Reference Manual

Generated by Doxygen 1.3.9.1

Tue Sep 25 17:37:40 2007

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 documented files with brief descriptions:

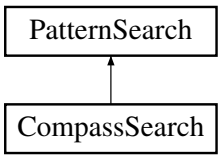
Chapter 4

Class Documentation

4.1 CompassSearch Class Reference

`#include <CompassSearch.h>`

Inheritance diagram for CompassSearch::



Public Member Functions

Constructors & Destructor

- **CompassSearch** (long numberOfVariables, Vector< double > &startPoint)
- **CompassSearch** (long dim, Vector< double > &startPoint, double startStep, double stopStep, void(*objective)(long vars, Vector< double > &x, double &func, bool &flag, void *an_obj), void *input_obj)
- virtual ~**CompassSearch** ()

Other public methods

- **CompassSearch** & **operator**= (const **CompassSearch** &A)
- void **BeginSearch** ()

Protected Member Functions

- void **ExploratoryMoves** ()
- void **CreatePattern** ()
- void **UpdatePattern** ()

4.1.1 Detailed Description

The CompassSearch class is derived from the **PatternSearch**(p. 15) class. A compass search checks the positive and negative quardinate vectors for each dimension until improvement in the function values is found. The search then relocates to the improving point and begins again.

Author:
Liz Dolan, The College of William and Mary, 1999, modified by P.L. Shepherd, 7/00

4.1.2 Constructor & Destructor Documentation

4.1.2.1 CompassSearch::CompassSearch (long *numberOfVariables*, Vector< double > & *startPoint*)

Constructor. This constructor has two parameters. All it does is call the **PatternSearch**(p. 15) constructor with the same signature, then change its ID number to 2200.

Parameters:
dim the dimension of the problem

startpoint the start point for the search

See also:
PatternSearch::PatternSearch(long dim, Vector<double> &startPoint)(p. 16)

4.1.2.2 CompassSearch::CompassSearch (long *dim*, Vector< double > & *startPoint*, double *startStep*, double *stopStep*, void(∗)(long vars, Vector< double > &x, double &func, bool &flag, void ∗an_obj) *objective*, void ∗ *input_obj*)

Special constructor using void function and object pointers. This constructor merely calls the **PatternSearch**(p. 15) constructor with the same signature, then change its ID number to 2200.

This constructor takes six input parameters.

Parameters:
dim the dimension of the problem

startPoint a Vector of doubles, the starting point for the search

startStep the beginning delta, or lattice step length

stopStep the stopping step length for the search

objective a pointer to the function to be minimized

input_obj used to send additional data as needed–will normally be set to NULL.

See also:
PatternSearch(p. 15)(long dim, Vector<double> &startPoint, double startStep, double stopStep, void (∗objective)(long vars, Vector<double> &x, double & func, bool& flag, void∗ an_obj), void ∗ input_obj)

4.1.2.3 virtual CompassSearch::~CompassSearch () [virtual]

Destructor

4.1.3 Member Function Documentation

4.1.3.1 void CompassSearch::BeginSearch () [virtual]

BeginSearch()(p. 9) will call the methods that implement the actual compass search algorithm.

Returns:
void

Implements **PatternSearch** (p. 17).

4.1.3.2 void CompassSearch::CreatePattern () [protected]

Initializes the pattern to one that contains one positive and one negative unit vector in each of the compass directions.

Returns:
void

4.1.3.3 void CompassSearch::ExploratoryMoves () [protected, virtual]

Exploratory Moves does the real work of the compass search.

Returns:
void

Implements **PatternSearch** (p. 18).

4.1.3.4 CompassSearch& CompassSearch::operator= (const CompassSearch & *A*)

Overloaded assignment operator

Parameters:
A the search to be assigned to *this.

Returns:
CompassSearch&

4.1.3.5 void CompassSearch::UpdatePattern () [protected]

Calls the **PatternSearch**(p. 15) method { **ScalePattern()**(p. 20)} to scale the trial steps by SCALE_FACTOR. SCALE_FACTOR is typically 0.5, so the steplength halves itself.

Returns:
void

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

- CompassSearch.h

4.2 KrigApprox Class Reference

This class provides different functions for the approximation using the Kriging model.

`#include <KrigApprox.h>`

Public Member Functions

- **KrigApprox** ()

  constructor

- ~**KrigApprox** ()

  destructor

- double **MSE** (Database Data)

  This function calculates the mean square error of the Kriging approximation considering the given database <data>.

- double **MSE** (Array< double > Input, Array< double > Target)

  This function calculates the mean square error of the approximation considering the given arrays <input> and <target>.

- double **MSE** (Population offsprings)

  This function calculates the mean square error of the approximation considering a given population <offsprings>.

- int **Evaluate** (Population &offsprings)

  This function evaluates the <offsprings> with the Kriging model and stores the results as the fitness value in each offspring.

- int **Evaluate** (Individual &offspring)

  This function evaluates the <offspring> with the Kriging model and stores the results as the fitness value in the offspring.

- int **Evaluate** (Array< double > InputData, Array< double > &OutputData)

  This function evaluates the data in <inputdata> with the Kriging approximation and stores the results in <outputdata>.

- int **Train_MLM** (Database Data)

  This function approximates the datas which are stored in the database <data> using the Kriging model.

- int **Train_MLM** (Array< double > InputData, Array< double > TargetData)

  This function approximates the datas which are stored in the arrays <inputdata> and <targetdata> using the Kriging model.

- int **Train** (Database Data)

  This function approximates the datas which are stored in the database <data> using the Kriging model (default algorithm).

- int **Train** (Array< double > InputData, Array< double > TargetData)

This function approximates the datas which are stored in the arrays <inputdata> and <targetdata> using the Kriging model (default algorithm).

- int **ScanSquare3D** (int lowerBorder, int upperBorder, Array< double > &OutputData)

  This function scans an area of the data which must have an input dimension of 2 and an output dimension of 1. The scanning area is between the borders given as the parameters <lowerborder> and <upperborder>. The resolution is fixed to 100. The results can be further used for matlab.

- int **ScanSquare3D** (int lowerBorder, int upperBorder, Array< double > &OutputData, int ind1, int ind2, Array< double > ParametersToKeepConstant)

  This function scans an area of the data in the parameters <ind1> and <ind2> . The scanning area is between the borders given as the parameters <lowerborder> and <upperborder>. The resolution is fixed to 100. The results can be further used for matlab.

4.2.1 Detailed Description

This class provides different functions for the approximation using the Kriging model.

This class is based on the Kriging approximation model. For using this class in an appropriate way it is required to create an instance of the class <database>. After the known data is stored in the database the functionality of the Kriging approximation can be applied. Therefore in the next step the model should be trained and after this the evaluation functions can be used. For further information on the basics of the Kriging model please check `http://www.cs.wm.edu/~va/software/krigifier/documentation/`

4.2.2 Member Function Documentation

4.2.2.1 int KrigApprox::Evaluate (Array< double > *InputData*, Array< double > & *OutputData*)

This function evaluates the data in <inputdata> with the Kriging approximation and stores the results in <outputdata>.

Parameters:

InputData Array containing the input parameters

OutputData Array containing the evaluation results

Return values:

0 ok

1 Model was not trained so far

4.2.2.2 int KrigApprox::Evaluate (Individual & *offspring*)

This function evaluates the <offspring> with the Kriging model and stores the results as the fitness value in the offspring.

Parameters:

offspring Individual to be evaluated

Return values:

0 ok

1 Model was not trained so far

4.2.2.3 int KrigApprox::Evaluate (Population & *offsprings*)

This function evaluates the <offsprings> with the Kriging model and stores the results as the fitness value in each offspring.

Parameters:

offsprings population of offsprings to be evaluated

Return values:

0 ok

1 Model was not trained so far

4.2.2.4 double KrigApprox::MSE (Population *offsprings*)

This function calculates the mean square error of the approximation considering a given population <offsprings>.

Parameters:

offsprings population containing the parameters in the first Chromosome and the fitness value as target value

Return values:

MSE mean square error of the approximation

4.2.2.5 double KrigApprox::MSE (Array< double > *Input*, Array< double > *Target*)

This function calculates the mean square error of the approximation considering the given arrays <input> and <target>.

Parameters:

Input array containing the input data

Target array containing the target data

Return values:

MSE mean square error of the approximation

4.2.2.6 double KrigApprox::MSE (Database *Data*)

This function calculates the mean square error of the Kriging approximation considering the given database <data>.

Parameters:

Data the database containing the input and target data

Return values:

error_MSE mean square error of the Kriging approximation

4.2.2.7 int KrigApprox::ScanSquare3D (int *lowerBorder*, int *upperBorder*, Array< double > & *OutputData*, int *ind1*, int *ind2*, Array< double > *ParametersToKeepConstant*)

This function scans an area of the data in the parameters <ind1> and <ind2> . The scanning area is between the borders given as the parameters <lowerborder> and <upperborder>. The resolution is fixed to 100. The results can be further used for matlab.

Parameters:

lowerBorder lower border of the area which is to be scanned

upperBorder upper border of the area which is to be scanned

OutputData Array containing the datas of the scanned areas in format [x-coordinate, y-coordinate, approximation value]

ind1 parameter 1

ind2 parameter 2

ParametersToKeepConstant one dimensional array containing the remaining values of the parameters which should be kept constant in increasing order

Return values:

0 ok

4.2.2.8 int KrigApprox::ScanSquare3D (int *lowerBorder*, int *upperBorder*, Array< double > & *OutputData*)

This function scans an area of the data which must have an input dimension of 2 and an output dimension of 1. The scanning area is between the borders given as the parameters <lowerborder> and <upperborder>. The resolution is fixed to 100. The results can be further used for matlab.

Parameters:

lowerBorder lower border of the area which is to be scanned

upperBorder upper border of the area which is to be scanned

OutputData Array containing the datas of the scanned areas in format [x-coordinate, y-coordinate, approximation value]

Return values:

0 ok

4.2.2.9 int KrigApprox::Train (Array< double > *InputData*, Array< double > *TargetData*)

This function approximates the datas which are stored in the arrays <inputdata> and <targetdata> using the Kriging model (default algorithm).

Parameters:

InputData array containing the inputdata which are used for approximation

TargetData array containing the targetdata which are used for approximation

Return values:

0 ok

4.2.2.10 int KrigApprox::Train (Database *Data*)

This function approximates the datas which are stored in the database <data> using the Kriging model (default algorithm).

Parameters:

Data Database containing the data which are used for approximation

Return values:

0 ok

4.2.2.11 int KrigApprox::Train_MLM (Array< double > *InputData*, Array< double > *TargetData*)

This function approximates the datas which are stored in the arrays <inputdata> and <targetdata> using the Kriging model.

Parameters:

InputData array containing the inputdata which are used for approximation

TargetData array containing the targetdata which are used for approximation

Return values:

0 ok

4.2.2.12 int KrigApprox::Train_MLM (Database *Data*)

This function approximates the datas which are stored in the database <data> using the Kriging model.

Parameters:

Data Database containing the data which are used for approximation

Return values:

0 ok

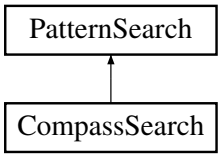
The documentation for this class was generated from the following file:

- **KrigApprox.h**

4.3 PatternSearch Class Reference

`#include <PatternSearch.h>`

Inheritance diagram for PatternSearch::



Public Member Functions

Constructors & Destructor

- **PatternSearch** (long dim, Vector< double > &startPoint)
- **PatternSearch** (const **PatternSearch** &Original)
- **PatternSearch** (long dim, Vector< double > &startPoint, double startStep, double stopStep, void(*objective)(long vars, Vector< double > &x, double &func, bool &flag, void *an_obj), void *input_obj)
- virtual ~**PatternSearch** ()

Other Initialization Methods

- virtual **PatternSearch** & **operator=** (const **PatternSearch** &A)
- virtual void **CleanSlate** (long dim, Vector< double > &startpoint)
- virtual void **CleanSlate** (long dim, Vector< double > &startpoint, double startStep, double stopStep, void(*objective)(long vars, Vector< double > &x, double &func, bool &flag, void *an_obj), void *input_obj)
- void **InitializeDesign** (long patternSize, const Matrix< double > *designPtr)
- void **ReadInFile** (istream &fp)
- void **PrintDesign** () const

Search method

- virtual void **BeginSearch** ()=0

Accessors

- void **GetPatternLength** (long &pattern) const
- double **GetDelta** () const
- void **GetInitialStepLength** (double &stepLen)
- void **SetInitialStepLength** (double &stepLen)

Protected Member Functions

- virtual bool **Stop** ()
- virtual void **ExploratoryMoves** ()=0
- virtual void **CopySearch** (const **PatternSearch** &Original)
- virtual void **NextPoint** (long index, const Vector< double > &currentPoint, Vector< double > &nextPoint)
- virtual void **ReplaceMinimum** (Vector< double > &newPoint, double newValue)
- virtual void **ScalePattern** (double scalar)

Protected Attributes

- long **patternLength**
- double **delta**
- double **initialStepLength**

4.3.1 Detailed Description

The PatternSearch class is derived from the DirectSearch class. PatternSearch is an abstract base class.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 PatternSearch::PatternSearch (long *dim*, Vector< double > & *startPoint*)

Constructor for class PatternSearch. This constructor has two parameters. Other data members will be set to default values, as follows (in addition to data members set by the DirectSearch constructor): {itemize} [] patternLength = 0 [] initialStepLength = .25 [] delta = initialStepLength [] IDnumber = 2000 {itemize}

Parameters:

dim the dimension of the problem

startpoint the start point for the search

See also:

DirectSearch::DirectSearch(long dim, Vector<double> &startPoint)

4.3.2.2 PatternSearch::PatternSearch (const PatternSearch & *Original*)

Deep copy constructor for class PatternSearch

Parameters:

Original a reference to the object to be copied. Note that this will in ordinary practice actually be a member of a concrete class derived from PatternSearch.

4.3.2.3 PatternSearch::PatternSearch (long *dim*, Vector< double > & *startPoint*, double *startStep*, double *stopStep*, void(∗)(long vars, Vector< double > &x, double &func, bool &flag, void ∗an_obj) *objective*, void ∗ *input_obj*)

Special constructor using void function and object pointers. The objective function can be sent in as the fifth parameter here; the last parameter is used for any other information that may be necessary; it is used in MAPS, for example, to send in an object from an outside class. Usually, though, the final parameter will simply be set to NULL. This constructor takes six input parameters.

Parameters:

dim the dimension of the problem

startPoint a Vector of doubles, the starting point for the search

startStep the beginning delta, or lattice step length

stopStep the stopping step length for the search

objective a pointer to the function to be minimized

input_obj used to send additional data as needed–will normally be set to NULL.

4.3.2.4 virtual PatternSearch::~PatternSearch () [virtual]

Destructor

4.3.3 Member Function Documentation

4.3.3.1 virtual void PatternSearch::BeginSearch () [pure virtual]

{ **BeginSearch()**(p. 17)} is an unimplemented virtual void method in the abstract base classes. It is to be implemented in the concrete classes. There, **BeginSearch()**(p. 17) will call the methods that implement the actual search algorithms.

Implemented in **CompassSearch** (p. 9).

4.3.3.2 virtual void PatternSearch::CleanSlate (long *dim*, Vector< double > & *startpoint*, double *startStep*, double *stopStep*, void(*)(long vars, Vector< double > &x, double &func, bool &flag, void *an_obj) *objective*, void * *input_obj*) [virtual]

overloaded version of { CleanSlate} using void function and object pointers. This version of CleanSlate takes six parameters.

Parameters:

dim the dimension of the problem

startPoint a Vector of doubles, the starting point for the search

startStep the beginning delta, or lattice step length

stopStep the stopping step length for the search

objective a pointer to the function to be minimized

input_obj used to send additional data as needed–will normally be set to NULL.

Returns:

void

See also:

CleanSlate(long dim, Vector<double> &startpoint)(p. 17)

4.3.3.3 virtual void PatternSearch::CleanSlate (long *dim*, Vector< double > & *startpoint*) [virtual]

Reinitializes values in order to reuse the same search object. This version of { CleanSlate} takes two parameters. Other data members are set by default as in the constructor **PatternSearch(long, Vector<double>&)**(p. 16).

Parameters:

dim the dimension of the problem

startPoint the new starting point for the search

Returns:

void

See also:

CleanSlate(p. 17)(long dim, Vector<double> &startpoint, double* startStep, double stopStep, void (*objective)(long vars, Vector<double> &x, double & func, bool& flag, void* an_obj), void * input_obj)

4.3.3.4 virtual void PatternSearch::CopySearch (const PatternSearch & *Original*) [protected, virtual]

Used to implement the overloaded assignment operator.

Parameters:

Original a reference to a PatternSearch object

Returns:

void

4.3.3.5 virtual void PatternSearch::ExploratoryMoves () [protected, pure virtual]

PatternSearch::Exploratory Moves is an unimplemented virtual void function. It will be implemented in the concrete classes as the workhorse for the algorithmic implementations

Implemented in **CompassSearch** (p. 9).

4.3.3.6 double PatternSearch::GetDelta () const

Returns delta, the lattice step length

Returns:

double

4.3.3.7 void PatternSearch::GetInitialStepLength (double & *stepLen*)

Returns initialStepLength, the initial value of delta.

Parameters:

stepLen a reference to a double: will be assigned the value of initialStepLength

Returns:

void

4.3.3.8 void PatternSearch::GetPatternLength (long & *pattern*) const

Returns the number of "columns" of the pattern matrix

Parameters:
pattern a long provided by the user; will be assigned the value of patternLength

Returns:
void

4.3.3.9 void PatternSearch::InitializeDesign (long *patternSize*, const Matrix< double > * *designPtr*)

Deletes any existing pattern and replaces it with the one pointed to by designPtr. Calls DesignSearch::InitializeDesign() and then initializes patternLength.

Parameters:
patternSize the number of "columns", i.e. trial points, in design matrix

pat a pointer to a design matrix

Returns:
void

4.3.3.10 virtual void PatternSearch::NextPoint (long *index*, const Vector< double > & *currentPoint*, Vector< double > & *nextPoint*) [protected, virtual]

NextPoint()(p. 19) calculates the next trial point by adding the design matrix column at index to the current vector. Returns the prospect in nextPoint.

Parameters:
index the index of a column of the design matrix

currentPoint a reference to the current vector

nextPoint a reference to a vector, which will be assigned the value of the appropriate point to be evaluated next.

Returns:
void

4.3.3.11 virtual PatternSearch& PatternSearch::operator= (const PatternSearch & *A*) [virtual]

Overloaded assignment operator. note that because PatternSearch is an abstract base class, the object actually returned will be of a type corresponding to one of the concrete classes.

Parameters:
A reference to a const PatternSearch object

Returns:
PatternSearch&

4.3.3.12 void PatternSearch::PrintDesign () const

Prints out useful information about the search

4.3.3.13 void PatternSearch::ReadInFile (istream & *fp*)

Reads the design in from a file. You may also pass cin as the input stream as desired. Input first the pattern length and then the values of each trial vector (i.e. input the pattern by column)

Parameters:
fp the filestream –or cin may be used

Returns:
void

4.3.3.14 virtual void PatternSearch::ReplaceMinimum (Vector< double > & *newPoint*, double *newValue*) [protected, virtual]

Replaces the minimizer & the minimum objective function value.

Parameters:
newPoint the point that will be assigned as the new minPoint
newValue the value that will be assigned to minValue

Returns:
void

4.3.3.15 virtual void PatternSearch::ScalePattern (double *scalar*) [protected, virtual]

Scale lattice step length by scalar

Parameters:
scalar the value by which to scale the pattern

Returns:
void

4.3.3.16 void PatternSearch::SetInitialStepLength (double & *stepLen*)

Assigns the value of steplen to initialStepLength.

Parameters:
stepLen a reference to a double:initialStepLength will be assigned this value.

Returns:
void

4.3.3.17 virtual bool PatternSearch::Stop () [protected, virtual]

Whether nor not to stop, based either on maxCalls or lattice resolution

Returns:
bool

4.3.4 Member Data Documentation

4.3.4.1 double PatternSearch::delta [protected]

The density of the underlying lattice

4.3.4.2 double PatternSearch::initialStepLength [protected]

The initial setting for delta

4.3.4.3 long PatternSearch::patternLength [protected]

The number of "columns" in design matrix, i.e. trial points

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

- PatternSearch.h

Chapter 5

File Documentation

5.1 KrigApprox.h File Reference

```
#include "EALib/Population.h"
#include <string>
#include "Database.h"
#include "krig.h"
```

Classes

- class **KrigApprox**

  This class provides different functions for the approximation using the Kriging model.

5.1.1 Detailed Description

Index