcentrations (Matrix< double > &C)
- void set_alkalinity (double alk)
- int list_size ()
- Molecule & get_species (int i)
- int get_index (std::string name)
- double charge (int i)
- double alkalinity ()
- std::string speciesName (int i)
- double Eval_ChargeResidual (const Matrix< double > &x)

## Protected Attributes

- int size
- std::vector< Molecule > species
- double residual_alkalinity

## Additional Inherited Members

### 4.21.1 Constructor & Destructor Documentation

#### 4.21.1.1 MasterSpeciesList::MasterSpeciesList ( )

#### 4.21.1.2 MasterSpeciesList::∼MasterSpeciesList ( )

#### 4.21.1.3 MasterSpeciesList::MasterSpeciesList ( const **MasterSpeciesList** & *msl* )

### 4.21.2 Member Function Documentation

#### 4.21.2.1 MasterSpeciesList& MasterSpeciesList::operator= ( const **MasterSpeciesList** & *msl* )

#### 4.21.2.2 void MasterSpeciesList::set_list_size ( int *i* )

#### 4.21.2.3 void MasterSpeciesList::set_species ( int *i,* std::string *formula* )

#### 4.21.2.4 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* )

#### 4.21.2.5 void MasterSpeciesList::DisplayInfo ( int *i* )

#### 4.21.2.6 void MasterSpeciesList::DisplayAll ( )

#### 4.21.2.7 void MasterSpeciesList::DisplayConcentrations ( **Matrix**< double > & *C* )

#### 4.21.2.8 void MasterSpeciesList::set_alkalinity ( double *alk* )

#### 4.21.2.9 int MasterSpeciesList::list_size ( )

#### 4.21.2.10 **Molecule**& MasterSpeciesList::get_species ( int *i* )

#### 4.21.2.11 int MasterSpeciesList::get_index ( std::string *name* )

**4.21.2.12   double MasterSpeciesList::charge ( int *i* )**

**4.21.2.13   double MasterSpeciesList::alkalinity (   )**

**4.21.2.14   std::string MasterSpeciesList::speciesName ( int *i* )**

**4.21.2.15   double MasterSpeciesList::Eval_ChargeResidual ( const Matrix$<$ double $>$ & *x* )**

### 4.21.3   Member Data Documentation

**4.21.3.1   int MasterSpeciesList::size**  `[protected]`

**4.21.3.2   std::vector$<$Molecule$>$ MasterSpeciesList::species**  `[protected]`

**4.21.3.3   double MasterSpeciesList::residual_alkalinity**  `[protected]`

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

- shark.h

## 4.22   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.*
- $\sim$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 Matrix &M)

    *Operator to multiply this matrix and matrix M and return the new matrix result.*

- Matrix & transpose (const Matrix &M)

    *Function to convert this matrix to the transpose of the given 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](#) & [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.*

### 4.22.1 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.

### 4.22.2 Constructor & Destructor Documentation

#### 4.22.2.1 template$<$class T $>$ Matrix$<$ T $>$::Matrix ( int *rows,* int *columns* )

Constructor for matrix with given number of rows and columns.

#### 4.22.2.2 template$<$class T $>$ Matrix$<$ T $>$::Matrix ( const Matrix$<$ T $>$ & *M* )

Copy constructor for constructing a matrix as a copy of another matrix.

#### 4.22.2.3 template$<$class T $>$ Matrix$<$ T $>$::Matrix ( )

Default constructor for creating an empty matrix.

#### 4.22.2.4 template$<$class T $>$ Matrix$<$ T $>$::$\sim$Matrix ( )

Default destructor for clearing out memory.

### 4.22.3 Member Function Documentation

#### 4.22.3.1 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))

#### 4.22.3.2 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.

#### 4.22.3.3 template$<$class T $>$ Matrix$<$ T $>$ & Matrix$<$ T $>$::operator= ( const Matrix$<$ T $>$ & *M* )

Equals operator for setting one matrix equal to another matrix.

#### 4.22.3.4 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.

#### 4.22.3.5 template$<$class T $>$ void Matrix$<$ T $>$::zeros ( )

Function to set/change all values in a matrix to zeros.

**4.22.3.6   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.

**4.22.3.7   template$<$class T $>$ int Matrix$<$ T $>$::rows (   )**

Function to return the number of rows in a given matrix.

**4.22.3.8   template$<$class T $>$ int Matrix$<$ T $>$::columns (   )**

Function to return the number of columns in a matrix.

**4.22.3.9   template$<$class T $>$ T Matrix$<$ T $>$::determinate (   )**

Function to compute the determinate of a matrix and return that value.

**4.22.3.10   template$<$class T $>$ T Matrix$<$ T $>$::norm (   )**

Function to compute the L2-norm of a matrix and return that value.

**4.22.3.11   template$<$class T $>$ T Matrix$<$ T $>$::sum (   )**

Function to compute the sum of all elements in a matrix and return that value.

**4.22.3.12   template$<$class T $>$ T Matrix$<$ T $>$::inner_product ( const Matrix$<$ T $>$ & *x* )**

Function to compute the inner product between this matrix and matrix x.

**4.22.3.13   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.

**4.22.3.14   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.

**4.22.3.15   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.

**4.22.3.16   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.

**4.22.3.17   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.

**4.22.3.18 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.

**4.22.3.19 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.

**4.22.3.20 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.

**4.22.3.21 template**$<$**class T** $>$ **Matrix**$<$ **T** $>$ **& Matrix**$<$ **T** $>$**::adjoint ( const Matrix**$<$ **T** $>$ **&** *M* **)**

Function to convert this matrix to the adjoint of the given matrix.

**4.22.3.22 template**$<$**class T** $>$ **Matrix**$<$ **T** $>$ **& Matrix**$<$ **T** $>$**::inverse ( const Matrix**$<$ **T** $>$ **&** *M* **)**

Function to convert this matrix to the inverse of the given matrix.

**4.22.3.23 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.

**4.22.3.24 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)

**4.22.3.25 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)

**4.22.3.26 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.

**4.22.3.27 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.

**4.22.3.28    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.

**4.22.3.29    template<class T> Matrix< T > & Matrix< T >::ConstantICFill ( const T *IC* )**

Function to set all values in a column matrix to a given constant.

**4.22.3.30    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.

**4.22.3.31    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 |

**4.22.3.32    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 |

**4.22.3.33    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 |

**4.22.3.34** **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.

**4.22.3.35** **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.

**4.22.3.36** **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.

**4.22.3.37** **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.

**4.22.3.38** **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)

**4.22.3.39** **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)

**4.22.3.40** **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)

**4.22.3.41** **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.

**4.22.3.42** **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.

**4.22.3.43    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)

**4.22.3.44    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)

**4.22.3.45    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.

**4.22.3.46    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.

**4.22.3.47    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.

**4.22.3.48    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.

**4.22.3.49    template$<$class T $>$ void Matrix$<$ T $>$::rowShrink (   )**

Function to delete the last row of this matrix.

**4.22.3.50    template$<$class T $>$ void Matrix$<$ T $>$::columnShrink (   )**

Function to delete the last column of this matrix.

**4.22.3.51    template$<$class T $>$ void Matrix$<$ T $>$::rowExtend ( const Matrix$<$ T $>$ & *v* )**

Function to add the row matrix v to the end of this matrix.

**4.22.3.52    template$<$class T $>$ void Matrix$<$ T $>$::columnExtend ( const Matrix$<$ T $>$ & *v* )**

Function to add the column matrix v to the end of this matrix.

**4.22.4   Member Data Documentation**

**4.22.4.1   template**$<$**class T**$>$ **int Matrix**$<$ **T** $>$**::num_rows**  `[protected]`

Number of rows of the matrix.

**4.22.4.2   template**$<$**class T**$>$ **int Matrix**$<$ **T** $>$**::num_cols**  `[protected]`

Number of columns of the matrix.

**4.22.4.3   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

## 4.23   Mechanism Class Reference

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

**Protected Attributes**

- MasterSpeciesList $*$ List
- std::vector$<$ UnsteadyReaction $>$ reactions
- std::vector$<$ double $>$ weight
- int species_index

**4.23.1   Member Data Documentation**

**4.23.1.1   MasterSpeciesList**$*$ **Mechanism::List**  `[protected]`

**4.23.1.2   std::vector**$<$**UnsteadyReaction**$>$ **Mechanism::reactions**  `[protected]`

**4.23.1.3   std::vector**$<$**double**$>$ **Mechanism::weight**  `[protected]`

**4.23.1.4   int Mechanism::species_index**  `[protected]`

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

- shark.h

## 4.24   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$^\wedge$3) {use RE3}.*
- double total_dyn_vis

    *Calculated: Total dynamic viscosity (g/cm/s)*
- double kinematic_viscosity

    *Calculated: Kinematic viscosity (cm$^\wedge$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$^\wedge$2/s)*
- std::vector< PURE_GAS > species_dat

    *Vector of the pure gas info of all species.*

### 4.24.1 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.

### 4.24.2 Member Data Documentation

#### 4.24.2.1 int MIXED_GAS::N

Given: Total number of gas species.

#### 4.24.2.2 bool MIXED_GAS::CheckMolefractions = true

Given: True = Check Molefractions for errors.

**4.24.2.3   double MIXED_GAS::total_pressure**

Given: Total gas pressure (kPa)

**4.24.2.4   double MIXED_GAS::gas_temperature**

Given: Gas temperature (K)

**4.24.2.5   double MIXED_GAS::velocity**

Given: Gas phase velocity (cm/s)

**4.24.2.6   double MIXED_GAS::char_length**

Given: Characteristic Length (cm)

**4.24.2.7   std::vector<double> MIXED_GAS::molefraction**

Given: Gas molefractions of each species (-)

**4.24.2.8   double MIXED_GAS::total_density**

Calculated: Total gas density (g/cm$^3$) {use RE3}.

**4.24.2.9   double MIXED_GAS::total_dyn_vis**

Calculated: Total dynamic viscosity (g/cm/s)

**4.24.2.10   double MIXED_GAS::kinematic_viscosity**

Calculated: Kinematic viscosity (cm$^2$/s)

**4.24.2.11   double MIXED_GAS::total_molecular_weight**

Calculated: Total molecular weight (g/mol)

**4.24.2.12   double MIXED_GAS::total_specific_heat**

Calculated: Total specific heat (J/g/K)

**4.24.2.13   double MIXED_GAS::Reynolds**

Calculated: Value of the Reynold's number (-)

**4.24.2.14   Matrix<double> MIXED_GAS::binary_diffusion**

Calculated: Tensor matrix of binary gas diffusivities (cm$^2$/s)

**4.24.2.15 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

## 4.25 Molecule Class Reference

C++ Molecule Object built from Atom Objects (click Molecule to go to function definitions)

```
#include <mola.h>
```

Inheritance diagram for Molecule:

```
┌──────────────────┐
│     Molecule     │
└──────────────────┘
          ▲
          ┊
┌──────────────────┐
│ MasterSpeciesList│
└──────────────────┘
```
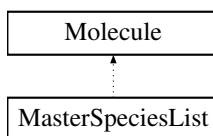
**Public Member Functions**

- 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 recalculateMolarWeight ()

    *Forces molecule to recalculate its molar weight.*

- void setMolarWeigth (double mw)

    *Set the molar weight of 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.*

- 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.*

- 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 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...)*

- 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.*

### 4.25.1 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.

### 4.25.2 Constructor & Destructor Documentation

#### 4.25.2.1 Molecule::Molecule ( )

Default Constructor (builds an empty molecule object)

#### 4.25.2.2 Molecule::∼Molecule ( )

Default Destructor (clears out memory)

#### 4.25.2.3 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) |

### 4.25.3 Member Function Documentation

**4.25.3.1 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) |

**4.25.3.2 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.

**4.25.3.3 void Molecule::setFormula ( std::string *form* )**

Sets the formula for a molecule.

**4.25.3.4 void Molecule::recalculateMolarWeight ( )**

Forces molecule to recalculate its molar weight.

**4.25.3.5 void Molecule::setMolarWeigth ( double *mw* )**

Set the molar weight of species to a constant.

**4.25.3.6 void Molecule::editCharge ( int *c* )**

Change the ionic charge of a molecule.

**4.25.3.7    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.

**4.25.3.8    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.

**4.25.3.9    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.

**4.25.3.10    void Molecule::editEnthalpy ( double *enthalpy* )**

Edit the molecules formation enthalpy (J/mol)

**4.25.3.11    void Molecule::editEntropy ( double *entropy* )**

Edit the molecules formation entropy (J/K/mol)

**4.25.3.12    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) |

**4.25.3.13    void Molecule::editEnergy ( double *energy* )**

Edit Gibb's formation energy.

**4.25.3.14    void Molecule::removeOneAtom ( std::string *Symbol* )**

Removes one atom of the symbol given (always the first atom found)

**4.25.3.15    void Molecule::removeAllAtoms ( std::string *Symbol* )**

Removes all atoms of the symbol given.

**4.25.3.16 int Molecule::Charge ( )**

Return the charge of the molecule.

**4.25.3.17 double Molecule::MolarWeight ( )**

Return the molar weight of the molecule.

**4.25.3.18 bool Molecule::HaveHS ( )**

Returns true if enthalpy and entropy are known.

**4.25.3.19 bool Molecule::HaveEnergy ( )**

Returns true if the Gibb's energy is known.

**4.25.3.20 bool Molecule::isRegistered ( )**

Returns true if the molecule has been registered.

**4.25.3.21 double Molecule::Enthalpy ( )**

Return the formation enthalpy of the molecule.

**4.25.3.22 double Molecule::Entropy ( )**

Return the formation entropy of the molecule.

**4.25.3.23 double Molecule::Energy ( )**

Return the Gibb's formation energy of the molecule.

**4.25.3.24 std::string Molecule::MoleculeName ( )**

Return the common name of the molecule.

**4.25.3.25 std::string Molecule::MolecularFormula ( )**

Return the molecular formula of the molecule.

**4.25.3.26 std::string Molecule::MoleculePhase ( )**

Return the phase of the molecule.

**4.25.3.27 void Molecule::DisplayInfo ( )**

Function to display molecule information.

### 4.25.4 Member Data Documentation

#### 4.25.4.1 int Molecule::charge `[protected]`

Ionic charge of the molecule - specified.

#### 4.25.4.2 double Molecule::molar_weight `[protected]`

Molar weight of the molecule (g/mol) - determined from atoms or specified.

#### 4.25.4.3 double Molecule::formation_enthalpy `[protected]`

Enthalpy of formation of the molecule (J/mol) - constant.

#### 4.25.4.4 double Molecule::formation_entropy `[protected]`

Entropy of formation of the molecule (J/K/mol) - constant.

#### 4.25.4.5 double Molecule::formation_energy `[protected]`

Gibb's energy of formation (J/mol) - given.

#### 4.25.4.6 std::string Molecule::Phase `[protected]`

Phase of the molecule (i.e. Solid, Liquid, Aqueous, Gas...)

#### 4.25.4.7 std::vector<**Atom**> Molecule::atoms `[protected]`

Atoms which make up the molecule - based on Formula.

#### 4.25.4.8 std::string Molecule::Name `[private]`

Name of the Molecule - Common Name (i.e. H2O = Water)

#### 4.25.4.9 std::string Molecule::Formula `[private]`

Formula for the molecule - specified (i.e. H2O)

#### 4.25.4.10 bool Molecule::haveG `[private]`

True = given Gibb's energy of formation.

#### 4.25.4.11 bool Molecule::haveHS `[private]`

True = give enthalpy and entropy of formation.

**4.25.4.12   bool Molecule::registered**  `[private]`

True = the object was registered.

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

- mola.h

## 4.26   MONKFISH_DATA Struct Reference

```
#include <monkfish.h>
```

**Public Attributes**

- unsigned long int total_steps = 0
- double time_old = 0.0
- double time = 0.0
- bool Print2File = true
- bool Print2Console = true
- bool DirichletBC = true
- bool NonLinear = false
- bool haveMinMax = false
- bool MultiScale = true
- int level = 2
- double t_counter = 0.0
- double t_print
- int NumComp
- double end_time
- double total_sorption_old
- double total_sorption
- double single_fiber_density
- double avg_fiber_density
- double max_fiber_density
- double min_fiber_density
- double max_porosity
- double min_porosity
- double domain_diameter
- FILE ∗ Output
- double(∗ eval_eps )(int i, int l, const void ∗user_data)
- double(∗ eval_rho )(int i, int l, const void ∗user_data)
- double(∗ eval_Dex )(int i, int l, const void ∗user_data)
- double(∗ eval_ads )(int i, int l, const void ∗user_data)
- double(∗ eval_Ret )(int i, int l, const void ∗user_data)
- double(∗ eval_Cex )(int i, const void ∗user_data)
- double(∗ eval_kf )(int i, const void ∗user_data)
- const void ∗ user_data
- std::vector< FINCH_DATA > finch_dat
- std::vector< MONKFISH_PARAM > param_dat
- std::vector< DOGFISH_DATA > dog_dat

## 4.26.1 Member Data Documentation

**4.26.1.1 unsigned long int MONKFISH_DATA::total_steps = 0**

**4.26.1.2 double MONKFISH_DATA::time_old = 0.0**

**4.26.1.3 double MONKFISH_DATA::time = 0.0**

**4.26.1.4 bool MONKFISH_DATA::Print2File = true**

**4.26.1.5 bool MONKFISH_DATA::Print2Console = true**

**4.26.1.6 bool MONKFISH_DATA::DirichletBC = true**

**4.26.1.7 bool MONKFISH_DATA::NonLinear = false**

**4.26.1.8 bool MONKFISH_DATA::haveMinMax = false**

**4.26.1.9 bool MONKFISH_DATA::MultiScale = true**

**4.26.1.10 int MONKFISH_DATA::level = 2**

**4.26.1.11 double MONKFISH_DATA::t_counter = 0.0**

**4.26.1.12 double MONKFISH_DATA::t_print**

**4.26.1.13 int MONKFISH_DATA::NumComp**

**4.26.1.14 double MONKFISH_DATA::end_time**

**4.26.1.15 double MONKFISH_DATA::total_sorption_old**

**4.26.1.16 double MONKFISH_DATA::total_sorption**

**4.26.1.17 double MONKFISH_DATA::single_fiber_density**

**4.26.1.18 double MONKFISH_DATA::avg_fiber_density**

**4.26.1.19 double MONKFISH_DATA::max_fiber_density**

**4.26.1.20 double MONKFISH_DATA::min_fiber_density**

**4.26.1.21 double MONKFISH_DATA::max_porosity**

**4.26.1.22 double MONKFISH_DATA::min_porosity**

**4.26.1.23 double MONKFISH_DATA::domain_diameter**

**4.26.1.24 FILE∗ MONKFISH_DATA::Output**

**4.26.1.25 double(∗ MONKFISH_DATA::eval_eps)(int i, int l, const void ∗user_data)**

**4.26.1.26 double(∗ MONKFISH_DATA::eval_rho)(int i, int l, const void ∗user_data)**

**4.26.1.27 double(∗ MONKFISH_DATA::eval_Dex)(int i, int l, const void ∗user_data)**

**4.26.1.28** **double(∗ MONKFISH DATA::eval ads)(int i, int l, const void ∗user_data)**

**4.26.1.29** **double(∗ MONKFISH DATA::eval Ret)(int i, int l, const void ∗user_data)**

**4.26.1.30** **double(∗ MONKFISH DATA::eval Cex)(int i, const void ∗user_data)**

**4.26.1.31** **double(∗ MONKFISH DATA::eval kf)(int i, const void ∗user_data)**

**4.26.1.32** **const void∗ MONKFISH DATA::user data**

**4.26.1.33** **std::vector<FINCH_DATA> MONKFISH DATA::finch dat**

**4.26.1.34** **std::vector<MONKFISH_PARAM> MONKFISH DATA::param dat**

**4.26.1.35** **std::vector<DOGFISH_DATA> MONKFISH DATA::dog dat**

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

- monkfish.h

## 4.27 MONKFISH PARAM Struct Reference

```
#include <monkfish.h>
```

**Public Attributes**

- double interparticle_diffusion
- double exterior_concentration
- double exterior_transfer_coeff
- double sorbed_molefraction
- double initial_sorption
- double sorption_bc
- double intraparticle_diffusion
- double film_transfer_coeff
- Matrix< double > avg_sorption
- Matrix< double > avg_sorption_old
- Molecule species

### 4.27.1 Member Data Documentation

**4.27.1.1** **double MONKFISH PARAM::interparticle diffusion**

**4.27.1.2** **double MONKFISH PARAM::exterior concentration**

**4.27.1.3** **double MONKFISH PARAM::exterior transfer coeff**

**4.27.1.4** **double MONKFISH PARAM::sorbed molefraction**

**4.27.1.5** **double MONKFISH PARAM::initial sorption**

**4.27.1.6** **double MONKFISH PARAM::sorption bc**

**4.27.1.7** **double MONKFISH PARAM::intraparticle diffusion**

**4.27.1.8  double MONKFISH_PARAM::film_transfer_coeff**

**4.27.1.9  Matrix$<$double$>$ MONKFISH_PARAM::avg_sorption**

**4.27.1.10  Matrix$<$double$>$ MONKFISH_PARAM::avg_sorption_old**

**4.27.1.11  Molecule MONKFISH_PARAM::species**

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

- monkfish.h

## 4.28  mSPD_DATA Struct Reference

MSPD Data Structure.

```
#include <magpie.h>
```

**Public Attributes**

- double s

  *Area shape factor.*
- double v

  *van der Waals Volume (cm$^\wedge$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.*

### 4.28.1  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.

### 4.28.2  Member Data Documentation

**4.28.2.1  double mSPD_DATA::s**

Area shape factor.

**4.28.2.2  double mSPD_DATA::v**

van der Waals Volume (cm$^\wedge$3/mol)

**4.28.2.3  double mSPD_DATA::eMax**

Maximum lateral interaction energy (J/mol)

**4.28.2.4 std::vector<double> mSPD_DATA::eta**

Binary interaction parameter matrix (i,j)

**4.28.2.5 double mSPD_DATA::gama**

Activity coefficient calculated from mSPD.

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

- magpie.h

# 4.29 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.*

## 4.29.1 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.

## 4.29.2 Member Data Documentation

**4.29.2.1 double NUM_JAC_DATA::eps = sqrt(DBL_EPSILON)**

Perturbation value.

**4.29.2.2 Matrix<double> NUM_JAC_DATA::Fx**

Vector of function evaluations at x.

**4.29.2.3 Matrix<double> NUM_JAC_DATA::Fxp**

Vector of function evaluations at x+eps.

**4.29.2.4 Matrix<double> NUM_JAC_DATA::dxj**

Vector of perturbed x values.

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

- lark.h

# 4.30 OPTRANS_DATA Struct Reference

Data structure for implementation of linear operator transposition.

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

**Public Attributes**

- Matrix< double > li

    *The ith column vector of the identity operator.*

- Matrix< double > Ai

    *The ith column vector of the user's linear operator.*

## 4.30.1 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 $A^{\wedge}T*r$ 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.

## 4.30.2 Member Data Documentation

**4.30.2.1 Matrix<double> OPTRANS_DATA::li**

The ith column vector of the identity operator.

**4.30.2.2 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

# 4.31 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.*

### 4.31.1 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.

### 4.31.2 Member Data Documentation

#### 4.31.2.1 int PCG_DATA::maxit = 0

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

#### 4.31.2.2 int PCG_DATA::iter = 0

Actual number of iterations taken.

#### 4.31.2.3 double PCG_DATA::alpha

Step size for new solution.

#### 4.31.2.4 double PCG_DATA::beta

Step size for new search direction.

#### 4.31.2.5 double PCG_DATA::tol_rel = 1e-6

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

#### 4.31.2.6 double PCG_DATA::tol_abs = 1e-6

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

#### 4.31.2.7 double PCG_DATA::res

Absolute residual norm.

#### 4.31.2.8 double PCG_DATA::relres

Relative residual norm.

#### 4.31.2.9 double PCG_DATA::relres_base

Initial residual norm.

#### 4.31.2.10 double PCG_DATA::bestres

Best found residual norm.

#### 4.31.2.11 bool PCG_DATA::Output = true

True = print messages to console.

#### 4.31.2.12 Matrix<double> PCG_DATA::x

Current solution to the linear system.

**4.31.2.13** **Matrix$<$double$>$ PCG DATA::bestx**

Best found solution to the linear system.

**4.31.2.14** **Matrix$<$double$>$ PCG DATA::r**

Residual vector for the linear system.

**4.31.2.15** **Matrix$<$double$>$ PCG DATA::r old**

Previous residual vector.

**4.31.2.16** **Matrix$<$double$>$ PCG DATA::z**

Preconditioned residual vector (result of precon function)

**4.31.2.17** **Matrix$<$double$>$ PCG DATA::z old**

Previous preconditioned residual vector.

**4.31.2.18** **Matrix$<$double$>$ PCG DATA::p**

Search direction.

**4.31.2.19** **Matrix$<$double$>$ PCG DATA::Ap**

Result of matrix-vector multiplication.

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

- lark.h

## 4.32 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.*

### 4.32.1 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.

### 4.32.2 Constructor & Destructor Documentation

#### 4.32.2.1 PeriodicTable::PeriodicTable ( )

Default Constructor - Build Perodic Table.

#### 4.32.2.2 PeriodicTable::∼PeriodicTable ( )

Default Destructor - Destroy the table.

#### 4.32.2.3 PeriodicTable::PeriodicTable ( int ∗ *n,* int *N* )

Construct a partial table from a list of atomic numbers.

#### 4.32.2.4 PeriodicTable::PeriodicTable ( std::vector< std::string > & *Symbol* )

Construct a partial table from a vector of atom symbols.

#### 4.32.2.5 PeriodicTable::PeriodicTable ( std::vector< int > & *n* )

Construct a partial table from a vector of atomic numbers.

### 4.32.3 Member Function Documentation

#### 4.32.3.1 void PeriodicTable::DisplayTable ( )

Displays the periodic table via symbols.

### 4.32.4 Member Data Documentation

#### 4.32.4.1 std::vector<Atom> PeriodicTable::Table `[protected]`

Storage vector for all atoms in the table.

**4.32.4.2 int PeriodicTable::number_elements** `[private]`

Number of atom objects being stored.

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

- eel.h

## 4.33 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.*

### 4.33.1 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.

### 4.33.2 Member Data Documentation

#### 4.33.2.1 int PICARD_DATA::maxit = 0

Maximum allowable iterations - default = min(3∗vec_size,1000)

#### 4.33.2.2 int PICARD_DATA::iter = 0

Actual number of iterations.

#### 4.33.2.3 double PICARD_DATA::tol_rel = 1e-6

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

#### 4.33.2.4 double PICARD_DATA::tol_abs = 1e-6

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

#### 4.33.2.5 double PICARD_DATA::res

Residual norm of the iterate.

#### 4.33.2.6 double PICARD_DATA::relres

Relative residual norm of the iterate.

#### 4.33.2.7 double PICARD_DATA::relres_base

Initial residual norm.

#### 4.33.2.8 double PICARD_DATA::bestres

Best found residual norm.

#### 4.33.2.9 bool PICARD_DATA::Output = true

True = print messages to console.

#### 4.33.2.10 Matrix<double> PICARD_DATA::x0

Previous iterate solution vector.

#### 4.33.2.11 Matrix<double> PICARD_DATA::bestx

Best found solution vector.

**4.33.2.12   Matrix⟨double⟩ PICARD_DATA::r**

Residual of the non-linear system.

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

- lark.h

## 4.34   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 nl_maxit = 0

    *Maximum allowable non-linear steps.*
- 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.*
- 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.*

- 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.*

### 4.34.1 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.

### 4.34.2 Member Data Documentation

#### 4.34.2.1 int PJFNK_DATA::nl_iter = 0

Number of non-linear iterations.

**4.34.2.2   int PJFNK_DATA::l_iter = 0**

Number of linear iterations.

**4.34.2.3   int PJFNK_DATA::nl_maxit = 0**

Maximum allowable non-linear steps.

**4.34.2.4   int PJFNK_DATA::linear_solver = -1**

Flag to denote which linear solver to use - default = PJFNK Chooses.

**4.34.2.5   double PJFNK_DATA::nl_tol_abs = 1e-6**

Absolute Convergence tolerance for non-linear system - default = 1e-6.

**4.34.2.6   double PJFNK_DATA::nl_tol_rel = 1e-6**

Relative Convergence tol for the non-linear system - default = 1e-6.

**4.34.2.7   double PJFNK_DATA::lin_tol_rel = 1e-6**

Relative tolerance of the linear solver - default = 1e-6.

**4.34.2.8   double PJFNK_DATA::lin_tol_abs = 1e-6**

Absolute tolerance of the linear solver - default = 1e-6.

**4.34.2.9   double PJFNK_DATA::nl_res**

Absolute redidual norm for the non-linear system.

**4.34.2.10   double PJFNK_DATA::nl_relres**

Relative residual for the non-linear system.

**4.34.2.11   double PJFNK_DATA::nl_res_base**

Initial residual norm for the non-linear system.

**4.34.2.12   double PJFNK_DATA::nl_bestres**

Best found residual norm.

**4.34.2.13   double PJFNK_DATA::eps =sqrt(DBL_EPSILON)**

Value of epsilon used jacvec - default = sqrt(DBL_EPSILON)

**4.34.2.14 bool PJFNK_DATA::NL_Output = true**

True = print PJFNK messages to console.

**4.34.2.15 bool PJFNK_DATA::L_Output = false**

True = print Linear messages to console.

**4.34.2.16 bool PJFNK_DATA::LineSearch = false**

True = use Backtracking Linesearch for global convergence.

**4.34.2.17 bool PJFNK_DATA::Bounce = false**

True = allow Linesearch to go outside local well, False = Strict local convergence.

**4.34.2.18 Matrix<double> PJFNK_DATA::F**

Stored fuction evaluation at x (also the residual)

**4.34.2.19 Matrix<double> PJFNK_DATA::Fv**

Stored function evaluation at x+eps∗v.

**4.34.2.20 Matrix<double> PJFNK_DATA::v**

Stored vector of x+eps∗v.

**4.34.2.21 Matrix<double> PJFNK_DATA::x**

Current solution vector for the non-linear system.

**4.34.2.22 Matrix<double> PJFNK_DATA::bestx**

Best found solution vector to the non-linear system.

**4.34.2.23 GMRESLP_DATA PJFNK_DATA::gmreslp_dat**

Data structure for the GMRESLP method.

**4.34.2.24 PCG_DATA PJFNK_DATA::pcg_dat**

Data structure for the PCG method.

**4.34.2.25 BiCGSTAB_DATA PJFNK_DATA::bicgstab_dat**

Data structure for the BiCGSTAB method.

**4.34.2.26 CGS_DATA PJFNK_DATA::cgs_dat**

Data structure for the CGS method.

**4.34.2.27 GMRESRP_DATA PJFNK_DATA::gmresrp_dat**

Data structure for the GMRESRP method.

**4.34.2.28 GCR_DATA PJFNK_DATA::gcr_dat**

Data structure for the GCR method.

**4.34.2.29 GMRESR_DATA PJFNK_DATA::gmresr_dat**

Data structure for the GMRESR method.

**4.34.2.30 BACKTRACK_DATA PJFNK_DATA::backtrack_dat**

Data structure for the Backtracking Linesearch algorithm.

**4.34.2.31 const void∗ PJFNK_DATA::res_data**

Data structure pointer for user's residual data.

**4.34.2.32 const void∗ PJFNK_DATA::precon_data**

Data structure pointer for user's preconditioning data.

**4.34.2.33 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.

**4.34.2.34 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

## 4.35 Precipitation Class Reference

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

Inheritance diagram for Precipitation:

```
┌─────────────────────────┐
│        Reaction         │
└─────────────────────────┘
             ▲
             ┊
┌─────────────────────────┐
│      Precipitation      │
└─────────────────────────┘
             ▲
             ┊
┌─────────────────────────┐
│  UnsteadyPrecipitation  │
└─────────────────────────┘
```
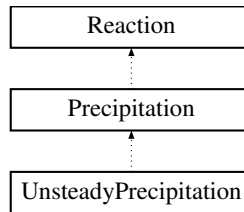
**Additional Inherited Members**

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

- shark.h

## 4.36   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 (-)*

### 4.36.1   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.

### 4.36.2 Member Data Documentation

#### 4.36.2.1 double PURE_GAS::molecular_weight

Given: molecular weights (g/mol)

#### 4.36.2.2 double PURE_GAS::Sutherland_Temp

Given: Sutherland's Reference Temperature (K)

#### 4.36.2.3 double PURE_GAS::Sutherland_Const

Given: Sutherland's Constant (K)

#### 4.36.2.4 double PURE_GAS::Sutherland_Viscosity

Given: Sutherland's Reference Viscosity (g/cm/s)

#### 4.36.2.5 double PURE_GAS::specific_heat

Given: Specific heat of the gas (J/g/K)

#### 4.36.2.6 double PURE_GAS::molecular_diffusion

Calculated: molecular diffusivities (cm$^2$/s)

#### 4.36.2.7 double PURE_GAS::dynamic_viscosity

Calculated: dynamic viscosities (g/cm/s)

#### 4.36.2.8 double PURE_GAS::density

Calculated: gas densities (g/cm$^3$) {use RE3}.

#### 4.36.2.9 double PURE_GAS::Schmidt

Calculated: Value of the Schmidt number (-)

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

- egret.h

## 4.37 Reaction Class Reference

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

Inheritance diagram for Reaction:

```
                          ┌─────────────────┐
                          │    Reaction     │
                          └─────────────────┘
                                   ▲
                    ┌──────────────┴──────────────┐
          ┌─────────────────┐          ┌─────────────────┐
          │  Precipitation  │          │ UnsteadyReaction│
          └─────────────────┘          └─────────────────┘
                   ▲
          ┌──────────────────────┐
          │ UnsteadyPrecipitation│
          └──────────────────────┘
```
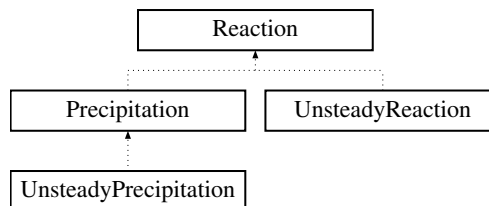
**Public Member Functions**

- Reaction ()
- ∼Reaction ()
- void Initialize_List (MasterSpeciesList &List)
- void Display_Info ()
- void Set_Stoichiometric (int i, double v)
- void Set_Equilibrium (double v)
- void Set_Enthalpy (double H)
- void Set_Entropy (double S)
- void Set_EnthalpyANDEntropy (double H, double S)
- void Set_Energy (double G)
- void checkSpeciesEnergies ()
- void calculateEnergies ()
- void calculateEquilibrium (double T)
- bool haveEquilibrium ()
- double Get_Stoichiometric (int i)
- double Get_Equilibrium ()
- double Get_Enthalpy ()
- double Get_Entropy ()
- double Get_Energy ()
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama)

**Protected Attributes**

- MasterSpeciesList ∗ List
- std::vector< double > Stoichiometric
- double Equilibrium
- double enthalpy
- double entropy
- double energy
- bool CanCalcHS
- bool CanCalcG
- bool HaveHS
- bool HaveG
- bool HaveEquil

### 4.37.1 Constructor & Destructor Documentation

#### 4.37.1.1 Reaction::Reaction ( )

#### 4.37.1.2 Reaction::∼Reaction ( )

### 4.37.2 Member Function Documentation

**4.37.2.1** **void Reaction::Initialize_List ( MasterSpeciesList &** *List* **)**

**4.37.2.2** **void Reaction::Display_Info ( )**

**4.37.2.3** **void Reaction::Set_Stoichiometric ( int** *i,* **double** *v* **)**

**4.37.2.4** **void Reaction::Set_Equilibrium ( double** *v* **)**

**4.37.2.5** **void Reaction::Set_Enthalpy ( double** *H* **)**

**4.37.2.6** **void Reaction::Set_Entropy ( double** *S* **)**

**4.37.2.7** **void Reaction::Set_EnthalpyANDEntropy ( double** *H,* **double** *S* **)**

**4.37.2.8** **void Reaction::Set_Energy ( double** *G* **)**

**4.37.2.9** **void Reaction::checkSpeciesEnergies ( )**

**4.37.2.10** **void Reaction::calculateEnergies ( )**

**4.37.2.11** **void Reaction::calculateEquilibrium ( double** *T* **)**

**4.37.2.12** **bool Reaction::haveEquilibrium ( )**

**4.37.2.13** **double Reaction::Get_Stoichiometric ( int** *i* **)**

**4.37.2.14** **double Reaction::Get_Equilibrium ( )**

**4.37.2.15** **double Reaction::Get_Enthalpy ( )**

**4.37.2.16** **double Reaction::Get_Entropy ( )**

**4.37.2.17** **double Reaction::Get_Energy ( )**

**4.37.2.18** **double Reaction::Eval_Residual ( const Matrix$<$ double $>$ &** *x,* **const Matrix$<$ double $>$ &** *gama* **)**

### 4.37.3 Member Data Documentation

**4.37.3.1** **MasterSpeciesList**∗ **Reaction::List** `[protected]`

**4.37.3.2** **std::vector$<$double$>$ Reaction::Stoichiometric** `[protected]`

**4.37.3.3** **double Reaction::Equilibrium** `[protected]`

**4.37.3.4** **double Reaction::enthalpy** `[protected]`

**4.37.3.5** **double Reaction::entropy** `[protected]`

**4.37.3.6** **double Reaction::energy** `[protected]`

**4.37.3.7** **bool Reaction::CanCalcHS** `[protected]`

**4.37.3.8** **bool Reaction::CanCalcG** `[protected]`

**4.37.3.9** **bool Reaction::HaveHS** `[protected]`

**4.37.3.10    bool Reaction::HaveG**    `[protected]`

**4.37.3.11    bool Reaction::HaveEquil**    `[protected]`

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

- shark.h

## 4.38    SCOPSOWL_DATA Struct Reference

```
#include <scopsowl.h>
```

**Public Attributes**

- unsigned long int total_steps
- int coord_macro
- int coord_micro
- int level = 2
- double sim_time
- double t_old
- double t
- double t_counter = 0.0
- double t_print
- bool Print2File = true
- bool Print2Console = true
- bool SurfDiff = true
- bool Heterogeneous = true
- double gas_velocity
- double total_pressure
- double gas_temperature
- double pellet_radius
- double crystal_radius
- double char_macro
- double char_micro
- double binder_fraction
- double binder_porosity
- double binder_poresize
- double pellet_density
- bool DirichletBC = false
- bool NonLinear = true
- std::vector< double > y
- std::vector< double > tempy
- FILE ∗ OutputFile
- double(∗ eval_ads )(int i, int l, const void ∗user_data)
- double(∗ eval_retard )(int i, int l, const void ∗user_data)
- double(∗ eval_diff )(int i, int l, const void ∗user_data)
- double(∗ eval_surfDiff )(int i, int l, const void ∗user_data)
- double(∗ eval_kf )(int i, const void ∗user_data)
- const void ∗ user_data
- MIXED_GAS ∗ gas_dat
- MAGPIE_DATA magpie_dat
- std::vector< FINCH_DATA > finch_dat
- std::vector< SCOPSOWL_PARAM_DATA > param_dat
- std::vector< SKUA_DATA > skua_dat

### 4.38.1 Member Data Documentation

#### 4.38.1.1 unsigned long int SCOPSOWL_DATA::total_steps

#### 4.38.1.2 int SCOPSOWL_DATA::coord_macro

#### 4.38.1.3 int SCOPSOWL_DATA::coord_micro

#### 4.38.1.4 int SCOPSOWL_DATA::level = 2

#### 4.38.1.5 double SCOPSOWL_DATA::sim_time

#### 4.38.1.6 double SCOPSOWL_DATA::t_old

#### 4.38.1.7 double SCOPSOWL_DATA::t

#### 4.38.1.8 double SCOPSOWL_DATA::t_counter = 0.0

#### 4.38.1.9 double SCOPSOWL_DATA::t_print

#### 4.38.1.10 bool SCOPSOWL_DATA::Print2File = true

#### 4.38.1.11 bool SCOPSOWL_DATA::Print2Console = true

#### 4.38.1.12 bool SCOPSOWL_DATA::SurfDiff = true

#### 4.38.1.13 bool SCOPSOWL_DATA::Heterogeneous = true

#### 4.38.1.14 double SCOPSOWL_DATA::gas_velocity

#### 4.38.1.15 double SCOPSOWL_DATA::total_pressure

#### 4.38.1.16 double SCOPSOWL_DATA::gas_temperature

#### 4.38.1.17 double SCOPSOWL_DATA::pellet_radius

#### 4.38.1.18 double SCOPSOWL_DATA::crystal_radius

#### 4.38.1.19 double SCOPSOWL_DATA::char_macro

#### 4.38.1.20 double SCOPSOWL_DATA::char_micro

#### 4.38.1.21 double SCOPSOWL_DATA::binder_fraction

#### 4.38.1.22 double SCOPSOWL_DATA::binder_porosity

#### 4.38.1.23 double SCOPSOWL_DATA::binder_poresize

#### 4.38.1.24 double SCOPSOWL_DATA::pellet_density

#### 4.38.1.25 bool SCOPSOWL_DATA::DirichletBC = false

#### 4.38.1.26 bool SCOPSOWL_DATA::NonLinear = true

#### 4.38.1.27 std::vector<double> SCOPSOWL_DATA::y

**4.38.1.28   std::vector⟨double⟩ SCOPSOWL_DATA::tempy**

**4.38.1.29   FILE∗ SCOPSOWL_DATA::OutputFile**

**4.38.1.30   double(∗ SCOPSOWL_DATA::eval_ads)(int i, int l, const void ∗user_data)**

**4.38.1.31   double(∗ SCOPSOWL_DATA::eval_retard)(int i, int l, const void ∗user_data)**

**4.38.1.32   double(∗ SCOPSOWL_DATA::eval_diff)(int i, int l, const void ∗user_data)**

**4.38.1.33   double(∗ SCOPSOWL_DATA::eval_surfDiff)(int i, int l, const void ∗user_data)**

**4.38.1.34   double(∗ SCOPSOWL_DATA::eval_kf)(int i, const void ∗user_data)**

**4.38.1.35   const void∗ SCOPSOWL_DATA::user_data**

**4.38.1.36   MIXED_GAS∗ SCOPSOWL_DATA::gas_dat**

**4.38.1.37   MAGPIE_DATA SCOPSOWL_DATA::magpie_dat**

**4.38.1.38   std::vector⟨FINCH_DATA⟩ SCOPSOWL_DATA::finch_dat**

**4.38.1.39   std::vector⟨SCOPSOWL_PARAM_DATA⟩ SCOPSOWL_DATA::param_dat**

**4.38.1.40   std::vector⟨SKUA_DATA⟩ SCOPSOWL_DATA::skua_dat**

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

- scopsowl.h

## 4.39   SCOPSOWL_OPT_DATA Struct Reference

```
#include <scopsowl_opt.h>
```

**Public Attributes**

- int num_curves
- int evaluation
- unsigned long int total_eval
- int current_points
- int num_params = 1
- int diffusion_type
- int adsorb_index
- int max_guess_iter = 20
- bool Optimize
- bool Rough
- double current_temp
- double current_press
- double current_equil
- double simulation_equil
- double max_bias
- double min_bias
- double e_norm
- double f_bias

- double e_norm_old
- double f_bias_old
- double param_guess
- double param_guess_old
- double rel_tol_norm = 0.01
- double abs_tol_bias = 1.0
- std::vector< double > y_base
- std::vector< double > q_data
- std::vector< double > q_sim
- std::vector< double > t
- FILE ∗ ParamFile
- FILE ∗ CompareFile
- SCOPSOWL_DATA owl_dat

### 4.39.1 Member Data Documentation

#### 4.39.1.1 int SCOPSOWL_OPT_DATA::num_curves

#### 4.39.1.2 int SCOPSOWL_OPT_DATA::evaluation

#### 4.39.1.3 unsigned long int SCOPSOWL_OPT_DATA::total_eval

#### 4.39.1.4 int SCOPSOWL_OPT_DATA::current_points

#### 4.39.1.5 int SCOPSOWL_OPT_DATA::num_params = 1

#### 4.39.1.6 int SCOPSOWL_OPT_DATA::diffusion_type

#### 4.39.1.7 int SCOPSOWL_OPT_DATA::adsorb_index

#### 4.39.1.8 int SCOPSOWL_OPT_DATA::max_guess_iter = 20

#### 4.39.1.9 bool SCOPSOWL_OPT_DATA::Optimize

#### 4.39.1.10 bool SCOPSOWL_OPT_DATA::Rough

#### 4.39.1.11 double SCOPSOWL_OPT_DATA::current_temp

#### 4.39.1.12 double SCOPSOWL_OPT_DATA::current_press

#### 4.39.1.13 double SCOPSOWL_OPT_DATA::current_equil

#### 4.39.1.14 double SCOPSOWL_OPT_DATA::simulation_equil

#### 4.39.1.15 double SCOPSOWL_OPT_DATA::max_bias

#### 4.39.1.16 double SCOPSOWL_OPT_DATA::min_bias

#### 4.39.1.17 double SCOPSOWL_OPT_DATA::e_norm

#### 4.39.1.18 double SCOPSOWL_OPT_DATA::f_bias

#### 4.39.1.19 double SCOPSOWL_OPT_DATA::e_norm_old

#### 4.39.1.20 double SCOPSOWL_OPT_DATA::f_bias_old

**4.39.1.21 double SCOPSOWL_OPT_DATA::param_guess**

**4.39.1.22 double SCOPSOWL_OPT_DATA::param_guess_old**

**4.39.1.23 double SCOPSOWL_OPT_DATA::rel_tol_norm = 0.01**

**4.39.1.24 double SCOPSOWL_OPT_DATA::abs_tol_bias = 1.0**

**4.39.1.25 std::vector$<$double$>$ SCOPSOWL_OPT_DATA::y_base**

**4.39.1.26 std::vector$<$double$>$ SCOPSOWL_OPT_DATA::q_data**

**4.39.1.27 std::vector$<$double$>$ SCOPSOWL_OPT_DATA::q_sim**

**4.39.1.28 std::vector$<$double$>$ SCOPSOWL_OPT_DATA::t**

**4.39.1.29 FILE∗ SCOPSOWL_OPT_DATA::ParamFile**

**4.39.1.30 FILE∗ SCOPSOWL_OPT_DATA::CompareFile**

**4.39.1.31 SCOPSOWL_DATA SCOPSOWL_OPT_DATA::owl_dat**

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

- scopsowl_opt.h

## 4.40 SCOPSOWL_PARAM_DATA Struct Reference

```
#include <scopsowl.h>
```

**Public Attributes**

- Matrix$<$ double $>$ qAvg
- Matrix$<$ double $>$ qAvg_old
- Matrix$<$ double $>$ Qst
- Matrix$<$ double $>$ Qst_old
- Matrix$<$ double $>$ dq_dc
- double xIC
- double qIntegralAvg
- double qIntegralAvg_old
- double QstAvg
- double QstAvg_old
- double qo
- double Qsto
- double dq_dco
- double pore_diffusion
- double film_transfer
- double activation_energy
- double ref_diffusion
- double ref_temperature
- double affinity
- double ref_pressure
- bool Adsorbable
- std::string speciesName

### 4.40.1 Member Data Documentation

#### 4.40.1.1 Matrix$<$double$>$ SCOPSOWL_PARAM_DATA::qAvg

#### 4.40.1.2 Matrix$<$double$>$ SCOPSOWL_PARAM_DATA::qAvg_old

#### 4.40.1.3 Matrix$<$double$>$ SCOPSOWL_PARAM_DATA::Qst

#### 4.40.1.4 Matrix$<$double$>$ SCOPSOWL_PARAM_DATA::Qst_old

#### 4.40.1.5 Matrix$<$double$>$ SCOPSOWL_PARAM_DATA::dq_dc

#### 4.40.1.6 double SCOPSOWL_PARAM_DATA::xIC

#### 4.40.1.7 double SCOPSOWL_PARAM_DATA::qIntegralAvg

#### 4.40.1.8 double SCOPSOWL_PARAM_DATA::qIntegralAvg_old

#### 4.40.1.9 double SCOPSOWL_PARAM_DATA::QstAvg

#### 4.40.1.10 double SCOPSOWL_PARAM_DATA::QstAvg_old

#### 4.40.1.11 double SCOPSOWL_PARAM_DATA::qo

#### 4.40.1.12 double SCOPSOWL_PARAM_DATA::Qsto

#### 4.40.1.13 double SCOPSOWL_PARAM_DATA::dq_dco

#### 4.40.1.14 double SCOPSOWL_PARAM_DATA::pore_diffusion

#### 4.40.1.15 double SCOPSOWL_PARAM_DATA::film_transfer

#### 4.40.1.16 double SCOPSOWL_PARAM_DATA::activation_energy

#### 4.40.1.17 double SCOPSOWL_PARAM_DATA::ref_diffusion

#### 4.40.1.18 double SCOPSOWL_PARAM_DATA::ref_temperature

#### 4.40.1.19 double SCOPSOWL_PARAM_DATA::affinity

#### 4.40.1.20 double SCOPSOWL_PARAM_DATA::ref_pressure

#### 4.40.1.21 bool SCOPSOWL_PARAM_DATA::Adsorbable

#### 4.40.1.22 std::string SCOPSOWL_PARAM_DATA::speciesName

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

- scopsowl.h

## 4.41 SHARK_DATA Struct Reference

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

**Public Attributes**

- MasterSpeciesList MasterList
- std::vector< Reaction > ReactionList
- std::vector< MassBalance > MassBalanceList
- std::vector< UnsteadyReaction > UnsteadyList
- std::vector< double(∗)(const Matrix< double > &x, SHARK_DATA ∗shark_dat, const void ∗data) > OtherList
- int numvar
- int num_ssr
- int num_mbe
- int num_usr
- int num_other = 0
- int act_fun = IDEAL
- int totalsteps = 0
- int timesteps = 0
- int pH_index = -1
- int pOH_index = -1
- double simulationtime = 0.0
- double dt = 0.1
- double dt_min = sqrt(DBL_EPSILON)
- double t_out = 0.0
- double t_count = 0.0
- double time = 0.0
- double time_old = 0.0
- double pH = 7.0
- double Norm = 0.0
- double dielectric_const = 78.325
- double temperature = 298.15
- bool steadystate = true
- bool TimeAdaptivity = false
- bool const_pH = false
- bool SpeciationCurve = false
- bool Console_Output = true
- bool File_Output = false
- bool Contains_pH = false
- bool Contains_pOH = false
- bool Converged = false
- Matrix< double > X_old
- Matrix< double > X_new
- Matrix< double > Conc_old
- Matrix< double > Conc_new
- Matrix< double > activity_new
- Matrix< double > activity_old
- int(∗ EvalActivity )(const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int(∗ Residual )(const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int(∗ lin_precon )(const Matrix< double > &r, Matrix< double > &p, const void ∗data)
- PJFNK_DATA Newton_data
- const void ∗ activity_data
- const void ∗ residual_data
- const void ∗ precon_data
- const void ∗ other_data
- FILE ∗ OutputFile
- yaml_cpp_class yaml_object

### 4.41.1 Member Data Documentation

#### 4.41.1.1 MasterSpeciesList SHARK_DATA::MasterList

#### 4.41.1.2 std::vector<**Reaction**> SHARK_DATA::ReactionList

#### 4.41.1.3 std::vector<**MassBalance**> SHARK_DATA::MassBalanceList

#### 4.41.1.4 std::vector<**UnsteadyReaction**> SHARK_DATA::UnsteadyList

#### 4.41.1.5 std::vector< double (∗) (const **Matrix**<double> &x, **SHARK_DATA** ∗shark_dat, const void ∗data) > SHARK_DATA::OtherList

#### 4.41.1.6 int SHARK_DATA::numvar

#### 4.41.1.7 int SHARK_DATA::num_ssr

#### 4.41.1.8 int SHARK_DATA::num_mbe

#### 4.41.1.9 int SHARK_DATA::num_usr

#### 4.41.1.10 int SHARK_DATA::num_other = 0

#### 4.41.1.11 int SHARK_DATA::act_fun = IDEAL

#### 4.41.1.12 int SHARK_DATA::totalsteps = 0

#### 4.41.1.13 int SHARK_DATA::timesteps = 0

#### 4.41.1.14 int SHARK_DATA::pH_index = -1

#### 4.41.1.15 int SHARK_DATA::pOH_index = -1

#### 4.41.1.16 double SHARK_DATA::simulationtime = 0.0

#### 4.41.1.17 double SHARK_DATA::dt = 0.1

#### 4.41.1.18 double SHARK_DATA::dt_min = sqrt(DBL_EPSILON)

#### 4.41.1.19 double SHARK_DATA::t_out = 0.0

#### 4.41.1.20 double SHARK_DATA::t_count = 0.0

#### 4.41.1.21 double SHARK_DATA::time = 0.0

#### 4.41.1.22 double SHARK_DATA::time_old = 0.0

#### 4.41.1.23 double SHARK_DATA::pH = 7.0

#### 4.41.1.24 double SHARK_DATA::Norm = 0.0

#### 4.41.1.25 double SHARK_DATA::dielectric_const = 78.325

#### 4.41.1.26 double SHARK_DATA::temperature = 298.15

#### 4.41.1.27 bool SHARK_DATA::steadystate = true

**4.41.1.28 bool SHARK_DATA::TimeAdaptivity = false**

**4.41.1.29 bool SHARK_DATA::const_pH = false**

**4.41.1.30 bool SHARK_DATA::SpeciationCurve = false**

**4.41.1.31 bool SHARK_DATA::Console_Output = true**

**4.41.1.32 bool SHARK_DATA::File_Output = false**

**4.41.1.33 bool SHARK_DATA::Contains_pH = false**

**4.41.1.34 bool SHARK_DATA::Contains_pOH = false**

**4.41.1.35 bool SHARK_DATA::Converged = false**

**4.41.1.36 Matrix$<$double$>$ SHARK_DATA::X_old**

**4.41.1.37 Matrix$<$double$>$ SHARK_DATA::X_new**

**4.41.1.38 Matrix$<$double$>$ SHARK_DATA::Conc_old**

**4.41.1.39 Matrix$<$double$>$ SHARK_DATA::Conc_new**

**4.41.1.40 Matrix$<$double$>$ SHARK_DATA::activity_new**

**4.41.1.41 Matrix$<$double$>$ SHARK_DATA::activity_old**

**4.41.1.42 int($*$ SHARK_DATA::EvalActivity)(const Matrix$<$ double $>$ &x, Matrix$<$ double $>$ &F, const void $*$data)**

**4.41.1.43 int($*$ SHARK_DATA::Residual)(const Matrix$<$ double $>$ &x, Matrix$<$ double $>$ &F, const void $*$data)**

**4.41.1.44 int($*$ SHARK_DATA::lin_precon)(const Matrix$<$ double $>$ &r, Matrix$<$ double $>$ &p, const void $*$data)**

**4.41.1.45 PJFNK_DATA SHARK_DATA::Newton_data**

**4.41.1.46 const void$*$ SHARK_DATA::activity_data**

**4.41.1.47 const void$*$ SHARK_DATA::residual_data**

**4.41.1.48 const void$*$ SHARK_DATA::precon_data**

**4.41.1.49 const void$*$ SHARK_DATA::other_data**

**4.41.1.50 FILE$*$ SHARK_DATA::OutputFile**

**4.41.1.51 yaml_cpp_class SHARK_DATA::yaml_object**

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

- shark.h

## 4.42 SKUA_DATA Struct Reference

```
#include <skua.h>
```

**Public Attributes**

- unsigned long int total_steps
- int coord
- double sim_time
- double t_old
- double t
- double t_counter = 0.0
- double t_print
- double qTn
- double qTnp1
- bool Print2File = true
- bool Print2Console = true
- double gas_velocity
- double pellet_radius
- double char_measure
- bool DirichletBC = true
- bool NonLinear = true
- std::vector< double > y
- FILE ∗ OutputFile
- double(∗ eval_diff )(int i, int l, const void ∗user_data)
- double(∗ eval_kf )(int i, const void ∗user_data)
- const void ∗ user_data
- MAGPIE_DATA magpie_dat
- MIXED_GAS ∗ gas_dat
- std::vector< FINCH_DATA > finch_dat
- std::vector< SKUA_PARAM > param_dat

### 4.42.1 Member Data Documentation

#### 4.42.1.1 unsigned long int SKUA_DATA::total_steps

#### 4.42.1.2 int SKUA_DATA::coord

#### 4.42.1.3 double SKUA_DATA::sim_time

#### 4.42.1.4 double SKUA_DATA::t_old

#### 4.42.1.5 double SKUA_DATA::t

#### 4.42.1.6 double SKUA_DATA::t_counter = 0.0

#### 4.42.1.7 double SKUA_DATA::t_print

#### 4.42.1.8 double SKUA_DATA::qTn

#### 4.42.1.9 double SKUA_DATA::qTnp1

#### 4.42.1.10 bool SKUA_DATA::Print2File = true

#### 4.42.1.11 bool SKUA_DATA::Print2Console = true

#### 4.42.1.12 double SKUA_DATA::gas_velocity

**4.42.1.13 double SKUA_DATA::pellet_radius**

**4.42.1.14 double SKUA_DATA::char_measure**

**4.42.1.15 bool SKUA_DATA::DirichletBC = true**

**4.42.1.16 bool SKUA_DATA::NonLinear = true**

**4.42.1.17 std::vector<double> SKUA_DATA::y**

**4.42.1.18 FILE∗ SKUA_DATA::OutputFile**

**4.42.1.19 double(∗ SKUA_DATA::eval_diff)(int i, int l, const void ∗user_data)**

**4.42.1.20 double(∗ SKUA_DATA::eval_kf)(int i, const void ∗user_data)**

**4.42.1.21 const void∗ SKUA_DATA::user_data**

**4.42.1.22 MAGPIE_DATA SKUA_DATA::magpie_dat**

**4.42.1.23 MIXED_GAS∗ SKUA_DATA::gas_dat**

**4.42.1.24 std::vector<FINCH_DATA> SKUA_DATA::finch_dat**

**4.42.1.25 std::vector<SKUA_PARAM> SKUA_DATA::param_dat**

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

- skua.h

## 4.43 SKUA_OPT_DATA Struct Reference

```
#include <skua_opt.h>
```

**Public Attributes**

- int num_curves
- int evaluation
- unsigned long int total_eval
- int current_points
- int num_params = 1
- int diffusion_type
- int adsorb_index
- int max_guess_iter = 20
- bool Optimize
- bool Rough
- double current_temp
- double current_press
- double current_equil
- double simulation_equil
- double max_bias
- double min_bias
- double e_norm
- double f_bias

- double e_norm_old
- double f_bias_old
- double param_guess
- double param_guess_old
- double rel_tol_norm = 0.1
- double abs_tol_bias = 0.1
- std::vector< double > y_base
- std::vector< double > q_data
- std::vector< double > q_sim
- std::vector< double > t
- FILE ∗ ParamFile
- FILE ∗ CompareFile
- SKUA_DATA skua_dat

### 4.43.1 Member Data Documentation

#### 4.43.1.1 int SKUA_OPT_DATA::num_curves

#### 4.43.1.2 int SKUA_OPT_DATA::evaluation

#### 4.43.1.3 unsigned long int SKUA_OPT_DATA::total_eval

#### 4.43.1.4 int SKUA_OPT_DATA::current_points

#### 4.43.1.5 int SKUA_OPT_DATA::num_params = 1

#### 4.43.1.6 int SKUA_OPT_DATA::diffusion_type

#### 4.43.1.7 int SKUA_OPT_DATA::adsorb_index

#### 4.43.1.8 int SKUA_OPT_DATA::max_guess_iter = 20

#### 4.43.1.9 bool SKUA_OPT_DATA::Optimize

#### 4.43.1.10 bool SKUA_OPT_DATA::Rough

#### 4.43.1.11 double SKUA_OPT_DATA::current_temp

#### 4.43.1.12 double SKUA_OPT_DATA::current_press

#### 4.43.1.13 double SKUA_OPT_DATA::current_equil

#### 4.43.1.14 double SKUA_OPT_DATA::simulation_equil

#### 4.43.1.15 double SKUA_OPT_DATA::max_bias

#### 4.43.1.16 double SKUA_OPT_DATA::min_bias

#### 4.43.1.17 double SKUA_OPT_DATA::e_norm

#### 4.43.1.18 double SKUA_OPT_DATA::f_bias

#### 4.43.1.19 double SKUA_OPT_DATA::e_norm_old

#### 4.43.1.20 double SKUA_OPT_DATA::f_bias_old

**4.43.1.21 double SKUA_OPT_DATA::param_guess**

**4.43.1.22 double SKUA_OPT_DATA::param_guess_old**

**4.43.1.23 double SKUA_OPT_DATA::rel_tol_norm = 0.1**

**4.43.1.24 double SKUA_OPT_DATA::abs_tol_bias = 0.1**

**4.43.1.25 std::vector$<$double$>$ SKUA_OPT_DATA::y_base**

**4.43.1.26 std::vector$<$double$>$ SKUA_OPT_DATA::q_data**

**4.43.1.27 std::vector$<$double$>$ SKUA_OPT_DATA::q_sim**

**4.43.1.28 std::vector$<$double$>$ SKUA_OPT_DATA::t**

**4.43.1.29 FILE$*$ SKUA_OPT_DATA::ParamFile**

**4.43.1.30 FILE$*$ SKUA_OPT_DATA::CompareFile**

**4.43.1.31 SKUA_DATA SKUA_OPT_DATA::skua_dat**

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

- skua_opt.h

## 4.44 SKUA_PARAM Struct Reference

```
#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

### 4.44.1 Member Data Documentation

**4.44.1.1 double SKUA_PARAM::activation_energy**

**4.44.1.2 double SKUA_PARAM::ref_diffusion**

**4.44.1.3    double SKUA_PARAM::ref_temperature**

**4.44.1.4    double SKUA_PARAM::affinity**

**4.44.1.5    double SKUA_PARAM::ref_pressure**

**4.44.1.6    double SKUA_PARAM::film_transfer**

**4.44.1.7    double SKUA_PARAM::xIC**

**4.44.1.8    double SKUA_PARAM::y_eff**

**4.44.1.9    double SKUA_PARAM::Qstn**

**4.44.1.10    double SKUA_PARAM::Qstnp1**

**4.44.1.11    double SKUA_PARAM::xn**

**4.44.1.12    double SKUA_PARAM::xnp1**

**4.44.1.13    bool SKUA_PARAM::Adsorbable**

**4.44.1.14    std::string SKUA_PARAM::speciesName**

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

- skua.h

## 4.45    Speciation_Test01_Data Struct Reference

```
#include <sandbox.h>
```

**Public Attributes**

- int N = 4
- const double logKw = -14.0
- const double logKa1 = -6.35
- const double logKa2 = -10.33
- double CT = 0.1786
- double NaT = 0.1786
- std::vector< Molecule > x
- Matrix< double > Jacobian
- Matrix< double > NumJac
- Matrix< double > logC
- Matrix< double > C

### 4.45.1    Member Data Documentation

**4.45.1.1    int Speciation_Test01_Data::N = 4**

**4.45.1.2    const double Speciation_Test01_Data::logKw = -14.0**

**4.45.1.3    const double Speciation_Test01_Data::logKa1 = -6.35**

**4.45.1.4    const double Speciation_Test01_Data::logKa2 = -10.33**

**4.45.1.5    double Speciation_Test01_Data::CT = 0.1786**

**4.45.1.6    double Speciation_Test01_Data::NaT = 0.1786**

**4.45.1.7    std::vector<Molecule> Speciation_Test01_Data::x**

**4.45.1.8    Matrix<double> Speciation_Test01_Data::Jacobian**

**4.45.1.9    Matrix<double> Speciation_Test01_Data::NumJac**

**4.45.1.10    Matrix<double> Speciation_Test01_Data::logC**

**4.45.1.11    Matrix<double> Speciation_Test01_Data::C**

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

- sandbox.h

## 4.46    SubHeader Class Reference

```
#include <yaml_wrapper.h>
```

Inheritance diagram for SubHeader:

```
         SubHeader
        /         \
  Document        Header
```
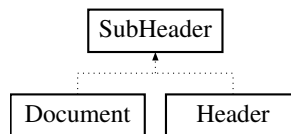
**Public Member Functions**

- SubHeader ()
- ∼SubHeader ()
- SubHeader (const SubHeader &subheader)
- SubHeader (const KeyValueMap &map)
- SubHeader (std::string name)
- SubHeader (std::string name, const KeyValueMap &map)
- SubHeader & operator= (const SubHeader &sub)
- ValueTypePair & operator[] (const std::string key)
- ValueTypePair operator[] (const std::string key) const
- KeyValueMap & getMap ()
- void clear ()
- void addPair (std::string key, std::string val)
- void addPair (std::string key, std::string val, int type)
- void setName (std::string name)
- void setAlias (std::string alias)
- void setAlias (std::string alias, int state)
- void setNameAliasPair (std::string name, std::string alias, int state)
- void setState (int state)
- void DisplayContents ()
- std::string getName ()

- std::string getAlias ()
- bool isAlias ()
- bool isAnchor ()
- int getState ()

**Protected Attributes**

- KeyValueMap Data_Map
- std::string name
- std::string alias
- int state

### 4.46.1 Constructor & Destructor Documentation

#### 4.46.1.1 SubHeader::SubHeader ( )

#### 4.46.1.2 SubHeader::∼SubHeader ( )

#### 4.46.1.3 SubHeader::SubHeader ( const **SubHeader** & *subheader* )

#### 4.46.1.4 SubHeader::SubHeader ( const **KeyValueMap** & *map* )

#### 4.46.1.5 SubHeader::SubHeader ( std::string *name* )

#### 4.46.1.6 SubHeader::SubHeader ( std::string *name,* const **KeyValueMap** & *map* )

### 4.46.2 Member Function Documentation

#### 4.46.2.1 **SubHeader& SubHeader::operator= (** const **SubHeader** & *sub* )

#### 4.46.2.2 **ValueTypePair& SubHeader::operator[] (** const std::string *key* )

#### 4.46.2.3 **ValueTypePair SubHeader::operator[] (** const std::string *key* ) const

#### 4.46.2.4 **KeyValueMap& SubHeader::getMap ( )**

#### 4.46.2.5 **void SubHeader::clear ( )**

#### 4.46.2.6 **void SubHeader::addPair (** std::string *key,* std::string *val* )

#### 4.46.2.7 **void SubHeader::addPair (** std::string *key,* std::string *val,* int *type* )

#### 4.46.2.8 **void SubHeader::setName (** std::string *name* )

#### 4.46.2.9 **void SubHeader::setAlias (** std::string *alias* )

#### 4.46.2.10 **void SubHeader::setAlias (** std::string *alias,* int *state* )

#### 4.46.2.11 **void SubHeader::setNameAliasPair (** std::string *name,* std::string *alias,* int *state* )

#### 4.46.2.12 **void SubHeader::setState (** int *state* )

#### 4.46.2.13 **void SubHeader::DisplayContents ( )**

**4.46.2.14   std::string SubHeader::getName ( )**

**4.46.2.15   std::string SubHeader::getAlias ( )**

**4.46.2.16   bool SubHeader::isAlias ( )**

**4.46.2.17   bool SubHeader::isAnchor ( )**

**4.46.2.18   int SubHeader::getState ( )**

### 4.46.3   Member Data Documentation

**4.46.3.1   KeyValueMap SubHeader::Data_Map** `[protected]`

**4.46.3.2   std::string SubHeader::name** `[protected]`

**4.46.3.3   std::string SubHeader::alias** `[protected]`

**4.46.3.4   int SubHeader::state** `[protected]`

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

- yaml_wrapper.h

## 4.47   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.*

### 4.47.1 Detailed Description

System Data Structure.

C-style object holding all the data associated with the overall system to be modeled.

### 4.47.2 Member Data Documentation

#### 4.47.2.1 double SYSTEM_DATA::T

System Temperature (K)

#### 4.47.2.2 double SYSTEM_DATA::PT

Total Pressure (kPa)

#### 4.47.2.3 double SYSTEM_DATA::qT

Total Amount adsorbed (mol/kg)

#### 4.47.2.4 double SYSTEM_DATA::PI

Total Lumped Spreading Pressure (mol/kg)

#### 4.47.2.5 double SYSTEM_DATA::pi

Actual Spreading pressure (J/m$^2$)

#### 4.47.2.6 double SYSTEM_DATA::As

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

**4.47.2.7   int SYSTEM␣DATA::N**

Total Number of Components.

**4.47.2.8   int SYSTEM␣DATA::I**

**4.47.2.9   int SYSTEM␣DATA::J**

**4.47.2.10   int SYSTEM␣DATA::K**

Special indices used to keep track of sub-systems.

**4.47.2.11   unsigned long int SYSTEM␣DATA::total␣eval**

Counter to keep track of total number of non-linear steps.

**4.47.2.12   double SYSTEM␣DATA::avg␣norm**

Used to store all norms from evaluations then average at end of run.

**4.47.2.13   double SYSTEM␣DATA::max␣norm**

Used to store the maximum e.norm calculated from non-linear iterations.

**4.47.2.14   int SYSTEM␣DATA::Sys**

Number of sub-systems to solve.

**4.47.2.15   int SYSTEM␣DATA::Par**

Number of binary parameters to solve for.

**4.47.2.16   bool SYSTEM␣DATA::Recover**

If Recover == false, standard GPAST using y's as knowns.

**4.47.2.17   bool SYSTEM␣DATA::Carrier**

If there is an inert carrier gas, Carrier == true.

**4.47.2.18   bool SYSTEM␣DATA::Ideal**

If the behavior of the system is determined to be ideal, then Ideal == true.

**4.47.2.19   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

## 4.48 TRAJECTORY_DATA Struct Reference

```
#include <Trajectory.h>
```

**Public Attributes**

- double mu_0 = 12.57e-7
- double rho_f = 1000.0
- double eta = 0.001
- double Hamaker = 1.3e-21
- double Temp = 298
- double k = 1.38e-23
- double Rs = 0.0026925
- double L = 0.0611
- double porosity = 0.8979
- double V_separator
- double a = 33.0e-6
- double V_wire
- double L_wire
- double A_separator
- double A_wire
- double B0 = 1.0
- double H0
- double Ms = 0.6
- double b = 0.25e-6
- double chi_p = 3.87e-6
- double rho_p = 8700.0
- 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 > X
- Matrix< double > Y
- Matrix< int > Cap

### 4.48.1 Member Data Documentation

**4.48.1.1 double TRAJECTORY_DATA::mu_0 = 12.57e-7**

**4.48.1.2 double TRAJECTORY_DATA::rho_f = 1000.0**

**4.48.1.3 double TRAJECTORY_DATA::eta = 0.001**

**4.48.1.4 double TRAJECTORY_DATA::Hamaker = 1.3e-21**

**4.48.1.5 double TRAJECTORY_DATA::Temp = 298**

**4.48.1.6 double TRAJECTORY_DATA::k = 1.38e-23**

**4.48.1.7 double TRAJECTORY_DATA::Rs = 0.0026925**

**4.48.1.8 double TRAJECTORY_DATA::L = 0.0611**

**4.48.1.9 double TRAJECTORY_DATA::porosity = 0.8979**

**4.48.1.10 double TRAJECTORY_DATA::V_separator**

**4.48.1.11 double TRAJECTORY_DATA::a = 33.0e-6**

**4.48.1.12 double TRAJECTORY_DATA::V_wire**

**4.48.1.13 double TRAJECTORY_DATA::L_wire**

**4.48.1.14 double TRAJECTORY_DATA::A_separator**

**4.48.1.15 double TRAJECTORY_DATA::A_wire**

**4.48.1.16 double TRAJECTORY_DATA::B0 = 1.0**

**4.48.1.17 double TRAJECTORY_DATA::H0**

**4.48.1.18 double TRAJECTORY_DATA::Ms = 0.6**

**4.48.1.19 double TRAJECTORY_DATA::b = 0.25e-6**

**4.48.1.20 double TRAJECTORY_DATA::chi_p = 3.87e-6**

**4.48.1.21 double TRAJECTORY_DATA::rho_p = 8700.0**

**4.48.1.22 double TRAJECTORY_DATA::Q_in**

**4.48.1.23 double TRAJECTORY_DATA::V0**

**4.48.1.24 double TRAJECTORY_DATA::Y_initial = 20.0**

**4.48.1.25 double TRAJECTORY_DATA::dt**

**4.48.1.26 double TRAJECTORY_DATA::M**

**4.48.1.27 double TRAJECTORY_DATA::mp**

**4.48.1.28 double TRAJECTORY_DATA::beta**

**4.48.1.29 double TRAJECTORY_DATA::q_bar**

**4.48.1.30 double TRAJECTORY_DATA::sigma_v**

**4.48.1.31 double TRAJECTORY_DATA::sigma_vz**

**4.48.1.32 double TRAJECTORY_DATA::sigma_z**

**4.48.1.33 double TRAJECTORY_DATA::sigma_n**

**4.48.1.34 double TRAJECTORY_DATA::sigma_m**

**4.48.1.35 double TRAJECTORY_DATA::n_rand**

**4.48.1.36 double TRAJECTORY_DATA::m_rand**

**4.48.1.37 double TRAJECTORY_DATA::s_rand**

**4.48.1.38 double TRAJECTORY_DATA::t_rand**

**4.48.1.39 Matrix<double> TRAJECTORY_DATA::POL**

**4.48.1.40 Matrix<double> TRAJECTORY_DATA::H**

**4.48.1.41 Matrix<double> TRAJECTORY_DATA::dX**

**4.48.1.42 Matrix<double> TRAJECTORY_DATA::dY**

**4.48.1.43 Matrix<double> TRAJECTORY_DATA::X**

**4.48.1.44 Matrix<double> TRAJECTORY_DATA::Y**

**4.48.1.45 Matrix<int> TRAJECTORY_DATA::Cap**

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

- Trajectory.h

## 4.49 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.*
- const char ∗ argv [ ]

    *Actual console arguments given at execution.*

### 4.49.1 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.

### 4.49.2 Member Data Documentation

#### 4.49.2.1 ValueTypePair UI_DATA::value_type

Data pair for input, tells what the input is and it's type.

#### 4.49.2.2 std::vector<std::string> UI_DATA::user_input

What is read in from the console at any point.

#### 4.49.2.3 std::vector<std::string> UI_DATA::input_files

A vector of input file names and directories given by user.

#### 4.49.2.4 std::string UI_DATA::path

Path to where input files are located.

#### 4.49.2.5 int UI_DATA::count = 0

Number of times a questing has been asked.

**4.49.2.6    int UI_DATA::max = 3**

Maximum allowable recursions of a question.

**4.49.2.7    int UI_DATA::option**

Current option choosen by the user.

**4.49.2.8    bool UI_DATA::Path = false**

True if user gives path as an option.

**4.49.2.9    bool UI_DATA::Files = false**

True if user gives input files as an option.

**4.49.2.10    bool UI_DATA::MissingArg = true**

True if an input argument is missing; False if everything is ok.

**4.49.2.11    bool UI_DATA::BasicUI = true**

True if using Basic UI; False if using Advanced UI.

**4.49.2.12    int UI_DATA::argc**

Number of console arguments given on input.

**4.49.2.13    const char∗ UI_DATA::argv[ ]**

Actual console arguments given at execution.

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

- ui.h

## 4.50    UnsteadyPrecipitation Class Reference

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

Inheritance diagram for UnsteadyPrecipitation:
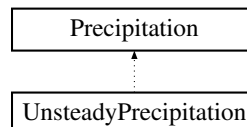


The documentation for this class was generated from the following file:

- shark.h

## 4.51 UnsteadyReaction Class Reference

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

Inheritance diagram for UnsteadyReaction:

```
┌─────────────────────┐
│      Reaction       │
└─────────────────────┘
           ▲
           ┊
┌─────────────────────┐
│  UnsteadyReaction   │
└─────────────────────┘
```

**Public Member Functions**

- UnsteadyReaction ()
- ~UnsteadyReaction ()
- void Initialize_List (MasterSpeciesList &List)
- void Display_Info ()
- void Set_Species_Index (int i)
- void Set_Species_Index (std::string formula)
- void Set_Stoichiometric (int i, double v)
- void Set_Equilibrium (double v)
- void Set_Enthalpy (double H)
- void Set_Entropy (double S)
- void Set_EnthalpyANDEntropy (double H, double S)
- void Set_Energy (double G)
- void Set_InitialValue (double ic)
- void Set_MaximumValue (double max)
- void Set_Forward (double forward)
- void Set_Reverse (double reverse)
- void Set_ForwardRef (double Fref)
- void Set_ReverseRef (double Rref)
- void Set_ActivationEnergy (double E)
- void Set_Affinity (double b)
- void Set_TimeStep (double dt)
- void checkSpeciesEnergies ()
- void calculateEnergies ()
- void calculateEquilibrium (double T)
- void calculateRate (double T)
- bool haveEquilibrium ()
- bool haveRate ()
- int Get_Species_Index ()
- double Get_Stoichiometric (int i)
- double Get_Equilibrium ()
- double Get_Enthalpy ()
- double Get_Entropy ()
- double Get_Energy ()
- double Get_InitialValue ()
- double Get_MaximumValue ()
- double Get_Forward ()
- double Get_Reverse ()
- double Get_ForwardRef ()
- double Get_ReverseRef ()
- double Get_ActivationEnergy ()

- double Get_Affinity ()
- double Get_TimeStep ()
- double Eval_ReactionRate (const Matrix< double > &x, const Matrix< double > &gama)
- double Eval_Residual (const Matrix< double > &x_new, const Matrix< double > &x_old, const Matrix< double > &gama_new, const Matrix< double > &gama_old)
- double Eval_Residual (const Matrix< double > &x, const Matrix< double > &gama)
- double Eval_IC_Residual (const Matrix< double > &x)
- double Explicit_Eval (const Matrix< double > &x, const Matrix< double > &gama)

**Protected Attributes**

- double initial_value
- double max_value
- double forward_rate
- double reverse_rate
- double forward_ref_rate
- double reverse_ref_rate
- double activation_energy
- double temperature_affinity
- double time_step
- bool HaveForward
- bool HaveReverse
- bool HaveForRef
- bool HaveRevRef
- int species_index

**Additional Inherited Members**

### 4.51.1 Constructor & Destructor Documentation

#### 4.51.1.1 UnsteadyReaction::UnsteadyReaction ( )

#### 4.51.1.2 UnsteadyReaction::∼UnsteadyReaction ( )

### 4.51.2 Member Function Documentation

#### 4.51.2.1 void UnsteadyReaction::Initialize_List ( MasterSpeciesList & *List* )

#### 4.51.2.2 void UnsteadyReaction::Display_Info ( )

#### 4.51.2.3 void UnsteadyReaction::Set_Species_Index ( int *i* )

#### 4.51.2.4 void UnsteadyReaction::Set_Species_Index ( std::string *formula* )

#### 4.51.2.5 void UnsteadyReaction::Set_Stoichiometric ( int *i,* double *v* )

#### 4.51.2.6 void UnsteadyReaction::Set_Equilibrium ( double *v* )

#### 4.51.2.7 void UnsteadyReaction::Set_Enthalpy ( double *H* )

#### 4.51.2.8 void UnsteadyReaction::Set_Entropy ( double *S* )

#### 4.51.2.9 void UnsteadyReaction::Set_EnthalpyANDEntropy ( double *H,* double *S* )

**4.51.2.10  void UnsteadyReaction::Set␣Energy ( double *G* )**

**4.51.2.11  void UnsteadyReaction::Set␣InitialValue ( double *ic* )**

**4.51.2.12  void UnsteadyReaction::Set␣MaximumValue ( double *max* )**

**4.51.2.13  void UnsteadyReaction::Set␣Forward ( double *forward* )**

**4.51.2.14  void UnsteadyReaction::Set␣Reverse ( double *reverse* )**

**4.51.2.15  void UnsteadyReaction::Set␣ForwardRef ( double *Fref* )**

**4.51.2.16  void UnsteadyReaction::Set␣ReverseRef ( double *Rref* )**

**4.51.2.17  void UnsteadyReaction::Set␣ActivationEnergy ( double *E* )**

**4.51.2.18  void UnsteadyReaction::Set␣Affinity ( double *b* )**

**4.51.2.19  void UnsteadyReaction::Set␣TimeStep ( double *dt* )**

**4.51.2.20  void UnsteadyReaction::checkSpeciesEnergies (  )**

**4.51.2.21  void UnsteadyReaction::calculateEnergies (  )**

**4.51.2.22  void UnsteadyReaction::calculateEquilibrium ( double *T* )**

**4.51.2.23  void UnsteadyReaction::calculateRate ( double *T* )**

**4.51.2.24  bool UnsteadyReaction::haveEquilibrium (  )**

**4.51.2.25  bool UnsteadyReaction::haveRate (  )**

**4.51.2.26  int UnsteadyReaction::Get␣Species␣Index (  )**

**4.51.2.27  double UnsteadyReaction::Get␣Stoichiometric ( int *i* )**

**4.51.2.28  double UnsteadyReaction::Get␣Equilibrium (  )**

**4.51.2.29  double UnsteadyReaction::Get␣Enthalpy (  )**

**4.51.2.30  double UnsteadyReaction::Get␣Entropy (  )**

**4.51.2.31  double UnsteadyReaction::Get␣Energy (  )**

**4.51.2.32  double UnsteadyReaction::Get␣InitialValue (  )**

**4.51.2.33  double UnsteadyReaction::Get␣MaximumValue (  )**

**4.51.2.34  double UnsteadyReaction::Get␣Forward (  )**

**4.51.2.35  double UnsteadyReaction::Get␣Reverse (  )**

**4.51.2.36  double UnsteadyReaction::Get␣ForwardRef (  )**

**4.51.2.37  double UnsteadyReaction::Get␣ReverseRef (  )**

**4.51.2.38  double UnsteadyReaction::Get_ActivationEnergy (   )**

**4.51.2.39  double UnsteadyReaction::Get_Affinity (   )**

**4.51.2.40  double UnsteadyReaction::Get_TimeStep (   )**

**4.51.2.41  double UnsteadyReaction::Eval_ReactionRate ( const Matrix$<$ double $>$ & $x$, const Matrix$<$ double $>$ & $gama$ )**

**4.51.2.42  double UnsteadyReaction::Eval_Residual ( const Matrix$<$ double $>$ & $x\_new$, const Matrix$<$ double $>$ & $x\_old$, const Matrix$<$ double $>$ & $gama\_new$, const Matrix$<$ double $>$ & $gama\_old$ )**

**4.51.2.43  double UnsteadyReaction::Eval_Residual ( const Matrix$<$ double $>$ & $x$, const Matrix$<$ double $>$ & $gama$ )**

**4.51.2.44  double UnsteadyReaction::Eval_IC_Residual ( const Matrix$<$ double $>$ & $x$ )**

**4.51.2.45  double UnsteadyReaction::Explicit_Eval ( const Matrix$<$ double $>$ & $x$, const Matrix$<$ double $>$ & $gama$ )**

### 4.51.3   Member Data Documentation

**4.51.3.1  double UnsteadyReaction::initial_value**  `[protected]`

**4.51.3.2  double UnsteadyReaction::max_value**  `[protected]`

**4.51.3.3  double UnsteadyReaction::forward_rate**  `[protected]`

**4.51.3.4  double UnsteadyReaction::reverse_rate**  `[protected]`

**4.51.3.5  double UnsteadyReaction::forward_ref_rate**  `[protected]`

**4.51.3.6  double UnsteadyReaction::reverse_ref_rate**  `[protected]`

**4.51.3.7  double UnsteadyReaction::activation_energy**  `[protected]`

**4.51.3.8  double UnsteadyReaction::temperature_affinity**  `[protected]`

**4.51.3.9  double UnsteadyReaction::time_step**  `[protected]`

**4.51.3.10   bool UnsteadyReaction::HaveForward**  `[protected]`

**4.51.3.11   bool UnsteadyReaction::HaveReverse**  `[protected]`

**4.51.3.12   bool UnsteadyReaction::HaveForRef**  `[protected]`

**4.51.3.13   bool UnsteadyReaction::HaveRevRef**  `[protected]`

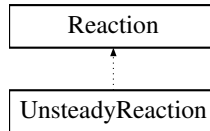**4.51.3.14   int UnsteadyReaction::species_index**  `[protected]`

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

- shark.h

## 4.52   ValueTypePair Class Reference

```
#include <yaml_wrapper.h>
```

**Public Member Functions**

- ValueTypePair ()
- ∼ValueTypePair ()
- ValueTypePair (const std::pair< std::string, int > &vt)
- ValueTypePair (std::string value, int type)
- ValueTypePair (const ValueTypePair &vt)
- ValueTypePair & operator= (const ValueTypePair &vt)
- void editValue (std::string value)
- void editPair (std::string value, int type)
- void findType ()
- void assertType (int type)
- void DisplayPair ()
- std::string getString ()
- bool getBool ()
- double getDouble ()
- int getInt ()
- std::string getValue ()
- int getType ()
- std::pair< std::string, int > & getPair ()

**Private Attributes**

- std::pair< std::string, int > Value_Type
- int type

**4.52.1  Constructor & Destructor Documentation**

**4.52.1.1  ValueTypePair::ValueTypePair (   )**

**4.52.1.2  ValueTypePair::∼ValueTypePair (   )**

**4.52.1.3  ValueTypePair::ValueTypePair ( const std::pair< std::string, int > & vt )**

**4.52.1.4  ValueTypePair::ValueTypePair ( std::string value, int type )**

**4.52.1.5  ValueTypePair::ValueTypePair ( const ValueTypePair & vt )**

**4.52.2  Member Function Documentation**

**4.52.2.1  ValueTypePair& ValueTypePair::operator= ( const ValueTypePair & vt )**

**4.52.2.2  void ValueTypePair::editValue ( std::string value )**

**4.52.2.3  void ValueTypePair::editPair ( std::string value, int type )**

**4.52.2.4  void ValueTypePair::findType (   )**

**4.52.2.5  void ValueTypePair::assertType ( int type )**

**4.52.2.6  void ValueTypePair::DisplayPair (   )**

**4.52.2.7  std::string ValueTypePair::getString (   )**

**4.52.2.8  bool ValueTypePair::getBool ( )**

**4.52.2.9  double ValueTypePair::getDouble ( )**

**4.52.2.10  int ValueTypePair::getInt ( )**

**4.52.2.11  std::string ValueTypePair::getValue ( )**

**4.52.2.12  int ValueTypePair::getType ( )**

**4.52.2.13  std::pair<std::string,int>& ValueTypePair::getPair ( )**

**4.52.3  Member Data Documentation**

**4.52.3.1  std::pair<std::string,int> ValueTypePair::Value_Type** `[private]`

**4.52.3.2  int ValueTypePair::type** `[private]`

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

- yaml_wrapper.h

## 4.53  yaml_cpp_class Class Reference

```
#include <yaml_wrapper.h>
```

**Public Member Functions**

- yaml_cpp_class ()
- ∼yaml_cpp_class ()
- int setInputFile (const char ∗file)
- int readInputFile ()
- int cleanup ()
- int executeYamlRead (const char ∗file)
- YamlWrapper & getYamlWrapper ()
- void DisplayContents ()

**Private Attributes**

- YamlWrapper yaml_wrapper
- FILE ∗ input_file
- const char ∗ file_name
- yaml_parser_t token_parser
- yaml_token_t current_token
- yaml_token_t previous_token

**4.53.1  Constructor & Destructor Documentation**

**4.53.1.1  yaml_cpp_class::yaml_cpp_class ( )**

**4.53.1.2  yaml_cpp_class::∼yaml_cpp_class ( )**

### 4.53.2 Member Function Documentation

**4.53.2.1 int yaml_cpp_class::setInputFile ( const char ∗ *file* )**

**4.53.2.2 int yaml_cpp_class::readInputFile ( )**

**4.53.2.3 int yaml_cpp_class::cleanup ( )**

**4.53.2.4 int yaml_cpp_class::executeYamlRead ( const char ∗ *file* )**

**4.53.2.5 YamlWrapper& yaml_cpp_class::getYamlWrapper ( )**

**4.53.2.6 void yaml_cpp_class::DisplayContents ( )**

### 4.53.3 Member Data Documentation

**4.53.3.1 YamlWrapper yaml_cpp_class::yaml_wrapper** `[private]`

**4.53.3.2 FILE∗ yaml_cpp_class::input_file** `[private]`

**4.53.3.3 const char∗ yaml_cpp_class::file_name** `[private]`

**4.53.3.4 yaml_parser_t yaml_cpp_class::token_parser** `[private]`

**4.53.3.5 yaml_token_t yaml_cpp_class::current_token** `[private]`

**4.53.3.6 yaml_token_t yaml_cpp_class::previous_token** `[private]`

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

- yaml_wrapper.h

## 4.54 YamlWrapper Class Reference

```
#include <yaml_wrapper.h>
```

**Public Member Functions**

- YamlWrapper ()
- ∼YamlWrapper ()
- YamlWrapper (const YamlWrapper &yaml)
- YamlWrapper (std::string key, const Document &doc)
- YamlWrapper & operator= (const YamlWrapper &yaml)
- Document & operator() (const std::string key)
- Document operator() (const std::string key) const
- std::map< std::string, Document > & getDocMap ()
- Document & getDocument (std::string key)
- std::map< std::string,
  Document >::const_iterator end () const
- std::map< std::string,
  Document >::iterator end ()
- std::map< std::string,
  Document >::const_iterator begin () const

- std::map< std::string, Document >::iterator begin ()
- void clear ()
- void resetKeys ()
- void changeKey (std::string oldKey, std::string newKey)
- void revalidateAllKeys ()
- void DisplayContents ()
- void addDocKey (std::string key)
- void copyAnchor2Alias (std::string alias, Document &ref)
- int size ()
- Document & getAnchoredDoc (std::string alias)
- Document & getDocFromHeadAlias (std::string alias)
- Document & getDocFromSubAlias (std::string alias)

**Private Attributes**

- std::map< std::string, Document > Doc_Map

### 4.54.1 Constructor & Destructor Documentation

#### 4.54.1.1 YamlWrapper::YamlWrapper ( )

#### 4.54.1.2 YamlWrapper::∼YamlWrapper ( )

#### 4.54.1.3 YamlWrapper::YamlWrapper ( const **YamlWrapper** & *yaml* )

#### 4.54.1.4 YamlWrapper::YamlWrapper ( std::string *key,* const **Document** & *doc* )

### 4.54.2 Member Function Documentation

#### 4.54.2.1 YamlWrapper& YamlWrapper::operator= ( const **YamlWrapper** & *yaml* )

#### 4.54.2.2 Document& YamlWrapper::operator() ( const std::string *key* )

#### 4.54.2.3 Document YamlWrapper::operator() ( const std::string *key* ) const

#### 4.54.2.4 std::map<std::string, **Document**>& YamlWrapper::getDocMap ( )

#### 4.54.2.5 Document& YamlWrapper::getDocument ( std::string *key* )

#### 4.54.2.6 std::map<std::string, **Document**>::const_iterator YamlWrapper::end ( ) const

#### 4.54.2.7 std::map<std::string, **Document**>::iterator YamlWrapper::end ( )

#### 4.54.2.8 std::map<std::string, **Document**>::const_iterator YamlWrapper::begin ( ) const

#### 4.54.2.9 std::map<std::string, **Document**>::iterator YamlWrapper::begin ( )

#### 4.54.2.10 void YamlWrapper::clear ( )

#### 4.54.2.11 void YamlWrapper::resetKeys ( )

#### 4.54.2.12 void YamlWrapper::changeKey ( std::string *oldKey,* std::string *newKey* )

**4.54.2.13   void YamlWrapper::revalidateAllKeys (   )**

**4.54.2.14   void YamlWrapper::DisplayContents (   )**

**4.54.2.15   void YamlWrapper::addDocKey (  std::string *key* )**

**4.54.2.16   void YamlWrapper::copyAnchor2Alias (  std::string *alias,*  Document & *ref* )**

**4.54.2.17   int YamlWrapper::size (   )**

**4.54.2.18   Document& YamlWrapper::getAnchoredDoc (  std::string *alias* )**

**4.54.2.19   Document& YamlWrapper::getDocFromHeadAlias (  std::string *alias* )**

**4.54.2.20   Document& YamlWrapper::getDocFromSubAlias (  std::string *alias* )**

## 4.54.3   Member Data Documentation

**4.54.3.1   std::map<std::string, Document> YamlWrapper::Doc_Map**   `[private]`

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

- yaml_wrapper.h

# Chapter 5

# File Documentation

## 5.1 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.*

### 5.1.1 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.

### 5.1.2 Function Documentation

#### 5.1.2.1 void print2file_species_header ( FILE * *Output,* DOGFISH_DATA * *dog_dat,* int *i* )

Function to print a species based header for the output file.

#### 5.1.2.2 void print2file_DOGFISH_header ( DOGFISH_DATA * *dog_dat* )

Function to print a time and space header for the output file.

**5.1.2.3 void print2file_DOGFISH_result_old ( DOGFISH_DATA ∗ dog_dat )**

Function to print out the old time results for the output file.

**5.1.2.4 void print2file_DOGFISH_result_new ( DOGFISH_DATA ∗ dog_dat )**

Function to print out the new time results for the output file.

**5.1.2.5 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.

**5.1.2.6 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.

**5.1.2.7 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.

**5.1.2.8 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.

**5.1.2.9 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.

**5.1.2.10 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.

**5.1.2.11  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.

**5.1.2.12  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.

**5.1.2.13  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.

**5.1.2.14  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 |

**5.1.2.15  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.

**5.1.2.16  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.

**5.1.2.17  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.

**5.1.2.18  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_TestOutput.-
txt file in a sub-directory "output" from the directory in which the executable was called.

## 5.2  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.*

**5.2.1  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.
```

**Warning**

The Atom class is mostly complete, but the PeriodicTable object is just a place holder.

**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.

### 5.2.2 Function Documentation

#### 5.2.2.1 int EEL_TESTS ( )

Test function to exercise the class objects and check for errors.

## 5.3 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 $cm^\wedge 3 \ast kPa/K/mol$ (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 ($CE3 = mol/cm^\wedge 3$)*
- #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 ($c = mol/cm^\wedge 3$)*
- #define Nu(mu, rho) ((mu)/(rho))

    *Calculation of kinematic viscosity from dynamic viscosity and density ($cm^\wedge 2/s$)*
- #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$^\wedge$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$^\wedge$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$^\wedge$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.*

### 5.3.1 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.

## 5.3.2 Macro Definition Documentation

### 5.3.2.1 #define Rstd 8.3144621

Gas Constant in J/K/mol (or) L∗kPa/K/mol (Standard Units)

### 5.3.2.2 #define RE3 8.3144621E+3

Gas Constant in cm$^\wedge$3∗kPa/K/mol (Convenient for density calculations)

### 5.3.2.3 #define Po 100.0

Standard state pressure (kPa)

### 5.3.2.4 #define Cstd( *p,  T* ) ((p)/(**Rstd**∗T))

Calculation of concentration/density from partial pressure (Cstd = mol/L)

### 5.3.2.5 #define CE3( *p,  T* ) ((p)/(**RE3**∗T))

Calculation of concentration/density from partial pressure (CE3 = mol/cm$^\wedge$3)

### 5.3.2.6 #define Pstd( *c,  T* ) ((c)∗**Rstd**∗T)

Calculation of partial pressure from concentration/density (c = mol/L)

### 5.3.2.7 #define PE3( *c,  T* ) ((c)∗**RE3**∗T)

Calculation of partial pressure from concentration/density (c = mol/cm$^\wedge$3)

### 5.3.2.8 #define Nu( *mu,  rho* ) ((mu)/(rho))

Calculation of kinematic viscosity from dynamic viscosity and density (cm$^\wedge$2/s)

### 5.3.2.9 #define PSI( *T* ) (0.873143 + (0.000072375∗T))

Calculation of temperature correction factor for dynamic viscosity.

### 5.3.2.10 #define Dp_ij( *Dij,  PT* ) ((PT∗Dij)/**Po**)

Calculation of the corrected binary diffusivity (cm$^\wedge$2/s)

### 5.3.2.11 #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$^\wedge$2/s)

### 5.3.2.12 #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)

**5.3.2.13   #define D_ii(   *rhoi,   mui*  ) (1.385∗mui/rhoi)**

Calculation of self-diffusivity (cm$^\wedge$2/s)

**5.3.2.14   #define ReNum(   *u,   L,   nu*  ) (u∗L/nu)**

Calculation of the Reynold's Number (-)

**5.3.2.15   #define ScNum(   *nu,   D*  ) (nu/D)**

Calculation of the Schmidt Number (-)

**5.3.2.16   #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)

## 5.3.3   Function Documentation

**5.3.3.1   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.

**5.3.3.2   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 |

**5.3.3.3   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.

**5.3.3.4   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.

## 5.4 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 }

  *List of names for error type.*

**Functions**

- void error (int flag)

  *Error function customizes output message based on flag.*

### 5.4.1 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.

## 5.4.2 Macro Definition Documentation

### 5.4.2.1 #define mError( *i* )

**Value:**

```
{error(i);            \
std::cout << "Source: " << __FILE__ << "\nLine: " << __LINE__ << std::endl;}
```

## 5.4.3 Enumeration Type Documentation

### 5.4.3.1 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*

*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*

### 5.4.4    Function Documentation

#### 5.4.4.1    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.

## 5.5    finch.h File Reference

Flux-limiting Implicit Non-oscillatory Conservative High-resolution scheme.

```
#include "macaw.h"
#include "lark.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< dou-
    ble > &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.*

### 5.5.1 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...

$z^\wedge d*R*du/dt = d/dz(z^\wedge d*D*du/dz) - d/dz(z^\wedge d*v*u) - z^\wedge d*k*u + z^\wedge d*S$

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.

### 5.5.2 Enumeration Type Documentation

#### 5.5.2.1 enum **finch_solve_type**

List of enum options to define the solver type in FINCH.

**Enumerator**

> ***FINCH_Picard***
> ***LARK_Picard***
> ***LARK_PJFNK***

#### 5.5.2.2 enum **finch_coord_type**

List of enum options to define the coordinate system in FINCH.

**Enumerator**

> ***Cartesian***
> ***Cylindrical***
> ***Spherical***

### 5.5.3 Function Documentation

#### 5.5.3.1 double max ( std::vector< double > & *values* )

Function returns the maximum in a list of values.

**5.5.3.2  double min ( std::vector$<$ double $>$ & *values* )**

Function returns the minimum in a list of values.

**5.5.3.3  double minmod ( std::vector$<$ double $>$ & *values* )**

Function returns the result of the minmod function acting on a list of values.

**5.5.3.4  int uTotal ( FINCH_DATA $*$ *dat* )**

Function integrates the conserved quantity to return it's total in the domain.

**5.5.3.5  int uAverage ( FINCH_DATA $*$ *dat* )**

Function integrates the conserved quantity to reture it's average in the domain.

**5.5.3.6  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.

**5.5.3.7  int l␣direct ( FINCH_DATA $*$ *dat* )**

Function solves the discretized FINCH problem directly by assuming it is linear.

**5.5.3.8  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)

**5.5.3.9  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.

**5.5.3.10  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.

### 5.5.3.11 void print2file_dim_header ( FILE ∗ *Output,* FINCH_DATA ∗ *dat* )

Function will print out a dimension header for FINCH output.

### 5.5.3.12 void print2file_time_header ( FILE ∗ *Output,* FINCH_DATA ∗ *dat* )

Function will print out a time header for FINCH output.

### 5.5.3.13 void print2file_result_old ( FILE ∗ *Output,* FINCH_DATA ∗ *dat* )

Function will print out the old results to the variable u.

### 5.5.3.14 void print2file_result_new ( FILE ∗ *Output,* FINCH_DATA ∗ *dat* )

Function will print out the new results to the variable u.

### 5.5.3.15 void print2file_newline ( FILE ∗ *Output,* FINCH_DATA ∗ *dat* )

Function will force print out a blank line.

### 5.5.3.16 void print2file_tab ( FILE ∗ *Output,* FINCH_DATA ∗ *dat* )

Function will force print out a tab.

### 5.5.3.17 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.

### 5.5.3.18 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.

### 5.5.3.19 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.

**5.5.3.20 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.

**5.5.3.21 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.

**5.5.3.22 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.

**5.5.3.23 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.

**5.5.3.24 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.

**5.5.3.25 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.

**5.5.3.26 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.

**5.5.3.27 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.

**5.5.3.28  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.

**5.5.3.29  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.

**5.5.3.30  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.

**5.5.3.31  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.

## 5.6  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"
```

### 5.6.1  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

**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.

## 5.7 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.*

### 5.7.1 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.

## 5.7.2 Macro Definition Documentation

### 5.7.2.1 #define Po 100.0

Standard State Pressure - Units: kPa.

### 5.7.2.2 #define R 8.3144621

Gas Constant - Units: J/(K∗mol) = kB ∗ Na.

### 5.7.2.3 #define Na 6.0221413E+23

Avagadro's Number - Units: molecules/mol.

## 5.7.3 Function Documentation

### 5.7.3.1 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.

### 5.7.3.2 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.

### 5.7.3.3 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 relevent in the optimization routine, we need to make the initial guesses to be around the same order of magnitude of the other GSTA parameters.

### 5.7.3.4 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.

### 5.7.3.5 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.

**5.7.3.6   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.

**5.7.3.7   double avgValue ( std::vector$<$ double $>$ & *array* )**

Function returns an average in an array of doubles.

**5.7.3.8   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 |

**5.7.3.9   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 |

**5.7.3.10   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.

**5.7.3.11   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) |

**5.7.3.12    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 |

**5.7.3.13    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 |

**5.7.3.14    double gstaObjFunc (  double ∗ t,  double ∗ y,  double ∗ par,  int m_dat,  void ∗ data )**

Function to evaualte 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.

**5.7.3.15    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.

**5.7.3.16    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.

## 5.8 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 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.*

## Enumerations

- enum krylov_method {
  GMRESLP, PCG, BiCGSTAB, CGS,
  FOM, GMRESRP, GCR, GMRESR }

    *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 $A^{\wedge}T*r=u$.*

- 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 gmresPreconditioner (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 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, BACKTRACK_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.*

### 5.8.1 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.

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.

### 5.8.2 Enumeration Type Documentation

#### 5.8.2.1 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*

### 5.8.3 Function Documentation

#### 5.8.3.1 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 |

#### 5.8.3.2 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.

-----------------------------------------------------------------------------

**5.8.3.3    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 RESidual 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.
>
> ----------------------------------------------------------------------------

**5.8.3.4 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.
>
> ----------------------------------------------------------------------------

**5.8.3.5 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.

    -------------------------------------------------------------------------

**5.8.3.6 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.
>
> --------------------------------------------------------------------------

### 5.8.3.7 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.
>
> --------------------------------------------------------------------------

**5.8.3.8** **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.
> ------------------------------------------------------------------------------

**5.8.3.9** **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 $A^{\wedge}T*r=u$.

This function takes a user supplied linear operator and forms the result of that operator transposed and multiplied by a given vector r ($A^{\wedge}T*r=u$). Transposition is accomplished 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 ($u_i = r^{\wedge}T*A*i$). 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 ($u=A^{\wedge}T*r$) |
| *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.

--------------------------------------------------------------------------------

**5.8.3.10   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.

--------------------------------------------------------------------------------

**5.8.3.11   int gmresPreconditioner ( 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 |

**5.8.3.12   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 RESidual 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 the only linear method that inherently includes preconditioning, 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.

------------------------------------------------------------------------

**5.8.3.13   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$^\wedge$2 + x - 1 Goal is to make this function equal zero

Example Weak Form: x = 1 - x0$^\wedge$2 Rearrage residual to form a weak solution

————————————————————————————————————————————————

**5.8.3.14 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 |

**5.8.3.15 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.

--------------------------------------------------------------------------―

**5.8.3.16 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.

----------------------------------------------------------------------------―

**5.8.3.17    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 |

**5.8.3.18    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.

## 5.9    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.*

### 5.9.1 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.

### 5.9.2 Macro Definition Documentation

#### 5.9.2.1 #define M_PI 3.14159265358979323846264338327950288

Value of PI with double precision.

### 5.9.3 Function Documentation

#### 5.9.3.1 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.

## 5.10 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"
```

### 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$^\wedge$2/mol)*
- #define V 18.92

  *Corresponding van der Waals standard volume for our coordination number (cm$^\wedge$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.*

### 5.10.1 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.

### 5.10.2 Macro Definition Documentation

#### 5.10.2.1 #define DBL_EPSILON 2.2204460492503131e-016

Machine precision value used for approximating gradients.

#### 5.10.2.2 #define Z 10.0

Surface coordination number used in the MSPD activity model.

#### 5.10.2.3 #define A 3.13E+09

Corresponding van der Waals standard area for our coordination number (cm$^\wedge$2/mol)

#### 5.10.2.4 #define V 18.92

Corresponding van der Waals standard volume for our coordination number (cm$^\wedge$3/mol)

#### 5.10.2.5 #define Po 100.0

Standard State Pressure - Units: kPa.

#### 5.10.2.6 #define R 8.3144621

Gas Constant - Units: J/(K$*$mol) = kB $*$ Na.

#### 5.10.2.7 #define Na 6.0221413E+23

Avagadro's Number - Units: molecules/mol.

#### 5.10.2.8 #define kB 1.3806488E-23

Boltzmann's Constant - Units: J/K.

#### 5.10.2.9 #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.

**5.10.2.10 #define lnKo( H, S, T ) -( H / ( R ∗ T ) ) + ( S / R )**

This macro calculates the natural log of the dimensionless isotherm parameter.

**5.10.2.11 #define He( qm, K1, m ) ( qm ∗ K1 ) / ( m ∗ Po )**

This macro calculates the Henry's Coefficient for the ith component.

### 5.10.3 Function Documentation

**5.10.3.1 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 |

**5.10.3.2 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 |

**5.10.3.3 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 |

**5.10.3.4 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 |

### 5.10.3.5  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 |

### 5.10.3.6  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.

### 5.10.3.7  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 |

### 5.10.3.8  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 |

**5.10.3.9   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 SYSTE-M_DATA structure.

**Parameters**

| | |
|---:|---|
| *par* | list of parameters representing variables to be solved for in GPAST |
| *data* | void pointer for the MAGPIE_DATA data structure |

**5.10.3.10   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 |

**5.10.3.11   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 |

**5.10.3.12   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 |

**5.10.3.13    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 |

**5.10.3.14    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 |

**5.10.3.15    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 |

**5.10.3.16    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 |

**5.10.3.17  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)

## 5.11   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)*

**Functions**

- int MOLA_TESTS ()

    *Function to run the MOLA tests.*

### 5.11.1 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**

CO3 2- (aq)

Cl - (aq)

H2O (l)

H + (aq)

H2CO3 (aq)

HCO3 - (aq)

HNO3 (aq)

HCl (aq)

NaHCO3 (aq)

NaCO3 - (aq)

Na + (aq)

NaCl (aq)

NaOH (aq)

NO3 - (aq)

OH - (aq)

UO2 2+ (aq)

UO2NO3 + (aq)

UO2(NO3)2 (aq)

UO2OH + (aq)

UO2(OH)2 (aq)

UO2(OH)3 - (aq)

UO2(OH)4 2- (aq)

(UO2)2OH 3+ (aq)

(UO2)2(OH)2 2+ (aq)

(UO2)3(OH)4 2+ (aq)

(UO2)3(OH)5 + (aq)

(UO2)3(OH)7 - (aq)

(UO2)4(OH)7 + (aq)

UO2CO3 (aq)

UO2(CO3)2 2- (aq)

UO2(CO3)3 4- (aq)

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**

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.

### 5.11.2 Function Documentation

#### 5.11.2.1 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.

## 5.12 monkfish.h File Reference

```
#include "dogfish.h"
```

### Classes

- struct MONKFISH_PARAM
- struct MONKFISH_DATA

**Functions**

- double default_porosity (int i, int l, const void ∗user_data)
- double default_density (int i, int l, const void ∗user_data)
- double default_interparticle_diffusion (int i, int l, const void ∗user_data)
- double default_monk_adsorption (int i, int l, const void ∗user_data)
- double default_monk_equilibrium (int i, int l, const void ∗user_data)
- double default_monkfish_retardation (int i, int l, const void ∗user_data)
- double default_exterior_concentration (int i, const void ∗user_data)
- double default_film_transfer (int i, const void ∗user_data)
- 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_DA-TA ∗monk_dat)
- int MONKFISH_TESTS ()

### 5.12.1 Function Documentation

#### 5.12.1.1 double default_porosity ( int *i,* int *l,* const void ∗ *user_data* )

#### 5.12.1.2 double default_density ( int *i,* int *l,* const void ∗ *user_data* )

#### 5.12.1.3 double default_interparticle_diffusion ( int *i,* int *l,* const void ∗ *user_data* )

#### 5.12.1.4 double default_monk_adsorption ( int *i,* int *l,* const void ∗ *user_data* )

#### 5.12.1.5 double default_monk_equilibrium ( int *i,* int *l,* const void ∗ *user_data* )

#### 5.12.1.6 double default_monkfish_retardation ( int *i,* int *l,* const void ∗ *user_data* )

#### 5.12.1.7 double default_exterior_concentration ( int *i,* const void ∗ *user_data* )

#### 5.12.1.8 double default_film_transfer ( int *i,* const void ∗ *user_data* )

#### 5.12.1.9 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* )

#### 5.12.1.10 int MONKFISH_TESTS ( )

## 5.13 sandbox.h File Reference

```
#include "flock.h"
#include "school.h"
```

**Classes**

- struct Speciation_Test01_Data

---

**Functions**

- int Speciation_Test01_Function (const Matrix< double > &x, Matrix< double > &F, const void ∗res_data)
- int Speciation_Test01_Jacobian (const Matrix< double > &x, Matrix< double > &J, const void ∗precon_data)
- int Speciation_Test01_Guess (const void ∗user_data)
- int Speciation_Test01_MatVec (const Matrix< double > &x, Matrix< double > &Ax, const void ∗matvec_-data)
- int RUN_SANDBOX ()

### 5.13.1 Function Documentation

#### 5.13.1.1 int Speciation_Test01_Function ( const Matrix< double > & *x,* Matrix< double > & *F,* const void ∗ *res_data* )

#### 5.13.1.2 int Speciation_Test01_Jacobian ( const Matrix< double > & *x,* Matrix< double > & *J,* const void ∗ *precon_data* )

#### 5.13.1.3 int Speciation_Test01_Guess ( const void ∗ *user_data* )

#### 5.13.1.4 int Speciation_Test01_MatVec ( const Matrix< double > & *x,* Matrix< double > & *Ax,* const void ∗ *matvec_data* )

#### 5.13.1.5 int RUN_SANDBOX ( )

## 5.14 school.h File Reference

```
#include "eel.h"
#include "mola.h"
#include "shark.h"
#include "dogfish.h"
#include "monkfish.h"
#include "yaml_wrapper.h"
```

## 5.15 scopsowl.h File Reference

```
#include "egret.h"
#include "skua.h"
```

**Classes**

- struct SCOPSOWL_PARAM_DATA
- struct SCOPSOWL_DATA

**Macros**

- #define SCOPSOWL_HPP_
- #define Dp(Dm, ep) (ep∗ep∗Dm)
- #define Dk(rp, T, MW) (9700.0∗rp∗pow((T/MW),0.5))
- #define avgDp(Dp, Dk) (pow(((1/Dp)+(1/Dk)),-1.0))

**Functions**

- void print2file_species_header (FILE ∗Output, SCOPSOWL_DATA ∗owl_dat, int i)
- void print2file_SCOPSOWL_time_header (FILE ∗Output, SCOPSOWL_DATA ∗owl_dat, int i)
- void print2file_SCOPSOWL_header (SCOPSOWL_DATA ∗owl_dat)
- void print2file_SCOPSOWL_result_old (SCOPSOWL_DATA ∗owl_dat)
- void print2file_SCOPSOWL_result_new (SCOPSOWL_DATA ∗owl_dat)
- double default_adsorption (int i, int l, const void ∗user_data)
- double default_retardation (int i, int l, const void ∗user_data)
- double default_pore_diffusion (int i, int l, const void ∗user_data)
- double default_surf_diffusion (int i, int l, const void ∗user_data)
- double default_effective_diffusion (int i, int l, const void ∗user_data)
- double const_pore_diffusion (int i, int l, const void ∗user_data)
- double default_filmMassTransfer (int i, const void ∗user_data)
- double const_filmMassTransfer (int i, const void ∗user_data)
- 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)
- int SCOPSOWL_Executioner (SCOPSOWL_DATA ∗owl_dat)
- int set_SCOPSOWL_ICs (SCOPSOWL_DATA ∗owl_dat)
- int set_SCOPSOWL_timestep (SCOPSOWL_DATA ∗owl_dat)
- int SCOPSOWL_preprocesses (SCOPSOWL_DATA ∗owl_dat)
- int set_SCOPSOWL_params (const void ∗user_data)
- int SCOPSOWL_postprocesses (SCOPSOWL_DATA ∗owl_dat)
- int SCOPSOWL_reset (SCOPSOWL_DATA ∗owl_dat)
- int SCOPSOWL (SCOPSOWL_DATA ∗owl_dat)
- int LARGE_CYCLE_TEST01 (SCOPSOWL_DATA ∗owl_dat)
- int SMALL_CYCLE_TEST02 (SCOPSOWL_DATA ∗owl_dat)
- int CURVE_TEST03 (SCOPSOWL_DATA ∗owl_dat)
- int CURVE_TEST04 (SCOPSOWL_DATA ∗owl_dat)
- int CURVE_TEST05 (SCOPSOWL_DATA ∗owl_dat)
- int SCOPSOWL_SCENARIOS (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate)
- int SCOPSOWL_TESTS ()

### 5.15.1 Macro Definition Documentation

#### 5.15.1.1 #define SCOPSOWL_HPP_

#### 5.15.1.2 #define Dp( *Dm, ep* ) (ep∗ep∗Dm)

#### 5.15.1.3 #define Dk( *rp, T, MW* ) (9700.0∗rp∗pow((T/MW),0.5))

#### 5.15.1.4 #define avgDp( Dp, Dk ) (pow(((1/Dp)+(1/Dk)),-1.0))

### 5.15.2 Function Documentation

#### 5.15.2.1 void print2file_species_header ( FILE ∗ *Output,* SCOPSOWL_DATA ∗ *owl_dat,* int *i* )

#### 5.15.2.2 void print2file_SCOPSOWL_time_header ( FILE ∗ *Output,* SCOPSOWL_DATA ∗ *owl_dat,* int *i* )

#### 5.15.2.3 void print2file_SCOPSOWL_header ( SCOPSOWL_DATA ∗ *owl_dat* )

**5.15.2.4** **void print2file SCOPSOWL result old ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.5** **void print2file SCOPSOWL result new ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.6** **double default adsorption ( int *i,* int *l,* const void ∗ *user data* )**

**5.15.2.7** **double default retardation ( int *i,* int *l,* const void ∗ *user data* )**

**5.15.2.8** **double default pore diffusion ( int *i,* int *l,* const void ∗ *user data* )**

**5.15.2.9** **double default surf diffusion ( int *i,* int *l,* const void ∗ *user data* )**

**5.15.2.10** **double default effective diffusion ( int *i,* int *l,* const void ∗ *user data* )**

**5.15.2.11** **double const pore diffusion ( int *i,* int *l,* const void ∗ *user data* )**

**5.15.2.12** **double default filmMassTransfer ( int *i,* const void ∗ *user data* )**

**5.15.2.13** **double const filmMassTransfer ( int *i,* const void ∗ *user data* )**

**5.15.2.14** **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* )**

**5.15.2.15** **int SCOPSOWL Executioner ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.16** **int set SCOPSOWL ICs ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.17** **int set SCOPSOWL timestep ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.18** **int SCOPSOWL preprocesses ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.19** **int set SCOPSOWL params ( const void ∗ *user data* )**

**5.15.2.20** **int SCOPSOWL postprocesses ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.21** **int SCOPSOWL reset ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.22** **int SCOPSOWL ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.23** **int LARGE CYCLE TEST01 ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.24** **int SMALL CYCLE TEST02 ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.25** **int CURVE TEST03 ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.26** **int CURVE TEST04 ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.27** **int CURVE TEST05 ( SCOPSOWL_DATA ∗ *owl dat* )**

**5.15.2.28** **int SCOPSOWL SCENARIOS ( const char ∗ *scene,* const char ∗ *sorbent,* const char ∗ *comp,* const char ∗ *sorbate* )**

**5.15.2.29** **int SCOPSOWL TESTS ( )**

## 5.16 scopsowl_opt.h File Reference

```
#include "scopsowl.h"
```

### Classes

- struct SCOPSOWL_OPT_DATA

### Functions

- int SCOPSOWL_OPT_set_y (SCOPSOWL_OPT_DATA ∗owl_opt)
- int initial_guess_SCOPSOWL (SCOPSOWL_OPT_DATA ∗owl_opt)
- void eval_SCOPSOWL_Uptake (const double ∗par, int m_dat, const void ∗data, double ∗fvec, int ∗info)
- int SCOPSOWL_OPTIMIZE (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate, const char ∗data)

### 5.16.1 Function Documentation

#### 5.16.1.1 int SCOPSOWL_OPT_set_y ( SCOPSOWL_OPT_DATA ∗ owl_opt )

#### 5.16.1.2 int initial_guess_SCOPSOWL ( SCOPSOWL_OPT_DATA ∗ owl_opt )

#### 5.16.1.3 void eval_SCOPSOWL_Uptake ( const double ∗ par, int m_dat, const void ∗ data, double ∗ fvec, int ∗ info )

#### 5.16.1.4 int SCOPSOWL_OPTIMIZE ( const char ∗ scene, const char ∗ sorbent, const char ∗ comp, const char ∗ sorbate, const char ∗ data )

## 5.17 shark.h File Reference

```
#include "mola.h"
#include "macaw.h"
#include "lark.h"
#include "yaml_wrapper.h"
```

### Classes

- class MasterSpeciesList
- class Reaction
- class MassBalance
- class UnsteadyReaction
- class Mechanism
- class Precipitation
- class UnsteadyPrecipitation
- struct SHARK_DATA

### Macros

- #define Rstd 8.3144621

**Typedefs**

- typedef struct SHARK_DATA SHARK_DATA

**Enumerations**

- enum valid_act {
  IDEAL, DAVIES, DEBYE_HUCKEL, DAVIES_LADSHAW,
  SIT, PITZER }

**Functions**

- void print2file_shark_info (SHARK_DATA ∗shark_dat)
- void print2file_shark_header (SHARK_DATA ∗shark_dat)
- void print2file_shark_results_new (SHARK_DATA ∗shark_dat)
- void print2file_shark_results_old (SHARK_DATA ∗shark_dat)
- int ideal_solution (const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int Davies_equation (const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int DebyeHuckel_equation (const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int DaviesLadshaw_equation (const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int act_choice (const std::string &input)
- bool linesearch_choice (const std::string &input)
- int linearsolve_choice (const std::string &input)
- int Convert2LogConcentration (const Matrix< double > &x, Matrix< double > &logx)
- int Convert2Concentration (const Matrix< double > &logx, Matrix< double > &x)
- int read_scenario (SHARK_DATA ∗shark_dat)
- int read_options (SHARK_DATA ∗shark_dat)
- int read_species (SHARK_DATA ∗shark_dat)
- int read_massbalance (SHARK_DATA ∗shark_dat)
- int read_equilrxn (SHARK_DATA ∗shark_dat)
- int read_unsteadyrxn (SHARK_DATA ∗shark_dat)
- 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)
- 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)
- int shark_parameter_check (SHARK_DATA ∗shark_dat)
- int shark_energy_calculations (SHARK_DATA ∗shark_dat)
- int shark_temperature_calculations (SHARK_DATA ∗shark_dat)
- int shark_pH_finder (SHARK_DATA ∗shark_dat)
- int shark_guess (SHARK_DATA ∗shark_dat)
- int shark_initial_conditions (SHARK_DATA ∗shark_dat)
- int shark_executioner (SHARK_DATA ∗shark_dat)
- int shark_timestep_const (SHARK_DATA ∗shark_dat)
- int shark_timestep_adapt (SHARK_DATA ∗shark_dat)
- int shark_preprocesses (SHARK_DATA ∗shark_dat)
- int shark_solver (SHARK_DATA ∗shark_dat)
- int shark_postprocesses (SHARK_DATA ∗shark_dat)
- int shark_reset (SHARK_DATA ∗shark_dat)
- int shark_residual (const Matrix< double > &x, Matrix< double > &F, const void ∗data)
- int SHARK (SHARK_DATA ∗shark_dat)
- int SHARK_SCENARIO (const char ∗yaml_input)
- int SHARK_TESTS ()

### 5.17.1 Macro Definition Documentation

#### 5.17.1.1 #define Rstd 8.3144621

### 5.17.2 Typedef Documentation

#### 5.17.2.1 typedef struct SHARK_DATA SHARK_DATA

### 5.17.3 Enumeration Type Documentation

#### 5.17.3.1 enum valid_act

**Enumerator**

> *IDEAL*
>
> *DAVIES*
>
> *DEBYE_HUCKEL*
>
> *DAVIES_LADSHAW*
>
> *SIT*
>
> *PITZER*

### 5.17.4 Function Documentation

#### 5.17.4.1 void print2file_shark_info ( SHARK_DATA ∗ *shark_dat* )

#### 5.17.4.2 void print2file_shark_header ( SHARK_DATA ∗ *shark_dat* )

#### 5.17.4.3 void print2file_shark_results_new ( SHARK_DATA ∗ *shark_dat* )

#### 5.17.4.4 void print2file_shark_results_old ( SHARK_DATA ∗ *shark_dat* )

#### 5.17.4.5 int ideal_solution ( const Matrix< double > & *x,* Matrix< double > & *F,* const void ∗ *data* )

#### 5.17.4.6 int Davies_equation ( const Matrix< double > & *x,* Matrix< double > & *F,* const void ∗ *data* )

#### 5.17.4.7 int DebyeHuckel_equation ( const Matrix< double > & *x,* Matrix< double > & *F,* const void ∗ *data* )

#### 5.17.4.8 int DaviesLadshaw_equation ( const Matrix< double > & *x,* Matrix< double > & *F,* const void ∗ *data* )

#### 5.17.4.9 int act_choice ( const std::string & *input* )

#### 5.17.4.10 bool linesearch_choice ( const std::string & *input* )

#### 5.17.4.11 int linearsolve_choice ( const std::string & *input* )

#### 5.17.4.12 int Convert2LogConcentration ( const Matrix< double > & *x,* Matrix< double > & *logx* )

#### 5.17.4.13 int Convert2Concentration ( const Matrix< double > & *logx,* Matrix< double > & *x* )

#### 5.17.4.14 int read_scenario ( SHARK_DATA ∗ *shark_dat* )

#### 5.17.4.15 int read_options ( SHARK_DATA ∗ *shark_dat* )

#### 5.17.4.16 int read_species ( SHARK_DATA ∗ *shark_dat* )

**5.17.4.17 int read_massbalance ( SHARK_DATA ∗ shark_dat )**

**5.17.4.18 int read_equilrxn ( SHARK_DATA ∗ shark_dat )**

**5.17.4.19 int read_unsteadyrxn ( SHARK_DATA ∗ shark_dat )**

**5.17.4.20 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* **)**

**5.17.4.21 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* **)**

**5.17.4.22 int shark_parameter_check ( SHARK_DATA ∗ shark_dat )**

**5.17.4.23 int shark_energy_calculations ( SHARK_DATA ∗ shark_dat )**

**5.17.4.24 int shark_temperature_calculations ( SHARK_DATA ∗ shark_dat )**

**5.17.4.25 int shark_pH_finder ( SHARK_DATA ∗ shark_dat )**

**5.17.4.26 int shark_guess ( SHARK_DATA ∗ shark_dat )**

**5.17.4.27 int shark_initial_conditions ( SHARK_DATA ∗ shark_dat )**

**5.17.4.28 int shark_executioner ( SHARK_DATA ∗ shark_dat )**

**5.17.4.29 int shark_timestep_const ( SHARK_DATA ∗ shark_dat )**

**5.17.4.30 int shark_timestep_adapt ( SHARK_DATA ∗ shark_dat )**

**5.17.4.31 int shark_preprocesses ( SHARK_DATA ∗ shark_dat )**

**5.17.4.32 int shark_solver ( SHARK_DATA ∗ shark_dat )**

**5.17.4.33 int shark_postprocesses ( SHARK_DATA ∗ shark_dat )**

**5.17.4.34 int shark_reset ( SHARK_DATA ∗ shark_dat )**

**5.17.4.35 int shark_residual ( const Matrix< double > &** *x,* **Matrix< double > &** *F,* **const void ∗** *data* **)**

**5.17.4.36 int SHARK ( SHARK_DATA ∗ shark_dat )**

**5.17.4.37 int SHARK_SCENARIO ( const char ∗** *yaml_input* **)**

**5.17.4.38 int SHARK_TESTS ( )**

## 5.18 skua.h File Reference

```
#include "finch.h"
#include "magpie.h"
#include "egret.h"
```

**Classes**

- struct SKUA_PARAM
- struct SKUA_DATA

**Macros**

- #define SKUA_HPP_
- #define D_inf(Dref, Tref, B, p, T) ( Dref ∗ pow(p+sqrt(DBL_EPSILON),(Tref/T)-B) )
- #define D_o(Diff, E, T) ( Diff ∗ exp(-E/(Rstd∗T)) )
- #define D_c(Diff, phi) ( Diff ∗ (1.0/((1.0+1.1E-6)-phi) ) )

**Functions**

- void print2file_species_header (FILE ∗Output, SKUA_DATA ∗skua_dat, int i)
- void print2file_SKUA_time_header (FILE ∗Output, SKUA_DATA ∗skua_dat, int i)
- void print2file_SKUA_header (SKUA_DATA ∗skua_dat)
- void print2file_SKUA_results_old (SKUA_DATA ∗skua_dat)
- void print2file_SKUA_results_new (SKUA_DATA ∗skua_dat)
- double default_Dc (int i, int l, const void ∗data)
- double default_kf (int i, const void ∗data)
- double const_Dc (int i, int l, const void ∗data)
- double simple_darken_Dc (int i, int l, const void ∗data)
- double theoretical_darken_Dc (int i, int l, const void ∗data)
- double empirical_kf (int i, const void ∗data)
- double const_kf (int i, const void ∗data)
- int molefractionCheck (SKUA_DATA ∗skua_dat)
- 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)
- int SKUA_Executioner (SKUA_DATA ∗skua_dat)
- int set_SKUA_ICs (SKUA_DATA ∗skua_dat)
- int set_SKUA_timestep (SKUA_DATA ∗skua_dat)
- int SKUA_preprocesses (SKUA_DATA ∗skua_dat)
- int set_SKUA_params (const void ∗user_data)
- int SKUA_postprocesses (SKUA_DATA ∗skua_dat)
- int SKUA_reset (SKUA_DATA ∗skua_dat)
- int SKUA (SKUA_DATA ∗skua_dat)
- int SKUA_CYCLE_TEST01 (SKUA_DATA ∗skua_dat)
- int SKUA_CYCLE_TEST02 (SKUA_DATA ∗skua_dat)
- int SKUA_LOW_TEST03 (SKUA_DATA ∗skua_dat)
- int SKUA_MID_TEST04 (SKUA_DATA ∗skua_dat)
- int SKUA_SCENARIOS (const char ∗scene, const char ∗sorbent, const char ∗comp, const char ∗sorbate)
- int SKUA_TESTS ()

### 5.18.1 Macro Definition Documentation

#### 5.18.1.1 #define SKUA_HPP_

#### 5.18.1.2 #define D_inf( *Dref,* *Tref,* *B,* *p,* *T* ) ( Dref ∗ pow(p+sqrt(DBL_EPSILON),(Tref/T)-B) )

#### 5.18.1.3 #define D_o( *Diff,* *E,* *T* ) ( Diff ∗ exp(-E/(Rstd∗T)) )

#### 5.18.1.4 #define D_c( *Diff,* *phi* ) ( Diff ∗ (1.0/((1.0+1.1E-6)-phi) ) )

## 5.18.2 Function Documentation

**5.18.2.1 void print2file_species_header ( FILE ∗ *Output,* SKUA_DATA ∗ *skua_dat,* int *i* )**

**5.18.2.2 void print2file_SKUA_time_header ( FILE ∗ *Output,* SKUA_DATA ∗ *skua_dat,* int *i* )**

**5.18.2.3 void print2file_SKUA_header ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.4 void print2file_SKUA_results_old ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.5 void print2file_SKUA_results_new ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.6 double default_Dc ( int *i,* int *l,* const void ∗ *data* )**

**5.18.2.7 double default_kf ( int *i,* const void ∗ *data* )**

**5.18.2.8 double const_Dc ( int *i,* int *l,* const void ∗ *data* )**

**5.18.2.9 double simple_darken_Dc ( int *i,* int *l,* const void ∗ *data* )**

**5.18.2.10 double theoretical_darken_Dc ( int *i,* int *l,* const void ∗ *data* )**

**5.18.2.11 double empirical_kf ( int *i,* const void ∗ *data* )**

**5.18.2.12 double const_kf ( int *i,* const void ∗ *data* )**

**5.18.2.13 int molefractionCheck ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.14 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* )**

**5.18.2.15 int SKUA_Executioner ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.16 int set_SKUA_ICs ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.17 int set_SKUA_timestep ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.18 int SKUA_preprocesses ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.19 int set_SKUA_params ( const void ∗ *user_data* )**

**5.18.2.20 int SKUA_postprocesses ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.21 int SKUA_reset ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.22 int SKUA ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.23 int SKUA_CYCLE_TEST01 ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.24 int SKUA_CYCLE_TEST02 ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.25 int SKUA_LOW_TEST03 ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.26 int SKUA_MID_TEST04 ( SKUA_DATA ∗ *skua_dat* )**

**5.18.2.27 int SKUA_SCENARIOS ( const char ∗ *scene,* const char ∗ *sorbent,* const char ∗ *comp,* const char ∗ *sorbate* )**

**5.18.2.28 int SKUA_TESTS ( )**

## 5.19 skua_opt.h File Reference

```
#include "skua.h"
```

**Classes**

- struct SKUA_OPT_DATA

**Functions**

- int SKUA_OPT_set_y (SKUA_OPT_DATA *skua_opt)
- int initial_guess_SKUA (SKUA_OPT_DATA *skua_opt)
- void eval_SKUA_Uptake (const double *par, int m_dat, const void *data, double *fvec, int *info)
- int SKUA_OPTIMIZE (const char *scene, const char *sorbent, const char *comp, const char *sorbate, const char *data)

**5.19.1 Function Documentation**

**5.19.1.1 int SKUA_OPT_set_y ( SKUA_OPT_DATA * skua_opt )**

**5.19.1.2 int initial_guess_SKUA ( SKUA_OPT_DATA * skua_opt )**

**5.19.1.3 void eval_SKUA_Uptake ( const double * par, int m_dat, const void * data, double * fvec, int * info )**

**5.19.1.4 int SKUA_OPTIMIZE ( const char * scene, const char * sorbent, const char * comp, const char * sorbate, const char * data )**

## 5.20 Trajectory.h File Reference

```
#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 RADIAL_FORCE (const Matrix< double > &POL, double eta, double b, double mp, double t, double a)
- double TANGENTIAL_FORCE (const Matrix< double > &POL, const Matrix< double > &dY, double eta, double b, double mp, double t, double a, int i)
- int CARTESIAN (const Matrix< double > &POL, Matrix< double > &H, const Matrix< double > &dY, double 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 ()

### 5.20.1 Function Documentation

#### 5.20.1.1 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* )

#### 5.20.1.2 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* )

#### 5.20.1.3 double Grav_R ( const Matrix< double > & *dX,* int *i,* double *b,* double *rho_p,* double *rho_f* )

#### 5.20.1.4 double Grav_T ( const Matrix< double > & *dX,* int *i,* double *b,* double *rho_p,* double *rho_f* )

#### 5.20.1.5 double Van_R ( const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *Hamaker,* double *b,* double *a* )

#### 5.20.1.6 double V_RAD ( const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *V0,* double *rho_f,* double *a,* double *eta* )

#### 5.20.1.7 double V_THETA ( const Matrix< double > & *dX,* const Matrix< double > & *dY,* int *i,* double *V0,* double *rho_f,* double *a,* double *eta* )

#### 5.20.1.8 double Brown_RAD ( double *n_rand,* double *m_rand,* double *sigma_n,* double *sigma_m* )

#### 5.20.1.9 double Brown_THETA ( double *s_rand,* double *t_rand,* double *sigma_n,* double *sigma_m* )

#### 5.20.1.10 int POLAR ( Matrix< double > & *POL,* const Matrix< double > & *dX,* const Matrix< double > & *dY,* const void ∗ *data,* int *i* )

#### 5.20.1.11 double RADIAL_FORCE ( const Matrix< double > & *POL,* double *eta,* double *b,* double *mp,* double *t,* double *a* )

#### 5.20.1.12 double TANGENTIAL_FORCE ( const Matrix< double > & *POL,* const Matrix< double > & *dY,* double *eta,* double *b,* double *mp,* double *t,* double *a,* int *i* )

**5.20.1.13** int CARTESIAN ( const **Matrix**< double > & *POL,* **Matrix**< double > & *H,* const **Matrix**< double > & *dY,* double *i,* const void ∗ *data* )

**5.20.1.14** int DISPLACEMENT ( **Matrix**< double > & *dX,* **Matrix**< double > & *dY,* const **Matrix**< double > & *H,* int *i* )

**5.20.1.15** int LOCATION ( const **Matrix**< double > & *dY,* const **Matrix**< double > & *dX,* **Matrix**< double > & *X,* **Matrix**< double > & *Y,* int *i* )

**5.20.1.16** double Removal␣Efficiency ( double *Sum␣Cap,* const void ∗ *data* )

**5.20.1.17** int Trajectory␣SetupConstants ( **TRAJECTORY_DATA** ∗ *dat* )

**5.20.1.18** int Number␣Generator ( **TRAJECTORY_DATA** ∗ *dat* )

**5.20.1.19** int Run␣Trajectory ( )

# 5.21 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 "0.0 alpha"

  *Macro expansion for executable current version number.*
- #define ECO_EXECUTABLE "eco0"

  *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 }

  *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.*

- std::string allLower (const std::string &input)

    *Function to return an all lower case string based on the passed argument.*

- 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.*

### 5.21.1 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.

### 5.21.2 Macro Definition Documentation

#### 5.21.2.1 #define UI_HPP_

#### 5.21.2.2 #define ECO_VERSION "0.0 alpha"

Macro expansion for executable current version number.

#### 5.21.2.3 #define ECO_EXECUTABLE "eco0"

Macro expansion for executable current name.

### 5.21.3 Enumeration Type Documentation

#### 5.21.3.1 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*

### 5.21.4 Function Documentation

#### 5.21.4.1 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.

#### 5.21.4.2 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.

#### 5.21.4.3 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 |

#### 5.21.4.4 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 |

**5.21.4.5 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 "&ndash;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 |

**5.21.4.6 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 "&ndash;version" and will tell the executable to display version information about the executable.

**Parameters**

| | |
|---|---|
| *input* | input string user gives to the console |

**5.21.4.7 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 "&ndash;test" and determine whether or not the user requests to run an ecosystem test function.

**Parameters**

| | |
|---|---|
| *input* | input string user gives to the console |

**5.21.4.8 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 "&ndash;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 |

**5.21.4.9 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 "&ndash;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 |

**5.21.4.10 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 "&ndash;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 |

**5.21.4.11 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 |

**5.21.4.12 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 |

**5.21.4.13 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 |

**5.21.4.14 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 |

**5.21.4.15  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 |

**5.21.4.16  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 |

**5.21.4.17  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 |

**5.21.4.18  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 |

**5.21.4.19 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 |

**5.21.4.20 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 |

**5.21.4.21 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 |

**5.21.4.22 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 |

**5.21.4.23 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 |

**5.21.4.24    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 |


**5.21.4.25    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 |


## 5.22    yaml_wrapper.h File Reference

```
#include "yaml.h"
#include "error.h"
#include <map>
#include <string>
#include <iostream>
#include <utility>
#include <stdexcept>
```

**Classes**

- class ValueTypePair
- class KeyValueMap
- class SubHeader
- class Header
- class Document
- class YamlWrapper
- class yaml_cpp_class

**Typedefs**

- typedef enum data_type data_type
- typedef enum header_state header_state

**Enumerations**

- enum data_type {
  STRING, BOOLEAN, DOUBLE, INT,
  UNKNOWN }
- enum header_state { ANCHOR, ALIAS, NONE }

**Functions**

- int YAML_WRAPPER_TESTS ()
- int YAML_CPP_TEST (const char ∗file)

### 5.22.1 Typedef Documentation

#### 5.22.1.1 typedef enum data_type data_type

#### 5.22.1.2 typedef enum header_state header_state

### 5.22.2 Enumeration Type Documentation

#### 5.22.2.1 enum data_type

**Enumerator**

> *STRING*
> *BOOLEAN*
> *DOUBLE*
> *INT*
> *UNKNOWN*

#### 5.22.2.2 enum header_state

**Enumerator**

> *ANCHOR*
> *ALIAS*
> *NONE*

### 5.22.3 Function Documentation

#### 5.22.3.1 int YAML_WRAPPER_TESTS ( )

#### 5.22.3.2 int YAML_CPP_TEST ( const char ∗ *file* )

# Index